# Setuptools Documentation

***Release 5.4.3***

## The fellowship of the packaging

August 02, 2014

Contents

Setuptools is a fully-featured, actively-maintained, and stable library designed to facilitate packaging Python projects, where packaging includes:

- Python package and module definitions
- Distribution package metadata
- Test hooks
- Project installation
- Platform-specific details
- Python 3 support

Documentation content:

CHAPTER 1

# Roadmap

Setuptools is primarily in maintenance mode. The project attempts to address user issues, concerns, and feature requests in a timely fashion.

CHAPTER 2

# Supporting both Python 2 and Python 3 with Setuptools

Starting with Distribute version 0.6.2 and Setuptools 0.7, the Setuptools project supported Python 3. Installing and using setuptools for Python 3 code works exactly the same as for Python 2 code, but Setuptools also helps you to support Python 2 and Python 3 from the same source code by letting you run 2to3 on the code as a part of the build process, by setting the keyword parameter `use_2to3` to True.

## 2.1 Setuptools as help during porting

Setuptools can make the porting process much easier by automatically running 2to3 as a part of the test running. To do this you need to configure the setup.py so that you can run the unit tests with `python setup.py test`.

See *test - Build package and run a unittest suite* for more information on this.

Once you have the tests running under Python 2, you can add the use_2to3 keyword parameters to setup(), and start running the tests under Python 3. The test command will now first run the build command during which the code will be converted with 2to3, and the tests will then be run from the build directory, as opposed from the source directory as is normally done.

Setuptools will convert all Python files, and also all doctests in Python files. However, if you have doctests located in separate text files, these will not automatically be converted. By adding them to the `convert_2to3_doctests` keyword parameter Setuptools will convert them as well.

By default, the conversion uses all fixers in the `lib2to3.fixers` package. To use additional fixers, the parameter `use_2to3_fixers` can be set to a list of names of packages containing fixers. To exclude fixers, the parameter `use_2to3_exclude_fixers` can be set to fixer names to be skipped.

A typical setup.py can look something like this:

```
from setuptools import setup

setup(
    name='your.module',
    version = '1.0',
    description='This is your awesome module',
    author='You',
    author_email='your@email',
    package_dir = {'': 'src'},
    packages = ['your', 'you.module'],
    test_suite = 'your.module.tests',
    use_2to3 = True,
    convert_2to3_doctests = ['src/your/module/README.txt'],
    use_2to3_fixers = ['your.fixers'],
```

```
    use_2to3_exclude_fixers = ['lib2to3.fixes.fix_import'],
)
```

### 2.1.1 Differential conversion

Note that a file will only be copied and converted during the build process if the source file has been changed. If you add a file to the doctests that should be converted, it will not be converted the next time you run the tests, since it hasn't been modified. You need to remove it from the build directory. Also if you run the build, install or test commands before adding the use_2to3 parameter, you will have to remove the build directory before you run the test command, as the files otherwise will seem updated, and no conversion will happen.

In general, if code doesn't seem to be converted, deleting the build directory and trying again is a good safeguard against the build directory getting "out of sync" with the source directory.

## 2.2 Distributing Python 3 modules

You can distribute your modules with Python 3 support in different ways. A normal source distribution will work, but can be slow in installing, as the 2to3 process will be run during the install. But you can also distribute the module in binary format, such as a binary egg. That egg will contain the already converted code, and hence no 2to3 conversion is needed during install.

## 2.3 Advanced features

If you don't want to run the 2to3 conversion on the doctests in Python files, you can turn that off by setting `setuptools.use_2to3_on_doctests = False`.

## 2.4 Note on compatibility with older versions of setuptools

Setuptools earlier than 0.7 does not know about the new keyword parameters to support Python 3. As a result it will warn about the unknown keyword parameters if you use those versions of setuptools instead of Distribute under Python 2. This output is not an error, and install process will continue as normal, but if you want to get rid of that error this is easy. Simply conditionally add the new parameters into an extra dict and pass that dict into setup():

```
from setuptools import setup
import sys

extra = {}
if sys.version_info >= (3,):
    extra['use_2to3'] = True
    extra['convert_2to3_doctests'] = ['src/your/module/README.txt']
    extra['use_2to3_fixers'] = ['your.fixers']

setup(
    name='your.module',
    version = '1.0',
    description='This is your awesome module',
    author='You',
    author_email='your@email',
    package_dir = {'': 'src'},
    packages = ['your', 'you.module'],
```

```
    test_suite = 'your.module.tests',
    **extra
)
```

This way the parameters will only be used under Python 3, where Distribute or Setuptools 0.7 or later is required.

CHAPTER 3

# Using Setuptools in your project

To use Setuptools in your project, the recommended way is to ship *ez_setup.py* alongside your *setup.py* script and call it at the very beginning of *setup.py* like this:

```
from ez_setup import use_setuptools
use_setuptools()
```

CHAPTER 4

# Building and Distributing Packages with Setuptools

`Setuptools` is a collection of enhancements to the Python `distutils` (for Python 2.6 and up) that allow developers to more easily build and distribute Python packages, especially ones that have dependencies on other packages.

Packages built and distributed using `setuptools` look to the user like ordinary Python packages based on the `distutils`. Your users don't need to install or even know about setuptools in order to use them, and you don't have to include the entire setuptools package in your distributions. By including just a single bootstrap module (a 12K .py file), your package will automatically download and install `setuptools` if the user is building your package from source and doesn't have a suitable version already installed.

Feature Highlights:

- Automatically find/download/install/upgrade dependencies at build time using the EasyInstall tool, which supports downloading via HTTP, FTP, Subversion, and SourceForge, and automatically scans web pages linked from PyPI to find download links. (It's the closest thing to CPAN currently available for Python.)
- Create Python Eggs - a single-file importable distribution format
- Enhanced support for accessing data files hosted in zipped packages.
- Automatically include all packages in your source tree, without listing them individually in setup.py
- Automatically include all relevant files in your source distributions, without needing to create a `MANIFEST.in` file, and without having to force regeneration of the `MANIFEST` file when your source tree changes.
- Automatically generate wrapper scripts or Windows (console and GUI) .exe files for any number of "main" functions in your project. (Note: this is not a py2exe replacement; the .exe files rely on the local Python installation.)
- Transparent Pyrex support, so that your setup.py can list `.pyx` files and still work even when the end-user doesn't have Pyrex installed (as long as you include the Pyrex-generated C in your source distribution)
- Command aliases - create project-specific, per-user, or site-wide shortcut names for commonly used commands and options
- PyPI upload support - upload your source distributions and eggs to PyPI
- Deploy your project in "development mode", such that it's available on `sys.path`, yet can still be edited directly from its source checkout.
- Easily extend the distutils with new commands or `setup()` arguments, and distribute/reuse your extensions for multiple projects, without copying code.
- Create extensible applications and frameworks that automatically discover extensions, using simple "entry points" declared in a project's setup script.

In addition to the PyPI downloads, the development version of `setuptools` is available from the Python SVN sandbox, and in-development versions of the 0.6 branch are available as well.

**Table of Contents**

## 4.1 Developer's Guide

### 4.1.1 Installing `setuptools`

Please follow the EasyInstall Installation Instructions to install the current stable version of setuptools. In particular, be sure to read the section on Custom Installation Locations if you are installing anywhere other than Python's `site-packages` directory.

If you want the current in-development version of setuptools, you should first install a stable version, and then run:

```
ez_setup.py setuptools==dev
```

This will download and install the latest development (i.e. unstable) version of setuptools from the Python Subversion sandbox.

### 4.1.2 Basic Use

For basic use of setuptools, just import things from setuptools instead of the distutils. Here's a minimal setup script using setuptools:

```
from setuptools import setup, find_packages
setup(
    name = "HelloWorld",
    version = "0.1",
    packages = find_packages(),
)
```

As you can see, it doesn't take much to use setuptools in a project. Just by doing the above, this project will be able to produce eggs, upload to PyPI, and automatically include all packages in the directory where the setup.py lives. See the Command Reference section below to see what commands you can give to this setup script.

Of course, before you release your project to PyPI, you'll want to add a bit more information to your setup script to help people find or learn about your project. And maybe your project will have grown by then to include a few dependencies, and perhaps some data files and scripts:

```
from setuptools import setup, find_packages
setup(
    name = "HelloWorld",
    version = "0.1",
    packages = find_packages(),
    scripts = ['say_hello.py'],

    # Project uses reStructuredText, so ensure that the docutils get
    # installed or upgraded on the target machine
    install_requires = ['docutils>=0.3'],

    package_data = {
        # If any package contains *.txt or *.rst files, include them:
        '': ['*.txt', '*.rst'],
        # And include any *.msg files found in the 'hello' package, too:
        'hello': ['*.msg'],
    },

    # metadata for upload to PyPI
    author = "Me",
    author_email = "me@example.com",
    description = "This is an Example Package",
```

```
    license = "PSF",
    keywords = "hello world example examples",
    url = "http://example.com/HelloWorld/",   # project home page, if any

    # could also include long_description, download_url, classifiers, etc.
)
```

In the sections that follow, we'll explain what most of these `setup()` arguments do (except for the metadata ones), and the various ways you might use them in your own project(s).

### Specifying Your Project's Version

Setuptools can work well with most versioning schemes; there are, however, a few special things to watch out for, in order to ensure that setuptools and EasyInstall can always tell what version of your package is newer than another version. Knowing these things will also help you correctly specify what versions of other projects your project depends on.

A version consists of an alternating series of release numbers and pre-release or post-release tags. A release number is a series of digits punctuated by dots, such as `2.4` or `0.5`. Each series of digits is treated numerically, so releases `2.1` and `2.1.0` are different ways to spell the same release number, denoting the first subrelease of release 2. But `2.10` is the *tenth* subrelease of release 2, and so is a different and newer release from `2.1` or `2.1.0`. Leading zeros within a series of digits are also ignored, so `2.01` is the same as `2.1`, and different from `2.0.1`.

Following a release number, you can have either a pre-release or post-release tag. Pre-release tags make a version be considered *older* than the version they are appended to. So, revision `2.4` is *newer* than revision `2.4c1`, which in turn is newer than `2.4b1` or `2.4a1`. Postrelease tags make a version be considered *newer* than the version they are appended to. So, revisions like `2.4-1` and `2.4pl3` are newer than `2.4`, but are *older* than `2.4.1` (which has a higher release number).

A pre-release tag is a series of letters that are alphabetically before "final". Some examples of prerelease tags would include `alpha`, `beta`, `a`, `c`, `dev`, and so on. You do not have to place a dot or dash before the prerelease tag if it's immediately after a number, but it's okay to do so if you prefer. Thus, `2.4c1` and `2.4.c1` and `2.4-c1` all represent release candidate 1 of version `2.4`, and are treated as identical by setuptools.

In addition, there are three special prerelease tags that are treated as if they were the letter `c`: `pre`, `preview`, and `rc`. So, version `2.4rc1`, `2.4pre1` and `2.4preview1` are all the exact same version as `2.4c1`, and are treated as identical by setuptools.

A post-release tag is either a series of letters that are alphabetically greater than or equal to "final", or a dash (`-`). Post-release tags are generally used to separate patch numbers, port numbers, build numbers, revision numbers, or date stamps from the release number. For example, the version `2.4-r1263` might denote Subversion revision 1263 of a post-release patch of version `2.4`. Or you might use `2.4-20051127` to denote a date-stamped post-release.

Notice that after each pre or post-release tag, you are free to place another release number, followed again by more pre- or post-release tags. For example, `0.6a9.dev-r41475` could denote Subversion revision 41475 of the in-development version of the ninth alpha of release 0.6. Notice that `dev` is a pre-release tag, so this version is a *lower* version number than `0.6a9`, which would be the actual ninth alpha of release 0.6. But the `-r41475` is a post-release tag, so this version is *newer* than `0.6a9.dev`.

For the most part, setuptools' interpretation of version numbers is intuitive, but here are a few tips that will keep you out of trouble in the corner cases:

- Don't stick adjoining pre-release tags together without a dot or number between them. Version `1.9adev` is the `adev` prerelease of `1.9`, *not* a development pre-release of `1.9a`. Use `.dev` instead, as in `1.9a.dev`, or separate the prerelease tags with a number, as in `1.9a0dev`. `1.9a.dev`, `1.9a0dev`, and even `1.9.a.dev` are identical versions from setuptools' point of view, so you can use whatever scheme you prefer.

- If you want to be certain that your chosen numbering scheme works the way you think it will, you can use the `pkg_resources.parse_version()` function to compare different version numbers:

```
>>> from pkg_resources import parse_version
>>> parse_version('1.9.a.dev') == parse_version('1.9a0dev')
True
>>> parse_version('2.1-rc2') < parse_version('2.1')
True
>>> parse_version('0.6a9dev-r41475') < parse_version('0.6a9')
True
```

Once you've decided on a version numbering scheme for your project, you can have setuptools automatically tag your in-development releases with various pre- or post-release tags. See the following sections for more details:

- Tagging and "Daily Build" or "Snapshot" Releases
- Managing "Continuous Releases" Using Subversion
- The egg_info command

### 4.1.3 New and Changed `setup()` Keywords

The following keyword arguments to `setup()` are added or changed by `setuptools`. All of them are optional; you do not have to supply them unless you need the associated `setuptools` feature.

**`include_package_data`** If set to `True`, this tells `setuptools` to automatically include any data files it finds inside your package directories, that are either under CVS or Subversion control, or which are specified by your `MANIFEST.in` file. For more information, see the section below on Including Data Files.

**`exclude_package_data`** A dictionary mapping package names to lists of glob patterns that should be *excluded* from your package directories. You can use this to trim back any excess files included by `include_package_data`. For a complete description and examples, see the section below on Including Data Files.

**`package_data`** A dictionary mapping package names to lists of glob patterns. For a complete description and examples, see the section below on Including Data Files. You do not need to use this option if you are using `include_package_data`, unless you need to add e.g. files that are generated by your setup script and build process. (And are therefore not in source control or are files that you don't want to include in your source distribution.)

**`zip_safe`** A boolean (True or False) flag specifying whether the project can be safely installed and run from a zip file. If this argument is not supplied, the `bdist_egg` command will have to analyze all of your project's contents for possible problems each time it builds an egg.

**`install_requires`** A string or list of strings specifying what other distributions need to be installed when this one is. See the section below on Declaring Dependencies for details and examples of the format of this argument.

**`entry_points`** A dictionary mapping entry point group names to strings or lists of strings defining the entry points. Entry points are used to support dynamic discovery of services or plugins provided by a project. See Dynamic Discovery of Services and Plugins for details and examples of the format of this argument. In addition, this keyword is used to support Automatic Script Creation.

**`extras_require`** A dictionary mapping names of "extras" (optional features of your project) to strings or lists of strings specifying what other distributions must be installed to support those features. See the section below on Declaring Dependencies for details and examples of the format of this argument.

**`setup_requires`** A string or list of strings specifying what other distributions need to be present in order for the *setup script* to run. `setuptools` will attempt to obtain these (even going so far as to download them using `EasyInstall`) before processing the rest of the setup script or commands. This argument is needed if you are

using distutils extensions as part of your build process; for example, extensions that process setup() arguments and turn them into EGG-INFO metadata files.

(Note: projects listed in `setup_requires` will NOT be automatically installed on the system where the setup script is being run. They are simply downloaded to the setup directory if they're not locally available already. If you want them to be installed, as well as being available when the setup script is run, you should add them to `install_requires` **and** `setup_requires`.)

**`dependency_links`** A list of strings naming URLs to be searched when satisfying dependencies. These links will be used if needed to install packages specified by `setup_requires` or `tests_require`. They will also be written into the egg's metadata for use by tools like EasyInstall to use when installing an `.egg` file.

**`namespace_packages`** A list of strings naming the project's "namespace packages". A namespace package is a package that may be split across multiple project distributions. For example, Zope 3's `zope` package is a namespace package, because subpackages like `zope.interface` and `zope.publisher` may be distributed separately. The egg runtime system can automatically merge such subpackages into a single parent package at runtime, as long as you declare them in each project that contains any subpackages of the namespace package, and as long as the namespace package's `__init__.py` does not contain any code other than a namespace declaration. See the section below on Namespace Packages for more information.

**`test_suite`** A string naming a `unittest.TestCase` subclass (or a package or module containing one or more of them, or a method of such a subclass), or naming a function that can be called with no arguments and returns a `unittest.TestSuite`. If the named suite is a module, and the module has an `additional_tests()` function, it is called and the results are added to the tests to be run. If the named suite is a package, any submodules and subpackages are recursively added to the overall test suite.

Specifying this argument enables use of the test command to run the specified test suite, e.g. via `setup.py test`. See the section on the test command below for more details.

**`tests_require`** If your project's tests need one or more additional packages besides those needed to install it, you can use this option to specify them. It should be a string or list of strings specifying what other distributions need to be present for the package's tests to run. When you run the `test` command, `setuptools` will attempt to obtain these (even going so far as to download them using `EasyInstall`). Note that these required projects will *not* be installed on the system where the tests are run, but only downloaded to the project's setup directory if they're not already installed locally.

**`test_loader`** If you would like to use a different way of finding tests to run than what setuptools normally uses, you can specify a module name and class name in this argument. The named class must be instantiable with no arguments, and its instances must support the `loadTestsFromNames()` method as defined in the Python `unittest` module's `TestLoader` class. Setuptools will pass only one test "name" in the *names* argument: the value supplied for the `test_suite` argument. The loader you specify may interpret this string in any way it likes, as there are no restrictions on what may be contained in a `test_suite` string.

The module name and class name must be separated by a `:`. The default value of this argument is `"setuptools.command.test:ScanningLoader"`. If you want to use the default `unittest` behavior, you can specify `"unittest:TestLoader"` as your `test_loader` argument instead. This will prevent automatic scanning of submodules and subpackages.

The module and class you specify here may be contained in another package, as long as you use the `tests_require` option to ensure that the package containing the loader class is available when the `test` command is run.

**`eager_resources`** A list of strings naming resources that should be extracted together, if any of them is needed, or if any C extensions included in the project are imported. This argument is only useful if the project will be installed as a zipfile, and there is a need to have all of the listed resources be extracted to the filesystem *as a unit*. Resources listed here should be '/'-separated paths, relative to the source root, so to list a resource `foo.png` in package `bar.baz`, you would include the string `bar/baz/foo.png` in this argument.

If you only need to obtain resources one at a time, or you don't have any C extensions that access other files in

the project (such as data files or shared libraries), you probably do NOT need this argument and shouldn't mess with it. For more details on how this argument works, see the section below on Automatic Resource Extraction.

**use_2to3** Convert the source code from Python 2 to Python 3 with 2to3 during the build process. See *Supporting both Python 2 and Python 3 with Setuptools* for more details.

**convert_2to3_doctests** List of doctest source files that need to be converted with 2to3. See *Supporting both Python 2 and Python 3 with Setuptools* for more details.

**use_2to3_fixers** A list of modules to search for additional fixers to be used during the 2to3 conversion. See *Supporting both Python 2 and Python 3 with Setuptools* for more details.

#### Using `find_packages()`

For simple projects, it's usually easy enough to manually add packages to the `packages` argument of `setup()`. However, for very large projects (Twisted, PEAK, Zope, Chandler, etc.), it can be a big burden to keep the package list updated. That's what `setuptools.find_packages()` is for.

`find_packages()` takes a source directory and two lists of package name patterns to exclude and include. If omitted, the source directory defaults to the same directory as the setup script. Some projects use a `src` or `lib` directory as the root of their source tree, and those projects would of course use `"src"` or `"lib"` as the first argument to `find_packages()`. (And such projects also need something like `package_dir = {'':'src'}` in their `setup()` arguments, but that's just a normal distutils thing.)

Anyway, `find_packages()` walks the target directory, filtering by inclusion patterns, and finds Python packages (any directory). On Python 3.2 and earlier, packages are only recognized if they include an `__init__.py` file. Finally, exclusion patterns are applied to remove matching packages.

Inclusion and exclusion patterns are package names, optionally including wildcards. For example, `find_packages(exclude=["*.tests"])` will exclude all packages whose last name part is `tests`. Or, `find_packages(exclude=["*.tests", "*.tests.*"])` will also exclude any subpackages of packages named `tests`, but it still won't exclude a top-level `tests` package or the children thereof. In fact, if you really want no `tests` packages at all, you'll need something like this:

```
find_packages(exclude=["*.tests", "*.tests.*", "tests.*", "tests"])
```

in order to cover all the bases. Really, the exclusion patterns are intended to cover simpler use cases than this, like excluding a single, specified package and its subpackages.

Regardless of the parameters, the `find_packages()` function returns a list of package names suitable for use as the `packages` argument to `setup()`, and so is usually the easiest way to set that argument in your setup script. Especially since it frees you from having to remember to modify your setup script whenever your project grows additional top-level packages or subpackages.

### 4.1.4 Automatic Script Creation

Packaging and installing scripts can be a bit awkward with the distutils. For one thing, there's no easy way to have a script's filename match local conventions on both Windows and POSIX platforms. For another, you often have to create a separate file just for the "main" script, when your actual "main" is a function in a module somewhere. And even in Python 2.4, using the `-m` option only works for actual `.py` files that aren't installed in a package.

`setuptools` fixes all of these problems by automatically generating scripts for you with the correct extension, and on Windows it will even create an `.exe` file so that users don't have to change their `PATHEXT` settings. The way to use this feature is to define "entry points" in your setup script that indicate what function the generated script should import and run. For example, to create two console scripts called `foo` and `bar`, and a GUI script called `baz`, you might do something like this:

```
setup(
    # other arguments here...
    entry_points = {
        'console_scripts': [
            'foo = my_package.some_module:main_func',
            'bar = other_module:some_func',
        ],
        'gui_scripts': [
            'baz = my_package_gui.start_func',
        ]
    }
)
```

When this project is installed on non-Windows platforms (using "setup.py install", "setup.py develop", or by using EasyInstall), a set of `foo`, `bar`, and `baz` scripts will be installed that import `main_func` and `some_func` from the specified modules. The functions you specify are called with no arguments, and their return value is passed to `sys.exit()`, so you can return an errorlevel or message to print to stderr.

On Windows, a set of `foo.exe`, `bar.exe`, and `baz.exe` launchers are created, alongside a set of `foo.py`, `bar.py`, and `baz.pyw` files. The `.exe` wrappers find and execute the right version of Python to run the `.py` or `.pyw` file.

You may define as many "console script" and "gui script" entry points as you like, and each one can optionally specify "extras" that it depends on, that will be added to `sys.path` when the script is run. For more information on "extras", see the section below on Declaring Extras. For more information on "entry points" in general, see the section below on Dynamic Discovery of Services and Plugins.

### "Eggsecutable" Scripts

Occasionally, there are situations where it's desirable to make an `.egg` file directly executable. You can do this by including an entry point such as the following:

```
setup(
    # other arguments here...
    entry_points = {
        'setuptools.installation': [
            'eggsecutable = my_package.some_module:main_func',
        ]
    }
)
```

Any eggs built from the above setup script will include a short executable prelude that imports and calls `main_func()` from `my_package.some_module`. The prelude can be run on Unix-like platforms (including Mac and Linux) by invoking the egg with `/bin/sh`, or by enabling execute permissions on the `.egg` file. For the executable prelude to run, the appropriate version of Python must be available via the `PATH` environment variable, under its "long" name. That is, if the egg is built for Python 2.3, there must be a `python2.3` executable present in a directory on `PATH`.

This feature is primarily intended to support ez_setup the installation of setuptools itself on non-Windows platforms, but may also be useful for other projects as well.

IMPORTANT NOTE: Eggs with an "eggsecutable" header cannot be renamed, or invoked via symlinks. They *must* be invoked using their original filename, in order to ensure that, once running, `pkg_resources` will know what project and version is in use. The header script will check this and exit with an error if the `.egg` file has been renamed or is invoked via a symlink that changes its base name.

### 4.1.5 Declaring Dependencies

`setuptools` supports automatically installing dependencies when a package is installed, and including information about dependencies in Python Eggs (so that package management tools like EasyInstall can use the information).

`setuptools` and `pkg_resources` use a common syntax for specifying a project's required dependencies. This syntax consists of a project's PyPI name, optionally followed by a comma-separated list of "extras" in square brackets, optionally followed by a comma-separated list of version specifiers. A version specifier is one of the operators <, >, <=, >=, == or !=, followed by a version identifier. Tokens may be separated by whitespace, but any whitespace or nonstandard characters within a project name or version identifier must be replaced with -.

Version specifiers for a given project are internally sorted into ascending version order, and used to establish what ranges of versions are acceptable. Adjacent redundant conditions are also consolidated (e.g. `">1, >2"` becomes `">1"`, and `"<2,<3"` becomes `"<3"`). `"!="` versions are excised from the ranges they fall within. A project's version is then checked for membership in the resulting ranges. (Note that providing conflicting conditions for the same version (e.g. "<2,>=2" or "==2,!=2") is meaningless and may therefore produce bizarre results.)

Here are some example requirement specifiers:

```
docutils >= 0.3

# comment lines and \ continuations are allowed in requirement strings
BazSpam ==1.1, ==1.2, ==1.3, ==1.4, ==1.5, \
    ==1.6, ==1.7  # and so are line-end comments

PEAK[FastCGI, reST]>=0.5a4

setuptools==0.5a7
```

The simplest way to include requirement specifiers is to use the `install_requires` argument to `setup()`. It takes a string or list of strings containing requirement specifiers. If you include more than one requirement in a string, each requirement must begin on a new line.

This has three effects:

1. When your project is installed, either by using EasyInstall, `setup.py install`, or `setup.py develop`, all of the dependencies not already installed will be located (via PyPI), downloaded, built (if necessary), and installed.
2. Any scripts in your project will be installed with wrappers that verify the availability of the specified dependencies at runtime, and ensure that the correct versions are added to `sys.path` (e.g. if multiple versions have been installed).
3. Python Egg distributions will include a metadata file listing the dependencies.

Note, by the way, that if you declare your dependencies in `setup.py`, you do *not* need to use the `require()` function in your scripts or modules, as long as you either install the project or use `setup.py develop` to do development work on it. (See "Development Mode" below for more details on using `setup.py develop`.)

#### Dependencies that aren't in PyPI

If your project depends on packages that aren't registered in PyPI, you may still be able to depend on them, as long as they are available for download as:

- an egg, in the standard distutils `sdist` format,
- a single `.py` file, or
- a VCS repository (Subversion, Mercurial, or Git).

You just need to add some URLs to the `dependency_links` argument to `setup()`.

The URLs must be either:

1. direct download URLs,
2. the URLs of web pages that contain direct download links, or
3. the repository's URL

In general, it's better to link to web pages, because it is usually less complex to update a web page than to release a new version of your project. You can also use a SourceForge `showfiles.php` link in the case where a package you depend on is distributed via SourceForge.

If you depend on a package that's distributed as a single `.py` file, you must include an `"#egg=project-version"` suffix to the URL, to give a project name and version number. (Be sure to escape any dashes in the name or version by replacing them with underscores.) EasyInstall will recognize this suffix and automatically create a trivial `setup.py` to wrap the single `.py` file as an egg.

In the case of a VCS checkout, you should also append `#egg=project-version` in order to identify for what package that checkout should be used. You can append `@REV` to the URL's path (before the fragment) to specify a revision. Additionally, you can also force the VCS being used by prepending the URL with a certain prefix. Currently available are:

- `svn+URL` for Subversion,
- `git+URL` for Git, and
- `hg+URL` for Mercurial

A more complete example would be:

```
vcs+proto://host/path@revision#egg=project-version
```

Be careful with the version. It should match the one inside the project files. If you want to disregard the version, you have to omit it both in the `requires` and in the URL's fragment.

This will do a checkout (or a clone, in Git and Mercurial parlance) to a temporary folder and run `setup.py bdist_egg`.

The `dependency_links` option takes the form of a list of URL strings. For example, the below will cause EasyInstall to search the specified page for eggs or source distributions, if the package's dependencies aren't already installed:

```
setup(
    ...
    dependency_links = [
        "http://peak.telecommunity.com/snapshots/"
    ],
)
```

### Declaring "Extras" (optional features with their own dependencies)

Sometimes a project has "recommended" dependencies, that are not required for all uses of the project. For example, a project might offer optional PDF output if ReportLab is installed, and reStructuredText support if docutils is installed. These optional features are called "extras", and setuptools allows you to define their requirements as well. In this way, other projects that require these optional features can force the additional requirements to be installed, by naming the desired extras in their `install_requires`.

For example, let's say that Project A offers optional PDF and reST support:

```
setup(
    name="Project-A",
    ...
    extras_require = {
        'PDF':  ["ReportLab>=1.2", "RXP"],
        'reST': ["docutils>=0.3"],
    }
)
```

As you can see, the `extras_require` argument takes a dictionary mapping names of "extra" features, to strings or lists of strings describing those features' requirements. These requirements will *not* be automatically installed unless another package depends on them (directly or indirectly) by including the desired "extras" in square brackets after the associated project name. (Or if the extras were listed in a requirement spec on the EasyInstall command line.)

Extras can be used by a project's entry points to specify dynamic dependencies. For example, if Project A includes a "rst2pdf" script, it might declare it like this, so that the "PDF" requirements are only resolved if the "rst2pdf" script is run:

```
setup(
    name="Project-A",
    ...
    entry_points = {
        'console_scripts': [
            'rst2pdf = project_a.tools.pdfgen [PDF]',
            'rst2html = project_a.tools.htmlgen',
            # more script entry points ...
        ],
    }
)
```

Projects can also use another project's extras when specifying dependencies. For example, if project B needs "project A" with PDF support installed, it might declare the dependency like this:

```
setup(
    name="Project-B",
    install_requires = ["Project-A[PDF]"],
    ...
)
```

This will cause ReportLab to be installed along with project A, if project B is installed – even if project A was already installed. In this way, a project can encapsulate groups of optional "downstream dependencies" under a feature name, so that packages that depend on it don't have to know what the downstream dependencies are. If a later version of Project A builds in PDF support and no longer needs ReportLab, or if it ends up needing other dependencies besides ReportLab in order to provide PDF support, Project B's setup information does not need to change, but the right packages will still be installed if needed.

Note, by the way, that if a project ends up not needing any other packages to support a feature, it should keep an empty requirements list for that feature in its `extras_require` argument, so that packages depending on that feature don't break (due to an invalid feature name). For example, if Project A above builds in PDF support and no longer needs ReportLab, it could change its setup to this:

```
setup(
    name="Project-A",
    ...
    extras_require = {
        'PDF':  [],
        'reST': ["docutils>=0.3"],
    }
)
```

so that Package B doesn't have to remove the `[PDF]` from its requirement specifier.

### 4.1.6 Including Data Files

The distutils have traditionally allowed installation of "data files", which are placed in a platform-specific location. However, the most common use case for data files distributed with a package is for use *by* the package, usually by including the data files in the package directory.

Setuptools offers three ways to specify data files to be included in your packages. First, you can simply use the `include_package_data` keyword, e.g.:

```
from setuptools import setup, find_packages
setup(
    ...
    include_package_data = True
)
```

This tells setuptools to install any data files it finds in your packages. The data files must be under CVS or Subversion control, or else they must be specified via the distutils' `MANIFEST.in` file. (They can also be tracked by another revision control system, using an appropriate plugin. See the section below on Adding Support for Other Revision Control Systems for information on how to write such plugins.)

