napari-plugin-engine

Holger Krekel & Talley Lambert

May 11, 2021

CONTENTS

`napari-plugin-engine` is a fork of pluggy modified by the napari team for use in napari.

While much of the original API described in the pluggy docs will still be valid here, there are definitely some breaking changes and different conventions here.

If you're just getting started, have a look at the *Usage Overview*

For details on specific classes in napari-plugin-engine, have a look at the *API Reference*

See also: documentation for writing a napari plugin.

CHAPTER

ONE

USAGE OVERVIEW

1.1 Create a plugin manager

A *PluginManager* is the main object that registers and organizes plugins. It is instantiated with a `project_name`:

```
plugin_manager = PluginManager('my_project')
```

All *hook specifications* and *hook implementations* must use the same `project_name` if they are to be recognized by this `plugin_manager` instance.

1.2 Add some hook specifications

You add hook specifications which outline the function signatures plugins may implement:

```
plugin_manager.add_hookspecs(some_class_or_module)
```

... where "`some_class_or_module`" is any namespace object (such as a class or module) that has some functions that have been decorated as hook specifications for `'my_project'` using a *HookSpecificationMarker* decorator.

```
# some_class_or_module.py

from napari_plugin_engine import HookSpecificationMarker

my_project_hook_specification = HookSpecificationMarker('my_project')

@my_project_hook_specification
def do_something(arg1: int, arg2: int): -> int:
    """Take two integers and return one integer."""
```

After calling *add_hookspecs()*, your `plugin_manager` instance will have a new *HookCaller* instance created under the `plugin_manager.hooks` namespace, for each hook specification discovered. In this case, there will be a new one at `plugin_manager.hooks.do_something`.

1.3 (Plugins) write hook implementations

Plugins may then provide *implementations* for your hook specifications, by creating classes or modules that contain functions that are decorated with an instance of a *HookImplementationMarker* that has been created using the *same* project name (in this example: `'my_project'`)

```
# some_plugin.py

from napari_plugin_engine import HookImplementationMarker

my_project_hook_implementation = HookImplementationMarker('my_project')

@my_project_hook_implementation
def do_something(arg1, arg2):
    return arg1 + arg2
```

1.4 Register plugins

You may directly *register* these modules with the `plugin_manager` ...

```
import some_plugin

plugin_manager.register(some_plugin)
```

1.4.1 Autodiscover plugins in the environment

However, it is more often the case that you will want to *discover* plugins in your environment. `napari-plugin-engine` provides two ways to discover plugins via two different conventions:

1. Using package metadata: looking for distributions that declare a specific entry_point in their `setup.py` file.

2. Using naming convention: looking for modules that begin with a specific prefix.

You can look for either or both, in single call to *discover()*, which will import any modules or entry_points that follow one of the aforementioned conventions, and search them for functions decorated with the appropriate *HookImplementationMarker* (as shown above in `some_plugin.py`)

```
plugin_manager.discover(
    entry_point='my_project.plugin', prefix='my_project_'
)
```

1.5 Use (call) the plugin implementations

Your *HookCaller* should now be populated with any of the implementations found in plugins, as *HookImplementation* objects on the *HookCaller*.

```
# show all implementations for do_something
plugin_manager.hooks.do_something.get_hookimpls()
```

Finally, you can call some or all of the plugin implementation functions by directly calling the *HookCaller* object:

```
result = plugin_manager.hooks.do_something(arg1=2, arg2=7)

# assuming only some_plugin.py from above is registered:
print(result)  # [9]
```

By default, *all* plugin implementations are called, and all non-`None` results are returned in a list. However, this is configurable and depends on how the `@my_project_hook_specification` was used, and how the *`HookCaller`* was called

1.6 How the `plugin_name` is chosen

1. If plugin discovery via entry_points is used

(e.g. `plugin_manager.discover(entry_point='app.plugin')`), then plugins will be named using the name of the entry_point provided by each plugin. Note, a single package may provide multiple plugins via entry points. For example, if a package had the following `entry_points` declared in their `setup.py` file:

```
# setup.py

setup(
...
entry_points={'app.plugin': ['plugin1 = module_a', 'plugin2 = module_b']},
...
)
```

... then `manager.discover(entry_point='app.plugin')` would register two plugins, named `"plugin1"` (which would inspect `module_a` for implementations) and `"plugin2"` (which would inspect `module_b` for implementations).

