Yapsy Documentation

Release 1.11.223

Thibauld Nion

Jun 07, 2017

Contents

A simple plugin system for Python applications

Quick links:

CHAPTER 1

IPlugin

Role

Defines the basic interfaces for a plugin. These interfaces are inherited by the *core* class of a plugin. The *core* class of a plugin is then the one that will be notified the activation/deactivation of a plugin via the `activate/deactivate` methods.

For simple (near trivial) plugin systems, one can directly use the following interfaces.

Extensibility

In your own software, you'll probably want to build derived classes of the `IPlugin` class as it is a mere interface with no specific functionality.

Your software's plugins should then inherit your very own plugin class (itself derived from `IPlugin`).

Where and how to code these plugins is explained in the section about the *PluginManager*.

API

class `yapsy.IPlugin.`**`IPlugin`**
The most simple interface to be inherited when creating a plugin.

`activate``()`
Called at plugin activation.

`deactivate``()`
Called when the plugin is disabled.

CHAPTER 2

PluginManager

Role

The `PluginManager` loads plugins that enforce the *Plugin Description Policy*, and offers the most simple methods to activate and deactivate the plugins once they are loaded.

Note: It may also classify the plugins in various categories, but this behaviour is optional and if not specified elseway all plugins are stored in the same default category.

Note: It is often more useful to have the plugin manager behave like singleton, this functionality is provided by `PluginManagerSingleton`

Plugin Description Policy

When creating a `PluginManager` instance, one should provide it with a list of directories where plugins may be found. In each directory, a plugin should contain the following elements:

For a *Standard* plugin:

`myplugin.yapsy-plugin`

A *plugin info file* identical to the one previously described.

`myplugin`

A directory ontaining an actual Python plugin (ie with a `__init__.py` file that makes it importable). The upper namespace of the plugin should present a class inheriting the `IPlugin` interface (the same remarks apply here as in the previous case).

For a *Single file* plugin:

`myplugin.yapsy-plugin`

A *plugin info file* which is identified thanks to its extension, see the *Plugin Info File Format* to see what should be in this file.

The extension is customisable at the `PluginManager`'s instanciation, since one may usually prefer the extension to bear the application name.

`myplugin.py`

The source of the plugin. This file should at least define a class inheriting the `IPlugin` interface. This class will be instanciated at plugin loading and it will be notified the activation/deactivation events.

Plugin Info File Format

The plugin info file is a text file *encoded in ASCII or UTF-8* and gathering, as its name suggests, some basic information about the plugin.

- it gives crucial information needed to be able to load the plugin
- it provides some documentation like information like the plugin author's name and a short description fo the plugin functionality.

Here is an example of what such a file should contain:

```
[Core]
Name = My plugin Name
Module = the_name_of_the_pluginto_load_with_no_py_ending

[Documentation]
Description = What my plugin broadly does
Author = My very own name
Version = the_version_number_of_the_plugin
Website = My very own website
```

Note: From such plugin descriptions, the `PluginManager` will built its own representations of the plugins as instances of the *PluginInfo* class.

Changing the default behaviour

The default behaviour for locating and loading plugins can be changed using the various options exposed on the interface via getters.

The plugin detection, in particular, can be fully customized by settting a custom plugin locator. See `IPluginLocator` for more details on this.

Extensibility

Several mechanisms have been put up to help extending the basic functionalities of the proivided classes.

A few *hints* to help you extend those classes:

If the new functionalities do not overlap the ones already implemented, then they should be implemented as a Decorator class of the base plugin. This should be done by inheriting the `PluginManagerDecorator`.

If this previous way is not possible, then the functionalities should be added as a subclass of `PluginManager`.

Note: The first method is highly prefered since it makes it possible to have a more flexible design where one can pick several functionalities and litterally *add* them to get an object corresponding to one's precise needs.

API

class `yapsy.PluginManager.`**PluginManager**(*categories_filter=None*, *directories_list=None*, *plugin_info_ext=None*, *plugin_locator=None*)

Manage several plugins by ordering them in categories.

The mechanism for searching and loading the plugins is already implemented in this class so that it can be used directly (hence it can be considered as a bit more than a mere interface)

The file describing a plugin must be written in the syntax compatible with Python's ConfigParser module as in the *Plugin Info File Format*

About the __init__:

Initialize the mapping of the categories and set the list of directories where plugins may be. This can also be set by direct call the methods:

- `setCategoriesFilter` for `categories_filter`
- `setPluginPlaces` for `directories_list`
- `setPluginInfoExtension` for `plugin_info_ext`

You may look at these function's documentation for the meaning of each corresponding arguments.

activatePluginByName(*name*, *category='Default'*)
Activate a plugin corresponding to a given category + name.

appendPluginCandidate(*candidateTuple*)
Append a new candidate to the list of plugins that should be loaded.

The candidate must be represented by the same tuple described in `getPluginCandidates`.

appendPluginToCategory(*plugin*, *category_name*)
Append a new plugin to the given category.

collectPlugins()
Walk through the plugins' places and look for plugins. Then for each plugin candidate look for its category, load it and stores it in the appropriate slot of the category_mapping.

deactivatePluginByName(*name*, *category='Default'*)
Desactivate a plugin corresponding to a given category + name.

getAllPlugins()
Return the list of all plugins (belonging to all categories).

getCategories()
Return the list of all categories.

getPluginByName(*name*, *category='Default'*)
Get the plugin correspoding to a given category and name

getPluginCandidates()
Return the list of possible plugins.

