Distil Documentation

Release 0.1.2

Vinay Sajip

Sep 27, 2017

Contents

Welcome to the documentation for `distil`, an experimental packaging tool built on top of distlib. With `distil`, you can install, upgrade, uninstall. build, test and package Python distributions, whether pure-Python, or incorporating C libraries and extensions.

CHAPTER 1

Overview

The distil tool is a testbed for distlib, which is a low-level library that implements basic packaging functionality, and is intended for use by packaging tools.

Why distil?

While maintainers of the pip project have expressed an interest in the possibility of implementing some parts of pip functionality using distlib, this is dependent on available volunteer time and overall project priorities, and so no specific timescales for any adoption of particular features of distlib can be expected. Of course, this also applies to packaging tools other than pip.

So, distil aims to fill the gap by providing a packaging tool, based only on distlib and the Python standard library. If it is to demonstrate that distlib is useful for and usable by packaging tools, distil cannot be a toy – it must aim to provide a large part of the functionality of tools like pip, and it must show, where possible, improvements over existing tools that are possible through the use of distlib.

Note: distil is not intended to supplant any existing packaging tool, but rather to act as a testbed for distlib and to explore useful areas in packaging. It should be used in virtual environments set up specifically for the purpose of experimenting with it. Remember, it is alpha software.

Features

The initial release of distil provides the following features:

- Install projects from PyPI and wheels (see PEP 427). Distil does not invoke setup.py, so projects that do significant computation in setup.py may not be installable by distil. However, a large number of projects on PyPI *can* be installed, and dependencies are detected, downloaded and installed.
- Optionally upgrade installed distributions, whether installed by distil or installed by pip.

- Uninstall distributions installed by `distil` or `pip`.
- Install links to local projects which remain editable so that changes are immediately reflected in an environment (like `pip install -e dir`).
- Build source distributions in `.tar.gz`, `.tar.bz2`, and `.zip` formats.
- Build binary distributions in wheel format. These can be pure-Python, or have C libraries and extensions. Support for Cython and Fortran (using `f2py`) is possible, though currently `distil` cannot install `Cython` or `Numpy` because of how they use `setup.py`. You can optionally use `pip` to build wheels, which can then be installed using `distil`.
- Run tests on built distributions.
- Register projects on PyPI.
- Upload distributions to PyPI.
- Upload documentation to http://pythonhosted.org/ (formerly http://packages.python.org/).
- Very simple deployment – just copy `distil.py` to a location on your path, optionally naming it to `distil` on POSIX platforms. There's no need to install `distlib` – it's all included.
- Display dependencies of a distribution – either as a list of what would be downloaded (and a suggested download order), or in Graphviz format suitable for conversion to an image.
- Uses either a system Python or one in a virtual environment, but by default installs to the user site rather than system Python library locations.
- Tab-completion of commands and parameters on Bash-compatible shells.

Logically, packaging activities can be divided into a number of categories or roles:

- *Archiver* – builds source distributions from a source tree
- *Builder* – builds binary distributions from source
- *Installer* – installs source or binary distributions

This version of `distil` incorporates (for convenience) all of the above roles. There is a school of thought which says that that these roles should be fulfilled by separate programs, and that's fine for production quality tools – it's just more convenient for now to have everything in one package for an experimental tool like `distil`.

Actual Improvements

Despite the fact that `distil` is in an alpha stage of development and has received no real-world exposure like the existing go-to packaging tools, it does offer some improvements over them:

- Dependency resolution can be performed without downloading any distributions. Unlike e.g. `pip`, you are told of which additional dependencies will be downloaded and installed, before any download occurs.
- Better information is stored for uninstallation. This allows better feedback to be given to users during uninstallation.
- Dependency checking is done during uninstallation. Say you've installed a distribution A, which pulled in dependencies B and C.
  - If you request an uninstallation of B (or C), `distil` will complain that you can't do this because A needs it.
  - When you uninstall A, you are offered the option to uninstall B and C as well (assuming you didn't install something *else* that depends on B or C, after installing A).

- By default, installation is to the user site and not to the system Python, so you shouldn't need to invoke `sudo` to install distributions for personal use which are not for specific projects/virtual environments.
- There's no need to install `distil` – the exact same script will run with any system Python or any venv (subject to Python version constraints of 2.6, 2.7, 3.2 or greater). There's no need to have umpteen copies of `setuptools`, `distribute` and `pip` lying about! Not that disk space is a particular constraint, of course.

Unlike the `pysetup` tool that is part of `distutils2`, `distil` can install many source distributions that are already on PyPI without the need to migrate them to `setup.cfg` first (or the need to convert them to wheels). This allows a more gradual transition to new packaging norms, by not requiring root-and-branch conversion of existing distributions.

More "actual" improvements can be achieved, dependent on feedback from you!

Possible Improvements

In line with recent thinking in packaging, `distlib` and `distil` rely on a declarative format for distribution metadata, as opposed to the *ad hoc* code approach (i.e. `setup.py`) taken by `distutils`/`setuptools`/`distribute`. This offers a means for different packaging tools to interoperate – because the declarative format is standardised – and this can lead to innovations in packaging and improved features for Python package developers and users. There should be no more "my way or the highway" in Python packaging, which the design of `distutils` has fostered.

Limitations

As already mentioned above, `distil` is not a replacement for `pip` or `easy_install`. The main reason why `distil` cannot do things that `pip` or `easy_install` can do is that some projects use code in `setup.py` to do things like creating new files and moving files around before invoking the actual setup code in `distutils`/`setuptools`/`distribute`. Because `distlib` and `distil` are declarative and don't actually execute `setup.py`, the code therein cannot be taken advantage of, and so `distil` can fail when trying to build or install such projects. However, Python packaging standards are expected to provide a migration path such that such code as currently lives in `setup.py` can be replaced by other mechanisms which allow better interoperability between packaging tools.

Distil *can* use `pip` to build wheels, for those cases where `setup.py` *must* be executed for a correct build. This only exercises the wheel-building functionality of `distlib`, but is provided as a convenience when experimenting with wheels.

There are other areas of functionality which `distil` does not currently provide, such as direct installation from VCS repositories. These *could* be provided, but are regarded as low priority at present.

Some of the metadata used by `distil` is generated automatically, and may not capure certain things accurately such as `2to3` options. This may result in `2to3` being sometimes run when it's not necessary.

Note that `distil` and `distlib` are free of dependencies other than the Python standard library, and are expected to remain so for the foreseeable future. (Although `distil` provides an option to use `pip`, it is not a hard dependency, and `distil` is useful even without `pip` being installed.)

Change Log

0.1.3

Released: Not yet.

0.1.2

Released: 2013-10-18

- This release is based on distlib 0.1.3, which brings support for PEP 426.
- Fixed a bug in pre-release handling which occurred when there were *only* pre-releases available.
- Added support to convert `bdist_wininst` installers to wheels. This is done by using `distil package`, specifying only the wheel format and giving the path to the installer in place of the source directory.
- Added –wheel-version option to package command to allow specifying a wheel version when building wheels.
- Fixed bugs in uninstallation code which occurred only for some installations.
- Fixed a bug in package data installation where data in packages not explicitly named in package-data was missed.
- Updated `distil link` to show existing links when invoked with no arguments.
- When building wheels, native executable launchers are not included, and scripts are not generated from exports.
- Added support for PEP 426 installation hooks.
- Added `--target` argument to `distil install` to allow partial installation to arbitrary directories.
- Added `--prereleases` argument to `distil install`, `distil test`, `distil download` to support inclusion of pre-releases when locating distributions and performing dependency resolution.
- Numerous documentation updates.

0.1.1

Released: 2013-04-30

- Added `distil init` to support creating an initial version of `package.json` metadata, which can then be added to during development.
- Added `distil link` to support "editable" installations, similar to `pip install -e local_dir`.
- Take into account pre-confirmation (`-y`) during uninstallation when dists that are no longer needed are found. These are now removed automatically when `-y` is specified.
- Fixed error in setting up SSL certificate verification, and adjusted PyPI URLs to be `https://` where specified as `http://` in metadata. Successful SSL verification is now logged.
- Added `--local-dists` option to allow local wheels and sdists to be used for installation.
- Fixed a bug in the handling of local archives (e.g. those returned through a configured `DirectoryLocator`). Local archives shouldn't be deleted after unpacking.
- Added `--python-tags` argument to `distil package` when building wheels to configure the tags for the built wheel.
- Added `--no-unpack` option to `distil download`.
- Fixed problem with rollback on error caused by not recording `SHARED` and `RECORD` correctly.
- Fixed bug in writing entry points (EXPORTS) file.
- Use of `2to3` now defaults to `True` when run under 3.x.
- Fixed bug when run in venvs which caused e.g. `dist-packages` to be used instead of `site-packages`.
- Improved error message when run with Python 2.5 (not supported, but this is now clear from the error message).

