pluginmanager Documentation

Release 0.1.8

Ben Hoff

January 24, 2016

Contents

python plugin management, simplified.

Source Code

Library under development. Contains rough edges/unfinished functionality. API subject to changes.

CHAPTER 1

Installation

```
pip install pluginmanager
```

-or-

```
pip install git+https://github.com/benhoff/pluginmanager.git
```

CHAPTER 2

Quickstart

```
from pluginmanager import PluginInterface

plugin_interface = PluginInterface()
plugin_interface.set_plugin_directories(plugin_directory_path)
plugin_interface.collect_plugins() # doctest: +SKIP

plugins = plugin_interface.get_instances()
print(plugins) # doctest: +SKIP +HIDE
```

CHAPTER 3

Custom Plugins

The quickstart will only work if you subclass *IPlugin* for your custom plugins.

```
import pluginmanager

class MyCustomPlugin(pluginmanager.IPlugin):
    def __init__(self):
        self.name = 'custom_name'
        super().__init__()
```

Or register your class as subclass of IPlugin.

```
import pluginmanager

pluginmanager.IPlugin.register(YourClassHere)
```

CHAPTER 4

Add Plugins Manually

Add classes.

```
import pluginmanager

class CustomClass(pluginmanager.IPlugin):
    pass

plugin_interface = pluginmanager.PluginInterface()
plugin_interface.add_plugins(CustomClass)

plugins = plugin_interface.get_instances()
print(plugins) # doctest: +SKIP
```

Alternatively, add instances.

```
import pluginmanager

class CustomClass(pluginmanager.IPlugin):
    pass

custom_class_instance = CustomClass()

plugin_interface = pluginmanager.PluginInterface()
plugin_interface.add_plugins(custom_class_instance)

plugins = plugin_interface.get_instances()
print(plugins) # doctest: +SKIP
```

pluginmanager is defaulted to automatically instantiate unique instances. Disable automatic instantiation.

```
import pluginmanager

plugin_interface = pluginmanager.PluginInterface()
plugin_manager = plugin_interface.plugin_manager

plugin_manager.instantiate_classes = False
```

Disable uniqueness (Only one instance of class per pluginmanager)

```
import pluginmanager

plugin_interface = pluginmanager.PluginInterface()
plugin_manager = plugin_interface.plugin_manager
```

```
plugin_manager.unique_instances = False
```

CHAPTER 5

Filter Instances

Pass in a class to get back just the instances of a class

```
import pluginmanager

class MyPluginClass(pluginmanager.IPlugin):
    pass

plugin_interface = pluginmanager.PluginInterface()
plugin_interface.add_plugins(MyPluginClass)

all_instances_of_class = plugin_interface.get_instances(MyPluginClass)
print(all_instances_of_class) # doctest: +SKIP
```

Alternatively, create and pass in your own custom filters. Either make a class based filter

```
# create a custom plugin class
class Plugin(pluginmanager.IPlugin):
    def __init__(self, name):
        self.name = name

# create a custom filter
class NameFilter(object):
    def __init__(self, name):
        self.stored_name = name

    def __call__(self, plugins):
        result = []
        for plugin in plugins:
            if plugin.name == self.stored_name:
                result.append(plugin)
        return result

# create an instance of our custom filter
mypluginclass_name_filter = NameFilter('good plugin')

plugin_interface = pluginmanager.PluginInterface()
plugin_interface.add_plugins([Plugin('good plugin'),
                              Plugin('bad plugin')])

filtered_plugins = plugin_interface.get_instances(mypluginclass_name_filter)
print(filtered_plugins[0].name) # doctest: +SKIP
```

