Eventize

Release 0.4.3

Grégory Salvan

December 01, 2014

Contents

Contents:

CHAPTER 1

Introduction

Eventize permits to listen to *"before"* and *"after"* methods events and *"on_get"*, *"on_change"*, *"on_set"*, *"on_del"* attributes events.

Features:

- Can be used as a simple subject/observer pattern
- Conditional events
- Use descriptors to lazily and unobstrusively listen to "on_get", "on_change","on_set", "on_del" events on attributes and "before" and "after" events on methods.
- Precise callbacks inheritance (see Subject)
- Statically and dynamically customizable (via inheritance, pattern visitor over a "*modifier*" protocol, decorators...)

Fine grained event dispatcher

It can listen to events at 3 levels (by order of execution):

- **Descriptor Class**: for all classes which use the given Attribute or Method descriptor class
- **Descriptor Instance**: for all objects of a given class
- **Object instance**: for the given object attribute value or method

CHAPTER 2

Installation

Install it from pypi:

```
pip install eventize
```

or from sources:

```
git clone git@github.com:apieum/eventize.git
cd eventize
python setup.py install
```

CHAPTER 3

Basic Usage

3.1 Simple subject/observer

events.Handler is the base class of all eventize handlers ("on_get", "before"...)

It is a simple callable list of functions wich receive the argument (of type *events.Event*) you've passed when calling your *Handler* object.

As a list an *Handler* support common methods *"append"*, *"remove"*, *"prepend"*, *"insert"*, *"extend"*, *"empty"*..., *"__setitem__"*, plus some syntactic sugar like *"__iadd__"* (+=) for append and *"__isub__"* (-=) for remove.

You can stop event propagation by raising an *events.StopPropagation* exception which store exception message in *"Event.messages"* by default.

You can hook event propagation by overriding methods *"before_propagation"* and *"after_propagation"* or dynamically change *Handler* behaviour at creation by passing visitors (object with a method *"visit"*) see *events.EventType* visitor for example.

An handler can build its proper events of the class defined in *Handler.event_type* when calling *Handler.make_event* (just create and returns an event instanciated with given arguments) or *Handler.notify* (create event with *make_event* and propagates it)

You can add conditional handlers by using method *"when"* or restrict the current handler execution by passing *"condition"* kwarg argument to constructor. Conditions can be chained with methods *"do"* or *"then"* (aliases of *"append"*)

Each time you trigger an event, it is stored in *Handler.events*. You can empty past events by calling *"clear_events"* or all (events and callbacks) with *"clear"*.

```
from eventize.events import Handler
from eventize.typing import Visitor

# a condition
def is_string(event):
  return isinstance(event.content, str)

# an observer
def titlecase(event):
  event.content = event.content.title()

# A visitor
class WeirdVisitor(Visitor):
  def visit(self, handler):
    # it add/prepend observer "save_default"
    handler.prepend([self.save_default])
```

```
  def save_default(self, event):
    self.default = event.content

my_visitor = WeirdVisitor()
handler = Handler(titlecase, my_visitor, condition=is_string)

# An Handler is a callable list
assert isinstance(handler, list)
assert callable(handler)

# handler contains 2 callbacks:
assert len(handler) == 2
assert titlecase in handler
assert my_visitor.save_default in handler
# it removes titlecase
handler -= titlecase
assert titlecase not in handler
# it adds titlecase
handler += titlecase


# Create an event with attribute content and trigger it
event1 = handler.notify(content="a string") # each kwarg is converted in event attribute

# result of first observer (added by WeirdVisitor)
assert my_visitor.default == "a string"
# result of all observers (i.e. titlecase here)
assert event1.content == "A String"

# if event.content is not a string propagation is stopped
# these 2 lines are sames as notify
event2 = handler.make_event(content=1234)
handler(event2)

assert len(handler.events) == 2
assert handler.events == (event1, event2)
expected_message = "Condition '%s' for event 'Event' return False" % id(is_string)
assert event2.messages[0] == expected_message

# we remove all past events:
handler.clear_events()
assert len(handler.events) == 0

# we remove all observers and events:
handler.clear()
assert len(handler) == 0

is_a_name = lambda event: event.content == "a name"
# create a new subhandler with a condition:
handler.when(is_a_name).do(my_visitor.save_default).then(titlecase)
event1 = handler.notify(content="a name")
event2 = handler.notify(content="a string")
# only "a name" is titlecased
assert event1.content == "A Name"
assert event2.content == "a string"

# save_default is called only for event1:
assert my_visitor.default == "a name"
```

3.2 Observe a method

To observe a method, you can:

- declare your method at class level with *"Method"* and a function as first argument
- decorate a method with *"Method"*
- use functions *"handle"*, *"before"* or *"after"* on class or object callable attribute with type of event in the optionalthird argument (recommended)

Method events objects are of type BeforeEvent and AfterEvent y default.

