

# Writing a Package in Python

This article by **Tarek Ziadé** focuses on a repeatable process to write and release Python packages. We will focus on how to install, uninstall, develop, test, register, and upload a package.

## Writing a Package

Its intents are:

- To shorten the time needed to set up everything before starting the real work, in other words the boiler-plate code
- To provide a standardized way to write packages
- To ease the use of a test-driven development approach
- To facilitate the releasing process

It is organized in the following four parts:

- A **common pattern** for all packages that describes the similarities between all Python packages, and how *distutils* and *setuptools* play a central role
- How **generative programming** ([http://en.wikipedia.org/wiki/Generative\\_programming](http://en.wikipedia.org/wiki/Generative_programming)) can help this through the template-based approach
- The package template creation, where everything needed to work is set
- Setting up a development cycle

## A Common Pattern for All Packages

The easiest way to organize the code of an application is to split it into several packages using eggs. This makes the code simpler, and easier to understand, maintain, and change. It also maximizes the reusability of each package. They act like components.

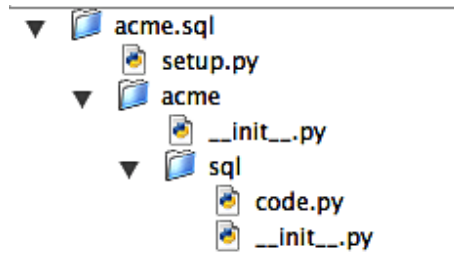
Applications for a given company can have a set of eggs glued together with a master egg.

Therefore, all packages can be built using egg structures.

This section presents how a **namespaced package** is organized, released, and distributed to the world through *distutils* and *setuptools*.

Writing an egg is done by layering the code in a nested folder that provides a common prefix namespace. For instance, for the Acme company, the common namespace can be `acme`. The result is a namespaced package.

For example, a package whose code relates to SQL can be called `acme.sql`. The best way to work with such a package is to create an `acme.sql` folder that contains the `acme` and then the `sql` folder:



## setup.py, the Script That Controls Everything

The root folder contains a *setup.py* script, which defines all metadata as described in the *distutils module*, combined as arguments in a call to the standard *setup* function. This function was extended by the third-party library *setuptools* that provides most of the egg infrastructure.

*The boundary between distutils and setuptools is getting fuzzy, and they might merge one day.*

Therefore, the minimum content for this file is:

```
from setuptools import setup
setup(name='acme.sql')
```

*name* gives the full name of the egg. From there, the script provides several commands that can be listed with the *-help-commands* option.

```
$ python setup.py --help-commands
Standard commands:
  build          build everything needed to install
  ...
  install        install everything from build directory
  sdist          create a source distribution
  register       register the distribution
  bdist          create a built (binary) distribution
Extra commands:
  develop        install package in 'development mode'
  ...
  test           run unit tests after in-place build
  alias          define a shortcut
  bdist_egg      create an "egg" distribution
```

The most important commands are the ones left in the preceding listing. **Standard commands** are the built-in commands provided by *distutils*, whereas **Extra commands** are the ones created by third-party packages such as *setuptools* or any other package that defines and registers a new command.

### **sdist**

The *sdist* command is the simplest command available. It creates a release tree where everything needed to run the package is copied. This tree is then archived in one or many archived files (often, it just creates one tar ball). The archive is basically a copy of the source tree.

This command is the easiest way to distribute a package from the target system independently. It creates a *dist* folder with the archives in it that can be distributed. To be able to use it, an extra argument has to be passed to *setup* to provide a version number. If you don't give it a *version* value, it will use *version = 0.0.0*:

```
from setuptools import setup
setup(name='acme.sql', version='0.1.1')
```

This number is useful to upgrade an installation. Every time a package is released, the number is raised so that the target system knows it has changed.

