KeepMePosted Documentation

Release 0.1

Kale Kundert

June 01, 2014

Contents

This module provides an object-oriented event handling framework. In this framework, events are registered by classes and then broadcasted by individual objects. Listening for events from specific objects is made easy.

CHAPTER 1

Simple Example

The most important parts of this framework are the Dispatcher class and the event() decorator. Dispatcher is a base class for objects that want to broadcast events and the event decorator is used to register events.

```
>>> from kemepo import Dispatcher, event
>>> class Button (Dispatcher):
        @event
        def on_press(self):
            print('Calling internal handler')
```

The method decorated by event() is taken to be the "internal handler", distinct from any "external observers" that may be attached using connect() later on. When an event is triggered using handle(), the internal handler is called before the external observers.

```
>>> button = Button()
>>> button.connect(on_press=lambda: print('Calling external observer.'))
>>> button.handle('on_press')
Calling internal handler.
Calling external observer.
```

CHAPTER 2

Installation

KeepMePosted can be installed from PyPI:

```
$ pip install kemepo
```

You can also download the source code directly from GitHub. The code is made available under the MIT license. If you find the code useful and want to make improvements, feel free to make pull requests:

```
$ git clone https://github.com/kalekundert/KeepMePosted.git kemepo
```

CHAPTER 3

Registering Events

Within this framework, objects can only broadcast events that have already been registered with their class. Typically, events are registered when the class is created using the event() decorator:

```
>>> class CheckBox (Dispatcher):
        @event
        def on_check(self):
            print('Calling internal handler')
```

This registers a new event based on the given method. The name of the event is the name of the method, and the method itself becomes the internal handler for that type of event. Furthermore, the argument signature and the docstring of the handler are used for error checking and documentation, respectively. This information is often useful even if the handler itself is left unimplemented.

It is possible to register new events without using the event decorator. The advantage of doing this is that you can register events after the class has been created. You also don't need to specify an internal handler (the second argument) when manually registering events, although doing so provides improved error checking and documentation, as discussed above.

```
>>> CheckBox.register_event('on_uncheck', lambda: None)
```

All that said, you should very rarely need to manually register events. In the typical case, the event() decorator should be preferred.

CHAPTER 4

Triggering Events

There are two ways to trigger an event: handle() and notify(). The difference concerns the internal handler (i.e. the callback used the register the event), which is called by handle() and not called by notify(). Referring back to the button example from the first section:

```
>>> button.handle('on_press')
Calling internal handler.
Calling external observer.
>>> button.notify('on_press')
Calling external observer.
```

Usually you should use notify(). Use notify() only in cases where notify() would create infinite recursion. If you simply don't want the internal handler to do anything, just leave it unimplemented.

CHAPTER 5

Reacting to Events

Although only objects that inherit from Dispatcher can broadcast events, any callback can be used to react to events. The connect() method provides a very flexible interface for connecting observers to dispatchers. In particular, observers can be provided either as keyword arguments mapping event names to callbacks or as objects with method names matching event names.

The former approach is probably more intuitive. Any type of callable can be used as an observer callback, including functions and lambda functions.

```
>>> def observer_function(): print("Calling an observer function.")
>>> button.connect(on_press=observer_function)
>>> button.connect(on_press=lambda: print("Calling an observer lambda."))
```

The latter approach provides a powerful way to listen to many events from the same object. Provide any number of arguments to connect, and each will be searched for methods with names matching registered events. Those methods will be connected as observers of those events.

```
>>> class Observer:
        def on_press(self):
            print("Calling an observer method.")

>>> observer_object = Observer()
>>> button.connect(observer_object)
```

No matter an observer is specified, it must have the same argument signature as the internal handler used to register the event. A TypeError will be raised otherwise.

Events can be disconnected using the disconnect() method.

```
>>> button.disconnect(observer_function)
>>> button.disconnect(observer_object)
```

CHAPTER 6

Error Checking

Strong error checking is possible because events are registered when the class is created. Exceptions are thrown if you attempt any of the following:

1. Connect to an undefined event.
2. Handle an undefined event.
3. Connect an observer that doesn't have the same argument signature as the internal handler.
4. Handle an event without providing the arugments expected by the internal handler.