- Numerous documentation updates.

0.1.0

Released: 2013-03-22

- Initial release.

CHAPTER 2

Installation

To install `distil`, just download this file and copy it to a location on your path. Optionally on POSIX platforms, you can rename it to `distil`. Make sure its executable bit is set.

On Windows, if you have the Python Launcher installed (included in Python 3.3, or available as a standalone installer here), you should be able to invoke it by just typing `distil` on the command-line.

CHAPTER 3

Usage overview

All command options are available by invoking `distil` with the `-h` parameter:

```
$ distil -h
usage: distil [-h] [-n] [-q] [-v] [--version] [-e VENVDIR] [-p INTERPRETER]
              [--local-dists DIRPATH]

              {download,graph,help,init,install,link,list,metadata,package,pip,
↪register,remove,uninstall,test,upload}
              ...

Perform operations on Python distributions.

positional arguments:
  {download,graph,help,init,install,link,list,metadata,package,pip,register,remove,
↪uninstall,test,upload}
                        The available commands
    download            Download a distribution
    graph               Show dependency graph for a distribution
    help                Provide help on a command
    init                Create initial metadata for a project
    install             Install distributions
    link                Link to local projects
    list                List one or all installed distributions
    metadata            Show metadata for a package
    package             Create source distributions or wheels
    pip                 Build wheels using pip
    register            Register a project on PyPI
    remove (uninstall)  Remove (uninstall) one or more distributions
    test                Test a distribution
    upload              Upload a release or documentation to PyPI

optional arguments:
  -h, --help            show this help message and exit
  -n, --dry-run         don't actually do anything
  -q, --quiet           run quietly (turn verbosity down)
```

```
  -v                    run verbosely (turn verbosity up)
  --version             show program's version number and exit
  -e VENVDIR            Run in the specified virtual environment
  -p INTERPRETER        Run with the specified Python interpreter
  --local-dists DIRPATH
                        Specify path to local wheels and sdists
```

You can get help on individual commands by invoking `distil help <command>` or `distil <command> -h`. Each command is covered in its own section below.

To check the version of distil you have, invoke `distil --version`:

```
$ distil --version
Distil Version 0.1.0 (distlib-0.1.1) Copyright (C) 2012-2013 Vinay Sajip.
vcs id: ecbf5a:c5ceac
Python: 2.7

There is NO WARRANTY. This is preliminary software, best used in virtual
environments.
```

Note that abbreviations can be used for commands, as long as they are unambiguous. For example, `distil down` is equivalent to `distil download`.

Pick your Python

By default, `distil` runs using your system Python. You can choose to run with a specific interpreter by invoking `distil` with `-p`, as in the following example:

```
$ distil -p python3.2 --version
Distil Version 0.1.0 (distlib-0.1.1) Copyright (C) 2012-2013 Vinay Sajip.
vcs id: ecbf5a:c5ceac
Python: 3.2

There is NO WARRANTY. This is preliminary software, best used in virtual
environments.
```

Run in a virtual environment

You can also choose to run using the Python interpreter in a virtual environment (venv) by invoking with the `-e` parameter and specifying the root of the venv:

```
$ pyvenv-3.4 /tmp/venv
$ distil -e /tmp/venv --version
Distil Version 0.1.0 (distlib-0.1.1) Copyright (C) 2012-2013 Vinay Sajip.
vcs id: ecbf5a:c5ceac
Python: 3.4

There is NO WARRANTY. This is preliminary software, best used in virtual
environments.
```

Getting started (POSIX)

When experimenting with `distil`, you may want to set up some virtual environments (venvs). This is how I set mine up:

```
mkdir -p $HOME/projects/scratch
cd $HOME/projects/scratch
virtualenv e2
virtualenv -p python3.2 e3
pyvenv-3.3 e33
cp -r e2 e2.backup
cp -r e3 e3.backup
cp -r e33 e33.backup
```

This sequence sets up `e2` and `e3` as `virtualenv` venvs, with `setuptools` / `distribute` and `pip`, and `e33` as a new-style **PEP 405** venv with no `setuptools`, `distribute` or `pip`. It also makes backup copies of the newly-created environments, allowing them to be easily reset to this initial state.

If you don't already have `virtualenv`, you should install it using your distro's package manager. If it's an old version, you may need to specify `--no-site-packages` in the `virtualenv` command to ensure that the venv is isolated from your system Python's libraries.

I also found it useful to define some shell functionality:

```
PROJECT=$HOME/projects/scratch

function use() { source $PROJECT/$1/bin/activate; }
function reuse() {
  echo $1.backup "->" $1 && rm -rf $PROJECT/$1 && cp -R $PROJECT/$1.backup $PROJECT/
↪$1;
}
complete -C `which distil` distil
```

If the above is saved as e.g. `use-distil`, then you can just invoke `source use-distil` to take advantage of its functionality. This sets up tab-completion (only available on Bash-compatible shells) and allows you to activate one of the test venvs or to reset it to its initial state.

You should replace `$HOME/projects/scratch` in the above with whichever directory you want to use.

Getting started (Windows)

The steps for getting started on Windows are analogous to those on POSIX. If you don't have `virtualenv`, follow the instructions on its PyPI page to see how to get it. Once you have it, use commands like this:

```
cd c:\Projects\scratch
virtualenv e2
virtualenv -p \Python32\python.exe e3
xcopy /s /i e2 e2.backup
xcopy /s /i e3 e3.backup
```

You can use the following scripts for `use` and `reuse` functionality:

```
@echo off
rem Put this in use.cmd somewhere in your path
c:\Projects\scratch\%1\Scripts\activate
```

and:

```
#!/usr/bin/env python
# Put this in reuse.py somewhere in your path
import os
import shutil
import sys

def main():
    if len(sys.argv) < 2:
        print('usage: reuse venvdir')
    else:
        rootdir = r'c:\Projects\scratch'
        source = os.path.join(rootdir, sys.argv[1] + '.backup')
        target = os.path.join(rootdir, sys.argv[1])
        if not os.path.isdir(source) or not os.path.isdir(target):
            print('Bad env: %s' % sys.argv[1])
        else:
            print('%s -> %s' % (source, target))
            shutil.rmtree(target)
            shutil.copytree(source, target)

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

You should replace `c:\Projects\scratch` in the above with whichever directory you want to use.

Specifying local directories for locating distributions

You can specify local directories where sdists and wheels are located via the `--local-dists` argument to `distil`, or through the `DISTIL_LOCAL_DISTS` environment variable. This should be a list of one or more directories where sdists and wheels may be found. If there is more than one directory, use the platform path separator (`os.pathsep`) to separate them, as in the following examples – for POSIX:

```
/home/me/distil/wheels
/home/me/distil/wheels:/home/me/distil/sdists
```

and for Windows:

```
C:\Users\Me\distil\wheels
C:\Users\Me\distil\wheels;C:\Users\Me\distil\sdists
```

When things don't go as planned

When `distil` runs commands, it creates a log file called `distil.log` in your home directory. (If you use tab-completion, it will also create a file called `distil-completion.log` in your home directory.) If you encounter a situation where `distil` doesn't behave as expected, and you'd like to provide feedback about this, please raise an issue, and attach these files if you think they might help or it's not obvious what the problem is.

CHAPTER 4

Installing distributions

Where things are installed

User site and virtual environments

By default, `distil` installs in the *user site* (see **PEP 370** for more information). This means that you don't need to e.g. use `sudo` to install packages for your personal use. If you specify a virtual environment using the `-e` flag, installation is done into that environment.

System site-packages

Because of its alpha status, it is not advisable to use `distil` to install into a system-wide location. However, `distil` does support this if you use the `--system` flag when installing, for test purposes. You will typically need administrative privileges (via `sudo` on POSIX systems) to do this.

Arbitrary directories

You can specify an arbitrary directory to install to through a `--target` argument passed to the `install` command. If this is specified, only the library components are installed to that directory: headers, scripts, out-of-package data and package metadata are not installed. This means that such installations cannot be uninstalled, as there is no metadata storing a list of what files were installed.

Installing new distributions

Before installing a distribution, we can check to see what's currently installed (see *Getting information about distributions* for more information about the `list` command):