They have 4 main attributes:

- *"subject"*: the object instance where event happens
- *"name"*: the method name of the object instance
- *"args"*: the tuple of passed args
- *"kwargs"*: the dict of named args

```
from eventize import before, after
from eventize.method import BeforeEvent, AfterEvent


class Observed(object):
  def __init__(self):
    self.valid = False

  def is_valid(self, *args):
    return self.valid

  def not_valid(self, event):
    assert event.name == "is_valid" # (event subject name)
    assert event.subject == self
    self.valid = not self.valid

class Logger(list):
  def log_before(self, event):
    assert type(event) is BeforeEvent
    self.append(self.message('before %s'  % event.name, *event.args, is_valid=event.subject.valid))

  def log_after(self, event):
    assert type(event) is AfterEvent
    self.append(self.message('after %s' % event.name, *event.args, is_valid=event.subject.valid))

  def message(self, event_name, *args, **kwargs):
    return "%s called with args: '%s', current:'%s'" % (event_name, args, kwargs['is_valid'])



args_have_permute = lambda event: 'permute' in event.args

my_object = Observed()
my_logs = Logger()

before_is_valid = before(my_object, 'is_valid')
before_is_valid += my_logs.log_before
before_is_valid.when(args_have_permute).do(my_object.not_valid)
```

```
after(my_object, 'is_valid').do(my_logs.log_after)

assert my_object.is_valid() is False
assert my_object.is_valid('permute') is True

assert my_logs == [
  my_logs.message('before is_valid', is_valid=False),
  my_logs.message('after is_valid', is_valid=False),
  my_logs.message('before is_valid', 'permute', is_valid=False),
  my_logs.message('after is_valid', 'permute', is_valid=True),
]
```

3.3 Observe an attribute

***"Attribute"* is like *"Method"*, to observe it you can:**

- declare your attribute at class level with *"Attribute"* and an optionnal default value as first argument
- decorate an existing attribute with *"Attribute"*
- use functions *"handle"*, *"on_get"*, *"on_change"*, *"on_set"*, *"on_del"* on class or object attribute with the type of event on the third argument (recommended)

Attribute events objects are of type OnGetEvent, OnChangeEvent, OnSetEvent, OnDelEvent.

They have 3 main attributes:

- *"subject"*: the object instance where event happens
- *"name"*: the attribute name of the object instance
- *"value"*: the attribute value if set

In addition each kwarg is added to event as an attribute. (like "content" in ex 0)

```
from eventize import handle, on_get, Attribute
from eventize.attribute import OnGetEvent, OnGetHandler


class Validator(object):
  def __init__(self, is_valid):
    self.valid = is_valid
  def __call__(self):
    return self.valid

class Observed(object):
  validate = Validator(False)

class Logger(list):
  def log_get(self, event):
    assert type(event) is OnGetEvent, "Get event of type %s" % type(event)
    self.append(self.message('on_get', event.name, event.value()))
  def log_change(self, event):
    self.append(self.message('on_change', event.name, event.value()))
  def log_set(self, event):
    self.append(self.message('on_set', event.name, event.value()))
  def log_del(self, event):
    self.append(self.message('on_del', event.name, event.value()))
```

```
  def message(self, event_name, attr_name, value):
    return "'%s' called for attribute '%s', with value '%s'" % (event_name, attr_name, value)

my_object = Observed()
my_logs = Logger()
# Adding observers at object instance:
my_object_validate = handle(my_object, 'validate')
my_object_validate.on_get += my_logs.log_get
my_object_validate.on_change += my_logs.log_change
my_object_validate.on_set += my_logs.log_set
my_object_validate.on_del += my_logs.log_del

# Adding observers for all objects of class Observed
Observed_validate = handle(Observed, 'validate')
Observed_validate.on_get += my_logs.log_get
Observed_validate.on_change += my_logs.log_change
Observed_validate.on_set += my_logs.log_set
Observed_validate.on_del += my_logs.log_del

# Trigger on_get event
is_valid = getattr(my_object, 'validate')  # same as my_object.validate
# check if default value is False as defined in class
assert is_valid() == False, '[error] Default value was not set'
# Trigger on_set event
setattr(my_object, 'validate', Validator(True))  # # same as my_object.validate = Validator(True)
# Trigger on_del event
delattr(my_object, 'validate') # same as del my_object.validate

assert my_logs == [
  my_logs.message('on_get', 'validate', False),  # Called at class level
  my_logs.message('on_get', 'validate', False),  # Called at instance level
  my_logs.message('on_set', 'validate', True),   # Called at class level
  my_logs.message('on_set', 'validate', True),   # Called at instance level
  my_logs.message('on_change', 'validate', True),   # Called at class level
  my_logs.message('on_change', 'validate', True),   # Called at instance level
  my_logs.message('on_del', 'validate', True),   # Called at class level
  my_logs.message('on_del', 'validate', True),   # Called at instance level
]

# You can use your own events types
class OnGetCall(OnGetEvent):
  def returns(self):
    return self.value()

# and override Attribute or Method types
class CallAttr(Attribute):
  # must be redefined otherwise callbacks are appended to class Attribute
  # see example 3 for callbacks inheritance
  on_get = OnGetHandler()


my_object = Observed()
# third argument permits to set new type of attribute
on_get_validate = on_get(my_object, 'validate', CallAttr)
# set event type
on_get_validate.event_type = OnGetCall

assert isinstance(Observed.validate, CallAttr)
```

```
# OnGetCall Event returns my_object.validate()
assert my_object.validate is False
assert len(on_get_validate) == 0, "Expect my_object.validate.on_get has no callbacks"

## Defining observers at Attribute level:
# an observer
def set_to_true(event):
  assert type(event) == OnGetCall
  event.value = Validator(True)

# All objects with CallAttr attribute will call set_to_true
CallAttr.on_get += set_to_true

# set_to_true change value and check if event is of type OnGetCall
assert my_object.validate is True

# remove all callbacks and events at descriptor, class and instance level
handle(my_object, 'validate').clear_all()

assert len(CallAttr.on_get) == 0
```