2. If plugin discovery via naming convention is used

(e.g. `plugin_manager.discover(prefix='app_')`), then...

> **2a. If a `dist-info` folder is found for the module**
>
> Then the plugin will be named using the Name key in the distribution `METADATA` file if one is available. Usually, this will come from having a `setup(name="distname", ...)` entry in a `setup.py` file. See Core metadata specifications and PEP 566 for details.
>
> **2a. If no distribution metadata can be located**
>
> The the plugin will be named using the name of the module itself.

3. If a plugin is directly registered

(e.g. `plugin_manager.register(object, name)`), then if a `name` argument is provided to the *`PluginManager.register()`* method, it will be used as the `plugin_name`, otherwise, the string form of the object is used: `str(id(object))`

CHAPTER

TWO

API REFERENCE

PluginManager	Core class which manages registration of plugin objects and hook calls.
HookCaller	The primary hook-calling object.
HookResult	A class to store/modify results from a *_multicall()* hook loop.
HookImplementation	A class to encapsulate hook implementations.
HookSpecification	A class to encapsulate hook specifications.
HookSpecificationMarker	Decorator helper class for marking functions as hook specifications.
HookImplementationMarker	Decorator helper class for marking functions as hook implementations.

2.1 PluginManager

class napari_plugin_engine.**PluginManager**(*project_name*, *, *discover_entry_point=None*, *discover_prefix=None*, *discover_path=None*)

Core class which manages registration of plugin objects and hook calls.

You can register new hooks by calling *add_hookspecs()*. You can register plugin objects (which contain hooks) by calling *register()*. The `PluginManager` is initialized with a `project_name` that is used when discovering *hook specifications* and *hook implementations*.

For debugging purposes you may call *PluginManager.enable_tracing()* which will subsequently send debug information to the trace helper.

Parameters

- **project_name** (*str*) – The name of the host project. All *HookImplementationMarker* and *HookSpecificationMarker* instances must be created using the same `project_name` to be detected by this plugin manager.
- **discover_entry_point** (*str, optional*) – The default entry_point group to search when discovering plugins with *PluginManager.discover()*, by default None
- **discover_prefix** (*str, optional*) – The default module prefix to use when discovering plugins with *PluginManager.discover()*, by default None
- **discover_path** (*str or list of str, optional*) – A path or paths to include when discovering plugins with *PluginManager.discover()*, by default None

Examples

```
from napari_plugin_engine import PluginManager
import my_hookspecs

plugin_manager = PluginManager(
    'my_project',
    discover_entry_point='app.plugin',
    discover_prefix='app_',
)
plugin_manager.add_hookspecs(my_hookspecs)
plugin_manager.discover()

# hooks now live in plugin_manager.hook
# plugin dict is at plugin_manager.plugins
```

_ensure_plugin(*name_or_object*)
Return plugin object given a name or object. Or raise an exception.

Parameters **name_or_object** (*Any*) – Either a string (in which case it is interpreted as a plugin name), or a non-string object (in which case it is assumed to be a plugin module or class).

Returns The plugin object, if found.

Return type Any

Raises **KeyError** – If the plugin does not exist.

_hookexec(*caller*, *methods*, *kwargs*)
Returns a function that will call a set of hookipmls with a caller.

This function will be passed to `HookCaller` instances that are created during hookspec and plugin registration.

If *enable_tracing()* is used, it will set it's own wrapper function at self._inner_hookexec to enable tracing of hook calls.

Parameters

- **caller** (`HookCaller`) – The HookCaller instance that will call the HookImplementations.
- **methods** (*List[HookImplementation]*) – A list of *HookImplementation* objects whose functions will be called during the hook call loop.
- **kwargs** (*dict*) – Keyword arguments to pass when calling the `HookImplementation`.

Returns The result object produced by the multicall loop.

Return type *HookResult*

_plugin2hookcallers: Dict[Any, List[HookCaller]] = None
mapping of `plugin` (object) → list of *HookCaller*

HookCaller s get added in *register()*

Type dict

_register_dict(*dct*, *name=None*, ***kwargs*)
Register a dict as a mapping of method name -> method.

Parameters

- **dct** (*Dict[str, Callable]*) – Mapping of method name to method.
- **name** (*Optional[str], optional*) – The plugin_name to assign to this object, by default None

Returns canonical plugin name, or `None` if the name is blocked from registering.