Or make a function based filter

```
# create a custom plugin class
class Plugin(pluginmanager.IPlugin):
    def __init__(self, name):
        self.name = name

# create a function based filter
def custom_filter(plugins):
    result = []
    for plugin in plugins:
        if plugin.name == 'good plugin':
            result.append(plugin)
    return result

plugin_interface = pluginmanager.PluginInterface()
plugin_interface.add_plugins([Plugin('good plugin'),
                              Plugin('bad plugin')])

filtered_plugins = plugin_interface.get_instances(mypluginclass_name_filter)
print(filtered_plugins[0].name)
```

5.1 API Reference

5.1.1 FileManager

class pluginmanager.**FileManager**(*file_filters=None*, *plugin_filepaths=None*, *blacklisted_filepaths=None*)

FileManager manages the file filter state and is responible for collecting filepaths from a set of directories and filtering the files through the filters. Without file filters, this class acts as a passthrough, collecting and returning every file in a given directory.

FileManager can also optionally manage the plugin filepath state through the use of the add/get/set plugin filepaths methods. Note that plugin interface is not automatically set up this way, although it is relatively trivial to do.

add_blacklisted_filepaths(*filepaths*, *remove_from_stored=True*)

Add *filepaths* to blacklisted filepaths. If *remove_from_stored* is *True*, any *filepaths* in *plugin_filepaths* will be automatically removed.

Recommend passing in absolute filepaths but method will attempt to convert to absolute filepaths based on current working directory.

add_file_filters(*file_filters*)

Adds *file_filters* to the internal file filters. *file_filters* can be single object or iterable.

add_plugin_filepaths(*filepaths*, *except_blacklisted=True*)

Adds *filepaths* to the *self.plugin_filepaths*. Recommend passing in absolute filepaths. Method will attempt to convert to absolute paths if they are not already.

filepaths can be a single object or an iterable

If *except_blacklisted* is *True*, all *filepaths* that have been blacklisted will not be added.

collect_filepaths(*directories*)

Collects and returns every filepath from each directory in *directories* that is filtered through the *file_filters*. If no *file_filters* are present, passes every file in directory as a result. Always returns a *set* object

directories can be a object or an iterable. Recommend using absolute paths.

get_blacklisted_filepaths()
Returns the blacklisted filepaths as a set object.

get_file_filters(*filter_function=None*)
Gets the file filters. *filter_function*, can be a user defined filter. Should be callable and return a list.

get_plugin_filepaths()
returns the plugin filepaths tracked internally as a *set* object.

remove_blacklisted_filepaths(*filepaths*)
Removes *filepaths* from blacklisted filepaths

Recommend passing in absolute filepaths but method will attempt to convert to absolute filepaths based on current working directory.

remove_file_filters(*file_filters*)
Removes the *file_filters* from the internal state. *file_filters* can be a single object or an iterable.

remove_plugin_filepaths(*filepaths*)
Removes *filepaths* from *self.plugin_filepaths*. Recommend passing in absolute filepaths. Method will attempt to convert to absolute paths if not passed in.

filepaths can be a single object or an iterable.

set_blacklisted_filepaths(*filepaths*, *remove_from_stored=True*)
Sets internal blacklisted filepaths to filepaths. If *remove_from_stored* is *True*, any *filepaths* in *self.plugin_filepaths* will be automatically removed.

Recommend passing in absolute filepaths but method will attempt to convert to absolute filepaths based on current working directory.

set_file_filters(*file_filters*)
Sets internal file filters to *file_filters* by tossing old state. *file_filters* can be single object or iterable.

set_plugin_filepaths(*filepaths*, *except_blacklisted=True*)
Sets *filepaths* to the *self.plugin_filepaths*. Recommend passing in absolute filepaths. Method will attempt to convert to absolute paths if they are not already.

filepaths can be a single object or an iterable.

If *except_blacklisted* is *True*, all *filepaths* that have been blacklisted will not be set.

5.1.2 DirectoryManager

class pluginmanager.**DirectoryManager**(*plugin_directories=None*, *recursive=True*, *blacklisted_directories=None*)
DirectoryManager manages the recursive search state and can optionally manage directory state. The default implementation of pluginmanager uses *DirectoryManager* to manage the directory state.