```
$ distil -e e2 list
pip   1.3.1
```

Currently, only `pip` (and `setuptools`, which doesn't show up in the list) are installed in this venv.

Pure-python distributions

Let's try installing a pure-Python distribution:

```
$ distil -e e2 install sarge
Checking requirements for sarge (0.1) ... done.
The following new packages will be downloaded and installed:
    sarge (0.1)
Downloading sarge-0.1.tar.gz to /tmp/tmpG5RzKC
    35KB @ 175 KB/s 100 % Done: 00:00:00
Unpacking ... done.
Building sarge (0.1) ...
    Running check ...
    Running build_ext ...
    Running build_py ...
    Build completed.
Installing sarge (0.1) ...
    Running install_headers ...
    Running install_lib ...
    Running install_distinfo ...
    Installation completed.
```

As you can see from the above, `distil` employs a set of steps when installing which should be familiar to users of `distribute` and `pip`. However, the implementation is different – there is no use of command classes, for example.

If we again list the installed distributions, we get the expected result:

```
$ distil -e e2 list
sarge 0.1
pip   1.3.1
```

Bootstrapping pip

You can use `distil` to bootstrap `pip`.

Let's see it bootstrapping `pip` on a vanilla machine which doesn't have `pip` installed. First we'll check for `pip` being available already:

```
vinay@nadia-cinnamon ~ $ which pip
vinay@nadia-cinnamon ~ $
```

So, there's no `pip` on this machine. I have a **PEP 370** user site set up at `~/.local`, and it's on my `PATH`:

```
vinay@nadia-cinnamon ~ $ echo $PATH
/home/vinay/.local/bin:/home/vinay/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/
↪bin:/sbin:/bin:/usr/games:/usr/local/games
```

Now let's install `pip` using `distil`:

```
vinay@nadia-cinnamon ~ $ distil install pip
Checking requirements for pip (1.3.1) ... done.
The following new packages will be downloaded and installed:
    pip (1.3.1)
Downloading pip-1.3.1.tar.gz to /tmp/tmpSdyftQ
```

```
    241KB @  45 KB/s 100 % Done: 00:00:05
Unpacking ... done.
Building pip (1.3.1) ...
    Running check ...
    Running build_ext ...
    Running build_py ...
    Build completed.
Installing pip (1.3.1) ...
    Running install_headers ...
    Running install_lib ...
    Running install_scripts ...
    Running install_distinfo ...
    Installation completed.
vinay@nadia-cinnamon ~ $ which pip
/home/vinay/.local/bin/pip
vinay@nadia-cinnamon ~ $ pip --version
pip 1.3.1 from /home/vinay/.local/lib/python2.7/site-packages (python 2.7)
vinay@nadia-cinnamon ~ $
```

Now, let's install using Python 3.2:

```
$ distil -p python3.2 install pip
Checking requirements for pip (1.3.1) ... done.
The following new packages will be downloaded and installed:
    pip (1.3.1)
Downloading pip-1.3.1.tar.gz to /tmp/tmpc6nk63
    241KB @  14 KB/s 100 % Done: 00:00:17
Unpacking ... done.
Building pip (1.3.1) ...
    Running check ...
    Running build_ext ...
    Running build_py ...
    Build completed.
Installing pip (1.3.1) ...
    Running install_headers ...
    Running install_lib ...
    Running install_scripts ...
    Running install_distinfo ...
    Installation completed.
vinay@nadia-cinnamon ~ $ pip --version
pip 1.3.1 from /home/vinay/.local/lib/python3.2/site-packages (python 3.2)
vinay@nadia-cinnamon ~ $
```

Distributions which include C extensions

Let's try installing a distribution which contains a C extension:

```
$ distil -e e2 install simplejson
Checking requirements for simplejson (3.1.0) ... done.
The following new packages will be downloaded and installed:
    simplejson (3.1.0)
Downloading simplejson-3.1.0.tar.gz to /tmp/tmp2aRbHT
    63KB @ 166 KB/s 100 % Done: 00:00:00
Unpacking ... done.
Building simplejson (3.1.0) ...
    Running check ...
```

```
    Running build_ext ...
Building 'simplejson._speedups' extension
gcc -pthread -fno-strict-aliasing -DNDEBUG -g -fwrapv -O2 -Wall -Wstrict-prototypes -
↪D_FORTIFY_SOURCE=2 -g -fstack-protector --param=ssp-buffer-size=4 -Wformat -
↪Werror=format-security -fPIC -I/tmp/tmp2aRbHT/simplejson-3.1.0 -I/usr/include/
↪python2.7 -c -o /tmp/tmp2aRbHT/simplejson-3.1.0/build/temp.linux-x86_64-2.7/
↪simplejson/_speedups.o /tmp/tmp2aRbHT/simplejson-3.1.0/simplejson/_speedups.c
gcc -pthread -shared -Wl,-O1 -Wl,-Bsymbolic-functions -Wl,-Bsymbolic-functions -Wl,-z,
↪relro /tmp/tmp2aRbHT/simplejson-3.1.0/build/temp.linux-x86_64-2.7/simplejson/_
↪speedups.o -L/tmp/tmp2aRbHT/simplejson-3.1.0/build/temp.linux-x86_64-2.7 -L/usr/lib␣
↪-o /tmp/tmp2aRbHT/simplejson-3.1.0/build/lib.linux-x86_64-2.7/simplejson/_speedups.
↪so
    Running build_py ...
    Build completed.
Installing simplejson (3.1.0) ...
    Running install_headers ...
    Running install_lib ...
    Running install_distinfo ...
    Installation completed.
```

Notice that `distil` invoked the C compiler to build `simplejson`'s C extension. This also works on Windows - simply run the command in a "Visual Studio Command Prompt" window:

Upgrading existing installed distributions

By default. `distil` only upgrades distributions if you ask it to. To illustrate, let's install a distribution which has a newer version:

```
$ reuse e2
$ distil -e e2 install "Jinja2 (2.5.5)"
Checking requirements for Jinja2 (2.5.5) ... done.
The following new packages will be downloaded and installed:
    Jinja2 (2.5.5)
Downloading Jinja2-2.5.5.tar.gz to /tmp/tmpLBegg4
    428KB @  28 KB/s 100 % Done: 00:00:15
Unpacking ... done.
Building Jinja2 (2.5.5) ...
    Running check ...
    Running build_ext ...
    Running build_py ...
    Build completed.
Installing Jinja2 (2.5.5) ...
    Running install_headers ...
    Running install_lib ...
    Running install_distinfo ...
    Installation completed.
```

Now, let's not specify the version:

```
$ distil -e e2 install Jinja2
Checking requirements for Jinja2 (2.6) ... done.
The following installed packages will be upgraded:
    Jinja2 (2.5.5) -> Jinja2 (2.6)
Proceed? (y/n)
```

In this case, `distil` assumes you want to upgrade the existing Jinja2 from 2.5.5 to 2.6. If you elect to proceed, the upgrade will be carried out. Instead, let's try installing a distribution which requires Jinja2 and can work with version 2.5.5:

```
$ distil -e e2 install Flask
Checking requirements for Flask (0.9) ... done.
The following new packages will be downloaded and installed:
    Werkzeug (0.8.3) [for Flask]
    Flask (0.9)
Proceed? (y/n)
```

As you can see, `distil` only offers to install the additional dependency – Werkzeug – and to use the existing Jinja2 installation. If you want to upgrade to the latest version of Jinja2, you can invoke:

```
$ distil -e e2 install --upgrade Flask
Checking requirements for Flask (0.9) ... done.
The following new packages will be downloaded and installed:
    Werkzeug (0.8.3) [for Flask]
    Flask (0.9)
The following installed packages will be upgraded:
    Jinja2 (2.5.5) -> Jinja2 (2.6) [for Flask]
Proceed? (y/n)
```

Now, `distil` offers to upgrade Jinja2 to the latest version, as well as downloading and installing Werkzeug. If we confirm that we want to proceed, `distil` does the expected thing:

