Yapsy Documentation

Release 1.10.423

Thibauld Nion

February 07, 2017

Contents

A simple plugin system for Python applications

Quick links:

CHAPTER 1

IPlugin

1.1 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.

1.2 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.

1.3 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

CHAPTER 3

PluginInfo

CHAPTER 4

Built-in Extensions

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

4.1 VersionedPluginManager

4.2 ConfigurablePluginManager

4.3 AutoInstallPluginManager

4.3.1 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.

4.3.2 API

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

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

`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.

4.4 FilteredPluginManager

4.4.1 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!=""
```

4.4.2 API

class yapsy.FilteredPluginManager.**FilteredPluginManager**(*decorated_manager=None*, *categories_filter={'Default': <class 'yapsy.IPlugin.IPlugin'>}*, *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.

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

4.5 PluginFileLocator

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

4.6 IPluginLocator

4.6.1 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).

4.6.2 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.

4.7 PluginManagerDecorator

4.7.1 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.

4.7.2 API

class `yapsy.PluginManagerDecorator.`**PluginManagerDecorator** (*decorated_object=None, categories_filter={'Default': <class 'yapsy.IPlugin.IPlugin'>}, directories_list=['/home/docs/checkouts/readthedocs.org/u*
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.

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.

CHAPTER 5

General advices and troubleshooting

5.1 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).

5.2 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)
```

5.3 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).

5.4 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
```

5.5 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.

5.6 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.

8.1 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(...)
```

8.2 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.

8.3 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.

8.4 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

10.1 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 :)

10.2 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.

10.3 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:

- with Python-2.x:
- with Python-3.x:

And if you're looking for the development version of the documentation, it is continuously updated on ReadTheDoc.

Last but not least, Yapsy's sources are mirrored on GitHub.

10.4 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.
- Evan Fosmark's A simple event-driven plugin system in Python where "plugins are just functions that get registered through the use of a decorator".
- 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

y

Y