If the data files are not under version control, or are not in a supported version control system, or if you want finer-grained control over what files are included (for example, if you have documentation files in your package directories and want to exclude them from installation), then you can also use the `package_data` keyword, e.g.:

```
from setuptools import setup, find_packages
setup(
    ...
    package_data = {
        # If any package contains *.txt or *.rst files, include them:
        '': ['*.txt', '*.rst'],
        # And include any *.msg files found in the 'hello' package, too:
        'hello': ['*.msg'],
    }
)
```

The `package_data` argument is a dictionary that maps from package names to lists of glob patterns. The globs may include subdirectory names, if the data files are contained in a subdirectory of the package. For example, if the package tree looks like this:

```
setup.py
src/
    mypkg/
        __init__.py
        mypkg.txt
        data/
            somefile.dat
            otherdata.dat
```

The setuptools setup file might look like this:

```
from setuptools import setup, find_packages
setup(
    ...
    packages = find_packages('src'),  # include all packages under src
    package_dir = {'':'src'},   # tell distutils packages are under src

    package_data = {
```

```
        # If any package contains *.txt files, include them:
        '': ['*.txt'],
        # And include any *.dat files found in the 'data' subdirectory
        # of the 'mypkg' package, also:
        'mypkg': ['data/*.dat'],
    }
)
```

Notice that if you list patterns in `package_data` under the empty string, these patterns are used to find files in every package, even ones that also have their own patterns listed. Thus, in the above example, the `mypkg.txt` file gets included even though it's not listed in the patterns for `mypkg`.

Also notice that if you use paths, you *must* use a forward slash (`/`) as the path separator, even if you are on Windows. Setuptools automatically converts slashes to appropriate platform-specific separators at build time.

(Note: although the `package_data` argument was previously only available in `setuptools`, it was also added to the Python `distutils` package as of Python 2.4; there is some documentation for the feature available on the python.org website. If using the setuptools-specific `include_package_data` argument, files specified by `package_data` will *not* be automatically added to the manifest unless they are tracked by a supported version control system, or are listed in the MANIFEST.in file.)

Sometimes, the `include_package_data` or `package_data` options alone aren't sufficient to precisely define what files you want included. For example, you may want to include package README files in your revision control system and source distributions, but exclude them from being installed. So, setuptools offers an `exclude_package_data` option as well, that allows you to do things like this:

```
from setuptools import setup, find_packages
setup(
    ...
    packages = find_packages('src'),  # include all packages under src
    package_dir = {'':'src'},   # tell distutils packages are under src

    include_package_data = True,    # include everything in source control

    # ...but exclude README.txt from all packages
    exclude_package_data = { '': ['README.txt'] },
)
```

The `exclude_package_data` option is a dictionary mapping package names to lists of wildcard patterns, just like the `package_data` option. And, just as with that option, a key of `''` will apply the given pattern(s) to all packages. However, any files that match these patterns will be *excluded* from installation, even if they were listed in `package_data` or were included as a result of using `include_package_data`.

In summary, the three options allow you to:

**`include_package_data`** Accept all data files and directories matched by `MANIFEST.in` or found in source control.

**`package_data`** Specify additional patterns to match files and directories that may or may not be matched by `MANIFEST.in` or found in source control.

**`exclude_package_data`** Specify patterns for data files and directories that should *not* be included when a package is installed, even if they would otherwise have been included due to the use of the preceding options.

NOTE: Due to the way the distutils build process works, a data file that you include in your project and then stop including may be "orphaned" in your project's build directories, requiring you to run `setup.py clean --all` to fully remove them. This may also be important for your users and contributors if they track intermediate revisions of your project using Subversion; be sure to let them know when you make changes that remove files from inclusion so they can run `setup.py clean --all`.

### Accessing Data Files at Runtime

Typically, existing programs manipulate a package's `__file__` attribute in order to find the location of data files. However, this manipulation isn't compatible with PEP 302-based import hooks, including importing from zip files and Python Eggs. It is strongly recommended that, if you are using data files, you should use the Resource Management API of `pkg_resources` to access them. The `pkg_resources` module is distributed as part of setuptools, so if you're using setuptools to distribute your package, there is no reason not to use its resource management API. See also Accessing Package Resources for a quick example of converting code that uses `__file__` to use `pkg_resources` instead.

### Non-Package Data Files

The `distutils` normally install general "data files" to a platform-specific location (e.g. `/usr/share`). This feature intended to be used for things like documentation, example configuration files, and the like. `setuptools` does not install these data files in a separate location, however. They are bundled inside the egg file or directory, alongside the Python modules and packages. The data files can also be accessed using the Resource Management API, by specifying a `Requirement` instead of a package name:

```
from pkg_resources import Requirement, resource_filename
filename = resource_filename(Requirement.parse("MyProject"),"sample.conf")
```

The above code will obtain the filename of the "sample.conf" file in the data root of the "MyProject" distribution.

Note, by the way, that this encapsulation of data files means that you can't actually install data files to some arbitrary location on a user's machine; this is a feature, not a bug. You can always include a script in your distribution that extracts and copies your the documentation or data files to a user-specified location, at their discretion. If you put related data files in a single directory, you can use `resource_filename()` with the directory name to get a filesystem directory that then can be copied with the `shutil` module. (Even if your package is installed as a zipfile, calling `resource_filename()` on a directory will return an actual filesystem directory, whose contents will be that entire subtree of your distribution.)

(Of course, if you're writing a new package, you can just as easily place your data files or directories inside one of your packages, rather than using the distutils' approach. However, if you're updating an existing application, it may be simpler not to change the way it currently specifies these data files.)

### Automatic Resource Extraction

If you are using tools that expect your resources to be "real" files, or your project includes non-extension native libraries or other files that your C extensions expect to be able to access, you may need to list those files in the `eager_resources` argument to `setup()`, so that the files will be extracted together, whenever a C extension in the project is imported.

This is especially important if your project includes shared libraries *other* than distutils-built C extensions, and those shared libraries use file extensions other than `.dll`, `.so`, or `.dylib`, which are the extensions that setuptools 0.6a8 and higher automatically detects as shared libraries and adds to the `native_libs.txt` file for you. Any shared libraries whose names do not end with one of those extensions should be listed as `eager_resources`, because they need to be present in the filesystem when he C extensions that link to them are used.

The `pkg_resources` runtime for compressed packages will automatically extract *all* C extensions and `eager_resources` at the same time, whenever *any* C extension or eager resource is requested via the `resource_filename()` API. (C extensions are imported using `resource_filename()` internally.) This ensures that C extensions will see all of the "real" files that they expect to see.

Note also that you can list directory resource names in `eager_resources` as well, in which case the directory's contents (including subdirectories) will be extracted whenever any C extension or eager resource is requested.

Please note that if you're not sure whether you need to use this argument, you don't! It's really intended to support projects with lots of non-Python dependencies and as a last resort for crufty projects that can't otherwise handle being compressed. If your package is pure Python, Python plus data files, or Python plus C, you really don't need this. You've got to be using either C or an external program that needs "real" files in your project before there's any possibility of `eager_resources` being relevant to your project.

### 4.1.7 Extensible Applications and Frameworks

#### Dynamic Discovery of Services and Plugins

`setuptools` supports creating libraries that "plug in" to extensible applications and frameworks, by letting you register "entry points" in your project that can be imported by the application or framework.

For example, suppose that a blogging tool wants to support plugins that provide translation for various file types to the blog's output format. The framework might define an "entry point group" called `blogtool.parsers`, and then allow plugins to register entry points for the file extensions they support.

This would allow people to create distributions that contain one or more parsers for different file types, and then the blogging tool would be able to find the parsers at runtime by looking up an entry point for the file extension (or mime type, or however it wants to).

Note that if the blogging tool includes parsers for certain file formats, it can register these as entry points in its own setup script, which means it doesn't have to special-case its built-in formats. They can just be treated the same as any other plugin's entry points would be.

If you're creating a project that plugs in to an existing application or framework, you'll need to know what entry points or entry point groups are defined by that application or framework. Then, you can register entry points in your setup script. Here are a few examples of ways you might register an `.rst` file parser entry point in the `blogtool.parsers` entry point group, for our hypothetical blogging tool:

```
setup(
    # ...
    entry_points = {'blogtool.parsers': '.rst = some_module:SomeClass'}
)

setup(
    # ...
    entry_points = {'blogtool.parsers': ['.rst = some_module:a_func']}
)

setup(
    # ...
    entry_points = """
        [blogtool.parsers]
        .rst = some.nested.module:SomeClass.some_classmethod [reST]
    """,
    extras_require = dict(reST = "Docutils>=0.3.5")
)
```

The `entry_points` argument to `setup()` accepts either a string with `.ini`-style sections, or a dictionary mapping entry point group names to either strings or lists of strings containing entry point specifiers. An entry point specifier consists of a name and value, separated by an = sign. The value consists of a dotted module name, optionally followed by a `:` and a dotted identifier naming an object within the module. It can also include a bracketed list of "extras" that are required for the entry point to be used. When the invoking application or framework requests loading of an entry point, any requirements implied by the associated extras will be passed to `pkg_resources.require()`, so that an appropriate error message can be displayed if the needed package(s) are missing. (Of course, the invoking app or framework can ignore such errors if it wants to make an entry point optional if a requirement isn't installed.)

#### Defining Additional Metadata

Some extensible applications and frameworks may need to define their own kinds of metadata to include in eggs, which they can then access using the `pkg_resources` metadata APIs. Ordinarily, this is done by having plugin developers include additional files in their `ProjectName.egg-info` directory. However, since it can be tedious to create such files by hand, you may want to create a distutils extension that will create the necessary files from arguments to `setup()`, in much the same way that `setuptools` does for many of the `setup()` arguments it adds. See the section below on Creating distutils Extensions for more details, especially the subsection on Adding new EGG-INFO Files.

### 4.1.8 "Development Mode"

Under normal circumstances, the `distutils` assume that you are going to build a distribution of your project, not use it in its "raw" or "unbuilt" form. If you were to use the `distutils` that way, you would have to rebuild and reinstall your project every time you made a change to it during development.

Another problem that sometimes comes up with the `distutils` is that you may need to do development on two related projects at the same time. You may need to put both projects' packages in the same directory to run them, but need to keep them separate for revision control purposes. How can you do this?

Setuptools allows you to deploy your projects for use in a common directory or staging area, but without copying any files. Thus, you can edit each project's code in its checkout directory, and only need to run build commands when you change a project's C extensions or similarly compiled files. You can even deploy a project into another project's checkout directory, if that's your preferred way of working (as opposed to using a common independent staging area or the site-packages directory).

To do this, use the `setup.py develop` command. It works very similarly to `setup.py install` or the EasyInstall tool, except that it doesn't actually install anything. Instead, it creates a special `.egg-link` file in the deployment directory, that links to your project's source code. And, if your deployment directory is Python's `site-packages` directory, it will also update the `easy-install.pth` file to include your project's source code, thereby making it available on `sys.path` for all programs using that Python installation.

If you have enabled the `use_2to3` flag, then of course the `.egg-link` will not link directly to your source code when run under Python 3, since that source code would be made for Python 2 and not work under Python 3. Instead the `setup.py develop` will build Python 3 code under the `build` directory, and link there. This means that after doing code changes you will have to run `setup.py build` before these changes are picked up by your Python 3 installation.

In addition, the `develop` command creates wrapper scripts in the target script directory that will run your in-development scripts after ensuring that all your `install_requires` packages are available on `sys.path`.

You can deploy the same project to multiple staging areas, e.g. if you have multiple projects on the same machine that are sharing the same project you're doing development work.

When you're done with a given development task, you can remove the project source from a staging area using `setup.py develop --uninstall`, specifying the desired staging area if it's not the default.

There are several options to control the precise behavior of the `develop` command; see the section on the develop command below for more details.

Note that you can also apply setuptools commands to non-setuptools projects, using commands like this:

```
python -c "import setuptools; execfile('setup.py')" develop
```

That is, you can simply list the normal setup commands and options following the quoted part.

### 4.1.9 Distributing a `setuptools`-based project

#### Using `setuptools`... Without bundling it!

Your users might not have `setuptools` installed on their machines, or even if they do, it might not be the right version. Fixing this is easy; just download ez_setup.py, and put it in the same directory as your `setup.py` script. (Be sure to add it to your revision control system, too.) Then add these two lines to the very top of your setup script, before the script imports anything from setuptools:

```
import ez_setup
ez_setup.use_setuptools()
```

That's it. The `ez_setup` module will automatically download a matching version of `setuptools` from PyPI, if it isn't present on the target system. Whenever you install an updated version of setuptools, you should also update your projects' `ez_setup.py` files, so that a matching version gets installed on the target machine(s).

By the way, setuptools supports the new PyPI "upload" command, so you can use `setup.py sdist upload` or `setup.py bdist_egg upload` to upload your source or egg distributions respectively. Your project's current version must be registered with PyPI first, of course; you can use `setup.py register` to do that. Or you can do it all in one step, e.g. `setup.py register sdist bdist_egg upload` will register the package, build source and egg distributions, and then upload them both to PyPI, where they'll be easily found by other projects that depend on them.

(By the way, if you need to distribute a specific version of `setuptools`, you can specify the exact version and base download URL as parameters to the `use_setuptools()` function. See the function's docstring for details.)

#### What Your Users Should Know

In general, a setuptools-based project looks just like any distutils-based project – as long as your users have an internet connection and are installing to `site-packages`, that is. But for some users, these conditions don't apply, and they may become frustrated if this is their first encounter with a setuptools-based project. To keep these users happy, you should review the following topics in your project's installation instructions, if they are relevant to your project and your target audience isn't already familiar with setuptools and `easy_install`.

**Network Access** If your project is using `ez_setup`, you should inform users of the need to either have network access, or to preinstall the correct version of setuptools using the EasyInstall installation instructions. Those instructions also have tips for dealing with firewalls as well as how to manually download and install setuptools.

**Custom Installation Locations** You should inform your users that if they are installing your project to somewhere other than the main `site-packages` directory, they should first install setuptools using the instructions for Custom Installation Locations, before installing your project.

**Your Project's Dependencies** If your project depends on other projects that may need to be downloaded from PyPI or elsewhere, you should list them in your installation instructions, or tell users how to find out what they are. While most users will not need this information, any users who don't have unrestricted internet access may have to find, download, and install the other projects manually. (Note, however, that they must still install those projects using `easy_install`, or your project will not know they are installed, and your setup script will try to download them again.)

If you want to be especially friendly to users with limited network access, you may wish to build eggs for your project and its dependencies, making them all available for download from your site, or at least create a page with links to all of the needed eggs. In this way, users with limited network access can manually download all the eggs to a single directory, then use the `-f` option of `easy_install` to specify the directory to find eggs in. Users who have full network access can just use `-f` with the URL of your download page, and `easy_install` will find all the needed eggs using your links directly. This is also useful when your target audience isn't able to compile packages (e.g. most Windows users) and your package or some of its dependencies include C code.

**Subversion or CVS Users and Co-Developers** Users and co-developers who are tracking your in-development code using CVS, Subversion, or some other revision control system should probably read this manual's sections regarding such development. Alternately, you may wish to create a quick-reference guide containing the tips from this manual that apply to your particular situation. For example, if you recommend that people use `setup.py develop` when tracking your in-development code, you should let them know that this needs to be run after every update or commit.

Similarly, if you remove modules or data files from your project, you should remind them to run `setup.py clean --all` and delete any obsolete `.pyc` or `.pyo`. (This tip applies to the distutils in general, not just setuptools, but not everybody knows about them; be kind to your users by spelling out your project's best practices rather than leaving them guessing.)

**Creating System Packages** Some users want to manage all Python packages using a single package manager, and sometimes that package manager isn't `easy_install`! Setuptools currently supports `bdist_rpm`, `bdist_wininst`, and `bdist_dumb` formats for system packaging. If a user has a locally- installed "bdist" packaging tool that internally uses the distutils `install` command, it should be able to work with `setuptools`. Some examples of "bdist" formats that this should work with include the `bdist_nsi` and `bdist_msi` formats for Windows.

However, packaging tools that build binary distributions by running `setup.py install` on the command line or as a subprocess will require modification to work with setuptools. They should use the `--single-version-externally-managed` option to the `install` command, combined with the standard `--root` or `--record` options. See the install command documentation below for more details. The `bdist_deb` command is an example of a command that currently requires this kind of patching to work with setuptools.

If you or your users have a problem building a usable system package for your project, please report the problem via the mailing list so that either the "bdist" tool in question or setuptools can be modified to resolve the issue.

### Setting the `zip_safe` flag

For maximum performance, Python packages are best installed as zip files. Not all packages, however, are capable of running in compressed form, because they may expect to be able to access either source code or data files as normal operating system files. So, `setuptools` can install your project as a zipfile or a directory, and its default choice is determined by the project's `zip_safe` flag.

You can pass a True or False value for the `zip_safe` argument to the `setup()` function, or you can omit it. If you omit it, the `bdist_egg` command will analyze your project's contents to see if it can detect any conditions that would prevent it from working in a zipfile. It will output notices to the console about any such conditions that it finds.

Currently, this analysis is extremely conservative: it will consider the project unsafe if it contains any C extensions or datafiles whatsoever. This does *not* mean that the project can't or won't work as a zipfile! It just means that the `bdist_egg` authors aren't yet comfortable asserting that the project *will* work. If the project contains no C or data files, and does no `__file__` or `__path__` introspection or source code manipulation, then there is an extremely solid chance the project will work when installed as a zipfile. (And if the project uses `pkg_resources` for all its data file access, then C extensions and other data files shouldn't be a problem at all. See the Accessing Data Files at Runtime section above for more information.)

However, if `bdist_egg` can't be *sure* that your package will work, but you've checked over all the warnings it issued, and you are either satisfied it *will* work (or if you want to try it for yourself), then you should set `zip_safe` to `True` in your `setup()` call. If it turns out that it doesn't work, you can always change it to `False`, which will force `setuptools` to install your project as a directory rather than as a zipfile.

Of course, the end-user can still override either decision, if they are using EasyInstall to install your package. And, if you want to override for testing purposes, you can just run `setup.py easy_install --zip-ok .` or `setup.py easy_install --always-unzip .` in your project directory. to install the package as a zipfile or directory, respectively.

In the future, as we gain more experience with different packages and become more satisfied with the robustness of the pkg_resources runtime, the "zip safety" analysis may become less conservative. However, we strongly recommend that you determine for yourself whether your project functions correctly when installed as a zipfile, correct any problems if you can, and then make an explicit declaration of True or False for the zip_safe flag, so that it will not be necessary for bdist_egg or EasyInstall to try to guess whether your project can work as a zipfile.

### Namespace Packages

Sometimes, a large package is more useful if distributed as a collection of smaller eggs. However, Python does not normally allow the contents of a package to be retrieved from more than one location. "Namespace packages" are a solution for this problem. When you declare a package to be a namespace package, it means that the package has no meaningful contents in its __init__.py, and that it is merely a container for modules and subpackages.

The pkg_resources runtime will then automatically ensure that the contents of namespace packages that are spread over multiple eggs or directories are combined into a single "virtual" package.

The namespace_packages argument to setup() lets you declare your project's namespace packages, so that they will be included in your project's metadata. The argument should list the namespace packages that the egg participates in. For example, the ZopeInterface project might do this:

```
setup(
    # ...
    namespace_packages = ['zope']
)
```

because it contains a zope.interface package that lives in the zope namespace package. Similarly, a project for a standalone zope.publisher would also declare the zope namespace package. When these projects are installed and used, Python will see them both as part of a "virtual" zope package, even though they will be installed in different locations.

Namespace packages don't have to be top-level packages. For example, Zope 3's zope.app package is a namespace package, and in the future PEAK's peak.util package will be too.

Note, by the way, that your project's source tree must include the namespace packages' __init__.py files (and the __init__.py of any parent packages), in a normal Python package layout. These __init__.py files *must* contain the line:

```
__import__('pkg_resources').declare_namespace(__name__)
```

This code ensures that the namespace package machinery is operating and that the current package is registered as a namespace package.

You must NOT include any other code and data in a namespace package's __init__.py. Even though it may appear to work during development, or when projects are installed as .egg files, it will not work when the projects are installed using "system" packaging tools – in such cases the __init__.py files will not be installed, let alone executed.

You must include the declare_namespace() line in the __init__.py of *every* project that has contents for the namespace package in question, in order to ensure that the namespace will be declared regardless of which project's copy of __init__.py is loaded first. If the first loaded __init__.py doesn't declare it, it will never *be* declared, because no other copies will ever be loaded!

#### TRANSITIONAL NOTE

Setuptools automatically calls declare_namespace() for you at runtime, but future versions may *not*. This is because the automatic declaration feature has some negative side effects, such as needing to import all namespace packages during the initialization of the pkg_resources runtime, and also the need for pkg_resources to be

explicitly imported before any namespace packages work at all. In some future releases, you'll be responsible for including your own declaration lines, and the automatic declaration feature will be dropped to get rid of the negative side effects.

During the remainder of the current development cycle, therefore, setuptools will warn you about missing `declare_namespace()` calls in your `__init__.py` files, and you should correct these as soon as possible before the compatibility support is removed. Namespace packages without declaration lines will not work correctly once a user has upgraded to a later version, so it's important that you make this change now in order to avoid having your code break in the field. Our apologies for the inconvenience, and thank you for your patience.

### Tagging and "Daily Build" or "Snapshot" Releases

When a set of related projects are under development, it may be important to track finer-grained version increments than you would normally use for e.g. "stable" releases. While stable releases might be measured in dotted numbers with alpha/beta/etc. status codes, development versions of a project often need to be tracked by revision or build number or even build date. This is especially true when projects in development need to refer to one another, and therefore may literally need an up-to-the-minute version of something!

To support these scenarios, `setuptools` allows you to "tag" your source and egg distributions by adding one or more of the following to the project's "official" version identifier:

- A manually-specified pre-release tag, such as "build" or "dev", or a manually-specified post-release tag, such as a build or revision number (`--tag-build=STRING, -bSTRING`)
- A "last-modified revision number" string generated automatically from Subversion's metadata (assuming your project is being built from a Subversion "working copy") (`--tag-svn-revision, -r`)
- An 8-character representation of the build date (`--tag-date, -d`), as a postrelease tag

You can add these tags by adding `egg_info` and the desired options to the command line ahead of the `sdist` or `bdist` commands that you want to generate a daily build or snapshot for. See the section below on the egg_info command for more details.

(Also, before you release your project, be sure to see the section above on Specifying Your Project's Version for more information about how pre- and post-release tags affect how setuptools and EasyInstall interpret version numbers. This is important in order to make sure that dependency processing tools will know which versions of your project are newer than others.)

Finally, if you are creating builds frequently, and either building them in a downloadable location or are copying them to a distribution server, you should probably also check out the rotate command, which lets you automatically delete all but the N most-recently-modified distributions matching a glob pattern. So, you can use a command line like:

```
setup.py egg_info -rbDEV bdist_egg rotate -m.egg -k3
```

to build an egg whose version info includes 'DEV-rNNNN' (where NNNN is the most recent Subversion revision that affected the source tree), and then delete any egg files from the distribution directory except for the three that were built most recently.

If you have to manage automated builds for multiple packages, each with different tagging and rotation policies, you may also want to check out the alias command, which would let each package define an alias like `daily` that would perform the necessary tag, build, and rotate commands. Then, a simpler script or cron job could just run `setup.py daily` in each project directory. (And, you could also define sitewide or per-user default versions of the `daily` alias, so that projects that didn't define their own would use the appropriate defaults.)

### Generating Source Distributions

`setuptools` enhances the distutils' default algorithm for source file selection, so that all files managed by CVS or Subversion in your project tree are included in any source distribution you build. This is a big improvement over

having to manually write a `MANIFEST.in` file and try to keep it in sync with your project. So, if you are using CVS or Subversion, and your source distributions only need to include files that you're tracking in revision control, don't create a a `MANIFEST.in` file for your project. (And, if you already have one, you might consider deleting it the next time you would otherwise have to change it.)

(NOTE: other revision control systems besides CVS and Subversion can be supported using plugins; see the section below on Adding Support for Other Revision Control Systems for information on how to write such plugins.)

If you need to include automatically generated files, or files that are kept in an unsupported revision control system, you'll need to create a `MANIFEST.in` file to specify any files that the default file location algorithm doesn't catch. See the distutils documentation for more information on the format of the `MANIFEST.in` file.

But, be sure to ignore any part of the distutils documentation that deals with `MANIFEST` or how it's generated from `MANIFEST.in`; setuptools shields you from these issues and doesn't work the same way in any case. Unlike the distutils, setuptools regenerates the source distribution manifest file every time you build a source distribution, and it builds it inside the project's `.egg-info` directory, out of the way of your main project directory. You therefore need not worry about whether it is up-to-date or not.

Indeed, because setuptools' approach to determining the contents of a source distribution is so much simpler, its `sdist` command omits nearly all of the options that the distutils' more complex `sdist` process requires. For all practical purposes, you'll probably use only the `--formats` option, if you use any option at all.

(By the way, if you're using some other revision control system, you might consider creating and publishing a revision control plugin for setuptools.)

### Making your package available for EasyInstall

If you use the `register` command (`setup.py register`) to register your package with PyPI, that's most of the battle right there. (See the docs for the register command for more details.)

If you also use the upload command to upload actual distributions of your package, that's even better, because EasyInstall will be able to find and download them directly from your project's PyPI page.

However, there may be reasons why you don't want to upload distributions to PyPI, and just want your existing distributions (or perhaps a Subversion checkout) to be used instead.

So here's what you need to do before running the `register` command. There are three `setup()` arguments that affect EasyInstall:

**`url` and `download_url`** These become links on your project's PyPI page. EasyInstall will examine them to see if they link to a package ("primary links"), or whether they are HTML pages. If they're HTML pages, EasyInstall scans all HREF's on the page for primary links

**`long_description`** EasyInstall will check any URLs contained in this argument to see if they are primary links.

A URL is considered a "primary link" if it is a link to a .tar.gz, .tgz, .zip, .egg, .egg.zip, .tar.bz2, or .exe file, or if it has an `#egg=project` or `#egg=project-version` fragment identifier attached to it. EasyInstall attempts to determine a project name and optional version number from the text of a primary link *without* downloading it. When it has found all the primary links, EasyInstall will select the best match based on requested version, platform compatibility, and other criteria.

So, if your `url` or `download_url` point either directly to a downloadable source distribution, or to HTML page(s) that have direct links to such, then EasyInstall will be able to locate downloads automatically. If you want to make Subversion checkouts available, then you should create links with either `#egg=project` or `#egg=project-version` added to the URL. You should replace `project` and `version` with the values they would have in an egg filename. (Be sure to actually generate an egg and then use the initial part of the filename, rather than trying to guess what the escaped form of the project name and version number will be.)

Note that Subversion checkout links are of lower precedence than other kinds of distributions, so EasyInstall will not select a Subversion checkout for downloading unless it has a version included in the `#egg=` suffix, and it's a higher version than EasyInstall has seen in any other links for your project.

As a result, it's a common practice to use mark checkout URLs with a version of "dev" (i.e., `#egg=projectname-dev`), so that users can do something like this:

```
easy_install --editable projectname==dev
```

in order to check out the in-development version of `projectname`.

### Managing "Continuous Releases" Using Subversion

If you expect your users to track in-development versions of your project via Subversion, there are a few additional steps you should take to ensure that things work smoothly with EasyInstall. First, you should add the following to your project's `setup.cfg` file:

```
[egg_info]
tag_build = .dev
tag_svn_revision = 1
```

This will tell `setuptools` to generate package version numbers like `1.0a1.dev-r1263`, which will be considered to be an *older* release than `1.0a1`. Thus, when you actually release `1.0a1`, the entire egg infrastructure (including `setuptools`, `pkg_resources` and EasyInstall) will know that `1.0a1` supersedes any interim snapshots from Subversion, and handle upgrades accordingly.

(Note: the project version number you specify in `setup.py` should always be the *next* version of your software, not the last released version. Alternately, you can leave out the `tag_build=.dev`, and always use the *last* release as a version number, so that your post-1.0 builds are labelled `1.0-r1263`, indicating a post-1.0 patchlevel. Most projects so far, however, seem to prefer to think of their project as being a future version still under development, rather than a past version being patched. It is of course possible for a single project to have both situations, using post-release numbering on release branches, and pre-release numbering on the trunk. But you don't have to make things this complex if you don't want to.)

Commonly, projects releasing code from Subversion will include a PyPI link to their checkout URL (as described in the previous section) with an `#egg=projectname-dev` suffix. This allows users to request EasyInstall to download `projectname==dev` in order to get the latest in-development code. Note that if your project depends on such in-progress code, you may wish to specify your `install_requires` (or other requirements) to include `==dev`, e.g.:

```
install_requires = ["OtherProject>=0.2a1.dev-r143,==dev"]
```

The above example says, "I really want at least this particular development revision number, but feel free to follow and use an `#egg=OtherProject-dev` link if you find one". This avoids the need to have actual source or binary distribution snapshots of in-development code available, just to be able to depend on the latest and greatest a project has to offer.

A final note for Subversion development: if you are using SVN revision tags as described in this section, it's a good idea to run `setup.py develop` after each Subversion checkin or update, because your project's version number will be changing, and your script wrappers need to be updated accordingly.

Also, if the project's requirements have changed, the `develop` command will take care of fetching the updated dependencies, building changed extensions, etc. Be sure to also remind any of your users who check out your project from Subversion that they need to run `setup.py develop` after every update in order to keep their checkout completely in sync.

#### Making "Official" (Non-Snapshot) Releases

When you make an official release, creating source or binary distributions, you will need to override the tag settings from `setup.cfg`, so that you don't end up registering versions like `foobar-0.7a1.dev-r34832`. This is easy to do if you are developing on the trunk and using tags or branches for your releases - just make the change to `setup.cfg` after branching or tagging the release, so the trunk will still produce development snapshots.

Alternately, if you are not branching for releases, you can override the default version options on the command line, using something like:

```
python setup.py egg_info -RDb "" sdist bdist_egg register upload
```

The first part of this command (`egg_info -RDb ""`) will override the configured tag information, before creating source and binary eggs, registering the project with PyPI, and uploading the files. Thus, these commands will use the plain version from your `setup.py`, without adding the Subversion revision number or build designation string.

Of course, if you will be doing this a lot, you may wish to create a personal alias for this operation, e.g.:

```
python setup.py alias -u release egg_info -RDb ""
```

You can then use it like this:

```
python setup.py release sdist bdist_egg register upload
```

Or of course you can create more elaborate aliases that do all of the above. See the sections below on the egg_info and alias commands for more ideas.

### Distributing Extensions compiled with Pyrex

`setuptools` includes transparent support for building Pyrex extensions, as long as you define your extensions using `setuptools.Extension`, *not* `distutils.Extension`. You must also not import anything from Pyrex in your setup script.

If you follow these rules, you can safely list `.pyx` files as the source of your `Extension` objects in the setup script. `setuptools` will detect at build time whether Pyrex is installed or not. If it is, then `setuptools` will use it. If not, then `setuptools` will silently change the `Extension` objects to refer to the `.c` counterparts of the `.pyx` files, so that the normal distutils C compilation process will occur.

Of course, for this to work, your source distributions must include the C code generated by Pyrex, as well as your original `.pyx` files. This means that you will probably want to include current `.c` files in your revision control system, rebuilding them whenever you check changes in for the `.pyx` source files. This will ensure that people tracking your project in CVS or Subversion will be able to build it even if they don't have Pyrex installed, and that your source releases will be similarly usable with or without Pyrex.

## 4.2 Command Reference

### 4.2.1 `alias` - Define shortcuts for commonly used commands

Sometimes, you need to use the same commands over and over, but you can't necessarily set them as defaults. For example, if you produce both development snapshot releases and "stable" releases of a project, you may want to put the distributions in different places, or use different `egg_info` tagging options, etc. In these cases, it doesn't make sense to set the options in a distutils configuration file, because the values of the options changed based on what you're trying to do.

Setuptools therefore allows you to define "aliases" - shortcut names for an arbitrary string of commands and options, using `setup.py alias aliasname expansion`, where aliasname is the name of the new alias, and the remainder of the command line supplies its expansion. For example, this command defines a sitewide alias called "daily", that sets various `egg_info` tagging options:

```
setup.py alias --global-config daily egg_info --tag-svn-revision \
    --tag-build=development
```

Once the alias is defined, it can then be used with other setup commands, e.g.:

```
setup.py daily bdist_egg        # generate a daily-build .egg file
setup.py daily sdist            # generate a daily-build source distro
setup.py daily sdist bdist_egg  # generate both
```

The above commands are interpreted as if the word `daily` were replaced with `egg_info --tag-svn-revision --tag-build=development`.

