Distlib Documentation

Release 0.1.5

Vinay Sajip

December 14, 2013

Contents

Welcome to the documentation for `distlib`, **a library of packaging functionality which is intended to be used as the basis for third-party packaging tools**. Using a common layer will improve interoperability and consistency of user experience across those tools which use the library.

Please note: this documentation is a *work in progress*.

CHAPTER 1

Overview

Start here for all things `distlib`.

1.1 Distlib evolved out of `packaging`

Distlib is a library which implements low-level functions that relate to packaging and distribution of Python software. It consists in part of the functions in the `packaging` Python package, which was intended to be released as part of Python 3.3, but was removed shortly before Python 3.3 entered beta testing.

1.2 What was the problem with `packaging`?

The `packaging` software just wasn't ready for inclusion in the Python standard library. The amount of work needed to get it into the desired state was too great, given the number of people able to work on the project, the time they could devote to it, and the Python 3.3 release schedule.

The approach taken by `packaging` was seen to be a good one: to ensure interoperability and consistency between different tools in the packaging space by defining standards for data formats through PEPs, and to do away with the *ad hoc* nature of installation encouraged by the `distutils` approach of using executable Python code in `setup.py`. Where custom code was needed, it could be provided in a standardised way using installation hooks.

While some very good work was done in defining PEPs to codify some of the best practices, `packaging` suffered from some drawbacks, too:

- Not all the PEPs may have been functionally complete, because some important use cases were not considered – for example, built (binary) distributions for Windows.
- It continued the command-based design of `distutils`, which had resulted in `distutils` being difficult to extend in a consistent, easily understood, and maintainable fashion.
- Some important features required by distribution authors were not considered – for example:
  - Access to data files stored in Python packages.
  - Support for plug-in extension points.
  - Support for native script execution on Windows.

  These features are supported by third-party tools (like `setuptools` / `Distribute`) using `pkg_resources`, *entry points* and *console scripts*.

- There were a lot of rough edges in the `packaging` implementation, both in terms of bugs and in terms of incompletely implemented features. This can be seen (with the benefit of hindsight) as due to the goals being set too ambitiously; the project developers bit off more than they could chew.

1.3 How Distlib can help

The idea behind Distlib is expressed in this python-dev mailing-list post, though a different name was suggested for the library. Basically, Distlib contains the implementations of the packaging PEPs and other low-level features which relate to packaging, distribution, and deployment of Python software. If Distlib can be made genuinely useful, then it is possible for third-party packaging tools to transition to using it. Their developers and users then benefit from standardised implementation of low-level functions, time saved by not having to reinvent wheels, and improved interoperability between tools.

1.4 How you can help

If you have some time and the inclination to improve the state of Python packaging, then you can help by trying out Distlib, raising issues where you find problems, contributing feedback and/or patches to the implementation, documentation, and underlying PEPs.

1.5 Main features

Distlib currently offers the following features:

- The package `distlib.database`, which implements a database of installed distributions, as defined by **PEP 376**, and distribution dependency graph logic. Support is also provided for non-installed distributions (i.e. distributions registered with metadata on an index like PyPI), including the ability to scan for dependencies and building dependency graphs.
- The package `distlib.index`, which implements an interface to perform operations on an index, such as registering a project, uploading a distribution or uploading documentation. Support is included for verifying SSL connections (with domain matching) and signing/verifying packages using GnuPG.
- The package `distlib.metadata`, which implements distribution metadata as defined by **PEP 426**, **PEP 345**, **PEP 314** and **PEP 241**.
- The package `distlib.markers`, which implements environment markers as defined by **PEP 426**.
- The package `distlib.manifest`, which implements lists of files used in packaging source distributions.
- The package `distlib.locators`, which allows finding distributions, whether on PyPI (XML-RPC or via the “simple” interface), local directories or some other source.
- The package `distlib.resources`, which allows access to data files stored in Python packages, both in the file system and in .zip files.
- The package `distlib.scripts`, which allows installing of scripts with adjustment of shebang lines and support for native Windows executable launchers.
- The package `distlib.version`, which implements version specifiers as defined by **PEP 440** / **PEP 426**, but also support for working with “legacy” versions (`setuptools`/`distribute`) and semantic versions.
- The package `distlib.wheel`, which provides support for building and installing from the Wheel format for binary distributions (see **PEP 427**).

- The package `distlib.util`, which contains miscellaneous functions and classes which are useful in packaging, but which do not fit neatly into one of the other packages in `distlib`.* The package implements enhanced globbing functionality such as the ability to use `**` in patterns to specify recursing into subdirectories.

1.6 Python version and platform compatibility

Distlib is intended to be used on any Python version >= 2.6 and is tested on Python versions 2.6, 2.7, 3.1, 3.2, and 3.3 on Linux, Windows, and Mac OS X (not all versions are tested on all platforms, but are expected to work correctly).

1.7 Project status

The project has reached alpha status in its development: there is a test suite and it has been exercised on Windows, Ubuntu and Mac OS X. To work with the project, you can download a release from PyPI, or clone the source repository or download a tarball from it.

Coverage results are available at:

http://www.red-dove.com/distlib/coverage/

Continuous integration test results are available at:

https://travis-ci.org/vsajip/distlib/

The source repository for the project is on BitBucket:

https://bitbucket.org/pypa/distlib/

You can leave feedback by raising a new issue on the issue tracker (BitBucket registration not necessary, but recommended).

1.8 Next steps

You might find it helpful to look at the *Tutorial*, or the *API Reference*.

CHAPTER 2

Tutorial

This is the place to start your practical exploration of `distlib`.

2.1 Installation

Distlib is a pure-Python library. You should be able to install it using:

```
pip install distlib
```

for installing `distlib` into a virtualenv or other directory where you have write permissions. On Posix platforms, you may need to invoke using `sudo` if you need to install `distlib` in a protected location such as your system Python's `site-packages` directory.

2.2 Testing

A full test suite is included with `distlib`. To run it, you'll need to download the source distribution, unpack it and run `python setup.py test` in the top-level directory of the package. You can of course also run `python setup.py install` to install the package (perhaps invoking with `sudo` if you need to install to a protected location).

Continuous integration test results are available at:

https://travis-ci.org/vsajip/distlib/

Coverage results are available at:

http://www.red-dove.com/distlib/coverage/

These are updated as and when time permits.

Note that the index tests are configured, by default, to use a local test server, though they can be configured to run against PyPI itself. This local test server is not bundled with `distlib`, but is available from:

https://raw.github.com/vsajip/pypiserver/standalone/pypi-server-standalone.py

This is a slightly modified version of Ralf Schmitt's pypiserver. To use, the script needs to be copied to the `tests` folder of the `distlib` distribution.

If the server script is not available, the tests which use it will be skipped.

2.2.1 PYPI availability

If PyPI is unavailable or slow, then some of the tests can fail or become painfully slow. To skip tests that might be sometimes slow, set the `SKIP_SLOW` environment variable:

```
$ SKIP_SLOW=1 python setup.py test
```

on Posix, or:

```
C:\> set SKIP_SLOW=1
C:\> python setup.py test
```

on Windows.

2.3 First steps

For now, we just list how to use particular parts of the API as they take shape.

2.3.1 Using the database API

You can use the `distlib.database` package to access information about installed distributions. This information is available through the following classes:

- `DistributionPath`, which represents a set of distributions installed on a path.
- `Distribution`, which represents an individual distribution, conforming to recent packaging PEPs (**PEP 426**, **PEP 386**, **PEP 376**, **PEP 345**, **PEP 314** and **PEP 241**).
- `EggInfoDistribution`, which represents a legacy distribution in egg format.

Distribution paths

The `Distribution` and `EggInfoDistribution` classes are normally not instantiated directly; rather, they are returned by querying `DistributionPath` for distributions. To create a `DistributionPath` instance, you can do

```
>>> from distlib.database import DistributionPath
>>> dist_path = DistributionPath()
```

Querying a path for distributions

In this most basic form, `dist_path` will provide access to all non-legacy distributions on `sys.path`. To get these distributions, you invoke the `get_distributions()` method, which returns an iterable. Let's try it:

```
>>> list(dist_path.get_distributions())
[]
```

This may seem surprising if you've just started looking at `distlib`, as you won't *have* any non-legacy distributions.

Including legacy distributions in the search results

To include distributions created and installed using `setuptools` or `distribute`, you need to create the `DistributionPath` by specifying an additional keyword argument, like so:

```
>>> dist_path = DistributionPath(include_egg=True)
```

and then you'll get a less surprising result:

```
>>> len(list(dist_path.get_distributions()))
77
```

The exact number returned will be different for you, of course. You can ask for a particular distribution by name, using the `get_distribution()` method:

```
>>> dist_path.get_distribution('setuptools')
<EggInfoDistribution u'setuptools' 0.6c11 at '/usr/lib/python2.7/dist-packages/setuptools.egg-info'>
```

If you want to look at a specific path other than `sys.path`, you specify it as a positional argument to the `DistributionPath` constructor:

```
>>> from pprint import pprint
>>> special_dists = DistributionPath(['tests/fake_dists'], include_egg=True)
>>> pprint([d.name for d in special_dists.get_distributions()])
['babar',
 'choxie',
 'towel-stuff',
 'grammar',
 'truffles',
 'coconuts-aster',
 'nut',
 'bacon',
 'banana',
 'cheese',
 'strawberry']
```

or, if you leave out egg-based distributions:

```
>>> special_dists = DistributionPath(['tests/fake_dists'])
>>> pprint([d.name for d in special_dists.get_distributions()])
['babar',
 'choxie',
 'towel-stuff',
 'grammar']
```

Distribution properties

Once you have a `Distribution` instance, you can use it to get more information about the distribution. For example:

- The `metadata` attribute gives access to the distribution's metadata (see *Using the metadata and markers APIs* for more information).
- The `name_and_version` attribute shows the name and version in the format `name (X.Y)`.
- The `key` attribute holds the distribution's name in lower-case, as you generally want to search for distributions without regard to case sensitivity.

Exporting things from Distributions

Each distribution has a dictionary of *exports*. The exports dictionary is functionally equivalent to "entry points" in `distribute` / `setuptools`.

The keys to the dictionary are just names in a hierarchical namespace delineated with periods (like Python packages, so we'll refer to them as *pkgnames* in the following discussion). The keys indicate categories of information which the distribution's author wishes to export. In each such category, a distribution may publish one or more entries.

The entries can be used for many purposes, and can point to callable code or data. A common purpose is for publishing callables in the distribution which adhere to a particular protocol.

To give a concrete example, the Babel library for internationalisation support provides a mechanism for extracting, from a variety of sources, message text to be internationalised. Babel itself provides functionality to extract messages from e.g. Python and JavaScript source code, but helpfully offers a mechanism whereby providers of other sources of message text can provide their own extractors. It does this by providing a category `'babel.extractors'`, under which other software can register extractors for their sources. The Jinja2 template engine, for example, makes use of this to provide a message extractor for Jinja2 templates. Babel itself registers its own extractors under the same category, so that a unified view of all extractors in a given Python environment can be obtained, and Babel's extractors are treated by other parts of Babel in exactly the same way as extractors from third parties.

Any installed distribution can offer up values for any category, and a set of distributions (such as the set of installed distributions on `sys.path`) conceptually has an aggregation of these values.

The values associated with a category are a list of strings with the format:

```
name = prefix [ ":" suffix ] [ "[" flags "]" ]
```

where `name`, `prefix`, and `suffix` are `pkgnames`. `suffix` and `flags` are optional and `flags` follow the description in *Flag formats*.

Any installed distribution can offer up values for any category, and a set of distributions (such as the set of installed distributions on `sys.path`) conceptually has an aggregation of these values.

For callables, the `prefix` is the package or module name which contains the callable, `suffix` is the path to the callable in the module, and flags can be used for any purpose determined by the distribution author (for example, the `extras` feature in `distribute` / `setuptools`).

This entry format is used in the `distlib.scripts` package for installing scripts based on Python callables.

Note: In **PEP 426**, the `flags` value is limited to a single flag representing an extra (optional set of dependencies, for optional features of a distribution).

2.3.2 Distribution dependencies

You can use the `distlib.locators` package to locate the dependencies that a distribution has. The `distlib.database` package has code which allow you to analyse the relationships between a set of distributions:

- `make_graph()`, which generates a dependency graph from a list of distributions.
- `get_dependent_dists()`, which takes a list of distributions and a specific distribution in that list, and returns the distributions that are dependent on that specific distribution.
- `get_required_dists()`, which takes a list of distributions and a specific distribution in that list, and returns the distributions that are required by that specific distribution.

The graph returned by `make_graph()` is an instance of `DependencyGraph`.

2.3.3 Using the locators API

Overview

To locate a distribution in an index, we can use the `locate()` function. This returns a potentially downloadable distribution (in the sense that it has a download URL – of course, there are no guarantees that there will actually be a downloadable resource at that URL). The return value is an instance of `distlib.database.Distribution` which can be queried for any distributions it requires, so that they can also be located if desired. Here is a basic example:

```
>>> from distlib.locators import locate
>>> flask = locate('flask')
>>> flask
<Distribution Flask (0.10.1) [https://pypi.python.org/packages/source/F/Flask/Flask-0.10.1.tar.gz]>
>>> dependencies = [locate(r) for r in flask.run_requires]
>>> from pprint import pprint
>>> pprint(dependencies)
[<Distribution Werkzeug (0.9.1) [https://pypi.python.org/packages/source/W/Werkzeug/Werkzeug-0.9.1.ta
 <Distribution Jinja2 (2.7) [https://pypi.python.org/packages/source/J/Jinja2/Jinja2-2.7.tar.gz]>,
 <Distribution itsdangerous (0.21) [https://pypi.python.org/packages/source/i/itsdangerous/itsdanger
>>>
```

The values in the `run_requires` property are just strings. Here's another example, showing a little more detail:

```
>>> authy = locate('authy')
>>> authy.run_requires
set(['httplib2 (>= 0.7, < 0.8)', 'simplejson'])
>>> authy
<Distribution authy (1.0.0) [http://pypi.python.org/packages/source/a/authy/authy-1.0.0.tar.gz]>
>>> deps = [locate(r) for r in authy.run_requires]
>>> pprint(deps)
[<Distribution httplib2 (0.7.7) [http://pypi.python.org/packages/source/h/httplib2/httplib2-0.7.7.zi
 <Distribution simplejson (3.3.0) [http://pypi.python.org/packages/source/s/simplejson/simplejson-3.
>>>
```

Note that the constraints on the dependencies were honoured by `locate()`.

Under the hood

Under the hood, `locate()` uses *locators*. Locators are a mechanism for finding distributions from a range of sources. Although the `pypi` subpackage has been copied from `distutils2` to `distlib`, there may be benefits in a higher-level API, and so the `distlib.locators` package has been created as an experiment. Locators are objects which locate distributions. A locator instance's `get_project()` method is called, passing in a project name: The method returns a dictionary containing information about distribution releases found for that project. The keys of the returned dictionary are versions, and the values are instances of `distlib.database.Distribution`.

The following locators are provided:

- `DirectoryLocator` – this is instantiated with a base directory and will look for archives in the file system tree under that directory. Name and version information is inferred from the filenames of archives, and the amount of information returned about the download is minimal. The locator searches all subdirectories by default, but can be set to only look in the specified directory by setting the `recursive` keyword argument to `False`.
- `PyPIRPCLocator`. – This takes a base URL for the RPC service and will locate packages using PyPI's XML-RPC API. This locator is a little slow (the scraping interface seems to work faster) and case-sensitive. For example, searching for `'flask'` will throw up no results, but you get the expected results when searching from `'Flask'`. This appears to be a limitation of the underlying XML-RPC API. Note that 20 versions of a project

necessitate 41 network calls (one to get the versions, and two more for each version – one to get the metadata, and another to get the downloads information).