Let's run the *sdist* command with this extra argument:

```
$ python setup.py sdist
running sdist
...
creating dist
tar -cf dist/acme.sql-0.1.1.tar acme.sql-0.1.1
gzip -f9 dist/acme.sql-0.1.1.tar
removing 'acme.sql-0.1.1' (and everything under it)
$ ls dist/
acme.sql-0.1.1.tar.gz
```

*Under Windows, the archive will be a ZIP file.*

The version is used to mark the name of the archive, which can be distributed and installed on any system having Python. In the *sdist* distribution, if the package contains C libraries or extensions, the target system is responsible for compiling them. This is very common for Linux-based systems or Mac OS because they commonly provide a compiler. But it is less usual to have it under Windows. That's why a package should always be distributed with a pre-built distribution as well, when it is intended to run under several platforms.

## The MANIFEST.in File

When building a distribution with *sdist*, *distutils* browse the package directory looking for files to include in the archive.

*distutils* will include:

- All Python source files implied by the *py\_modules*, *packages*, and *scripts* option
- All C source files listed in the *ext\_modules* option
- Files that match the glob pattern *test/test\*.py*
- README, *README.txt*, *setup.py*, and *setup.cfg* files

Besides, if your package is under Subversion or CVS, *sdist* will browse folders such as *.svn* to look for files to include. *sdist* builds a *MANIFEST* file that lists all files and includes them into the archive.

Let's say you are not using these version control systems, and need to include more files. Now, you can define a template called *MANIFEST.in* in the same directory as that of *setup.py* for the *MANIFEST* file, where you indicate to *sdist* which files to include.

This template defines one inclusion or exclusion rule per line, for example:

```
include HISTORY.txt
include README.txt
include CHANGES.txt
include CONTRIBUTORS.txt
include LICENSE
recursive-include *.txt *.py
```

The full list of commands is available at <http://docs.python.org/dist/sdist-cmd.html#sdist-cmd>.

## build and bdist

To be able to distribute a pre-built distribution, *distutils* provide the *build* command, which compiles the package in four steps:

- *build\_py*: Builds pure Python modules by byte-compiling them and copying them into the build folder.
- *build\_clib*: Builds C libraries, when the package contains any, using Python compiler and creating a static library in the build folder.
- *build\_ext*: Builds C extensions and puts the result in the build folder like *build\_clib*.
- *build\_scripts*: Builds the modules that are marked as scripts. It also changes the interpreter path when the first line was set (!#) and fixes the file mode so that it is executable.

Each of these steps is a command that can be called independently. The result of the compilation process is a build folder that contains everything needed for the package to be installed. There's no cross-compiler option yet in the *distutils* package. This means that the result of the command is always specific to the system it was build on.

*Some people have recently proposed patches in the Python tracker to make distutils able to cross-compile the C parts. So this feature might be available in the future.*

When some C extensions have to be created, the build process uses the system compiler and the Python header file (*Python.h*). This include file is available from the time Python was built from the sources. For a packaged distribution, an extra package called *python-dev* often contains it, and has to be installed as well.

The C compiler used is the system compiler. For Linux-based system or Mac OS X, this would be **gcc**. For Windows, Microsoft Visual C++ can be used (there's a free command-line version available) and the open-source project MinGW as well. This can be configured in *distutils*.

The *build* command is used by the *bdist* command to build a binary distribution. It calls *build* and all dependent commands, and then creates an archive in the same way as *sdist* does.

Let's create a binary distribution for *acme.sql* under Mac OS X:

```
$ python setup.py bdist
running bdist
running bdist_dumb
running build
...
running install_scripts
tar -cf dist/acme.sql-0.1.1.macosx-10.3-fat.tar .
gzip -f9 acme.sql-0.1.1.macosx-10.3-fat.tar
removing 'build/bdist.macosx-10.3-fat/dumb' (and everything under it)
$ ls dist/
acme.sql-0.1.1.macosx-10.3-fat.tar.gz  acme.sql-0.1.1.tar.gz
```

Notice that the newly created archive's name contains the name of the system and the distribution it was built under (*Mac OS X 10.3*).

The same command called under Windows will create a specific distribution archive:

```
C:acme.sql> python.exe setup.py bdist
...
C:acme.sql> dir dist
25/02/2008  08:18      <DIR>          .
25/02/2008  08:18      <DIR>          ..
25/02/2008  08:24                16 055 acme.sql-0.1.win32.zip
           1 File(s)                16 055 bytes
           2 Dir(s)    22 239 752 192 bytes free
```

If a package contains C code, apart from a source distribution, it's important to release as many different binary distributions as possible. At the very least, a Windows binary distribution is important for those who don't have a C compiler installed.