```
Downloading Werkzeug-0.8.3.tar.gz to /tmp/tmpWZ7TB6 [for Flask]
    1082KB @  56 KB/s 100 % Done: 00:00:19
Unpacking ... done.
Downloading Jinja2-2.6.tar.gz to /tmp/tmp1SMzzt [for Flask]
    380KB @ 131 KB/s 100 % Done: 00:00:02
Unpacking ... done.
Downloading Flask-0.9.tar.gz to /tmp/tmpuhUOcp
    470KB @  52 KB/s 100 % Done: 00:00:09
Unpacking ... done.
Building Werkzeug (0.8.3) ...
    Running check ...
    Running build_ext ...
    Running build_py ...
    Build completed.
Building Jinja2 (2.6) ...
    Running check ...
    Running build_ext ...
    Running build_py ...
    Build completed.
Building Flask (0.9) ...
    Running check ...
    Running build_ext ...
    Running build_py ...
    Build completed.
Installing Werkzeug (0.8.3) ...
    Running install_headers ...
    Running install_lib ...
    Running install_distinfo ...
    Installation completed.
Installing Jinja2 (2.6) ...
    Running install_headers ...
    Running install_lib ...
```

```
    Running install_distinfo ...
    Installation completed.
Installing Flask (0.9) ...
    Running install_headers ...
    Running install_lib ...
    Running install_distinfo ...
    Installation completed.
```

Listing the distributions shows the expected results:

```
$ distil -e e2 list
Werkzeug 0.8.3
Flask    0.9
Jinja2   2.6
pip      1.3.1
```

Dependency handling

As you see from the above examples, `distil` automatically detects a distribution's dependencies and downloads and installs them, after getting confirmation from the user. (No confirmation is requested if only a single distribution – the one requested by the user – would be downloaded and installed.)

Installing from wheels

To install from a wheel, you just specify the path of the wheel when you invoke the `install` command:

```
$ distil -e e2 install /tmp/Flask-0.9-py27-none-any.whl
Checking requirements for Flask (0.9) ... done.
The following new packages will be downloaded and installed:
    Jinja2 (2.6) [for Flask]
    Werkzeug (0.8.3) [for Flask]
The following new packages will be installed from a local location:
    Flask (0.9)
Proceed? (y/n)
```

As with distributions installed by name, any dependencies declared in a wheel will be downloaded and installed.

See *Binary distributions* for information on how to build wheels.

Using requirements files

As well as specifying individual distributions on the command line, you can also specify requirements files, which have the requirements:

```
$ cat reqts.txt
Flask
Sphinx
$ distil -e e2 install -r reqts.txt
Checking requirements for Flask (0.9) ... done.
Checking requirements for Sphinx (1.1.3) ... done.
The following new packages will be downloaded and installed:
```

```
    Jinja2 (2.6) [for Sphinx, Flask]
    Pygments (1.6) [for Sphinx]
    docutils (0.10) [for Sphinx]
    Werkzeug (0.8.3) [for Flask]
    Sphinx (1.1.3)
    Flask (0.9)
Proceed? (y/n)
```

Installing just library components

You can install to an arbitrary directory rather than a `site-packages` location by specifying `--target`:

```
$ distil install --target /tmp/scratch sarge
```

The directory must exist. If given, only library components are written to it: headers, scripts, out-of-package data and package metadata are not written. If you use this option, you will not be able to uninstall from the target location (since no metadata is written indicating what was installed).

Finding pre-releases during dependency resolution

As per **PEP 426**, `distil` will return pre-releases if they are all that is available. To specify that pre-releases are included when doing dependency resolution (i.e. treated like normal releases), you can specify `--prereleases`. This flag works with all commands which do dependency resolution and location of distributions.

Installing from version control systems

Currently, `distil` does not support installing from VCS repositories. This is not hard to do – you just check out an appropriate revision to a temporary directory, and install from there – but installation requires a `package.json` to be present (to indicate source/data files, build options etc.). Most projects will not have this file (yet), making the install-from-VCS feature one of limited usefulness.

Command line reference – `distil install`

Here is the complete help for `distil`'s `install` command:

```
$ distil help install
usage: distil install [-h] [-y] [-u] [--prereleases] [-s] [-t DESTDIR]
                      [-r REQTFILE [REQTFILE ...]]
                      [REQT [REQT ...]]

Install one or more distributions.

positional arguments:
  REQT                  A requirement using a distribution on PyPI, or the
                        path of a wheel file.

optional arguments:
  -h, --help            show this help message and exit
```

```
  -y, --yes             Don't ask for confirmation before installing.
  -u, --upgrade         Upgrade dependencies if possible. The default
                        behaviour is to upgrade dependencies only if
                        necessary.
  --prereleases         Include pre-releases when installing. By default, pre-
                        releases are skipped, unless they are all that is
                        available.
  -s, --system          Install to system Python (may require sufficient
                        privileges). By default, installations are written to
                        the user site (~/.local), or to a virtual environment
                        specified with -e.
  -t DESTDIR, --target DESTDIR
                        Install modules and packages to the specified
                        directory. Scripts, headers, non-package data and
                        package metadata are not installed. This means that
                        you can't uninstall anything installed in this way.
  -r REQTFILE [REQTFILE ...]
                        Get requirements from specified file(s)
```

CHAPTER 5

Uninstalling distributions

Assume we've installed Flask as shown in the section *Upgrading existing installed distributions*. To uninstall distributions, you can invoke the `uninstall` command (which has an alias, `remove`) as in the following example:

```
$ distil -e e2 remove werkzeug
Removal cannot proceed: other software needs what you are trying to remove:
    Flask (0.9)
```

As you can see, `distil` correctly stated that Werkzeug can't be removed, as it would leave Flask in a non-working state. The same thing happens with the other Flask dependency, Jinja2:

```
$ distil -e e2 remove jinja2
Removal cannot proceed: other software needs what you are trying to remove:
    Flask (0.9)
```

However, if we try to uninstall Flask, `distil` doesn't complain, but helpfully offers to remove Werkzeug and Jinja2 as they will no longer be required once Flask is gone:

```
$ distil -e e2 remove flask
The following distributions will not be needed any more:
    Jinja2 (2.6)
    Werkzeug (0.8.3)
Remove them too? (y/n)
```

If we confirm that we want to do this, a final prompt showing what would be removed is displayed:

```
The following directories will be removed:
    e2/lib/python2.7/site-packages/flask
    e2/lib/python2.7/site-packages/Flask-0.9.dist-info
    e2/lib/python2.7/site-packages/jinja2
    e2/lib/python2.7/site-packages/Jinja2-2.6.dist-info
    e2/lib/python2.7/site-packages/werkzeug
    e2/lib/python2.7/site-packages/Werkzeug-0.8.3.dist-info
Proceed? (y/n)
```

If we say yes to this, the indicated items will be removed.

Uninstalling distributions installed with `pip`

In addition to removing distributions installed by itself, `distil` can also uninstall distributions installed by `pip`. Let's reset the environment and install Flask into it using `pip`:

```
$ reuse e2
$ e2/bin/pip install flask
Downloading/unpacking flask
  Downloading Flask-0.9.tar.gz (481kB): 481kB downloaded
... (lines omitted for brevity)
Successfully installed flask Werkzeug Jinja2
Cleaning up...
```

You can remove these individual distributions with `distil`, but because `pip` installation does not store information about which distributions were installed by user request and which were installed as dependencies, you don't get as good a user experience:

```
$ distil -e e2 remove werkzeug
The following directories will be removed:
    e2/lib/python2.7/site-packages/werkzeug
    e2/lib/python2.7/site-packages/Werkzeug-0.8.3-py2.7.egg-info
Proceed? (y/n) y
Removal completed.
$ distil -e e2 remove flask
The following directories will be removed:
    e2/lib/python2.7/site-packages/flask
    e2/lib/python2.7/site-packages/Flask-0.9-py2.7.egg-info
Proceed? (y/n) y
Removal completed.
$ distil -e e2 remove jinja2
The following directories will be removed:
    e2/lib/python2.7/site-packages/jinja2
    e2/lib/python2.7/site-packages/Jinja2-2.6-py2.7.egg-info
Proceed? (y/n) y
Removal completed.
```

Command line reference – `distil uninstall`

Here is the complete help for `distil`'s `uninstall` command:

```
$ distil help uninstall
usage: distil remove [-h] [-y] DIST [DIST ...]