- PyPIJSONLocator. – This takes a base URL for the JSON service and will locate packages using PyPI's JSON API. This locator is case-sensitive. For example, searching for 'flask' will throw up no results, but you get the expected results when searching from 'Flask'. This appears to be a limitation of the underlying JSON API. Note that unlike the XML-RPC service, only non-hidden releases will be returned.
- SimpleScrapingLocator – this takes a base URL for the site to scrape, and locates packages using a similar approach to the PackageFinder class in pip, or as documented in the setuptools documentation as the approach used by easy_install.
- DistPathLocator – this takes a DistributionPath instance and locates installed distributions. This can be used with AggregatingLocator to satisfy requirements from installed distributions before looking elsewhere for them.
- JSONLocator – this uses an improved JSON metadata schema and returns data on all versions of a distribution, including dependencies, using a single network request.
- AggregatingLocator – this takes a list of other aggregators and delegates finding projects to them. It can either return the first result found (i.e. from the first aggregator in the list provided which returns a non-empty result), or a merged result from all the aggregators in the list.

There is a default locator, available at distlib.locators.default_locator.

The locators package also contains a function, get_all_distribution_names(), which retrieves the names of all distributions registered on PyPI:

```
>>> from distlib.locators import get_all_distribution_names
>>> names = get_all_distribution_names()
>>> len(names)
31905
>>>
```

This is implemented using the XML-RPC API.

Apart from JSONLocator, none of the locators currently returns enough metadata to allow dependency resolution to be carried out, but that is a result of the fact that metadata relating to dependencies are not indexed, and would require not just downloading the distribution archives and inspection of contained metadata files, but potentially also introspecting setup.py! This is the downside of having vital information only available via keyword arguments to the setup() call: hopefully, a move to fully declarative metadata will facilitate indexing it and allowing the provision of improved features.

The locators will skip binary distributions other than wheels. (.egg files are currently treated as binary distributions).

The PyPI locator classes don't yet support the use of mirrors, but that can be added in due course – once the basic functionality is working satisfactorily.

2.3.4 Using the index API

You can use the distlib.index package to perform operations relating to a package index compatible with PyPI. This includes things like registering a project, uploading a distribution or uploading documentation.

Overview

You access index functionality through an instance of the PackageIndex class. This is instantiated with the URL of the repository (which can be omitted if you want to use PyPI itself):

```
>>> from distlib.index import PackageIndex
>>> index = PackageIndex()
>>> index.url
'http://pypi.python.org/pypi'
```

To use a local test server, you might do this:

```
>>> index = PackageIndex('http://localhost:8080/')
```

Registering a project

Registering a project can be done using a `Metadata` instance which holds the index metadata used for registering. A simple example:

```
>>> from distlib.metadata import Metadata
>>> metadata = Metadata()
>>> metadata.name = 'tatterdemalion'
>>> metadata.version = '0.1'
>>> # other fields omitted
>>> response = index.register(metadata)
```

The `register()` method returns an HTTP response, such as might be returned by a call to `urlopen`. If an error occurs, a `HTTPError` will be raised. Otherwise, the `response.code` should be 200.

Uploading a source distribution

To upload a source distribution, you need to do the following as a minimum:

```
>>> metadata = ... # get a populated Metadata instance
>>> response = index.upload_file(metadata, archive_name)
```

The `upload_file()` method returns an HTTP response or, in case of error, raises an `HTTPError`.

Uploading binary distributions

When uploading binary distributions, you need to specify the file type and Python version, as in the following example:

```
>>> response = index.upload_file(metadata, archive_name,
...                              filetype='bdist_dumb',
...                              pyversion='2.6')
```

Signing a distribution

To sign a distribution, you will typically need GnuPG. The default implementation looks for `gpg` or `gpg2` on the path, but if not available there, you can can explicitly specify an absbolute path indicating where the signing program is to be found:

```
>>> index.gpg = '/path/to/gpg'
```

If the location of the signing key is not the default location, you can specify that too:

```
>>> index.gpg_home = '/path/to/keys'
```

where the `keys` folder will hold the GnuPG key database (files like `pubring.gpg`, `secring.gpg`, and `trustdb.gpg`).

Once these are set, you can sign the archive before uploading, as follows:

```
>>> response = index.upload_file(metadata, archive_name,
...                              signer='Test User',
...                              sign_password='secret')
```

When you sign a distribution, both the distribution and the signature are uploaded to the index.

Downloading files

The `PackageIndex` class contains a utility method which allows you to download distributions (and other files, such as signatures):

```
>>> index.download_file(url, destfile, digest=None, reporthook=None)
```

This is similar in function to `urlretrieve()` in the standard library. Provide a `digest` if you want the call to check that the has digest of the downloaded file matches a specific value: if not provided, no matching is done. The value passed can just be a plain string in the case of an MD5 digest or, if you want to specify the hashing algorithm to use, specify a tuple such as `('sha1', '0123456789abcdef...')`. The hashing algorithm must be one that's supported by the `hashlib` module.

Benefits to using this method over plain `urlretrieve()` are:

- It will use the `ssl_verifier`, if set, to ensure that the download is coming from where you think it is (see *Verifying HTTPS connections*).
- It will compute the digest as it downloads, saving you from having to read the whole of the downloaded file just to compute its digest.

Note that the url you download from doesn't actually need to be on the index – in theory, it could be from some other site. Note that if you have an `ssl_verifier` set on the index, it will perform its checks according to whichever `url` you supply – whether it's a resource on the index or not.

Verifying signatures

For any archive downloaded from an index, you can retrieve any signature by just appending `.asc` to the path portion of the download URL for the archive, and downloading that. The index class offers a `verify_signature()` method for validating a signature. Before invoking it, you may need to specify the location of the signing public key:

```
>>> index.gpg_home = '/path/to/keys'
```

If you have files 'good.bin', 'bad.bin' which are different from each other, and 'good.bin.asc' has the signature for 'good.bin', then you can verify signatures like this:

```
>>> index.verify_signature('good.bin.asc', 'good.bin')
True
>>> index.verify_signature('good.bin.asc', 'bad.bin')
False
```

Note that if you don't have the `gpg` or `gpg2` programs on the path, you may need to specify the location of the verifier program explicitly:

```
>>> index.gpg = '/path/to/gpg'
```

Some caveats about verified signatures

In order to be able to perform signature verification, you'll have to ensure that the public keys of whoever signed those distributions are in your key store (where you set `index.gpg_home` to point to). However, having these keys shouldn't give you a false sense of security; unless you can be sure that those keys actually belong to the people or organisations they purport to represent, the signature has no real value, even if it is verified without error. For you to be able to trust a key, it would need to be signed by someone you trust, who vouches for it – and this requires there to be either a signature from a valid certifying authority (e.g. Verisign, Thawte etc.) or a Web of Trust around the keys that you want to rely on.

An index may itself countersign distributions (so *it* deals with the keys of the distribution publishers, but you need only deal with the public signing key belonging to the index). If you trust the index, you can trust the verified signature if it's signed by the index.

Uploading documentation

To upload documentation, you need to specify the metadata and the directory which is the root of the documentation (typically, if you use Sphinx to build your documentation, this will be something like `<project>/docs/_build/html`):

```
>>> response = index.upload_documentation(metadata, doc_dir)
```

The `upload_documentation()` method returns an HTTP response or, in case of error, raises an `HTTPError`. The call will zip up the entire contents of the passed directory `doc_dir` and upload the zip file to the index.

Authentication

Operations which update the index (all of the above) will require authenticated requests. You can specify a username and password to use for requests sent to the index:

```
>>> index.username = 'test'
>>> index.password = 'secret'
```

For your convenience, these will be automatically read from any `.pypirc` file which you have; if it contains entries for multiple indexes, a `repository` key in `.pypirc` must match `index.url` to identify which username and password are to be read from `.pypirc`. Note that to ensure compatibility, `distlib` uses `distutils` code to read the `.pypirc` configuration. Thus, given the `.pypirc` file:

```
[distutils]
index-servers =
    pypi
    test

[pypi]
username: me
password: my_strong_password

[test]
repository: http://localhost:8080/
username: test
password: secret
```

you would see the following:

```
>>> index = PackageIndex()
>>> index.username
'me'
>>> index.password
'my_strong_password'
>>> index = PackageIndex('http://localhost:8080/')
>>> index.username
'test'
>>> index.password
'secret'
```

Verifying HTTPS connections

Although Python has full support for SSL, it does not, by default, verify SSL connections to servers. That's because in order to do so, a set of certificates which certify the identity of the server needs to be provided (see the relevant Python documentation for details).

Support for verifying SSL connections is provided in distlib through a handler, `distlib.util.HTTPSHandler`. To use it, set the `ssl_verifier` attribute of the index to a suitably configured instance. For example:

```
>>> from distlib.util import HTTPSHandler
>>> verifier = HTTPSHandler('/path/to/root/certs.pem')
>>> index.ssl_verifier = verifier
```

By default, the handler will attempt to match domains, including wildcard matching. This means that (for example) you access `foo.org` or `www.foo.org` which have a certificate for `*.foo.org`, the domains will match. If the domains don't match, the handler raises a `CertificateError` (a subclass of `ValueError`).

Domain mismatches can, however, happen for valid reasons. Say a hosting server `bar.com` hosts `www.foo.org`, which we are trying to access using SSL. If the server holds a certificate for `www.foo.org`, it will present it to the client, as long as both support Server Name Indication (SNI). While `distlib` supports SNI where Python supports it, Python 2.x does not include SNI support. For this or some other reason , you may wish to turn domain matching off. To do so, instantiate the verifier like this:

```
>>> verifier = HTTPSHandler('/path/to/root/certs.pem', False)
```

Ensuring that *only* HTTPS connections are made

You may want to ensure that traffic is *only* HTTPS for a particular interaction with a server – for example:

- Deal with a Man-In-The-Middle proxy server which listens on port 443 but talks HTTP rather than HTTPS
- Deal with situations where an index page obtained via HTTPS contains links with a scheme of `http` rather than `https`.

To do this, instead of using `HTTPSHandler` as shown above, use the `HTTPSOnlyHandler` class instead, which disallows any HTTP traffic. It's used in the same way as `HTTPSHandler`:

```
>>> from distlib.util import HTTPSOnlyHandler
>>> verifier = HTTPSOnlyHandler('/path/to/root/certs.pem')
>>> index.ssl_verifier = verifier
```

Note that with this handler, you can't make *any* HTTP connections at all - it will raise `URLError` if you try.

Getting hold of root certificates

At the time of writing, you can find a file in the appropriate format on the cURL website. Just download the `cacert.pem` file and pass the path to it when instantiating your verifier.

Saving a default configuration

If you don't have a `.pypirc` file but want to save one, you can do this by setting the username and password and calling the `save_configuration()` method:

```
>>> index = PackageIndex()
>>> index.username = 'fred'
>>> index.password = 'flintstone'
>>> index.save_configuration()
```

This will use `distutils` code to save a default `.pypirc` file which specifies a single index – PyPI – with the specified username and password.

2.3.5 Using the metadata and markers APIs

The metadata API is exposed through a `Metadata` class. This class can read and write metadata files complying with any of the defined versions: 1.0 (**PEP 241**), 1.1 (**PEP 314**), 1.2 (**PEP 345**) and 2.0 (**PEP 426**). It implements methods to parse and write metadata files.

Instantiating metadata

You can simply instantiate a `Metadata` instance and start populating it:

```
>>> from distlib.metadata import Metadata
>>> md = Metadata()
>>> md.name = 'foo'
>>> md.version = '1.0'
```

An instance so created may not be valid unless it has some minimal properties which meet certain constraints, as specified in PEP 426.

These constraints aren't applicable to legacy metadata. Therefore, when creating `Metadata` instances to deal with such metadata, you can specify the `scheme` keyword when creating the instance:

```
>>> legacy_metadata = Metadata(scheme='legacy')
```

The term 'legacy' is somewhat ambiguous, as it could refer to either the metadata format (legacy => key-value, non-legacy =< JSON as described in **PEP 426**) or the version specification (legacy => setuptools-compatible, non-legacy => as described in **PEP 440**). In this case, it refers to the **version scheme** and *not* the metadata format. Legacy metadata is also subject to constraints, but they are less stringent (for example, the name and version number are less constrained).

Whether dealing with current or legacy metadata. an instance's `validate()` method can be called to ensure that the metadata has no missing or invalid data. This raises a `DistlibException` (either `MetadataMissingError` or `MetadataInvalidError`) if the metadata isn't valid.

You can initialise an instance with a dictionary which conforms to **PEP 426** using the following form:

```
>>> metadata = Metadata(mapping=a_dictionary)
```

Reading metadata from files and streams

The Metadata class can be instantiated with the path of the metadata file. Here's an example with legacy metadata:

```
>>> from distlib.metadata import Metadata
>>> metadata = Metadata(path='PKG-INFO')
>>> metadata.name
'CLVault'
>>> metadata.version
'0.5'
>>> metadata.run_requires
['keyring']
```

Instead of using the path keyword argument to specify a file location, you can also specify a fileobj keyword argument to specify a file-like object which contains the data.

Writing metadata to paths and streams

Writing metadata can be done using the write method:

```
>>> metadata.write(path='/to/my/pydist.json')
```

You can also specify a file-like object to write to, using the fileobj keyword argument.

Using markers

Environment markers are implemented in the distlib.markers package and accessed via a single function, interpret().

See PEP 426 for more information about environment markers. The interpret() function takes a string argument which represents a Boolean expression, and returns either True or False:

```
>>> from distlib.markers import interpret
>>> interpret('python_version >= "1.0"')
True
```

You can pass in a context dictionary which is checked for values before the environment:

```
>>> interpret('python_version >= "1.0"', {'python_version': '0.5'})
False
```

You won't normally need to work with markers in this way – they are dealt with by the Metadata and Distribution logic when needed.

2.3.6 Using the resource API

You can use the distlib.resources package to access data stored in Python packages, whether in the file system or .zip files. Consider a package which contains data alongside Python code:

```
foofoo
-- bar
|   -- bar_resource.bin
|   -- baz.py
|   -- __init__.py
-- foo_resource.bin
-- __init__.py
-- nested
    -- nested_resource.bin
```

Access to resources in the file system

You can access these resources like so:

```
>>> from distlib.resources import finder
>>> f = finder('foofoo')
>>> r = f.find('foo_resource.bin')
>>> r.is_container
False
>>> r.size
10
>>> r.bytes
b'more_data\n'
>>> s = r.as_stream()
>>> s.read()
b'more_data\n'
>>> s.close()
>>> r = f.find('nested')
>>> r.is_container
True
>>> r.resources
{'nested_resource.bin'}
>>> r = f.find('nested/nested_resource.bin')
>>> r.size
12
>>> r.bytes
b'nested data\n'
>>> f = finder('foofoo.bar')
>>> r = f.find('bar_resource.bin')
>>> r.is_container
False
>>> r.bytes
b'data\n'
```

Access to resources in the `.zip` files

It works the same way if the package is in a .zip file. Given the zip file `foo.zip`:

```
$ unzip -l foo.zip
Archive:  foo.zip
  Length      Date    Time    Name
---------  ---------- -----   ----
       10  2012-09-20 21:34   foo/foo_resource.bin
        8  2012-09-20 21:42   foo/__init__.py
       14  2012-09-20 21:42   foo/bar/baz.py
        8  2012-09-20 21:42   foo/bar/__init__.py
        5  2012-09-20 21:33   foo/bar/bar_resource.bin
---------                     -------
       45                     5 files
```

You can access its resources as follows:

```
>>> import sys
>>> sys.path.append('foo.zip')
>>> from distlib.resources import finder
>>> f = finder('foo')
>>> r = f.find('foo_resource.bin')
>>> r.is_container
False
>>> r.size
10
>>> r.bytes
'more_data\n'
```

and so on.