Note that setuptools will expand each alias *at most once* in a given command line. This serves two purposes. First, if you accidentally create an alias loop, it will have no effect; you'll instead get an error message about an unknown command. Second, it allows you to define an alias for a command, that uses that command. For example, this (project-local) alias:

```
setup.py alias bdist_egg bdist_egg rotate -k1 -m.egg
```

redefines the `bdist_egg` command so that it always runs the `rotate` command afterwards to delete all but the newest egg file. It doesn't loop indefinitely on `bdist_egg` because the alias is only expanded once when used.

You can remove a defined alias with the `--remove` (or `-r`) option, e.g.:

```
setup.py alias --global-config --remove daily
```

would delete the "daily" alias we defined above.

Aliases can be defined on a project-specific, per-user, or sitewide basis. The default is to define or remove a project-specific alias, but you can use any of the configuration file options (listed under the saveopts command, below) to determine which distutils configuration file an aliases will be added to (or removed from).

Note that if you omit the "expansion" argument to the `alias` command, you'll get output showing that alias' current definition (and what configuration file it's defined in). If you omit the alias name as well, you'll get a listing of all current aliases along with their configuration file locations.

### 4.2.2 `bdist_egg` - Create a Python Egg for the project

This command generates a Python Egg (`.egg` file) for the project. Python Eggs are the preferred binary distribution format for EasyInstall, because they are cross-platform (for "pure" packages), directly importable, and contain project metadata including scripts and information about the project's dependencies. They can be simply downloaded and added to `sys.path` directly, or they can be placed in a directory on `sys.path` and then automatically discovered by the egg runtime system.

This command runs the egg_info command (if it hasn't already run) to update the project's metadata (`.egg-info`) directory. If you have added any extra metadata files to the `.egg-info` directory, those files will be included in the new egg file's metadata directory, for use by the egg runtime system or by any applications or frameworks that use that metadata.

You won't usually need to specify any special options for this command; just use `bdist_egg` and you're done. But there are a few options that may be occasionally useful:

**`--dist-dir=DIR, -d DIR`** Set the directory where the `.egg` file will be placed. If you don't supply this, then the `--dist-dir` setting of the `bdist` command will be used, which is usually a directory named `dist` in the project directory.

**--plat-name=PLATFORM, -p PLATFORM** Set the platform name string that will be embedded in the egg's filename (assuming the egg contains C extensions). This can be used to override the distutils default platform name with something more meaningful. Keep in mind, however, that the egg runtime system expects to see eggs with distutils platform names, so it may ignore or reject eggs with non-standard platform names. Similarly, the EasyInstall program may ignore them when searching web pages for download links. However, if you are cross-compiling or doing some other unusual things, you might find a use for this option.

**--exclude-source-files** Don't include any modules' `.py` files in the egg, just compiled Python, C, and data files. (Note that this doesn't affect any `.py` files in the EGG-INFO directory or its subdirectories, since for example there may be scripts with a `.py` extension which must still be retained.) We don't recommend that you use this option except for packages that are being bundled for proprietary end-user applications, or for "embedded" scenarios where space is at an absolute premium. On the other hand, if your package is going to be installed and used in compressed form, you might as well exclude the source because Python's `traceback` module doesn't currently understand how to display zipped source code anyway, or how to deal with files that are in a different place from where their code was compiled.

There are also some options you will probably never need, but which are there because they were copied from similar `bdist` commands used as an example for creating this one. They may be useful for testing and debugging, however, which is why we kept them:

**--keep-temp, -k** Keep the contents of the `--bdist-dir` tree around after creating the `.egg` file.

**--bdist-dir=DIR, -b DIR** Set the temporary directory for creating the distribution. The entire contents of this directory are zipped to create the `.egg` file, after running various installation commands to copy the package's modules, data, and extensions here.

**--skip-build** Skip doing any "build" commands; just go straight to the install-and-compress phases.

### 4.2.3 `develop` - Deploy the project source in "Development Mode"

This command allows you to deploy your project's source for use in one or more "staging areas" where it will be available for importing. This deployment is done in such a way that changes to the project source are immediately available in the staging area(s), without needing to run a build or install step after each change.

The `develop` command works by creating an `.egg-link` file (named for the project) in the given staging area. If the staging area is Python's `site-packages` directory, it also updates an `easy-install.pth` file so that the project is on `sys.path` by default for all programs run using that Python installation.

The `develop` command also installs wrapper scripts in the staging area (or a separate directory, as specified) that will ensure the project's dependencies are available on `sys.path` before running the project's source scripts. And, it ensures that any missing project dependencies are available in the staging area, by downloading and installing them if necessary.

Last, but not least, the `develop` command invokes the `build_ext -i` command to ensure any C extensions in the project have been built and are up-to-date, and the `egg_info` command to ensure the project's metadata is updated (so that the runtime and wrappers know what the project's dependencies are). If you make any changes to the project's setup script or C extensions, you should rerun the `develop` command against all relevant staging areas to keep the project's scripts, metadata and extensions up-to-date. Most other kinds of changes to your project should not require any build operations or rerunning `develop`, but keep in mind that even minor changes to the setup script (e.g. changing an entry point definition) require you to re-run the `develop` or `test` commands to keep the distribution updated.

Here are some of the options that the `develop` command accepts. Note that they affect the project's dependencies as well as the project itself, so if you have dependencies that need to be installed and you use `--exclude-scripts` (for example), the dependencies' scripts will not be installed either! For this reason, you may want to use EasyInstall to install the project's dependencies before using the `develop` command, if you need finer control over the installation options for dependencies.

**--uninstall, -u** Un-deploy the current project. You may use the --install-dir or -d option to designate the staging area. The created .egg-link file will be removed, if present and it is still pointing to the project directory. The project directory will be removed from easy-install.pth if the staging area is Python's site-packages directory.

Note that this option currently does *not* uninstall script wrappers! You must uninstall them yourself, or overwrite them by using EasyInstall to activate a different version of the package. You can also avoid installing script wrappers in the first place, if you use the --exclude-scripts (aka -x) option when you run develop to deploy the project.

**--multi-version, -m** "Multi-version" mode. Specifying this option prevents develop from adding an easy-install.pth entry for the project(s) being deployed, and if an entry for any version of a project already exists, the entry will be removed upon successful deployment. In multi-version mode, no specific version of the package is available for importing, unless you use pkg_resources.require() to put it on sys.path, or you are running a wrapper script generated by setuptools or EasyInstall. (In which case the wrapper script calls require() for you.)

Note that if you install to a directory other than site-packages, this option is automatically in effect, because .pth files can only be used in site-packages (at least in Python 2.3 and 2.4). So, if you use the --install-dir or -d option (or they are set via configuration file(s)) your project and its dependencies will be deployed in multi- version mode.

**--install-dir=DIR, -d DIR** Set the installation directory (staging area). If this option is not directly specified on the command line or in a distutils configuration file, the distutils default installation location is used. Normally, this will be the site-packages directory, but if you are using distutils configuration files, setting things like prefix or install_lib, then those settings are taken into account when computing the default staging area.

**--script-dir=DIR, -s DIR** Set the script installation directory. If you don't supply this option (via the command line or a configuration file), but you *have* supplied an --install-dir (via command line or config file), then this option defaults to the same directory, so that the scripts will be able to find their associated package installation. Otherwise, this setting defaults to the location where the distutils would normally install scripts, taking any distutils configuration file settings into account.

**--exclude-scripts, -x** Don't deploy script wrappers. This is useful if you don't want to disturb existing versions of the scripts in the staging area.

**--always-copy, -a** Copy all needed distributions to the staging area, even if they are already present in another directory on sys.path. By default, if a requirement can be met using a distribution that is already available in a directory on sys.path, it will not be copied to the staging area.

**--egg-path=DIR** Force the generated .egg-link file to use a specified relative path to the source directory. This can be useful in circumstances where your installation directory is being shared by code running under multiple platforms (e.g. Mac and Windows) which have different absolute locations for the code under development, but the same *relative* locations with respect to the installation directory. If you use this option when installing, you must supply the same relative path when uninstalling.

In addition to the above options, the develop command also accepts all of the same options accepted by easy_install. If you've configured any easy_install settings in your setup.cfg (or other distutils config files), the develop command will use them as defaults, unless you override them in a [develop] section or on the command line.

### 4.2.4 easy_install - Find and install packages

This command runs the EasyInstall tool for you. It is exactly equivalent to running the easy_install command. All command line arguments following this command are consumed and not processed further by the distutils, so this must be the last command listed on the command line. Please see the EasyInstall documentation for the options reference and usage examples. Normally, there is no reason to use this command via the command line, as you can

just use `easy_install` directly. It's only listed here so that you know it's a distutils command, which means that you can:

- create command aliases that use it,
- create distutils extensions that invoke it as a subcommand, and
- configure options for it in your `setup.cfg` or other distutils config files.

### 4.2.5 `egg_info` - Create egg metadata and set build tags

This command performs two operations: it updates a project's `.egg-info` metadata directory (used by the `bdist_egg`, `develop`, and `test` commands), and it allows you to temporarily change a project's version string, to support "daily builds" or "snapshot" releases. It is run automatically by the `sdist`, `bdist_egg`, `develop`, `register`, and `test` commands in order to update the project's metadata, but you can also specify it explicitly in order to temporarily change the project's version string while executing other commands. (It also generates the‘‘.egg-info/SOURCES.txt‘‘ manifest file, which is used when you are building source distributions.)

In addition to writing the core egg metadata defined by `setuptools` and required by `pkg_resources`, this command can be extended to write other metadata files as well, by defining entry points in the `egg_info.writers` group. See the section on Adding new EGG-INFO Files below for more details. Note that using additional metadata writers may require you to include a `setup_requires` argument to `setup()` in order to ensure that the desired writers are available on `sys.path`.

#### Release Tagging Options

The following options can be used to modify the project's version string for all remaining commands on the setup command line. The options are processed in the order shown, so if you use more than one, the requested tags will be added in the following order:

**`--tag-build=NAME, -b NAME`** Append NAME to the project's version string. Due to the way setuptools processes "pre-release" version suffixes beginning with the letters "a" through "e" (like "alpha", "beta", and "candidate"), you will usually want to use a tag like ".build" or ".dev", as this will cause the version number to be considered *lower* than the project's default version. (If you want to make the version number *higher* than the default version, you can always leave off –tag-build and then use one or both of the following options.)

If you have a default build tag set in your `setup.cfg`, you can suppress it on the command line using `-b ""` or `--tag-build=""` as an argument to the `egg_info` command.

**`--tag-svn-revision, -r`** If the current directory is a Subversion checkout (i.e. has a `.svn` subdirectory, this appends a string of the form "-rNNNN" to the project's version string, where NNNN is the revision number of the most recent modification to the current directory, as obtained from the `svn info` command.

If the current directory is not a Subversion checkout, the command will look for a `PKG-INFO` file instead, and try to find the revision number from that, by looking for a "-rNNNN" string at the end of the version number. (This is so that building a package from a source distribution of a Subversion snapshot will produce a binary with the correct version number.)

If there is no `PKG-INFO` file, or the version number contained therein does not end with `-r` and a number, then `-r0` is used.

**`--no-svn-revision, -R`** Don't include the Subversion revision in the version number. This option is included so you can override a default setting put in `setup.cfg`.

**`--tag-date, -d`** Add a date stamp of the form "-YYYYMMDD" (e.g. "-20050528") to the project's version number.

**--no-date, -D** Don't include a date stamp in the version number. This option is included so you can override a default setting in `setup.cfg`.

(Note: Because these options modify the version number used for source and binary distributions of your project, you should first make sure that you know how the resulting version numbers will be interpreted by automated tools like EasyInstall. See the section above on Specifying Your Project's Version for an explanation of pre- and post-release tags, as well as tips on how to choose and verify a versioning scheme for your your project.)

For advanced uses, there is one other option that can be set, to change the location of the project's `.egg-info` directory. Commands that need to find the project's source directory or metadata should get it from this setting:

#### Other `egg_info` Options

**--egg-base=SOURCEDIR, -e SOURCEDIR** Specify the directory that should contain the .egg-info directory. This should normally be the root of your project's source tree (which is not necessarily the same as your project directory; some projects use a `src` or `lib` subdirectory as the source root). You should not normally need to specify this directory, as it is normally determined from the `package_dir` argument to the `setup()` function, if any. If there is no `package_dir` set, this option defaults to the current directory.

#### `egg_info` Examples

Creating a dated "nightly build" snapshot egg:

```
python setup.py egg_info --tag-date --tag-build=DEV bdist_egg
```

Creating and uploading a release with no version tags, even if some default tags are specified in `setup.cfg`:

```
python setup.py egg_info -RDb "" sdist bdist_egg register upload
```

(Notice that `egg_info` must always appear on the command line *before* any commands that you want the version changes to apply to.)

### 4.2.6 `install` - Run `easy_install` or old-style installation

The setuptools `install` command is basically a shortcut to run the `easy_install` command on the current project. However, for convenience in creating "system packages" of setuptools-based projects, you can also use this option:

**--single-version-externally-managed** This boolean option tells the `install` command to perform an "old style" installation, with the addition of an `.egg-info` directory so that the installed project will still have its metadata available and operate normally. If you use this option, you *must* also specify the `--root` or `--record` options (or both), because otherwise you will have no way to identify and remove the installed files.

This option is automatically in effect when `install` is invoked by another distutils command, so that commands like `bdist_wininst` and `bdist_rpm` will create system packages of eggs. It is also automatically in effect if you specify the `--root` option.

### 4.2.7 `install_egg_info` - Install an `.egg-info` directory in `site-packages`

Setuptools runs this command as part of `install` operations that use the `--single-version-externally-managed` options. You should not invoke it directly; it is documented here for completeness and so that distutils extensions such as system package builders can make use of it. This command has only one option:

**--install-dir=DIR, -d DIR** The parent directory where the .egg-info directory will be placed. Defaults to the same as the --install-dir option specified for the install_lib command, which is usually the system site-packages directory.

This command assumes that the egg_info command has been given valid options via the command line or setup.cfg, as it will invoke the egg_info command and use its options to locate the project's source .egg-info directory.

### 4.2.8 rotate - Delete outdated distribution files

As you develop new versions of your project, your distribution (dist) directory will gradually fill up with older source and/or binary distribution files. The rotate command lets you automatically clean these up, keeping only the N most-recently modified files matching a given pattern.

**--match=PATTERNLIST, -m PATTERNLIST** Comma-separated list of glob patterns to match. This option is *required*. The project name and -* is prepended to the supplied patterns, in order to match only distributions belonging to the current project (in case you have a shared distribution directory for multiple projects). Typically, you will use a glob pattern like .zip or .egg to match files of the specified type. Note that each supplied pattern is treated as a distinct group of files for purposes of selecting files to delete.

**--keep=COUNT, -k COUNT** Number of matching distributions to keep. For each group of files identified by a pattern specified with the --match option, delete all but the COUNT most-recently-modified files in that group. This option is *required*.

**--dist-dir=DIR, -d DIR** Directory where the distributions are. This defaults to the value of the bdist command's --dist-dir option, which will usually be the project's dist subdirectory.

**Example 1**: Delete all .tar.gz files from the distribution directory, except for the 3 most recently modified ones:

```
setup.py rotate --match=.tar.gz --keep=3
```

**Example 2**: Delete all Python 2.3 or Python 2.4 eggs from the distribution directory, except the most recently modified one for each Python version:

```
setup.py rotate --match=-py2.3*.egg,-py2.4*.egg --keep=1
```

### 4.2.9 saveopts - Save used options to a configuration file

Finding and editing distutils configuration files can be a pain, especially since you also have to translate the configuration options from command-line form to the proper configuration file format. You can avoid these hassles by using the saveopts command. Just add it to the command line to save the options you used. For example, this command builds the project using the mingw32 C compiler, then saves the –compiler setting as the default for future builds (even those run implicitly by the install command):

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

The saveopts command saves all options for every command specified on the command line to the project's local setup.cfg file, unless you use one of the configuration file options to change where the options are saved. For example, this command does the same as above, but saves the compiler setting to the site-wide (global) distutils configuration:

```
setup.py build --compiler=mingw32 saveopts -g
```

Note that it doesn't matter where you place the saveopts command on the command line; it will still save all the options specified for all commands. For example, this is another valid way to spell the last example:

```
setup.py saveopts -g build --compiler=mingw32
```

Note, however, that all of the commands specified are always run, regardless of where `saveopts` is placed on the command line.

#### Configuration File Options

Normally, settings such as options and aliases are saved to the project's local `setup.cfg` file. But you can override this and save them to the global or per-user configuration files, or to a manually-specified filename.

**`--global-config, -g`** Save settings to the global `distutils.cfg` file inside the `distutils` package directory. You must have write access to that directory to use this option. You also can't combine this option with `-u` or `-f`.

**`--user-config, -u`** Save settings to the current user's `~/.pydistutils.cfg` (POSIX) or `$HOME/pydistutils.cfg` (Windows) file. You can't combine this option with `-g` or `-f`.

**`--filename=FILENAME, -f FILENAME`** Save settings to the specified configuration file to use. You can't combine this option with `-g` or `-u`. Note that if you specify a non-standard filename, the `distutils` and `setuptools` will not use the file's contents. This option is mainly included for use in testing.

These options are used by other `setuptools` commands that modify configuration files, such as the alias and setopt commands.

### 4.2.10 `setopt` - Set a distutils or setuptools option in a config file

This command is mainly for use by scripts, but it can also be used as a quick and dirty way to change a distutils configuration option without having to remember what file the options are in and then open an editor.

**Example 1**. Set the default C compiler to `mingw32` (using long option names):

```
setup.py setopt --command=build --option=compiler --set-value=mingw32
```

**Example 2**. Remove any setting for the distutils default package installation directory (short option names):

```
setup.py setopt -c install -o install_lib -r
```

Options for the `setopt` command:

**`--command=COMMAND, -c COMMAND`** Command to set the option for. This option is required.

**`--option=OPTION, -o OPTION`** The name of the option to set. This option is required.

**`--set-value=VALUE, -s VALUE`** The value to set the option to. Not needed if `-r` or `--remove` is set.

**`--remove, -r`** Remove (unset) the option, instead of setting it.

In addition to the above options, you may use any of the configuration file options (listed under the saveopts command, above) to determine which distutils configuration file the option will be added to (or removed from).

### 4.2.11 `test` - Build package and run a unittest suite

When doing test-driven development, or running automated builds that need testing before they are deployed for downloading or use, it's often useful to be able to run a project's unit tests without actually deploying the project anywhere, even using the `develop` command. The `test` command runs a project's unit tests without actually deploying it, by temporarily putting the project's source on `sys.path`, after first running `build_ext -i` and `egg_info` to ensure that any C extensions and project metadata are up-to-date.

To use this command, your project's tests must be wrapped in a `unittest` test suite by either a function, a `TestCase` class or method, or a module or package containing `TestCase` classes. If the named suite is a module, and the module has an `additional_tests()` function, it is called and the result (which must be a `unittest.TestSuite`) is added to the tests to be run. If the named suite is a package, any submodules and subpackages are recursively added to the overall test suite. (Note: if your project specifies a `test_loader`, the rules for processing the chosen `test_suite` may differ; see the test_loader documentation for more details.)

Note that many test systems including `doctest` support wrapping their non-`unittest` tests in `TestSuite` objects. So, if you are using a test package that does not support this, we suggest you encourage its developers to implement test suite support, as this is a convenient and standard way to aggregate a collection of tests to be run under a common test harness.

By default, tests will be run in the "verbose" mode of the `unittest` package's text test runner, but you can get the "quiet" mode (just dots) if you supply the `-q` or `--quiet` option, either as a global option to the setup script (e.g. `setup.py -q test`) or as an option for the `test` command itself (e.g. `setup.py test -q`). There is one other option available:

**`--test-suite=NAME, -s NAME`** Specify the test suite (or module, class, or method) to be run (e.g. `some_module.test_suite`). The default for this option can be set by giving a `test_suite` argument to the `setup()` function, e.g.:

```
setup(
    # ...
    test_suite = "my_package.tests.test_all"
)
```

If you did not set a `test_suite` in your `setup()` call, and do not provide a `--test-suite` option, an error will occur.

### 4.2.12 `upload` - Upload source and/or egg distributions to PyPI

PyPI now supports uploading project files for redistribution; uploaded files are easily found by EasyInstall, even if you don't have download links on your project's home page.

Although Python 2.5 will support uploading all types of distributions to PyPI, setuptools only supports source distributions and eggs. (This is partly because PyPI's upload support is currently broken for various other file types.) To upload files, you must include the `upload` command *after* the `sdist` or `bdist_egg` commands on the setup command line. For example:

```
setup.py bdist_egg upload         # create an egg and upload it
setup.py sdist upload             # create a source distro and upload it
setup.py sdist bdist_egg upload   # create and upload both
```

Note that to upload files for a project, the corresponding version must already be registered with PyPI, using the distutils `register` command. It's usually a good idea to include the `register` command at the start of the command line, so that any registration problems can be found and fixed before building and uploading the distributions, e.g.:

```
setup.py register sdist bdist_egg upload
```

This will update PyPI's listing for your project's current version.

Note, by the way, that the metadata in your `setup()` call determines what will be listed in PyPI for your package. Try to fill out as much of it as possible, as it will save you a lot of trouble manually adding and updating your PyPI listings. Just put it in `setup.py` and use the `register` command to keep PyPI up to date.

The `upload` command has a few options worth noting:

**--sign, -s** Sign each uploaded file using GPG (GNU Privacy Guard). The `gpg` program must be available for execution on the system `PATH`.

**--identity=NAME, -i NAME** Specify the identity or key name for GPG to use when signing. The value of this option will be passed through the `--local-user` option of the `gpg` program.

**--show-response** Display the full response text from server; this is useful for debugging PyPI problems.

**--repository=URL, -r URL** The URL of the repository to upload to. Defaults to https://pypi.python.org/pypi (i.e., the main PyPI installation).

### 4.2.13 `upload_docs` - Upload package documentation to PyPI

PyPI now supports uploading project documentation to the dedicated URL https://pythonhosted.org/<project>/.

The `upload_docs` command will create the necessary zip file out of a documentation directory and will post to the repository.

Note that to upload the documentation of a project, the corresponding version must already be registered with PyPI, using the distutils `register` command – just like the `upload` command.

Assuming there is an `Example` project with documentation in the subdirectory `docs`, e.g.:

```
Example/
|-- example.py
|-- setup.cfg
|-- setup.py
|-- docs
|   |-- build
|   |   `-- html
|   |   |   |-- index.html
|   |   |   `-- tips_tricks.html
|   |-- conf.py
|   |-- index.txt
|   `-- tips_tricks.txt
```

You can simply pass the documentation directory path to the `upload_docs` command:

```
python setup.py upload_docs --upload-dir=docs/build/html
```

If no `--upload-dir` is given, `upload_docs` will attempt to run the `build_sphinx` command to generate uploadable documentation. For the command to become available, Sphinx must be installed in the same environment as distribute.

As with other `setuptools`-based commands, you can define useful defaults in the `setup.cfg` of your Python project, e.g.:

```
[upload_docs]
upload-dir = docs/build/html
```

The `upload_docs` command has the following options:

**--upload-dir** The directory to be uploaded to the repository.

**--show-response** Display the full response text from server; this is useful for debugging PyPI problems.

**--repository=URL, -r URL** The URL of the repository to upload to. Defaults to https://pypi.python.org/pypi (i.e., the main PyPI installation).

## 4.3 Extending and Reusing Setuptools

### 4.3.1 Creating `distutils` Extensions

It can be hard to add new commands or setup arguments to the distutils. But the `setuptools` package makes it a bit easier, by allowing you to distribute a distutils extension as a separate project, and then have projects that need the extension just refer to it in their `setup_requires` argument.

With `setuptools`, your distutils extension projects can hook in new commands and `setup()` arguments just by defining "entry points". These are mappings from command or argument names to a specification of where to import a handler from. (See the section on Dynamic Discovery of Services and Plugins above for some more background on entry points.)

#### Adding Commands

You can add new `setup` commands by defining entry points in the `distutils.commands` group. For example, if you wanted to add a `foo` command, you might add something like this to your distutils extension project's setup script:

```
setup(
    # ...
    entry_points = {
        "distutils.commands": [
            "foo = mypackage.some_module:foo",
        ],
    },
)
```

(Assuming, of course, that the `foo` class in `mypackage.some_module` is a `setuptools.Command` subclass.)

Once a project containing such entry points has been activated on `sys.path`, (e.g. by running "install" or "develop" with a site-packages installation directory) the command(s) will be available to any `setuptools`-based setup scripts. It is not necessary to use the `--command-packages` option or to monkeypatch the `distutils.command` package to install your commands; `setuptools` automatically adds a wrapper to the distutils to search for entry points in the active distributions on `sys.path`. In fact, this is how setuptools' own commands are installed: the setuptools project's setup script defines entry points for them!

#### Adding `setup()` Arguments

Sometimes, your commands may need additional arguments to the `setup()` call. You can enable this by defining entry points in the `distutils.setup_keywords` group. For example, if you wanted a `setup()` argument called `bar_baz`, you might add something like this to your distutils extension project's setup script:

```
setup(
    # ...
    entry_points = {
        "distutils.commands": [
            "foo = mypackage.some_module:foo",
        ],
        "distutils.setup_keywords": [
            "bar_baz = mypackage.some_module:validate_bar_baz",
        ],
    },
)
```

The idea here is that the entry point defines a function that will be called to validate the `setup()` argument, if it's supplied. The `Distribution` object will have the initial value of the attribute set to `None`, and the validation function will only be called if the `setup()` call sets it to a non-None value. Here's an example validation function:

```
def assert_bool(dist, attr, value):
    """Verify that value is True, False, 0, or 1"""
    if bool(value) != value:
        raise DistutilsSetupError(
            "%r must be a boolean value (got %r)" % (attr,value)
        )
```

Your function should accept three arguments: the `Distribution` object, the attribute name, and the attribute value. It should raise a `DistutilsSetupError` (from the `distutils.errors` module) if the argument is invalid. Remember, your function will only be called with non-None values, and the default value of arguments defined this way is always None. So, your commands should always be prepared for the possibility that the attribute will be `None` when they access it later.

If more than one active distribution defines an entry point for the same `setup()` argument, *all* of them will be called. This allows multiple distutils extensions to define a common argument, as long as they agree on what values of that argument are valid.

Also note that as with commands, it is not necessary to subclass or monkeypatch the distutils `Distribution` class in order to add your arguments; it is sufficient to define the entry points in your extension, as long as any setup script using your extension lists your project in its `setup_requires` argument.

### Adding new EGG-INFO Files

Some extensible applications or frameworks may want to allow third parties to develop plugins with application or framework-specific metadata included in the plugins' EGG-INFO directory, for easy access via the `pkg_resources` metadata API. The easiest way to allow this is to create a distutils extension to be used from the plugin projects' setup scripts (via `setup_requires`) that defines a new setup keyword, and then uses that data to write an EGG-INFO file when the `egg_info` command is run.

The `egg_info` command looks for extension points in an `egg_info.writers` group, and calls them to write the files. Here's a simple example of a distutils extension defining a setup argument `foo_bar`, which is a list of lines that will be written to `foo_bar.txt` in the EGG-INFO directory of any project that uses the argument:

```
setup(
    # ...
    entry_points = {
        "distutils.setup_keywords": [
            "foo_bar = setuptools.dist:assert_string_list",
        ],
        "egg_info.writers": [
            "foo_bar.txt = setuptools.command.egg_info:write_arg",
        ],
    },
)
```

This simple example makes use of two utility functions defined by setuptools for its own use: a routine to validate that a setup keyword is a sequence of strings, and another one that looks up a setup argument and writes it to a file. Here's what the writer utility looks like:

```
def write_arg(cmd, basename, filename):
    argname = os.path.splitext(basename)[0]
    value = getattr(cmd.distribution, argname, None)
    if value is not None:
```

```
        value = '\n'.join(value)+'\n'
    cmd.write_or_delete_file(argname, filename, value)
```

As you can see, `egg_info.writers` entry points must be a function taking three arguments: a `egg_info` command instance, the basename of the file to write (e.g. `foo_bar.txt`), and the actual full filename that should be written to.

In general, writer functions should honor the command object's `dry_run` setting when writing files, and use the `distutils.log` object to do any console output. The easiest way to conform to this requirement is to use the `cmd` object's `write_file()`, `delete_file()`, and `write_or_delete_file()` methods exclusively for your file operations. See those methods' docstrings for more details.

### Adding Support for Other Revision Control Systems

If you would like to create a plugin for `setuptools` to find files in other source control systems besides CVS and Subversion, you can do so by adding an entry point to the `setuptools.file_finders` group. The entry point should be a function accepting a single directory name, and should yield all the filenames within that directory (and any subdirectories thereof) that are under revision control.

For example, if you were going to create a plugin for a revision control system called "foobar", you would write a function something like this:

```
def find_files_for_foobar(dirname):
    # loop to yield paths that start with `dirname`
```

And you would register it in a setup script using something like this:

```
entry_points = {
    "setuptools.file_finders": [
        "foobar = my_foobar_module:find_files_for_foobar"
    ]
}
```

Then, anyone who wants to use your plugin can simply install it, and their local setuptools installation will be able to find the necessary files.

It is not necessary to distribute source control plugins with projects that simply use the other source control system, or to specify the plugins in `setup_requires`. When you create a source distribution with the `sdist` command, setuptools automatically records what files were found in the `SOURCES.txt` file. That way, recipients of source distributions don't need to have revision control at all. However, if someone is working on a package by checking out with that system, they will need the same plugin(s) that the original author is using.

A few important points for writing revision control file finders:

- Your finder function MUST return relative paths, created by appending to the passed-in directory name. Absolute paths are NOT allowed, nor are relative paths that reference a parent directory of the passed-in directory.
- Your finder function MUST accept an empty string as the directory name, meaning the current directory. You MUST NOT convert this to a dot; just yield relative paths. So, yielding a subdirectory named `some/dir` under the current directory should NOT be rendered as `./some/dir` or `/somewhere/some/dir`, but *always* as simply `some/dir`
- Your finder function SHOULD NOT raise any errors, and SHOULD deal gracefully with the absence of needed programs (i.e., ones belonging to the revision control system itself. It *may*, however, use `distutils.log.warn()` to inform the user of the missing program(s).

#### Subclassing `Command`

Sorry, this section isn't written yet, and neither is a lot of what's below this point, except for the change log. You might want to subscribe to changes in this page to see when new documentation is added or updated.

XXX

### 4.3.2 Reusing `setuptools` Code

#### `ez_setup`

XXX

#### `setuptools.archive_util`

XXX

#### `setuptools.sandbox`

XXX

#### `setuptools.package_index`

XXX

### 4.3.3 Mailing List and Bug Tracker

Please use the distutils-sig mailing list for questions and discussion about setuptools, and the setuptools bug tracker ONLY for issues you have confirmed via the list are actual bugs, and which you have reduced to a minimal set of steps to reproduce.

CHAPTER 5

# Easy Install

Easy Install is a python module (`easy_install`) bundled with `setuptools` that lets you automatically download, build, install, and manage Python packages.

Please share your experiences with us! If you encounter difficulty installing a package, please contact us via the distutils mailing list. (Note: please DO NOT send private email directly to the author of setuptools; it will be discarded. The mailing list is a searchable archive of previously-asked and answered questions; you should begin your research there before reporting something as a bug – and then do so via list discussion first.)

(Also, if you'd like to learn about how you can use `setuptools` to make your own packages work better with EasyInstall, or provide EasyInstall-like features without requiring your users to use EasyInstall directly, you'll probably want to check out the full setuptools documentation as well.)

**Table of Contents**



## 5.1 Using "Easy Install"

### 5.1.1 Installing "Easy Install"

Please see the setuptools PyPI page for download links and basic installation instructions for each of the supported platforms.

You will need at least Python 2.6. An `easy_install` script will be installed in the normal location for Python scripts on your platform.

