Using Entry Points to Write Plugins

Introduction

An entry point is a Python object identified by a string in a project’s setup.py file. The entry point is referenced by a group and a name so that the object may be discoverable. This means that another application can search for all the installed software that has an entry point with a particular group name, and then access the Python object associated with that name.

This is extremely useful because it means it is possible to write plugins for an appropriately-designed application that can be loaded at run time.

It is important to understand that entry points are a feature of the eggs package format and are *not* a standard feature of Python. To learn about eggs, their benefits, how to install them and how to set them up, see:

* [Python Eggs](http://peak.telecommunity.com/DevCenter/PythonEggs)
* [Easy Install](http://peak.telecommunity.com/DevCenter/EasyInstall)
* [Setuptools](http://peak.telecommunity.com/DevCenter/setuptools)

It is suffice it to say that eggs are created via a similar setup.py file to the setup.py file used by Python’s own [distutils](http://docs.python.org/lib/module-distutils.html) module — except that eggs have some powerful extra features such as entry points and the ability to specify module dependencies and have them automatically installed by easy\_install when the application itself is installed.

For those developers unfamiliar with distutils: it is the standard mechanism by which Python packages should be distributed. To use it, add a setup.py file to the desired project, insert the required metadata and specify the important files. The setup.py file can be used to issue various commands which create distributions of the pacakge in various formats for users to install.

Creating Plugins

This document describes how to use entry points to create a plugin mechansim which allows new types of content to be added to a content management system but we are going to start by looking at the plugin.

Say the standard way the CMS creates a plugin is with the make\_plugin() function. In order for a plugin to be a plugin it must therefore have the function which takes the same arguments as the make\_plugin() function and returns a plugin. We are going to add some image plugins to the CMS so we setup a project with the following directory structure:

+ image\_plugins

+ \_\_init\_\_.py

+ setup.py

The image\_plugins/\_\_init\_\_.py file looks like this:

**def** make\_jpeg\_image\_plugin():

**return** "This would return the JPEG image plugin"

**def** make\_png\_image\_plugin():

**return** "This would return the PNG image plugin"

We have now defined our plugins so we need to define our entry points. First let’s write a basic setup.py for the project:

**from** **setuptools** **import** setup, find\_packages

setup(

name='ImagePlugins',

version="1.0",

description="Image plugins for the imaginary CMS 1.0 project",

author="James Gardner",

packages=find\_packages(),

include\_package\_data=True,

)

When using setuptools we can specify the find\_packages() function and include\_package\_data=True rather than having to manually list all the modules and package data like we had to do in the old distutils setup.py.

Because the plugin is designed to work with the (imaginary) CMS 1.0 package, we need to specify that the plugin requires the CMS to be installed too and so we add this line to the call to setup() function:

install\_requires=["CMS>=1.0"],

Now when the plugins are installed, CMS 1.0 or above will be installed automatically if it is not already present.

There are lots of other arguments such as author\_email or url which you can add to the setup.py function too.

We are interested in adding the entry points. We need to specify a group name for the entry points. It is traditional to use the name of the package, separated by a . character and then use a name that describes what the entry point does. For our example cms.plugin might be an appropriate name for the entry point. Since the image\_plugin module contains two plugins we will need two entries. Add the following to the setup.py function:

entry\_points="""

[cms.plugin]

jpg\_image=image\_plugin:make\_jpeg\_image\_plugin

png\_image=image\_plugin:make\_png\_image\_plugin

""",

Group names are specified in square brackets, plugin names (i.e. entry point name) are specified in the format:

**name**=**module**.import.path:**object**\_within\_the\_module

The **object** doesn’t have to be a function and can have any valid Python name. The **module** import path doesn’t have to be a top level component as it is in this example and the **name** of the entry point doesn’t have to be the same as the name of the object it is pointing to.

The developer can add as many entries as desired in each group as long as the names are different and the same holds for adding groups. It is also possible to specify the entry points as a Python dictionary rather than a string if that approach is preferred.

There are two more things we need to do to complete the plugin. The first is to include an ez\_setup module so that if the user installing the plugin doesn’t have setuptools installed, it will be installed for them. We do this by adding the following to the very top of the setup.py file before the import:

**from** **ez\_setup** **import** use\_setuptools

use\_setuptools()

We also need to download the ez\_setup.py file into our project directory at the same level as setup.py.

If you keep your project in SVN there is a [trick you can use with the `SVN:externals](http://peak.telecommunity.com/DevCenter/setuptools#managing-multiple-projects) to keep the ez\_setup.py file up to date.