Return type str or None

_verify_hook(*hook_caller*, *hookimpl*)
Check validity of a `hookimpl`

Parameters

- **hook_caller** (`HookCaller`) – A *HookCaller* instance.
- **hookimpl** (`HookImplementation`) – A *HookImplementation* instance, implementing the hook in `hook_caller`.

Raises

- ***PluginValidationError*** – If hook_caller is historic and the hookimpl is a hookwrapper.
- ***PluginValidationError*** – If there are any argument names in the `hookimpl` that are not in the `hook_caller.spec`.

Warns **Warning** – If the hookspec has `warn_on_impl` flag (usually a deprecation).

add_hookcall_monitoring(*before*, *after*)
Add before/after tracing functions for all hooks.

return an undo function which, when called, will remove the added tracers.

`before(hook_name, hook_impls, kwargs)` will be called ahead of all hook calls and receive a hookcaller instance, a list of HookImplementation instances and the keyword arguments for the hook call.

`after(outcome, hook_name, hook_impls, kwargs)` receives the same arguments as `before` but also a `napari_plugin_engine.callers._Result` object which represents the result of the overall hook call.

Return type `Callable`[[], `None`]

add_hookspecs(*namespace*)
Add new hook specifications defined in the given `namespace`.

Functions are recognized if they have been decorated accordingly.

check_pending()
Make sure all hooks have a specification, or are optional.

Raises ***PluginValidationError*** – If a hook implementation that was *not* marked as `optionalhook` has been registered for a non-existent hook specification.

discover(*path=None*, *entry_point=None*, *prefix=None*, *ignore_errors=True*)
Discover and load plugins.

Parameters

- **path** (*str, optional*) – If a string is provided, it is added to `sys.path` (and `self.discover_path`) before importing, and removed at the end.
- **entry_point** (*str, optional*) – An entry_point group to search for, by default `self.discover_entry_point` is used

- **prefix** (*str, optional*) – If `provided`, modules in the environment starting with `prefix` will be imported and searched for hook implementations by default `self.discover_prefix` is used
- **ignore_errors** (*bool, optional*) – If `True`, errors will be gathered and returned at the end. Otherwise, they will be raised immediately. by default True

Returns (count, errs) – The number of succefully loaded modules, and a list of errors that occurred (if `ignore_errors` was `True`)

Return type Tuple[int, List[*PluginError*]]

discovery_blocked()
A context manager that temporarily blocks discovery of new plugins.

Return type `Generator`

enable_tracing()
Enable tracing of hook calls and return an undo function.

get_errors(*plugin=<Empty.token: 0>*, *error_type=<Empty.token: 0>*)
Return a list of PluginErrors associated with `plugin`.

Parameters

- **plugin** (*Any*) – If provided, will restrict errors to those that were raised by `plugin`. If a string is provided, it will be interpreted as the name of the plugin, otherwise it is assumed to be the actual plugin object itself.
- **error_type** (`PluginError`) – If provided, will restrict errors to instances of `error_type`.

Return type `List[PluginError]`

get_hookcallers(*plugin*)
get all hook callers for the specified plugin.

Return type `Optional[List[HookCaller]]`

get_metadata(*plugin*, **values*)
Return metadata values for a given plugin

Parameters

- **plugin** (*Any*) – Either a string (in which case it is interpreted as a plugin name), or a non-string object (in which case it is assumed to be a plugin module or class).
- ***values** (*str*) – key(s) to lookup in the plugin object distribution metadata. At least one value must be supplied.

Raises

- **TypeError** – If no values are supplied.
- **KeyError** – If the plugin does not exist.

Return type `Union[str, Dict[str, Optional[str]], None]`

get_name(*plugin*)
Return name for registered plugin or `None` if not registered.

get_standard_metadata(*plugin*)
Return a standard metadata dict for `plugin`.

Parameters **plugin** (*Any*) – A plugin name or any object. If it is a plugin name, it *must* be a registered plugin.

Returns

metadata – A dicts with plugin metadata. The dict is guaranteed to have the following keys:

- **plugin_name**: The name of the plugin as registered
- **package**: The name of the package
- **version**: The version of the plugin package
- **summary**: A one-line summary of what the distribution does
- **author**: The author's name
- **email**: The author's (or maintainer's) e-mail address.
- **license**: The license covering the distribution
- **url**: The home page for the package, or dowload url if N/A.
- **hooks**: A list of hookspec names that this plugin implements.