Remove one or more installed distributions.

positional arguments:
  DIST        The name of an installed distribution.

optional arguments:
  -h, --help  show this help message and exit
  -y, --yes   Don't ask for confirmation of removals.
```

CHAPTER 6

Installing links to local projects

Sometimes developers want to have a project that's being worked on available in an environment as if it was installed there, but without actually installing it because they want any changes they make to the project to be immediately reflected in the environment without the need to reinstall. (`pip` calls these *editable* installations.)

You can get the equivalent effect using the `distil link` command, which works like `distil install` but only writes a link to the target environment (see *Where things are installed* to see where links are installed).

The command is run like this:

```
$ distil -e e2 link /path/to/my/project
```

You can specify more than one project path. Projects to be linked need to have a `package.json` file in the project directory, so that the project name can be determined. Here's an example:

```
$ e2/bin/python -c "import sarge; print(sarge.__file__)"
Traceback (most recent call last):
  File "<string>", line 1, in <module>
ImportError: No module named sarge
```

There's no `sarge` project available in the environment. Let's download it to a local directory:

```
$ distil download -d /tmp sarge
Downloading https://pypi.python.org/packages/source/s/sarge/sarge-0.1.tar.gz to /tmp/
↪sarge-0.1
    35KB @ 210 KB/s 100 % Done: 00:00:00
Unpacking ... done.
```

Now let's link the project:

```
$ distil -e e2 link /tmp/sarge-0.1
```

The command doesn't output anything unless an error occurs. Now, `sarge` should be available in the environment:

```
$ e2/bin/python -c "import sarge; print(sarge.__file__)"
/tmp/sarge-0.1/sarge/__init__.py
```

As you can see, the `sarge` project available in the environment is the one in the editable location `/tmp/sarge-0.1`.

Removing links

To remove links, you specify `-r` and the names of the projects you want to remove links to:

```
$ distil -e e2 link -r sarge
```

The command doesn't output anything unless an error occurs. Now, `sarge` will no longer be available in the environment:

```
$ e2/bin/python -c "import sarge; print(sarge.__file__)"
Traceback (most recent call last):
  File "<string>", line 1, in <module>
ImportError: No module named sarge
```

Command line reference – `distil link`

Here is the complete help for `distil`'s `link` command:

```
$ distil help link
usage: distil link [-h] [-r] [-s] [DIR_OR_NAME [DIR_OR_NAME ...]]

Install links to one or more local projects so that they can be used while
remaining editable.

positional arguments:
  DIR_OR_NAME   A directory of a local project when linking, or the project
                name when unlinking. If not specified, a list of existing
                links is displayed.

optional arguments:
  -h, --help    show this help message and exit
  -r            Remove links to named projects.
  -s, --system  Install to system Python (may require sufficient privileges).
                By default, installations are written to the user site
                (~/.local), or to a virtual environment specified with -e.
```

CHAPTER 7

Packaging distributions

You can use `distil` to package distributions. When using `distutils` or `setuptools` / `distribute`, you specify what will be packaged by passing arguments to the `setup()` function in `setup.py`. However, we are moving away from executable code and towards declarative metadata. Accordingly, `distil` uses declarative metadata in a file, `package.json`, that it uses instead of `setup.py` to describe how to package distributions. A description of this metadata is provided in *Packaging metadata*, but you can see how the metadata looks for most distributions on PyPI by using `distil` to download them:

```
$ distil download -d /tmp config
Downloading https://pypi.python.org/packages/source/c/config/config-0.3.9.tar.gz to /
↪tmp/config-0.3.9
    20KB @ 422 KB/s 100 % Done: 00:00:00
Unpacking ... done.
$ cat /tmp/config-0.3.9/package.json
{
  "source": {
    "modules": [
      "config"
    ]
  },
  "version": 1,
  "index-metadata": {
    "license": "Copyright (C) 2004-2010 by Vinay Sajip. All Rights Reserved. See␣
↪LICENSE for license.",
    "description": "This module allows a hierarchical configuration scheme with␣
↪support for mappings\nand sequences, cross-references between one part of the␣
↪configuration and\nanother, the ability to flexibly access real Python objects␣
↪without full-blown\neval(), an include facility, simple expression evaluation and␣
↪the ability to\nchange, save, cascade and merge configurations. Interfaces easily␣
↪with\nenvironment variables and command-line options. It has been developed on␣
↪python\n2.3 but should work on version 2.2 or greater.",
    "metadata_version": "2.0",
    "contacts": [
      {
        "role": "author",
```

```
        "name": "Vinay Sajip",
        "email": "vinay_sajip@red-dove.com"
      },
      {
        "role": "maintainer",
        "name": "Vinay Sajip",
        "email": "vinay_sajip@red-dove.com"
      }
    ],
    "summary": "A hierarchical, easy-to-use, powerful configuration module for Python
↪",
    "project_urls": {
      "Home": "http://www.red-dove.com/python_config.html"
    },
    "version": "0.3.9",
    "name": "config"
  }
}
```

This metadata is automatically generated from distributions which are on PyPI. If a particular distribution you download using `distil` comes without a `package.json` file, there could be a number of reasons for this:

- The distribution has recently been uploaded to PyPI, and the processing machinery hasn't got around to it yet. Try again in a few hours.
- The processing machinery has failed to process the distribution on PyPI, which could be due to a bug in the automatic processing code or a bug in the distribution's `setup.py`.

Source distributions

You can use `distil` to build source distributions in `.tar.gz`, `.tar.bz2` or `.zip` formats.

Let's consider a simple distribution called `frobozz`. The source tree looks like this:

```
.
- docs
|    - _build
|    - conf.py
|    - index.rst
|    - make.bat
|    - Makefile
|    - _static
|    - _templates
- frobozz.py
- MANIFEST
- README
- setup.py
```

The `setup.py` looks like this:

```
from distutils.core import setup

setup(
    name='frobozz',
    version='0.1',
    py_modules=['frobozz'],
    author='Distlib User',
```

```
    author_email='distlib.user@dummy.org',
)
```

Let's replace the `setup.py` with the equivalent `package.json`:

```
{
  "source": {
    "include": [
      "README"
    ],
    "modules": [
      "frobozz"
    ]
  },
  "version": 1,
  "metadata": {
    "version": "0.1",
    "name": "frobozz",
    "author-email": "distlib.user@dummy.org",
    "author": "Distlib User"
  }
}
```

If we're in the root directory of the `frobozz` project, we can package it by simply issuing the command:

```
$ distil package
The following packages were built:
  frobozz-0.1.tar.gz
```

By default, a `.tar.gz` source archive in `dist` is built. Let's look at its contents:

```
$ tar tzvf dist/frobozz-0.1.tar.gz
drwxrwxr-x vinay/vinay       0 2013-03-21 12:46 frobozz-0.1/
-rw-rw-r-- vinay/vinay       0 2013-03-20 09:58 frobozz-0.1/README
-rw-rw-r-- vinay/vinay       0 2013-03-20 09:57 frobozz-0.1/frobozz.py
-rw-rw-r-- vinay/vinay     257 2013-03-21 12:40 frobozz-0.1/package.json
```

To build other formats, you can specify them in a `--formats` parameter:

```
$ distil package --formats=gztar,bztar,zip
The following packages were built:
  frobozz-0.1.tar.bz2
  frobozz-0.1.tar.gz
  frobozz-0.1.zip
```

Binary distributions

Currently, `distil` only supports building binary distributions using the Wheel format (see PEP 427).

The method for creating wheels is just the same as for source distributions:

```
$ distil package --formats=wheel
The following packages were built:
  /home/vinay/projects/frobozz/dist/frobozz-0.1-py27-none-any.whl
$ unzip -l dist/frobozz-0.1-py27-none-any.whl
Archive:  dist/frobozz-0.1-py27-none-any.whl
```