Finally in order for the CMS to find the plugins we need to install them. We can do this with:

$ python setup.py install

as usual or, since we might go on to develop the plugins further we can install them using a special development mode which sets up the paths to run the plugins from the source rather than installing them to Python’s site-packages directory:

$ python setup.py develop

Both commands will download and install setuptools if you don’t already have it installed.

Using Plugins

Now that the plugin is written we need to write the code in the CMS package to load it. Luckily this is even easier.

There are actually lots of ways of discovering plugins. For example: by distribution name and version requirement (such as ImagePlugins>=1.0) or by the entry point group and name (eg jpg\_image). For this example we are choosing the latter, here is a simple script for loading the plugins:

**from** **pkg\_resources** **import** iter\_entry\_points

**for** entry\_point **in** iter\_entry\_points(group='cms.plugin', name=None):

**print**(entry\_point)

**from** **pkg\_resources** **import** iter\_entry\_points

available\_methods = []

**for** entry\_point **in** iter\_entry\_points(group='authkit.method', name=None):

available\_methods.append(entry\_point.load())

Executing this short script, will result in the following output:

This would return the JPEG image plugin

This would return the PNG image plugin

The iter\_entry\_points() function has looped though all the objects in the cms.plugin group and returned the function they were associated with. The application then called the function that the entry point was pointing to.

We hope that we have demonstrated the power of entry points for building extensible code and developers are encouraged to read the [pkg\_resources](http://peak.telecommunity.com/DevCenter/PkgResources) module documentation to learn about some more features of the eggs format.

Creating Plugins

After a lot of trial and error, the easiest way I have found to define an API is to follow these steps:

1. Use the [abc module](http://docs.python.org/2/library/abc.html) to create a base abstract class to define the behaviors required of plugins of the API. Developers don’t have to subclass from the base class, but it provides a convenient way to document the API, and using an abstract base class keeps you honest.
2. Create plugins by subclassing the base class and implementing the required methods.
3. Define a unique namespace for each API by combining the name of the application (or library) and a name of the API. Keep it shallow. For example, “cliff.formatters” or “ceilometer.pollsters.compute”.

Example Plugin Set

The example program in this tutorial will create a plugin set with several data formatters, like what might be used by a command line program to prepare data to be printed to the console. Each formatter will take as input a dictionary with string keys and built-in data types as values. It will return as output an iterator that produces the string with the data structure formatted based on the rules of the specific formatter being used. The formatter’s constructor lets the caller specify the maximum width the output should have.

A Plugin Base Class

Step 1 above is to define an abstract base class for the API that needs to be implemented by each plugin.

*# stevedore/example/base.py*

**import** **abc**

**import** **six**

**@six.add\_metaclass**(abc.ABCMeta)

**class** **FormatterBase**(object):

*"""Base class for example plugin used in the tutorial.*

*"""*

**def** \_\_init\_\_(self, max\_width=60):

self.max\_width = max\_width

**@abc.abstractmethod**

**def** format(self, data):

*"""Format the data and return unicode text.*

*:param data: A dictionary with string keys and simple types as*

*values.*

*:type data: dict(str:?)*

*:returns: Iterable producing the formatted text.*

*"""*

The constructor is a concrete method because subclasses do not need to override it, but the format() method does not do anything useful because there is no “default” implementation available.

Concrete Plugins

The next step is to create a couple of plugin classes with concrete implementations of format(). A simple example formatter produces output with each variable name and value on a single line.

*# stevedore/example/simple.py*

**from** **stevedore.example** **import** base

**class** **Simple**(base.FormatterBase):

*"""A very basic formatter.*

*"""*

**def** format(self, data):

*"""Format the data and return unicode text.*

*:param data: A dictionary with string keys and simple types as*

*values.*

*:type data: dict(str:?)*

*"""*

**for** name, value **in** sorted(data.items()):

line = '{name} = {value}**\n**'.format(

name=name,

value=value,

)

**yield** line

There are plenty of other formatting options, but this example will give us enough to work with to demonstrate registering and using plugins.

Registering the Plugins

To use setuptools entry points, you must package your application or library using setuptools. The build and packaging process generates metadata which is available after installation to find the plugins provided by each python distribution.