2.3.7 Using the scripts API

You can use the `distlib.scripts` API to install scripts. Installing scripts is slightly more involved than just copying files:

- You may need to adjust shebang lines in scripts to point to the interpreter to be used to run scripts. This is important in virtual environments (venvs), and also in other situations where you may have multiple Python installations on a single computer.
- On Windows, on systems where the **PEP 397** launcher isn't installed, it is not easy to ensure that the correct Python interpreter is used for a script. You may wish to install native Windows executable launchers which run the correct interpreter, based on a shebang line in the script.

Specifying scripts to install

To install scripts, create a `ScriptMaker` instance, giving it the source and target directories for scripts:

```
>>> from distlib.scripts import ScriptMaker
>>> maker = ScriptMaker(source_dir, target_dir)
```

You can then install a script `foo.py` like this:

```
>>> maker.make('foo.py')
```

The string passed to make can take one of the following forms:

- A filename, relative to the source directory for scripts, such as `foo.py` or `subdir/bar.py`.
- A reference to a callable, given in the form:

  ```
  name = some_package.some_module:some_callable [flags]
  ```

  where the *flags* part is optional.

  For more information about flags, see *Flag formats*.

  Note that this format is exactly the same as for export entries in a distribution (see *Exporting things from Distributions*).

  When this form is passed to the `ScriptMaker.make()` method, a Python stub script is created with the appropriate shebang line and with code to load and call the specified callable with no arguments, returning its value as the return code from the script.

  You can pass an optional `options` dictionary to the `make()` method. This is meant to contain options which control script generation. The only option currently in use is `'gui'`, which indicates on Windows that a

Windows executable launcher (rather than a launcher which is a console application) should be used. (This only applies if `add_launchers` is true.)

For example, you can pass `{'gui': True}` to generate a windowed script.

Wrapping callables with scripts

Let's see how wrapping a callable works. Consider the following file:

```
 $ cat scripts/foo.py
   def main():
     print('Hello from foo')

 def other_main():
     print('Hello again from foo')

we can try wrapping ``main`` and ``other_main`` as callables::

   >>> from distlib.scripts import ScriptMaker
   >>> maker = ScriptMaker('scripts', '/tmp/scratch')
   >>> maker.make_multiple(('foo = foo:main', 'bar = foo:other_main'))
   ['/tmp/scratch/foo', '/tmp/scratch/bar']
   >>>

we can inspect the resulting scripts. First, ``foo``::

 $ ls /tmp/scratch/
 bar  foo
 $ cat /tmp/scratch/foo
 #!/usr/bin/python

 if __name__ == '__main__':
     import sys, re

     def _resolve(module, func):
         __import__(module)
         mod = sys.modules[module]
         parts = func.split('.')
         result = getattr(mod, parts.pop(0))
         for p in parts:
             result = getattr(result, p)
         return result

     try:
         sys.argv[0] = re.sub('-script.pyw?$', '', sys.argv[0])

         func = _resolve('foo', 'main')
         rc = func() # None interpreted as 0
     except Exception as e:  # only supporting Python >= 2.6
         sys.stderr.write('%s\n' % e)
         rc = 1
     sys.exit(rc)
```

The other script, `bar`, is different only in the essentials:

```
$ diff /tmp/scratch/foo /tmp/scratch/bar
16c16
<          func = _resolve('foo', 'main')
```

```
---
>         func = _resolve('foo', 'other_main')
```

Specifying a custom executable for shebangs

You may need to specify a custom executable for shebang lines. To do this, set the `executable` attribute of a `ScriptMaker` instance to the absolute Unicode path of the executable which you want to be written to the shebang lines of scripts. If not specified, the executable running the `ScriptMaker` code is used.

Generating variants of a script

When installing a script `foo`, it is not uncommon to want to install version-specific variants such as `foo3` or `foo-3.2`. You can control exactly which variants of the script get written through the `ScriptMaker` instance's `variants` attribute. This defaults to `set((", 'X.Y'))`, which means that by default a script `foo` would be installed as `foo` and `foo-3.2` under Python 3.2. If the value of the `variants` attribute were `set((", 'X', 'X.Y'))` then the `foo` script would be installed as `foo`, `foo3` and `foo-3.2` when run under Python 3.2.

Avoiding overwriting existing scripts

In some scenarios, you might overwrite existing scripts when you shouldn't. For example, if you use Python 2.7 to install a distribution with script `foo` in the user site (see **PEP 370**), you will write (on POSIX) scripts `~/.local/bin/foo` and `~/.local/bin/foo-2.7`. If you then install the same distribution with Python 3.2, you would write (on POSIX) scripts `~/.local/bin/foo` and `~/.local/bin/foo-3.2`. However, by overwriting the `~/.local/bin/foo` script, you may prevent verification or removal of the 2.7 installation to fail, because the overwritten file may be different (and so have a different hash from what was computed during the 2.7 installation).

To control overwriting of generated scripts this way, you can use the `clobber` attribute of a `ScriptMaker` instance. This is set to `False` by default, which prevents overwriting; to force overwriting, set it to `True`.

Generating windowed scripts on Windows

The `make()` and `make_multiple()` methods take an optional second `options` argument, which can be used to control script generation. If specified, this should be a dictionary of options. Currently, only the value for the `gui` key in the dictionary is inspected: if `True`, it generates scripts with `.pyw` extensions (rather than `.py`) and, if `add_launchers` is specified as `True` in the `ScriptMaker` instance, then (on Windows) a windowed native executable launcher is created (otherwise, the native executable launcher will be a console application).

2.3.8 Using the version API

Overview

The `NormalizedVersion` class implements a **PEP 426** compatible version:

```
>>> from distlib.version import NormalizedVersion
>>> v1 = NormalizedVersion('1.0')
>>> v2 = NormalizedVersion('1.0a1')
>>> v3 = NormalizedVersion('1.0b1')
>>> v4 = NormalizedVersion('1.0c1')
>>> v5 = NormalizedVersion('1.0.post1')
>>>
```

These sort in the expected order:

```
>>> v2 < v3 < v4 < v1 < v5
True
>>>
```

You can’t pass any old thing as a version number:

```
>>> NormalizedVersion('foo')
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
File "distlib/version.py", line 49, in __init__
 self._parts = parts = self.parse(s)
File "distlib/version.py", line 254, in parse
 def parse(self, s): return normalized_key(s)
File "distlib/version.py", line 199, in normalized_key
 raise UnsupportedVersionError(s)
distlib.version.UnsupportedVersionError: foo
>>>
```

Matching versions against constraints

The `NormalizedMatcher` is used to match version constraints against versions:

```
>>> from distlib.version import NormalizedMatcher
>>> m = NormalizedMatcher('foo (1.0b1)')
>>> m
NormalizedMatcher('foo (1.0b1)')
>>> [m.match(v) for v in v1, v2, v3, v4, v5]
[False, False, True, False, False]
>>>
```

Specifying `'foo (1.0b1)'` is equivalent to specifying `'foo (==1.0b1)'`, i.e. only the exact version is matched. You can also specify inequality constraints:

```
>>> m = NormalizedMatcher('foo (<1.0c1)')
>>> [m.match(v) for v in v1, v2, v3, v4, v5]
[False, True, True, False, False]
>>>
```

and multiple constraints:

```
>>> m = NormalizedMatcher('foo (>= 1.0b1, <1.0.post1)')
>>> [m.match(v) for v in v1, v2, v3, v4, v5]
[True, False, True, True, False]
>>>
```

You can do exactly the same thing as above with `setuptools`/ `distribute` version numbering (use `LegacyVersion` and `LegacyMatcher`) or with semantic versioning (use `SemanticVersion` and `SemanticMatcher`). However, you can’t mix and match versions of different types:

```
>>> from distlib.version import SemanticVersion, LegacyVersion
>>> nv = NormalizedVersion('1.0.0')
>>> lv = LegacyVersion('1.0.0')
>>> sv = SemanticVersion('1.0.0')
>>> lv == sv
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
File "distlib/version.py", line 61, in __eq__
```

```
 self._check_compatible(other)
File "distlib/version.py", line 58, in _check_compatible
raise TypeError('cannot compare %r and %r' % (self, other))
TypeError: cannot compare LegacyVersion('1.0.0') and SemanticVersion('1.0.0')
>>> nv == sv
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
File "distlib/version.py", line 61, in __eq__
 self._check_compatible(other)
File "distlib/version.py", line 58, in _check_compatible
raise TypeError('cannot compare %r and %r' % (self, other))
TypeError: cannot compare NormalizedVersion('1.0.0') and SemanticVersion('1.0.0')
>>>
```

2.3.9 Using the wheel API

You can use the `distlib.wheel` package to build and install from files in the Wheel format, defined in **PEP 427**.

Building wheels

Building wheels is straightforward:

```
from distlib.wheel import Wheel

wheel = Wheel()

# Set the distribution's identity
wheel.name = 'name_of_distribution'
wheel.version = '0.1'

# Indicate where the files to go in the wheel are to be found
paths = {
    'prefix': '/path/to/installation/prefix',
    'purelib': '/path/to/purelib',   # only one of purelib
    'platlib': '/path/to/platlib',   # or platlib should be set
    'scripts': '/path/to/scripts',
    'headers': '/path/to/headers',
    'data': '/path/to/data',
}

wheel.dirname = '/where/you/want/the/wheel/to/go'
# Now build
wheel.build(paths)
```

If the `'data'`, `'headers'` and `'scripts'` keys are absent, or point to paths which don't exist, nothing will be added to the wheel for these categories. The `'prefix'` key and one of `'purelib'` or `'platlib'` *must* be provided, and the paths referenced should exist.

Customising tags during build

By default, the `build()` method will use default tags depending on whether or not the build is a pure-Python build:

- For a pure-Python build, the `pyver` will be set to `pyXY` where `XY` is the version of the building Python. The `abi` tag will be `none` and the `arch` tag will be `any`.

- For a build which is not pure-Python (i.e. contains C code), the `pyver` will be set to e.g. `cpXY`, and the `abi` and `arch` tags will be set according to the building Python.

If you want to override these default tags, you can pass a `tags` parameter to the `build()` method which has the tags you want to declare. For example, for a pure build where we know that the code in the wheel will be compatible with the major version of the building Python:

```
from wheel import PYVER
tags = {
    'pyver': [PYVER[:-1], PYVER],
}
wheel.build(paths, tags)
```

This would set the `pyver` tags to be `pyX.pyXY` where `X` and `Y` relate to the building Python. You can similarly pass values using the `abi` and `arch` keys in the `tags` dictionary.

Specifying a wheel's version

You can also specify a particular "Wheel-Version" to be written to the wheel metadata of a wheel you're building. Simply pass a (major, minor) tuple in the `wheel_version` keyword argument to `build()`. If not specified, the most recent version supported is written.

Installing from wheels

Installing from wheels is similarly straightforward. You just need to indicate where you want the files in the wheel to be installed:

```
from distlib.wheel import Wheel
from distlib.scripts import ScriptMaker

wheel = Wheel('/path/to/my_dist-0.1-py32-none-any.whl')

# Indicate where the files in the wheel are to be installed to.
# All the keys should point to writable paths.
paths = {
    'prefix': '/path/to/installation/prefix',
    'purelib': '/path/to/purelib',
    'platlib': '/path/to/platlib',
    'scripts': '/path/to/scripts',
    'headers': '/path/to/headers',
    'data': '/path/to/data',
}

maker = ScriptMaker(None, None)
# You can specify a custom executable in script shebang lines, whether
# or not to install native executable launchers, whether to do a dry run
# etc. by setting attributes on the maker, wither when creating it or
# subsequently.

# Now install. The method accepts optional keyword arguments:
#
# - A ``warner`` argument which, if specified, should be a callable that
#   will be called with (software_wheel_version, file_wheel_version) if
#   they differ. They will both be in the form (major_ver, minor_ver).
#
# - A ``lib_only`` argument which indicates that only the library portion
#   of the wheel should be installed - no scripts, header files or
#   non-package data.

wheel.install(paths, maker)
```

Only one of the `purelib` or `platlib` paths will actually be written to (assuming that they are different, which isn't often the case). Which one it is depends on whether the wheel metadata declares that the wheel contains pure Python code.

Mounting wheels

One of Python's perhaps under-used features is `zipimport`, which gives the ability to import Python source from `.zip` files. Since wheels are `.zip` files, they can sometimes be used to provide functionality without needing to be installed. Whereas `.zip` files contain no convention for indicating compatibility with a particular Python, wheels *do* contain this compatibility information. Thus, it is possible to check if a wheel can be directly imported from, and the wheel support in `distlib` allows you to take advantage of this using the `mount()` and `unmount()` methods. When you mount a wheel, its absolute path name is added to `sys.path`, allowing the Python code in it to be imported. (A `DistlibException` is raised if the wheel isn't compatible with the Python which calls the `mount()` method.)

The `mount()` method takes an optional keyword parameter `append` which defaults to `False`, meaning the a mounted wheel's pathname is added to the beginning of `sys.path`. If you pass `True`, the pathname is appended to `sys.path`.

The `mount()` method goes further than just enabling Python imports – any C extensions in the wheel are also made available for import. For this to be possible, the wheel has to be built with additional metadata about extensions – a JSON file called `EXTENSIONS` which serialises an extension mapping dictionary. This maps extension module names to the names in the wheel of the shared libraries which implement those modules.

Running `unmount()` on the wheel removes its absolute pathname from `sys.path` and makes its C extensions, if any, also unavailable for import.