Note that the instructions on the setuptools PyPI page assume that you are are installing to Python's primary `site-packages` directory. If this is not the case, you should consult the section below on Custom Installation Locations before installing. (And, on Windows, you should not use the `.exe` installer when installing to an alternate

location.)

Note that `easy_install` normally works by downloading files from the internet. If you are behind an NTLM-based firewall that prevents Python programs from accessing the net directly, you may wish to first install and use the APS proxy server, which lets you get past such firewalls in the same way that your web browser(s) do.

(Alternately, if you do not wish easy_install to actually download anything, you can restrict it from doing so with the `--allow-hosts` option; see the sections on restricting downloads with –allow-hosts and command-line options for more details.)

#### Troubleshooting

If EasyInstall/setuptools appears to install correctly, and you can run the `easy_install` command but it fails with an `ImportError`, the most likely cause is that you installed to a location other than `site-packages`, without taking any of the steps described in the Custom Installation Locations section below. Please see that section and follow the steps to make sure that your custom location will work correctly. Then re-install.

Similarly, if you can run `easy_install`, and it appears to be installing packages, but then you can't import them, the most likely issue is that you installed EasyInstall correctly but are using it to install packages to a non-standard location that hasn't been properly prepared. Again, see the section on Custom Installation Locations for more details.

#### Windows Notes

Installing setuptools will provide an `easy_install` command according to the techniques described in Executables and Launchers. If the `easy_install` command is not available after installation, that section provides details on how to configure Windows to make the commands available.

### 5.1.2 Downloading and Installing a Package

For basic use of `easy_install`, you need only supply the filename or URL of a source distribution or .egg file (Python Egg).

**Example 1**. Install a package by name, searching PyPI for the latest version, and automatically downloading, building, and installing it:

```
easy_install SQLObject
```

**Example 2**. Install or upgrade a package by name and version by finding links on a given "download page":

```
easy_install -f http://pythonpaste.org/package_index.html SQLObject
```

**Example 3**. Download a source distribution from a specified URL, automatically building and installing it:

```
easy_install http://example.com/path/to/MyPackage-1.2.3.tgz
```

**Example 4**. Install an already-downloaded .egg file:

```
easy_install /my_downloads/OtherPackage-3.2.1-py2.3.egg
```

**Example 5**. Upgrade an already-installed package to the latest version listed on PyPI:

```
easy_install --upgrade PyProtocols
```

**Example 6**. Install a source distribution that's already downloaded and extracted in the current directory (New in 0.5a9):

```
easy_install .
```

**Example 7**. (New in 0.6a1) Find a source distribution or Subversion checkout URL for a package, and extract it or check it out to `~/projects/sqlobject` (the name will always be in all-lowercase), where it can be examined or edited. (The package will not be installed, but it can easily be installed with `easy_install ~/projects/sqlobject`. See Editing and Viewing Source Packages below for more info.):

```
easy_install --editable --build-directory ~/projects SQLObject
```

**Example 7**. (New in 0.6.11) Install a distribution within your home dir:

```
easy_install --user SQLAlchemy
```

Easy Install accepts URLs, filenames, PyPI package names (i.e., `distutils` "distribution" names), and package+version specifiers. In each case, it will attempt to locate the latest available version that meets your criteria.

When downloading or processing downloaded files, Easy Install recognizes distutils source distribution files with extensions of .tgz, .tar, .tar.gz, .tar.bz2, or .zip. And of course it handles already-built .egg distributions as well as `.win32.exe` installers built using distutils.

By default, packages are installed to the running Python installation's `site-packages` directory, unless you provide the `-d` or `--install-dir` option to specify an alternative directory, or specify an alternate location using distutils configuration files. (See Configuration Files, below.)

By default, any scripts included with the package are installed to the running Python installation's standard script installation location. However, if you specify an installation directory via the command line or a config file, then the default directory for installing scripts will be the same as the package installation directory, to ensure that the script will have access to the installed package. You can override this using the `-s` or `--script-dir` option.

Installed packages are added to an `easy-install.pth` file in the install directory, so that Python will always use the most-recently-installed version of the package. If you would like to be able to select which version to use at runtime, you should use the `-m` or `--multi-version` option.

### 5.1.3 Upgrading a Package

You don't need to do anything special to upgrade a package: just install the new version, either by requesting a specific version, e.g.:

```
easy_install "SomePackage==2.0"
```

a version greater than the one you have now:

```
easy_install "SomePackage>2.0"
```

using the upgrade flag, to find the latest available version on PyPI:

```
easy_install --upgrade SomePackage
```

or by using a download page, direct download URL, or package filename:

```
easy_install -f http://example.com/downloads ExamplePackage
```

```
easy_install http://example.com/downloads/ExamplePackage-2.0-py2.4.egg
```

```
easy_install my_downloads/ExamplePackage-2.0.tgz
```

If you're using `-m` or `--multi-version` , using the `require()` function at runtime automatically selects the newest installed version of a package that meets your version criteria. So, installing a newer version is the only step needed to upgrade such packages.

If you're installing to a directory on PYTHONPATH, or a configured "site" directory (and not using `-m`), installing a package automatically replaces any previous version in the `easy-install.pth` file, so that Python will import the most-recently installed version by default. So, again, installing the newer version is the only upgrade step needed.

If you haven't suppressed script installation (using `--exclude-scripts` or `-x`), then the upgraded version's scripts will be installed, and they will be automatically patched to `require()` the corresponding version of the package, so that you can use them even if they are installed in multi-version mode.

`easy_install` never actually deletes packages (unless you're installing a package with the same name and version number as an existing package), so if you want to get rid of older versions of a package, please see Uninstalling Packages, below.

### 5.1.4 Changing the Active Version

If you've upgraded a package, but need to revert to a previously-installed version, you can do so like this:

```
easy_install PackageName==1.2.3
```

Where `1.2.3` is replaced by the exact version number you wish to switch to. If a package matching the requested name and version is not already installed in a directory on `sys.path`, it will be located via PyPI and installed.

If you'd like to switch to the latest installed version of `PackageName`, you can do so like this:

```
easy_install PackageName
```

This will activate the latest installed version. (Note: if you have set any `find_links` via distutils configuration files, those download pages will be checked for the latest available version of the package, and it will be downloaded and installed if it is newer than your current version.)

Note that changing the active version of a package will install the newly active version's scripts, unless the `--exclude-scripts` or `-x` option is specified.

### 5.1.5 Uninstalling Packages

If you have replaced a package with another version, then you can just delete the package(s) you don't need by deleting the PackageName-versioninfo.egg file or directory (found in the installation directory).

If you want to delete the currently installed version of a package (or all versions of a package), you should first run:

```
easy_install -m PackageName
```

This will ensure that Python doesn't continue to search for a package you're planning to remove. After you've done this, you can safely delete the .egg files or directories, along with any scripts you wish to remove.

### 5.1.6 Managing Scripts

Whenever you install, upgrade, or change versions of a package, EasyInstall automatically installs the scripts for the selected package version, unless you tell it not to with `-x` or `--exclude-scripts`. If any scripts in the script directory have the same name, they are overwritten.

Thus, you do not normally need to manually delete scripts for older versions of a package, unless the newer version of the package does not include a script of the same name. However, if you are completely uninstalling a package, you may wish to manually delete its scripts.

EasyInstall's default behavior means that you can normally only run scripts from one version of a package at a time. If you want to keep multiple versions of a script available, however, you can simply use the `--multi-version` or `-m` option, and rename the scripts that EasyInstall creates. This works because EasyInstall installs scripts as short

code stubs that `require()` the matching version of the package the script came from, so renaming the script has no effect on what it executes.

For example, suppose you want to use two versions of the `rst2html` tool provided by the docutils package. You might first install one version:

```
easy_install -m docutils==0.3.9
```

then rename the `rst2html.py` to `r2h_039`, and install another version:

```
easy_install -m docutils==0.3.10
```

This will create another `rst2html.py` script, this one using docutils version 0.3.10 instead of 0.3.9. You now have two scripts, each using a different version of the package. (Notice that we used `-m` for both installations, so that Python won't lock us out of using anything but the most recently-installed version of the package.)

### 5.1.7 Executables and Launchers

On Unix systems, scripts are installed with as natural files with a "#!" header and no extension and they launch under the Python version indicated in the header.

On Windows, there is no mechanism to "execute" files without extensions, so EasyInstall provides two techniques to mirror the Unix behavior. The behavior is indicated by the SETUPTOOLS_LAUNCHER environment variable, which may be "executable" (default) or "natural".

Regardless of the technique used, the script(s) will be installed to a Scripts directory (by default in the Python installation directory). It is recommended for EasyInstall that you ensure this directory is in the PATH environment variable. The easiest way to ensure the Scripts directory is in the PATH is to run `Tools\Scripts\win_add2path.py` from the Python directory (requires Python 2.6 or later).

Note that instead of changing your `PATH` to include the Python scripts directory, you can also retarget the installation location for scripts so they go on a directory that's already on the `PATH`. For more information see Command-Line Options and Configuration Files. During installation, pass command line options (such as `--script-dir`) to `ez_setup.py` to control where `easy_install.exe` will be installed.

#### Windows Executable Launcher

If the "executable" launcher is used, EasyInstall will create a '.exe' launcher of the same name beside each installed script (including `easy_install` itself). These small .exe files launch the script of the same name using the Python version indicated in the '#!' header.

This behavior is currently default. To force the use of executable launchers, set `SETUPTOOLS_LAUNCHER` to "executable".

#### Natural Script Launcher

EasyInstall also supports deferring to an external launcher such as pylauncher for launching scripts. Enable this experimental functionality by setting the `SETUPTOOLS_LAUNCHER` environment variable to "natural". EasyInstall will then install scripts as simple scripts with a .pya (or .pyw) extension appended. If these extensions are associated with the pylauncher and listed in the PATHEXT environment variable, these scripts can then be invoked simply and directly just like any other executable. This behavior may become default in a future version.

EasyInstall uses the .pya extension instead of simply the typical '.py' extension. This distinct extension is necessary to prevent Python from treating the scripts as importable modules (where name conflicts exist). Current releases of pylauncher do not yet associate with .pya files by default, but future versions should do so.

### 5.1.8 Tips & Techniques

#### Multiple Python Versions

EasyInstall installs itself under two names: `easy_install` and `easy_install-N.N`, where `N.N` is the Python version used to install it. Thus, if you install EasyInstall for both Python 3.2 and 2.7, you can use the `easy_install-3.2` or `easy_install-2.7` scripts to install packages for the respective Python version.

Setuptools also supplies easy_install as a runnable module which may be invoked using `python -m easy_install` for any Python with Setuptools installed.

#### Restricting Downloads with `--allow-hosts`

You can use the `--allow-hosts` (`-H`) option to restrict what domains EasyInstall will look for links and downloads on. `--allow-hosts=None` prevents downloading altogether. You can also use wildcards, for example to restrict downloading to hosts in your own intranet. See the section below on Command-Line Options for more details on the `--allow-hosts` option.

By default, there are no host restrictions in effect, but you can change this default by editing the appropriate configuration files and adding:

```
[easy_install]
allow_hosts = *.myintranet.example.com,*.python.org
```

The above example would then allow downloads only from hosts in the `python.org` and `myintranet.example.com` domains, unless overridden on the command line.

#### Installing on Un-networked Machines

Just copy the eggs or source packages you need to a directory on the target machine, then use the `-f` or `--find-links` option to specify that directory's location. For example:

```
easy_install -H None -f somedir SomePackage
```

will attempt to install SomePackage using only eggs and source packages found in `somedir` and disallowing all remote access. You should of course make sure you have all of SomePackage's dependencies available in somedir.

If you have another machine of the same operating system and library versions (or if the packages aren't platform-specific), you can create the directory of eggs using a command like this:

```
easy_install -zmaxd somedir SomePackage
```

This will tell EasyInstall to put zipped eggs or source packages for SomePackage and all its dependencies into `somedir`, without creating any scripts or .pth files. You can then copy the contents of `somedir` to the target machine. (`-z` means zipped eggs, `-m` means multi-version, which prevents .pth files from being used, `-a` means to copy all the eggs needed, even if they're installed elsewhere on the machine, and `-d` indicates the directory to place the eggs in.)

You can also build the eggs from local development packages that were installed with the `setup.py develop` command, by including the `-l` option, e.g.:

```
easy_install -zmaxld somedir SomePackage
```

This will use locally-available source distributions to build the eggs.

#### Packaging Others' Projects As Eggs

Need to distribute a package that isn't published in egg form? You can use EasyInstall to build eggs for a project. You'll want to use the `--zip-ok`, `--exclude-scripts`, and possibly `--no-deps` options (`-z`, `-x` and `-N`, respectively). Use `-d` or `--install-dir` to specify the location where you'd like the eggs placed. By placing them in a directory that is published to the web, you can then make the eggs available for download, either in an intranet or to the internet at large.

If someone distributes a package in the form of a single `.py` file, you can wrap it in an egg by tacking an `#egg=name-version` suffix on the file's URL. So, something like this:

```
easy_install -f "http://some.example.com/downloads/foo.py#egg=foo-1.0" foo
```

will install the package as an egg, and this:

```
easy_install -zmaxd. \
    -f "http://some.example.com/downloads/foo.py#egg=foo-1.0" foo
```

will create a `.egg` file in the current directory.

#### Creating your own Package Index

In addition to local directories and the Python Package Index, EasyInstall can find download links on most any web page whose URL is given to the `-f` (`--find-links`) option. In the simplest case, you can simply have a web page with links to eggs or Python source packages, even an automatically generated directory listing (such as the Apache web server provides).

If you are setting up an intranet site for package downloads, you may want to configure the target machines to use your download site by default, adding something like this to their configuration files:

```
[easy_install]
find_links = http://mypackages.example.com/somedir/
             http://turbogears.org/download/
             http://peak.telecommunity.com/dist/
```

As you can see, you can list multiple URLs separated by whitespace, continuing on multiple lines if necessary (as long as the subsequent lines are indented.

If you are more ambitious, you can also create an entirely custom package index or PyPI mirror. See the `--index-url` option under Command-Line Options, below, and also the section on Package Index "API".

### 5.1.9 Password-Protected Sites

If a site you want to download from is password-protected using HTTP "Basic" authentication, you can specify your credentials in the URL, like so:

```
http://some_userid:some_password@some.example.com/some_path/
```

You can do this with both index page URLs and direct download URLs. As long as any HTML pages read by easy_install use *relative* links to point to the downloads, the same user ID and password will be used to do the downloading.

### 5.1.10 Using .pypirc Credentials

In additional to supplying credentials in the URL, `easy_install` will also honor credentials if present in the .pypirc file. Teams maintaining a private repository of packages may already have defined access credentials for uploading

packages according to the distutils documentation. `easy_install` will attempt to honor those if present. Refer to the distutils documentation for Python 2.5 or later for details on the syntax.

### Controlling Build Options

EasyInstall respects standard distutils Configuration Files, so you can use them to configure build options for packages that it installs from source. For example, if you are on Windows using the MinGW compiler, you can configure the default compiler by putting something like this:

```
[build]
compiler = mingw32
```

into the appropriate distutils configuration file. In fact, since this is just normal distutils configuration, it will affect any builds using that config file, not just ones done by EasyInstall. For example, if you add those lines to `distutils.cfg` in the `distutils` package directory, it will be the default compiler for *all* packages you build. See Configuration Files below for a list of the standard configuration file locations, and links to more documentation on using distutils configuration files.

### Editing and Viewing Source Packages

Sometimes a package's source distribution contains additional documentation, examples, configuration files, etc., that are not part of its actual code. If you want to be able to examine these files, you can use the `--editable` option to EasyInstall, and EasyInstall will look for a source distribution or Subversion URL for the package, then download and extract it or check it out as a subdirectory of the `--build-directory` you specify. If you then wish to install the package after editing or configuring it, you can do so by rerunning EasyInstall with that directory as the target.

Note that using `--editable` stops EasyInstall from actually building or installing the package; it just finds, obtains, and possibly unpacks it for you. This allows you to make changes to the package if necessary, and to either install it in development mode using `setup.py develop` (if the package uses setuptools, that is), or by running `easy_install projectdir` (where `projectdir` is the subdirectory EasyInstall created for the downloaded package.

In order to use `--editable` (`-e` for short), you *must* also supply a `--build-directory` (`-b` for short). The project will be placed in a subdirectory of the build directory. The subdirectory will have the same name as the project itself, but in all-lowercase. If a file or directory of that name already exists, EasyInstall will print an error message and exit.

Also, when using `--editable`, you cannot use URLs or filenames as arguments. You *must* specify project names (and optional version requirements) so that EasyInstall knows what directory name(s) to create. If you need to force EasyInstall to use a particular URL or filename, you should specify it as a `--find-links` item (`-f` for short), and then also specify the project name, e.g.:

```
easy_install -eb ~/projects \
 -fhttp://prdownloads.sourceforge.net/ctypes/ctypes-0.9.6.tar.gz?download \
 ctypes==0.9.6
```

### Dealing with Installation Conflicts

(NOTE: As of 0.6a11, this section is obsolete; it is retained here only so that people using older versions of EasyInstall can consult it. As of version 0.6a11, installation conflicts are handled automatically without deleting the old or system-installed packages, and without ignoring the issue. Instead, eggs are automatically shifted to the front of `sys.path` using special code added to the `easy-install.pth` file. So, if you are using version 0.6a11 or better of setuptools, you do not need to worry about conflicts, and the following issues do not apply to you.)

EasyInstall installs distributions in a "managed" way, such that each distribution can be independently activated or deactivated on `sys.path`. However, packages that were not installed by EasyInstall are "unmanaged", in that they usually live all in one directory and cannot be independently activated or deactivated.

As a result, if you are using EasyInstall to upgrade an existing package, or to install a package with the same name as an existing package, EasyInstall will warn you of the conflict. (This is an improvement over `setup.py install`, becuase the `distutils` just install new packages on top of old ones, possibly combining two unrelated packages or leaving behind modules that have been deleted in the newer version of the package.)

EasyInstall will stop the installation if it detects a conflict between an existing, "unmanaged" package, and a module or package in any of the distributions you're installing. It will display a list of all of the existing files and directories that would need to be deleted for the new package to be able to function correctly. To proceed, you must manually delete these conflicting files and directories and re-run EasyInstall.

Of course, once you've replaced all of your existing "unmanaged" packages with versions managed by EasyInstall, you won't have any more conflicts to worry about!

#### Compressed Installation

EasyInstall tries to install packages in zipped form, if it can. Zipping packages can improve Python's overall import performance if you're not using the `--multi-version` option, because Python processes zipfile entries on `sys.path` much faster than it does directories.

As of version 0.5a9, EasyInstall analyzes packages to determine whether they can be safely installed as a zipfile, and then acts on its analysis. (Previous versions would not install a package as a zipfile unless you used the `--zip-ok` option.)

The current analysis approach is fairly conservative; it currenly looks for:

- Any use of the `__file__` or `__path__` variables (which should be replaced with `pkg_resources` API calls)
- Possible use of `inspect` functions that expect to manipulate source files (e.g. `inspect.getsource()`)
- Top-level modules that might be scripts used with `python -m` (Python 2.4)

If any of the above are found in the package being installed, EasyInstall will assume that the package cannot be safely run from a zipfile, and unzip it to a directory instead. You can override this analysis with the `-zip-ok` flag, which will tell EasyInstall to install the package as a zipfile anyway. Or, you can use the `--always-unzip` flag, in which case EasyInstall will always unzip, even if its analysis says the package is safe to run as a zipfile.

Normally, however, it is simplest to let EasyInstall handle the determination of whether to zip or unzip, and only specify overrides when needed to work around a problem. If you find you need to override EasyInstall's guesses, you may want to contact the package author and the EasyInstall maintainers, so that they can make appropriate changes in future versions.

(Note: If a package uses `setuptools` in its setup script, the package author has the option to declare the package safe or unsafe for zipped usage via the `zip_safe` argument to `setup()`. If the package author makes such a declaration, EasyInstall believes the package's author and does not perform its own analysis. However, your command-line option, if any, will still override the package author's choice.)

## 5.2 Reference Manual

### 5.2.1 Configuration Files

(New in 0.4a2)

You may specify default options for EasyInstall using the standard distutils configuration files, under the command heading `easy_install`. EasyInstall will look first for a `setup.cfg` file in the current directory, then a `~/.pydistutils.cfg` or `$HOME\\pydistutils.cfg` (on Unix-like OSes and Windows, respectively), and finally a `distutils.cfg` file in the `distutils` package directory. Here's a simple example:

```
[easy_install]

# set the default location to install packages
install_dir = /home/me/lib/python

# Notice that indentation can be used to continue an option
# value; this is especially useful for the "--find-links"
# option, which tells easy_install to use download links on
# these pages before consulting PyPI:
#
find_links = http://sqlobject.org/
             http://peak.telecommunity.com/dist/
```

In addition to accepting configuration for its own options under `[easy_install]`, EasyInstall also respects defaults specified for other distutils commands. For example, if you don't set an `install_dir` for `[easy_install]`, but *have* set an `install_lib` for the `[install]` command, this will become EasyInstall's default installation directory. Thus, if you are already using distutils configuration files to set default install locations, build options, etc., EasyInstall will respect your existing settings until and unless you override them explicitly in an `[easy_install]` section.

For more information, see also the current Python documentation on the use and location of distutils configuration files.

Notice that `easy_install` will use the `setup.cfg` from the current working directory only if it was triggered from `setup.py` through the `install_requires` option. The standalone command will not use that file.

### 5.2.2 Command-Line Options

**`--zip-ok, -z`** Install all packages as zip files, even if they are marked as unsafe for running as a zipfile. This can be useful when EasyInstall's analysis of a non-setuptools package is too conservative, but keep in mind that the package may not work correctly. (Changed in 0.5a9; previously this option was required in order for zipped installation to happen at all.)

**`--always-unzip, -Z`** Don't install any packages as zip files, even if the packages are marked as safe for running as a zipfile. This can be useful if a package does something unsafe, but not in a way that EasyInstall can easily detect. EasyInstall's default analysis is currently very conservative, however, so you should only use this option if you've had problems with a particular package, and *after* reporting the problem to the package's maintainer and to the EasyInstall maintainers.

(Note: the `-z`/`-Z` options only affect the installation of newly-built or downloaded packages that are not already installed in the target directory; if you want to convert an existing installed version from zipped to unzipped or vice versa, you'll need to delete the existing version first, and re-run EasyInstall.)

**`--multi-version, -m`** "Multi-version" mode. Specifying this option prevents `easy_install` from adding an `easy-install.pth` entry for the package being installed, and if an entry for any version the package already exists, it will be removed upon successful installation. In multi-version mode, no specific version of the package is available for importing, unless you use `pkg_resources.require()` to put it on `sys.path`. This can be as simple as:

```
from pkg_resources import require
require("SomePackage", "OtherPackage", "MyPackage")
```

which will put the latest installed version of the specified packages on `sys.path` for you. (For more advanced uses, like selecting specific versions and enabling optional dependencies, see the `pkg_resources` API doc.)

Changed in 0.6a10: this option is no longer silently enabled when installing to a non-PYTHONPATH, non-"site" directory. You must always explicitly use this option if you want it to be active.

**`--upgrade, -U` (New in 0.5a4)** By default, EasyInstall only searches online if a project/version requirement can't be met by distributions already installed on sys.path or the installation directory. However, if you supply the `--upgrade` or `-U` flag, EasyInstall will always check the package index and `--find-links` URLs before selecting a version to install. In this way, you can force EasyInstall to use the latest available version of any package it installs (subject to any version requirements that might exclude such later versions).

**`--install-dir=DIR, -d DIR`** Set the installation directory. It is up to you to ensure that this directory is on `sys.path` at runtime, and to use `pkg_resources.require()` to enable the installed package(s) that you need.

(New in 0.4a2) If this option is not directly specified on the command line or in a distutils configuration file, the distutils default installation location is used. Normally, this would be the `site-packages` directory, but if you are using distutils configuration files, setting things like `prefix` or `install_lib`, then those settings are taken into account when computing the default installation directory, as is the `--prefix` option.

**`--script-dir=DIR, -s DIR`** Set the script installation directory. If you don't supply this option (via the command line or a configuration file), but you *have* supplied an `--install-dir` (via command line or config file), then this option defaults to the same directory, so that the scripts will be able to find their associated package installation. Otherwise, this setting defaults to the location where the distutils would normally install scripts, taking any distutils configuration file settings into account.

**`--exclude-scripts, -x`** Don't install scripts. This is useful if you need to install multiple versions of a package, but do not want to reset the version that will be run by scripts that are already installed.

**`--user` (New in 0.6.11)** Use the the user-site-packages as specified in **PEP 370** instead of the global site-packages.

**`--always-copy, -a` (New in 0.5a4)** Copy all needed distributions to the installation directory, even if they are already present in a directory on sys.path. In older versions of EasyInstall, this was the default behavior, but now you must explicitly request it. By default, EasyInstall will no longer copy such distributions from other sys.path directories to the installation directory, unless you explicitly gave the distribution's filename on the command line.

Note that as of 0.6a10, using this option excludes "system" and "development" eggs from consideration because they can't be reliably copied. This may cause EasyInstall to choose an older version of a package than what you expected, or it may cause downloading and installation of a fresh copy of something that's already installed. You will see warning messages for any eggs that EasyInstall skips, before it falls back to an older version or attempts to download a fresh copy.

**`--find-links=URLS_OR_FILENAMES, -f URLS_OR_FILENAMES`** Scan the specified "download pages" or directories for direct links to eggs or other distributions. Any existing file or directory names or direct download URLs are immediately added to EasyInstall's search cache, and any indirect URLs (ones that don't point to eggs or other recognized archive formats) are added to a list of additional places to search for download links. As soon as EasyInstall has to go online to find a package (either because it doesn't exist locally, or because `--upgrade` or `-U` was used), the specified URLs will be downloaded and scanned for additional direct links.

Eggs and archives found by way of `--find-links` are only downloaded if they are needed to meet a requirement specified on the command line; links to unneeded packages are ignored.

If all requested packages can be found using links on the specified download pages, the Python Package Index will not be consulted unless you also specified the `--upgrade` or `-U` option.

(Note: if you want to refer to a local HTML file containing links, you must use a `file:` URL, as filenames that do not refer to a directory, egg, or archive are ignored.)

You may specify multiple URLs or file/directory names with this option, separated by whitespace. Note that on the command line, you will probably have to surround the URL list with quotes, so that it is recognized as a single option value. You can also specify URLs in a configuration file; see Configuration Files, above.

Changed in 0.6a10: previously all URLs and directories passed to this option were scanned as early as possible, but from 0.6a10 on, only directories and direct archive links are scanned immediately; URLs are not retrieved unless a package search was already going to go online due to a package not being available locally, or due to the use of the `--update` or `-U` option.

**`--no-find-links` Blocks the addition of any link.** This parameter is useful if you want to avoid adding links defined in a project easy_install is installing (whether it's a requested project or a dependency). When used, `--find-links` is ignored.

Added in Distribute 0.6.11 and Setuptools 0.7.

**`--index-url=URL, -i URL` (New in 0.4a1; default changed in 0.6c7)** Specifies the base URL of the Python Package Index. The default is https://pypi.python.org/simple if not specified. When a package is requested that is not locally available or linked from a `--find-links` download page, the package index will be searched for download pages for the needed package, and those download pages will be searched for links to download an egg or source distribution.

**`--editable, -e` (New in 0.6a1)** Only find and download source distributions for the specified projects, unpacking them to subdirectories of the specified `--build-directory`. EasyInstall will not actually build or install the requested projects or their dependencies; it will just find and extract them for you. See Editing and Viewing Source Packages above for more details.

**`--build-directory=DIR, -b DIR` (UPDATED in 0.6a1)** Set the directory used to build source packages. If a package is built from a source distribution or checkout, it will be extracted to a subdirectory of the specified directory. The subdirectory will have the same name as the extracted distribution's project, but in all-lowercase. If a file or directory of that name already exists in the given directory, a warning will be printed to the console, and the build will take place in a temporary directory instead.

This option is most useful in combination with the `--editable` option, which forces EasyInstall to *only* find and extract (but not build and install) source distributions. See Editing and Viewing Source Packages, above, for more information.

**`--verbose, -v, --quiet, -q` (New in 0.4a4)** Control the level of detail of EasyInstall's progress messages. The default detail level is "info", which prints information only about relatively time-consuming operations like running a setup script, unpacking an archive, or retrieving a URL. Using `-q` or `--quiet` drops the detail level to "warn", which will only display installation reports, warnings, and errors. Using `-v` or `--verbose` increases the detail level to include individual file-level operations, link analysis messages, and distutils messages from any setup scripts that get run. If you include the `-v` option more than once, the second and subsequent uses are passed down to any setup scripts, increasing the verbosity of their reporting as well.

**`--dry-run, -n` (New in 0.4a4)** Don't actually install the package or scripts. This option is passed down to any setup scripts run, so packages should not actually build either. This does *not* skip downloading, nor does it skip extracting source distributions to a temporary/build directory.

**`--optimize=LEVEL, -O LEVEL` (New in 0.4a4)** If you are installing from a source distribution, and are *not* using the `--zip-ok` option, this option controls the optimization level for compiling installed `.py` files to `.pyo` files. It does not affect the compilation of modules contained in `.egg` files, only those in `.egg` directories. The optimization level can be set to 0, 1, or 2; the default is 0 (unless it's set under `install` or `install_lib` in one of your distutils configuration files).

**`--record=FILENAME` (New in 0.5a4)** Write a record of all installed files to FILENAME. This is basically the same as the same option for the standard distutils "install" command, and is included for compatibility with tools that expect to pass this option to "setup.py install".

**`--site-dirs=DIRLIST, -S DIRLIST` (New in 0.6a1)** Specify one or more custom "site" directories (separated by commas). "Site" directories are directories where `.pth` files are processed, such as the main Python

site-packages directory. As of 0.6a10, EasyInstall automatically detects whether a given directory processes .pth files (or can be made to do so), so you should not normally need to use this option. It is is now only necessary if you want to override EasyInstall's judgment and force an installation directory to be treated as if it supported .pth files.

**--no-deps, -N (New in 0.6a6)** Don't install any dependencies. This is intended as a convenience for tools that wrap eggs in a platform-specific packaging system. (We don't recommend that you use it for anything else.)

**--allow-hosts=PATTERNS, -H PATTERNS (New in 0.6a6)** Restrict downloading and spidering to hosts matching the specified glob patterns. E.g. -H *.python.org restricts web access so that only packages listed and downloadable from machines in the python.org domain. The glob patterns must match the *entire* user/host/port section of the target URL(s). For example, *.python.org will NOT accept a URL like http://python.org/foo or http://www.python.org:8080/. Multiple patterns can be specified by separating them with commas. The default pattern is *, which matches anything.

In general, this option is mainly useful for blocking EasyInstall's web access altogether (e.g. -Hlocalhost), or to restrict it to an intranet or other trusted site. EasyInstall will do the best it can to satisfy dependencies given your host restrictions, but of course can fail if it can't find suitable packages. EasyInstall displays all blocked URLs, so that you can adjust your --allow-hosts setting if it is more strict than you intended. Some sites may wish to define a restrictive default setting for this option in their configuration files, and then manually override the setting on the command line as needed.

**--prefix=DIR (New in 0.6a10)** Use the specified directory as a base for computing the default installation and script directories. On Windows, the resulting default directories will be prefix\\Lib\\site-packages and prefix\\Scripts, while on other platforms the defaults will be prefix/lib/python2.X/site-packages (with the appropriate version substituted) for libraries and prefix/bin for scripts.

Note that the --prefix option only sets the *default* installation and script directories, and does not override the ones set on the command line or in a configuration file.

**--local-snapshots-ok, -l (New in 0.6c6)** Normally, EasyInstall prefers to only install *released* versions of projects, not in-development ones, because such projects may not have a currently-valid version number. So, it usually only installs them when their setup.py directory is explicitly passed on the command line.

