Setuptools Learning

Tuesday, April 01, 2014 14:45

1. **Official Documentation:** https://pythonhosted.org/setuptools/#

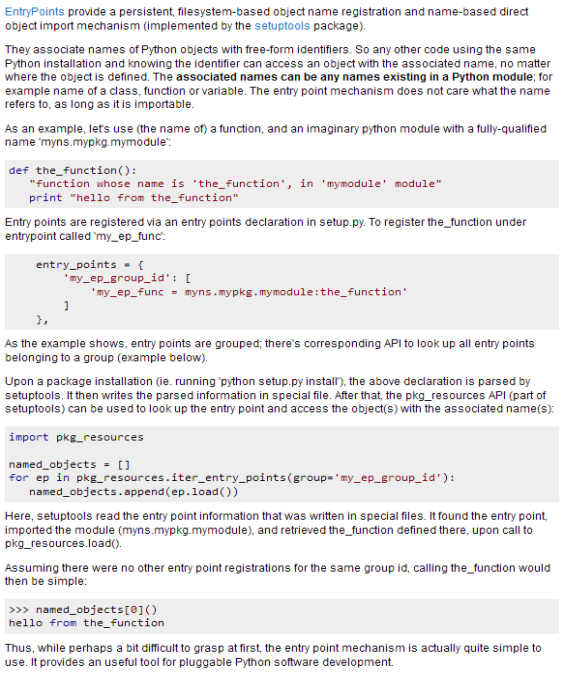
2. **Building and Distributing Packages with Setuptools**
   a. **Example**

```
from setuptools import setup, find_packages

setup(name='crypto',
    version='1.0',

    # packages = find_packages('src'),  # include all packages under src
    # package_dir = {'':'src'},   # tell distutils packages are under src
    packages=find_packages(),

    # Project uses plumbum, so ensure that the plumbum get
    # installed or upgraded on the target machine
    install_requires = ['plumbum>=1.4.0'],

    # The data files must be under CVS or Subversion control,
    # or else they must be specified via the distutils' MANIFEST.in file
    # include_package_data = True

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

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

    # One caveat is that you will have to also set zip_safe = False
    # in setup.py so that all the files are unzipped during installation.
    zip_safe=True,

    # metadata for upload to PyPI
    author = "Wenyu",
    author_email = "xxx@cisco.com",
    description = "This is Crypto Package",
    license = "xxx",
    keywords = "crypto package",
    url = "http://www.cisco.com",   # project home page, if any
)
```

   b. **Setup keywords**
      i. **Meta-Data**

Meta-Data	Description	Value	Notes
name	name of the package	short string	(1)
version	version of this release	short string	(1)(2)
author	package author's name	short string	(3)
author_email	email address of the package author	email address	(3)
maintainer	package maintainer's name	short string	(3)
maintainer_email	email address of the package maintainer	email address	(3)
url	home page for the package	URL	(1)
description	short, summary description of the package	short string	
long_description	longer description of the package	long string	(5)
download_url	location where the package may be downloaded	URL	(4)
classifiers	a list of classifiers	list of strings	(4)
platforms	a list of platforms	list of strings	
license	license for the package	short string	(6)

      ii. **List whole packages**
         - Package_dir
         - Packages
      iii. **Listing individual modules:** list all modules rather than listing packages
         - py_modules = ['mod1', 'pkg.mod2']
      iv. **Describing extension modules**
         - ext_modules=[Extension('foo', ['foo.c'])]
      v. **Installing Scripts:** if the first line of the script starts with #! and contains the word "python", the Distutils will adjust the first line to refer to the current interpreter location. By default, it is replaced with the current interpreter location.
         - scripts=['scripts/xmlproc_parse', 'scripts/xmlproc_val']
      vi. **New Features in Setuptools:**
         - **include_package_data**: If set to True, this tells setuptools to automatically include any data files it finds inside your package directories, that are either under CVS or Subversion control, or which are specified by your MANIFEST.in file. For more information, see the section below on Including Data Files.
         - **exclude_package_data**: A dictionary mapping package names to lists of glob patterns that should be excluded from your package directories. You can use this to trim back any excess files included by include_package_data. For a complete description and examples, see the section below on Including Data Files.
         - **package_data:** A dictionary mapping package names to lists of glob patterns. For a complete description and examples, see the section below on Including Data Files. You do not need to use this option if you are using include_package_data, unless you need to add e.g. files that are generated by your setup script and build process. (And are therefore not in source control or are files that you don't want to include in your source distribution.)
         - **zip_safe**: A boolean (True or False) flag specifying whether the project can be safely installed and run from a zip file. If this argument is not supplied, the bdist_egg command will have to analyze all of your project's contents for possible problems each time it builds an egg.
         - **install_requires**: A string or list of strings specifying what other distributions need to be installed when this one is. See the section below on Declaring Dependencies for details and examples of the format of this argument.
         - **entry_points**: A dictionary mapping entry point group names to strings or lists of strings defining the entry points. Entry points are used to support dynamic discovery of services or plugins provided by a project. See Dynamic Discovery of Services and Plugins for details and examples of the format of this argument. In addition, this keyword is used to support Automatic Script Creation.