Each possible plugin (ie a candidate) is described by a 3-uple: (info file path, python file path, plugin info instance)

getPluginInfoClass()
DEPRECATED(>1.9): directly control that with the IPluginLocator instance instead !

Get the class that holds PluginInfo.

getPluginLocator()
Grant direct access to the plugin locator.

getPluginsOf(***kwargs*)
Returns a set of plugins whose properties match the named arguments provided here along with their correspoding values.

getPluginsOfCategory(*category_name*)
Return the list of all plugins belonging to a category.

instanciateElement(*element*)
Override this method to customize how plugins are instanciated

loadPlugins(*callback=None*, *callback_after=None*)
Load the candidate plugins that have been identified through a previous call to locatePlugins. For each plugin candidate look for its category, load it and store it in the appropriate slot of the `category_mapping`.

You can specify 2 callbacks: callback, and callback_after. If either of these are passed a function, (in the case of callback), it will get called before each plugin load attempt and (for callback_after), after each attempt. The `plugin_info` instance is passed as an argument to each callback. This is meant to facilitate code that needs to run for each plugin, such as adding the directory it resides in to sys.path (so imports of other files in the plugin's directory work correctly). You can use callback_after to remove anything you added to the path.

locatePlugins()
Convenience method (actually call the IPluginLocator method)

removePluginCandidate(*candidateTuple*)
Remove a given candidate from the list of plugins that should be loaded.

The candidate must be represented by the same tuple described in `getPluginCandidates`.

removePluginFromCategory(*plugin*, *category_name*)
Remove a plugin from the category where it's assumed to belong.

setCategoriesFilter(*categories_filter*)
Set the categories of plugins to be looked for as well as the way to recognise them.

The `categories_filter` first defines the various categories in which the plugins will be stored via its keys and it also defines the interface tha has to be inherited by the actual plugin class belonging to each category.

setPluginInfoClass(*picls*, *strategies=None*)
DEPRECATED(>1.9): directly configure the IPluginLocator instance instead !

Convenience method (actually call self.getPluginLocator().setPluginInfoClass)

When using a `PluginFileLocator` you may restrict the strategies to which the change of PluginInfo class will occur by just giving the list of strategy names in the argument "strategies"

setPluginInfoExtension(*ext*)
DEPRECATED(>1.9): for backward compatibility. Directly configure the IPluginLocator instance instead !

> **Warning:** This will only work if the strategy "info_ext" is active for locating plugins.

setPluginLocator(*plugin_locator*, *dir_list=None*, *picls=None*)
Sets the strategy used to locate the basic information.

See `IPluginLocator` for the policy that plugin_locator must enforce.

setPluginPlaces(*directories_list*)
DEPRECATED(>1.9): directly configure the IPluginLocator instance instead !

Convenience method (actually call the IPluginLocator method)

updatePluginPlaces(*directories_list*)
DEPRECATED(>1.9): directly configure the IPluginLocator instance instead !

Convenience method (actually call the IPluginLocator method)

class `yapsy.PluginManager.`**PluginManagerSingleton**
Singleton version of the most basic plugin manager.

Being a singleton, this class should not be initialised explicitly and the `get` classmethod must be called instead.

To call one of this class's methods you have to use the `get` method in the following way: `PluginManagerSingleton.get().themethodname(theargs)`

To set up the various coonfigurables variables of the PluginManager's behaviour please call explicitly the following methods:

- `setCategoriesFilter` for `categories_filter`
- `setPluginPlaces` for `directories_list`
- `setPluginInfoExtension` for `plugin_info_ext`

classmethod get()
Actually create an instance

classmethod setBehaviour(*list_of_pmd*)
Set the functionalities handled by the plugin manager by giving a list of `PluginManager` decorators.

This function shouldn't be called several time in a same process, but if it is only the first call will have an effect.

It also has an effect only if called before the initialisation of the singleton.

In cases where the function is indeed going to change anything the `True` value is return, in all other cases, the `False` value is returned.

CHAPTER 3

PluginInfo

Role

Encapsulate a plugin instance as well as some metadata.

API

class `yapsy.PluginInfo.`**`PluginInfo`**(*plugin_name*, *plugin_path*)

Representation of the most basic set of information related to a given plugin such as its name, author, description...

Any additional information can be stored ad retrieved in a PluginInfo, when this one is created with a `ConfigParser.ConfigParser` instance.

This typically means that when metadata is read from a text file (the original way for yapsy to describe plugins), all info that is not part of the basic variables (name, path, version etc), can still be accessed though the `details` member variables that behaves like Python's `ConfigParser.ConfigParser`.

Warning: the instance associated with the `details` member variable is never copied and used to store all plugin infos. If you set it to a custom instance, it will be modified as soon as another member variale of the plugin info is changed. Alternatively, if you change the instance "outside" the plugin info, it will also change the plugin info.

Ctor Arguments:

plugin_name is a simple string describing the name of

the plugin.

plugin_path describe the location where the plugin can be

found.