```
  Length      Date    Time    Name
---------  ---------- -----   ----
        0  2013-06-23 17:59   frobozz.py
      148  2013-06-23 17:59   frobozz-0.1.dist-info/pymeta.json
       89  2013-06-23 17:59   frobozz-0.1.dist-info/WHEEL
      266  2013-06-23 17:59   frobozz-0.1.dist-info/RECORD
---------                     -------
      503                     4 files
```

Building wheels for dependencies

When you build a wheel using the `package` command, only the package itself is built - not its dependencies. If you need to build dependencies of your package, use `distil`'s `pip` command (see below).

Using pip when building wheels

Sometimes, the distribution you want to build a wheel for uses custom code in `setup.py` to set things up correctly for the build. In such cases, you may need to use `pip` to build your wheel. For this purpose, `distil` provides the `pip` command. This works from requirements rather than source directories, and is intended to be used for PyPI-hosted dependencies of your package rather than for your package itself (use `distil package` for that).

When using `distil pip`, note that `pip` is used to do a customised installation from which `distil` then builds the wheel using `distlib`. This means that you should use a clean venv when running `distil pip`, because `pip` won't install any already-installed distributions, and a clean venv won't have any of those, minimising problems in your workflow. Here's an example of running `distil pip`:

```
$ distil -e d2 pip Flask
Checking requirements for Flask (0.9) ... done.
Pipping Jinja2==2.6 ...
Pipping Werkzeug==0.8.3 ...
Pipping Flask==0.9 ...
The following wheels were built:
  Jinja2-2.6-py27-none-any.whl
  Werkzeug-0.8.3-py27-none-any.whl
  Flask-0.9-py27-none-any.whl
```

The wheels are written in the current directory by default.

You can also use requirements files, just as with `distil install`. Here is the complete help for `distil`'s `pip` command:

```
$ distil help pip
usage: distil pip [-h] [-r REQTFILE [REQTFILE ...]] [-d DESTDIR] [--no-deps]
                  [--prereleases]
                  [REQT [REQT ...]]

Build wheels using pip. Use this when distil's build logic doesn't work
because of code in setup.py which needs to be run.

positional arguments:
  REQT                  A requirement using a distribution on PyPI.

optional arguments:
  -h, --help            show this help message and exit
  -r REQTFILE [REQTFILE ...]
```

```
                        Get requirements from specified file(s)
  -d DESTDIR, --destination DESTDIR
                        Location to write the wheels to. Defaults to the
                        current directory.
  --no-deps             Don't build wheels for dependencies when building
                        wheels. The default behaviour is to build wheels for
                        all dependencies.
  --prereleases         Include pre-releases when downloading. By default,
                        pre-releases are skipped, unless they are all that is
                        available.
```

Command line reference – `distil package`

Here is the complete help for `distil`'s `package` command:

```
$ distil help package
usage: distil package [-h] [--formats {gztar,bztar,zip,wheel}]
                      [--python-tags PYTAGS] [-d DESTDIR]
                      [DIR]

Create source distributions or wheels.

positional arguments:
  DIR                   The directory containing the software to package.
                        If not specified, the current directory is used. If
                        only packaging to a wheel, you can also specify
                        the path to a bdist_wininst installer, which will
                        be converted to a wheel in the same directory.

optional arguments:
  -h, --help            show this help message and exit
  --formats {gztar,bztar,zip,wheel}
                        The formats to produce packages in.
  --python-tags PYTAGS  Specify Python compatibility tags when building
                        wheels. This option is ignored when building source
                        archives.
  -d DESTDIR, --destination DESTDIR
                        Location to write the packaged distributions to.
  --wheel-version WHEEL_VERSION
                        Wheel-Version value to write to built wheels, in
                        the form M.N. This option is ignored when building
                        source archives.
```

Packaging metadata

The metadata discussed in **PEP 426**, and its precursor PEPs, is a (potentially) small subset of the total metadata relating to a distribution. If we call the PEP 426 metadata "index metadata", then the overall metadata for a distribution would comprise (the list may be incomplete):

- The index metadata (PEP 426 et al)
- Metadata about how to build a source distribution from a source tree
- Metadata about how to build a binary distribution from a source tree/source distribution

- Metadata about things a distribution exports for use by other distributions
- Metadata used by installers to install the distribution

We're attempting in the PEPs to standardise the next revision of the first of these, but the other categories haven't been considered at all (from a standardisation point of view). At present, in the `distribute` / `setuptools` / `distutils` world, they are provided by a mixture of `MANIFEST.in` files and a bunch of keyword arguments passed to `setup()`. This, coupled with the command-class design of `distutils`, has led to a lot of *ad hoc* approaches to extending `distutils` where it fell short – monkey-patching, custom command classes etc., which has led to the present less than ideal situation.

A declarative approach is now generally considered better than `setup.py`: it allows for multiple, competing implementations which should be interoperable. The `distutils2` approach was to focus on the declarative `setup.cfg`, and the new wheel format is also essentially declarative in nature.

A flat key-value structure for representing the other types of metadata doesn't seem ideal. It might seem heretical, but backward compatibility aside, it's not clear why we aren't thinking about JSON as a metadata format. It has mature support in the stdlib, handles Unicode, and allows more meaningful structuring of the metadata.

Examples of JSON metadata can be seen in the following examples:

- **Assimulo (a Fortan-heavy numerical package):** http://www.red-dove.com/pypi/projects/A/Assimulo/package-2.3.json
- **Twisted:** http://www.red-dove.com/pypi/projects/T/Twisted/package-12.3.0.json
- **Pyramid:** http://www.red-dove.com/pypi/projects/P/pyramid/package-1.4b3.json
- **Django:** http://www.red-dove.com/pypi/projects/D/Django/package-1.4.3.json

The index metadata is a small part of the overall metadata (it appears at key `index-metadata` in the top-level dict expressed in the JSON). You will most likely find metadata for distributions of interest to you by using the URI scheme indicated by the above examples.

Using this type of metadata, `distil` can:

- Build a source archive from the metadata and the source archive which is essentially the same as the source.
- Build wheels.
- Install into a virtualenv such that the venv layout after installation is identical to that following an installation with `pip`.

So, while the metadata schema used is provisional and can be improved, it brings across what *can* be brought across from `setup()`, such that it can be used to install software identically to `pip`, for a large number of distributions currently on PyPI. Such a declarative solution can work, provided there isn't custom code which runs at installation time. Where code *is* called at installation time from `setup.py`, because the effects of that clearly can't be brought over into a declarative format, it may not be possible to install affected distributions correctly. (For such distributions, `distil` offers the `pip` command to create wheels which can then be installed using `distil`'s `install` command. See *Using pip when building wheels* for more information.)

The other things that structured metadata makes possible is that it's relatively easy to transform into useful forms. For example, when `distlib` does dependency resolution, it can make effective use of selected metadata across all versions of a project. For example, you can see all the versions of a project, what the download URLs are and their digests and sizes, and the distribution dependencies – just by rearranging and aggregating the data.

Examples:

- **contextlib:** http://www.red-dove.com/pypi/projects/C/contextlib2/project.json
- **Sphinx:** http://www.red-dove.com/pypi/projects/S/Sphinx/project.json

This allows `distlib`-using code to resolve dependencies without ever downloading a distribution, as `pip` has to do. The overall effect is more like `RPM` and `apt-get`: You get told before downloading any distribution what other dependencies will be installed, and whether any existing distributions will get upgraded.

Schema for the extended JSON metadata

The schema for the extended metadata is given by the following sample JSON (not valid JSON, due to the comments). The example may not be exhaustive, but gives a flavour of what is covered by the metadata:

```
{
  "version": 1, # version of this schema
  "exports": {
    # equivalent to setuptools' entry points
    "frobozz.processors": [
      "name1 = frobozz.sub.package:do_nothing",
      "name2 = frobozz.sub.package:do_something"
    ],
    "scripts": {
      "console": [
        "frobozz = frobozz.cli:main"
      ],
      "gui": [
        "frobozzw = frobozz.gui:main"
      ]
    }
  },
  "requirements": {
    "install": [
      # list of requirements needed post-install
      "foo (>= 1.0)"
    ]
    "setup": [
      # list of requirements needed for setup
      "bar (>= 2.0)"
    ]
    "test": [
      # list of requirements needed for testing
      "nose (>= 1.2)"
    ],
    "extras": {
      # dict of requirements needed for extras
      # list of requirements keyed by extra name
      "i18n": [
        "Babel (>= 0.8)"
      ]
    }
  },
  "source": {
    "include-package-data": true,
    "data-files": [
        # a list of lists. Each entry in the outer list is
        # a directory followed by a list of data files in
        # that directory.
      [
        "dir1",
        [
          "file1_in_dir1.ext",
```