Return type dict

Raises **KeyError** – If `plugin` is a string, but is not a registered plugin_name.

property hooks
An alias for PluginManager.hook

Return type `_HookRelay`

is_blocked(*plugin_name*)
Return `True` if the given plugin name is blocked.

Return type `bool`

is_registered(*obj*)
Return `True` if the plugin is already registered.

Return type `bool`

iter_available(*path=None*, *entry_point=None*, *prefix=None*)
Iterate over available plugins.

Parameters

- **path** (*str, optional*) – If a string is provided, it is added to `sys.path` (and `self.discover_path`) before importing, and removed at the end.
- **entry_point** (*str, optional*) – An entry_point group to search for, by default `self.discover_entry_point` is used
- **prefix** (*str, optional*) – If `provided`, modules in the environment starting with `prefix` will be imported and searched for hook implementations by default `self.discover_prefix` is used

:param See docstring of `iter_available_plugins()` for details.:

Return type `Generator[Tuple[str, str, Optional[str]], None, None]`

list_plugin_metadata()
Return list of standard metadata dicts for every registered plugin.

Returns **metadata** – A list of dicts with plugin metadata. Every dict in the list is guaranteed to have the following keys mentioned in *get_standard_metadata()*

Return type dict

plugins: Dict[str, Any] = None
mapping of `plugin_name` → `plugin` (object)

Plugins get added to this dict in *`register()`*

Type dict

prune()
Unregister modules that can no longer be imported.

Useful if pip uninstall has been run during the session.

register(*namespace*, *name=None*)
Register a plugin and return its canonical name or `None`.

Parameters

- **plugin** (*Any*) – The namespace (class, module, dict, etc...) to register
- **name** (*str, optional*) – Optional name for plugin, by default `get_canonical_name(plugin)`

Returns canonical plugin name, or `None` if the name is blocked from registering.

Return type str or None

Raises

- **TypeError** – If `namespace` is a string.
- **ValueError** – if the plugin `name` or `namespace` is already registered.

set_blocked(*plugin_name*, *blocked=True*)
Block registrations of `plugin_name`, unregister if registered.

Parameters

- **plugin_name** (*str*) – A plugin name to block.
- **blocked** (*bool, optional*) – Whether to block the plugin. If `False` will "unblock" `plugin_name`. by default True

unregister(*name_or_object*)
Unregister a plugin object or `plugin_name`.

Parameters **name_or_object** (*str or Any*) – A module/class object or a plugin name (string).

Returns **module** – The module object, or None if the `name_or_object` was not found.

Return type Any or None

2.2 HookCaller

class napari_plugin_engine.**HookCaller**(*name*, *hook_execute*, *namespace=None*, *spec_opts=None*)
The primary hook-calling object.

A *`PluginManager`* may have multiple `HookCaller` objects and they are stored in the `plugin_manager.hook` namespace, named after the *hook specification* that they represent. For instance:

```
pm = PluginManager("demo")
pm.add_hookspec(some_module)
# assuming `some_module` had an @hookspec named `my specification`
assert isinstance(pm.hook.my_specification, HookCaller)
```

Each `HookCaller` instance stores all of the *`HookImplementation`* objects discovered during *`plugin registration`* (each of which capture the implementation of a specific plugin for this hook specification).

The `HookCaller` instance also usually creates and stores a reference to the *`HookSpecification`* instance that encapsulates information about the hook specification, (at `HookCaller.spec`)

Parameters

- **`name`** (*`str`*) – The name of the *hook specification* that this `HookCaller` represents.
- **`hook_execute`** (*`Callable`*) – A *`HookExecFunc`* function. In almost every case, this will be provided by the *`PluginManager`* during hook registration as *`PluginManager._hookexec()`*... which is, in turn, mostly just a wrapper around *`_multicall()`*.
- **`namespace`** (*`Any, optional`*) – An namespace (such as a module or class) to search during *HookSpecification* creation for functions decorated with `@hookspec` named with the string `name`.
- **`spec_opts`** (*`Optional[dict], optional`*) – keyword arguments to be passed when creating the *`HookSpecification`* instance at `self.spec`.

`__call__`(**args*, *_plugin=None*, *_skip_impls=[]*, ***kwargs*)

Call hook implementation(s) for this spec and return result(s).

This is the primary way to call plugin hook implementations.

Note: Parameters are prefaced by underscores to reduce potential conflicts with argument names in hook specifications. There is a test in `test_hook_specifications.test_annotation_on_hook_specification()` to ensure that these argument names are never used in one of our hookspecs.