However, if this option is used, then any in-development projects that were installed using the setup.py develop command, will be used to build eggs, effectively upgrading the "in-development" project to a snapshot release. Normally, this option is used only in conjunction with the --always-copy option to create a distributable snapshot of every egg needed to run an application.

Note that if you use this option, you must make sure that there is a valid version number (such as an SVN revision number tag) for any in-development projects that may be used, as otherwise EasyInstall may not be able to tell what version of the project is "newer" when future installations or upgrades are attempted.

### 5.2.3 Custom Installation Locations

By default, EasyInstall installs python packages into Python's main site-packages directory, and manages them using a custom .pth file in that same directory.

Very often though, a user or developer wants easy_install to install and manage python packages in an alternative location, usually for one of 3 reasons:

1. They don't have access to write to the main Python site-packages directory.
2. They want a user-specific stash of packages, that is not visible to other users.
3. They want to isolate a set of packages to a specific python application, usually to minimize the possibility of version conflicts.

Historically, there have been many approaches to achieve custom installation. The following section lists only the easiest and most relevant approaches [1].



#### Use the "–user" option

With Python 2.6 came the User scheme for installation, which means that all python distributions support an alternative install location that is specific to a user [2] [3]. The Default location for each OS is explained in the python documentation for the `site.USER_BASE` variable. This mode of installation can be turned on by specifying the `--user` option to `setup.py install` or `easy_install`. This approach serves the need to have a user-specific stash of packages.

#### Use the "–user" option and customize "PYTHONUSERBASE"

The User scheme install location can be customized by setting the `PYTHONUSERBASE` environment variable, which updates the value of `site.USER_BASE`. To isolate packages to a specific application, simply set the OS environment of that application to a specific value of `PYTHONUSERBASE`, that contains just those packages.

#### Use "virtualenv"

"virtualenv" is a 3rd-party python package that effectively "clones" a python installation, thereby creating an isolated location to install packages. The evolution of "virtualenv" started before the existence of the User installation scheme. "virtualenv" provides a version of `easy_install` that is scoped to the cloned python install and is used in the normal way. "virtualenv" does offer various features that the User installation scheme alone does not provide, e.g. the ability to hide the main python site-packages.

Please refer to the virtualenv documentation for more details.

### 5.2.4 Package Index "API"

Custom package indexes (and PyPI) must follow the following rules for EasyInstall to be able to look up and download packages:

1. Except where stated otherwise, "pages" are HTML or XHTML, and "links" refer to `href` attributes.
2. Individual project version pages' URLs must be of the form `base/projectname/version`, where `base` is the package index's base URL.
3. Omitting the `/version` part of a project page's URL (but keeping the trailing `/`) should result in a page that is either:
   (a) The single active version of that project, as though the version had been explicitly included, OR
   (b) A page with links to all of the active version pages for that project.

[1] There are older ways to achieve custom installation using various `easy_install` and `setup.py install` options, combined with `PYTHONPATH` and/or `PYTHONUSERBASE` alterations, but all of these are effectively deprecated by the User scheme brought in by PEP-370 in Python 2.6.

[2] Prior to Python2.6, Mac OS X offered a form of the User scheme. That is now subsumed into the User scheme introduced in Python 2.6.

[3] Prior to the User scheme, there was the Home scheme, which is still available, but requires more effort than the User scheme to get packages recognized.

4. Individual project version pages should contain direct links to downloadable distributions where possible. It is explicitly permitted for a project's "long_description" to include URLs, and these should be formatted as HTML links by the package index, as EasyInstall does no special processing to identify what parts of a page are index-specific and which are part of the project's supplied description.

5. Where available, MD5 information should be added to download URLs by appending a fragment identifier of the form `#md5=...`, where `...` is the 32-character hex MD5 digest. EasyInstall will verify that the downloaded file's MD5 digest matches the given value.

6. Individual project version pages should identify any "homepage" or "download" URLs using `rel="homepage"` and `rel="download"` attributes on the HTML elements linking to those URLs. Use of these attributes will cause EasyInstall to always follow the provided links, unless it can be determined by inspection that they are downloadable distributions. If the links are not to downloadable distributions, they are retrieved, and if they are HTML, they are scanned for download links. They are *not* scanned for additional "homepage" or "download" links, as these are only processed for pages that are part of a package index site.

7. The root URL of the index, if retrieved with a trailing `/`, must result in a page containing links to *all* projects' active version pages.

   (Note: This requirement is a workaround for the absence of case-insensitive `safe_name()` matching of project names in URL paths. If project names are matched in this fashion (e.g. via the PyPI server, mod_rewrite, or a similar mechanism), then it is not necessary to include this all-packages listing page.)

8. If a package index is accessed via a `file://` URL, then EasyInstall will automatically use `index.html` files, if present, when trying to read a directory with a trailing `/` on the URL.

### Backward Compatibility

Package indexes that wish to support setuptools versions prior to 0.6b4 should also follow these rules:

- Homepage and download links must be preceded with `"<th>Home Page"` or `"<th>Download URL"`, in addition to (or instead of) the `rel=""` attributes on the actual links. These marker strings do not need to be visible, or uncommented, however! For example, the following is a valid homepage link that will work with any version of setuptools:

  ```
  <li>
   <strong>Home Page:</strong>
   <!-- <th>Home Page -->
   <a rel="homepage" href="http://sqlobject.org">http://sqlobject.org</a>
  </li>
  ```

  Even though the marker string is in an HTML comment, older versions of EasyInstall will still "see" it and know that the link that follows is the project's home page URL.

- The pages described by paragraph 3(b) of the preceding section *must* contain the string `"Index of Packages</title>"` somewhere in their text. This can be inside of an HTML comment, if desired, and it can be anywhere in the page. (Note: this string MUST NOT appear on normal project pages, as described in paragraphs 2 and 3(a)!)

In addition, for compatibility with PyPI versions that do not use `#md5=` fragment IDs, EasyInstall uses the following regular expression to match PyPI's displayed MD5 info (broken onto two lines for readability):

```
<a href="([^"#]+)">([^<]+)</a>\n\s+\(<a href="[^?]+\?:action=show_md5
&amp;digest=([0-9a-f]{32})">md5</a>\)
```

## 5.3 History

**0.6c9**

- Fixed `win32.exe` support for .pth files, so unnecessary directory nesting is flattened out in the resulting egg. (There was a case-sensitivity problem that affected some distributions, notably `pywin32`.)
- Prevent `--help-commands` and other junk from showing under Python 2.5 when running `easy_install --help`.
- Fixed GUI scripts sometimes not executing on Windows
- Fixed not picking up dependency links from recursive dependencies.
- Only make `.py`, `.dll` and `.so` files executable when unpacking eggs
- Changes for Jython compatibility
- Improved error message when a requirement is also a directory name, but the specified directory is not a source package.
- Fixed `--allow-hosts` option blocking `file:` URLs
- Fixed HTTP SVN detection failing when the page title included a project name (e.g. on SourceForge-hosted SVN)
- Fix Jython script installation to handle `#!` lines better when `sys.executable` is a script.
- Removed use of deprecated `md5` module if `hashlib` is available
- Keep site directories (e.g. `site-packages`) from being included in `.pth` files.

**0.6c7**

- `ftp:` download URLs now work correctly.
- The default `--index-url` is now `https://pypi.python.org/simple`, to use the Python Package Index's new simpler (and faster!) REST API.

**0.6c6**

- EasyInstall no longer aborts the installation process if a URL it wants to retrieve can't be downloaded, unless the URL is an actual package download. Instead, it issues a warning and tries to keep going.
- Fixed distutils-style scripts originally built on Windows having their line endings doubled when installed on any platform.
- Added `--local-snapshots-ok` flag, to allow building eggs from projects installed using `setup.py develop`.
- Fixed not HTML-decoding URLs scraped from web pages

**0.6c5**

- Fixed `.dll` files on Cygwin not having executable permissions when an egg is installed unzipped.

**0.6c4**

- Added support for HTTP "Basic" authentication using `http://user:pass@host` URLs. If a password-protected page contains links to the same host (and protocol), those links will inherit the credentials used to access the original page.
- Removed all special support for Sourceforge mirrors, as Sourceforge's mirror system now works well for non-browser downloads.
- Fixed not recognizing `win32.exe` installers that included a custom bitmap.

- Fixed not allowing `os.open()` of paths outside the sandbox, even if they are opened read-only (e.g. reading `/dev/urandom` for random numbers, as is done by `os.urandom()` on some platforms).
- Fixed a problem with `.pth` testing on Windows when `sys.executable` has a space in it (e.g., the user installed Python to a `Program Files` directory).

**0.6c3**

- You can once again use "python -m easy_install" with Python 2.4 and above.
- Python 2.5 compatibility fixes added.

**0.6c2**

- Windows script wrappers now support quoted arguments and arguments containing spaces. (Patch contributed by Jim Fulton.)
- The `ez_setup.py` script now actually works when you put a setuptools `.egg` alongside it for bootstrapping an offline machine.
- A writable installation directory on `sys.path` is no longer required to download and extract a source distribution using `--editable`.
- Generated scripts now use `-x` on the `#!` line when `sys.executable` contains non-ASCII characters, to prevent deprecation warnings about an unspecified encoding when the script is run.

**0.6c1**

- EasyInstall now includes setuptools version information in the `User-Agent` string sent to websites it visits.

**0.6b4**

- Fix creating Python wrappers for non-Python scripts
- Fix `ftp://` directory listing URLs from causing a crash when used in the "Home page" or "Download URL" slots on PyPI.
- Fix `sys.path_importer_cache` not being updated when an existing zipfile or directory is deleted/overwritten.
- Fix not recognizing HTML 404 pages from package indexes.
- Allow `file://` URLs to be used as a package index. URLs that refer to directories will use an internally-generated directory listing if there is no `index.html` file in the directory.
- Allow external links in a package index to be specified using `rel="homepage"` or `rel="download"`, without needing the old PyPI-specific visible markup.
- Suppressed warning message about possibly-misspelled project name, if an egg or link for that project name has already been seen.

**0.6b3**

- Fix local `--find-links` eggs not being copied except with `--always-copy`.
- Fix sometimes not detecting local packages installed outside of "site" directories.
- Fix mysterious errors during initial `setuptools` install, caused by `ez_setup` trying to run `easy_install` twice, due to a code fallthru after deleting the egg from which it's running.

**0.6b2**

- Don't install or update a `site.py` patch when installing to a `PYTHONPATH` directory with `--multi-version`, unless an `easy-install.pth` file is already in use there.

- Construct `.pth` file paths in such a way that installing an egg whose name begins with `import` doesn't cause a syntax error.
- Fixed a bogus warning message that wasn't updated since the 0.5 versions.

**0.6b1**

- Better ambiguity management: accept `#egg` name/version even if processing what appears to be a correctly-named distutils file, and ignore `.egg` files with no `-`, since valid Python `.egg` files always have a version number (but Scheme eggs often don't).
- Support `file://` links to directories in `--find-links`, so that easy_install can build packages from local source checkouts.
- Added automatic retry for Sourceforge mirrors. The new download process is to first just try dl.sourceforge.net, then randomly select mirror IPs and remove ones that fail, until something works. The removed IPs stay removed for the remainder of the run.
- Ignore bdist_dumb distributions when looking at download URLs.

**0.6a11**

- Process `dependency_links.txt` if found in a distribution, by adding the URLs to the list for scanning.
- Use relative paths in `.pth` files when eggs are being installed to the same directory as the `.pth` file. This maximizes portability of the target directory when building applications that contain eggs.
- Added `easy_install-N.N` script(s) for convenience when using multiple Python versions.
- Added automatic handling of installation conflicts. Eggs are now shifted to the front of sys.path, in an order consistent with where they came from, making EasyInstall seamlessly co-operate with system package managers.

  The `--delete-conflicting` and `--ignore-conflicts-at-my-risk` options are now no longer necessary, and will generate warnings at the end of a run if you use them.
- Don't recursively traverse subdirectories given to `--find-links`.

**0.6a10**

- Added exhaustive testing of the install directory, including a spawn test for `.pth` file support, and directory writability/existence checks. This should virtually eliminate the need to set or configure `--site-dirs`.
- Added `--prefix` option for more do-what-I-mean-ishness in the absence of RTFM-ing. :)
- Enhanced `PYTHONPATH` support so that you don't have to put any eggs on it manually to make it work. `--multi-version` is no longer a silent default; you must explicitly use it if installing to a non-PYTHONPATH, non-"site" directory.
- Expand `$variables` used in the `--site-dirs`, `--build-directory`, `--install-dir`, and `--script-dir` options, whether on the command line or in configuration files.
- Improved SourceForge mirror processing to work faster and be less affected by transient HTML changes made by SourceForge.
- PyPI searches now use the exact spelling of requirements specified on the command line or in a project's `install_requires`. Previously, a normalized form of the name was used, which could lead to unnecessary full-index searches when a project's name had an underscore (_) in it.
- EasyInstall can now download bare `.py` files and wrap them in an egg, as long as you include an `#egg=name-version` suffix on the URL, or if the `.py` file is listed as the "Download URL" on the project's PyPI page. This allows third parties to "package" trivial Python modules just by linking to them (e.g. from within their own PyPI page or download links page).

- The `--always-copy` option now skips "system" and "development" eggs since they can't be reliably copied. Note that this may cause EasyInstall to choose an older version of a package than what you expected, or it may cause downloading and installation of a fresh version of what's already installed.
- The `--find-links` option previously scanned all supplied URLs and directories as early as possible, but now only directories and direct archive links are scanned immediately. URLs are not retrieved unless a package search was already going to go online due to a package not being available locally, or due to the use of the `--update` or `-U` option.
- Fixed the annoying `--help-commands` wart.

**0.6a9**

- Fixed `.pth` file processing picking up nested eggs (i.e. ones inside "baskets") when they weren't explicitly listed in the `.pth` file.
- If more than one URL appears to describe the exact same distribution, prefer the shortest one. This helps to avoid "table of contents" CGI URLs like the ones on effbot.org.
- Quote arguments to python.exe (including python's path) to avoid problems when Python (or a script) is installed in a directory whose name contains spaces on Windows.
- Support full roundtrip translation of eggs to and from `bdist_wininst` format. Running `bdist_wininst` on a setuptools-based package wraps the egg in an .exe that will safely install it as an egg (i.e., with metadata and entry-point wrapper scripts), and `easy_install` can turn the .exe back into an `.egg` file or directory and install it as such.

**0.6a8**

- Update for changed SourceForge mirror format
- Fixed not installing dependencies for some packages fetched via Subversion
- Fixed dependency installation with `--always-copy` not using the same dependency resolution procedure as other operations.
- Fixed not fully removing temporary directories on Windows, if a Subversion checkout left read-only files behind
- Fixed some problems building extensions when Pyrex was installed, especially with Python 2.4 and/or packages using SWIG.

**0.6a7**

- Fixed not being able to install Windows script wrappers using Python 2.3

**0.6a6**

- Added support for "traditional" PYTHONPATH-based non-root installation, and also the convenient `virtual-python.py` script, based on a contribution by Ian Bicking. The setuptools egg now contains a hacked `site` module that makes the PYTHONPATH-based approach work with .pth files, so that you can get the full EasyInstall feature set on such installations.
- Added `--no-deps` and `--allow-hosts` options.
- Improved Windows `.exe` script wrappers so that the script can have the same name as a module without confusing Python.
- Changed dependency processing so that it's breadth-first, allowing a depender's preferences to override those of a dependee, to prevent conflicts when a lower version is acceptable to the dependee, but not the depender. Also, ensure that currently installed/selected packages aren't given precedence over ones desired by a package being installed, which could cause conflict errors.

**0.6a3**

- Improved error message when trying to use old ways of running `easy_install`. Removed the ability to run via `python -m` or by running `easy_install.py`; `easy_install` is the command to run on all supported platforms.
- Improved wrapper script generation and runtime initialization so that a VersionConflict doesn't occur if you later install a competing version of a needed package as the default version of that package.
- Fixed a problem parsing version numbers in `#egg=` links.

**0.6a2**

- EasyInstall can now install "console_scripts" defined by packages that use `setuptools` and define appropriate entry points. On Windows, console scripts get an `.exe` wrapper so you can just type their name. On other platforms, the scripts are installed without a file extension.
- Using `python -m easy_install` or running `easy_install.py` is now DEPRECATED, since an `easy_install` wrapper is now available on all platforms.

**0.6a1**

- EasyInstall now does MD5 validation of downloads from PyPI, or from any link that has an "#md5=..." trailer with a 32-digit lowercase hex md5 digest.
- EasyInstall now handles symlinks in target directories by removing the link, rather than attempting to overwrite the link's destination. This makes it easier to set up an alternate Python "home" directory (as described above in the Non-Root Installation section).
- Added support for handling MacOS platform information in `.egg` filenames, based on a contribution by Kevin Dangoor. You may wish to delete and reinstall any eggs whose filename includes "darwin" and "Power_Macintosh", because the format for this platform information has changed so that minor OS X upgrades (such as 10.4.1 to 10.4.2) do not cause eggs built with a previous OS version to become obsolete.
- easy_install's dependency processing algorithms have changed. When using `--always-copy`, it now ensures that dependencies are copied too. When not using `--always-copy`, it tries to use a single resolution loop, rather than recursing.
- Fixed installing extra `.pyc` or `.pyo` files for scripts with `.py` extensions.
- Added `--site-dirs` option to allow adding custom "site" directories. Made `easy-install.pth` work in platform-specific alternate site directories (e.g. `~/Library/Python/2.x/site-packages` on Mac OS X).
- If you manually delete the current version of a package, the next run of EasyInstall against the target directory will now remove the stray entry from the `easy-install.pth` file.
- EasyInstall now recognizes URLs with a `#egg=project_name` fragment ID as pointing to the named project's source checkout. Such URLs have a lower match precedence than any other kind of distribution, so they'll only be used if they have a higher version number than any other available distribution, or if you use the `--editable` option. The `#egg` fragment can contain a version if it's formatted as `#egg=proj-ver`, where `proj` is the project name, and `ver` is the version number. You *must* use the format for these values that the `bdist_egg` command uses; i.e., all non-alphanumeric runs must be condensed to single underscore characters.
- Added the `--editable` option; see Editing and Viewing Source Packages above for more info. Also, slightly changed the behavior of the `--build-directory` option.
- Fixed the setup script sandbox facility not recognizing certain paths as valid on case-insensitive platforms.

**0.5a12**

- Fix `python -m easy_install` not working due to setuptools being installed as a zipfile. Update safety scanner to check for modules that might be used as `python -m` scripts.

- Misc. fixes for win32.exe support, including changes to support Python 2.4's changed `bdist_wininst` format.

**0.5a10**

- Put the `easy_install` module back in as a module, as it's needed for `python -m` to run it!
- Allow `--find-links/-f` to accept local directories or filenames as well as URLs.

**0.5a9**

- EasyInstall now automatically detects when an "unmanaged" package or module is going to be on `sys.path` ahead of a package you're installing, thereby preventing the newer version from being imported. By default, it will abort installation to alert you of the problem, but there are also new options (`--delete-conflicting` and `--ignore-conflicts-at-my-risk`) available to change the default behavior. (Note: this new feature doesn't take effect for egg files that were built with older `setuptools` versions, because they lack the new metadata file required to implement it.)
- The `easy_install` distutils command now uses `DistutilsError` as its base error type for errors that should just issue a message to stderr and exit the program without a traceback.
- EasyInstall can now be given a path to a directory containing a setup script, and it will attempt to build and install the package there.
- EasyInstall now performs a safety analysis on module contents to determine whether a package is likely to run in zipped form, and displays information about what modules may be doing introspection that would break when running as a zipfile.
- Added the `--always-unzip/-Z` option, to force unzipping of packages that would ordinarily be considered safe to unzip, and changed the meaning of `--zip-ok/-z` to "always leave everything zipped".

**0.5a8**

- There is now a separate documentation page for setuptools; revision history that's not specific to EasyInstall has been moved to that page.

**0.5a5**

- Made `easy_install` a standard `setuptools` command, moving it from the `easy_install` module to `setuptools.command.easy_install`. Note that if you were importing or extending it, you must now change your imports accordingly. `easy_install.py` is still installed as a script, but not as a module.

**0.5a4**

- Added `--always-copy/-a` option to always copy needed packages to the installation directory, even if they're already present elsewhere on sys.path. (In previous versions, this was the default behavior, but now you must request it.)
- Added `--upgrade/-U` option to force checking PyPI for latest available version(s) of all packages requested by name and version, even if a matching version is available locally.
- Added automatic installation of dependencies declared by a distribution being installed. These dependencies must be listed in the distribution's `EGG-INFO` directory, so the distribution has to have declared its dependencies by using setuptools. If a package has requirements it didn't declare, you'll still have to deal with them yourself. (E.g., by asking EasyInstall to find and install them.)
- Added the `--record` option to `easy_install` for the benefit of tools that run `setup.py install --record=filename` on behalf of another packaging system.)

**0.5a3**

- Fixed not setting script permissions to allow execution.

- Improved sandboxing so that setup scripts that want a temporary directory (e.g. pychecker) can still run in the sandbox.

**0.5a2**

- Fix stupid stupid refactoring-at-the-last-minute typos. :(

**0.5a1**

- Added support for converting `.win32.exe` installers to eggs on the fly. EasyInstall will now recognize such files by name and install them.
- Fixed a problem with picking the "best" version to install (versions were being sorted as strings, rather than as parsed values)

**0.4a4**

- Added support for the distutils "verbose/quiet" and "dry-run" options, as well as the "optimize" flag.
- Support downloading packages that were uploaded to PyPI (by scanning all links on package pages, not just the homepage/download links).

**0.4a3**

- Add progress messages to the search/download process so that you can tell what URLs it's reading to find download links. (Hopefully, this will help people report out-of-date and broken links to package authors, and to tell when they've asked for a package that doesn't exist.)

**0.4a2**

- Added support for installing scripts
- Added support for setting options via distutils configuration files, and using distutils' default options as a basis for EasyInstall's defaults.
- Renamed `--scan-url/-s` to `--find-links/-f` to free up `-s` for the script installation directory option.
- Use `urllib2` instead of `urllib`, to allow use of `https:` URLs if Python includes SSL support.

**0.4a1**

- Added `--scan-url` and `--index-url` options, to scan download pages and search PyPI for needed packages.

**0.3a4**

- Restrict `--build-directory=DIR/-b DIR` option to only be used with single URL installs, to avoid running the wrong setup.py.

**0.3a3**

- Added `--build-directory=DIR/-b DIR` option.
- Added "installation report" that explains how to use 'require()' when doing a multiversion install or alternate installation directory.
- Added SourceForge mirror auto-select (Contributed by Ian Bicking)
- Added "sandboxing" that stops a setup script from running if it attempts to write to the filesystem outside of the build area
- Added more workarounds for packages with quirky `install_data` hacks

**0.3a2**

- Added subversion download support for `svn:` and `svn+` URLs, as well as automatic recognition of HTTP subversion URLs (Contributed by Ian Bicking)
- Misc. bug fixes

**0.3a1**

- Initial release.

## 5.4 Future Plans

- Additional utilities to list/remove/verify packages
- Signature checking? SSL? Ability to suppress PyPI search?
- Display byte progress meter when downloading distributions and long pages?
- Redirect stdout/stderr to log during run_setup?

CHAPTER 6

# Package Discovery and Resource Access using `pkg_resources`

The `pkg_resources` module distributed with `setuptools` provides an API for Python libraries to access their resource files, and for extensible applications and frameworks to automatically discover plugins. It also provides runtime support for using C extensions that are inside zipfile-format eggs, support for merging packages that have separately-distributed modules or subpackages, and APIs for managing Python's current "working set" of active packages.

**Table of Contents**



## 6.1 Overview

The `pkg_resources` module provides runtime facilities for finding, introspecting, activating and using installed Python distributions. Some of the more advanced features (notably the support for parallel installation of multiple versions) rely specifically on the "egg" format (either as a zip archive or subdirectory), while others (such as plugin discovery) will work correctly so long as "egg-info" metadata directories are available for relevant distributions.

Eggs are a distribution format for Python modules, similar in concept to Java's "jars" or Ruby's "gems", or the "wheel" format defined in PEP 427. However, unlike a pure distribution format, eggs can also be installed and added directly to `sys.path` as an import location. When installed in this way, eggs are *discoverable*, meaning that they carry metadata that unambiguously identifies their contents and dependencies. This means that an installed egg can be *automatically* found and added to `sys.path` in response to simple requests of the form, "get me everything I need to

use docutils' PDF support". This feature allows mutually conflicting versions of a distribution to co-exist in the same Python installation, with individual applications activating the desired version at runtime by manipulating the contents of `sys.path` (this differs from the virtual environment approach, which involves creating isolated environments for each application).

The following terms are needed in order to explain the capabilities offered by this module:

**project** A library, framework, script, plugin, application, or collection of data or other resources, or some combination thereof. Projects are assumed to have "relatively unique" names, e.g. names registered with PyPI.

**release** A snapshot of a project at a particular point in time, denoted by a version identifier.

**distribution** A file or files that represent a particular release.

**importable distribution** A file or directory that, if placed on `sys.path`, allows Python to import any modules contained within it.

**pluggable distribution** An importable distribution whose filename unambiguously identifies its release (i.e. project and version), and whose contents unambiguously specify what releases of other projects will satisfy its runtime requirements.

**extra** An "extra" is an optional feature of a release, that may impose additional runtime requirements. For example, if docutils PDF support required a PDF support library to be present, docutils could define its PDF support as an "extra", and list what other project releases need to be available in order to provide it.

**environment** A collection of distributions potentially available for importing, but not necessarily active. More than one distribution (i.e. release version) for a given project may be present in an environment.

**working set** A collection of distributions actually available for importing, as on `sys.path`. At most one distribution (release version) of a given project may be present in a working set, as otherwise there would be ambiguity as to what to import.

**eggs** Eggs are pluggable distributions in one of the three formats currently supported by `pkg_resources`. There are built eggs, development eggs, and egg links. Built eggs are directories or zipfiles whose name ends with `.egg` and follows the egg naming conventions, and contain an `EGG-INFO` subdirectory (zipped or otherwise). Development eggs are normal directories of Python code with one or more `ProjectName.egg-info` subdirectories. The development egg format is also used to provide a default version of a distribution that is available to software that doesn't use `pkg_resources` to request specific versions. Egg links are `*.egg-link` files that contain the name of a built or development egg, to support symbolic linking on platforms that do not have native symbolic links (or where the symbolic link support is limited).

(For more information about these terms and concepts, see also this architectural overview of `pkg_resources` and Python Eggs in general.)

## 6.2 API Reference

### 6.2.1 Namespace Package Support

A namespace package is a package that only contains other packages and modules, with no direct contents of its own. Such packages can be split across multiple, separately-packaged distributions. They are normally used to split up large packages produced by a single organization, such as in the `zope` namespace package for Zope Corporation packages, and the `peak` namespace package for the Python Enterprise Application Kit.

To create a namespace package, you list it in the `namespace_packages` argument to `setup()`, in your project's `setup.py`. (See the setuptools documentation on namespace packages for more information on this.) Also, you must add a `declare_namespace()` call in the package's `__init__.py` file(s):

**declare_namespace(name)** Declare that the dotted package name *name* is a "namespace package" whose contained packages and modules may be spread across multiple distributions. The named package's `__path__` will be extended to include the corresponding package in all distributions on `sys.path` that contain a package of that name. (More precisely, if an importer's `find_module(name)` returns a loader, then it will also be searched for the package's contents.) Whenever a Distribution's `activate()` method is invoked, it checks for the presence of namespace packages and updates their `__path__` contents accordingly.

Applications that manipulate namespace packages or directly alter `sys.path` at runtime may also need to use this API function:

**fixup_namespace_packages(path_item)** Declare that *path_item* is a newly added item on `sys.path` that may need to be used to update existing namespace packages. Ordinarily, this is called for you when an egg is automatically added to `sys.path`, but if your application modifies `sys.path` to include locations that may contain portions of a namespace package, you will need to call this function to ensure they are added to the existing namespace packages.

Although by default `pkg_resources` only supports namespace packages for filesystem and zip importers, you can extend its support to other "importers" compatible with PEP 302 using the `register_namespace_handler()` function. See the section below on Supporting Custom Importers for details.

### 6.2.2 `WorkingSet` Objects

The `WorkingSet` class provides access to a collection of "active" distributions. In general, there is only one meaningful `WorkingSet` instance: the one that represents the distributions that are currently active on `sys.path`. This global instance is available under the name `working_set` in the `pkg_resources` module. However, specialized tools may wish to manipulate working sets that don't correspond to `sys.path`, and therefore may wish to create other `WorkingSet` instances.

It's important to note that the global `working_set` object is initialized from `sys.path` when `pkg_resources` is first imported, but is only updated if you do all future `sys.path` manipulation via `pkg_resources` APIs. If you manually modify `sys.path`, you must invoke the appropriate methods on the `working_set` instance to keep it in sync. Unfortunately, Python does not provide any way to detect arbitrary changes to a list object like `sys.path`, so `pkg_resources` cannot automatically update the `working_set` based on changes to `sys.path`.

**WorkingSet(entries=None)** Create a `WorkingSet` from an iterable of path entries. If *entries* is not supplied, it defaults to the value of `sys.path` at the time the constructor is called.

Note that you will not normally construct `WorkingSet` instances yourself, but instead you will implicitly or explicitly use the global `working_set` instance. For the most part, the `pkg_resources` API is designed so that the `working_set` is used by default, such that you don't have to explicitly refer to it most of the time.

All distributions available directly on `sys.path` will be activated automatically when `pkg_resources` is imported. This behaviour can cause version conflicts for applications which require non-default versions of those distributions. To handle this situation, `pkg_resources` checks for a `__requires__` attribute in the `__main__` module when initializing the default working set, and uses this to ensure a suitable version of each affected distribution is activated. For example:

```
__requires__ = ["CherryPy < 3"] # Must be set before pkg_resources import
import pkg_resources
```

#### Basic `WorkingSet` Methods

The following methods of `WorkingSet` objects are also available as module- level functions in `pkg_resources` that apply to the default `working_set` instance. Thus, you can use e.g. `pkg_resources.require()` as an abbreviation for `pkg_resources.working_set.require()`:

**require(*requirements)** Ensure that distributions matching *requirements* are activated

*requirements* must be a string or a (possibly-nested) sequence thereof, specifying the distributions and versions required. The return value is a sequence of the distributions that needed to be activated to fulfill the requirements; all relevant distributions are included, even if they were already activated in this working set.

For the syntax of requirement specifiers, see the section below on Requirements Parsing.

In general, it should not be necessary for you to call this method directly. It's intended more for use in quick-and-dirty scripting and interactive interpreter hacking than for production use. If you're creating an actual library or application, it's strongly recommended that you create a "setup.py" script using `setuptools`, and declare all your requirements there. That way, tools like EasyInstall can automatically detect what requirements your package has, and deal with them accordingly.

Note that calling `require('SomePackage')` will not install `SomePackage` if it isn't already present. If you need to do this, you should use the `resolve()` method instead, which allows you to pass an `installer` callback that will be invoked when a needed distribution can't be found on the local machine. You can then have this callback display a dialog, automatically download the needed distribution, or whatever else is appropriate for your application. See the documentation below on the `resolve()` method for more information, and also on the `obtain()` method of `Environment` objects.

**run_script(requires, script_name)** Locate distribution specified by *requires* and run its *script_name* script. *requires* must be a string containing a requirement specifier. (See Requirements Parsing below for the syntax.)

The script, if found, will be executed in *the caller's globals*. That's because this method is intended to be called from wrapper scripts that act as a proxy for the "real" scripts in a distribution. A wrapper script usually doesn't need to do anything but invoke this function with the correct arguments.

If you need more control over the script execution environment, you probably want to use the `run_script()` method of a `Distribution` object's Metadata API instead.

**iter_entry_points(group, name=None)** Yield entry point objects from *group* matching *name*

If *name* is None, yields all entry points in *group* from all distributions in the working set, otherwise only ones matching both *group* and *name* are yielded. Entry points are yielded from the active distributions in the order that the distributions appear in the working set. (For the global `working_set`, this should be the same as the order that they are listed in `sys.path`.) Note that within the entry points advertised by an individual distribution, there is no particular ordering.