A binary release contains a tree that can be copied directly into the Python tree. It mainly contains a folder that is copied into Python's *site-packages* folder.

## **bdist\_egg**

The *bdist\_egg* command is an extra command provided by *setuptools*. It basically creates a binary distribution like *bdist*, but with a tree comparable to the one found in the source distribution. In other words, the archive can be downloaded, uncompressed, and used as it is by adding the folder to the Python search path (*sys.path*).

These days, this distribution mode should be used instead of the *bdist*-generated one.

## **install**

The *install* command installs the package into Python. It will try to build the package if no previous build was made and then inject the result into the Python tree. When a source

distribution is provided, it can be uncompressed in a temporary folder and then installed with this command. The *install* command will also install dependencies that are defined in the *install\_requires* metadata.

This is done by looking at the packages in the Python Package Index (PyPI). For instance, to install *pysqlite* and *SQLAlchemy* together with *acme.sql*, the setup call can be changed to:

```
from setuptools import setup
setup(name='acme.sql', version='0.1.1',
      install_requires=['pysqlite', 'SQLAlchemy'])
```

When we run the command, both dependencies will be installed.

---

**This article is extracted from:**  
**[Expert Python Programming](#)**

Best practices for designing, coding, and distributing your Python software



- Learn Python development best practices from an expert, with detailed coverage of naming and coding conventions
- Apply object-oriented principles, design patterns, and advanced syntax tricks
- Manage your code with distributed version control
- Profile and optimize your code
- Practice test-driven development and continuous integration

For more information, please visit:

<http://www.PacktPub.com/expert-python-programming/book>

---

## How to Uninstall a Package

The command to uninstall a previously installed package is missing in *setup.py*. This feature was proposed earlier too. This is not trivial at all because an installer might change files that are used by other elements of the system.

The best way would be to create a snapshot of all elements that are being changed, and a record of all files and directories created.

A *record* option exists in *install* to record all files that have been created in a text file:

```
$ python setup.py install --record installation.txt
running install
...
writing list of installed files to 'installation.txt'
```

This will not create any backup on any existing file, so removing the file mentioned might break the system. There are platform-specific solutions to deal with this. For example, *distutils* allow you to distribute the package as an RPM package. But there's no universal way to handle it as yet.

The simplest way to remove a package at this time is to erase the files created, and then remove any reference in the *easy-install.pth* file that is located in the *sitepackages* folder.

## develop

*setuptools* added a useful command to work with the package. The *develop* command builds and installs the package in place, and then adds a simple link into the Python site-packages folder. This allows the user to work with a local copy of the code, even though it's available within Python's *site-packages* folder. All packages that are being created are linked with the *develop* command to the interpreter.

When a package is installed this way, it can be removed specifically with the *-u* option, unlike the regular install:

```
$ sudo python setup.py develop
running develop
...
Adding iw.recipe.fss 0.1.3dev-r7606 to easy-install.pth file
Installed /Users/repos/ingeniweb.sourceforge.net/iw.recipe.fss/trunk
Processing dependencies ...
$ sudo python setup.py develop -u
running develop
Removing
...
Removing iw.recipe.fss 0.1.3dev-r7606 from easy-install.pth file
```

Notice that a package installed with *develop* will always prevail over other versions of the same package installed.

## test

Another useful command is *test*. It provides a way to run all tests contained in the package. It scans the folder and aggregates the test suites it finds. The test runner tries to collect tests in the package but is quite limited. A good practice is to hook an extended test runner such as *zope.testing* or *Nose* that provides more options.

To hook *Nose* transparently to the *test* command, the *test\_suite* metadata can be set to *'nose.collector'* and *Nose* added in the *test\_requires* list:

```
setup(
    ...
    test_suite='nose.collector',
    test_requires=['Nose'],
    ...
)
```

## register and upload

To distribute a package to the world, two commands are available:

- *register*: This will upload all metadata to a server.
- *upload*: This will upload to the server all archives previously built in the *dist* folder.