CHAPTER 7

Docstring Generation

One advantage of registering events using the event() decorator (e.g. before the class in question has been created) is that those events can be incorporated into the class docstring. This is useful both for use with help() in the python interpreter and for use with Sphinx for online documentation.

To incorporate event documentation into the docstring of a Dispatcher subclass, just include the string '{events}'. This will be replaced by a list of the events that are registered with that class. (Note that only events registered using the event() decorator will be included.) Replacement is done using standard string formatting, so this is roughly what's going on behind the scenes:

```
>>> cls.__doc__ = cls.__doc__.format(events=events_docstring)
```

You can control the exact format of the event documentation using the set_docstring_formatter() function. This function takes one argument, which can either be the name of a built-in formatter or a custom formatter function.

Currently, the two built-in formatters are named *pretty* and *sphinx*. The *pretty* formatter is the default. It's the more readable of the two and it's meant to look good in interpreter sessions, but it's not rendered very nicely by Sphinx (although it does produce legal restructured text). The *sphinx* formatter is a more heavily marked-up alternative that looks better when rendered by Sphinx. To use the *sphinx* formatter in Sphinx, but these lines in `docs/conf.py`:

```
>>> import kemepo
>>> kemepo.set_docstring_formatter('sphinx')
```

This must be done before you import any of your Dispatcher subclasses, because the docstrings are created at the same time as the class itself.

If you want to write a custom formatter, provide a function that accepts a single OrderedDict argument. This is a mapping between event names and EventMetaData objects, in the order that the events were defined. Return a string to incorporate into the class docstring. You may find the format_arg_spec() and format_description() functions useful.

CHAPTER 8

API Documentation

class kemepo.**Dispatcher**

Provide event handling functionality. Meant to be subclassed.

Only subclasses of Dispatcher can register and broadcast events. Registration of events is mostly done using the event decorator, but can also be done manually. Each type of event can be associated with one internal handler and any number of external observers. The internal handler must be specified when the event is registered (e.g. using the event decorator) while external observers can be added or removed at any time using connect() and disconnect().

Two methods are provided to broadcast events: handle() and notify(). The former invokes the internal handler and the external observers, while the latter only invokes the external observers.

Note that you can include documentation for every event registered to a Dispatcher subclass by adding the string '{events}' to the subclass's docstring. This will be replaced by a informative message at runtime. This is useful both for interactive usage and for Sphinx documentation.

classmethod register_event(*event*, *handler=None*)

Register the given event name.

Typically you would not directly call this method. Instead you would use the event() decorator to define events from methods.

If a handler is given, it will be set as the internal handler for this event. The internal handler is invoked before any external observers when this event is triggered. The internal handler is also used for error-checking and documentation purposes, so it is good to provide one even if it doesn't do anything.

classmethod get_registered_events()

Return a list of all the events that have been registered with this class. This list will include events defined in parent classes, and will be in the same order that the events were defined in.

classmethod get_registered_event_metadata(*event*)

Return the metadata for the given event. If the given event has not been defined for this class, a TypeError is raised.

classmethod list_registered_events()

Print out every event registered by this class. This is meant to make debugging easier.

handle(*event*, **args*, ***kwargs*)

Handle the given event.

This invokes the internal handler for this event and any external observers attached to this event using connect().

notify(*event*, **args*, ***kwargs*)

Notify observers about the given event.

Unlike handle(), this method doesn't invoke the internal handler for the given event. In other words, it only invokes external observers attached to the event. This is a useful distinction when you're trying to avoid infinite loops, but otherwise handle() should be preferred.

connect (**observer_objects*, ***observer_callbacks*)
Attach observers to events.

This method is very flexible in the arguments it takes. The keyword arguments are the simpler case. These arguments are expected to be simple 'event=callback' pairs. The callback will then be invoked whenever the event is triggered, until it is disconnected. An exception will be thrown if the specified event isn't registered.