```
          "file2_in_dir1.ext"
        ]
      ],
      [
        "dir2",
        [
          "file1_in_dir2.ext",
          "file2_in_dir2.ext"
        ]
      ]
      # and so on
    ],
    "include": [
      # e.g. scripts
      "bin/script1",
      "bin/script2"
    ],
    "packages": [
      # Python packages
      "frobozz.foo",
      "frobozz.foo.bar",
      "frobozz.foo.bar.baz"
    ],
    "modules": [
      "mod1",
      "mod2",
      "mod3"
    ],
    "manifest": [
      "include data/global.dat",
      "include data/localedata/*.dat",
      "include doc/api/*.*",
      "include doc/*.html"
    ]
  },
  "extensions": {
    # C extensions
    "frobozz.foo.ext_one": {
      "extra_link_args": [],
      "swig_opts": [],
      "language": null,
      "define_macros": [],
      "extra_objects": [],
      "runtime_library_dirs": [],
      "libraries": [],
      "sources": [
        "foo/extension/module_one.c"
      ],
      "depends": [],
      "export_symbols": [],
      "extra_compile_args": [],
      "undef_macros": [],
      "include_dirs": [],
      "library_dirs": [],
      "name": "frobozz.foo.ext_one"
    },
    "frobozz.foo.ext_two": {
      "extra_link_args": [],
```

```
      "swig_opts": [],
      "language": null,
      "define_macros": [],
      "extra_objects": [],
      "runtime_library_dirs": [],
      "libraries": [],
      "sources": [
        "foo/extension/module_two.c"
      ],
      "depends": [],
      "export_symbols": [],
      "extra_compile_args": [],
      "undef_macros": [],
      "include_dirs": [],
      "library_dirs": [],
      "name": "frobozz.foo.ext_two"
    }
  },
  "scripts": [
    # list of scripts to install
    "bin/script1",
    "bin/script2"
  ],
  "index-metadata": {
    # This dict conforms to PEP 426
    "metadata_version": "2.0",
    "name": "frobozz",
    "version": "1.3.0",
    "license": "MIT",
    "summary": "An example project",
    "description": "An example description.\n"
    "classifiers": [
      "Programming Language :: Python :: 2.6",
      "Programming Language :: Python :: 2.7"
    ],
    "contacts": [
      {
        "name": "Frobozz Developers",
        "email": "frobozz.devel@some.domain.com",
        "role": "author"
      },
      {
        "name": "Some User",
        "email": "some.user@some.domain.com",
        "role": "maintainer"
      }
    ],
    "project_urls": {
      "Home": "http://frobozz.com/"
    }
  },
  "test": {
    # test specifications
    "test-suite": "frobozz.tests.suite",
    "test-runner": "frobozz.tests.runner"
  },
  "build": {
    # build specifications
```

```
    "use-2to3": true,
    "use-2to3-fixers": [
      "custom_fixers"
    ]
  }
}
```

You may find it useful to locate metadata for actual distributions on PyPI which you may be familiar with. You can start exploring here.

Creating an initial version of metadata for new projects

The `distil init` command creates an initial `pydist.json` file in a specified directory. You can provide some command-line default values, as shown in the next section. The module names are written as a list to the "modules" element of the metadata (see **PEP 426** for more information).

Command line reference – `distil init`

Here is the complete command-line help for `distil init`:

```
$ distil help init
usage: distil init [-h] [--name NAME] [--projver PROJVER] [--author AUTHOR]
                   [--email EMAIL] [--home-page HOMEPAGE]
                   PATH [MODULE [MODULE ...]]

Create minimal metadata for a project that you can then add to during
development.

positional arguments:
  PATH                  A directory of a local project where the metadata file
                        is to be created.
  MODULE                An optional set of modules or packages in the project.

optional arguments:
  -h, --help            show this help message and exit
  --name NAME           The name of the project.
  --projver PROJVER     The version of the project.
  --author AUTHOR       The author of the project.
  --email EMAIL         The author's email address.
  --home-page HOMEPAGE  The home page URL of the project.
```

CHAPTER 8

Adding content to PyPI

Registering a project on PyPI

To register a project on PyPI, you are required (*by PyPI*) to provide both a name (which is unclaimed on PyPI) and a version number (even if you are not actually uploading a release yet). You can register a project using the `register` command with a requirement giving the name and version number:

```
$ distil register "frobozz (0.1)"
Project registered: frobozz (0.1)
```

By default, any credentials you have set up in your PyPI configuration will be used. If you have no PyPI configuration, you will be prompted for username and password:

```
$ distil register "frobozz 0.1"
Enter your PyPI username:distlib_user
Enter your PyPI password:
Project registered: frobozz (0.1)
```

Note that the password is not echoed to the console.

You can specify an overriding username on the command line, which will be used instead of any value in your PyPI configuration:

```
$ distil register -u distlib_user "frobozz 0.1"
Enter your PyPI password:
Project registered: frobozz (0.1)
```

If you wish, you can also specify the password on the command line:

```
$ distil register -u distlib_user --pass secret "frobozz 0.1"
Project registered: frobozz (0.1)
```

Command line reference – `distil register`

Here is the complete help for `distil`'s `register` command:

```
$ distil help register
usage: distil register [-h] [-u USERNAME] [--password PASSWORD] REQT

Register a project on PyPI.

positional arguments:
  REQT                  The requirement giving the package and version.

optional arguments:
  -h, --help            show this help message and exit
  -u USERNAME, --username USERNAME
                        The username to use when registering.
  --password PASSWORD   The password to use when registering.
```

Uploading a release to PyPI

Uploading releases is done by invoking the `upload` command and specifying the project name/version and path to the archive for the release:

```
$ distil upload "dummy 0.1" /tmp/dummy-0.1.tar.gz
Release uploaded.
```

The reason you need to specify the name and version explicitly is that there can be ambiguities if `distil` just tries to parse the archive filename.

If a source archive is specified, it is expected to contain an entry called `PKG-INFO` in the "name-version" directory in the root of the archive, which has the complete metadata for the release. If a wheel is being uploaded, the metadata for the release is contained in the conventional location in the wheel.

Currently, `distil` only allows uploading of source archives and wheels.

Command line reference – `distil upload`

Here is the complete help for `distil`'s `upload` command:

```
$ distil help upload
usage: distil upload [-h] [-u USERNAME] [--password PASSWORD]
                     REQT ARCHIVE_OR_DIR

Upload a release or documentation to PyPI.

positional arguments:
  REQT                  The requirement giving the package and version.
  ARCHIVE_OR_DIR        A source archive, wheel or documentation directory.

optional arguments:
  -h, --help            show this help message and exit
  -u USERNAME, --username USERNAME
                        The username to use when uploading.
  --password PASSWORD   The password to use when uploading.
```

Uploading HTML documentation

Uploading documentation is also done using the `upload` command, but instead of specifying a release archive to upload, you indicate a directory which contains HTML documentation (say, the `docs/_build/html` directory in your project, if you are using normal Python conventions used by Sphinx):

```
$ distil upload "dummy 0.1" docs/_build/html
Documentation uploaded.
```

CHAPTER 9

Testing distributions

You can use `distil` to test distributions. This is primarily intended for trying out potential dependencies and exercises `distlib`'s logic for distinguishing between dependencies needed for setup, testing and post-installation:

```
$ distil test jinja2
Checking requirements for Jinja2 (2.6) ... done.
The following packages will be downloaded, built and tested:
    Jinja2 (2.6)
Downloading Jinja2-2.6.tar.gz to /tmp/tmpkd4eUK
    380KB @  14 KB/s 100 % Done: 00:00:26
Unpacking ... done.
Building Jinja2 (2.6) ...
    Running check ...
    Running build_ext ...
    Running build_py ...
    Build completed.
Testing Jinja2 (2.6) ...
Running test ...
/tmp/tmpkd4eUK/Jinja2-2.6/build/lib.linux-x86_64-2.7/jinja2/loaders.py:214:␣
↪UserWarning: Module jinja2 was already imported from /tmp/tmpkd4eUK/Jinja2-2.6/
↪build/lib.linux-x86_64-2.7/jinja2/__init__.py, but /usr/lib/python2.7/dist-packages␣
↪is being added to sys.path
  from pkg_resources import DefaultProvider, ResourceManager, \
test_do (jinja2.testsuite.ext.ExtensionsTestCase) ... ok
test_extend_late (jinja2.testsuite.ext.ExtensionsTestCase) ... ok
... (many tests omitted for brevity)
module (jinja2.environment.Template)
Doctest: jinja2.environment.Template.module ... ok
FileSystemBytecodeCache (jinja2.bccache)
Doctest: jinja2.bccache.FileSystemBytecodeCache ... ok