Warning: The `path` attribute is the full path to the plugin if it is organised as a directory or the full path to a file without the `.py` extension if the plugin is defined by a simple file. In the later case, the actual plugin is reached via `plugin_info.path+'.py'`.

author

category
DEPRECATED (>1.9): Mimic former behaviour when what is noz the first category was considered as the only one the plugin belonged to.

copyright

description

details

is_activated
Return the activated state of the plugin object. Makes it possible to define a property.

name

path

setVersion(*vstring*)
Set the version of the plugin.

Used by subclasses to provide different handling of the version number.

version

website

CHAPTER 4

Built-in Extensions

The followig ready-to-use classes give you this exact extra functionality you need for your plugin manager:

VersionedPluginManager

Role

Defines the basic interface for a plugin manager that also keeps track of versions of plugins

API

class `yapsy.VersionedPluginManager.`**`VersionedPluginInfo`**(*plugin_name*, *plugin_path*)

Gather some info about a plugin such as its name, author, description...

`setVersion`(*vstring*)

class `yapsy.VersionedPluginManager.`**`VersionedPluginManager`**(*decorated_manager=None*, *categories_filter={'Default': <class 'yapsy.IPlugin.IPlugin'>}*, *directories_list=None*, *plugin_info_ext='yapsy-plugin'*)

Handle plugin versioning by making sure that when several versions are present for a same plugin, only the latest version is manipulated via the standard methods (eg for activation and deactivation)

More precisely, for operations that must be applied on a single named plugin at a time (`getPluginByName`, `activatePluginByName`, `deactivatePluginByName` etc) the targetted plugin will always be the one with the latest version.

Note: The older versions of a given plugin are still reachable via the `getPluginsOfCategoryFromAttic` method.

getLatestPluginsOfCategory(*category_name*)
DEPRECATED(>1.8): Please consider using getPluginsOfCategory instead.

Return the list of all plugins belonging to a category.

getPluginsOfCategoryFromAttic(*categ*)
Access the older version of plugins for which only the latest version is available through standard methods.

loadPlugins(*callback=None*, *callback_after=None*)
Load the candidate plugins that have been identified through a previous call to locatePlugins.

In addition to the baseclass functionality, this subclass also needs to find the latest version of each plugin.

setCategoriesFilter(*categories_filter*)
Set the categories of plugins to be looked for as well as the way to recognise them.

Note: will also reset the attic toa void inconsistencies.

ConfigurablePluginManager

Role

Defines plugin managers that can handle configuration files similar to the ini files manipulated by Python's ConfigParser module.

API

class `yapsy.ConfigurablePluginManager`.**ConfigurablePluginManager**(*configparser_instance=None*, *config_change_trigger=<function <lambda>>*, *decorated_manager=None*, *categories_filter=None*, *directories_list=None*, *plugin_info_ext='yapsy-plugin'*)

A plugin manager that also manages a configuration file.

The configuration file will be accessed through a `ConfigParser` derivated object. The file can be used for other purpose by the application using this plugin manager as it will only add a new specific section `[Plugin Management]` for itself and also new sections for some plugins that will start with `[Plugin:...]` (only the plugins that explicitly requires to save configuration options will have this kind of section).

Warning: when giving/building the list of plugins to activate by default, there must not be any space in the list (neither in the names nor in between)

The `config_change_trigger` argument can be used to set a specific method to call when the configuration is altered. This will let the client application manage the way they want the configuration to be updated (e.g. write on file at each change or at precise time intervalls or whatever....)

> **Warning:** when no `config_change_trigger` is given and if the provided `configparser_instance` doesn't handle it implicitely, recording the changes persistently (ie writing on the config file) won't happen.

CONFIG_SECTION_NAME = 'Plugin Management'

activatePluginByName(*plugin_name*, *category_name='Default'*, *save_state=True*)
Activate a plugin, , and remember it (in the config file).

If you want the plugin to benefit from the configuration utility defined by this manager, it is crucial to use this method to activate a plugin and not call the plugin object's `activate` method. In fact, this method will also "decorate" the plugin object so that it can use this class's methods to register its own options.

By default, the plugin's activation is registered in the config file but if you d'ont want this set the 'save_state' argument to False.

deactivatePluginByName(*plugin_name*, *category_name='Default'*, *save_state=True*)
Deactivate a plugin, and remember it (in the config file).

By default, the plugin's deactivation is registered in the config file but if you d'ont want this set the `save_state` argument to False.

hasOptionFromPlugin(*category_name*, *plugin_name*, *option_name*)
To be called from a plugin object, return True if the option has already been registered.

loadPlugins(*callback=None*, *callback_after=None*)
Walk through the plugins' places and look for plugins. Then for each plugin candidate look for its category, load it and stores it in the appropriate slot of the `category_mapping`.

readOptionFromPlugin(*category_name*, *plugin_name*, *option_name*)
To be called from a plugin object, read a given option in the name of a given plugin.

registerOptionFromPlugin(*category_name*, *plugin_name*, *option_name*, *option_value*)
To be called from a plugin object, register a given option in the name of a given plugin.

setConfigParser(*configparser_instance*, *config_change_trigger*)
Set the ConfigParser instance.

AutoInstallPluginManager

Role

Defines plugin managers that can handle the installation of plugin files into the right place. Then the end-user does not have to browse to the plugin directory to install them.

API

class yapsy.AutoInstallPluginManager.**AutoInstallPluginManager**(*plugin_install_dir=None*, *decorated_manager=None*, *categories_filter=None*, *directories_list=None*, *plugin_info_ext='yapsy-plugin'*)

A plugin manager that also manages the installation of the plugin files into the appropriate directory.