Parameters

- **`_plugin`** (*`str, optional`*) – The name of a specific plugin to use. By default all implementations will be called (though if `firstresult==True`, only the first non-None result will be returned).
- **`_skip_impls`** (*`List[HookImplementation], optional`*) – A list of HookImplementation instances that should be *skipped* when calling hook implementations, by default None
- **`**kwargs`** – keys should match the names of arguments in the corresponding hook specification, values will be passed as arguments to the hook implementations.

Raises

- ***`HookCallError`*** – If one or more of the keys in `kwargs` is not present in one of the `hook_impl.argnames`.
- ***`PluginCallError`*** – If `firstresult == True` and a plugin raises an Exception.

Returns

If the hookspec was declared with `firstresult==True`, a single result will be returned. Otherwise will return a list of results from all hook implementations for this hook caller.

If `_plugin` is provided, will return the single result from the specified plugin.

Return type result

_add_hookimpl(*hookimpl*)
Add an implementation to the callback chain.

_call_plugin(*plugin_name*, **args*, ***kwargs*)
Call the hook implementation for a specific plugin

Note: This method is not intended to be called directly. Instead, just call the instance directly, specifing the `_plugin` argument. See the *__call__()* method.

Parameters **plugin_name** (*str*) – Name of the plugin

Returns Result of implementation call provided by plugin

Return type Any

Raises

- **TypeError** – If the implementation is a hook wrapper (cannot be called directly)
- **TypeError** – If positional arguments are provided
- ***HookCallError*** – If one of the required arguments in the hook specification is not present in `kwargs`.
- ***PluginCallError*** – If an exception is raised when calling the plugin

_set_plugin_enabled(*plugin_name*, *enabled*)
Enable or disable the hook implementation for a specific plugin.

Parameters

- **plugin_name** (*str*) – The name of a plugin implementing `hook_spec`.
- **enabled** (*bool*) – Whether or not the implementation should be enabled.

Raises **KeyError** – If `plugin_name` has not provided a hook implementation for this hook specification.

bring_to_front(*new_order*)
Move items in `new_order` to the front of the call order.

By default, hook implementations are called in last-in-first-out order of registration, and pluggy does not provide a built-in way to rearrange the call order of hook implementations.

This function accepts a *HookCaller* instance and the desired `new_order` of the hook implementations (in the form of list of plugin names, or a list of actual *HookImplementation* instances) and reorders the implementations in the hook caller accordingly.

Note: Hook implementations are actually stored in *two* separate list attributes in the hook caller: `HookCaller._wrappers` and `HookCaller._nonwrappers`, according to whether the corresponding *HookImplementation* instance was marked as a wrapper or not. This method *only* sorts _nonwrappers.

Parameters **new_order** (list of str or list of *HookImplementation*) – instances The desired CALL ORDER of the hook implementations. The list does *not* need to include every

hook implementation in `get_hookimpls()`, but those that are not included will be left at the end of the call order.

Raises

- **`TypeError`** – If any item in `new_order` is neither a string (plugin_name) or a `HookImplementation` instance.
- **`ValueError`** – If any item in `new_order` is neither the name of a plugin or a `HookImplementation` instance that is present in self._nonwrappers.
- **`ValueError`** – If `new_order` argument has multiple entries for the same implementation.

Examples

Imagine you had a hook specification named `print_plugin_name`, that expected plugins to simply print their own name. An implementation might look like:

```
>>> # hook implementation for ``plugin_1``
>>> @hook_implementation
... def print_plugin_name():
...     print("plugin_1")
```

If three different plugins provided hook implementations. An example call for that hook might look like:

```
>>> plugin_manager.hook.print_plugin_name()
plugin_1
plugin_2
plugin_3
```

If you wanted to rearrange their call order, you could do this:

```
>>> new_order = ["plugin_2", "plugin_3", "plugin_1"]
>>> plugin_manager.hook.print_plugin_name.bring_to_front(new_order)
>>> plugin_manager.hook.print_plugin_name()
plugin_2
plugin_3
plugin_1
```

You can also just specify one or more item to move them to the front of the call order: >>> plugin_manager.hook.print_plugin_name.bring_to_front([“plugin_3”]) >>> plugin_manager.hook.print_plugin_name() plugin_3 plugin_2 plugin_1

`call_extra`(*methods*, *kwargs*)

Call the hook with some additional temporarily participating methods using the specified `kwargs` as call parameters.