The main PyPI server, previously named the Cheeseshop, is located at <http://pypi.python.org/pypi> and contains over 3000 packages from the community. It is a default server used by the *distutils* package, and an initial call to the *register* command will generate a *.pypirc* file in your home directory.

Since the PyPI server authenticates people, when changes are made to a package, you will be asked to create a user over there. This can also be done at the prompt:

```
$ python setup.py register
running register
...
We need to know who you are, so please choose either:
  1. use your existing login,
  2. register as a new user,
  3. have the server generate a new password for you (and email it to
you), or
  4. quit
Your selection [default 1]:
```

Now, a *.pypirc* file will appear in your home directory containing the user and password you have entered. These will be used every time *register* or *upload* is called:

```
[server-index]
username: tarek
password: secret
```

*There is a bug on Windows with Python 2.4 and 2.5. The home directory is not found by distutils unless a HOME environment variable is added. But, this has been fixed in 2.6. To add it, use the technique where we modify the PATH variable. Then add a HOME variable for your user that points to the directory returned by os.path.expanduser('~').*

When the *download\_url* metadata or the *url* is specified, and is a valid URL, the PyPI server will make it available to the users on the project web page as well.

Using the *upload* command will make the archive directly available at PyPI, so the *download\_url* can be omitted:

python™

Package Index -> eggchecker 0.1.3dev

PACKAGE INDEX

eggchecker 0.1.3dev

setuptools command extensions to run QA and tests on the code

This is a small set of command extensions for setuptools that allows to run:

- QA tests on your package
- zope.testing

**qa: QA tests**

Runs as long as it has a setup.py on the root. It adds a new command called qa. To run it, go into your package and type:

```
$ python setup.py qa
```

At this time it simply runs pyflakes over the code.

**ztest: zope.testing**

Runs zope.testing over the package:

```
$ python setup.py ztest
```

File	Type	Py Version	Size	# downloads
eggchecker-0.1.3dev-py2.4.egg (md5)	Python Egg	2.4	7KB	62
eggchecker-0.1.3dev-py2.5.egg (md5)	Python Egg	2.5	7KB	43

Distutils defines a Trove categorization (see PEP 301: <http://www.python.org/dev/peps/pep-0301/#distutils-trove-classification>) to classify the packages, such as the one defined at Sourceforge. The trove is a static list that can be found at [http://pypi.python.org/pypi?action=list\\_classifiers](http://pypi.python.org/pypi?action=list_classifiers), and that is augmented from time to time with a new entry.

Each line is composed of levels separated by "::":

```
...
Topic :: Terminals
Topic :: Terminals :: Serial
Topic :: Terminals :: Telnet
Topic :: Terminals :: Terminal Emulators/X Terminals
Topic :: Text Editors Topic :: Text Editors :: Documentation
Topic :: Text Editors :: Emacs
...
```

A package can be classified in several categories, which can be listed in the classifiers meta-data. A GPL package that deals with low-level Python code (for instance) can use:

```
Programming Language :: Python
Topic :: Software Development :: Libraries :: Python Modules
License :: OSI Approved :: GNU General Public License (GPL)
```

## Python 2.6 .pypirc Format

The `.pypirc` file has evolved under Python 2.6, so several users and their passwords can be managed along with several PyPI-like servers. A Python 2.6 configuration file will look somewhat like this:

```
[distutils]
index-servers =
    pypi
    alternative-server
    alternative-account-on-pypi
```

```
[pypi]
username:tarek
password:secret
[alternative-server]
username:tarek
password:secret
repository:http://example.com/pypi
```

The *register* and *upload* commands can pick a server with the help of the *-r* option, using the repository full URL or the section name:

```
# upload to http://example.com/pypi
$ python setup.py sdist upload -r alternative-server
# registers with default account (tarek at pypi)
$ python setup.py register
# registers to http://example.com
$ python setup.py register -r http://example.com/pypi
```

This feature allows interaction with servers other than PyPI. When dealing with a lot of packages that are not to be published at PyPI, a good practice is to run your own PyPI-like server. The Plone Software Center (see <http://plone.org/products/plonesoftwarecenter>) can be used, for example, to deploy a web server that can interact with *distutils* *upload* and *register* commands.