Ctor Arguments:

`plugin_install_dir` The directory where new plugins to be installed will be copied.

> **Warning:** If `plugin_install_dir` does not correspond to an element of the `directories_list`, it is appended to the later.

getInstallDir()
Return the directory where new plugins should be installed.

install(*directory*, *plugin_info_filename*)
Giving the plugin's info file (e.g. `myplugin.yapsy-plugin`), and the directory where it is located, get all the files that define the plugin and copy them into the correct directory.

Return `True` if the installation is a success, `False` if it is a failure.

installFromZIP(*plugin_ZIP_filename*)
Giving the plugin's zip file (e.g. `myplugin.zip`), check that their is a valid info file in it and correct all the plugin files into the correct directory.

> **Warning:** Only available for python 2.6 and later.

Return `True` if the installation is a success, `False` if it is a failure.

setInstallDir(*plugin_install_dir*)
Set the directory where to install new plugins.

FilteredPluginManager

Role

Defines the basic mechanisms to have a plugin manager filter the available list of plugins after locating them and before loading them.

One use fo this would be to prevent untrusted plugins from entering the system.

To use it properly you must reimplement or monkey patch the `IsPluginOk` method, as in the following example:

```
# define a plugin manager (with you prefered options)
pm = PluginManager(...)
# decorate it with the Filtering mechanics
```

```
pm = FilteredPluginManager(pm)
# define a custom predicate that filters out plugins without descriptions
pm.isPluginOk = lambda x: x.description!=""
```

API

class yapsy.FilteredPluginManager.**FilteredPluginManager**(*decorated_manager=None*, *categories_filter=None*, *directories_list=None*, *plugin_info_ext='yapsy-plugin'*)

Base class for decorators which filter the plugins list before they are loaded.

appendPluginCandidate(*pluginTuple*)
Add a new candidate.

filterPlugins()
Go through the currently available candidates, and and either leaves them, or moves them into the list of rejected Plugins.

Can be overridden if overriding `isPluginOk` sentinel is not powerful enough.

getRejectedPlugins()
Return the list of rejected plugins.

isPluginOk(*info*)
Sentinel function to detect if a plugin should be filtered.

`info` is an instance of a `PluginInfo` and this method is expected to return True if the corresponding plugin can be accepted, and False if it must be filtered out.

Subclasses should override this function and return false for any plugin which they do not want to be loadable.

locatePlugins()
locate and filter plugins.

rejectPluginCandidate(*pluginTuple*)
Move a plugin from the candidates list to the rejected List.

removePluginCandidate(*pluginTuple*)
Remove a plugin from the list of candidates.

unrejectPluginCandidate(*pluginTuple*)
Move a plugin from the rejected list to into the candidates list.

MultiprocessPluginManager

Role

Defines a plugin manager that runs all plugins in separate process linked by pipes.

API

class yapsy.MultiprocessPluginManager.**MultiprocessPluginManager**(*categories_filter=None*, *directories_list=None*, *plugin_info_ext=None*, *plugin_locator=None*)

Subclass of the PluginManager that runs each plugin in a different process

instanciateElement(*element*)

This method instanciate each plugin in a new process and link it to the parent with a pipe.

In the parent process context, the plugin's class is replaced by the MultiprocessPluginProxy class that hold the information about the child process and the pipe to communicate with it. See *IMultiprocessChildPlugin*

The following item offer customization for the way plugins are described and detected:

PluginFileLocator

Role

The PluginFileLocator locates plugins when they are accessible via the filesystem.

It's default behaviour is to look for text files with the '.yapsy-plugin' extensions and to read the plugin's decription in them.

Customization

The behaviour of a PluginFileLocator can be customized by instanciating it with a specific 'analyzer'.

Two analyzers are already implemented and provided here:

PluginFileAnalyzerWithInfoFile

the default 'analyzer' that looks for plugin 'info files' as text file with a predefined extension. This implements the way yapsy looks for plugin since version 1.

PluginFileAnalyzerMathingRegex

look for files matching a regex and considers them as being the plugin itself.

All analyzers must enforce the

It enforces the plugin locator policy as defined by IPluginLocator and used by PluginManager.

info_ext

expects a plugin to be discovered through its *plugin info file*. User just needs to provide an extension (without '.') to look for *plugin_info_file*.

regexp

looks for file matching the given regular pattern expression. User just needs to provide the regular pattern expression.

All analyzers must enforce the policy represented by the IPluginFileAnalyzer interface.

API

class yapsy.PluginFileLocator.**IPluginFileAnalyzer**(*name*)
Define the methods expected by PluginFileLocator for its 'analyzer'.

getInfosDictFromPlugin(*dirpath*, *filename*)
Returns the extracted plugin informations as a dictionary. This function ensures that "name" and "path" are provided.

dirpath is the full path to the directory where the plugin file is

filename is the name (ie the basename) of the plugin file.