The regular arguments are more complicated. These are taken to be objects with methods that are meant to be observers. In particular, a method is taken to be an observer if its name matches the name of an event registered with this dispatcher. Every such method found will be invoked whenever its corresponding event is triggered.

If an internal handler was specified when the event was registered (the usual case), every observer is check to make sure it takes the same arguments as that handler. This is a useful way to catch programming mistakes. Even if the internal handler doesn't do anything, it can still help catch errors in the observers.

disconnect (*observer*)
Disconnect the given observer from any events it may be connected to.

kemepo.**event** (*handler*)
Register a new type of event.

New events can be registered by providing a handler function. The name of the function is used as the name of the event, and the function itself is setup to be called whenever the event needs to be handled. This decorator makes it easy register events from method in Dispatcher subclasses.

```
>>> class Button (Dispatcher):
        @event
        def on_press(self):
            print('Calling internal handler.')
```

Methods that get decorated by event() become "internal handlers", distinct from "external observers" that can be attached later on using connect(). When an event is triggered, the handler for that event is always called before any observers.

Technical detail: This decorator can't actually register the new event, because it is called before the class is created. Instead, it just marks the method so that a new event will be created when the class is created. The actual event registration is handled by DispatcherMetaclass.

kemepo.**set_docstring_formatter** (*formatter*)
Set the function used to format event docstrings.

The formatter argument can either be a function which accepts a dictionary of registered events (i.e. mapping event names to EventMetadata objects) or the name of one of the builtin formatters:

- "pretty" – Easier to read in the terminal, used by default.
- "sphinx" – More verbose, but valid restructured text.

kemepo.**pretty_docstring_formatter** (*cls*, *registered_events*)
Use a human-readable format to display information about the given events.

kemepo.**sphinx_docstring_formatter** (*cls*, *registered_events*)
Use a strict restructured-text format the display information about the given events. This is less readable that the "pretty" format, but it is rendered nicely by Sphinx. If you want to make sphinx documentation, put these lines in your configuration file:

```
>>> import kemepo
>>> kemepo.set_docstring_formatter('sphinx')
>>> import your_dispatcher_subclasses
```

kemepo.**format_arg_spec**(*metadata*)
Return a string representing the arguments taken by the handler for the given event. This is meant to be used by the docstring formatters.

kemepo.**format_description**(*metadata*, *indent=4*)
Return a properly indented paragraph describing the given event. This is meant to be used by the docstring formatters.

class kemepo.**DispatcherMetaclass**(*name*, *bases*, *dict*)
Instantiate Dispatcher subclasses.

This class has two primary roles:

1. Register events marked by the event() decorator.
2. Generate docstring based on the registered events.

This first role is just a technical detail. Decorators are a convenient way to specify events to register, but they can't actually register events because they're invoked before the class has been created. So instead they just label the methods, and the metaclass does the real registration.

The second role is a convenience. If you put the string '{events}' in the docstring of a Dispatcher subclass, it will be replaced at runtime by a description of the events registered in that subclass.

class kemepo.**EventMetadata**(*handler=None*)
Store information about a registered event.

The information stored in this object is used in almost every aspect of this module. A brief overview of this information is given below. The name of the event is conspicuously not included. This is because dispatchers store event metadata in dictionaries where the keys are the event names, so storing the names again here would be redundant.

The internal handler: Called before any observers when the event is being handled.

The handler's argument specification: Used the validate external observers.

The documentation for the event: Used to add event info to Dispatcher docstrings. This documentation usually comes from the docstring of the internal handler.

validate_arguments(*args*, *kwargs*)
Raise an exception if the given arguments are not compatible with the handler for this event.

This method is currently left unimplemented, because an exception will be thrown anyway once the handler is called with the wrong arguments. I left this machinery in because I think more stringent error-checking might be useful in the future.

validate_observer(*observer*)
Raise an exception if the given observer takes different arguments that the internal handler for this type of event.