Events Documentation

Release 0.3

Nicola Iarocci

Jun 02, 2017

Contents

Bringing the elegance of C# EventHandler to Python

The concept of events is heavily used in GUI libraries and is the foundation for most implementations of the MVC (Model, View, Controller) design pattern. Another prominent use of events is in communication protocol stacks, where lower protocol layers need to inform upper layers of incoming data and the like. Here is a handy class that encapsulates the core to event subscription and event firing and feels like a "natural" part of the language.

The package has been tested under Python 2.6, 2.7, 3.3 and 3.4.

CHAPTER 1

Usage

The C# language provides a handy way to declare, subscribe to and fire events. Technically, an event is a “slot” where callback functions (event handlers) can be attached to - a process referred to as subscribing to an event. To subscribe to an event:

```
>>> def something_changed(reason):
...     print "something changed because %s" % reason
...

>>> from events import Events
>>> events = Events()
>>> events.on_change += something_changed
```

Multiple callback functions can subscribe to the same event. When the event is fired, all attached event handlers are invoked in sequence. To fire the event, perform a call on the slot:

```
>>> events.on_change('it had to happen')
something changed because it had to happen
```

Usually, instances of `Events` will not hang around loosely like above, but will typically be embedded in model objects, like here:

```
class MyModel(object):
    def __init__(self):
        self.events = Events()
        ...
```

Similarly, view and controller objects will be the prime event subscribers:

```
class MyModelView(SomeWidget):
    def __init__(self, model):
        ...
        self.model = model
        model.events.on_change += self.display_value
        ...
    def display_value(self):
        ...
```

Introspection

The `Events` and `_EventSlot` classes provide some introspection support. This is usefull for example for automatic event subscription based on method name patterns.

```
>>> from events import Events
>>> events = Events()
>>> print events
<events.events.Events object at 0x104e5d5f0>

>>> def changed():
...     print "something changed"
...

>>> def another_one():
...     print "something changed here too"
...

>>> def deleted():
...     print "something got deleted!"
...

>>> events.on_change += changed
>>> events.on_change += another_one
>>> events.on_delete += deleted

>>> print len(events)
2

>>> for event in events:
...     print event.__name__
...
on_change
on_delete

>>> event = events.on_change
>>> print event
event 'on_change'

>>> print len(event)
2

>>> for handler in event:
...     print handler.__name__
...
changed
another_one

>>> print event[0]
<function changed at 0x104e5c230>
```

```
>>> print event[0].__name__
changed

>>> print len(events.on_delete)
1

>>> events.on_change()
something changed
somethind changed here too

>>> events.on_delete()
something got deleted!
```

Event names

Note that by default `Events` does not check if an event that is being subscribed to can actually be fired, unless the class attribute `__events__` is defined. This can cause a problem if an event name is slightly misspelled. If this is an issue, subclass `Events` and list the possible events, like:

```
class MyEvents(Events):
    __events__ = ('on_this', 'on_that', )

events = MyEvents()

# this will raise an EventsException as `on_change` is unknown to MyEvents:
events.on_change += changed
```

You can also predefine events for a single `Events` instance by passing an iterator to the constructor.

```
events = Events(('on_this', 'on_that'))

# this will raise an EventsException as `on_change` is unknown to MyEvents:
events.on_change += changed
```

It is recommended to use the constructor method for one time use cases. For more complicated use cases, it is recommended to subclass `Events` and define `__events__`.

You can also leverage both the constructor method and the `__events__` attribute to restrict events for specific instances:

```
DatabaseEvents(Events):
    __events__ = ('insert', 'update', 'delete', 'select')

audit_events = ('select')

AppDatabaseEvents = DatabaseEvents()

# only knows the 'select' event from DatabaseEvents
AuditDatabaseEvents = DatabaseEvents(audit_events)
```

CHAPTER 2

Installing

Events is on PyPI so all you need to do is:

```
pip install events
```

CHAPTER 3

Testing

Just run:

```
python setup.py test
```

The package has been tested under Python 2.6, Python 2.7 and Python 3.3.

CHAPTER 4

Source Code

Source code is available at GitHub.

CHAPTER 5

Attribution

Based on the excellent recipe by Zoran Isailovski, Copyright (c) 2005.

CHAPTER 6

Copyright Notice