Using vanilla pip to build wheels for existing distributions on PyPI

Although work is afoot to add wheel support to `pip`, you don't need this to build wheels for existing PyPI distributions if you use `distlib`. The following script shows how you can use an unpatched, vanilla `pip` to build wheels:

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
#
# Copyright (C) 2013 Vinay Sajip. License: MIT
#

import logging
import optparse     # for 2.6
import os
import re
import shutil
import subprocess
import sys
import tempfile

logger = logging.getLogger('wheeler')

from distlib.compat import configparser, filter
from distlib.database import DistributionPath, Distribution, make_graph
from distlib.locators import (JSONLocator, SimpleScrapingLocator,
                              AggregatingLocator, DependencyFinder)
```

```
from distlib.manifest import Manifest
from distlib.metadata import Metadata
from distlib.util import parse_requirement, get_package_data
from distlib.wheel import Wheel

EGG_INFO_RE = re.compile(r'(-py\d\.\d)?\.egg-info', re.I)

INSTALLED_DISTS = DistributionPath(include_egg=True)


def get_requirements(data):
    lines = []
    for line in data.splitlines():
        line = line.strip()
        if not line or line[0] == '#':
            continue
        lines.append(line)
    reqts = []
    extras = {}
    result = {'install': reqts, 'extras': extras}
    for line in lines:
        if line[0] != '[':
            reqts.append(line)
        else:
            i = line.find(']', 1)
            if i < 0:
                raise ValueError('unrecognised line: %r' % line)
            extra = line[1:i]
            extras[extra] = reqts = []
    return result


def convert_egg_info(libdir, prefix, options):
    files = os.listdir(libdir)
    ei = list(filter(lambda d: d.endswith('.egg-info'), files))[0]
    olddn = os.path.join(libdir, ei)
    di = EGG_INFO_RE.sub('.dist-info', ei)
    newdn = os.path.join(libdir, di)
    os.rename(olddn, newdn)
    if options.compatible:
        renames = {}
    else:
        renames = {
            'entry_points.txt': 'EXPORTS',
        }
    excludes = set([
        'SOURCES.txt',          # of no interest in/post WHEEL
        'installed-files.txt',  # replaced by RECORD, so not needed
        'requires.txt',         # added to METADATA, so not needed
        'PKG-INFO',             # replaced by METADATA
        'not-zip-safe',         # not applicable
    ])
    files = os.listdir(newdn)
    metadata = mdname = reqts = None
    for oldfn in files:
        pn = os.path.join(newdn, oldfn)
        if oldfn in renames:
            os.rename(pn, os.path.join(newdn, renames[oldfn]))
        else:
            if oldfn == 'requires.txt':
                with open(pn, 'r') as f:
                    reqts = get_requirements(f.read())
            elif oldfn == 'PKG-INFO':
                metadata = Metadata(path=pn)
                pd = get_package_data(metadata.name, metadata.version)
                metadata = Metadata(mapping=pd['index-metadata'])
                mdname = os.path.join(newdn, 'pydist.json')
            if oldfn in excludes or not options.compatible:
                os.remove(pn)
    if metadata:
        # Use Metadata 1.2 or later
        metadata.provides += ['%s (%s)' % (metadata.name,
                                           metadata.version)]
        # Update if not set up by get_package_data
        if reqts and not metadata.run_requires:
            metadata.dependencies = reqts
        metadata.write(path=mdname)
    manifest = Manifest(os.path.dirname(libdir))
    manifest.findall()
    paths = manifest.allfiles
    dp = DistributionPath([libdir])
    dist = next(dp.get_distributions())
    dist.write_installed_files(paths, prefix)


def install_dist(distname, workdir, options):
    pfx = '--install-option='
    purelib = pfx + '--install-purelib=%s/purelib' % workdir
    platlib = pfx + '--install-platlib=%s/platlib' % workdir
    headers = pfx + '--install-headers=%s/headers' % workdir
    scripts = pfx + '--install-scripts=%s/scripts' % workdir
    data = pfx + '--install-data=%s/data' % workdir
    # Use the pip adjacent to sys.executable, if any (for virtualenvs)
    d = os.path.dirname(sys.executable)
    files = filter(lambda o: o in ('pip', 'pip.exe'), os.listdir(d))
    if not files:
        prog = 'pip'
    else:
        prog = os.path.join(d, next(files))
    cmd = [prog, 'install',
           '--no-deps', '--quiet',
           '--index-url', 'http://pypi.python.org/simple/',
           '--timeout', '3', '--default-timeout', '3',
           purelib, platlib, headers, scripts, data, distname]
    result = {
        'scripts': os.path.join(workdir, 'scripts'),
        'headers': os.path.join(workdir, 'headers'),
        'data': os.path.join(workdir, 'data'),
    }
    print('Pipping %s ...' % distname)
    p = subprocess.Popen(cmd, shell=False, stdout=sys.stdout,
                         stderr=subprocess.STDOUT)
    stdout, _ = p.communicate()
    if p.returncode:
        raise ValueError('pip failed to install %s:\n%s' % (distname, stdout))
    for dn in ('purelib', 'platlib'):
        libdir = os.path.join(workdir, dn)
        if os.path.isdir(libdir):
            result[dn] = libdir
            break
    convert_egg_info(libdir, workdir, options)
    dp = DistributionPath([libdir])
    dist = next(dp.get_distributions())
    md = dist.metadata
    result['name'] = md.name
    result['version'] = md.version
    return result


def build_wheel(distname, options):
    result = None
    r = parse_requirement(distname)
    if not r:
        print('Invalid requirement: %r' % distname)
    else:
        dist = INSTALLED_DISTS.get_distribution(r.name)
        if dist:
            print('Can\'t build a wheel from already-installed '
                  'distribution %s' % dist.name_and_version)
        else:
            workdir = tempfile.mkdtemp()    # where the Wheel input files will live
            try:
                paths = install_dist(distname, workdir, options)
                paths['prefix'] = workdir
                wheel = Wheel()
                wheel.name = paths.pop('name')
                wheel.version = paths.pop('version')
                wheel.dirname = options.destdir
                wheel.build(paths)
                result = wheel
            finally:
                shutil.rmtree(workdir)
    return result


def main(args=None):
    parser = optparse.OptionParser(usage='%prog [options] requirement [requirement ...]')
    parser.add_option('-d', '--dest', dest='destdir', metavar='DESTDIR',
                      default=os.getcwd(), help='Where you want the wheels '
                      'to be put.')
    parser.add_option('-n', '--no-deps', dest='deps', default=True,
                      action='store_false',
                      help='Don\'t build dependent wheels.')
    options, args = parser.parse_args(args)
    options.compatible = True   # may add flag to turn off later
    if not args:
        parser.print_usage()
    else:
        # Check if pip is available; no point in continuing, otherwise
        try:
            with open(os.devnull, 'w') as f:
                p = subprocess.call(['pip', '--version'], stdout=f, stderr=subprocess.STDOUT)
        except Exception:
            p = 1
        if p:
            print('pip appears not to be available. Wheeler needs pip to '
                  'build  wheels.')
            return 1
        if options.deps:
            # collect all the requirements, including dependencies
            u = 'http://pypi.python.org/simple/'
            locator = AggregatingLocator(JSONLocator(),
                                         SimpleScrapingLocator(u, timeout=3.0),
                                         scheme='legacy')
            finder = DependencyFinder(locator)
            wanted = set()
            for arg in args:
                r = parse_requirement(arg)
                if not r.constraints:
                    dname = r.name
                else:
                    dname = '%s (%s)' % (r.name, ', '.join(r.constraints))
                print('Finding the dependencies of %s ...' % arg)
                dists, problems = finder.find(dname)
                if problems:
                    print('There were some problems resolving dependencies '
                          'for %r.' % arg)
                    for _, info in problems:
                        print('  Unsatisfied requirement %r' % info)
                wanted |= dists
            want_ordered = True     # set to False to skip ordering
            if not want_ordered:
                wanted = list(wanted)
            else:
                graph = make_graph(wanted, scheme=locator.scheme)
                slist, cycle = graph.topological_sort()
                if cycle:
                    # Now sort the remainder on dependency count.
                    cycle = sorted(cycle, reverse=True,
                                   key=lambda d: len(graph.reverse_list[d]))
                wanted = slist + cycle

                # get rid of any installed distributions from the list
                for w in list(wanted):
                    dist = INSTALLED_DISTS.get_distribution(w.name)
                    if dist or w.name in ('setuptools', 'distribute'):
                        wanted.remove(w)
                        s = w.name_and_version
                        print('Skipped already-installed distribution %s' % s)

            # converted wanted list to pip-style requirements
            args = ['%s==%s' % (dist.name, dist.version) for dist in wanted]

        # Now go build
        built = []
        for arg in args:
            wheel = build_wheel(arg, options)
            if wheel:
                built.append(wheel)
        if built:
            if options.destdir == os.getcwd():
                dest = ''
            else:
                dest = ' in %s' % options.destdir
            print('The following wheels were built%s:' % dest)
            for wheel in built:
                print('  %s' % wheel.filename)

if __name__ == '__main__':
    logging.basicConfig(format='%(levelname)-8s %(name)s %(message)s',
                        filename='wheeler.log', filemode='w')
    try:
        rc = main()
    except Exception as e:
        print('Failed - sorry! Reason: %s\nPlease check the log.' % e)
        logger.exception('Failed.')
        rc = 1
    sys.exit(rc)
```

This script, `wheeler.py`, is also available here. Note that by default, it downloads dependencies of any distribution you specify and builds separate wheels for each distribution. It's smart about not repeating work if dependencies are common across multiple distributions you specify:

```
$ python wheeler.py sphinx flask
Finding the dependencies of sphinx ...
Finding the dependencies of flask ...
Pipping Jinja2==2.6 ...
Pipping docutils==0.10 ...
Pipping Pygments==1.6 ...
Pipping Werkzeug==0.8.3 ...
Pipping Sphinx==1.1.3 ...
Pipping Flask==0.9 ...
The following wheels were built:
  Jinja2-2.6-py27-none-any.whl
  docutils-0.10-py27-none-any.whl
  Pygments-1.6-py27-none-any.whl
  Werkzeug-0.8.3-py27-none-any.whl
  Sphinx-1.1.3-py27-none-any.whl
  Flask-0.9-py27-none-any.whl
```

Note that the common dependency – `Jinja2` – was only built once.

You can opt to not build dependent wheels by specifying `--no-deps` on the command line.

Note that the script also currently uses an http: URL for PyPI – this may need to change to an https: URL in the future.

Note: It can't be used to build wheels from existing distributions, as `pip` will either refuse to install to custom locations (because it views a distribution as already installed), or will try to upgrade and thus uninstall the existing distribution, even though installation is requested to a custom location (and uninstallation is not desirable). For best results, run it in a fresh venv:

$ my_env/bin/python wheeler.py some_dist

It should use the venv's `pip`, if one is found.

2.3.10 Using the manifest API

You can use the `distlib.manifest` API to construct lists of files when creating distributions. This functionality is an improved version of the equivalent functionality in `distutils`, where it was not a public API.

You can create instances of the `Manifest` class to work with a set of files rooted in a particular directory:

```
>>> from distlib.manifest import Manifest
>>> manifest = Manifest('/path/to/my/sources')
```

This sets the `base` attribute to the passed in root directory. You can add one or multiple files using names relative to the base directory:

```
>>> manifest.add('abc')
>>> manifest.add_many(['def', 'ghi'])
```

As a result of the above two statements, the manifest will consist of `'/path/to/my/sources/abc'`, `'/path/to/my/sources/def'` and `'/path/to/my/sources/ghi'`. No check is made regarding the existence of these files.

You can get all the files below the base directory of the manifest:

```
>>> manifest.findall()
```

This will populate the `allfiles` attribute of `manifest` with a list of all files in the directory tree rooted at the base. However, the manifest is still empty:

```
>>> manifest.files
>>> set()
```

You can populate the manifest – the `files` attribute – by running a number of *directives*, using the `process_directive()` method. Each directive will either add files from `allfiles` to `files`, or remove files from `allfiles` if they were added by a previous directive. A directive is a string which must have a specific syntax: malformed lines will result in a `DistlibException` being raised. The following directives are available: they are compatible with the syntax of `MANIFEST.in` files processed by `distutils`.

Consider the following directory tree:

```
testsrc/
-- keep
|   -- keep.txt
-- LICENSE
-- README.txt
-- subdir
    -- lose
    |   -- lose.txt
    -- somedata.txt
    -- subsubdir
        -- somedata.bin
```

This will be used to illustrate how the directives work, in the following sections.

The `include` directive

This takes the form of the word `include` (case-sensitive) followed by a number of file-name patterns (as used in `MANIFEST.in` in `distutils`). All files in `allfiles`` matching the patterns (considered relative to the base directory) are added to `files`. For example:

```
>>> manifest.process_directive('include R*.txt LIC* keep/*.txt')
```

This will add `README.txt`, `LICENSE` and `keep/keep.txt` to the manifest.

The `exclude` directive

This takes the form of the word `exclude` (case-sensitive) followed by a number of file-name patterns (as used in `MANIFEST.in` in `distutils`). All files in `files`` matching the patterns (considered relative to the base directory) are removed from `files`. For example:

```
>>> manifest.process_directive('exclude LIC*')
```

This will remove 'LICENSE' from the manifest, as it was added in the section above.

The `global-include` directive

This works just like `include`, but will add matching files at all levels of the directory tree:

```
>>> manifest.process_directive('global-include *.txt')
```

This will add `subdir/somedata.txt` and `subdir/lose/lose.txt` from the manifest.

The `global-exclude` directive

This works just like `exclude`, but will remove matching files at all levels of the directory tree:

```
>>> manifest.process_directive('global-exclude l*.txt')
```

This will remove `subdir/lose/lose.txt` from the manifest.

The `recursive-include` directive

This directive takes a directory name (relative to the base) and a set of patterns. The patterns are used as in `global-include`, but only for files under the specified directory:

```
>>> manifest.process_directive('recursive-include subdir l*.txt')
```

This will add `subdir/lose/lose.txt` back to the manifest.

The `recursive-exclude` directive

This works like `recursive-include`, but excludes matching files under the specified directory if they were already added by a previous directive:

```
>>> manifest.process_directive('recursive-exclude subdir lose*')
```

This will remove `subdir/lose/lose.txt` from the manifest again.

The `graft` directive

This directive takes the name of a directory (relative to the base) and copies all the names under it from `allfiles` to `files`.

The `prune` directive

This directive takes the name of a directory (relative to the base) and removes all the names under it from `files`.

2.4 Next steps

You might find it helpful to look at information about *Distlib's design* – or peruse the *API Reference*.

CHAPTER 3

Distlib's design

This is the section containing some discussion of how `distlib`'s design was arrived at, as and when time permits.

3.1 The `locators` API

This section describes the design of the `distlib` API relating to accessing distribution metadata, whether stored locally or in indexes like PyPI.

3.1.1 The problem we're trying to solve

People who use distributions need to locate, download and install them. Distributions can be found in a number of places, such as:

- An Internet index such as The Python Packages Index (PyPI), or a mirror thereof.
- Other Internet resources, such as the developer's website, or a source code repository such as GitHub, BitBucket, Google Code or similar.
- File systems, whether local to one computer or shared between several.
- Distributions which have already been installed, and are available in the `sys.path` of a running Python interpreter.

When we're looking for distributions, we don't always know exactly what we want: often, we just want the latest version, but it's not uncommon to want a specific older version, or perhaps the most recent version that meets some constraints on the version. Since we need to be concerned with matching versions, we need to consider the version schemes in use (see *The version API*).

It's useful to separate the notion of a *project* from a distribution: The project is the version-independent part of the distribution, i.e. it's described by the *name* of the distribution and encompasses all released distributions which use that name.

We often don't just want a single distribution, either: a common requirement, when installing a distribution, is to locate all distributions that it relies on, which aren't already installed. So we need a *dependency finder*, which itself needs to locate depended-upon distributions, and recursively search for dependencies until all that are available have been found.

We may need to distinguish between different types of dependencies:

- Post-installation dependencies. These are needed by the distribution after it has been installed, and is in use.
- Build dependencies. These are needed for building and/or installing the distribution, but are not needed by the distribution itself after installation.
- Test dependencies. These are only needed for testing the distribution, but are not needed by the distribution itself after installation.

When testing a distribution, we need all three types of dependencies. When installing a distribution, we need the first two, but not the third.

3.1.2 A minimal solution

Locating distributions

It seems that the simplest API to locate a distribution would look like `locate(requirement)`, where `requirement` is a string giving the distribution name and optional version constraints. Given that we know that distributions can be found in different places, it's best to consider a `Locator` class which has a `locate()` method with a corresponding signature, with subclasses for each of the different types of location that distributions inhabit. It's also reasonable to provide a default locator in a module attribute `default_locator`, and a module-level `locate()` function which calls the `locate()` method on the default locator.

Since we'll often need to locate all the versions of a project before picking one, we can imagine that a locator would need a `get_project()` method for fetching all versions of a project; and since we will be likely to want to use caching, we can assume there will be a `_get_project()` method to do the actual work of fetching the version data, which the higher-level `get_project()` will call (and probably cache). So our locator base class will look something like this:

```
class Locator(object):
    """
    Locate distributions.
    """

    def __init__(self, version_scheme='default'):
        """
        Initialise a locator with the specified version scheme.
        """

    def locate(self, requirement):
        """
        Locate the highest-version distribution which satisfies
        the constraints in ``requirement``, and return a
        ``Distribution`` instance if found, or else ``None``.
        """

    def get_project(self, name):
        """
        Return all known distributions for a project named ``name``,
        returning a dictionary mapping version to ``Distribution``
        instance, or an empty dictionary if nothing was found.
        Use _get_project to do the actual work, and cache the results for
        future use.
        """

    def _get_project(self, name):
        """
        Return all known distributions for a project named ``name``,
        returning a dictionary mapping version to ``Distribution``
        instance, or an empty dictionary if nothing was found.
        """
```

When attempting to `locate()`, it would be useful to pass requirement information to `get_project()` / `_get_project()`. This can be done in a `matcher` attribute which is normally `None` but set to a `distlib.version.Matcher` instance when a `locate()` call is in progress.

Finding dependencies

A dependency finder will depend on a locator to locate dependencies. A simple approach will be to consider a `DependencyFinder` class which takes a locator as a constructor argument. It might look something like this:

```
class DependencyFinder(object):
    """
    Locate dependencies for distributions.
    """

    def __init__(self, locator):
        """
        Initialise an instance, using the specified locator
        to locate distributions.
        """

    def find(self, requirement, meta_extras=None, prereleases=False):
        """
        Find a distribution matching requirement and all distributions
        it depends on. Use the ``meta_extras`` argument to determine
        whether distributions used only for build, test etc. should be
        included in the results. Allow ``requirement`` to be either a
        :class:`Distribution` instance or a string expressing a
        requirement. If ``prereleases`` is True, treat pre-releases as
        normal releases; otherwise only return pre-releases if they're
        all that's available.

