straight.plugin Documentation

Release 1.4.0

Calvin Spealman

February 04, 2016

Contents

1 Getting Started 1

2 Writing Plugins 3

3 Plugin Loaders 5

4 Straight Plugin API 7

5 Glossary 9

6 Overview 11

7 Indices and tables 13

CHAPTER 1

Getting Started

1.1 Install

```
pip install straight.plugin
```

That was super easy.

1.2 Decide on a Namespace

You'll want to decide on a namespace within your package where you'll keep your own plugins and where other developers can add more plugins for your package to use.

For example, if you're writing a log filtering library named `logfilter` you may choose `logfilter.plugins` as a package to hold your plugins, so you'll create the empty package as you would any other python package. However, the only contents of `logfilter/plugins/__init__.py` will be a little bit of special code telling python this is a *namespace package*.

```
# This file will not be needed in Python 3.3
from pkgutil import extend_path
__path__ = extend_path(__path__, __name__)
```

Now, any modules you place in this package are plugin modules able to be loaded by `straight.plugin`.

```
from straight.plugin import load

plugins = load("logfilter.plugins")
```

If you'll be using more plugins than writing them, you should read more about the loaders available and how they work.

1.3 Write a Plugin

Writing a plugin is even easier than loading them. There are two important plugin types to learn: Module plugins and class Plugins. Every module in your *namespace package* is a module plugin. Every class they define is a class plugin.

When you load module plugins, you get all of them.

When you load class plugins, you filter them by a common base and only get those class plugins which inherit it.

Module plugins are simple and usually define a few functions with names expected by whoever is loading and using the plugins.

```
# This is a module plugin

def add_extra(data):
    if 'x' in data and 'y' in data:
        data['z'] = x * y

# This was a fairly useless plugin
```

Class plugins are only a little longer, but can be a bit more controlled to work with. They depend on a common class the plugins inherit, and this would be defined by the project loading and using the plugins.

```
# This is a class plugin

class RstContentParser(ContentPlugin):
    """Parses any .rst files in a bundle."""

    extensions = ('.rst',)

    def parse(self, content_file):
        src = content_file.read()
        return self.parse_string(src)

    def parse_string(self, src):
        parts = publish_parts(source=src, writer_name='html')
        return parts['html_body']
```

You can fit as many class plugins inside a module plugin as you want, and to load them instead of the modules you simply pass a `subclasses` parameter to `load()`.

```
from straight.plugin import load

plugins = load("jules.plugins", subclasses=ContentPlugin)
```

The resulting set of plugins are all the classes found which inherit from ContentPlugin. You can do whatever you want with these, but there are some helpful tools to make it easier to work with Class plugins.

You can easily create instances of all the classes, which gives you a set of Instance plugins.

```
instances = plugins.produce()
```

You can even pass initialization parameters to `produce()` and they'll be used when creating instances of all the classes. You can see the *API docs* for the `PluginManager` to see the other ways you can work with groups of plugins.

CHAPTER 2

Writing Plugins

Plugins can exist inside your existing packages or in special namespace packages, which exist only to house plugins.

The only requirement is that any package containing plugins be designated a "namespace package", which is currently performed in Python via the `pkgutil.extend_path` utility, seen below. This allows the namespace to be provided in multiple places on the python `sys.path`, where `import` looks, and all the contents will be combined.

Use a *namespace package*

This allows multiple packages installed on your system to share this name, so they may come from different installed projects and all combine to provide a larger set of plugins.

2.1 Example

```
# logfilter/__init__.py

from pkgutil import extend_path
__path__ = extend_path(__path__, __name__)
```

```
# logfilter/hide_extra.py

from logfilter import Skip

def filter(log_entry):
    level = log_entry.split(':', 1)[0]
    if level != 'EXTRA':
        return log_entry
    else:
        raise Skip()
```

2.1.1 Using the plugin

In our log tool, we might load all the modules in the `logfilter` namespace, and then use them all to process each entry in our logs. We don't need to know all the filters ahead of time, and other packages can be installed on a user's system providing additional modules in the namespace, which we never even knew about.

```
from straight.plugin import load

class Skip(Exception):
    pass
```

```
plugins = load('logfilter')

def filter_entry(log_entry):
    for plugin in plugins:
        try:
            log_entry = plugin.filter(log_entry)
        except Skip:
            pass
    return log_entry
```

2.1.2 Distributing Plugins

If you are writing plugins inside your own project to use, they'll be distributed like any other modules in your package. There is no extra work to do here.

However, if you want to release and distribute plugins on their own, you'll need to tell your setup.py about your *namespace package*.

```
setup(
    # ...
    namespace_packages = ['logfilter.plugins']
)
```

This will make sure when your plugins are installed alongside the original project, both are importable, even though they came from their own distributions.

You can read more about this at the Distribute documentation on namespace packages.

CHAPTER 3

Plugin Loaders

Currently, three simple loaders are provided.

- The *ModuleLoader* simply loads the modules found
- The *ClassLoader* loads the subclasses of a given type
- The *ObjectLoader* loads arbitrary objects from the modules

3.1 ClassLoader

The recommended loader is the `ClassLoader`, used to load all the classes from all of the modules in the namespace given. Optionally, you can pass a `subclasses` parameter to `load()`, which will filter the loaded classes to those which are a sub-class of any given type.