CHAPTER 4

Advanced Usage

4.1 Observers inheritance:

Eventize heavily uses Descriptors, which in python don't know their owner until a getter is called. Yet, as they help to define classes, it could be interesting to bind them to their class at class creation.

It's the aim of Subject decorator. A Subject is a class that contains descriptors handlers (on_get, before...)

Subject does 2 things:

- it makes children handlers inheriting their parent handlers observers (parent handlers are found by their attribute name).
- it calls method handler.bind (if exists) with the owner class as an argument while class is declared.

Subject decorator searches only for types of descriptors given when instanciating events.Subject class.

You can create your own subjects with *"events.Subject([descriptor_type1, [...]])"*.

Eventize comes with already built Subjects for Attributes and Method:

Attribute Subject (*attribute.Subject*) is equivalent to *events.Subject(OnGetHandler, OnSetHandler, OnDelHandler, OnChangeHandler)*

Method Subject (*method.Subject*) is equivalent to *events.Subject(BeforeHandler, AfterHandler)*

```
from eventize import Attribute
from eventize.attribute import Subject, OnSetHandler

def validate_string(event):
    if isinstance(event.value, type('')): return

    message = "%s.%s must be a string!" % (type(event.subject).__name__, event.name)
    raise TypeError(message)

# an observer
def titlecase(event):
    event.value = event.value.title()

# user defined attribute with preloaded observer
class StringAttribute(Attribute):
    on_set = OnSetHandler(validate_string)

# @Subject with StringAttribute inheritance is equivalent to
# resetting on_get, on_del... + defining:
# on_set = OnSetHandler(validate_string, titlecase)
```

```
@Subject  # Bind handlers to the class
class Name(StringAttribute):
    on_set = OnSetHandler(titlecase)

assert titlecase not in StringAttribute.on_set
assert titlecase in Name.on_set

class Person(object):
    name = Name('john doe')

john = Person()

validation_fails = False
try:
    john.name = 0x007
except TypeError:
    validation_fails = True

assert validation_fails, "Validation should fail"
assert john.name == 'John Doe'  # Name is set in title case
```

Remember when inheriting a Method or Attribute descriptor if you don't override each event handler (on_get, on_set, before...) they are parent's ones. That's where *Subject* comes handy.

```
from eventize import Attribute

def titlecase(event):
    event.value = event.value.title()

class Name(Attribute):
    """nothing new"""

# when doing this:
Name.on_set.do(titlecase)
# all classes which use Attribute will have titlecase callback
assert titlecase in Attribute.on_set
# because without Subject:
assert Name.on_set is Attribute.on_set
```

CHAPTER 5

Development

Contributions are greatly appreciated.

Please use github (issue tracker, pull requests...) or contact me at apieum [at] gmail [dot] com

5.1 Testing

Tests are my specs so code (except refactorings) without tests won't probably be accepted. If you want to contribute please add tests.

Test recommended requirements:

```
pip install -r dev-requirements.txt
```

Sometimes –spec-color doesn't function. You should uninstall nosecolor and nosespec then reinstall nosecolor and nosespec separatly (nosecolor first).

You can fix it like this:

```
pip uninstall nosespec nosecolor
pip install nosecolor && pip install nosespec
```

In order to have fast feedback with TDD loops, I develop with two virtualenvs (2.7 and 3.3 python versions) launched in a splited shell (tmux) which runs tests each time a file changes. Use the code below with option –with-watch to launch tests this way.

Launching tests:

```
git clone git@github.com:apieum/eventize.git
cd eventize
nosetests --with-spec --spec-color ./eventize
# or with watch
# nosetests --with-spec --spec-color --with-watch ./eventize
```

5.2 Documentation

Documentation is generated by sphinx from restructured text localized in doc/source. It is build by make (see doc/Makefile).

Except to explain implementation choices (why), **please avoid comments in code**. It aims at keeping focus on coding and avoiding outdated comments. Keep long names (vars, functions, classes...) and explicit tests names (complete sentences) to have understandable code.

Building doc:

```
git clone git@github.com:apieum/eventize.git
cd eventize/eventize/doc
make all  # or make *target* (see in Makefile for *target*)
```

5.3 Continuous Integration

CI is made by travis for python versions 2.7, 3.2 and 3.3.

It checks:

- test suites (nosetest)
- rst-lint this README
- doc building
- code coverage

CHAPTER 6

Indices and tables

- *genindex*
- *modindex*
- *search*