Please see the section below on Entry Points for more information.

### `WorkingSet` Methods and Attributes

These methods are used to query or manipulate the contents of a specific working set, so they must be explicitly invoked on a particular `WorkingSet` instance:

**add_entry(entry)** Add a path item to the `entries`, finding any distributions on it. You should use this when you add additional items to `sys.path` and you want the global `working_set` to reflect the change. This method is also called by the `WorkingSet()` constructor during initialization.

This method uses `find_distributions(entry,True)` to find distributions corresponding to the path entry, and then `add()` them. *entry* is always appended to the `entries` attribute, even if it is already present, however. (This is because `sys.path` can contain the same value more than once, and the `entries` attribute should be able to reflect this.)

**__contains__(dist)** True if *dist* is active in this `WorkingSet`. Note that only one distribution for a given project can be active in a given `WorkingSet`.

**__iter__()** Yield distributions for non-duplicate projects in the working set. The yield order is the order in which the items' path entries were added to the working set.

**find(req)** Find a distribution matching *req* (a `Requirement` instance). If there is an active distribution for the requested project, this returns it, as long as it meets the version requirement specified by *req*. But, if there is an active distribution for the project and it does *not* meet the *req* requirement, `VersionConflict` is raised. If there is no active distribution for the requested project, `None` is returned.

**resolve(requirements, env=None, installer=None)** List all distributions needed to (recursively) meet *requirements*

*requirements* must be a sequence of `Requirement` objects. *env*, if supplied, should be an `Environment` instance. If not supplied, an `Environment` is created from the working set's `entries`. *installer*, if supplied, will be invoked with each requirement that cannot be met by an already-installed distribution; it should return a `Distribution` or `None`. (See the `obtain()` method of Environment Objects, below, for more information on the *installer* argument.)

**add(dist, entry=None)** Add *dist* to working set, associated with *entry*

If *entry* is unspecified, it defaults to `dist.location`. On exit from this routine, *entry* is added to the end of the working set's `.entries` (if it wasn't already present).

*dist* is only added to the working set if it's for a project that doesn't already have a distribution active in the set. If it's successfully added, any callbacks registered with the `subscribe()` method will be called. (See Receiving Change Notifications, below.)

Note: `add()` is automatically called for you by the `require()` method, so you don't normally need to use this method directly.

**entries** This attribute represents a "shadow" `sys.path`, primarily useful for debugging. If you are experiencing import problems, you should check the global `working_set` object's `entries` against `sys.path`, to ensure that they match. If they do not, then some part of your program is manipulating `sys.path` without updating the `working_set` accordingly. IMPORTANT NOTE: do not directly manipulate this attribute! Setting it equal to `sys.path` will not fix your problem, any more than putting black tape over an "engine warning" light will fix your car! If this attribute is out of sync with `sys.path`, it's merely an *indicator* of the problem, not the cause of it.

### Receiving Change Notifications

Extensible applications and frameworks may need to receive notification when a new distribution (such as a plug-in component) has been added to a working set. This is what the `subscribe()` method and `add_activation_listener()` function are for.

**subscribe(callback)** Invoke `callback(distribution)` once for each active distribution that is in the set now, or gets added later. Because the callback is invoked for already-active distributions, you do not need to loop over the working set yourself to deal with the existing items; just register the callback and be prepared for the fact that it will be called immediately by this method.

Note that callbacks *must not* allow exceptions to propagate, or they will interfere with the operation of other callbacks and possibly result in an inconsistent working set state. Callbacks should use a try/except block to ignore, log, or otherwise process any errors, especially since the code that caused the callback to be invoked is unlikely to be able to handle the errors any better than the callback itself.

`pkg_resources.add_activation_listener()` is an alternate spelling of `pkg_resources.working_set.subscribe()`.

### Locating Plugins

Extensible applications will sometimes have a "plugin directory" or a set of plugin directories, from which they want to load entry points or other metadata. The `find_plugins()` method allows you to do this, by scanning an environment for the newest version of each project that can be safely loaded without conflicts or missing requirements.

**`find_plugins(plugin_env, full_env=None, fallback=True)`** Scan *plugin_env* and identify which distributions could be added to this working set without version conflicts or missing requirements.

Example usage:

```
distributions, errors = working_set.find_plugins(
    Environment(plugin_dirlist)
)
map(working_set.add, distributions)  # add plugins+libs to sys.path
print "Couldn't load", errors        # display errors
```

The *plugin_env* should be an `Environment` instance that contains only distributions that are in the project's "plugin directory" or directories. The *full_env*, if supplied, should be an `Environment` instance that contains all currently-available distributions.

If *full_env* is not supplied, one is created automatically from the `WorkingSet` this method is called on, which will typically mean that every directory on `sys.path` will be scanned for distributions.

This method returns a 2-tuple: (*distributions*, *error_info*), where *distributions* is a list of the distributions found in *plugin_env* that were loadable, along with any other distributions that are needed to resolve their dependencies. *error_info* is a dictionary mapping unloadable plugin distributions to an exception instance describing the error that occurred. Usually this will be a `DistributionNotFound` or `VersionConflict` instance.

Most applications will use this method mainly on the master `working_set` instance in `pkg_resources`, and then immediately add the returned distributions to the working set so that they are available on sys.path. This will make it possible to find any entry points, and allow any other metadata tracking and hooks to be activated.

The resolution algorithm used by `find_plugins()` is as follows. First, the project names of the distributions present in *plugin_env* are sorted. Then, each project's eggs are tried in descending version order (i.e., newest version first).

An attempt is made to resolve each egg's dependencies. If the attempt is successful, the egg and its dependencies are added to the output list and to a temporary copy of the working set. The resolution process continues with the next project name, and no older eggs for that project are tried.

If the resolution attempt fails, however, the error is added to the error dictionary. If the *fallback* flag is true, the next older version of the plugin is tried, until a working version is found. If false, the resolution process continues with the next plugin project name.

Some applications may have stricter fallback requirements than others. For example, an application that has a database schema or persistent objects may not be able to safely downgrade a version of a package. Others may want to ensure that a new plugin configuration is either 100% good or else revert to a known-good configuration. (That is, they may wish to revert to a known configuration if the *error_info* return value is non-empty.)

Note that this algorithm gives precedence to satisfying the dependencies of alphabetically prior project names in case of version conflicts. If two projects named "AaronsPlugin" and "ZekesPlugin" both need different versions of "TomsLibrary", then "AaronsPlugin" will win and "ZekesPlugin" will be disabled due to version conflict.

## 6.2.3 `Environment` Objects

An "environment" is a collection of `Distribution` objects, usually ones that are present and potentially importable on the current platform. `Environment` objects are used by `pkg_resources` to index available distributions during dependency resolution.

**Environment(search_path=None, platform=get_supported_platform(), python=PY_MAJOR)**
Create an environment snapshot by scanning *search_path* for distributions compatible with *platform* and *python*. *search_path* should be a sequence of strings such as might be used on `sys.path`. If a *search_path* isn't supplied, `sys.path` is used.

*platform* is an optional string specifying the name of the platform that platform-specific distributions must be compatible with. If unspecified, it defaults to the current platform. *python* is an optional string naming the desired version of Python (e.g. `'2.4'`); it defaults to the currently-running version.

You may explicitly set *platform* (and/or *python*) to `None` if you wish to include *all* distributions, not just those compatible with the running platform or Python version.

Note that *search_path* is scanned immediately for distributions, and the resulting `Environment` is a snapshot of the found distributions. It is not automatically updated if the system's state changes due to e.g. installation or removal of distributions.

**__getitem__(project_name)** Returns a list of distributions for the given project name, ordered from newest to oldest version. (And highest to lowest format precedence for distributions that contain the same version of the project.) If there are no distributions for the project, returns an empty list.

**__iter__()** Yield the unique project names of the distributions in this environment. The yielded names are always in lower case.

**add(dist)** Add *dist* to the environment if it matches the platform and python version specified at creation time, and only if the distribution hasn't already been added. (i.e., adding the same distribution more than once is a no-op.)

**remove(dist)** Remove *dist* from the environment.

**can_add(dist)** Is distribution *dist* acceptable for this environment? If it's not compatible with the `platform` and `python` version values specified when the environment was created, a false value is returned.

**__add__(dist_or_env) (+ operator)** Add a distribution or environment to an `Environment` instance, returning a *new* environment object that contains all the distributions previously contained by both. The new environment will have a `platform` and `python` of `None`, meaning that it will not reject any distributions from being added to it; it will simply accept whatever is added. If you want the added items to be filtered for platform and Python version, or you want to add them to the *same* environment instance, you should use in-place addition (+=) instead.

**__iadd__(dist_or_env) (+= operator)** Add a distribution or environment to an `Environment` instance *in-place*, updating the existing instance and returning it. The `platform` and `python` filter attributes take effect, so distributions in the source that do not have a suitable platform string or Python version are silently ignored.

**best_match(req, working_set, installer=None)** Find distribution best matching *req* and usable on *working_set*

This calls the `find(req)` method of the *working_set* to see if a suitable distribution is already active. (This may raise `VersionConflict` if an unsuitable version of the project is already active in the specified *working_set*.) If a suitable distribution isn't active, this method returns the newest distribution in the environment that meets the `Requirement` in *req*. If no suitable distribution is found, and *installer* is supplied, then the result of calling the environment's `obtain(req, installer)` method will be returned.

**obtain(requirement, installer=None)** Obtain a distro that matches requirement (e.g. via download). In the base `Environment` class, this routine just returns `installer(requirement)`, unless *installer* is None, in which case None is returned instead. This method is a hook that allows subclasses to attempt other ways of obtaining a distribution before falling back to the *installer* argument.

**scan(search_path=None)** Scan *search_path* for distributions usable on *platform*

Any distributions found are added to the environment. *search_path* should be a sequence of strings such as might be used on `sys.path`. If not supplied, `sys.path` is used. Only distributions conforming

to the platform/python version defined at initialization are added. This method is a shortcut for using the `find_distributions()` function to find the distributions from each item in *search_path*, and then calling `add()` to add each one to the environment.

### 6.2.4 `Requirement` Objects

`Requirement` objects express what versions of a project are suitable for some purpose. These objects (or their string form) are used by various `pkg_resources` APIs in order to find distributions that a script or distribution needs.

#### Requirements Parsing

**`parse_requirements(s)`** Yield `Requirement` objects for a string or iterable of lines. Each requirement must start on a new line. See below for syntax.

**`Requirement.parse(s)`** Create a `Requirement` object from a string or iterable of lines. A `ValueError` is raised if the string or lines do not contain a valid requirement specifier, or if they contain more than one specifier. (To parse multiple specifiers from a string or iterable of strings, use `parse_requirements()` instead.)

The syntax of a requirement specifier can be defined in EBNF as follows:

```
requirement  ::= project_name versionspec? extras?
versionspec  ::= comparison version (',' comparison version)*
comparison   ::= '<' | '<=' | '!=' | '==' | '>=' | '>'
extras       ::= '[' extralist? ']'
extralist    ::= identifier (',' identifier)*
project_name ::= identifier
identifier   ::= [-A-Za-z0-9_]+
version      ::= [-A-Za-z0-9_.]+
```

Tokens can be separated by whitespace, and a requirement can be continued over multiple lines using a backslash (\\). Line-end comments (using #) are also allowed.

Some examples of valid requirement specifiers:

```
FooProject >= 1.2
Fizzy [foo, bar]
PickyThing<1.6,>1.9,!=1.9.6,<2.0a0,==2.4c1
SomethingWhoseVersionIDontCareAbout
```

The project name is the only required portion of a requirement string, and if it's the only thing supplied, the requirement will accept any version of that project.

The "extras" in a requirement are used to request optional features of a project, that may require additional project distributions in order to function. For example, if the hypothetical "Report-O-Rama" project offered optional PDF support, it might require an additional library in order to provide that support. Thus, a project needing Report-O-Rama's PDF features could use a requirement of `Report-O-Rama[PDF]` to request installation or activation of both Report-O-Rama and any libraries it needs in order to provide PDF support. For example, you could use:

```
easy_install.py Report-O-Rama[PDF]
```

To install the necessary packages using the EasyInstall program, or call `pkg_resources.require('Report-O-Rama[PDF]')` to add the necessary distributions to sys.path at runtime.

#### `Requirement` Methods and Attributes

**`__contains__(dist_or_version)`** Return true if *dist_or_version* fits the criteria for this requirement. If *dist_or_version* is a `Distribution` object, its project name must match the requirement's project name, and its version must meet the requirement's version criteria. If *dist_or_version* is a string, it is parsed using the `parse_version()` utility function. Otherwise, it is assumed to be an already-parsed version.

The `Requirement` object's version specifiers (`.specs`) are internally sorted into ascending version order, and used to establish what ranges of versions are acceptable. Adjacent redundant conditions are effectively consolidated (e.g. `">1, >2"` produces the same results as `">1"`, and `"<2,<3"` produces the same results as`""<3"`). ``"!="`` versions are excised from the ranges they fall within. The version being tested for acceptability is then checked for membership in the resulting ranges. (Note that providing conflicting conditions for the same version (e.g. `"<2,>=2"` or `"==2,!=2"`) is meaningless and may therefore produce bizarre results when compared with actual version number(s).)

**`__eq__(other_requirement)`** A requirement compares equal to another requirement if they have case-insensitively equal project names, version specifiers, and "extras". (The order that extras and version specifiers are in is also ignored.) Equal requirements also have equal hashes, so that requirements can be used in sets or as dictionary keys.

**`__str__()`** The string form of a `Requirement` is a string that, if passed to `Requirement.parse()`, would return an equal `Requirement` object.

**`project_name`** The name of the required project

**`key`** An all-lowercase version of the `project_name`, useful for comparison or indexing.

**`extras`** A tuple of names of "extras" that this requirement calls for. (These will be all-lowercase and normalized using the `safe_extra()` parsing utility function, so they may not exactly equal the extras the requirement was created with.)

**`specs`** A list of `(op,version)` tuples, sorted in ascending parsed-version order. The *op* in each tuple is a comparison operator, represented as a string. The *version* is the (unparsed) version number. The relative order of tuples containing the same version numbers is undefined, since having more than one operator for a given version is either redundant or self-contradictory.

### 6.2.5 Entry Points

Entry points are a simple way for distributions to "advertise" Python objects (such as functions or classes) for use by other distributions. Extensible applications and frameworks can search for entry points with a particular name or group, either from a specific distribution or from all active distributions on sys.path, and then inspect or load the advertised objects at will.

Entry points belong to "groups" which are named with a dotted name similar to a Python package or module name. For example, the `setuptools` package uses an entry point named `distutils.commands` in order to find commands defined by distutils extensions. `setuptools` treats the names of entry points defined in that group as the acceptable commands for a setup script.

In a similar way, other packages can define their own entry point groups, either using dynamic names within the group (like `distutils.commands`), or possibly using predefined names within the group. For example, a blogging framework that offers various pre- or post-publishing hooks might define an entry point group and look for entry points named "pre_process" and "post_process" within that group.

To advertise an entry point, a project needs to use `setuptools` and provide an `entry_points` argument to `setup()` in its setup script, so that the entry points will be included in the distribution's metadata. For more details, see the `setuptools` documentation. (XXX link here to setuptools)

Each project distribution can advertise at most one entry point of a given name within the same entry point group. For example, a distutils extension could advertise two different `distutils.commands` entry points, as long as they

had different names. However, there is nothing that prevents *different* projects from advertising entry points of the same name in the same group. In some cases, this is a desirable thing, since the application or framework that uses the entry points may be calling them as hooks, or in some other way combining them. It is up to the application or framework to decide what to do if multiple distributions advertise an entry point; some possibilities include using both entry points, displaying an error message, using the first one found in sys.path order, etc.

### Convenience API

In the following functions, the *dist* argument can be a `Distribution` instance, a `Requirement` instance, or a string specifying a requirement (i.e. project name, version, etc.). If the argument is a string or `Requirement`, the specified distribution is located (and added to sys.path if not already present). An error will be raised if a matching distribution is not available.

The *group* argument should be a string containing a dotted identifier, identifying an entry point group. If you are defining an entry point group, you should include some portion of your package's name in the group name so as to avoid collision with other packages' entry point groups.

**`load_entry_point(dist, group, name)`** Load the named entry point from the specified distribution, or raise `ImportError`.

**`get_entry_info(dist, group, name)`** Return an `EntryPoint` object for the given *group* and *name* from the specified distribution. Returns `None` if the distribution has not advertised a matching entry point.

**`get_entry_map(dist, group=None)`** Return the distribution's entry point map for *group*, or the full entry map for the distribution. This function always returns a dictionary, even if the distribution advertises no entry points. If *group* is given, the dictionary maps entry point names to the corresponding `EntryPoint` object. If *group* is None, the dictionary maps group names to dictionaries that then map entry point names to the corresponding `EntryPoint` instance in that group.

**`iter_entry_points(group, name=None)`** Yield entry point objects from *group* matching *name*.

If *name* is None, yields all entry points in *group* from all distributions in the working set on sys.path, otherwise only ones matching both *group* and *name* are yielded. Entry points are yielded from the active distributions in the order that the distributions appear on sys.path. (Within entry points for a particular distribution, however, there is no particular ordering.)

(This API is actually a method of the global `working_set` object; see the section above on Basic WorkingSet Methods for more information.)

### Creating and Parsing

**`EntryPoint(name, module_name, attrs=(), extras=(), dist=None)`** Create an `EntryPoint` instance. *name* is the entry point name. The *module_name* is the (dotted) name of the module containing the advertised object. *attrs* is an optional tuple of names to look up from the module to obtain the advertised object. For example, an *attrs* of `("foo","bar")` and a *module_name* of `"baz"` would mean that the advertised object could be obtained by the following code:

```
import baz
advertised_object = baz.foo.bar
```

The *extras* are an optional tuple of "extra feature" names that the distribution needs in order to provide this entry point. When the entry point is loaded, these extra features are looked up in the *dist* argument to find out what other distributions may need to be activated on sys.path; see the `load()` method for more details. The *extras* argument is only meaningful if *dist* is specified. *dist* must be a `Distribution` instance.

**`EntryPoint.parse(src, dist=None)` (classmethod)** Parse a single entry point from string *src*

Entry point syntax follows the form:

```
name = some.module:some.attr [extra1,extra2]
```

The entry name and module name are required, but the `:attrs` and `[extras]` parts are optional, as is the whitespace shown between some of the items. The *dist* argument is passed through to the `EntryPoint()` constructor, along with the other values parsed from *src*.

**`EntryPoint.parse_group(group, lines, dist=None)` (classmethod)** Parse *lines* (a string or sequence of lines) to create a dictionary mapping entry point names to `EntryPoint` objects. `ValueError` is raised if entry point names are duplicated, if *group* is not a valid entry point group name, or if there are any syntax errors. (Note: the *group* parameter is used only for validation and to create more informative error messages.) If *dist* is provided, it will be used to set the `dist` attribute of the created `EntryPoint` objects.

**`EntryPoint.parse_map(data, dist=None)` (classmethod)** Parse *data* into a dictionary mapping group names to dictionaries mapping entry point names to `EntryPoint` objects. If *data* is a dictionary, then the keys are used as group names and the values are passed to `parse_group()` as the *lines* argument. If *data* is a string or sequence of lines, it is first split into .ini-style sections (using the `split_sections()` utility function) and the section names are used as group names. In either case, the *dist* argument is passed through to `parse_group()` so that the entry points will be linked to the specified distribution.

#### `EntryPoint` Objects

For simple introspection, `EntryPoint` objects have attributes that correspond exactly to the constructor argument names: `name`, `module_name`, `attrs`, `extras`, and `dist` are all available. In addition, the following methods are provided:

**`load(require=True, env=None, installer=None)`** Load the entry point, returning the advertised Python object, or raise `ImportError` if it cannot be obtained. If *require* is a true value, then `require(env, installer)` is called before attempting the import.

**`require(env=None, installer=None)`** Ensure that any "extras" needed by the entry point are available on sys.path. `UnknownExtra` is raised if the `EntryPoint` has `extras`, but no `dist`, or if the named extras are not defined by the distribution. If *env* is supplied, it must be an `Environment`, and it will be used to search for needed distributions if they are not already present on sys.path. If *installer* is supplied, it must be a callable taking a `Requirement` instance and returning a matching importable `Distribution` instance or None.

**`__str__()`** The string form of an `EntryPoint` is a string that could be passed to `EntryPoint.parse()` to produce an equivalent `EntryPoint`.

### 6.2.6 `Distribution` Objects

`Distribution` objects represent collections of Python code that may or may not be importable, and may or may not have metadata and resources associated with them. Their metadata may include information such as what other projects the distribution depends on, what entry points the distribution advertises, and so on.

#### Getting or Creating Distributions

Most commonly, you'll obtain `Distribution` objects from a `WorkingSet` or an `Environment`. (See the sections above on WorkingSet Objects and Environment Objects, which are containers for active distributions and available distributions, respectively.) You can also obtain `Distribution` objects from one of these high-level APIs:

**`find_distributions(path_item, only=False)`** Yield distributions accessible via *path_item*. If *only* is true, yield only distributions whose `location` is equal to *path_item*. In other words, if *only* is true, this yields any distributions that would be importable if *path_item* were on `sys.path`. If *only* is false, this also yields

distributions that are "in" or "under" *path_item*, but would not be importable unless their locations were also added to `sys.path`.

**`get_distribution(dist_spec)`** Return a `Distribution` object for a given `Requirement` or string. If *dist_spec* is already a `Distribution` instance, it is returned. If it is a `Requirement` object or a string that can be parsed into one, it is used to locate and activate a matching distribution, which is then returned.

However, if you're creating specialized tools for working with distributions, or creating a new distribution format, you may also need to create `Distribution` objects directly, using one of the three constructors below.

These constructors all take an optional *metadata* argument, which is used to access any resources or metadata associated with the distribution. *metadata* must be an object that implements the `IResourceProvider` interface, or None. If it is None, an `EmptyProvider` is used instead. `Distribution` objects implement both the IResourceProvider and IMetadataProvider Methods by delegating them to the *metadata* object.

**`Distribution.from_location(location, basename, metadata=None, **kw)` (classmethod)**
Create a distribution for *location*, which must be a string such as a URL, filename, or other string that might be used on `sys.path`. *basename* is a string naming the distribution, like `Foo-1.2-py2.4.egg`. If *basename* ends with `.egg`, then the project's name, version, python version and platform are extracted from the filename and used to set those properties of the created distribution. Any additional keyword arguments are forwarded to the `Distribution()` constructor.

**`Distribution.from_filename(filename, metadata=None**kw)` (classmethod)**
Create a distribution by parsing a local filename. This is a shorter way of saying `Distribution.from_location(normalize_path(filename), os.path.basename(filename), metadata)`. In other words, it creates a distribution whose location is the normalize form of the filename, parsing name and version information from the base portion of the filename. Any additional keyword arguments are forwarded to the `Distribution()` constructor.

**`Distribution(location,metadata,project_name,version,py_version,platform,precedence)`**
Create a distribution by setting its properties. All arguments are optional and default to None, except for *py_version* (which defaults to the current Python version) and *precedence* (which defaults to `EGG_DIST`; for more details see `precedence` under Distribution Attributes below). Note that it's usually easier to use the `from_filename()` or `from_location()` constructors than to specify all these arguments individually.

### `Distribution` Attributes

**location** A string indicating the distribution's location. For an importable distribution, this is the string that would be added to `sys.path` to make it actively importable. For non-importable distributions, this is simply a filename, URL, or other way of locating the distribution.

**project_name** A string, naming the project that this distribution is for. Project names are defined by a project's setup script, and they are used to identify projects on PyPI. When a `Distribution` is constructed, the *project_name* argument is passed through the `safe_name()` utility function to filter out any unacceptable characters.

**key** `dist.key` is short for `dist.project_name.lower()`. It's used for case-insensitive comparison and indexing of distributions by project name.

**extras** A list of strings, giving the names of extra features defined by the project's dependency list (the `extras_require` argument specified in the project's setup script).

**version** A string denoting what release of the project this distribution contains. When a `Distribution` is constructed, the *version* argument is passed through the `safe_version()` utility function to filter out any unacceptable characters. If no *version* is specified at construction time, then attempting to access this attribute later will cause the `Distribution` to try to discover its version by reading its `PKG-INFO` metadata file. If `PKG-INFO` is unavailable or can't be parsed, `ValueError` is raised.

**parsed_version** The `parsed_version` is a tuple representing a "parsed" form of the distribution's `version`. `dist.parsed_version` is a shortcut for calling `parse_version(dist.version)`. It is used to

compare or sort distributions by version. (See the Parsing Utilities section below for more information on the `parse_version()` function.) Note that accessing `parsed_version` may result in a `ValueError` if the `Distribution` was constructed without a *version* and without *metadata* capable of supplying the missing version info.

**py_version** The major/minor Python version the distribution supports, as a string. For example, "2.7" or "3.4". The default is the current version of Python.

**platform** A string representing the platform the distribution is intended for, or `None` if the distribution is "pure Python" and therefore cross-platform. See Platform Utilities below for more information on platform strings.

**precedence** A distribution's `precedence` is used to determine the relative order of two distributions that have the same `project_name` and `parsed_version`. The default precedence is `pkg_resources.EGG_DIST`, which is the highest (i.e. most preferred) precedence. The full list of predefined precedences, from most preferred to least preferred, is: `EGG_DIST`, `BINARY_DIST`, `SOURCE_DIST`, `CHECKOUT_DIST`, and `DEVELOP_DIST`. Normally, precedences other than `EGG_DIST` are used only by the `setuptools.package_index` module, when sorting distributions found in a package index to determine their suitability for installation. "System" and "Development" eggs (i.e., ones that use the `.egg-info` format), however, are automatically given a precedence of `DEVELOP_DIST`.

### `Distribution` Methods

**`activate(path=None)`** Ensure distribution is importable on *path*. If *path* is None, `sys.path` is used instead. This ensures that the distribution's `location` is in the *path* list, and it also performs any necessary namespace package fixups or declarations. (That is, if the distribution contains namespace packages, this method ensures that they are declared, and that the distribution's contents for those namespace packages are merged with the contents provided by any other active distributions. See the section above on Namespace Package Support for more information.)

`pkg_resources` adds a notification callback to the global `working_set` that ensures this method is called whenever a distribution is added to it. Therefore, you should not normally need to explicitly call this method. (Note that this means that namespace packages on `sys.path` are always imported as soon as `pkg_resources` is, which is another reason why namespace packages should not contain any code or import statements.)

**`as_requirement()`** Return a `Requirement` instance that matches this distribution's project name and version.

**`requires(extras=())`** List the `Requirement` objects that specify this distribution's dependencies. If *extras* is specified, it should be a sequence of names of "extras" defined by the distribution, and the list returned will then include any dependencies needed to support the named "extras".

**`clone(**kw)`** Create a copy of the distribution. Any supplied keyword arguments override the corresponding argument to the `Distribution()` constructor, allowing you to change some of the copied distribution's attributes.

**`egg_name()`** Return what this distribution's standard filename should be, not including the ".egg" extension. For example, a distribution for project "Foo" version 1.2 that runs on Python 2.3 for Windows would have an `egg_name()` of `Foo-1.2-py2.3-win32`. Any dashes in the name or version are converted to underscores. (`Distribution.from_location()` will convert them back when parsing a ".egg" file name.)

**`__cmp__(other), __hash__()`** Distribution objects are hashed and compared on the basis of their parsed version and precedence, followed by their key (lowercase project name), location, Python version, and platform.

The following methods are used to access `EntryPoint` objects advertised by the distribution. See the section above on Entry Points for more detailed information about these operations:

**`get_entry_info(group, name)`** Return the `EntryPoint` object for *group* and *name*, or None if no such point is advertised by this distribution.

**get_entry_map(group=None)** Return the entry point map for *group*. If *group* is None, return a dictionary mapping group names to entry point maps for all groups. (An entry point map is a dictionary of entry point names to `EntryPoint` objects.)

**load_entry_point(group, name)** Short for `get_entry_info(group, name).load()`. Returns the object advertised by the named entry point, or raises `ImportError` if the entry point isn't advertised by this distribution, or there is some other import problem.

In addition to the above methods, `Distribution` objects also implement all of the IResourceProvider and IMetadataProvider Methods (which are documented in later sections):

- `has_metadata(name)`
- `metadata_isdir(name)`
- `metadata_listdir(name)`
- `get_metadata(name)`
- `get_metadata_lines(name)`
- `run_script(script_name, namespace)`
- `get_resource_filename(manager, resource_name)`
- `get_resource_stream(manager, resource_name)`
- `get_resource_string(manager, resource_name)`
- `has_resource(resource_name)`
- `resource_isdir(resource_name)`
- `resource_listdir(resource_name)`

If the distribution was created with a *metadata* argument, these resource and metadata access methods are all delegated to that *metadata* provider. Otherwise, they are delegated to an `EmptyProvider`, so that the distribution will appear to have no resources or metadata. This delegation approach is used so that supporting custom importers or new distribution formats can be done simply by creating an appropriate IResourceProvider implementation; see the section below on Supporting Custom Importers for more details.

### 6.2.7 `ResourceManager` API

The `ResourceManager` class provides uniform access to package resources, whether those resources exist as files and directories or are compressed in an archive of some kind.

Normally, you do not need to create or explicitly manage `ResourceManager` instances, as the `pkg_resources` module creates a global instance for you, and makes most of its methods available as top-level names in the `pkg_resources` module namespace. So, for example, this code actually calls the `resource_string()` method of the global `ResourceManager`:

```
import pkg_resources
my_data = pkg_resources.resource_string(__name__, "foo.dat")
```

Thus, you can use the APIs below without needing an explicit `ResourceManager` instance; just import and use them as needed.

#### Basic Resource Access

In the following methods, the *package_or_requirement* argument may be either a Python package/module name (e.g. `foo.bar`) or a `Requirement` instance. If it is a package or module name, the named module or package must

be importable (i.e., be in a distribution or directory on `sys.path`), and the *resource_name* argument is interpreted relative to the named package. (Note that if a module name is used, then the resource name is relative to the package immediately containing the named module. Also, you should not use use a namespace package name, because a namespace package can be spread across multiple distributions, and is therefore ambiguous as to which distribution should be searched for the resource.)

If it is a `Requirement`, then the requirement is automatically resolved (searching the current `Environment` if necessary) and a matching distribution is added to the `WorkingSet` and `sys.path` if one was not already present. (Unless the `Requirement` can't be satisfied, in which case an exception is raised.) The *resource_name* argument is then interpreted relative to the root of the identified distribution; i.e. its first path segment will be treated as a peer of the top-level modules or packages in the distribution.

Note that resource names must be `/`-separated paths and cannot be absolute (i.e. no leading `/`) or contain relative names like `"../"`. Do *not* use `os.path` routines to manipulate resource paths, as they are *not* filesystem paths.

**resource_exists(package_or_requirement, resource_name)** Does the named resource exist? Return `True` or `False` accordingly.

**resource_stream(package_or_requirement, resource_name)** Return a readable file-like object for the specified resource; it may be an actual file, a `StringIO`, or some similar object. The stream is in "binary mode", in the sense that whatever bytes are in the resource will be read as-is.

**resource_string(package_or_requirement, resource_name)** Return the specified resource as a string. The resource is read in binary fashion, such that the returned string contains exactly the bytes that are stored in the resource.

**resource_isdir(package_or_requirement, resource_name)** Is the named resource a directory? Return `True` or `False` accordingly.

**resource_listdir(package_or_requirement, resource_name)** List the contents of the named resource directory, just like `os.listdir` except that it works even if the resource is in a zipfile.

Note that only `resource_exists()` and `resource_isdir()` are insensitive as to the resource type. You cannot use `resource_listdir()` on a file resource, and you can't use `resource_string()` or `resource_stream()` on directory resources. Using an inappropriate method for the resource type may result in an exception or undefined behavior, depending on the platform and distribution format involved.