For example,

```
import os
from straight.plugin.loaders import ClassLoader
from myapp import FileHandler

plugins = ClassLoader().load('myplugins', subclasses=FileHandler)

for filename in os.listdir('.'):
    for handler_cls in plugins:
        handler = handler_cls(filename)
        if handler.valid():
            handler.process()
```

However, it is preferred that you use the `load()` helper provided.

```
from straight.plugin import load

plugins = load('myplugins', subclasses=FileHandler)
```

This will automatically use the `ClassLoader` when given a `subclasses` argument.

3.2 ModuleLoader

Before anything else, `straight.plugin` loads modules. The `ModuleLoader` is used to do this.

```
from straight.plugin.loaders import ModuleLoader

plugins = ModuleLoader().load('myplugins')
```

A note about PEP-420:

Python 3.3 will support a new type of package, the Namespace Package. This allows language-level support for the namespaces that make `straight.plugin` work and when 3.3 lands, you can create addition plugins to be found in a namespace. For now, continue to use the `pkgutil` boilerplate, but when 3.3 is released, `straight.plugin` already supports both forms of namespace package!

3.3 ObjectLoader

If you need to combine multiple plugins inside each module, you can load all the objects from the modules, rather than the modules themselves.

```
from straight.plugin.loaders import ObjectLoader

plugins = ObjectLoader().load('myplugins')
```

CHAPTER 4

Straight Plugin API

4.1 Loaders

straight.plugin.loaders.**unified_load**(*namespace*, *subclasses=None*, *recurse=False*)
Provides a unified interface to both the module and class loaders, finding modules by default or classes if given a `subclasses` parameter.

class straight.plugin.loaders.**Loader**(**args*, ***kwargs*)
Base loader class. Only used as a base-class for other loaders.

class straight.plugin.loaders.**ModuleLoader**(*recurse=False*)
Performs the work of locating and loading straight plugins.

This looks for plugins in every location in the import path.

class straight.plugin.loaders.**ObjectLoader**(*recurse=False*)
Loads classes or objects out of modules in a namespace, based on a provided criteria.

The load() method returns all objects exported by the module.

class straight.plugin.loaders.**ClassLoader**(*recurse=False*)
Loads classes out of plugin modules which are subclasses of a single given base class.

4.2 PluginManager

class straight.plugin.manager.**PluginManager**(*plugins*)

call(*methodname*, **args*, ***kwargs*)
Call a common method on all the plugins, if it exists.

first(*methodname*, **args*, ***kwargs*)
Call a common method on all the plugins, if it exists. Return the first result (the first non-None)

pipe(*methodname*, *first_arg*, **args*, ***kwargs*)
Call a common method on all the plugins, if it exists. The return value of each call becomes the replaces the first argument in the given argument list to pass to the next.

Useful to utilize plugins as sets of filters.

produce(**args*, ***kwargs*)
Produce a new set of plugins, treating the current set as plugin factories.

CHAPTER 5

Glossary

distribution Separately installable sets of Python modules as stored in the Python package index, and installed by distutils or setuptools.

definition taken from PEP 382 *text*

module An importable python namespace defined in a single file.

namespace package Mechanism for splitting a single Python package across multiple directories on disk. One or more distributions (see *distribution*) may provide modules which exist inside the same *namespace package*.

definition taken from PEP 382 *text*

package A Python package is a module defined by a directory, containing a `__init__.py` file, and can contain other modules or other packages within it.

```
package/
    __init__.py
    subpackage/
        __init__.py
    submodule.py
```

see also, *namespace package*

vendor package Groups of files installed by an operating system's packaging mechanism (e.g. Debian or Redhat packages install on Linux systems).

definition taken from PEP 382 *text*

CHAPTER 6

Overview

Straight Plugin is very easy.

Straight Plugin provides a type of plugin you can create from almost any existing Python modules, and an easy way for outside developers to add functionality and customization to your projects with their own plugins.

Using any available plugins is a snap.

```
from straight.plugin import load

plugins = load('theproject.plugins', subclasses=FileHandler)

handlers = plugins.produce()
for line in open(filename):
    print handlers.pipe(line)
```

And, writing plugins is just as easy.

```
from theproject import FileHandler

class LineNumbers(FileHandler):
    def __init__(self):
        self.lineno = 0
    def pipe(line):
        self.lineno += 1
        return "%04d %s" % (self.lineno, line)
```

Plugins are found from a namespace, which means the above example would find any `FileHandler` classes defined in modules you might import as `theproject.plugins.default` or `theproject.plugins.extra`. Through the magic of *namespace packages*, we can even split these up into separate installations, even managed by different teams. This means you can ship a project with a set of default plugins implementing its behavior, and allow other projects to hook in new functionality simply by shipping their own plugins under the same namespace.

Get started and learn more, today

6.1 More Resources

- Full Documentation: http://readthedocs.org/docs/straightplugin/
- Mailing List: https://groups.google.com/forum/#!forum/straight.plugin

CHAPTER 7

Indices and tables

- genindex
- modindex
- search

C

call() (straight.plugin.manager.PluginManager method), 7
ClassLoader (class in straight.plugin.loaders), 7

D

distribution, **9**

F

first() (straight.plugin.manager.PluginManager method), 7

L

Loader (class in straight.plugin.loaders), 7

M

module, **9**
ModuleLoader (class in straight.plugin.loaders), 7

N

namespace package, **9**

O

ObjectLoader (class in straight.plugin.loaders), 7

P

package, **9**
pipe() (straight.plugin.manager.PluginManager method), 7
PluginManager (class in straight.plugin.manager), 7
produce() (straight.plugin.manager.PluginManager method), 7

U

unified_load() (in module straight.plugin.loaders), 7

V

vendor package, **9**