If *callback* function has not been provided for this strategy, we use the filename alone to extract minimal informations.

isValidPlugin(*filename*)
Check if the resource found at filename is a valid plugin.

class yapsy.PluginFileLocator.**PluginFileAnalyzerMathingRegex**(*name*, *regexp*)
An analyzer that targets plugins decribed by files whose name match a given regex.

getInfosDictFromPlugin(*dirpath*, *filename*)
Returns the extracted plugin informations as a dictionary. This function ensures that "name" and "path" are provided.

isValidPlugin(*filename*)
Checks if the given filename is a valid plugin for this Strategy

class yapsy.PluginFileLocator.**PluginFileAnalyzerWithInfoFile**(*name*, *extensions='yapsy-plugin'*)
Consider plugins described by a textual description file.

A plugin is expected to be described by a text file ('ini' format) with a specific extension (.yapsy-plugin by default).

This file must contain at least the following information:

```
[Core]
Name = name of the module
Module = relative_path/to/python_file_or_directory
```

Optionnally the description file may also contain the following section (in addition to the above one):

```
[Documentation]
Author = Author Name
Version = Major.minor
Website = url_for_plugin
Description = A simple one-sentence description
```

Ctor Arguments:

name name of the analyzer.

extensions the expected extensions for the plugin info file. May be a string or a tuple of strings if several extensions are expected.

getInfosDictFromPlugin(*dirpath*, *filename*)
Returns the extracted plugin informations as a dictionary. This function ensures that "name" and "path" are provided.

If *callback* function has not been provided for this strategy, we use the filename alone to extract minimal informations.

getPluginNameAndModuleFromStream(*infoFileObject*, *candidate_infofile=None*)
Extract the name and module of a plugin from the content of the info file that describes it and which is stored in `infoFileObject`.

Note: Prefer using `_extractCorePluginInfo` instead, whenever possible...

Warning: `infoFileObject` must be a file-like object: either an opened file for instance or a string buffer wrapped in a StringIO instance as another example.

Note: `candidate_infofile` must be provided whenever possible to get better error messages.

Return a 3-uple with the name of the plugin, its module and the config_parser used to gather the core data *in a tuple*, if the required info could be localised, else return `(None,None,None)`.

Note: This is supposed to be used internally by subclasses and decorators.

---

isValidPlugin(*filename*)
Check if it is a valid plugin based on the given plugin info file extension(s). If several extensions are provided, the first matching will cause the function to exit successfully.

setPluginInfoExtension(*extensions*)
Set the extension that will identify a plugin info file.

extensions May be a string or a tuple of strings if several extensions are expected.

class `yapsy.PluginFileLocator.`**PluginFileLocator**(*analyzers=None*, *plugin_info_cls=<class 'yapsy.PluginInfo.PluginInfo'>*)
Locates plugins on the file system using a set of analyzers to determine what files actually corresponds to plugins.

If more than one analyzer is being used, the first that will discover a new plugin will avoid other strategies to find it too.

By default each directory set as a "plugin place" is scanned recursively. You can change that by a call to `disableRecursiveScan`.

appendAnalyzer(*analyzer*)
Append an analyzer to the existing list.

disableRecursiveScan()
Disable recursive scan of the directories given as plugin places.

gatherCorePluginInfo(*directory*, *filename*)
Return a `PluginInfo` as well as the `ConfigParser` used to build it.

If filename is a valid plugin discovered by any of the known strategy in use. Returns None,None otherwise.

getPluginNameAndModuleFromStream(*infoFileObject*, *candidate_infofile=None*)

locatePlugins()
Walk through the plugins' places and look for plugins.

Return the candidates and number of plugins found.

removeAllAnalyzer()
Remove all analyzers.

removeAnalyzers(*name*)
Removes analyzers of a given name.

setAnalyzers(*analyzers*)
Sets a new set of analyzers.

> **Warning:** the new analyzers won't be aware of the plugin info class that may have been set via a previous call to `setPluginInfoClass`.

setPluginInfoClass(*picls*, *name=None*)
Set the class that holds PluginInfo. The class should inherit from `PluginInfo`.

If name is given, then the class will be used only by the corresponding analyzer.

If name is None, the class will be set for all analyzers.

setPluginInfoExtension(*ext*)
DEPRECATED(>1.9): for backward compatibility. Directly configure the IPluginLocator instance instead !

This will only work if the strategy "info_ext" is active for locating plugins.

setPluginPlaces(*directories_list*)
Set the list of directories where to look for plugin places.

updatePluginPlaces(*directories_list*)
Updates the list of directories where to look for plugin places.

If you want to build your own extensions, have a look at the following interfaces:

IPluginLocator

Role

`IPluginLocator` defines the basic interface expected by a `PluginManager` to be able to locate plugins and get basic info about each discovered plugin (name, version etc).

API

class `yapsy.IPluginLocator.`**IPluginLocator**
Plugin Locator interface with some methods already implemented to manage the awkward backward compatible stuff.

gatherCorePluginInfo(*directory*, *filename*)
Return a `PluginInfo` as well as the `ConfigParser` used to build it.

If filename is a valid plugin discovered by any of the known strategy in use. Returns None,None otherwise.

getPluginInfoClass()
DEPRECATED(>1.9): kept for backward compatibility with existing PluginManager child classes.

Get the class that holds PluginInfo.

getPluginNameAndModuleFromStream(*fileobj*)
DEPRECATED(>1.9): kept for backward compatibility with existing PluginManager child classes.

Return a 3-uple with the name of the plugin, its module and the config_parser used to gather the core data *in a tuple*, if the required info could be localised, else return `(None,None,None)`.

locatePlugins()
Walk through the plugins' places and look for plugins.