### Resource Extraction

**resource_filename(package_or_requirement, resource_name)** Sometimes, it is not sufficient to access a resource in string or stream form, and a true filesystem filename is needed. In such cases, you can use this method (or module-level function) to obtain a filename for a resource. If the resource is in an archive distribution (such as a zipped egg), it will be extracted to a cache directory, and the filename within the cache will be returned. If the named resource is a directory, then all resources within that directory (including subdirectories) are also extracted. If the named resource is a C extension or "eager resource" (see the `setuptools` documentation for details), then all C extensions and eager resources are extracted at the same time.

Archived resources are extracted to a cache location that can be managed by the following two methods:

**set_extraction_path(path)** Set the base path where resources will be extracted to, if needed.

If you do not call this routine before any extractions take place, the path defaults to the return value of `get_default_cache()`. (Which is based on the `PYTHON_EGG_CACHE` environment variable, with various platform-specific fallbacks. See that routine's documentation for more details.)

Resources are extracted to subdirectories of this path based upon information given by the resource provider. You may set this to a temporary directory, but then you must call `cleanup_resources()` to delete the extracted files when done. There is no guarantee that `cleanup_resources()` will be able to remove all extracted files. (On Windows, for example, you can't unlink .pyd or .dll files that are still in use.)

Note that you may not change the extraction path for a given resource manager once resources have been extracted, unless you first call `cleanup_resources()`.

**`cleanup_resources(force=False)`** Delete all extracted resource files and directories, returning a list of the file and directory names that could not be successfully removed. This function does not have any concurrency protection, so it should generally only be called when the extraction path is a temporary directory exclusive to a single process. This method is not automatically called; you must call it explicitly or register it as an `atexit` function if you wish to ensure cleanup of a temporary directory used for extractions.

#### "Provider" Interface

If you are implementing an `IResourceProvider` and/or `IMetadataProvider` for a new distribution archive format, you may need to use the following `IResourceManager` methods to co-ordinate extraction of resources to the filesystem. If you're not implementing an archive format, however, you have no need to use these methods. Unlike the other methods listed above, they are *not* available as top-level functions tied to the global `ResourceManager`; you must therefore have an explicit `ResourceManager` instance to use them.

**`get_cache_path(archive_name, names=())`** Return absolute location in cache for *archive_name* and *names*

The parent directory of the resulting path will be created if it does not already exist. *archive_name* should be the base filename of the enclosing egg (which may not be the name of the enclosing zipfile!), including its ".egg" extension. *names*, if provided, should be a sequence of path name parts "under" the egg's extraction location.

This method should only be called by resource providers that need to obtain an extraction location, and only for names they intend to extract, as it tracks the generated names for possible cleanup later.

**`extraction_error()`** Raise an `ExtractionError` describing the active exception as interfering with the extraction process. You should call this if you encounter any OS errors extracting the file to the cache path; it will format the operating system exception for you, and add other information to the `ExtractionError` instance that may be needed by programs that want to wrap or handle extraction errors themselves.

**`postprocess(tempname, filename)`** Perform any platform-specific postprocessing of *tempname*. Resource providers should call this method ONLY after successfully extracting a compressed resource. They must NOT call it on resources that are already in the filesystem.

*tempname* is the current (temporary) name of the file, and *filename* is the name it will be renamed to by the caller after this routine returns.

### 6.2.8 Metadata API

The metadata API is used to access metadata resources bundled in a pluggable distribution. Metadata resources are virtual files or directories containing information about the distribution, such as might be used by an extensible application or framework to connect "plugins". Like other kinds of resources, metadata resource names are `/`-separated and should not contain `..` or begin with a `/`. You should not use `os.path` routines to manipulate resource paths.

The metadata API is provided by objects implementing the `IMetadataProvider` or `IResourceProvider` interfaces. `Distribution` objects implement this interface, as do objects returned by the `get_provider()` function:

**`get_provider(package_or_requirement)`** If a package name is supplied, return an `IResourceProvider` for the package. If a `Requirement` is supplied, resolve it by returning a `Distribution` from the current working set (searching the current `Environment` if necessary and adding the newly found `Distribution` to the working set). If the named package can't be imported, or the `Requirement` can't be satisfied, an exception is raised.

NOTE: if you use a package name rather than a `Requirement`, the object you get back may not be a pluggable distribution, depending on the method by which the package was installed. In particular, "development" packages and "single-version externally-managed" packages do not have any way to map from a package name to the corresponding project's metadata. Do not write code that passes a package name to `get_provider()` and then tries to retrieve project metadata from the returned object. It may appear to work when the named package is in an `.egg` file or directory, but it will fail in other installation scenarios. If you want project metadata, you need to ask for a *project*, not a package.

#### `IMetadataProvider` Methods

The methods provided by objects (such as `Distribution` instances) that implement the `IMetadataProvider` or `IResourceProvider` interfaces are:

**`has_metadata(name)`** Does the named metadata resource exist?

**`metadata_isdir(name)`** Is the named metadata resource a directory?

**`metadata_listdir(name)`** List of metadata names in the directory (like `os.listdir()`)

**`get_metadata(name)`** Return the named metadata resource as a string. The data is read in binary mode; i.e., the exact bytes of the resource file are returned.

**`get_metadata_lines(name)`** Yield named metadata resource as list of non-blank non-comment lines. This is short for calling `yield_lines(provider.get_metadata(name))`. See the section on yield_lines() below for more information on the syntax it recognizes.

**`run_script(script_name, namespace)`** Execute the named script in the supplied namespace dictionary. Raises `ResolutionError` if there is no script by that name in the `scripts` metadata directory. *namespace* should be a Python dictionary, usually a module dictionary if the script is being run as a module.

### 6.2.9 Exceptions

`pkg_resources` provides a simple exception hierarchy for problems that may occur when processing requests to locate and activate packages:

```
ResolutionError
    DistributionNotFound
    VersionConflict
    UnknownExtra

ExtractionError
```

**`ResolutionError`** This class is used as a base class for the other three exceptions, so that you can catch all of them with a single "except" clause. It is also raised directly for miscellaneous requirement-resolution problems like trying to run a script that doesn't exist in the distribution it was requested from.

**`DistributionNotFound`** A distribution needed to fulfill a requirement could not be found.

**`VersionConflict`** The requested version of a project conflicts with an already-activated version of the same project.

**`UnknownExtra`** One of the "extras" requested was not recognized by the distribution it was requested from.

**`ExtractionError`** A problem occurred extracting a resource to the Python Egg cache. The following attributes are available on instances of this exception:

**manager** The resource manager that raised this exception

**cache_path** The base directory for resource extraction

**original_error** The exception instance that caused extraction to fail

### 6.2.10 Supporting Custom Importers

By default, `pkg_resources` supports normal filesystem imports, and `zipimport` importers. If you wish to use the `pkg_resources` features with other (PEP 302-compatible) importers or module loaders, you may need to register various handlers and support functions using these APIs:

**register_finder(importer_type, distribution_finder)** Register *distribution_finder* to find distributions in `sys.path` items. *importer_type* is the type or class of a PEP 302 "Importer" (`sys.path` item handler), and *distribution_finder* is a callable that, when passed a path item, the importer instance, and an *only* flag, yields `Distribution` instances found under that path item. (The *only* flag, if true, means the finder should yield only `Distribution` objects whose `location` is equal to the path item provided.)

See the source of the `pkg_resources.find_on_path` function for an example finder function.

**register_loader_type(loader_type, provider_factory)** Register *provider_factory* to make `IResourceProvider` objects for *loader_type*. *loader_type* is the type or class of a PEP 302 `module.__loader__`, and *provider_factory* is a function that, when passed a module object, returns an IResourceProvider for that module, allowing it to be used with the ResourceManager API.

**register_namespace_handler(importer_type, namespace_handler)** Register *namespace_handler* to declare namespace packages for the given *importer_type*. *importer_type* is the type or class of a PEP 302 "importer" (sys.path item handler), and *namespace_handler* is a callable with a signature like this:

```
def namespace_handler(importer, path_entry, moduleName, module):
    # return a path_entry to use for child packages
```

Namespace handlers are only called if the relevant importer object has already agreed that it can handle the relevant path item. The handler should only return a subpath if the module `__path__` does not already contain an equivalent subpath. Otherwise, it should return None.

For an example namespace handler, see the source of the `pkg_resources.file_ns_handler` function, which is used for both zipfile importing and regular importing.

#### IResourceProvider

`IResourceProvider` is an abstract class that documents what methods are required of objects returned by a *provider_factory* registered with `register_loader_type()`. `IResourceProvider` is a subclass of `IMetadataProvider`, so objects that implement this interface must also implement all of the IMetadataProvider Methods as well as the methods shown here. The *manager* argument to the methods below must be an object that supports the full ResourceManager API documented above.

**get_resource_filename(manager, resource_name)** Return a true filesystem path for *resource_name*, coordinating the extraction with *manager*, if the resource must be unpacked to the filesystem.

**get_resource_stream(manager, resource_name)** Return a readable file-like object for *resource_name*.

**get_resource_string(manager, resource_name)** Return a string containing the contents of *resource_name*.

**has_resource(resource_name)** Does the package contain the named resource?

**resource_isdir(resource_name)** Is the named resource a directory? Return a false value if the resource does not exist or is not a directory.

**resource_listdir(resource_name)** Return a list of the contents of the resource directory, ala `os.listdir()`. Requesting the contents of a non-existent directory may raise an exception.

Note, by the way, that your provider classes need not (and should not) subclass `IResourceProvider` or `IMetadataProvider`! These classes exist solely for documentation purposes and do not provide any useful implementation code. You may instead wish to subclass one of the built-in resource providers.

#### Built-in Resource Providers

`pkg_resources` includes several provider classes that are automatically used where appropriate. Their inheritance tree looks like this:

```
NullProvider
    EggProvider
        DefaultProvider
            PathMetadata
        ZipProvider
            EggMetadata
    EmptyProvider
        FileMetadata
```

**NullProvider** This provider class is just an abstract base that provides for common provider behaviors (such as running scripts), given a definition for just a few abstract methods.

**EggProvider** This provider class adds in some egg-specific features that are common to zipped and unzipped eggs.

**DefaultProvider** This provider class is used for unpacked eggs and "plain old Python" filesystem modules.

**ZipProvider** This provider class is used for all zipped modules, whether they are eggs or not.

**EmptyProvider** This provider class always returns answers consistent with a provider that has no metadata or resources. `Distribution` objects created without a `metadata` argument use an instance of this provider class instead. Since all `EmptyProvider` instances are equivalent, there is no need to have more than one instance. `pkg_resources` therefore creates a global instance of this class under the name `empty_provider`, and you may use it if you have need of an `EmptyProvider` instance.

**PathMetadata(path, egg_info)** Create an `IResourceProvider` for a filesystem-based distribution, where *path* is the filesystem location of the importable modules, and *egg_info* is the filesystem location of the distribution's metadata directory. *egg_info* should usually be the `EGG-INFO` subdirectory of *path* for an "unpacked egg", and a `ProjectName.egg-info` subdirectory of *path* for a "development egg". However, other uses are possible for custom purposes.

**EggMetadata(zipimporter)** Create an `IResourceProvider` for a zipfile-based distribution. The *zipimporter* should be a `zipimport.zipimporter` instance, and may represent a "basket" (a zipfile containing multiple ".egg" subdirectories) a specific egg *within* a basket, or a zipfile egg (where the zipfile itself is a ".egg"). It can also be a combination, such as a zipfile egg that also contains other eggs.

**FileMetadata(path_to_pkg_info)** Create an `IResourceProvider` that provides exactly one metadata resource: `PKG-INFO`. The supplied path should be a distutils PKG-INFO file. This is basically the same as an `EmptyProvider`, except that requests for `PKG-INFO` will be answered using the contents of the designated file. (This provider is used to wrap `.egg-info` files installed by vendor-supplied system packages.)

### 6.2.11 Utility Functions

In addition to its high-level APIs, `pkg_resources` also includes several generally-useful utility routines. These routines are used to implement the high-level APIs, but can also be quite useful by themselves.

### Parsing Utilities

**parse_version(version)** Parse a project's version string, returning a value that can be used to compare versions by chronological order. Semantically, the format is a rough cross between distutils' `StrictVersion` and `LooseVersion` classes; if you give it versions that would work with `StrictVersion`, then they will compare the same way. Otherwise, comparisons are more like a "smarter" form of `LooseVersion`. It is *possible* to create pathological version coding schemes that will fool this parser, but they should be very rare in practice.

The returned value will be a tuple of strings. Numeric portions of the version are padded to 8 digits so they will compare numerically, but without relying on how numbers compare relative to strings. Dots are dropped, but dashes are retained. Trailing zeros between alpha segments or dashes are suppressed, so that e.g. "2.4.0" is considered the same as "2.4". Alphanumeric parts are lower-cased.

The algorithm assumes that strings like "-" and any alpha string that alphabetically follows "final" represents a "patch level". So, "2.4-1" is assumed to be a branch or patch of "2.4", and therefore "2.4.1" is considered newer than "2.4-1", which in turn is newer than "2.4".

Strings like "a", "b", "c", "alpha", "beta", "candidate" and so on (that come before "final" alphabetically) are assumed to be pre-release versions, so that the version "2.4" is considered newer than "2.4a1". Any "-" characters preceding a pre-release indicator are removed. (In versions of setuptools prior to 0.6a9, "-" characters were not removed, leading to the unintuitive result that "0.2-rc1" was considered a newer version than "0.2".)

Finally, to handle miscellaneous cases, the strings "pre", "preview", and "rc" are treated as if they were "c", i.e. as though they were release candidates, and therefore are not as new as a version string that does not contain them. And the string "dev" is treated as if it were an "@" sign; that is, a version coming before even "a" or "alpha".

**yield_lines(strs)** Yield non-empty/non-comment lines from a string/unicode or a possibly- nested sequence thereof. If *strs* is an instance of `basestring`, it is split into lines, and each non-blank, non-comment line is yielded after stripping leading and trailing whitespace. (Lines whose first non-blank character is `#` are considered comment lines.)

If *strs* is not an instance of `basestring`, it is iterated over, and each item is passed recursively to `yield_lines()`, so that an arbitrarily nested sequence of strings, or sequences of sequences of strings can be flattened out to the lines contained therein. So for example, passing a file object or a list of strings to `yield_lines` will both work. (Note that between each string in a sequence of strings there is assumed to be an implicit line break, so lines cannot bridge two strings in a sequence.)

This routine is used extensively by `pkg_resources` to parse metadata and file formats of various kinds, and most other `pkg_resources` parsing functions that yield multiple values will use it to break up their input. However, this routine is idempotent, so calling `yield_lines()` on the output of another call to `yield_lines()` is completely harmless.

**split_sections(strs)** Split a string (or possibly-nested iterable thereof), yielding `(section, content)` pairs found using an `.ini`-like syntax. Each `section` is a whitespace-stripped version of the section name ("`[section]`") and each `content` is a list of stripped lines excluding blank lines and comment-only lines. If there are any non-blank, non-comment lines before the first section header, they're yielded in a first `section` of `None`.

This routine uses `yield_lines()` as its front end, so you can pass in anything that `yield_lines()` accepts, such as an open text file, string, or sequence of strings. `ValueError` is raised if a malformed section header is found (i.e. a line starting with `[` but not ending with `]`).

Note that this simplistic parser assumes that any line whose first nonblank character is `[` is a section heading, so it can't support .ini format variations that allow `[` as the first nonblank character on other lines.

**safe_name(name)** Return a "safe" form of a project's name, suitable for use in a `Requirement` string, as a distribution name, or a PyPI project name. All non-alphanumeric runs are condensed to single "-" characters,

such that a name like "The $$$ Tree" becomes "The-Tree". Note that if you are generating a filename from this value you should combine it with a call to `to_filename()` so all dashes ("-") are replaced by underscores ("_"). See `to_filename()`.

**`safe_version(version)`** Similar to `safe_name()` except that spaces in the input become dots, and dots are allowed to exist in the output. As with `safe_name()`, if you are generating a filename from this you should replace any "-" characters in the output with underscores.

**`safe_extra(extra)`** Return a "safe" form of an extra's name, suitable for use in a requirement string or a setup script's `extras_require` keyword. This routine is similar to `safe_name()` except that non-alphanumeric runs are replaced by a single underbar (_), and the result is lowercased.

**`to_filename(name_or_version)`** Escape a name or version string so it can be used in a dash-separated filename (or `#egg=name-version` tag) without ambiguity. You should only pass in values that were returned by `safe_name()` or `safe_version()`.

### Platform Utilities

**`get_build_platform()`** Return this platform's identifier string. For Windows, the return value is `"win32"`, and for Mac OS X it is a string of the form `"macosx-10.4-ppc"`. All other platforms return the same uname-based string that the `distutils.util.get_platform()` function returns. This string is the minimum platform version required by distributions built on the local machine. (Backward compatibility note: setuptools versions prior to 0.6b1 called this function `get_platform()`, and the function is still available under that name for backward compatibility reasons.)

**`get_supported_platform()` (New in 0.6b1)** This is the similar to `get_build_platform()`, but is the maximum platform version that the local machine supports. You will usually want to use this value as the `provided` argument to the `compatible_platforms()` function.

**`compatible_platforms(provided, required)`** Return true if a distribution built on the *provided* platform may be used on the *required* platform. If either platform value is `None`, it is considered a wildcard, and the platforms are therefore compatible. Likewise, if the platform strings are equal, they're also considered compatible, and `True` is returned. Currently, the only non-equal platform strings that are considered compatible are Mac OS X platform strings with the same hardware type (e.g. `ppc`) and major version (e.g. `10`) with the *provided* platform's minor version being less than or equal to the *required* platform's minor version.

**`get_default_cache()`** Determine the default cache location for extracting resources from zipped eggs. This routine returns the `PYTHON_EGG_CACHE` environment variable, if set. Otherwise, on Windows, it returns a "Python-Eggs" subdirectory of the user's "Application Data" directory. On all other systems, it returns `os.path.expanduser("~/.python-eggs")` if `PYTHON_EGG_CACHE` is not set.

### PEP 302 Utilities

**`get_importer(path_item)`** Retrieve a PEP 302 "importer" for the given path item (which need not actually be on `sys.path`). This routine simulates the PEP 302 protocol for obtaining an "importer" object. It first checks for an importer for the path item in `sys.path_importer_cache`, and if not found it calls each of the `sys.path_hooks` and caches the result if a good importer is found. If no importer is found, this routine returns an `ImpWrapper` instance that wraps the builtin import machinery as a PEP 302-compliant "importer" object. This `ImpWrapper` is *not* cached; instead a new instance is returned each time.

(Note: When run under Python 2.5, this function is simply an alias for `pkgutil.get_importer()`, and instead of `pkg_resources.ImpWrapper` instances, it may return `pkgutil.ImpImporter` instances.)

### File/Path Utilities

**ensure_directory(path)** Ensure that the parent directory (`os.path.dirname`) of *path* actually exists, using `os.makedirs()` if necessary.

**normalize_path(path)** Return a "normalized" version of *path*, such that two paths represent the same filesystem location if they have equal `normalized_path()` values. Specifically, this is a shortcut for calling `os.path.realpath` and `os.path.normcase` on *path*. Unfortunately, on certain platforms (notably Cygwin and Mac OS X) the `normcase` function does not accurately reflect the platform's case-sensitivity, so there is always the possibility of two apparently-different paths being equal on such platforms.

### History

**0.6c9**

- Fix `resource_listdir('')` always returning an empty list for zipped eggs.

**0.6c7**

- Fix package precedence problem where single-version eggs installed in `site-packages` would take precedence over `.egg` files (or directories) installed in `site-packages`.

**0.6c6**

- Fix extracted C extensions not having executable permissions under Cygwin.
- Allow `.egg-link` files to contain relative paths.
- Fix cache dir defaults on Windows when multiple environment vars are needed to construct a path.

**0.6c4**

- Fix "dev" versions being considered newer than release candidates.

**0.6c3**

- Python 2.5 compatibility fixes.

**0.6c2**

- Fix a problem with eggs specified directly on `PYTHONPATH` on case-insensitive filesystems possibly not showing up in the default working set, due to differing normalizations of `sys.path` entries.

**0.6b3**

- Fixed a duplicate path insertion problem on case-insensitive filesystems.

**0.6b1**

- Split `get_platform()` into `get_supported_platform()` and `get_build_platform()` to work around a Mac versioning problem that caused the behavior of `compatible_platforms()` to be platform specific.
- Fix entry point parsing when a standalone module name has whitespace between it and the extras.

**0.6a11**

- Added `ExtractionError` and `ResourceManager.extraction_error()` so that cache permission problems get a more user-friendly explanation of the problem, and so that programs can catch and handle extraction errors if they need to.

**0.6a10**

- Added the `extras` attribute to `Distribution`, the `find_plugins()` method to `WorkingSet`, and the `__add__()` and `__iadd__()` methods to `Environment`.

- `safe_name()` now allows dots in project names.
- There is a new `to_filename()` function that escapes project names and versions for safe use in constructing egg filenames from a Distribution object's metadata.
- Added `Distribution.clone()` method, and keyword argument support to other `Distribution` constructors.
- Added the `DEVELOP_DIST` precedence, and automatically assign it to eggs using `.egg-info` format.

**0.6a9**

- Don't raise an error when an invalid (unfinished) distribution is found unless absolutely necessary. Warn about skipping invalid/unfinished eggs when building an Environment.
- Added support for `.egg-info` files or directories with version/platform information embedded in the filename, so that system packagers have the option of including `PKG-INFO` files to indicate the presence of a system-installed egg, without needing to use `.egg` directories, zipfiles, or `.pth` manipulation.
- Changed `parse_version()` to remove dashes before pre-release tags, so that `0.2-rc1` is considered an *older* version than `0.2`, and is equal to `0.2rc1`. The idea that a dash *always* meant a post-release version was highly non-intuitive to setuptools users and Python developers, who seem to want to use `-rc` version numbers a lot.

**0.6a8**

- Fixed a problem with `WorkingSet.resolve()` that prevented version conflicts from being detected at runtime.
- Improved runtime conflict warning message to identify a line in the user's program, rather than flagging the `warn()` call in `pkg_resources`.
- Avoid giving runtime conflict warnings for namespace packages, even if they were declared by a different package than the one currently being activated.
- Fix path insertion algorithm for case-insensitive filesystems.
- Fixed a problem with nested namespace packages (e.g. `peak.util`) not being set as an attribute of their parent package.

**0.6a6**

- Activated distributions are now inserted in `sys.path` (and the working set) just before the directory that contains them, instead of at the end. This allows e.g. eggs in `site-packages` to override unmanaged modules in the same location, and allows eggs found earlier on `sys.path` to override ones found later.
- When a distribution is activated, it now checks whether any contained non-namespace modules have already been imported and issues a warning if a conflicting module has already been imported.
- Changed dependency processing so that it's breadth-first, allowing a depender's preferences to override those of a dependee, to prevent conflicts when a lower version is acceptable to the dependee, but not the depender.
- Fixed a problem extracting zipped files on Windows, when the egg in question has had changed contents but still has the same version number.

**0.6a4**

- Fix a bug in `WorkingSet.resolve()` that was introduced in 0.6a3.

**0.6a3**

- Added `safe_extra()` parsing utility routine, and use it for Requirement, EntryPoint, and Distribution objects' extras handling.

**0.6a1**

- Enhanced performance of `require()` and related operations when all requirements are already in the working set, and enhanced performance of directory scanning for distributions.
- Fixed some problems using `pkg_resources` w/PEP 302 loaders other than `zipimport`, and the previously-broken "eager resource" support.
- Fixed `pkg_resources.resource_exists()` not working correctly, along with some other resource API bugs.
- Many API changes and enhancements:
  - Added `EntryPoint`, `get_entry_map`, `load_entry_point`, and `get_entry_info` APIs for dynamic plugin discovery.
  - `list_resources` is now `resource_listdir` (and it actually works)
  - Resource API functions like `resource_string()` that accepted a package name and resource name, will now also accept a `Requirement` object in place of the package name (to allow access to non-package data files in an egg).
  - `get_provider()` will now accept a `Requirement` instance or a module name. If it is given a `Requirement`, it will return a corresponding `Distribution` (by calling `require()` if a suitable distribution isn't already in the working set), rather than returning a metadata and resource provider for a specific module. (The difference is in how resource paths are interpreted; supplying a module name means resources path will be module-relative, rather than relative to the distribution's root.)
  - `Distribution` objects now implement the `IResourceProvider` and `IMetadataProvider` interfaces, so you don't need to reference the (no longer available) `metadata` attribute to get at these interfaces.
  - `Distribution` and `Requirement` both have a `project_name` attribute for the project name they refer to. (Previously these were `name` and `distname` attributes.)
  - The `path` attribute of `Distribution` objects is now `location`, because it isn't necessarily a filesystem path (and hasn't been for some time now). The `location` of `Distribution` objects in the filesystem should always be normalized using `pkg_resources.normalize_path()`; all of the setuptools and EasyInstall code that generates distributions from the filesystem (including `Distribution.from_filename()`) ensure this invariant, but if you use a more generic API like `Distribution()` or `Distribution.from_location()` you should take care that you don't create a distribution with an un-normalized filesystem path.
  - `Distribution` objects now have an `as_requirement()` method that returns a `Requirement` for the distribution's project name and version.
  - Distribution objects no longer have an `installed_on()` method, and the `install_on()` method is now `activate()` (but may go away altogether soon). The `depends()` method has also been renamed to `requires()`, and `InvalidOption` is now `UnknownExtra`.
  - `find_distributions()` now takes an additional argument called `only`, that tells it to only yield distributions whose location is the passed-in path. (It defaults to False, so that the default behavior is unchanged.)
  - `AvailableDistributions` is now called `Environment`, and the `get()`, `__len__()`, and `__contains__()` methods were removed, because they weren't particularly useful. `__getitem__()` no longer raises `KeyError`; it just returns an empty list if there are no distributions for the named project.
  - The `resolve()` method of `Environment` is now a method of `WorkingSet` instead, and the `best_match()` method now uses a working set instead of a path list as its second argument.

  - There is a new `pkg_resources.add_activation_listener()` API that lets you register a callback for notifications about distributions added to `sys.path` (including the distributions already on it). This is basically a hook for extensible applications and frameworks to be able to search for plugin metadata in distributions added at runtime.

**0.5a13**

- Fixed a bug in resource extraction from nested packages in a zipped egg.

**0.5a12**

- Updated extraction/cache mechanism for zipped resources to avoid inter- process and inter-thread races during extraction. The default cache location can now be set via the `PYTHON_EGGS_CACHE` environment variable, and the default Windows cache is now a `Python-Eggs` subdirectory of the current user's "Application Data" directory, if the `PYTHON_EGGS_CACHE` variable isn't set.

**0.5a10**

- Fix a problem with `pkg_resources` being confused by non-existent eggs on `sys.path` (e.g. if a user deletes an egg without removing it from the `easy-install.pth` file).
- Fix a problem with "basket" support in `pkg_resources`, where egg-finding never actually went inside `.egg` files.
- Made `pkg_resources` import the module you request resources from, if it's not already imported.

**0.5a4**

- `pkg_resources.AvailableDistributions.resolve()` and related methods now accept an `installer` argument: a callable taking one argument, a `Requirement` instance. The callable must return a `Distribution` object, or `None` if no distribution is found. This feature is used by EasyInstall to resolve dependencies by recursively invoking itself.

**0.4a4**

- Fix problems with `resource_listdir()`, `resource_isdir()` and resource directory extraction for zipped eggs.

**0.4a3**

- Fixed scripts not being able to see a `__file__` variable in `__main__`
- Fixed a problem with `resource_isdir()` implementation that was introduced in 0.4a2.

**0.4a1**

- Fixed a bug in requirements processing for exact versions (i.e. `==` and `!=`) when only one condition was included.
- Added `safe_name()` and `safe_version()` APIs to clean up handling of arbitrary distribution names and versions found on PyPI.

**0.3a4**

- `pkg_resources` now supports resource directories, not just the resources in them. In particular, there are `resource_listdir()` and `resource_isdir()` APIs.
- `pkg_resources` now supports "egg baskets" – .egg zipfiles which contain multiple distributions in subdirectories whose names end with `.egg`. Having such a "basket" in a directory on `sys.path` is equivalent to having the individual eggs in that directory, but the contained eggs can be individually added (or not) to `sys.path`. Currently, however, there is no automated way to create baskets.
- Namespace package manipulation is now protected by the Python import lock.

**0.3a1**

- Initial release.

CHAPTER 7

# Development on Setuptools

Setuptools is maintained by the Python community under the Python Packaging Authority (PyPA) and led by Jason R. Coombs.

This document describes the process by which Setuptools is developed. This document assumes the reader has some passing familiarity with *using* setuptools, the `pkg_resources` module, and EasyInstall. It does not attempt to explain basic concepts like inter-project dependencies, nor does it contain detailed lexical syntax for most file formats. Neither does it explain concepts like "namespace packages" or "resources" in any detail, as all of these subjects are covered at length in the setuptools developer's guide and the `pkg_resources` reference manual.

Instead, this is **internal** documentation for how those concepts and features are *implemented* in concrete terms. It is intended for people who are working on the setuptools code base, who want to be able to troubleshoot setuptools problems, want to write code that reads the file formats involved, or want to otherwise tinker with setuptools-generated files and directories.

Note, however, that these are all internal implementation details and are therefore subject to change; stick to the published API if you don't want to be responsible for keeping your code from breaking when setuptools changes. You have been warned.

## 7.1 Developer's Guide for Setuptools

If you want to know more about contributing on Setuptools, this is the place.

**Table of Contents**



### 7.1.1 Recommended Reading

Please read How to write the perfect pull request for some tips on contributing to open source projects. Although the article is not authoritative, it was authored by the maintainer of Setuptools, so reflects his opinions and will improve the likelihood of acceptance and quality of contribution.

### 7.1.2 Project Management

Setuptools is maintained primarily in Bitbucket at this home. Setuptools is maintained under the Python Packaging Authority (PyPA) with several core contributors. All bugs for Setuptools are filed and the canonical source is maintained in Bitbucket.

User support and discussions are done through the issue tracker (for specific) issues, through the distutils-sig mailing list, or on IRC (Freenode) at #pypa.

Discussions about development happen on the pypa-dev mailing list or on IRC (Freenode) at #pypa-dev.

### 7.1.3 Authoring Tickets

Before authoring any source code, it's often prudent to file a ticket describing the motivation behind making changes. First search to see if a ticket already exists for your issue. If not, create one. Try to think from the perspective of the reader. Explain what behavior you expected, what you got instead, and what factors might have contributed to the unexpected behavior. In Bitbucket, surround a block of code or traceback with the triple backtick "```" so that it is formatted nicely.

Filing a ticket provides a forum for justification, discussion, and clarification. The ticket provides a record of the purpose for the change and any hard decisions that were made. It provides a single place for others to reference when trying to understand why the software operates the way it does or why certain changes were made.

Setuptools makes extensive use of hyperlinks to tickets in the changelog so that system integrators and other users can get a quick summary, but then jump to the in-depth discussion about any subject referenced.

### 7.1.4 Source Code

Grab the code at Bitbucket:

```
$ hg clone https://bitbucket.org/pypa/setuptools
```

If you want to contribute changes, we recommend you fork the repository on Bitbucket, commit the changes to your repository, and then make a pull request on Bitbucket. If you make some changes, don't forget to:

- add a note in CHANGES.txt

Please commit all changes in the 'default' branch against the latest available commit or for bug-fixes, against an earlier commit or release in which the bug occurred.

If you find yourself working on more than one issue at a time, Setuptools generally prefers Git-style branches, so use Mercurial bookmarks or Git branches or multiple forks to maintain separate efforts.

Setuptools also maintains an unofficial Git mirror in Github. Contributors are welcome to submit pull requests here, but because they are not integrated with the Bitbucket Issue tracker, linking pull requests to tickets is more difficult. The Continuous Integration tests that validate every release are run from this mirror.