## Creating a New Command

*distutils* allows you to create new commands, as described in <http://docs.python.org/dist/node84.html>. A new command can be registered with an entry point, which was introduced by *setuptools* as a simple way to define packages as plug-ins.

An entry point is a named link to a class or a function that is made available through some APIs in *setuptools*. Any application can scan for all registered packages and use the linked code as a plug-in.

To link the new command, the *entry\_points* metadata can be used in the setup call:

```
setup(name="my.command",
      entry_points="""
        [distutils.commands]
        my_command = my.command.module.Class
      """)
```

All named links are gathered in named sections. When *distutils* is loaded, it scans for links that were registered under *distutils.commands*.

This mechanism is used by numerous Python applications that provide extensibility.

## setup.py Usage Summary

There are three main actions to take with *setup.py*:

- Build a package.

- Install it, possibly in develop mode.
- Register and upload it to PyPI.

Since all the commands can be combined in the same call, some typical usage patterns are:

```
# register the package with PyPI, creates a source and
# an egg distribution, then upload them
$ python setup.py register sdist bdist_egg upload
# installs it in-place, for development purpose
$ python setup.py develop
# installs it
$ python setup.py install
```

## The alias Command

To make the command line work easily, a new command has been introduced by *setuptools* called *alias*. In a file called *setup.cfg*, it creates an alias for a given combination of commands. For instance, a release command can be created to perform all actions needed to upload a source and a binary distribution to PyPI:

```
$ python setup.py alias release register sdist bdist_egg upload
running alias
Writing setup.cfg
$ python setup.py release
...
```

## Other Important Metadata

Besides the name and the version of the package being distributed, the most important arguments *setup* can receive are:

- *description*: A few sentences to describe the package
- *long\_description*: A full description that can be in reStructuredText
- *keywords*: A list of keywords that define the package
- *author*: The author's name or organization
- *author\_email*: The contact email address
- *url*: The URL of the project
- *license*: The license (GPL, LGPL, and so on)
- *packages*: A list of all names in the package; *setuptools* provides a small function called *find\_packages* that calculates this
- *namespace\_packages*: A list of namespaced packages

A completed *setup.py* file for *acme.sql* would be:

```
import os
from setuptools import setup, find_packages
version = '0.1.0'
README = os.path.join(os.path.dirname(__file__), 'README.txt')
long_description = open(README).read() + '\n\n'
setup(name='acme.sql',
      version=version,
```

```
description=("A package that deals with SQL, "
            "from ACME inc"),
long_description=long_description,
classifiers=[
    "Programming Language :: Python",
    ("Topic :: Software Development :: Libraries ::
     Python Modules"),
],
keywords='acme sql',
author='Tarek',
author_email='tarek@ziade.org',
url='http://ziade.org',
license='GPL',
packages=find_packages(),
namespace_packages=['acme'],
install_requires=['pysqlite', 'SQLAlchemy']
)
```

*The two comprehensive guides to keep under your pillow are: The distutils guide at <http://docs.python.org/dist/dist.html> The setuptools guide at <http://peak.telecommunity.com/DevCenter/setuptools>*

---

**This article is extracted from:**  
**[Expert Python Programming](#)**

Best practices for designing, coding, and distributing your Python software



- Learn Python development best practices from an expert, with detailed coverage of naming and coding conventions
- Apply object-oriented principles, design patterns, and advanced syntax tricks
- Manage your code with distributed version control
- Profile and optimize your code
- Practice test-driven development and continuous integration

For more information, please visit:

<http://www.PacktPub.com/expert-python-programming/book>

---

## About the Author

**Tarek Ziade** is CTO at Ingeniweb in Paris, working on Python, Zope, and Plone technology and on Quality Assurance. He has been involved for 5 years in the Zope community and has contributed to the Zope code itself.

Tarek has also created Afpy, the French Python User Group and has written two books in French about Python. He has gave numerous talks and tutorials in French and international events like Solutions Linux, Pycon, OSCON, and EuroPython.