Return the discovered plugins as a list of `(candidate_infofile_path, candidate_file_path,plugin_info_instance)` and their number.

setPluginInfoClass(*picls*, *names=None*)
DEPRECATED(>1.9): kept for backward compatibility with existing PluginManager child classes.

Set the class that holds PluginInfo. The class should inherit from `PluginInfo`.

setPluginPlaces(*directories_list*)
DEPRECATED(>1.9): kept for backward compatibility with existing PluginManager child classes.

Set the list of directories where to look for plugin places.

updatePluginPlaces(*directories_list*)
DEPRECATED(>1.9): kept for backward compatibility with existing PluginManager child classes.

Updates the list of directories where to look for plugin places.

PluginManagerDecorator

Role

Provide an easy way to build a chain of decorators extending the functionalities of the default plugin manager, when it comes to activating, deactivating or looking into loaded plugins.

The `PluginManagerDecorator` is the base class to be inherited by each element of the chain of decorator.

Warning: If you want to customise the way the plugins are detected and loaded, you should not try to do it by implementing a new `PluginManagerDecorator`. Instead, you'll have to reimplement the *PluginManager* itself. And if you do so by enforcing the `PluginManager` interface, just giving an instance of your new manager class to the `PluginManagerDecorator` should be transparent to the "stantard" decorators.

API

class `yapsy.PluginManagerDecorator.`**PluginManagerDecorator**(*decorated_object=None*, *categories_filter=None*, *directories_list=None*, *plugin_info_ext='yapsy-plugin'*)
Add several responsibilities to a plugin manager object in a more flexible way than by mere subclassing. This is indeed an implementation of the Decorator Design Patterns.

There is also an additional mechanism that allows for the automatic creation of the object to be decorated when this object is an instance of PluginManager (and not an instance of its subclasses). This way we can keep the plugin managers creation simple when the user don't want to mix a lot of 'enhancements' on the base class.

About the __init__:

Mimics the PluginManager's __init__ method and wraps an instance of this class into this decorator class.

- *If the decorated_object is not specified*, then we use the PluginManager class to create the 'base' manager, and to do so we will use the arguments: `categories_filter`, `directories_list`, and `plugin_info_ext` or their default value if they are not given.
- *If the decorated object is given*, these last arguments are simply **ignored** !

All classes (and especially subclasses of this one) that want to be a decorator must accept the decorated manager as an object passed to the init function under the exact keyword `decorated_object`.

collectPlugins()
This function will usually be a shortcut to successively call `self.locatePlugins` and then `self.loadPlugins` which are very likely to be redefined in each new decorator.

So in order for this to keep on being a "shortcut" and not a real pain, I'm redefining it here.

If you want to isolate your plugins in separate processes with the `MultiprocessPluginManager`, you should look at the following classes too:

IMultiprocessChildPlugin

Role

Defines the basic interfaces for multiprocessed plugins.

Extensibility

In your own software, you'll probably want to build derived classes of the `IMultiprocessChildPlugin` class as it is a mere interface with no specific functionality.

Your software's plugins should then inherit your very own plugin class (itself derived from `IMultiprocessChildPlugin`).

Override the run method to include your code. Use the self.parent_pipe to send and receive data with the parent process or create your own communication mecanism.

Where and how to code these plugins is explained in the section about the *PluginManager*.

API

class `yapsy.IMultiprocessChildPlugin.`**IMultiprocessChildPlugin**(*parent_pipe*)
Base class for multiprocessed plugin.

run()
Override this method in your implementation

MultiprocessPluginProxy

Role

The `MultiprocessPluginProxy` is instanciated by the MultiprocessPluginManager to replace the real implementation that is run in a different process.

You cannot access your plugin directly from the parent process. You should use the child_pipe to communicate with your plugin. The *MultiprocessPluginProxy'* role is to keep reference of the communication pipe to the child process as well as the process informations.

API

class yapsy.MultiprocessPluginProxy.**MultiprocessPluginProxy**

This class contains two members that are initialized by the *MultiprocessPluginManager*.

self.proc is a reference that holds the multiprocessing.Process instance of the child process.

self.child_pipe is a reference that holds the multiprocessing.Pipe instance to communicate with the child.

CHAPTER 5

General advices and troubleshooting

Getting code samples

Yapsy is used enough for your favorite search provider to have good chances of finding some examples of yapsy being used in the wild.

However if you wonder how a specific functionality can be used, you can also look at the corresponding unit test (in the test folder packaged with yapsy's sources).

Use the logging system

Yapsy uses Python's standard `logging` module to record most important events and especially plugin loading failures.

When developping an application based on yapsy, you'll benefit from looking at the 'debug' level logs, which can easily be done from your application code with the following snippet:

```
import logging
logging.basicConfig(level=logging.DEBUG)
```

Also, please note that yapsy uses a named logger for all its logs, so that you can selectively activage debug logs for yapsy with the following snippet:

```
import logging
logging.getLogger('yapsy').setLevel(logging.DEBUG)
```

Categorization by inheritance caveat

If your application defines various categories of plugins with the yapsy's built-in mechanism for that, please keep in mind the following facts:

- a plugin instance is attributed to a given category by looking if it is an instance, *even via a subclass*, of the class associated to this category;
- a plugin may be attributed to several categories.