### 7.1.5 Testing

The primary tests are run using py.test. To run the tests:

```
$ python setup.py ptr
```

Or install py.test into your environment and run `py.test`.

Under continuous integration, additional tests may be run. See the `.travis.yml` file for full details on the tests run under Travis-CI.

### 7.1.6 Semantic Versioning

Setuptools follows `semver` with some exceptions:

- Uses two-segment version when three segment version ends in zero
- Omits 'v' prefix for tags.

## 7.2 The Internal Structure of Python Eggs

STOP! This is not the first document you should read!

**Table of Contents**



### 7.2.1 Eggs and their Formats

A "Python egg" is a logical structure embodying the release of a specific version of a Python project, comprising its code, resources, and metadata. There are multiple formats that can be used to physically encode a Python egg, and others can be developed. However, a key principle of Python eggs is that they should be discoverable and importable. That is, it should be possible for a Python application to easily and efficiently find out what eggs are present on a system, and to ensure that the desired eggs' contents are importable.

There are two basic formats currently implemented for Python eggs:

1. `.egg` format: a directory or zipfile *containing* the project's code and resources, along with an `EGG-INFO` subdirectory that contains the project's metadata
2. `.egg-info` format: a file or directory placed *adjacent* to the project's code and resources, that directly contains the project's metadata.

Both formats can include arbitrary Python code and resources, including static data files, package and non-package directories, Python modules, C extension modules, and so on. But each format is optimized for different purposes.

The `.egg` format is well-suited to distribution and the easy uninstallation or upgrades of code, since the project is essentially self-contained within a single directory or file, unmingled with any other projects' code or resources. It also makes it possible to have multiple versions of a project simultaneously installed, such that individual programs can select the versions they wish to use.

The `.egg-info` format, on the other hand, was created to support backward-compatibility, performance, and ease of installation for system packaging tools that expect to install all projects' code and resources to a single directory (e.g. `site-packages`). Placing the metadata in that same directory simplifies the installation process, since it isn't necessary to create `.pth` files or otherwise modify `sys.path` to include each installed egg.

Its disadvantage, however, is that it provides no support for clean uninstallation or upgrades, and of course only a single version of a project can be installed to a given directory. Thus, support from a package management tool is required. (This is why setuptools' "install" command refers to this type of egg installation as "single-version, externally managed".) Also, they lack sufficient data to allow them to be copied from their installation source. easy_install can "ship" an application by copying `.egg` files or directories to a target location, but it cannot do this for `.egg-info` installs, because there is no way to tell what code and resources belong to a particular egg – there may be several eggs "scrambled" together in a single installation location, and the `.egg-info` format does not currently include a way to list the files that were installed. (This may change in a future version.)

### Code and Resources

The layout of the code and resources is dictated by Python's normal import layout, relative to the egg's "base location".

For the `.egg` format, the base location is the `.egg` itself. That is, adding the `.egg` filename or directory name to `sys.path` makes its contents importable.

For the `.egg-info` format, however, the base location is the directory that *contains* the `.egg-info`, and thus it is the directory that must be added to `sys.path` to make the egg importable. (Note that this means that the "normal" installation of a package to a `sys.path` directory is sufficient to make it an "egg" if it has an `.egg-info` file or directory installed alongside of it.)

### Project Metadata

If eggs contained only code and resources, there would of course be no difference between them and any other directory or zip file on `sys.path`. Thus, metadata must also be included, using a metadata file or directory.

For the `.egg` format, the metadata is placed in an `EGG-INFO` subdirectory, directly within the `.egg` file or directory. For the `.egg-info` format, metadata is stored directly within the `.egg-info` directory itself.

The minimum project metadata that all eggs must have is a standard Python `PKG-INFO` file, named `PKG-INFO` and placed within the metadata directory appropriate to the format. Because it's possible for this to be the only metadata file included, `.egg-info` format eggs are not required to be a directory; they can just be a `.egg-info` file that directly contains the `PKG-INFO` metadata. This eliminates the need to create a directory just to store one file. This option is *not* available for `.egg` formats, since setuptools always includes other metadata. (In fact, setuptools itself never generates `.egg-info` files, either; the support for using files was added so that the requirement could easily be satisfied by other tools, such as the distutils in Python 2.5).

In addition to the `PKG-INFO` file, an egg's metadata directory may also include files and directories representing various forms of optional standard metadata (see the section on Standard Metadata, below) or user-defined metadata

required by the project. For example, some projects may define a metadata format to describe their application plugins, and metadata in this format would then be included by plugin creators in their projects' metadata directories.

### Filename-Embedded Metadata

To allow introspection of installed projects and runtime resolution of inter-project dependencies, a certain amount of information is embedded in egg filenames. At a minimum, this includes the project name, and ideally will also include the project version number. Optionally, it can also include the target Python version and required runtime platform if platform-specific C code is included. The syntax of an egg filename is as follows:

```
name ["-" version ["-py" pyver ["-" required_platform]]] "." ext
```

The "name" and "version" should be escaped using the `to_filename()` function provided by `pkg_resources`, after first processing them with `safe_name()` and `safe_version()` respectively. These latter two functions can also be used to later "unescape" these parts of the filename. (For a detailed description of these transformations, please see the "Parsing Utilities" section of the `pkg_resources` manual.)

The "pyver" string is the Python major version, as found in the first 3 characters of `sys.version`. "required_platform" is essentially a distutils `get_platform()` string, but with enhancements to properly distinguish Mac OS versions. (See the `get_build_platform()` documentation in the "Platform Utilities" section of the `pkg_resources` manual for more details.)

Finally, the "ext" is either `.egg` or `.egg-info`, as appropriate for the egg's format.

Normally, an egg's filename should include at least the project name and version, as this allows the runtime system to find desired project versions without having to read the egg's PKG-INFO to determine its version number.

Setuptools, however, only includes the version number in the filename when an `.egg` file is built using the `bdist_egg` command, or when an `.egg-info` directory is being installed by the `install_egg_info` command. When generating metadata for use with the original source tree, it only includes the project name, so that the directory will not have to be renamed each time the project's version changes.

This is especially important when version numbers change frequently, and the source metadata directory is kept under version control with the rest of the project. (As would be the case when the project's source includes project-defined metadata that is not generated from by setuptools from data in the setup script.)

### Egg Links

In addition to the `.egg` and `.egg-info` formats, there is a third egg-related extension that you may encounter on occasion: `.egg-link` files.

These files are not eggs, strictly speaking. They simply provide a way to reference an egg that is not physically installed in the desired location. They exist primarily as a cross-platform alternative to symbolic links, to support "installing" code that is being developed in a different location than the desired installation location. For example, if a user is developing an application plugin in their home directory, but the plugin needs to be "installed" in an application plugin directory, running "setup.py develop -md /path/to/app/plugins" will install an `.egg-link` file in `/path/to/app/plugins`, that tells the egg runtime system where to find the actual egg (the user's project source directory and its `.egg-info` subdirectory).

`.egg-link` files are named following the format for `.egg` and `.egg-info` names, but only the project name is included; no version, Python version, or platform information is included. When the runtime searches for available eggs, `.egg-link` files are opened and the actual egg file/directory name is read from them.

Each `.egg-link` file should contain a single file or directory name, with no newlines. This filename should be the base location of one or more eggs. That is, the name must either end in `.egg`, or else it should be the parent directory of one or more `.egg-info` format eggs.

As of setuptools 0.6c6, the path may be specified as a platform-independent (i.e. /-separated) relative path from the directory containing the .egg-link file, and a second line may appear in the file, specifying a platform-independent relative path from the egg's base directory to its setup script directory. This allows installation tools such as EasyInstall to find the project's setup directory and build eggs or perform other setup commands on it.

### 7.2.2 Standard Metadata

In addition to the minimum required PKG-INFO metadata, projects can include a variety of standard metadata files or directories, as described below. Except as otherwise noted, these files and directories are automatically generated by setuptools, based on information supplied in the setup script or through analysis of the project's code and resources.

Most of these files and directories are generated via "egg-info writers" during execution of the setuptools egg_info command, and are listed in the egg_info.writers entry point group defined by setuptools' own setup.py file.

Project authors can register their own metadata writers as entry points in this group (as described in the setuptools manual under "Adding new EGG-INFO Files") to cause setuptools to generate project-specific metadata files or directories during execution of the egg_info command. It is up to project authors to document these new metadata formats, if they create any.

#### .txt File Formats

Files described in this section that have .txt extensions have a simple lexical format consisting of a sequence of text lines, each line terminated by a linefeed character (regardless of platform). Leading and trailing whitespace on each line is ignored, as are blank lines and lines whose first nonblank character is a # (comment symbol). (This is the parsing format defined by the yield_lines() function of the pkg_resources module.)

All .txt files defined by this section follow this format, but some are also "sectioned" files, meaning that their contents are divided into sections, using square-bracketed section headers akin to Windows .ini format. Note that this does *not* imply that the lines within the sections follow an .ini format, however. Please see an individual metadata file's documentation for a description of what the lines and section names mean in that particular file.

Sectioned files can be parsed using the split_sections() function; see the "Parsing Utilities" section of the pkg_resources manual for for details.

#### Dependency Metadata

##### requires.txt

This is a "sectioned" text file. Each section is a sequence of "requirements", as parsed by the parse_requirements() function; please see the pkg_resources manual for the complete requirement parsing syntax.

The first, unnamed section (i.e., before the first section header) in this file is the project's core requirements, which must be installed for the project to function. (Specified using the install_requires keyword to setup()).

The remaining (named) sections describe the project's "extra" requirements, as specified using the extras_require keyword to setup(). The section name is the name of the optional feature, and the section body lists that feature's dependencies.

Note that it is not normally necessary to inspect this file directly; pkg_resources.Distribution objects have a requires() method that can be used to obtain Requirement objects describing the project's core and optional dependencies.

#### dependency_links.txt

A list of dependency URLs, one per line, as specified using the `dependency_links` keyword to `setup()`. These may be direct download URLs, or the URLs of web pages containing direct download links, and will be used by EasyInstall to find dependencies, as though the user had manually provided them via the `--find-links` command line option. Please see the setuptools manual and EasyInstall manual for more information on specifying this option, and for information on how EasyInstall processes `--find-links` URLs.

#### depends.txt – Obsolete, do not create!

This file follows an identical format to `requires.txt`, but is obsolete and should not be used. The earliest versions of setuptools required users to manually create and maintain this file, so the runtime still supports reading it, if it exists. The new filename was created so that it could be automatically generated from `setup()` information without overwriting an existing hand-created `depends.txt`, if one was already present in the project's source `.egg-info` directory.

### namespace_packages.txt – Namespace Package Metadata

A list of namespace package names, one per line, as supplied to the `namespace_packages` keyword to `setup()`. Please see the manuals for setuptools and `pkg_resources` for more information about namespace packages.

### entry_points.txt – "Entry Point"/Plugin Metadata

This is a "sectioned" text file, whose contents encode the `entry_points` keyword supplied to `setup()`. All sections are named, as the section names specify the entry point groups in which the corresponding section's entry points are registered.

Each section is a sequence of "entry point" lines, each parseable using the `EntryPoint.parse` classmethod; please see the `pkg_resources` manual for the complete entry point parsing syntax.

Note that it is not necessary to parse this file directly; the `pkg_resources` module provides a variety of APIs to locate and load entry points automatically. Please see the setuptools and `pkg_resources` manuals for details on the nature and uses of entry points.

### The scripts Subdirectory

This directory is currently only created for `.egg` files built by the setuptools `bdist_egg` command. It will contain copies of all of the project's "traditional" scripts (i.e., those specified using the `scripts` keyword to `setup()`). This is so that they can be reconstituted when an `.egg` file is installed.

The scripts are placed here using the distutils' standard `install_scripts` command, so any `#!` lines reflect the Python installation where the egg was built. But instead of copying the scripts to the local script installation directory, EasyInstall writes short wrapper scripts that invoke the original scripts from inside the egg, after ensuring that sys.path includes the egg and any eggs it depends on. For more about script wrappers, see the section below on Installation and Path Management Issues.

### Zip Support Metadata

#### native_libs.txt

A list of C extensions and other dynamic link libraries contained in the egg, one per line. Paths are /-separated and relative to the egg's base location.

This file is generated as part of bdist_egg processing, and as such only appears in .egg files (and .egg directories created by unpacking them). It is used to ensure that all libraries are extracted from a zipped egg at the same time, in case there is any direct linkage between them. Please see the Zip File Issues section below for more information on library and resource extraction from .egg files.

#### eager_resources.txt

A list of resource files and/or directories, one per line, as specified via the eager_resources keyword to setup(). Paths are /-separated and relative to the egg's base location.

Resource files or directories listed here will be extracted simultaneously, if any of the named resources are extracted, or if any native libraries listed in native_libs.txt are extracted. Please see the setuptools manual for details on what this feature is used for and how it works, as well as the Zip File Issues section below.

#### zip-safe and not-zip-safe

These are zero-length files, and either one or the other should exist. If zip-safe exists, it means that the project will work properly when installed as an .egg zipfile, and conversely the existence of not-zip-safe means the project should not be installed as an .egg file. The zip_safe option to setuptools' setup() determines which file will be written. If the option isn't provided, setuptools attempts to make its own assessment of whether the package can work, based on code and content analysis.

If neither file is present at installation time, EasyInstall defaults to assuming that the project should be unzipped. (Command-line options to EasyInstall, however, take precedence even over an existing zip-safe or not-zip-safe file.)

Note that these flag files appear only in .egg files generated by bdist_egg, and in .egg directories created by unpacking such an .egg file.

### top_level.txt – Conflict Management Metadata

This file is a list of the top-level module or package names provided by the project, one Python identifier per line.

Subpackages are not included; a project containing both a foo.bar and a foo.baz would include only one line, foo, in its top_level.txt.

This data is used by pkg_resources at runtime to issue a warning if an egg is added to sys.path when its contained packages may have already been imported.

(It was also once used to detect conflicts with non-egg packages at installation time, but in more recent versions, setuptools installs eggs in such a way that they always override non-egg packages, thus preventing a problem from arising.)

### SOURCES.txt – Source Files Manifest

This file is roughly equivalent to the distutils' MANIFEST file. The differences are as follows:

- The filenames always use / as a path separator, which must be converted back to a platform-specific path whenever they are read.
- The file is automatically generated by setuptools whenever the `egg_info` or `sdist` commands are run, and it is *not* user-editable.

Although this metadata is included with distributed eggs, it is not actually used at runtime for any purpose. Its function is to ensure that setuptools-built *source* distributions can correctly discover what files are part of the project's source, even if the list had been generated using revision control metadata on the original author's system.

In other words, `SOURCES.txt` has little or no runtime value for being included in distributed eggs, and it is possible that future versions of the `bdist_egg` and `install_egg_info` commands will strip it before installation or distribution. Therefore, do not rely on its being available outside of an original source directory or source distribution.

### 7.2.3 Other Technical Considerations

#### Zip File Issues

Although zip files resemble directories, they are not fully substitutable for them. Most platforms do not support loading dynamic link libraries contained in zipfiles, so it is not possible to directly import C extensions from `.egg` zipfiles. Similarly, there are many existing libraries – whether in Python or C – that require actual operating system filenames, and do not work with arbitrary "file-like" objects or in-memory strings, and thus cannot operate directly on the contents of zip files.

To address these issues, the `pkg_resources` module provides a "resource API" to support obtaining either the contents of a resource, or a true operating system filename for the resource. If the egg containing the resource is a directory, the resource's real filename is simply returned. However, if the egg is a zipfile, then the resource is first extracted to a cache directory, and the filename within the cache is returned.

The cache directory is determined by the `pkg_resources` API; please see the `set_cache_path()` and `get_default_cache()` documentation for details.

##### The Extraction Process

Resources are extracted to a cache subdirectory whose name is based on the enclosing `.egg` filename and the path to the resource. If there is already a file of the correct name, size, and timestamp, its filename is returned to the requester. Otherwise, the desired file is extracted first to a temporary name generated using `mkstemp(".$extract",target_dir)`, and then its timestamp is set to match the one in the zip file, before renaming it to its final name. (Some collision detection and resolution code is used to handle the fact that Windows doesn't overwrite files when renaming.)

If a resource directory is requested, all of its contents are recursively extracted in this fashion, to ensure that the directory name can be used as if it were valid all along.

If the resource requested for extraction is listed in the `native_libs.txt` or `eager_resources.txt` metadata files, then *all* resources listed in *either* file will be extracted before the requested resource's filename is returned, thus ensuring that all C extensions and data used by them will be simultaneously available.

##### Extension Import Wrappers

Since Python's built-in zip import feature does not support loading C extension modules from zipfiles, the setuptools `bdist_egg` command generates special import wrappers to make it work.

The wrappers are `.py` files (along with corresponding `.pyc` and/or `.pyo` files) that have the same module name as the corresponding C extension. These wrappers are located in the same package directory (or top-level directory)

within the zipfile, so that say, `foomodule.so` will get a corresponding `foo.py`, while `bar/baz.pyd` will get a corresponding `bar/baz.py`.

These wrapper files contain a short stanza of Python code that asks `pkg_resources` for the filename of the corresponding C extension, then reloads the module using the obtained filename. This will cause `pkg_resources` to first ensure that all of the egg's C extensions (and any accompanying "eager resources") are extracted to the cache before attempting to link to the C library.

Note, by the way, that `.egg` directories will also contain these wrapper files. However, Python's default import priority is such that C extensions take precedence over same-named Python modules, so the import wrappers are ignored unless the egg is a zipfile.

### Installation and Path Management Issues

Python's initial setup of `sys.path` is very dependent on the Python version and installation platform, as well as how Python was started (i.e., script vs. `-c` vs. `-m` vs. interactive interpreter). In fact, Python also provides only two relatively robust ways to affect `sys.path` outside of direct manipulation in code: the `PYTHONPATH` environment variable, and `.pth` files.

However, with no cross-platform way to safely and persistently change environment variables, this leaves `.pth` files as EasyInstall's only real option for persistent configuration of `sys.path`.

But `.pth` files are rather strictly limited in what they are allowed to do normally. They add directories only to the *end* of `sys.path`, after any locally-installed `site-packages` directory, and they are only processed *in* the `site-packages` directory to start with.

This is a double whammy for users who lack write access to that directory, because they can't create a `.pth` file that Python will read, and even if a sympathetic system administrator adds one for them that calls `site.addsitedir()` to allow some other directory to contain `.pth` files, they won't be able to install newer versions of anything that's installed in the systemwide `site-packages`, because their paths will still be added *after* `site-packages`.

So EasyInstall applies two workarounds to solve these problems.

The first is that EasyInstall leverages `.pth` files' "import" feature to manipulate `sys.path` and ensure that anything EasyInstall adds to a `.pth` file will always appear before both the standard library and the local `site-packages` directories. Thus, it is always possible for a user who can write a Python-read `.pth` file to ensure that their packages come first in their own environment.

Second, when installing to a `PYTHONPATH` directory (as opposed to a "site" directory like `site-packages`) EasyInstall will also install a special version of the `site` module. Because it's in a `PYTHONPATH` directory, this module will get control before the standard library version of `site` does. It will record the state of `sys.path` before invoking the "real" `site` module, and then afterwards it processes any `.pth` files found in `PYTHONPATH` directories, including all the fixups needed to ensure that eggs always appear before the standard library in sys.path, but are in a relative order to one another that is defined by their `PYTHONPATH` and `.pth`-prescribed sequence.

The net result of these changes is that `sys.path` order will be as follows at runtime:

1. The `sys.argv[0]` directory, or an empty string if no script is being executed.
2. All eggs installed by EasyInstall in any `.pth` file in each `PYTHONPATH` directory, in order first by `PYTHONPATH` order, then normal `.pth` processing order (which is to say alphabetical by `.pth` filename, then by the order of listing within each `.pth` file).
3. All eggs installed by EasyInstall in any `.pth` file in each "site" directory (such as `site-packages`), following the same ordering rules as for the ones on `PYTHONPATH`.
4. The `PYTHONPATH` directories themselves, in their original order
5. Any paths from `.pth` files found on `PYTHONPATH` that were *not* eggs installed by EasyInstall, again following the same relative ordering rules.

6. The standard library and "site" directories, along with the contents of any `.pth` files found in the "site" directories.

Notice that sections 1, 4, and 6 comprise the "normal" Python setup for `sys.path`. Sections 2 and 3 are inserted to support eggs, and section 5 emulates what the "normal" semantics of `.pth` files on `PYTHONPATH` would be if Python natively supported them.

For further discussion of the tradeoffs that went into this design, as well as notes on the actual magic inserted into `.pth` files to make them do these things, please see also the following messages to the distutils-SIG mailing list:

- http://mail.python.org/pipermail/distutils-sig/2006-February/006026.html
- http://mail.python.org/pipermail/distutils-sig/2006-March/006123.html

#### Script Wrappers

EasyInstall never directly installs a project's original scripts to a script installation directory. Instead, it writes short wrapper scripts that first ensure that the project's dependencies are active on sys.path, before invoking the original script. These wrappers have a #! line that points to the version of Python that was used to install them, and their second line is always a comment that indicates the type of script wrapper, the project version required for the script to run, and information identifying the script to be invoked.

The format of this marker line is:

```
"# EASY-INSTALL-" script_type ": " tuple_of_strings "\n"
```

The `script_type` is one of `SCRIPT`, `DEV-SCRIPT`, or `ENTRY-SCRIPT`. The `tuple_of_strings` is a comma-separated sequence of Python string constants. For `SCRIPT` and `DEV-SCRIPT` wrappers, there are two strings: the project version requirement, and the script name (as a filename within the `scripts` metadata directory). For `ENTRY-SCRIPT` wrappers, there are three: the project version requirement, the entry point group name, and the entry point name. (See the "Automatic Script Creation" section in the setuptools manual for more information about entry point scripts.)

In each case, the project version requirement string will be a string parseable with the `pkg_resources` modules' `Requirement.parse()` classmethod. The only difference between a `SCRIPT` wrapper and a `DEV-SCRIPT` is that a `DEV-SCRIPT` actually executes the original source script in the project's source tree, and is created when the "setup.py develop" command is run. A `SCRIPT` wrapper, on the other hand, uses the "installed" script written to the `EGG-INFO/scripts` subdirectory of the corresponding `.egg` zipfile or directory. (`.egg-info` eggs do not have script wrappers associated with them, except in the "setup.py develop" case.)

The purpose of including the marker line in generated script wrappers is to facilitate introspection of installed scripts, and their relationship to installed eggs. For example, an uninstallation tool could use this data to identify what scripts can safely be removed, and/or identify what scripts would stop working if a particular egg is uninstalled.

## 7.3 Release Process

In order to allow for rapid, predictable releases, Setuptools uses a mechanical technique for releases. The release script, `release.py` in the repository, defines the details of the releases, and is executed by the jaraco.packaging release module. The script does some checks (some interactive) and fully automates the release process.

A Setuptools release manager must have maintainer access on PyPI to the project and administrative access to the Bitbucket project.

To make a release, run the following from a Mercurial checkout at the revision slated for release:

```
python -m jaraco.packaging.release
```

### 7.3.1 Release Managers

Jason R. Coombs is the primary release manager. Additionally, the following people have access to create releases:

- Matthew Iversen (Ivoz)

CHAPTER 8

# Merge with Distribute

In 2013, the fork of Distribute was merged back into Setuptools. This document describes some of the details of the merge.

## 8.1 Setuptools/Distribute Merge FAQ

### 8.1.1 How do I upgrade from Distribute?

Distribute specifically prohibits installation of Setuptools 0.7 from Distribute 0.6. There are then two options for upgrading.

Note that after upgrading using either technique, the only option to downgrade to either version is to completely uninstall Distribute and Setuptools 0.7 versions before reinstalling an 0.6 release.

#### Use Distribute 0.7

The PYPA has put together a compatibility wrapper, a new release of Distribute version 0.7. This package will install over Distribute 0.6.x installations and will replace Distribute with a simple wrapper that requires Setuptools 0.7 or later. This technique is experimental, but initial results indicate this technique is the easiest upgrade path.

#### Uninstall

First, completely uninstall Distribute. Since Distribute does not have an automated installation routine, this process is manual. Follow the instructions in the README for uninstalling.

### 8.1.2 How do I upgrade from Setuptools 0.6?

There are no special instructions for upgrading over older versions of Setuptools. Simply use *easy_install -U* or run the latest *ez_setup.py*.

### 8.1.3 Where does the merge occur?

The merge is occurring between the heads of the default branch of Distribute and the setuptools-0.6 branch of Setuptools. The Setuptools SVN repo has been converted to a Mercurial repo hosted on Bitbucket. The work is still underway, so the exact changesets included may change, although the anticipated merge targets are Setuptools at 0.6c12 and Distribute at 0.6.35.

### 8.1.4 What happens to other branches?

Distribute 0.7 was abandoned long ago and won't be included in the resulting code tree, but may be retained for posterity in the original repo.

Setuptools default branch (also 0.7 development) may also be abandoned or may be incorporated into the new merged line if desirable (and as resources allow).

### 8.1.5 What history is lost/changed?

As setuptools was not on Mercurial when the fork occurred and as Distribute did not include the full setuptools history (prior to the creation of the setuptools-0.6 branch), the two source trees were not compatible. In order to most effectively communicate the code history, the Distribute code was grafted onto the (originally private) setuptools Mercurial repo. Although this grafting maintained the full code history with names, dates, and changes, it did lose the original hashes of those changes. Therefore, references to changes by hash (including tags) are lost.

Additionally, any heads that were not actively merged into the Distribute 0.6.35 release were also omitted. As a result, the changesets included in the merge repo are those from the original setuptools repo and all changesets ancestral to the Distribute 0.6.35 release.

### 8.1.6 What features will be in the merged code base?

In general, all "features" added in distribute will be included in setuptools. Where there exist conflicts or undesirable features, we will be explicit about what these limitations are. Changes that are backward-incompatible from setuptools 0.6 to distribute will likely be removed, and these also will be well documented.

Bootstrapping scripts (ez_setup/distribute_setup) and docs, as with distribute, will be maintained in the repository and built as part of the release process. Documentation and bootstrapping scripts will be hosted at python.org, as they are with distribute now. Documentation at telecommunity will be updated to refer or redirect to the new, merged docs.

On the whole, the merged setuptools should be largely compatible with the latest releases of both setuptools and distribute and will be an easy transition for users of either library.

### 8.1.7 Who is invited to contribute? Who is excluded?

While we've worked privately to initiate this merge due to the potential sensitivity of the topic, no one is excluded from this effort. We invite all members of the community, especially those most familiar with Python packaging and its challenges to join us in the effort.

We have lots of ideas for how we'd like to improve the codebase, release process, everything. Like distribute, the post-merge setuptools will have its source hosted on Bitbucket. (So if you're currently a distribute contributor, about the only thing that's going to change is the URL of the repository you follow.) Also like distribute, it'll support Python 3, and hopefully we'll soon merge Vinay Sajip's patches to make it run on Python 3 without needing 2to3 to be run on the code first.

While we've worked privately to initiate this merge due to the potential sensitivity of the topic, no one is excluded from this effort. We invite all members of the community, especially those most familiar with Python packaging and its challenges to join us in the effort.

### 8.1.8 Why Setuptools and not Distribute or another name?

We do, however, understand that this announcement might be unsettling for some. The setuptools name has been subjected to a lot of deprecation in recent years, so the idea that it will now be the preferred name instead of distribute

might be somewhat difficult or disorienting for some. We considered use of another name (Distribute or an entirely new name), but that would serve to only complicate matters further. Instead, our goal is to simplify the packaging landscape but without losing any hard-won advancements. We hope that the people who worked to spread the first message will be equally enthusiastic about spreading the new one, and we especially look forward to seeing the new posters and slogans celebrating setuptools.

### 8.1.9 What is the timeframe of release?

There are no hard timeframes for any of this effort, although progress is underway and a draft merge is underway and being tested privately. As an unfunded volunteer effort, our time to put in on it is limited, and we've both had some recent health and other challenges that have made working on this difficult, which in part explains why we haven't met our original deadline of a completed merge before PyCon.

The final Setuptools 0.7 was cut on June 1, 2013 and will be released to PyPI shortly thereafter.

### 8.1.10 What version number can I expect for the new release?

The new release will roughly follow the previous trend for setuptools and release the new release as 0.7. This number is somewhat arbitrary, but we wanted something other than 0.6 to distinguish it from its ancestor forks but not 1.0 to avoid putting too much emphasis on the release itself and to focus on merging the functionality. In the future, the project will likely adopt a versioning scheme similar to semver to convey semantic meaning about the release in the version number.

## 8.2 Process

In order to try to accurately reflect the fork and then re-merge of the projects, the merge process brought both code trees together into one repository and grafted the Distribute fork onto the Setuptools development line (as if it had been created as a branch in the first place).

The rebase to get distribute onto setuptools went something like this:

```
hg phase -d -f -r 26b4c29b62db
hg rebase -s 26b4c29b62db -d 7a5cf59c78d7
```

The technique required a late version of mercurial (2.5) to work correctly.

The only code that was included was the code that was ancestral to the public releases of Distribute 0.6. Additionally, because Setuptools was not hosted on Mercurial at the time of the fork and because the Distribute fork did not include a complete conversion of the Setuptools history, the Distribute changesets had to be re-applied to a new, different conversion of the Setuptools SVN repository. As a result, all of the hashes have changed.

Distribute was grafted in a 'distribute' branch and the 'setuptools-0.6' branch was targeted for the merge. The 'setuptools' branch remains with unreleased code and may be incorporated in the future.

## 8.3 Reconciling Differences

There were both technical and philosophical differences between Setuptools and Distribute. To reconcile these differences in a manageable way, the following technique was undertaken:

Create a 'Setuptools-Distribute merge' branch, based on a late release of Distribute (0.6.35). This was done with a00b441856c4.

In that branch, first remove code that is no longer relevant to Setuptools (such as the setuptools patching code).

Next, in the the merge branch, create another base from at the point where the fork occurred (such that the code is still essentially an older but pristine setuptools). This base can be found as 955792b069d0. This creates two heads in the merge branch, each with a basis in the fork.

Then, repeatedly copy changes for a single file or small group of files from a late revision of that file in the 'setuptools-0.6' branch (1aae1efe5733 was used) and commit those changes on the setuptools-only head. That head is then merged with the head with Distribute changes. It is in this Mercurial merge operation that the fundamental differences between Distribute and Setuptools are reconciled, but since only a single file or small set of files are used, the scope is limited.

Finally, once all the challenging files have been reconciled and merged, the remaining changes from the setuptools-0.6 branch are merged, deferring to the reconciled changes (a1fa855a5a62 and 160ccaa46be0).

Originally, jaraco attempted all of this using anonymous heads in the Distribute branch, but later realized this technique made for a somewhat unclear merge process, so the changes were re-committed as described above for clarity. In this way, the "distribute" and "setuptools" branches can continue to track the official Distribute changesets.

## 8.4 Concessions

With the merge of Setuptools and Distribute, the following concessions were made:

Differences from setuptools 0.6c12:

### 8.4.1 Major Changes

- Python 3 support.
- Improved support for GAE.
- Support PEP-370 per-user site packages.
- Sort order of Distributions in pkg_resources now prefers PyPI to external links (Distribute issue 163).
- Python 2.4 or greater is required (drop support for Python 2.3).

### 8.4.2 Minor Changes

- Wording of some output has changed to replace contractions with their canonical form (i.e. prefer "could not" to "couldn't").
- Manifest files are only written for 32-bit .exe launchers.

Differences from Distribute 0.6.36:

### 8.4.3 Major Changes

- The _distribute property of the setuptools module has been removed.
- Distributions are once again installed as zipped eggs by default, per the rationale given in the seminal bug report indicates that the feature should remain and no substantial justification was given in the Distribute report.

### 8.4.4 Minor Changes

- The patch for #174 has been rolled-back, as the comment on the ticket indicates that the patch addressed a symptom and not the fundamental issue.
- `easy_install` (the command) once again honors setup.cfg if found in the current directory. The "misbehavior" characterized in #99 is actually intended behavior, and no substantial rationale was given for the deviation.