EntryPoints provide a persistent, filesystem-based object name registration and name-based direct object import mechanism (implemented by the setuptools package).

They associate names of Python objects with free-form identifiers. So any other code using the same Python installation and knowing the identifier can access an object with the associated name, no matter where the object is defined. The **associated names can be any names existing in a Python module**; for example name of a class, function or variable. The entry point mechanism does not care what the name refers to, as long as it is importable.

As an example, let's use (the name of) a function, and an imaginary python module with a fully-qualified name 'myns.mypkg.mymodule':

```
def the_function():
   "function whose name is 'the_function', in 'mymodule' module"
   print "hello from the_function"
```

Entry points are registered via an entry points declaration in setup.py. To register the_function under entrypoint called 'my_ep_func':

```
    entry_points = {
        'my_ep_group_id': [
            'my_ep_func = myns.mypkg.mymodule:the_function'
        ]
    },
```

As the example shows, entry points are grouped; there's corresponding API to look up all entry points belonging to a group (example below).

Upon a package installation (ie. running 'python setup.py install'), the above declaration is parsed by setuptools. It then writes the parsed information in special file. After that, the pkg_resources API (part of setuptools) can be used to look up the entry point and access the object(s) with the associated name(s):

```
import pkg_resources

named_objects = []
for ep in pkg_resources.iter_entry_points(group='my_ep_group_id'):
   named_objects.append(ep.load())
```

Here, setuptools read the entry point information that was written in special files. It found the entry point, imported the module (myns.mypkg.mymodule), and retrieved the_function defined there, upon call to pkg_resources.load().

Assuming there were no other entry point registrations for the same group id, calling the_function would then be simple:

```
>>> named_objects[0]()
hello from the_function
```

Thus, while perhaps a bit difficult to grasp at first, the entry point mechanism is actually quite simple to use. It provides an useful tool for pluggable Python software development.