        Return a set of :class:`Distribution` instances and a set of
        problems.

        The distributions returned should be such that they have the
        :attr:`required` attribute set to ``True`` if they were
        from the ``requirement`` passed to ``find()``, and they have the
        :attr:`build_time_dependency` attribute set to ``True`` unless they
        are post-installation dependencies of the ``requirement``.

        The problems should be a tuple consisting of the string
        ``'unsatisfied'`` and the requirement which couldn't be satisfied
        by any distribution known to the locator.
        """
```

3.2 The `index` API

This section describes the design of the `distlib` API relating to performing certain operations on Python package indexes like PyPI. Note that this API does not support *finding* distributions - the `locators` API is used for that.

3.2.1 The problem we're trying to solve

Operations on a package index that are commonly performed by distribution developers are:

- Register projects on the index.
- Upload distributions relating to projects on the index, with support for signed distributions.
- Upload documentation relating to projects.

Less common operations are:

- Find a list of hosts which mirror the index.
- Save a default .pypirc file with default username and password to use.

3.2.2 A minimal solution

The distutils approach was to have several separate command classes called `register`, `upload` and `upload_doc`, where really all that was needed was some methods. That's the approach `distlib` takes, by implementing a `PackageIndex` class with `register()`, `upload_file()` and `upload_documentation()` methods. The `PackageIndex` class contains no user interface code whatsoever: that's assumed to be the domain of the packaging tool. The packaging tool is expected to get the required information from a user using whatever means the developers of that tool deem to be the most appropriate; the required attributes are then set on the `PackageIndex` instance. (Examples of this kind of information: user name, password, whether the user wants to save a default configuration, where the signing program and its keys live.)

The minimal interface to provide the required functionality thus looks like this:

```
class PackageIndex(object):
    def __init__(self, url=None, mirror_host=None):
        """
        Initialise an instance using a specific index URL, and
        a DNS name for a mirror host which can be used to
        determine available mirror hosts for the index.
        """

    def save_configuration(self):
        """
        Save the username and password attributes of this
        instance in a default .pypirc file.
        """
    def register(self, metadata):
        """
        Register a project on the index, using the specified metadata.
        """

    def upload_file(self, metadata, filename, signer=None,
                    sign_password=None, filetype='sdist',
                    pyversion='source'):
        """
        Upload a distribution file to the index using the
        specified metadata to identify it, with options
        for signing and for binary distributions which are
        specific to Python versions.
        """

    def upload_documentation(self, metadata, doc_dir):
        """
        Upload documentation files in a specified directory
        using the specified metadata to identify it, after
        archiving the directory contents into a .zip file.
        """
```

The following additional attributes can be identified on `PackageIndex` instances:

- `username` - the username to use for authentication.
- `password` - the password to use for authentication.
- `mirrors` (read-only) - a list of hostnames of known mirrors.

3.3 The `resources` API

This section describes the design of the `distlib` API relating to accessing 'resources', which is a convenient label for data files associated with Python packages.

3.3.1 The problem we're trying to solve

Developers often have a need to co-locate data files with their Python packages. Examples of these might be:

- Templates, commonly used in web applications
- Translated messages used in internationalisation/localisation

The stdlib does not provide a uniform API to access these resources. A common approach is to use `__file__` like this:

```
base = os.path.dirname(__file__)
data_filename = os.path.join(base, 'data.bin')
with open(data_filename, 'rb') as f:
    # read the data from f
```

However, this approach fails if the package is deployed in a .zip file.

To consider how to provide a minimal uniform API to access resources in Python packages, we'll assume that the requirements are as follows:

- All resources are regarded as binary. The consuming application is expected to know how to convert resources to text, where appropriate.
- All resources are read-only.
- It should be possible to access resources either as streams, or as their entire data as a byte-string.
- Resources will have a unique, identifying name which is text. Resources will be hierarchical and named using filesystem-like paths using '/' as a separator. The library will be responsible for converting resource names to the names of the underlying representations (e.g. encoding of file names corresponding to resource names).
- Some resources are containers of other resources, some are not. For example, a resource `nested/nested_resource.bin` in a package would not contain other resources, but implies the existence of a resource `nested`, which contains `nested_resource.bin`.
- Resources can only be associated with packages, not with modules. That's because with peer modules `a.py` and `b.py`, there's no obvious location for data associated only with `a`: both `a` and `b` are in the same directory. With a package, there's no ambiguity, as a package is associated with a specific directory, and no other package can be associated with that directory.

- Support should be provided for access to data deployed in the file system or in packages contained in .zip files, and third parties should be able to extend the facilities to work with other storage formats which support import of Python packages.
- It should be possible to access the contents of any resource through a file on the file system. This is to cater for any external APIs which need to access the resource data as files (examples would be a shared library for linking using `dlopen()` on POSIX, or any APIs which need access to resource data via OS-level file handles rather than Python streams).

3.3.2 A minimal solution

We know that we will have to deal with resources, so it seems natural that there would be a `Resource` class in the solution. From the requirements, we can see that a `Resource` would have the following:

- A `name` property identifying the resource.
- A `as_stream` method allowing access to the resource data as a binary stream. This is not a property, because a new stream is returned each time this method is called. The returned stream should be closed by the caller.
- A `bytes` property returning the entire contents of the resource as a byte string.
- A `size` property indicating the size of the resource (in bytes).
- An `is_container` property indicating whether the resource is a container of other resources.
- A `resources` property returning the names of resources contained within the resource.

The `Resource` class would be the logical place to perform sanity checks which relate to all resources. For example:

- It doesn't make sense to ask for the `bytes` or `size` properties or call the `as_stream` method of a container resource.
- It doesn't make sense to ask for the `resources` property of a resource which is *not* a container.

It seems reasonable to raise exceptions for incorrect property or method accesses.

We know that we need to support resource access in the file system as well as .zip files, and to support other sources of storage which might be used to import Python packages. Since import and loading of Python packages happens through **PEP 302** importers and loaders, we can deduce that the mechanism used to find resources in a package will be closely tied to the loader for that package.

We could consider an API for finding resources in a package like this:

```
def find_resource(pkg, resource_name):
    # return a Resource instance for the resource
```

and then use it like this:

```
r1 = find_resource(pkg, 'foo')
r2 = find_resource(pkg, 'bar')
```

However, we'll often have situations where we will want to get multiple resources from a package, and in certain applications we might want to implement caching or other processing of resources before returning them. The above API doesn't facilitate this, so let's consider delegating the finding of resources in a package to a *finder* for that package. Once we get a finder, we can hang on to it and ask it to find multiple resources. Finders can be extended to provide whatever caching and preprocessing an application might need.

To get a finder for a package, let's assume there's a `finder` function:

```
def finder(pkg):
    # return a finder for the specified package
```

We can use it like this:

```
f = finder(pkg)
r1 = f.find('foo')
r2 = f.find('bar')
```

The `finder` function knows what kind of finder to return for a particular package through the use of a registry. Given a package, `finder` can determine the loader for that package, and based on the type of loader, it can instantiate the right kind of finder. The registry maps loader types to callables that return finders. The callable is called with a single argument – the Python module object for the package.

Given that we have finders in the design, we can identify `ResourceFinder` and `ZipResourceFinder` classes for the two import systems we're going to support. We'll make `ResourceFinder` a concrete class rather than an interface - it'll implement getting resources from packages stored in the file system. `ZipResourceFinder` will be a subclass of `ResourceFinder`.

Since there is no loader for file system packages when the C-based import system is used, the registry will come with the following mappings:

- `type(None)` -> `ResourceFinder`
- `_frozen_importlib.SourceFileLoader -> ``ResourceFinder`
- `zipimport.zipimporter` -> `ZipResourceFinder`

Users of the API can add new or override existing mappings using the following function:

```
def register_finder(loader, finder_maker):
    # register ``finder_maker`` to make finders for packages with a loader
    # of the same type as ``loader``.
```

Typically, the `finder_maker` will be a class like `ResourceFinder` or `ZipResourceFinder`, but it can be any callable which takes the Python module object for a package and returns a finder.

Let's consider in more detail what finders look like and how they interact with the `Resource` class. We'll keep the Resource class minimal; API users never instantiate `Resource` directly, but call a finder's `find` method to return a `Resource` instance. A finder could return an instance of a `Resource` subclass if really needed, though it shouldn't be necessary in most cases. If a finder can't find a resource, it should return `None`.

The Resource constructor will look like this:

```
def __init__(self, finder, name):
    self.finder = finder
    self.name = name
    # other initialisation, not specified
```

and delegate as much work as possible to its finder. That way, new import loader types can be supported just by implementing a suitable `XXXResourceFinder` for that loader type.

What a finder needs to do can be exemplified by the following skeleton for `ResourceFinder`:

```
class ResourceFinder(object):
    def __init__(self, module):
        "Initialise finder for the specified package"

    def find(self, resource_name):
        "Find and return a ``Resource`` instance or ``None``"

    def is_container(self, resource):
        "Return whether resource is a container"

    def get_bytes(self, resource):
        "Return the resource's data as bytes"

    def get_size(self, resource):
        "Return the size of the resource's data in bytes"

    def get_stream(self, resource):
        "Return the resource's data as a binary stream"

    def get_resources(self, resource):
        """
        Return the resources contained in this resource as a set of
        (relative) resource names
        """
```

3.3.3 Dealing with the requirement for access via file system files

To cater for the requirement that the contents of some resources be made available via a file on the file system, we'll assume a simple caching solution that saves any such resources to a local file system cache, and returns the filename of the resource in the cache. We need to divide the work between the finder and the cache. We'll deliver the cache function through a `Cache` class, which will have the following methods:

- A constructor which takes an optional base directory for the cache. If none is provided, we'll construct a base directory of the form:

  `<rootdir>/.distlib/resource-cache`

  where `<rootdir>` is the user's home directory. On Windows, if the environment specifies a variable named `LOCALAPPDATA`, its value will be used as `<rootdir>` – otherwise, the user's home directory will be used.

- A `get()` method which takes a `Resource` and returns a file system filename, such that the contents of that named file will be the contents of the resource.

- An `is_stale()` method which takes a `Resource` and its corresponding file system filename, and returns whether the file system file is stale when compared with the resource. Knowing that cache invalidation is hard, the default implementation just returns `True`.

- A `prefix_to_dir()` method which converts a prefix to a directory name. We'll assume that for the cache, a resource path can be divided into two parts: the *prefix* and the *subpath*. For resources in a .zip file, the prefix would be the pathname of the archive, while the subpath would be the path inside the archive. For a file system resource, since it is already in the file system, the prefix would be `None` and the subpath would be the absolute path name of the resource. The `prefix_to_dir()` method's job is to convert a prefix (if not `None`) to a subdirectory in the cache that holds the cached files for all resources with that prefix. We'll delegate the determination of a resource's prefix and subpath to its finder, using a `get_cache_info()` method on finders, which takes a `Resource` and returns a (`prefix`, `subpath`) tuple.

  The default implementation will use `os.splitdrive()` to see if there's a Windows drive, if present, and convert its `':'` to `'---'`. The rest of the prefix will be converted by replacing `'/'` by `'--'`, and appending `'.cache'` to the result.

The cache will be activated when the `file_path` property of a `Resource` is accessed. This will be a cached property, and will call the cache's `get()` method to obtain the file system path.

3.4 The `scripts` API

This section describes the design of the `distlib` API relating to installing scripts.

3.4.1 The problem we're trying to solve

Installing scripts is slightly more involved than simply copying files from source to target, for the following reasons:

- On POSIX systems, scripts need to be made executable. To cater for scenarios where there are multiple Python versions installed on a computer, scripts need to have their shebang lines adjusted to point to the correct interpreter. This requirement is commonly found when virtual environments (venvs) are in use, but also in other multiple-interpreter scenarios.
- On Windows systems, which don't support shebang lines natively, some alternate means of finding the correct interpreter need to be provided. Following the acceptance and implementation of PEP 397, a shebang- interpreting launcher will be available in Python 3.3 and later and a standalone version of it for use with earlier Python versions is also available. However, where this can't be used, an alternative approach using executable launchers installed with the scripts may be necessary. (That is the approach taken by `setuptools`.) Windows also has two types of launchers - console applications and Windows applications. The appropriate launcher needs to be used for scripts.
- Some scripts are effectively just callable code in a Python package, with boilerplate for importing that code, calling it and returning its return value as the script's return code. It would be useful to have the boilerplate standardised, so that developers need just specify which callables to expose as scripts, and under what names, using e.g. a `name = callable` syntax. (This is the approach taken by `setuptools` for the popular `console_scripts` feature).

3.4.2 A minimal solution

Script handling in `distutils` and `setuptools` is done in two phases: 'build' and 'install'. Whether a particular packaging tool chooses to do the 'heavy lifting' of script creation (i.e. the things referred to above, beyond simple copying) in 'build' or 'install' phases, the job is the same. To abstract out just the functionality relating to scripts, in an extensible way, we can just delegate the work to a class, unimaginatively called `ScriptMaker`. Given the above requirements, together with the more basic requirement of being able to do 'dry-run' installation, we need to provide a `ScriptMaker` with the following items of information:

- Where source scripts are to be found.
- Where scripts are to be installed.
- Whether, on Windows, executable launchers should be added.
- Whether a dry-run mode is in effect.

These dictate the form that `ScriptMaker.__init__()` will take.

In addition, other methods suggest themselves for `ScriptMaker`:

- A `make()` method, which takes a *specification*, which is either a filename or a 'wrap me a callable' indicator which looks like this:

  ```
  name = some_package.some_module:some_callable [ flag(=value) ... ]
  ```

  The `name` would need to be a valid filename for a script, and the `some_package.some_module` part would indicate the module where the callable resides. The `some_callable` part identifies the callable, and optionally you can have flags, which the `ScriptMaker` instance must know how to interpret. One flag would be `'gui'`, indicating that the launcher should be a Windows application rather than a console application, for GUI-based scripts which shouldn't show a console window.

  The above specification is used by `setuptools` for the 'console_scripts' feature. See *Flag formats* for more information about flags.

> **Note:** Both setuptools and **PEP 426** interpret flags as a single value, which represents an extra (a set of optional dependencies needed for optional features of a distribution).

  It seems sensible for this method to return a list of absolute paths of files that were installed (or would have been installed, but for the dry-run mode being in effect).

- A make_multiple() method, which takes an iterable of specifications and just runs calls make() on each item iterated over, aggregating the results to return a list of absolute paths of all files that were installed (or would have been installed, but for the dry-run mode being in effect).

  One advantage of having this method is that you can override it in a subclass for post-processing, e.g. to run a tool like 2to3, or an analysis tool, over all the installed files.

- The details of the callable specification can be encapsulated in a utility function, get_exports_entry(). This would take a specification and return None, if the specification didn't match the callable format, or an instance of ExportEntry if it did match.

In addition, the following attributes on a ScriptMaker could be further used to refine its behaviour:

- force to indicate when scripts should be copied from source to target even when timestamps show the target is up to date.
- set_mode to indicate whether, on Posix, the execute mode bits should be set on the target script.

Flag formats

Flags, if present, are enclosed by square brackets. Each flag can have the format of just an alphanumeric string, optionally followed by an '=' and a value (with no intervening spaces). Multiple flags can be separated by ',' and whitespace. The following would be valid flag sections:

```
[a,b,c]
[a, b, c]
[a=b, c=d, e, f=g, 9=8]
```

whereas the following would be invalid:

```
[]
[\]
[a,]
[a,,b]
[a=,b,c]
```

Note: Both setuptools and **PEP 426** restrict flag formats to a single value, without an =. This value represents an extra (a set of optional dependencies needed for optional features of a distribution).

3.5 The version API

This section describes the design of the distlib API relating to versions.

3.5.1 The problem we're trying to solve

Distribution releases are named by versions and versions have two principal uses:

- Identifying a particular release and determining whether or not it is earlier or later than some other release.

- When specifying other distributions that a distribution release depends on, specifying constraints governing the releases of those distributions that are depended upon.

In addition, qualitative information may be given by the version format about the quality of the release: e.g. alpha versions, beta versions, stable releases, hot-fixes following a stable release. The following excerpt from **PEP 386** defines the requirements for versions:

- It should be possible to express more than one versioning level (usually this is expressed as major and minor revision and, sometimes, also a micro revision).
- A significant number of projects need special meaning versions for "pre-releases" (such as "alpha", "beta", "rc"), and these have widely used aliases ("a" stands for "alpha", "b" for "beta" and "c" for "rc"). And these pre-release versions make it impossible to use a simple alphanumerical ordering of the version string components. (e.g. 3.1a1 < 3.1)
- Some projects also need "post-releases" of regular versions, mainly for maintenance purposes, which can't be clearly expressed otherwise.
- Development versions allow packagers of unreleased work to avoid version clashes with later stable releases.

There are a number of version schemes in use. The ones of most interest in the Python ecosystem are:

- Loose versioning in `distutils`. *Any* version number is allowed, with lexicographical ordering. No support exists for pre- and post-releases, and lexicographical ordering can be unintuitive (e.g. '1.10' < '1.2.1')
- Strict versioning in `distutils`, which supports slightly more structure. It allows for up to three dot-separated numeric components, and support for multiple alpha and beta releases. However, there is no support for release candidates, nor for post-release versions.
- Versioning in `setuptools/distribute`. This is described in **PEP 386** in this section – it's perhaps the most widely used Python version scheme, but since it tries to be very flexible and work with a wide range of conventions, it ends up allowing a very chaotic mess of version conventions in the Python community as a whole.
- The proposed versioning scheme described in **PEP 440**.
- Semantic versioning, which is rational, simple and well-regarded in the software community in general.

Although the new versioning scheme mentioned in PEP 386 was implemented in `distutils2` and that code has been copied over to `distlib`, there are many projects on PyPI which do not conform to it, but rather to the "legacy" versioning schemes in `distutils/setuptools/distribute`. These schemes are deserving of some support not because of their intrinsic qualities, but due to their ubiquity in projects registered on PyPI. Below are some results from testing actual projects on PyPI:

```
Packages processed: 24891
Packages with no versions: 217
Packages with versions: 24674
Number of packages clean for all schemes: 19010 (77%)
Number of packages clean for PEP 386: 21072 (85%)
Number of packages clean for PEP 386 + suggestion: 23685 (96%)
Number of packages clean for legacy: 24674 (100%, by you would expect)
Number of packages clean for semantic: 19278 (78%)
```

where "+ suggestion" refers to using the suggested version algorithm to derive a version from a version which would otherwise be incompatible with **PEP 386**.

3.5.2 A minimal solution

Since `distlib` is a low-level library which might be used by tools which work with existing projects, the internal implementation of versions has changed slightly from `distutils2` to allow better support for legacy version numbering. Since the re-implementation facilitated adding semantic version support at minimal cost, this has also been provided.

Versions

The basic scheme is as follows. The differences between versioning schemes is catered for by having a single function for each scheme which converts a string version to an appropriate tuple which acts as a key for sorting and comparison of versions. We have a base class, `Version`, which defines any common code. Then we can have subclasses `NormalizedVersion` (PEP-386), `LegacyVersion` (`distribute/setuptools`) and `SemanticVersion`.

To compare versions, we just check type compatibility and then compare the corresponding tuples.

Matchers

Matchers take a name followed by a set of constraints in parentheses. Each constraint is an operation together with a version string which needs to be converted to the corresponding version instance.

In summary, the following attributes can be identified for `Version` and `Matcher`:

Version:

- version string passed in to constructor (stripped)
- parser to convert string string to tuple
- compare functions to compare with other versions of same type

Matcher:

- version string passed in to constructor (stripped)
- name of distribution
- list of constraints
- parser to convert string to name and set of constraints, using the same function as for `Version` to convert the version strings in the constraints to version instances
- method to match version to constraints and return True/False

Given the above, it appears that all the functionality *could* be provided with a single class per versioning scheme, with the *only* difference between them being the function to convert from version string to tuple. Any instance would act as either version or predicate, would display itself differently according to which it is, and raise exceptions if the wrong type of operation is performed on it (matching only allowed for predicate instances; <=, <, >=, > comparisons only allowed for version instances; and == and != allowed for either.

However, the use of the same class to implement versions and predicates leads to ambiguity, because of the very loose project naming and versioning schemes allowed by PyPI. For example, “Hello 2.0” could be a valid project name, and “5” is a project name actually registered on PyPI. If distribution names can look like versions, it’s hard to discern the developer’s intent when creating an instance with the string “5”. So, we make separate classes for Version and Matcher.

For ease of testing, the module will define, for each of the supported schemes, a function to do the parsing (as no information is needed other than the string), and the parse method of the class will call that function:

```
def normalized_key(s):
    "parse using PEP-386 logic"

def legacy_key(s):
    "parse using distribute/setuptools logic"
```

```
def semantic_key(s):
    "parse using semantic versioning logic"

class Version:
    # defines all common code

    def parse(self, s):
        raise NotImplementedError('Please implement in a subclass')
```

and then:

```
class NormalizedVersion(Version):
    def parse(self, s): return normalized_key(s)

class LegacyVersion(Version):
    def parse(self, s): return legacy_key(s)

class SemanticVersion(Version):
    def parse(self, s): return semantic_key(s)
```

And a custom versioning scheme can be devised to work in the same way:

```
def custom_key(s):
   """
   convert s to tuple using custom logic, raise UnsupportedVersionError
   on problems
   """

class CustomVersion(Version):
    def parse(self, s): return custom_key(s)
```

The matcher classes are pretty minimal, too:

```
class Matcher(object):
    version_class = None

    def match(self, string_or_version):
        """
        If passed a string, convert to version using version_class,
        then do matching in a way independent of version scheme in use
        """
```

and then:

```
class NormalizedMatcher(Matcher):
    version_class = NormalizedVersion

class LegacyMatcher(Matcher):
    version_class = LegacyVersion

class SemanticMatcher(Matcher):
    version_class = SemanticVersion
```

Version schemes

Ideally one would want to work with the PEP 386 scheme, but there might be times when one needs to work with the legacy scheme (for example, when investigating dependency graphs of existing PyPI projects). Hence, the important aspects of each scheme are bundled into a simple `VersionScheme` class:

```
class VersionScheme(object):
    def __init__(self, key, matcher):
        self.key = key          # version string -> tuple converter
        self.matcher = matcher  # Matcher subclass for the scheme
```

Of course, the version class is also available through the matcher's `version_class` attribute.

`VersionScheme` makes it easier to work with alternative version schemes. For example, say we decide to experiment with an "adaptive" version scheme, which is based on the PEP 386 scheme, but when handed a non-conforming version, automatically tries to convert it to a normalized version using `suggest_normalized_version()`. Then, code which has to deal with version schemes just has to pick the appropriate scheme by name.

Creating the adaptive scheme is easy:

```
def adaptive_key(s):
    try:
        result = normalized_key(s, False)
    except UnsupportedVersionError:
        s = suggest_normalized_version(s)
        if s is None:
            raise
        result = normalized_key(s, False)
    return result


class AdaptiveVersion(NormalizedVersion):
    def parse(self, s): return adaptive_key(s)

class AdaptiveMatcher(Matcher):
    version_class = AdaptiveVersion
```

The appropriate scheme can be fetched by using the `get_scheme()` function, which is defined thus:

```
def get_scheme(scheme_name):
    "Get a VersionScheme for the given scheme_name."
```

Allowed names are `'normalized'`, `'legacy'`, `'semantic'`, `'adaptive'` and `'default'` (which points to the same as `'adaptive'`). If an unrecognised name is passed in, a `ValueError` is raised.

The reimplemented `distlib.version` module is shorter than the corresponding module in `distutils2`, but the entire test suite passes and there is support for working with three versioning schemes as opposed to just one. However, the concept of "final" versions, which is not in the PEP but which was in the `distutils2` implementation, has been removed because it appears of little value (there's no way to determine the "final" status of versions for many of the project releases registered on PyPI).

3.6 The `wheel` API

This section describes the design of the `wheel` API which failitates building and installing from *wheels*, the new binary distribution format for Python described in **PEP 427**.

3.6.1 The problem we're trying to solve

There are basically two operations which need to be performed on wheels:

- Building a wheel from a source distribution.
- Installing a distribution which has been packaged as a wheel.

3.6.2 A minimal solution

Since we're talking about wheels, it seems likely that a `Wheel` class would be part of the design. This allows for extensibility over a purely function-based API. The `Wheel` would be expected to have methods that support the required operations:

```
class Wheel(object):
    def __init__(self, spec):
        """
        Initialise an instance from a specification. This can either be a
        valid filename for a wheel (for when you want to work with an
        existing wheel), or just the ``name-version-buildver`` portion of
        a wheel's filename (for when you're going to build a wheel for a
        known version and build of a named project).
        """

    def build(self, paths, tags=None):
        """
        Build a wheel. The ``name`, ``version`` and ``buildver`` should
        already have been set correctly. The ``paths`` should be a
        dictionary with keys 'prefix', 'scripts', 'headers', 'data' and one
        of 'purelib' and 'platlib'. These must point to valid paths if
        they are to be included in the wheel. The optional ``tags``
        argument should, if specified, be a dictionary with optional keys
        'pyver', 'abi' and 'arch' indicating lists of tags which
        indicate environments with which the wheel is compatible.
        """

    def install(self, paths, maker, **kwargs):
        """
        Install from a wheel. The ``paths`` should be a dictionary with
        keys 'prefix', 'scripts', 'headers', 'data', 'purelib' and
        'platlib'. These must point to valid paths to which files may
        be written if they are in the wheel. Only one of the 'purelib'
        and 'platlib' paths will be used (in the case where they are
        different), depending on whether the wheel is for a pure-
        Python distribution.

        The ``maker`` argument should be a suitably configured
        :class:`ScriptMaker` instance. The ``source_dir`` and
        ``target_dir`` arguments can be set to ``None`` when creating the
        instance - these will be set to appropriate values inside this
        method.

        The following keyword arguments are recognised:

        * ``warner``, if specified, should be a callable
          that will be called with (software_wheel_ver, file_wheel_ver)
          if they differ. They will both be in the form of tuples
          (major_ver, minor_ver). The ``warner`` defaults to ``None``.