Considering this, and if you consider using several categories, you should consider the following tips:

- **don't associate any category to ''IPlugin''** (unless you want all plugins to be attributed to the corresponding category)
- **design a specific subclass** of `IPlugin` for each category
- if you want to regroup plugins of some categories into a common category: do this by attributing a subclass of `IPlugin` to the common category and attribute to the other categories specific subclasses to this intermediate mother class so that **the plugin class inheritance hierarchy reflects the hierarchy between categories** (and if you want something more complex that a hierarchy, you can consider using mixins).

Plugin class detection caveat

There must be **only one plugin defined per module**. This means that you can't have two plugin description files pointing at the same module for instance.

Because of the "categorization by inheritance" system, you **musn't directly import the subclass** of `IPlugin` in the main plugin file, instead import its containing module and make your plugin class inherit from `ContainingModule.SpecificPluginClass` as in the following example.

The following code won't work (the class `MyBasePluginClass` will be detected as the plugin's implementation instead of `MyPlugin`):

```
from myapp.plugintypes import MyBasePluginClass

class MyPlugin(MyBasePluginClass):
    pass
```

Instead you should do the following:

```
import myapp.plugintypes as plugintypes

class MyPlugin(plugintypes.MyBasePluginClass):
    pass
```

Plugin packaging

When packaging plugins in a distutils installer or as parts of an application (like for instance with *py2exe*), you may want to take care about the following points:

- when you set specific directories where to look for plugins with a hardcoded path, be very carefully about the way you write these paths because depending on the cases **using ‘‘__file__‘‘ or relative paths may be unreliable**. For instance with py2exe, you may want to follow the tips from the Where Am I FAQ.
- you’d should either **package the plugins as plain Python modules or data files** (if you want to consider you application as the only module), either using the dedicated *setup* argument for *py2exe* or using distutils’ *MANIFEST.in*
- if you do package the plugins as data files, **make sure that their dependencies are correctly indicated as dependencies of your package** (or packaged with you application if you use *py2exe*).

See also a more detailed example for py2exe on Simon on Tech’s Using python plugin scripts with py2exe.

Code conventions

If you intend to modify yapsy’s sources and to contribute patches back, please respect the following conventions:

- CamelCase (upper camel case) for class names and functions
- camelCase (lower camel case) for methods
- UPPERCASE for global variables (with a few exceptions)
- tabulations are used for indentation (and not spaces !)
- unit-test each new functionality

On this page

CHAPTER 6

Overview

Yapsy's main purpose is to offer a way to easily design a plugin system in Python, and motivated by the fact that many other Python plugin system are either too complicated for a basic use or depend on a lot of libraries. Yapsy only depends on Python's standard library.

Yapsy basically defines two core classes:

- a fully functional though very simple `PluginManager` class
- an interface `IPlugin` which defines the interface of plugin instances handled by the `PluginManager`

CHAPTER 7

Getting started

The basic classes defined by **Yapsy** should work "as is" and enable you to load and activate your plugins. So that the following code should get you a fully working plugin management system:

```
from yapsy.PluginManager import PluginManager

# Build the manager
simplePluginManager = PluginManager()
# Tell it the default place(s) where to find plugins
simplePluginManager.setPluginPlaces(["path/to/myplugins"])
# Load all plugins
simplePluginManager.collectPlugins()

# Activate all loaded plugins
for pluginInfo in simplePluginManager.getAllPlugins():
   simplePluginManager.activatePluginByName(pluginInfo.name)
```

Note: The `plugin_info` object (typically an instance of `IPlugin`) plays as *the entry point of each plugin*. That's also where **Yapsy** ceases to guide you: it's up to you to define what your plugins can do and how you want to talk to them ! Talking to your plugin will then look very much like the following:

```
# Trigger 'some action' from the loaded plugins
for pluginInfo in simplePluginManager.getAllPlugins():
   pluginInfo.plugin_object.doSomething(...)
```

yapsy.**NormalizePluginNameForModuleName**(*pluginName*)
Normalize a plugin name into a safer name for a module name.

Note: may do a little more modifications than strictly necessary and is not optimized for speed.

yapsy.**PLUGIN_NAME_FORBIDEN_STRING** = **';;'**

Warning: This string (';;' by default) is forbidden in plugin names, and will be usable to describe lists of plugins for instance (see *ConfigurablePluginManager*)

CHAPTER 8

Make it your own

For applications that require the plugins and their managers to be more sophisticated, several techniques make such enhancement easy. The following sections detail the most frequent needs for extensions and what you can do about it.

More sophisticated plugin classes

You can define a plugin class with a richer interface than `IPlugin`, so long as it inherits from IPlugin, it should work the same. The only thing you need to know is that the plugin instance is accessible via the `PluginInfo` instance from its `PluginInfo.plugin_object`.

It is also possible to define a wider variety of plugins, by defining as much subclasses of IPlugin. But in such a case you have to inform the manager about that before collecting plugins:

```
# Build the manager
simplePluginManager = PluginManager()
# Tell it the default place(s) where to find plugins
simplePluginManager.setPluginPlaces(["path/to/myplugins"])
# Define the various categories corresponding to the different
# kinds of plugins you have defined
simplePluginManager.setCategoriesFilter({
   "Playback" : IPlaybackPlugin,
   "SongInfo" : ISongInfoPlugin,
   "Visualization" : IVisualisation,
   })
```

Note: Communicating with the plugins belonging to a given category might then be achieved with some code looking like the following:

```
# Trigger 'some action' from the "Visualization" plugins
for pluginInfo in simplePluginManager.getPluginsOfCategory("Visualization"):
   pluginInfo.plugin_object.doSomething(...)
```

Enhance the plugin manager's interface

To make the plugin manager more helpful to the other components of an application, you should consider decorating it.