`call_historic`(*result_callback=None*, *kwargs=None*, *with_impl=False*)

Call the hook with given `kwargs` for all registered plugins and for all plugins which will be registered afterwards.

If `result_callback` is not `None` it will be called for for each non-`None` result obtained from a hook implementation.

If `with_impl` is `True`, the caller is indicating that `result_callback` has a signature of `callback(result, hookimpl)`, and will be called as such.

call_with_result_obj(*, *_skip_impls=[]*, ***kwargs*)
Call hook implementation(s) for this spec and return HookResult.

The *HookResult* object carries the result (in its `result` property) but also additional information about the hook call, such as the implementation that returned each result and any call errors.

Parameters

- **_skip_impls** (*List[HookImplementation], optional*) – A list of HookImplementation instances that should be *skipped* when calling hook implementations, by default None
- ****kwargs** – keys should match the names of arguments in the corresponding hook specification, values will be passed as arguments to the hook implementations.

Returns **result** – A *HookResult* object that contains the results returned by plugins along with other metadata about the call.

Return type *HookResult*

Raises

- ***HookCallError*** – If one or more of the keys in `kwargs` is not present in one of the `hook_impl.argnames`.
- ***PluginCallError*** – If `firstresult == True` and a plugin raises an Exception.

disable_plugin(*plugin_name*)
disable implementation for `plugin_name`.

enable_plugin(*plugin_name*)
enable implementation for `plugin_name`.

get_plugin_implementation(*plugin_name*)
Return hook implementation instance for `plugin_name` if found.

index(*value*)
Return index of plugin_name or a HookImplementation in self._nonwrappers

Return type `int`

2.3 HookResult

class napari_plugin_engine.**HookResult**(*result*, *excinfo*, *firstresult=False*, *plugin_errors=None*)
A class to store/modify results from a *_multicall()* hook loop.

Results are accessed in `.result` property, which will also raise any exceptions that occured during the hook loop.

Parameters

- **results** (*List[Tuple[Any, HookImplementation]]*) – A list of (result, HookImplementation) tuples, with the result and HookImplementation object responsible for each result collected during a _multicall loop.
- **excinfo** (*tuple*) – The output of sys.exc_info() if raised during the multicall loop.
- **firstresult** (*bool, optional*) – Whether the hookspec had `firstresult == True`, by default False. If True, self._result, and self.implementation will be single values, otherwise they will be lists.

- **plugin_errors** (*list*) – A list of any *PluginCallError* instances that were created during the multicall loop.

force_result(*result*)
Force the result(s) to `result`.

This may be used by hookwrappers to alter this result object.

If the hook was marked as a `firstresult` a single value should be set otherwise set a (modified) list of results. Any exceptions found during invocation will be deleted.

classmethod from_call(*func*)
Used when hookcall monitoring is enabled.

https://pluggy.readthedocs.io/en/latest/#call-monitoring

implementation: Optional[Union[HookImplementation, List[HookImplementation]]] = None
The HookImplementation(s) that were responsible for each result in `result`

is_firstresult: bool = None
Whether this HookResult came from a `firstresult` multicall.

property result
Return the result(s) for this hook call.

If the hook was marked as a `firstresult` only a single value will be returned otherwise a list of results.

Return type Union[Any, List[Any]]

2.4 HookSpecification

class napari_plugin_engine.**HookSpecification**(*namespace*, *name*, *, *firstresult=False*, *historic=False*, *warn_on_impl=None*)
A class to encapsulate hook specifications.

2.5 HookImplementation

class napari_plugin_engine.**HookImplementation**(*function*, *plugin=None*, *plugin_name=None*, *hookwrapper=False*, *optionalhook=False*, *tryfirst=False*, *trylast=False*, *specname=''*, *enabled=True*)
A class to encapsulate hook implementations.

2.6 Decorators & Markers

2.6.1 HookSpecificationMarker

class napari_plugin_engine.**HookSpecificationMarker**(*project_name*)
Decorator helper class for marking functions as hook specifications.

You can instantiate it with a project_name to get a decorator. Calling *PluginManager.add_hookspecs()* later will discover all marked functions if the *PluginManager* uses the same project_name.