The entry points must be declared as belonging to a specific namespace, so we need to pick one before going any further. These plugins are formatters from the stevedore examples, so I will use the namespace “stevedore.example.formatter”. Now it is possible to provide all of the necessary information in the packaging instructions:

*# stevedore/example/setup.py*

**from** **setuptools** **import** setup, find\_packages

setup(

name='stevedore-examples',

version='1.0',

description='Demonstration package for stevedore',

author='Doug Hellmann',

author\_email='doug@doughellmann.com',

url='http://git.openstack.org/cgit/openstack/stevedore',

classifiers=['Development Status :: 3 - Alpha',

'License :: OSI Approved :: Apache Software License',

'Programming Language :: Python',

'Programming Language :: Python :: 2',

'Programming Language :: Python :: 2.7',

'Programming Language :: Python :: 3',

'Programming Language :: Python :: 3.4',

'Intended Audience :: Developers',

'Environment :: Console',

],

platforms=['Any'],

scripts=[],

provides=['stevedore.examples',

],

packages=find\_packages(),

include\_package\_data=True,

entry\_points={

'stevedore.example.formatter': [

'simple = stevedore.example.simple:Simple',

'plain = stevedore.example.simple:Simple',

],

},

zip\_safe=False,

)

The important lines are near the bottom where the entry\_points argument to setup() is set. The value is a dictionary mapping the namespace for the plugins to a list of their definitions. Each item in the list should be a string with name = module:importable where *name* is the user-visible name for the plugin, *module* is the Python import reference for the module, and *importable* is the name of something that can be imported from inside the module.

'simple = stevedore.example.simple:Simple',

'plain = stevedore.example.simple:Simple',

],

},

zip\_safe=False,

)

In this case, there are two plugins registered. The “simple” plugin defined above, and a “plain” plugin, which is just an alias for the simple plugin.

setuptools Metadata

During the build, setuptools copies entry point definitions to a file in the ”.egg-info” directory for the package. For example, the file for stevedore is located in stevedore.egg-info/entry\_points.txt:

[stevedore.example.formatter]

simple = stevedore.example.simple:Simple

plain = stevedore.example.simple:Simple

[stevedore.test.extension]

t2 = stevedore.tests.test\_extension:FauxExtension

t1 = stevedore.tests.test\_extension:FauxExtension

pkg\_resources uses the entry\_points.txt file from all of the installed packages on the import path to find plugins. You should not modify these files, except by changing the list of entry points in setup.py.

Adding Plugins in Other Packages

Part of the appeal of using entry points for plugins is that they can be distributed independently of an application. The namespace setuptools uses to find the plugins is different from the Python source code namespace. It is common to use a plugin namespace prefixed with the name of the application or library that loads the plugins, to ensure it is unique, but that name has no bearing on what Python package the code for the plugin should live in.

For example, we can add an alternate implementation of a formatter plugin that produces a reStructuredText [field list](http://docutils.sourceforge.net/docs/ref/rst/restructuredtext.html#field-lists).

*# stevedore/example2/fields.py*

**import** **textwrap**

**from** **stevedore.example** **import** base

**class** **FieldList**(base.FormatterBase):

*"""Format values as a reStructuredText field list.*

*For example::*

*: name1 : value*

*: name2 : value*

*: name3 : a long value*

*will be wrapped with*

*a hanging indent*

*"""*

**def** format(self, data):

*"""Format the data and return unicode text.*

*:param data: A dictionary with string keys and simple types as*

*values.*

*:type data: dict(str:?)*

*"""*

**for** name, value **in** sorted(data.items()):

full\_text = ': {name} : {value}'.format(

name=name,

value=value,

)

wrapped\_text = textwrap.fill(

full\_text,

initial\_indent='',

subsequent\_indent=' ',

width=self.max\_width,

)

**yield** wrapped\_text + '**\n**'

The new plugin can then be packaged using a setup.py containing

*# stevedore/example2/setup.py*

**from** **setuptools** **import** setup, find\_packages

setup(

name='stevedore-examples2',

version='1.0',

description='Demonstration package for stevedore',

author='Doug Hellmann',

author\_email='doug@doughellmann.com',

url='http://git.openstack.org/cgit/openstack/stevedore',

classifiers=['Development Status :: 3 - Alpha',

'License :: OSI Approved :: Apache Software License',

'Programming Language :: Python',

'Programming Language :: Python :: 2',

'Programming Language :: Python :: 2.7',

'Programming Language :: Python :: 3',

'Programming Language :: Python :: 3.4',

'Intended Audience :: Developers',

'Environment :: Console',

],

platforms=['Any'],

scripts=[],

provides=['stevedore.examples2',

],

packages=find\_packages(),

include\_package\_data=True,

entry\_points={

'stevedore.example.formatter': [

'field = stevedore.example2.fields:FieldList',

],

},

zip\_safe=False,

)

The new plugin is in a separate stevedore-examples2 package.

setup(

name='stevedore-examples2',

However, the plugin is registered as part of the stevedore.example.formatter namespace.

'stevedore.example.formatter': [

'field = stevedore.example2.fields:FieldList',

],

},

When the plugin namespace is scanned, all packages on the current PYTHONPATH are examined and the entry point from the second package is found and can be loaded without the application having to know where the plugin is actually installed.