Actually a "template" for such decoration is provided as *PluginManagerDecorator*, which must be inherited in order to implement the right decorator for your application.

Such decorators can be chained, so that you can take advantage of the ready-made decorators such as:

ConfigurablePluginManager

> Implements a `PluginManager` that uses a configuration file to save the plugins to be activated by default and also grants access to this file to the plugins.

AutoInstallPluginManager

> Automatically copy the plugin files to the right plugin directory.

A full list of pre-implemented decorators is available at *Built-in Extensions*.

Modify plugin descriptions and detections

By default, plugins are described by a text file called the plugin "info file" expected to have a ".yapsy-plugin" extension.

You may want to use another way to describe and detect your application's plugin and happily yapsy (since version 1.10) makes it possible to provide the `PluginManager` with a custom strategy for plugin detection.

See *IPluginLocator* for the required interface of such strategies and *PluginFileLocator* for a working example of such a detection strategy.

Modify the way plugins are loaded

To tweak the plugin loading phase it is highly advised to re-implement your own manager class.

The nice thing is, if your new manager inherits `PluginManager`, then it will naturally fit as the start point of any decoration chain. You just have to provide an instance of this new manager to the first decorators, like in the following:

```
# build and configure a specific manager
baseManager = MyNewManager()
# start decorating this manager to add some more responsibilities
myFirstDecorator = AFirstPluginManagerDecorator(baseManager)
# add even more stuff
mySecondDecorator = ASecondPluginManagerDecorator(myFirstDecorator)
```

Note: Some decorators have been implemented that modify the way plugins are loaded, this is however not the easiest way to do it and it makes it harder to build a chain of decoration that would include these decorators. Among those are *VersionedPluginManager* and *FilteredPluginManager*

CHAPTER 9

Showcase and tutorials

Yapsy 's development has been originally motivated by the MathBench project but it is now used in other (more advanced) projects like:

- peppy : "an XEmacs-like editor in Python. Eventually. "
- MysteryMachine : "an application for writing freeform games."
- Aranduka : "A simple e-book manager and reader"
- err : "a plugin based chatbot"
- nikola : "a Static Site and Blog Generator"

Nowadays, the development is clearly motivated by such external projects and the enthusiast developpers who use the library.

If you're interested in using yapsy, feel free to look into the following links:

- *General advices and troubleshooting*
- A minimal example on stackoverflow
- Making your app modular: Yapsy (applied to Qt apps)
- Python plugins with yapsy (applied to GTK apps)

CHAPTER 10

Development

Contributing or forking ?

You're always welcome if you suggest any kind of enhancements, any new decorators or any new pluginmanager. Even more if there is some code coming with it though this is absolutely not compulsory.

It is also really fine to *fork* the code ! In the past, some people found **Yapsy** just good enough to be used as a "code base" for their own plugin system, which they evolved in a more or less incompatible way with the "original" **Yapsy**, if you think about it, with such a small library this is actually a clever thing to do.

In any case, please remember that just providing some feedback on where you're using **Yapsy** (original or forked) and how it is useful to you, is in itself a appreciable contribution :)

License

The work is placed under the simplified BSD license in order to make it as easy as possible to be reused in other projects.

Please note that the icon is not under the same license but under the Creative Common Attribution-ShareAlike license.

Forge

The project is hosted by Sourceforge where you can access the code, documentation and a tracker to share your feedback and ask for support.

Any suggestion and help are much welcome !

Yapsy is also tested on the continous integration service TravisCI:

A few alternative sites are available:

- Yapsy's sources are mirrored on GitHub.

- To use pip for a development install you can do something like:

```
pip install -e "git+https://github.com/tibonihoo/yapsy.git#egg=yapsy&
↪subdirectory=package"
pip install -e "hg+http://hg.code.sf.net/p/yapsy/code#egg=yapsy&
↪subdirectory=package"
```

- A development version of the documentation is available on ReadTheDoc.

References

Other Python plugin systems already existed before **Yapsy** and some have appeared after that. **Yapsy**'s creation is by no mean a sign that these others plugin systems sucks :) It is just the results of me being slighlty lazy and as I had already a good idea of how a simple plugin system should look like, I wanted to implement my own[1].

- setuptools seems to be designed to allow applications to have a plugin system.
- Sprinkles seems to be also quite lightweight and simple but just maybe too far away from the design I had in mind.
- PlugBoard is certainly quite good also but too complex for me. It also depends on zope which considered what I want to do here is way too much.
- Marty Alchin's simple plugin framework is a quite interesting description of a plugin architecture with code snippets as illustrations.
- stevedor looks quite promising and actually seems to make setuptools relevant to build plugin systems.
- You can look up more example on a stackoverflow's discution about minimal plugin systems in Python

[1] All the more because it seems that my modest design ideas slightly differ from what has been done in other libraries.

CHAPTER 11

Indices and tables

- genindex
- modindex
- search

Python Module Index

y

Index

H

I

L

M

N

P

R

S

U

V

W

Y