__call__(*function=None*, *firstresult=False*, *historic=False*, *warn_on_impl=None*)
if passed a function, directly sets attributes on the function which will make it discoverable to *PluginManager.add_hookspecs()*. If passed no function, returns a decorator which can be applied to a function later using the attributes supplied.

If `firstresult` is `True` the 1:N hook call (N being the number of registered hook implementation functions) will stop at I<=N when the I'th function returns a non-`None` result.

If `historic` is `True` calls to a hook will be memorized and replayed on later registered plugins.

2.6.2 HookImplementationMarker

class napari_plugin_engine.**HookImplementationMarker**(*project_name*)
Decorator helper class for marking functions as hook implementations.

You can instantiate with a `project_name` to get a decorator. Calling *PluginManager.register()* later will discover all marked functions if the *PluginManager* uses the same project_name.

Parameters **project_name** (*str*) – A namespace for plugin implementations. Implementations decorated with this class will be discovered by `PluginManager.register` if and only if `project_name` matches the `project_name` of the `PluginManager`.

__call__(*function=None*, ***, *hookwrapper=False*, *optionalhook=False*, *tryfirst=False*, *trylast=False*, *specname=''*)
Call the marker instance.

If passed a function, directly sets attributes on the function which will make it discoverable to *PluginManager.register()*. If passed no function, returns a decorator which can be applied to a function later using the attributes supplied.

Parameters

- **function** (*callable, optional*) – A function to decorate as a hook implementation, If `function` is None, this method returns a function that can be used to decorate other functions.
- **hookwrapper** (*bool, optional*) – Whether this hook implementation behaves as a hookwrapper. by default False
- **optionalhook** (*bool, optional*) – If `True`, a missing matching hook specification will not result in an error (by default it is an error if no matching spec is found), by default False.
- **tryfirst** (*bool, optional*) – If `True` this hook implementation will run as early as possible in the chain of N hook implementations for a specification, by default False
- **trylast** (*bool, optional*) – If `True` this hook implementation will run as late as possible in the chain of N hook implementations, by default False
- **specname** (*str, optional*) – If provided, `specname` will be used instead of the function name when matching this hook implementation to a hook specification during registration, by default the implementation function name must match the name of the corresponding hook specification.

Returns If `function` is not `None`, will decorate the function with attributes, and return the function. If `function` is None, will return a decorator that can be used to decorate functions.

Return type Callable

2.7 Exceptions

PluginError	Base class for exceptions relating to plugins.
PluginImportError	Plugin module is unimportable.
PluginRegistrationError	If an unexpected error occurs during registration.
PluginImplementationError	Base class for errors pertaining to a specific hook implementation.
PluginValidationError	When a plugin implementation fails validation.
PluginCallError	Raised when an error is raised when calling a plugin implementation.
HookCallError	If a hook is called incorrectly.

2.7.1 PluginError

class napari_plugin_engine.**PluginError**(*message=''*, ***, *plugin=None*, *plugin_name=None*, *cause=None*)

Bases: Exception

Base class for exceptions relating to plugins.

Parameters

- **message** (*str, optional*) – An optional error message, by default ‘’
- **namespace** (*Optional[Any], optional*) – The python object that caused the error, by default None
- **cause** (*Exception, optional*) – Exception that caused the error. Same as `raise * from`. by default None

classmethod get(***, *plugin=<Empty.token: 0>*, *plugin_name=<Empty.token: 0>*, *error_type=<Empty.token: 0>*)

Return errors that have been logged, filtered by parameters.

Parameters

- **manager** (*PluginManager, optional*) – If provided, will restrict errors to those that are owned by `manager`.
- **plugin_name** (*str*) – If provided, will restrict errors to those that were raised by `plugin_name`.
- **error_type** (*Exception*) – If provided, will restrict errors to instances of `error_type`.

Returns A list of PluginErrors that have been instantiated during this session that match the provided parameters.

Return type list of PluginError

Raises **TypeError** – If `error_type` is provided and is not an exception class.

info()

Return info as would be returned from sys.exc_info().

Return type Tuple[Type[Exception], Exception, Optional[traceback]]

log(*package_info=True*, *logger=None*, *level=40*)

Log this error with metadata, optionally provide logger and level.

Parameters

- **package_info** (*bool, optional*) – If true, will include package metadata in log, by default True
- **logger** (*logging.Logger or str, optional*) – A Logger instance or name of a logger to use, by default None
- **level** (*int, optional*) – The logging level to use, by default logging.ERROR

2.7.2 PluginImportError

class napari_plugin_engine.**PluginImportError**(*message=''*, ***, *plugin=None*, *plugin_name=None*, *cause=None*)

Bases: napari_plugin_engine.exceptions.PluginError, ImportError

Plugin module is unimportable.