----------------------------------------------------------------------
Ran 278 tests in 0.967s

OK
```

Dependencies of the distribution under test are automatically downloaded and built:

```
$ distil -e e2 test argproc
Checking requirements for argproc (1.4) ... done.
The following packages will be downloaded and built:
    nose (1.2.1) [for argproc]
    ply (3.4) [for argproc]
The following packages will be downloaded, built and tested:
    argproc (1.4)
Proceed? (y/n) y
Downloading nose-1.2.1.tar.gz to /tmp/tmp_REn8s [for argproc]
    390KB @  11 KB/s 100 % Done: 00:00:35
Unpacking ... done.
Downloading ply-3.4.tar.gz to /tmp/tmpzU9dnC [for argproc]
    135KB @  12 KB/s 100 % Done: 00:00:11
Unpacking ... done.
Downloading argproc-1.4.tar.gz to /tmp/tmp5mgBIO
    10KB @  20 KB/s 100 % Done: 00:00:00
Unpacking ... done.
Building nose (1.2.1) ...
    Running check ...
    Running build_ext ...
    Running build_py ...
    Build completed.
Building ply (3.4) ...
    Running check ...
    Running build_ext ...
    Running build_py ...
    Build completed.
Building argproc (1.4) ...
    Running check ...
    Running build_ext ...
    Running build_py ...
    Build completed.
Testing argproc (1.4) ...
Running test ...
argproc.test.test_processor.TestProcessor.test_attribute_reference ... ok
argproc.test.test_processor.TestProcessor.test_bidirectional ... ok
argproc.test.test_processor.TestProcessor.test_comment ... ok
argproc.test.test_processor.TestProcessor.test_dict ... ok
argproc.test.test_processor.TestProcessor.test_false ... ok
argproc.test.test_processor.TestProcessor.test_float ... ok
argproc.test.test_processor.TestProcessor.test_function_call ... ok
argproc.test.test_processor.TestProcessor.test_function_call_with_multiple_arguments .
↪.. ok
argproc.test.test_processor.TestProcessor.test_function_with_validator ... ok
argproc.test.test_processor.TestProcessor.test_global_name ... ok
argproc.test.test_processor.TestProcessor.test_int ... ok
argproc.test.test_processor.TestProcessor.test_list ... ok
argproc.test.test_processor.TestProcessor.test_local_name ... ok
argproc.test.test_processor.TestProcessor.test_mandatory ... ok
argproc.test.test_processor.TestProcessor.test_multiple_rules ... ok
argproc.test.test_processor.TestProcessor.test_multiple_tags ... ok
argproc.test.test_processor.TestProcessor.test_negated_tag ... ok
argproc.test.test_processor.TestProcessor.test_none ... ok
argproc.test.test_processor.TestProcessor.test_optional ... ok
argproc.test.test_processor.TestProcessor.test_reverse ... ok
argproc.test.test_processor.TestProcessor.test_simple ... ok
argproc.test.test_processor.TestProcessor.test_slicing ... ok
```

```
argproc.test.test_processor.TestProcessor.test_str_double_quoted ... ok
argproc.test.test_processor.TestProcessor.test_str_single_quoted ... ok
argproc.test.test_processor.TestProcessor.test_subscription ... ok
argproc.test.test_processor.TestProcessor.test_tags ... ok
argproc.test.test_processor.TestProcessor.test_true ... ok
argproc.test.test_processor.TestProcessor.test_tuple ... ok
argproc.test.test_processor.TestProcessor.test_tuple_with_one_entry ... ok
argproc.test.test_processor.TestProcessor.test_unidirectional ... ok
argproc.test.test_processor.TestProcessor.test_validator_callable ... ok
argproc.test.test_processor.TestProcessor.test_validator_function_call ... ok
argproc.test.test_processor.TestProcessor.test_validator_list ... ok
argproc.test.test_processor.TestProcessor.test_validator_literal ... ok
argproc.test.test_processor.TestProcessor.test_validator_tuple ... ok

----------------------------------------------------------------------
Ran 35 tests in 0.180s

OK
```

Note that the dependencies are not installed - they are downloaded and built purely for the purposes of testing, and discarded afterwards.

Command line reference – `distil test`

Here is the complete help for the `distil`'s `test` command:

```
$ distil help test
usage: distil test [-h] [--prereleases] [DIST [DIST ...]]

Test a distribution without installing it.

positional arguments:
  DIST           The name of a distribution.

optional arguments:
  -h, --help     show this help message and exit
  --prereleases  Include pre-releases when testing. By default, pre-releases
                 are skipped.
```

CHAPTER 10

Getting information about distributions

Displaying installed distributions as lists

You can use `distil list` to see information about installed distributions:

```
$ distil list
MarkupSafe     0.15
lxml           2.3.5
protobuf       2.4.1
defer          1.0.6
configglue     1.0
dirspec        4.0.0
Mako           0.7.1
zope.interface 3.6.1
feedparser     5.1.2
python-debian  0.1.21-nmu2ubuntu1
debtagshw      0.1
oauth          1.0.1
```

This display is roughly equivalent to `pip freeze`. If you want a more verbose display, you can specify `-v`:

```
$ distil -v list
MarkupSafe     0.15*
lxml           2.3.5*
protobuf       2.4.1*
defer          1.0.6*
configglue     1.0*
dirspec        4.0.0*
Mako           0.7.1*
zope.interface 3.6.1*
feedparser     5.1.2*
python-debian  0.1.21-nmu2ubuntu1*
debtagshw      0.1*
oauth          1.0.1*
* These are distutils/setuptools/distribute distributions
```

If you want to see whether there are any more recent versions of installed distributions, specify `--latest` (which needs `-v` specified as well):

```
$ distil -v list --latest
MarkupSafe     0.15*
lxml           2.3.5               (latest: 3.1.0)*
protobuf       2.4.1               (latest: 2.5.0)*
defer          1.0.6*
configglue     1.0                 (latest: 1.0.3)*
dirspec        4.0.0               (latest: 4.1.90)*
Mako           0.7.1               (latest: 0.7.3)*
zope.interface 3.6.1               (latest: 4.0.5)*
feedparser     5.1.2               (latest: 5.1.3)*
python-debian  0.1.21-nmu2ubuntu1*
debtagshw      0.1*
oauth          1.0.1*
* These are distutils/setuptools/distribute distributions
```

Command line reference – `distil list`

Here is the complete help for `distil`'s `list` command:

```
$ distil help list
usage: distil list [-h] [-l] [DIST [DIST ...]]

List one or all installed distributions.

positional arguments:
  DIST          The name of an installed package. If not specified, all
                packages are listed.

optional arguments:
  -h, --help    show this help message and exit
  -l, --latest  show latest versions
```

Displaying dependency graphs as lists

The `graph` command of `distil` is used to display dependencies of a distribution as a topologically sorted list:

```
$ distil graph pyramid
PasteDeploy (1.5.0) [for pyramid]
repoze.lru (0.6) [for pyramid]
MarkupSafe (0.15) [for Mako]
translationstring (1.1) [for pyramid]
Chameleon (2.11) [for pyramid]
zope.interface (4.0.5) [for pyramid]
zope.deprecation (4.0.2) [for pyramid]
WebOb (1.2.3) [for pyramid]
Mako (0.7.3) [for pyramid]
pyramid (1.4)
```

Displaying dependency graphs as images

You can use the `--image` parameter to `distil`'s `graph` command to produce images through the GraphViz package, specifically the `dot` command which it includes:

```
$ distil graph --image flask-sqlalchemy | dot -T png > depend-1.png
```

This produces the following image:

Command line reference – `distil graph`

Here is the complete help for `distil`'s `graph` command:

```
$ distil help graph
usage: distil graph [-h] [-i] REQT

Show the dependency graph for a distribution.

positional arguments:
  REQT         A requirement identifying a distribution.

optional arguments:
  -h, --help   show this help message and exit
  -i, --image  Produce output suitable for GraphViz
```

Index