        * It's conceivable that one might want to install only the library
          portion of a package -- not installing scripts, headers data and
          so on. If ``lib_only`` is specified as ``True``, only the
          ``site-packages`` contents will be installed. The default value
          is ``False`` (meaning everything will be installed).
        """
```

In addition to the above, the following attributes can be identified for a `Wheel` instance:

- `name` – the name of the distribution
- `version` – the version of the distribution
- `buildver` – the build tag for the distribution
- `pyver` – a list of Python versions with which the wheel is compatible
- `abi` – a list of application binary interfaces (ABIs) with which the wheel is compatible
- `arch` – a list of architectures with which the wheel is compatible
- `dirname` – The directory in which a wheel file is found/to be created
- `filename` – The filename of the wheel (computed from the other attributes)

3.7 Next steps

You might find it helpful to look at the *API Reference*.

CHAPTER 4

API Reference

This is the place where the functions and classes in `distlib's` public API are described.

4.1 The `distlib.database` package

4.1.1 Classes

class **`DistributionPath`**

This class represents a set of distributions which are installed on a Python path (like `PYTHONPATH` / `sys.path`). Both new-style (`distlib`) and legacy (egg) distributions are catered for.

Methods:

`__init__`(*path=None*, *include_egg=False*)

Initialise the instance using a particular path.

Parameters

- **path** (*list of str*) – The path to use when looking for distributions. If `None` is specified, `sys.path` is used.
- **include_egg** – If `True`, legacy distributions (eggs) are included in the search; otherwise, they aren't.

`enable_cache`()

Enables a cache, so that metadata information doesn't have to be fetched from disk. The cache is per instance of the `DistributionPath` instance and is enabled by default. It can be disabled using `disable_cache()` and cleared using `clear_cache()` (disabling won't automatically clear it).

`disable_cache`()

Disables the cache, but doesn't clear it.

`clear_cache`()

Clears the cache, but doesn't change its enabled/disabled status. If enabled, the cache will be re-populated when querying for distributions.

`get_distributions`()

The main querying method if you want to look at all the distributions. It returns an iterator which returns `Distribution` and, if `include_egg` was specified as `True` for the instance, also instances of any `EggInfoDistribution` for any legacy distributions found.

`get_distribution`(*name*)
Looks for a distribution by name. It returns the first one found with that name (there should only be one distribution with a given name on a given search path). Returns `None` if no distribution was found, or else an instance of `Distribution` (or, if `include_egg` was specified as `True` for the instance, an instance of `EggInfoDistribution` if a legacy distribution was found with that name).

Parameters **name** (*str*) – The name of the distribution to search for.

`get_exported_entries`(*category*, *name=None*)
Returns an iterator for entries exported by distributions on the path.

Parameters

- **category** (*str*) – The export category to look in.
- **name** (*str*) – A specific name to search for. If not specified, all entries in the category are returned.

Returns An iterator which iterates over exported entries (instances of `ExportEntry`).

class `Distribution`
A class representing a distribution, typically one which hasn't been installed (most likely, one which has been obtained from an index like PyPI).

Properties:

`name`
The name of the distribution.

`version`
The version of the distribution.

`metadata`
The metadata for the distribution. This is a `distlib.metadata.Metadata` instance.

`source_url`
The download URL for the source distribution.

`digest`
The digest for the source distribution. This is either `None` or a 2-tuple consisting of the hashing algorithm and the digest using that algorithm, e.g. `('sha256', '01234...')`.

`locator`
The locator for an instance which has been retrieved through a locator. This is `None` for an installed distribution.

class `InstalledDistribution`(*Distribution*)
A class representing an installed distribution. This class is not instantiated directly, except by packaging tools. Instances of it are returned from querying a `DistributionPath`.

Properties:

`requested`
Whether the distribution was installed by user request (if not, it may have been installed as a dependency of some other distribution).

`exports`
The distribution's exports, as described in *Exporting things from Distributions*. This is a cached property.

Methods:

`list_installed_files`(*local=False*)
Returns an iterator over all of the individual files installed as part of the distribution, including metadata

files. The iterator returns tuples of the form (path, hash, size). The list of files is written by the installer to the `RECORD` metadata file.

Parameters **local** – If `True`, the paths returned are local absolute paths (i.e. with platform-specific directory separators as indicated by `os.sep`); otherwise, they are the values stored in the `RECORD` metadata file.

`list_distinfo_files`(*local=False*)

Similar to `list_installed_files()`, but only returns metadata files.

Parameters **local** – As for `list_installed_files()`.

`check_installed_files`()

Runs over all the installed files to check that the size and checksum are unchanged from the values in the `RECORD` file, written when the distribution was installed. It returns a list of mismatches. If the files in the distribution haven't been corrupted , an empty list will be returned; otherwise, a list of mismatches will be returned.

Returns

A list which, if non-empty, will contain tuples with the following elements:

- The path in `RECORD` which failed to match.
- One of the strings 'exists', 'size' or 'hash' according to what didn't match (existence is checked first, then size, then hash).
- The expected value of what didn't match (as obtained from `RECORD`).
- The actual value of what didn't match (as obtained from the file system).

`read_exports`(*filename=None*)

Read exports information from a file.

Normal access to a distribution's exports should be through its `exports` attribute. This method is called from there as needed. If no filename is specified, the `EXPORTS` file in the `.dist-info` directory is read (it is expected to be present).

Parameters **filename** (*str*) – The filename to read from, or `None` to read from the default location.

Returns The exports read from the file.

Return type dict

`write_exports`(*exports, filename=None*)

Write exports information to a file.

If no filename is specified, the `EXPORTS` file in the `.dist-info` directory is written.

Parameters

- **exports** (*dict*) – A dictionary whose keys are categories and whose values are dictionaries which contain `ExportEntry` instances keyed on their name.
- **filename** (*str*) – The filename to read from, or `None` to read from the default location.

class **`EggInfoDistribution`**

Analogous to `Distribution`, but covering legacy distributions. This class is not instantiated directly. Instances of it are returned from querying a `DistributionPath`.

Properties:

`name`

The name of the distribution.

`version`
The version of the distribution.

`metadata`
The metadata for the distribution. This is a `distlib.metadata.Metadata` instance.

Methods:

`list_installed_files`(*local=False*)
Returns a list all of the individual files installed as part of the distribution.

Parameters **local** – If `True`, the paths returned are local absolute paths (i.e. with platform-specific directory separators as indicated by `os.sep`).

class **`DependencyGraph`**
This class represents a dependency graph between releases. The nodes are distribution instances; the edges model dependencies. An edge from `a` to `b` means that `a` depends on `b`.

`add_distribution`(*distribution*)
Add *distribution* to the graph.

`add_edge`(*x*, *y*, *label=None*)
Add an edge from distribution *x* to distribution *y* with the given *label* (string).

`add_missing`(*distribution*, *requirement*)
Add a missing *requirement* (string) for the given *distribution*.

`repr_node`(*dist*, *level=1*)
Print a subgraph starting from *dist*. *level* gives the depth of the subgraph.

Direct access to the graph nodes and edges is provided through these attributes:

`adjacency_list`
Dictionary mapping distributions to a list of `(other, label)` tuples where `other` is a distribution and the edge is labelled with `label` (i.e. the version specifier, if such was provided).

`reverse_list`
Dictionary mapping distributions to a list of predecessors. This allows efficient traversal.

`missing`
Dictionary mapping distributions to a list of requirements that were not provided by any distribution.

4.2 The `distlib.resources` package

4.2.1 Attributes

`cache`
An instance of `Cache`, which uses the default base location for the cache (as described in the documentation for `Cache.__init__()`).

4.2.2 Functions

`finder`(*package*)
Get a finder for the specified package.

If the package hasn't been imported yet, an attempt will be made to import it. If importing fails, an `ImportError` will be raised.

Parameters **package** (*str*) – The name of the package for which a finder is desired.

Returns A finder for the package.

`register_finder`(*loader*, *finder_maker*)
Register a callable which makes finders for a particular type of **PEP 302** loader.

Parameters

- **loader** – The loader for which a finder is to be returned.
- **finder_maker** – A callable to be registered, which is called when a loader of the specified type is used to load a package. The callable is called with a single argument – the Python module object corresponding to the package – and must return a finder for that package.

4.2.3 Classes

class **`Resource`**
A class representing resources. It is never instantiated directly, but always through calling a finder's `find` method.

Properties:

`is_container`
Whether this instance is a container of other resources.

`bytes`
All of the resource data as a byte string. Raises an exception if accessed on a container resource.

`size`
The size of the resource data in bytes. Raises an exception if accessed on a container resource.

`resources`
The relative names of all the contents of this resource. Raises an exception if accessed on a resource which is *not* a container.

`path`
This attribute is set by the resource's finder. It is a textual representation of the path, such that if a PEP 302 loader's `get_data()` method is called with the path, the resource's bytes are returned by the loader. This attribute is analogous to the `resource_filename` API in `setuptools`. Note that for resources in zip files, the path will be a pointer to the resource in the zip file, and not directly usable as a filename. While `setuptools` deals with this by extracting zip entries to cache and returning filenames from the cache, this does not seem an appropriate thing to do in this package, as a resource is already made available to callers either as a stream or a string of bytes.

`file_path`
This attribute is the same as the path for file-based resource. For resources in a .zip file, the relevant resource is extracted to a file in a cache in the file system, and the name of the cached file is returned. This is for use with APIs that need file names or to be able to access data through OS-level file handles.

Methods:

`as_stream`()
A binary stream of the resource's data. This must be closed by the caller when it's finished with.

Raises an exception if called on a container resource.

class **`ResourceFinder`**
A base class for resource finders, which finds resources for packages stored in the file system.

`__init__`(*module*)
Initialise the finder for the package specified by `module`.

Parameters **module** – The Python module object representing a package.

find(*resource_name*)

Find a resource with the name specified by `resource_name` and return a `Resource` instance which represents it.

Parameters **resource_name** – A fully qualified resource name, with hierarchical components separated by '/'.

Returns A `Resource` instance, or `None` if a resource with that name wasn't found.

is_container(*resource*)

Return whether a resource is a container of other resources.

Parameters **resource** (a `Resource` instance) – The resource whose status as container is wanted.

Returns `True` or `False`.

get_stream(*resource*)

Return a binary stream for the specified resource.

Parameters **resource** (a `Resource` instance) – The resource for which a stream is wanted.

Returns A binary stream for the resource.

get_bytes(*resource*)

Return the contents of the specified resource as a byte string.

Parameters **resource** (a `Resource` instance) – The resource for which the bytes are wanted.

Returns The data in the resource as a byte string.

get_size(*resource*)

Return the size of the specified resource in bytes.

Parameters **resource** (a `Resource` instance) – The resource for which the size is wanted.

Returns The size of the resource in bytes.

class ZipResourceFinder

This has the same interface as `ResourceFinder`.

class Cache

This class implements a cache for resources which must be accessible as files in the file system.

__init__(*base=None*)

Initialise a cache instance with a specific directory which holds the cache. If `base` is specified but does not exist, it is created. If `base` is not specified, it defaults to `os.expanduser('~/.distlib/resource-cache') on POSIX platforms (``os.name == 'posix')`. On Windows, if the environment contains `LOCALAPPDATA`, the cache will be placed in

```
os.path.expandvars(r'$localappdata\.distlib\resource-cache')
```

Otherwise, the location will be