2.7.3 PluginRegistrationError

class napari_plugin_engine.**PluginRegistrationError**(*message=''*, ***, *plugin=None*, *plugin_name=None*, *cause=None*)

Bases: napari_plugin_engine.exceptions.PluginError

If an unexpected error occurs during registration.

2.7.4 PluginImplementationError

class napari_plugin_engine.**PluginImplementationError**(*hook_implementation*, *msg=None*, *cause=None*)

Bases: napari_plugin_engine.exceptions.PluginError

Base class for errors pertaining to a specific hook implementation.

2.7.5 PluginValidationError

class napari_plugin_engine.**PluginValidationError**(*hook_implementation*, *msg=None*, *cause=None*)

Bases: napari_plugin_engine.exceptions.PluginImplementationError

When a plugin implementation fails validation.

2.7.6 PluginCallError

class napari_plugin_engine.**PluginCallError**(*hook_implementation*, *msg=None*, *cause=None*)

Bases: napari_plugin_engine.exceptions.PluginImplementationError

Raised when an error is raised when calling a plugin implementation.

2.7.7 HookCallError

class napari_plugin_engine.**HookCallError**(*message=''*, ***, *plugin=None*, *plugin_name=None*, *cause=None*)

Bases: napari_plugin_engine.exceptions.PluginError

If a hook is called incorrectly.

Usually this results when a HookCaller is called without the appropriate arguments.

2.8 Extra Functions

napari_plugin_engine.hooks.**_multicall**(*hook_impls*, *caller_kwargs*, *firstresult=False*)

The primary *HookImplementation* call loop.

Parameters

- **hook_impls** (*list*) – A sequence of hook implementation (HookImplementation) objects
- **caller_kwargs** (*dict*) – Keyword:value pairs to pass to each hook_impl.function. Every key in the dict must be present in the argnames property for each hook_impl in hook_impls.
- **firstresult** (*bool, optional*) – If True, return the first non-null result found, otherwise, return a list of results from all hook implementations, by default False

Returns **outcome** – A HookResult object that contains the results returned by plugins along with other metadata about the call.

Return type *HookResult*

Raises

- ***HookCallError*** – If one or more of the keys in caller_kwargs is not present in one of the hook_impl.argnames.
- ***PluginCallError*** – If firstresult == True and a plugin raises an Exception.

napari_plugin_engine.manager.**ensure_namespace**(*obj*, *name='orphan'*)

Convert a dict to an object that provides getattr.

Parameters

- **obj** (*Any*) – An object, may be a dict, or a regular namespace object.
- **name** (*str, optional*) – A name to use for the new namespace, if created. by default 'orphan'

Returns A namespace object. If obj is a dict, creates a new type named name, prepopulated with the key:value pairs from obj. Otherwise, if obj is not a dict, will return the original obj.

Return type type

Raises **ValueError** – If obj is a dict that contains keys that are not valid identifiers.

napari_plugin_engine.manager.**temp_path_additions**(*path*)

A context manager that temporarily adds path to sys.path.

Parameters **path** (*str or list of str*) – A path or list of paths to add to sys.path

Yields **sys_path** (*list of str*) – The current sys.path for the context.

Return type Generator

napari_plugin_engine.dist.**get_dist**(*obj*)
Return a `importlib.metadata.Distribution` for any python object.

Parameters **obj** (*Any*) – A python object. If a string, will be interpreted as a dist name.

Returns **dist** – The distribution object for the corresponding package, if found.

Return type Distribution

2.9 Types

napari_plugin_engine.hooks.**HookExecFunc = typing.Callable[[ForwardRef('HookCaller'), typing**
A function that loops calling a list of *HookImplementation* s and returns a *HookResult*.

Parameters

- **hook_caller** (HookCaller) – a `HookCaller` instance.
- **hook_impls** (*List*[HookImplementation]) – a list of *HookImplementation* instances to call.
- **kwargs** (*dict*) – a mapping of keyword arguments to provide to the implementation.

Returns **result** – The *HookResult* object resulting from the call loop.

Return type *HookResult*

INDEX

Symbols

A

B

C

D

E

F

G

H

I

L

P

R

S

T

U