DirectoryManager contains a directory blacklist, which can be used to stop from collecting from uninteresting directories.

DirectoryManager manages directory state through the add/get/set directories methods.

NOTE: When calling *collect_directories* the directories must be explicitly passed into the method call. This is to avoid tight coupling from the internal state and promote reuse at the Interface level.

add_blacklisted_directories(*directories*, *remove_from_stored_directories=True*)
Adds *directories* to be blacklisted. Blacklisted directories will not be returned or searched recursively when calling the *collect_directories* method.

directories may be a single instance or an iterable. Recommend passing in absolute paths, but method will try to convert to absolute paths based on the current working directory.

If *remove_from_stored_directories* is true, all *directories* will be removed from *self.plugin_directories*

add_directories(*directories*, *except_blacklisted=True*)
Adds *directories* to the set of plugin directories.

directories may be either a single object or a iterable.

directories can be relative paths, but will be converted into absolute paths based on the current working directory.

if *except_blacklisted* is *True* all *directories* in *self.blacklisted_directories* will be removed

add_site_packages_paths()
A helper method to add all of the site packages tracked by python to the set of plugin directories.

NOTE that if using a virtualenv, there is an outstanding bug with the method used here. While there is a workaround implemented, when using a virutalenv this method WILL NOT track every single path tracked by python. See: https://github.com/pypa/virtualenv/issues/355

collect_directories(*directories*)
Collects all the directories into a *set* object.

If *self.recursive* is set to *True* this method will iterate through and return all of the directories and the subdirectories found from *directories* that are not blacklisted.

if *self.recursive* is set to *False* this will return all the directories that are not balcklisted.

directories may be either a single object or an iterable. Recommend passing in absolute paths instead of relative. *collect_directories* will attempt to convert *directories* to absolute paths if they are not already.

get_blacklisted_directories()
Returns the set of the blacklisted directories.

get_directories()
Returns the plugin directories in a *set* object

remove_blacklisted_directories(*directories*)
Attempts to remove the *directories* from the set of blacklisted directories. If a particular directory is not found in the set of blacklisted, method will continue on silently.

directories may be a single instance or an iterable. Recommend passing in absolute paths. Method will try to convert to an absolute path if it is not already using the current working directory.

remove_directories(*directories*)
Removes any *directories* from the set of plugin directories.

directories may be a single object or an iterable.

Recommend passing in all paths as absolute, but the method will attemmpt to convert all paths to absolute if they are not already based on the current working directory.

set_blacklisted_directories(*directories*, *remove_from_stored_directories=True*)
Sets the *directories* to be blacklisted. Blacklisted directories will not be returned or searched recursively when calling *collect_directories*.

This will replace the previously stored set of blacklisted paths.

directories may be a single instance or an iterable. Recommend passing in absolute paths. Method will try to convert to absolute path based on current working directory.

set_directories(*directories*, *except_blacklisted=True*)
Sets the plugin directories to *directories*. This will delete the previous state stored in *self.plugin_directories* in favor of the *directories* passed in.

directories may be either a single object or an iterable.

directories can contain relative paths but will be converted into absolute paths based on the current working directory.

if *except_blacklisted* is *True* all *directories* in *self.blacklisted_directories* will be removed

5.1.3 ModuleManager

class pluginmanager.**ModuleManager**(*module_plugin_filters=None*)
ModuleManager manages the module plugin filter state and is responsible for both loading the modules from source code and collecting the plugins from each of the modules.

ModuleManager can also optionally manage modules explicitly through the use of the add/get/set loaded modules methods. The default implementation is hardwired to use the tracked loaded modules if no modules are passed into the *collect_plugins* method.

add_module_plugin_filters(*module_plugin_filters*)
Adds *module_plugin_filters* to the internal module filters. May be a single object or an iterable.