```
os.path.expanduser(r'~\.distlib\resource-cache')
```

get(*resource*)

Ensures that the resource is available as a file in the file system, and returns the name of that file. This method calls the resource's finder's `get_cache_info()` method.

is_stale(*resource*, *path*)

Returns whether the data in the resource which is cached in the file system is stale compared to the resource's current data. The default implementation returns `True`, causing the resource's data to be re-written to the file every time.

prefix_to_dir(*prefix*)
Converts a prefix for a resource (e.g. the name of its containing .zip) into a directory name in the cache. This implementation delegates the work to `path_to_cache_dir()`.

4.3 The `distlib.scripts` package

4.3.1 Classes

class **ScriptMaker**
A class used to install scripts based on specifications.

source_dir
The directory where script sources are to be found.

target_dir
The directory where scripts are to be created.

add_launchers
Whether to create native executable launchers on Windows.

force
Whether to overwrite scripts even when timestamps show they're up to date.

set_mode
Whether, on Posix, the scripts should have their execute mode set.

script_template
The text of a template which should contain `%(shebang)s`, `%(module)s` and `%(func)s` in the appropriate places.

The attribute is defined at class level. You can override it at the instance level to customise your scripts.

__init__(*source_directory*, *target_directory*, *add_launchers=True*, *dry_run=False*)
Initialise the instance with options that control its behaviour.

Parameters

- **source_directory** (*str*) – Where to find scripts to install.
- **target_directory** (*str*) – Where to install scripts to.
- **add_launchers** (*bool*) – If true, create executable launchers on Windows. The executables are currently generated from the following project:

  https://bitbucket.org/vinay.sajip/simple_launcher/
- **dry_run** – If true, don't actually install scripts - just pretend to.

make(*specification*, *options=None*)
Make a script in the target directory.

Parameters

- **specification** (*str*) – A specification, which can take one of the following forms:
  - A filename, relative to `source_directory`, such as `foo.py` or `subdir/bar.py`.
  - A reference to a callable, given in the form:

    ```
    name = some_package.some_module:some_callable [flags]
    ```

where the *flags* part is optional.

When this form is passed, a Python stub script is created with the appropriate shebang line and with code to load and call the specified callable with no arguments, returning its value as the return code from the script.

For more information about flags, see *Flag formats*.

- **options** – If specified, a dictionary of options used to control script creation. Currently, only the key `gui` is checked: this should be a `bool` which, if `True`, indicates that the script is a windowed application. This distinction is only drawn on Windows if `add_launchers` is `True`, and results in a windowed native launcher application if `options['gui']` is `True` (otherwise, the native executable launcher is a console application).

Returns A list of absolute pathnames of files installed (or which would have been installed, but for `dry_run` being true).

make_multiple(*specifications*, *options*)
Make multiple scripts from an iterable.

This method just calls `make()` once for each value returned by the iterable, but it might be convenient to override this method in some scenarios to do post-processing of the installed files (for example, running `2to3` on them).

Parameters

- **specifications** – an iterable giving the specifications to follow.
- **options** – As for the `make()` method.

Returns A list of absolute pathnames of files installed (or which would have been installed, but for `dry_run` being true).

4.4 The `distlib.locators` package

4.4.1 Classes

class **Locator**
The base class for locators. Implements logic common to multiple locators.

__init__(*scheme='default'*)
Initialise an instance of the locator. :param scheme: The version scheme to use. :type scheme: str

get_project(*name*)
This method should be implemented in subclasses. It returns a (potentially empty) dictionary whose keys are the versions located for the project named by `name`, and whose values are instances of `distlib.util.Distribution`.

convert_url_to_download_info(*url*, *project_name*)
Extract information from a URL about the name and version of a distribution.

Parameters

- **url** (*str*) – The URL potentially of an archive (though it needn't be).
- **project_name** (*str*) – This must match the project name determined from the archive (case-insensitive matching is used).

Returns

`None` if the URL does not appear to be that of a distribution archive for the named project. Otherwise, a dictionary is returned with the following keys at a minimum:

- url – the URL passed in, minus any fragment portion.
- filename – a suitable filename to use for the archive locally.

Optional keys returned are:

- md5_digest – the MD5 hash of the archive, for verification after downloading. This is extracted from the fragment portion, if any, of the passed-in URL.
- sha256_digest – the SHA256 hash of the archive, for verification after downloading. This is extracted from the fragment portion, if any, of the passed-in URL.

Return type dict

`get_distribution_names``()`

Get the names of all distributions known to this locator.

The base class raises `NotImplementedError`; this method should be implemented in a subclass.

Returns All distributions known to this locator.

Return type set

`locate`(*requirement*, *prereleases=False*)

This tries to locate the latest version of a potentially downloadable distribution which matches a requirement (name and version constraints). If a potentially downloadable distribution (i.e. one with a download URL) is not found, `None` is returned – otherwise, an instance of `Distribution` is returned. The returned instance will have, at a minimum, `name`, `version` and `source_url` populated.

Parameters

- **requirement** (*str*) – The name and optional version constraints of the distribution to locate, e.g. `'Flask'` or `'Flask (>= 0.7, < 0.9)'`.
- **prereleases** (*bool*) – If `True`, prereleases are treated like normal releases. The default behaviour is to not return any prereleases unless they are the only ones which match the requirement.

Returns A matching instance of `Distribution`, or `None`.

class `DirectoryLocator`(*Locator*)

This locator scans the file system under a base directory, looking for distribution archives. The locator scans all subdirectories recursively, unless the `recursive` flag is set to `False`.

`__init__`(*base_dir*, ***kwargs*)

Parameters

- **base_dir** (*str*) – The base directory to scan for distribution archives.
- **kwargs** – Passed to base class constructor, apart from the following keyword arguments:
  - `recursive` (defaults to `True`) – if `False`, no recursion into subdirectories occurs.

class `PyPIRPCLocator`(*Locator*)

This locator uses the PyPI XML-RPC interface to locate distribution archives and other data about downloads.

`__init__`(*url*, ***kwargs*)

param url The base URL to use for the XML-RPC service.

type url str

param kwargs Passed to base class constructor.

`get_project`(*name*)
See `Locator.get_project()`.

class **`PyPIJSONLocator`**(*Locator*)
This locator uses the PyPI JSON interface to locate distribution archives and other data about downloads. It gets the metadata and URL information in a single call, so it should perform better than the XML-RPC locator.

`__init__`(*url*, ***kwargs*)

param url The base URL to use for the JSON service.

type url str

param kwargs Passed to base class constructor.

`get_project`(*name*)
See `Locator.get_project()`.

class **`SimpleScrapingLocator`**
This locator uses the PyPI 'simple' interface – a Web scraping interface – to locate distribution archives.

`__init__`(*url*, *timeout=None*, *num_workers=10*, ***kwargs*)

Parameters

- **url** (*str*) – The base URL to use for the simple service HTML pages.
- **timeout** (*float*) – How long (in seconds) to wait before giving up on a remote resource.
- **num_workers** (*int*) – The number of worker threads created to perform scraping activities.
- **kwargs** – Passed to base class constructor.

class **`DistPathLocator`**
This locator uses a `DistributionPath` instance to locate installed distributions.

`__init__`(*url*, *distpath*, ***kwargs*)

Parameters

- **distpath** (`DistributionPath`) – The distribution path to use.
- **kwargs** – Passed to base class constructor.

class **`AggregatingLocator`**(*Locator*)
This locator uses a list of other aggregators and delegates finding projects to them. It can either return the first result found (i.e. from the first aggregator in the list provided which returns a non-empty result), or a merged result from all the aggregators in the list.

`__init__`(**locators*, ***kwargs*)

Parameters

- **locators** (*sequence of locators*) – A list of aggregators to delegate finding projects to.
- **merge** (*bool*) – If this *kwarg* is `True`, each aggregator in the list is asked to provide results, which are aggregated into a results dictionary. If `False`, the first non-empty return value from the list of aggregators is returned. The locators are consulted in the order in which they're passed in.

class DependencyFinder

This class allows you to recursively find all the distributions which a particular distribution depends on.

`__init__`(*locator*)

Initialise an instance with the locator to be used for locating distributions.

`find`(*requirement*, *metas_extras=None*, *prereleases=False*)

Find all the distributions needed to fulfill `requirement`.

Parameters

- **requirement** – A string of the from `name (version)` where version can include an inequality constraint, or an instance of `Distribution` (e.g. representing a distribution on the local hard disk).
- **meta_extras** – A list of meta extras such as :test:, :build: and so on, to be included in the dependencies.
- **prereleases** – If `True`, allow pre-release versions to be returned - otherwise, don't return prereleases unless they're all that's available.

Returns

A 2-tuple. The first element is a set of `Distribution` instances. The second element is a set of problems encountered during dependency resolution. Currently, if this set is non-empty, it will contain 2-tuples whose first element is the string 'unsatisfied' and whose second element is a requirement which couldn't be satisfied.

In the set of `Distribution` instances returned, some attributes will be set:

- The instance representing the passed-in `requirement` will have the `requested` attribute set to `True`.
- All requirements which are not installation requirements (in other words, are needed only for build and test) will have the `build_time_dependency` attribute set to `True`.

4.4.2 Functions

`get_all_distribution_names`(*url=None*)

Retrieve the names of all distributions registered on an index.

Parameters **url** (*str*) – The XML-RPC service URL of the node to query. If not specified, The main PyPI index is queried.

Returns A list of the names of distributions registered on the index. Note that some of the names may be Unicode.

Return type list

`locate`(*requirement*, *prereleases=False*)

This convenience function returns the latest version of a potentially downloadable distribution which matches a requirement (name and version constraints). If a potentially downloadable distribution (i.e. one with a download URL) is not found, `None` is returned – otherwise, an instance of `Distribution` is returned. The returned instance will have, at a minimum, `name`, `version` and `download_url`.

Parameters

- **requirement** (*str*) – The name and optional version constraints of the distribution to locate, e.g. `'Flask'` or `'Flask (>= 0.7, < 0.9)'`.

- **prereleases** (*bool*) – If `True`, prereleases are treated like normal releases. The default behaviour is to not return any prereleases unless they are the only ones which match the requirement.

Returns A matching instance of `Distribution`, or `None`.

4.4.3 Variables

`default_locator`
This attribute holds a locator which is used by `locate()` to locate distributions.

4.5 The `distlib.index` package

4.5.1 Classes

class **`PackageIndex`**
This class represents a package index which is compatible with PyPI, the Python Package Index. It allows you to register projects, upload source and binary distributions (with support for digital signatures), upload documentation, verify signatures and get a list of hosts which are mirrors for the index.

Methods:

`__init__`(*url=None*, *mirror_host=None*)

Initialise an instance, setting instance attributes named from the keyword arguments.

Parameters

- **url** – The root URL for the index. If not specified, the URL for PyPI is used (‘http://pypi.python.org/pypi‘).
- **mirror_host** – The DNS name for a host which can be used to determine available mirror hosts for the index. If not specified, the value ‘last.pypi.python.org’ is used.

`register`(*metadata*)
Register a project with the index.

Parameters metadata – A `Metadata` instance. This should have at least the `Name` and `Version` fields set, and ideally as much metadata as possible about this distribution. Though it might seem odd to have to specify a version when you are initially registering a project, this is required by PyPI. You can see this in PyPI’s Web UI when you click the “Package submission” link in the left-hand side menu.

Returns An `urllib` HTTP response returned by the index. If an error occurs, an `HTTPError` exception will be raised.

`upload_file`(*metadata*, *filename*, *signer=None*, *sign_password=None*, *filetype=’sdist’*, *pyversion=’source’*)
Upload a distribution to the index.

Parameters

- **metadata** – A `Metadata` instance. This should have at least the `Name` and `Version` fields set, and ideally as much metadata as possible about this distribution.
- **file_name** – The path to the file which is to be uploaded.

- **signer** – If specified, this needs to be a string identifying the GnuPG private key which is to be used for signing the distribution.
- **sign_password** – The passphrase which allows access to the private key used for the signature.
- **filetype** – The type of the file being uploaded. This would have values such as `sdist` (for a source distribution), `bdist_wininst` for a Windows installer, and so on. Consult the `distutils` documentation for the full set of possible values.
- **pyversion** – The Python version this distribution is compatible with. If it's a pure-Python distribution, the value to use would be `source` - for distributions which are for specific Python versions, you would use the Python version in the form `X.Y`.

Returns An `urllib` HTTP response returned by the index. If an error occurs, an `HTTPError` exception will be raised.

`upload_documentation`(*metadata*, *doc_dir*)
Upload HTML documentation to the index. The contents of the specified directory tree will be packed into a .zip file which is then uploaded to the index.

Parameters

- **metadata** – A `Metadata` instance. This should have at least the `Name` and `Version` fields set.
- **doc_dir** – The path to the root directory for the HTML documentation. This directory should be the one that contains `index.html`.

Returns An `urllib` HTTP response returned by the index. If an error occurs, an `HTTPError` exception will be raised.

`verify_signature`(*self*, *signature_filename*, *data_filename*)
Verify a digital signature against a downloaded distribution.

Parameters

- **signature_filename** – The path to the file which contains the digital signature.
- **data_filename** – The path to the file which was supposedly signed to obtain the signature in `signature_filename`.

Returns `True` if the signature can be verified, else `False`. If an error occurs (e.g. unable to locate the public key used to verify the signature), a `ValueError` is raised.

Additional attributes:

`username`
The username to use when authenticating with the index.

`password`
The password to use when authenticating with the index.

`gpg`
The path to the signing and verification program.

`gpg_home`
The location of the key database for the signing and verification program.

`mirrors`
The list of hosts which are mirrors for this index.

boundary
The boundary value to use when MIME-encoding requests to be sent to the index. This should be a byte-string.

4.6 The `distlib.util` package

4.6.1 Classes

class **ExportEntry**
Attributes:

A class holding information about a exports entry.

name
The name of the entry.

prefix
The prefix part of the entry. For a callable or data item in a module, this is the name of the package or module containing the item.

suffix
The suffix part of the entry. For a callable or data item in a module, this is a dotted path which points to the item in the module.

flags
A list of flags. See *Flag formats* for more information.

value
The actual value of the entry (a callable or data item in a module, or perhaps just a module). This is a cached property of the instance, and is determined by calling `resolve()` with the `prefix` and `suffix` properties.

dist
The distribution which exports this entry. This is normally an instance of `InstalledDistribution`.

4.6.2 Functions

get_cache_base()
Return the base directory which will hold distlib caches. If the directory does not exist, it is created.

On Windows, if `LOCALAPPDATA` is defined in the environment, then it is assumed to be a directory, and will be the parent directory of the result. On POSIX, and on Windows if `LOCALAPPDATA` is not defined, the user's home directory – as determined using `os.expanduser('~')` – will be the parent directory of the result.

The result is just the directory `'.distlib'` in the parent directory as determined above.

path_to_cache_dir(*path*)
Converts a path (e.g. the name of an archive) into a directory name suitable for use in a cache. The following algorithm is used:

1. On Windows, any `':'` in the drive is replaced with `'---'`.
2. Any occurrence of `os.sep` is replaced with `'--'`.
3. `'.cache'` is appended.

get_export_entry(*specification*)
Return a export entry from a specification, if it matches the expected format, or else `None`.

Parameters specification (*str*) – A specification, as documented for the `distlib.scripts.ScriptMaker.make()` method.

Returns `None` if the specification didn't match the expected form for an entry, or else an instance of `ExportEntry` holding information about the entry.

`resolve`(*module_name*, *dotted_path*)

Given a `module name` and a `dotted_path` representing an object in that module, resolve the passed parameters to an object and return that object.

If the module has not already been imported, this function attempts to import it, then access the object represented by `dotted_path` in the module's namespace. If `dotted_path` is `None`, the module is returned. If import or attribute access fails, an `ImportError` or `AttributeError` will be raised.

Parameters

- **module_name** (*str*) – The name of a Python module or package, e.g. `os` or `os.path`.
- **dotted_path** (*str*) – The path of an object expected to be in the module's namespace, e.g. `'environ'`, `'sep'` or `'path.supports_unicode_filenames'`.

4.7 The `distlib.wheel` package

This package has functionality which allows you to work with wheels (see **PEP 427**).

4.7.1 Classes

class `Wheel`

This class represents wheels – either existing wheels, or wheels to be built.

`__init__`(*spec*)

Initialise an instance from a specification.

Parameters spec (*str*) – This can either be a valid filename for a wheel (for when you want to work with an existing wheel), or just the `name-version-buildver` portion of a wheel's filename (for when you're going to build a wheel for a known version and build of a named project).

`build`(*paths*, *tags=None*, *wheel_version=None*)

Build a wheel. The `name`, `version` and `buildver` should already have been set correctly.

Parameters

- **paths** – This should be a dictionary with keys `'prefix'`, `'scripts'`, `'headers'`, `'data'` and one of `'purelib'` or `'platlib'`. These must point to valid paths if they are to be included in the wheel.
- **tags** – If specified, this should be a dictionary with optional keys `'pyver'`, `'abi'` and `'arch'` indicating lists of tags which indicate environments with which the wheel is compatible.
- **wheel_version** – If specified, this is written to the wheel's "Wheel-Version" metadata. If not specified, the implementation's latest supported wheel version is used.

`install`(*self*, *paths*, *maker*, ***kwargs*)

Install from a wheel.

Parameters

- **paths** – This should be a dictionary with keys `'prefix'`, `'scripts'`, `'headers'`, `'data'`, `'purelib'` and `'platlib'`. These must point to valid paths to which files may be written if they are in the wheel. Only one of the `'purelib'` and `'platlib'` paths will be used (in the case where they are different), depending on whether the wheel is for a pure-Python distribution.
- **maker** – This should be set to a suitably configured instance of `ScriptMaker`. The `source_dir` and `target_dir` arguments can be set to `None` when creating the instance - these will be set to appropriate values inside this method.
- **warner** – If specified, should be a callable that will be called with (software_wheel_ver, file_wheel_ver) if they differ. They will both be in the form of tuples (major_ver, minor_ver).
- **lib_only** – It's conceivable that one might want to install only the library portion of a package – not installing scripts, headers, data and so on. If `lib_only` is specified as `True`, only the `site-packages` contents will be installed.

`mount`(*append=False*)

Mount the wheel so that its contents can be imported directly, without the need to install the wheel. If the wheel contains C extensions and has metadata about these extensions, the extensions are also available for import.

If the wheels tags indicate it is not compatible with the running Python, a `DistlibException` is raised.

param append If `True`, the wheel's pathname is added to the end of `sys.path`. By default, it is added to the beginning.

`unmount`()

Unmount the wheel so that its contents can no longer be imported directly. If the wheel contains C extensions and has metadata about these extensions, the extensions are also made unavailable for import.

`name`

The name of the distribution.

`version`

The version of the distribution

`buildver`

The build tag for the distribution.

`pyver`

A list of Python versions with which the wheel is compatible. See **PEP 427** and **PEP 425** for details.

`abi`

A list of application binary interfaces (ABIs) with which the wheel is compatible. See **PEP 427** and **PEP 425** for details.

`arch`

A list of architectures with which the wheel is compatible. See **PEP 427** and **PEP 425** for details.

`dirname`

The directory in which a wheel file is found/to be created.

`filename`

The filename of the wheel (computed from the other attributes)

`metadata`

The metadata for the distribution in the wheel, as a `Metadata` instance.

info
The wheel metadata (contents of the `WHEEL` metadata file) as a dictionary.

4.7.2 Functions

is_compatible(*wheel*, *tags=None*)
Indicate if a wheel is compatible with a set of tags. If any combination of the tags of `wheel` is found in `tags`, then the wheel is considered to be compatible.

Parameters

- **wheel** – A `Wheel` instance or the filename of a wheel.
- **tags** – A set of tags to check for compatibility. If not specified, it defaults to the set of tags which are compatible with this Python implementation.

Returns `True` if compatible, else `False`.

4.7.3 Attributes

COMPATIBLE_TAGS
A list of (`pyver`, `abi`, `arch`) tags which are compatible with this Python implementation.

4.8 Next steps

You might find it helpful to look at the mailing list.

CHAPTER 5

Migrating from older APIs

This section has information on migrating from older APIs.

5.1 The `pkg_resources` resource API

5.1.1 Basic resource access

`resource_exists(package, resource_name)` `finder(package).find(resource_name) is not None`

`resource_stream(package, resource_name)` `finder(package).find(resource_name).as_stream()`

`resource_string(package, resource_name)` `finder(package).find(resource_name).bytes`

`resource_isdir(package, resource_name)` `finder(package).find(resource_name).is_container`

`resource_listdir(package, resource_name)` `finder(package).find(resource_name).resources`

5.1.2 Resource extraction

`resource_filename(package, resource_name)` `finder(package).find(resource_name).file_path`

`set_extraction_path(extraction_path)` This has no direct analogue, but you can achieve equivalent results by doing something like the following:

```
from distlib import resources

resources.cache = resources.Cache(extraction_path)
```

before accessing the `file_path` property of any `Resource`. Note that if you have accessed the `file_path` property for a resource *before* doing this, the cache may already have extracted files.

`cleanup_resources(force=False)` This is not actually implemented in `pkg_resources` – it's a no-op. You could achieve the analogous result using:

```
from distlib import resources

not_removed = resources.cache.clear()
```

5.1.3 Provider interface

You can provide an `XXXResourceFinder` class which finds resources in custom storage containers, and works like `ResourceFinder`. Although it shouldn't be necessary, you could also return a subclass of `Resource` from your finders, to deal with custom requirements which aren't catered for.

`get_cache_path(archive_name, names=())` There's no analogue for this, as you shouldn't need to care about whether particular resources are implemented in archives or not. If you need this API, please give feedback with more information about your use cases.

`extraction_error()` There's no analogue for this. The `Cache.get()` method, which writes a resource's bytes to a file in the cache, will raise any exception caused by underlying I/O. If you need to handle this in the cache layer, you can subclass `Cache` and override `get()`. If that doesn't work for you, please give feedback with more information about your use cases.

`postprocess(tempname, filename)` There's no analogue for this. The `Cache.get()` method, which writes a resource's bytes to a file in the cache, can be overridden to perform any custom post-processing. If that doesn't work for you, please give feedback with more information about your use cases.

5.2 The `pkg_resources` entry point API

Entry points in `pkg_resources` are equivalent to per-distribution exports dictionary (see *Exporting things from Distributions*). The keys to the dictionary are just names in a hierarchical namespace delineated with periods (like Python packages). These keys are called *groups* in `pkg_resources` documentation, though that term is a little ambiguous. In Eclipse, for example, they are called *extension point IDs*, which is a little closer to the intended usage, but a bit of a mouthful. In `distlib`, we'll use the term `category` or `export category`.

In `distlib`, the implementation of exports is slightly different from entry points of `pkg_resources`. A `Distribution` instance has an `exports` attribute, which is a dictionary keyed by category and whose values are dictionaries that map names to `ExportEntry` instances.

Below are the `pkg_resources` functions and how to achieve the equivalent in `distlib`. In cases where the `pkg_resources` functions take distribution names, in `distlib` you get the corresponding `Distribution` instance, using:

```
dist = dist_path.get_distribution(distname)
```

and then ask that instance (or the `dist_path` instance) for the things you need.

`load_entry_point(distname, groupname, name)` `dist.exports[groupname][name].value`

`get_entry_info(distname, groupname, name)` `dist.exports[groupname][name]`

`get_entry_map(distname, groupname=None)` `dist.exports` or `dist.exports[groupname]`

`iter_entry_points(groupname, name=None)` `dist_path.get_exported_entries(groupname, name=None)`