- **extras_require**: A dictionary mapping names of “extras” (optional features of your project) to strings or lists of strings specifying what other distributions must be installed to support those features. See the section below on Declaring Dependencies for details and examples of the format of this argument.
- **setup_requires**: A string or list of strings specifying what other distributions need to be present in order for the setup script to run. setuptools will attempt to obtain these (even going so far as to download them using EasyInstall) before processing the rest of the setup script or commands. This argument is needed if you are using distutils extensions as part of your build process; for example, extensions that process setup() arguments and turn them into EGG-INFO metadata files.
  (Note: projects listed in setup_requires will NOT be automatically installed on the system where the setup script is being run. They are simply downloaded to the setup directory if they’re not locally available already. If you want them to be installed, as well as being available when the setup script is run, you should add them to install_requires and setup_requires.)
- **dependency_links**: A list of strings naming URLs to be searched when satisfying dependencies. These links will be used if needed to install packages specified by setup_requires or tests_require. They will also be written into the egg’s metadata for use by tools like EasyInstall to use when installing an .egg file.
- **namespace_packages**: A list of strings naming the project’s “namespace packages”. A namespace package is a package that may be split across multiple project distributions. For example, Zope’s zope package is a namespace package, because subpackages like zope.interface and zope.publisher may be distributed separately. The egg runtime system can automatically merge such subpackages into a single parent package at runtime, as long as you declare them in each project that contains any subpackages of the namespace package, and as long as the namespace package’s __init__.py does not contain any code other than a namespace declaration. See the section below on Namespace Packages for more information.
- **test_suite**: A string naming a unittest.TestCase subclass (or a package or module containing one or more of them, or a method of such a subclass), or naming a function that can be called with no arguments and returns a unittest.TestSuite. If the named suite is a module, and the module has an additional_tests() function, it is called and the results are added to the tests to be run. If the named suite is a package, any submodules and subpackages are recursively added to the overall test suite. Specifying this argument enables use of the test command to run the specified test suite, e.g. via setup.py test. See the section on the test command below for more details.
- **tests_require**: If your project’s tests need one or more additional packages besides those needed to install it, you can use this option to specify them. It should be a string or list of strings specifying what other distributions need to be present for the package’s tests to run. When you run the test command, setuptools will attempt to obtain these (even going so far as to download them using EasyInstall). Note that these required projects will not be installed on the system where the tests are run, but only downloaded to the project’s setup directory if they’re not already installed locally.
- **test_loader**: If you would like to use a different way of finding tests to run than what setuptools normally uses, you can specify a module name and class name in this argument. The named class must be instantiable with no arguments, and its instances must support the loadTestsFromNames() method as defined in the Python unittest module’s TestLoader class. Setuptools will pass only one test “name” in the names argument: the value supplied for the test_suite argument. The loader you specify may interpret this string in any way it likes, as there are no restrictions on what may be contained in a test_suite string. The module name and class name must be separated by a :. The default value of this argument is "setuptools.command.test:ScanningLoader". If you want to use the default unittest behavior, you can specify "unittest:TestLoader" as your test_loader argument instead. This will prevent automatic scanning of submodules and subpackages. The module and class you specify here may be contained in another package, as long as you use the tests_require option to ensure that the package containing the loader class is available when the test command is run.
- **eager_resources**: A list of strings naming resources that should be extracted together, if any of them is needed, or if any C extensions included in the project are imported. This argument is only useful if the project will be installed as a zipfile, and there is a need to have all of the listed resources be extracted to the filesystem as a unit. Resources listed here should be ‘/’-separated paths, relative to the source root, so to list a resource foo.png in package bar.baz, you would include the string bar/baz/foo.png in this argument. If you only need to obtain resources one at a time, or you don’t have any C extensions that access other files in the project (such as data files or shared libraries), you probably do NOT need this argument and shouldn’t mess with it. For more details on how this argument works, see the section below on Automatic Resource Extraction.
- **use_2to3**: Convert the source code from Python 2 to Python 3 with 2to3 during the build process. See Supporting both Python 2 and Python 3 with Setuptools for more details.
- **convert_2to3_doctests**: List of doctest source files that need to be converted with 2to3. See Supporting both Python 2 and Python 3 with Setuptools for more details.
- **use_2to3_fixers**: A list of modules to search for additional fixers to be used during the 2to3 conversion. See Supporting both Python 2 and Python 3 with Setuptools for more details.

c. Development mode

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

d. Setup.py Command Usage

```
Standard commands:
  build             build everything needed to install
  build_py          "build" pure Python modules (copy to build directory)
  build_ext         build C/C++ extensions (compile/link to build directory)
  build_clib        build C/C++ libraries used by Python extensions
  build_scripts     "build" scripts (copy and fixup #! line)
  clean             clean up temporary files from 'build' command
  install           install everything from build directory
  install_lib       install all Python modules (extensions and pure Python)
  install_headers   install C/C++ header files
  install_scripts   install scripts (Python or otherwise)
  install_data      install data files
  sdist             create a source distribution (tarball, zip file, etc.)
  register          register the distribution with the Python package index
  bdist             create a built (binary) distribution
  bdist_dumb        create a "dumb" built distribution
  bdist_rpm         create an RPM distribution
  bdist_wininst     create an executable installer for MS Windows
  upload            upload binary package to PyPI

Extra commands:
  rotate            delete older distributions, keeping N newest files
  develop           install package in 'development mode'
  setopt            set an option in setup.cfg or another config file
  saveopts          save supplied options to setup.cfg or other config file
  egg_info          create a distribution's .egg-info directory
  upload_sphinx     Upload Sphinx documentation to PyPI
  install_egg_info  Install an .egg-info directory for the package
  alias             define a shortcut to invoke one or more commands
  easy_install      Find/get/install Python packages
  bdist_egg         create an "egg" distribution
  test              run unit tests after in-place build
  build_sphinx      Build Sphinx documentation

usage: setup.py [global_opts] cmd1 [cmd1_opts] [cmd2 [cmd2_opts] ...]
   or: setup.py --help [cmd1 cmd2 ...]
   or: setup.py --help-commands
   or: setup.py cmd --help
```