Every module filters must be a callable and take in a list of plugins and their associated names.

add_to_loaded_modules(*modules*)
Manually add in *modules* to be tracked by the module manager.

modules may be a single object or an iterable.

collect_plugins(*modules=None*)
Collects all the plugins from *modules*. If modules is None, collects the plugins from the loaded modules.

All plugins are passed through the module filters, if any are any, and returned as a list.

get_loaded_modules()
Returns all modules loaded by this instance.

get_module_plugin_filters(*filter_function=None*)
Gets the internal module filters. Returns a list object.

If supplied, the *filter_function* should take in a single list argument and return back a list. *filter_function* is designed to given the option for a custom filter on the module filters.

load_modules(*filepaths*)
Loads the modules from their *filepaths*. A filepath may be a directory filepath if there is an *__init__.py* file in the directory.

If a filepath errors, the exception will be caught and logged in the logger.

Returns a list of modules.

remove_module_plugin_filters(*module_plugin_filters*)
Removes *module_plugin_filters* from the internal module filters. If the filters are not found in the internal representation, the function passes on silently.

module_plugin_filters may be a single object or an iterable.

set_module_plugin_filters(*module_plugin_filters*)
Sets the internal module filters to *module_plugin_filters module_plugin_filters* may be a single object or an iterable.

Every module filters must be a callable and take in a list of plugins and their associated names.

5.1.4 PluginManager

class pluginmanager.**PluginManager**(*unique_instances=True*, *instantiate_classes=True*, *plugins=None*, *blacklisted_plugins=None*)

PluginManager manages the plugin state. It can automatically instantiate classes and enforce uniqueness, which it does by default.

activate_plugins()
helper method that attempts to activate plugins checks to see if plugin has method call before calling it.

add_blacklisted_plugins(*plugins*)
add blacklisted plugins. *plugins* may be a single object or iterable.

add_plugins(*plugins*)
Adds plugins to the internal state. *plugins* may be a single object or an iterable.

If *instantiate_classes* is True and the plugins have class instances in them, attempts to instatiate the classes.

If *unique_instances* is True and duplicate instances are passed in, this method will not track the new instances internally.

deactivate_plugins()
helper method that attempts to deactivate plugins. checks to see if plugin has method call before calling it.

get_blacklisted_plugins()
gets blacklisted plugins tracked in the internal state Returns a list object.

get_instances(*filter_function=<class 'pluginmanager.iplugin.IPlugin'>*)
Gets instances out of the internal state using the default filter supplied in filter_function. By default, it is the class IPlugin.

Can optionally pass in a list or tuple of classes in for *filter_function* which will accomplish the same goal.

lastly, a callable can be passed in, however it is up to the user to determine if the objects are instances or not.

get_plugins(*filter_function=None*)
Gets out the plugins from the internal state. Returns a list object. If the optional filter_function is supplied, applies the filter function to the arguments before returning them. Filters should be callable and take a list argument of plugins.

register_classes(*classes*)
Register classes as plugins that are not subclassed from IPlugin. *classes* may be a single object or an iterable.

remove_blacklisted_plugins(*plugins*)
removes *plugins* from the blacklisted plugins. *plugins* may be a single object or iterable.

remove_instance(*instances*)
removes *instances* from the internal state.

Note that this method is syntatic sugar for the *remove_plugins* acts as a passthrough for that function. *instances* may be a single object or an iterable

remove_plugins(*plugins*)
removes *plugins* from the internal state

plugins may be a single object or an iterable.

set_blacklisted_plugins(*plugins*)
sets blacklisted plugins. *plugins* may be a single object or iterable.

set_plugins(*plugins*)
sets plugins to the internal state. *plugins* may be a single object or an iterable.

If *instatntiate_classes* is True and the plugins have class instances in them, attempts to instatiate the classes.

If *unique_instances* is True and duplicate instances are passed in, this method will not track the new instances internally.

CHAPTER 6

Indices and tables

- genindex
- modindex
- search

A

C

D

F

G

L

M

P

R

S