python-dispatch Documentation

Release 0.1.31

Matthew Reid

Aug 02, 2020

Contents

CHAPTER 1

python-dispatch

Lightweight event handling for Python

1.1 Description

This is an implementation of the "Observer Pattern" with inspiration from the Kivy framework. Many of the features though are intentionally stripped down and more generalized. The goal is to have a simple drop-in library with no dependencies that stays out of the programmer's way.

1.2 Installation

```
pip install python-dispatch
```

1.3 Links

Project Home	https://github.com/nocarryr/python-dispatch
PyPI	https://pypi.python.org/pypi/python-dispatch
Documentation	https://python-dispatch.readthedocs.io

1.4 Usage

1.4.1 Events

```
from pydispatch import Dispatcher

class MyEmitter(Dispatcher):
    # Events are defined in classes and subclasses with the '_events_' attribute
    _events_ = ['on_state', 'new_data']
    def do_some_stuff(self):
        # do stuff that makes new data
        data = self.get_some_data()
        # Then emit the change with optional positional and keyword arguments
        self.emit('new_data', data=data)

# An observer - could inherit from Dispatcher or any other class
class MyListener(object):
    def on_new_data(self, *args, **kwargs):
        data = kwargs.get('data')
        print('I got data: {}'.format(data))
    def on_emitter_state(self, *args, **kwargs):
        print('emitter state changed')

emitter = MyEmitter()
listener = MyListener()

emitter.bind(on_state=listener.on_emitter_state)
emitter.bind(new_data=listener.on_new_data)

emitter.do_some_stuff()
# >>> I got data: ...

emitter.emit('on_state')
# >>> emitter state changed
```

1.4.2 Properties

```
from pydispatch import Dispatcher, Property

class MyEmitter(Dispatcher):
    # Property objects are defined and named at the class level.
    # They will become instance attributes that will emit events when their values␣
↪change
    name = Property()
    value = Property()

class MyListener(object):
    def on_name(self, instance, value, **kwargs):
        print('emitter name is {}'.format(value))
    def on_value(self, instance, value, **kwargs):
        print('emitter value is {}'.format(value))

emitter = MyEmitter()
listener = MyListener()
```

(continues on next page)

(continued from previous page)

```
emitter.bind(name=listener.on_name, value=listener.on_value)

emitter.name = 'foo'
# >>> emitter name is foo
emitter.value = 42
# >>> emitter value is 42
```

1.5 Contributing

Contributions are welcome!

If you want to contribute through code or documentation, please see the Contributing Guide for information.

1.6 License

This project is released under the MIT License. See the LICENSE file for more information.

1.6.1 Overview

Dispatcher

Dispatcher is the core component of the framework. Subclassing this enables all of the event functionality.

Usage

```
from pydispatch import Dispatcher

class MyEmitter(Dispatcher):
    _events_ = ['on_state', 'new_data']
    def do_some_stuff(self):
        # do stuff that makes new data
        data = self.get_some_data()
        self.emit('new_data', data=data)

# An observer - could inherit from Dispatcher or any other class
class MyListener(object):
    def on_new_data(self, *args, **kwargs):
        data = kwargs.get('data')
        print('I got data: {}'.format(data))
    def on_emitter_state(self, *args, **kwargs):
        print('emitter state changed')

emitter = MyEmitter()
listener = MyListener()

emitter.bind(on_state=listener.on_emitter_state)
emitter.bind(new_data=listener.on_new_data)
```

(continues on next page)

(continued from previous page)

```
emitter.do_some_stuff()
# >>> I got data: ...

emitter.emit('on_state')
# >>> emitter state changed
```

The *bind* method above could also be combined:

```
emitter.bind(on_state=listener.on_emitter_state,
             new_data=listener.on_new_data)
```

Events can also be created after object creation:

```
emitter.register_event('need_data')

# Multiple events can also be created:
emitter.register_event('value_changed', 'something_happened')
```

Stop listening by calling *unbind*:

```
emitter.unbind(listener.on_emitter_state)

# Or to unbind all events, just supply the instance object:
emitter.unbind(listener)
```

Callback Return Values

Event propagation will stop if any callback returns `False`. Any other return value is ignored.

Event Names

There are no restrictions on event names. The idea is to keep things as simple and non-restrictive as possible. When calling *emit*, and positional or keyword arguments supplied will be passed along to listeners.

Subclasses and `__init__`

The *Dispatcher* class does not use `__init__` for any of its functionality. This is again to keep things simple and get the framework out of your way. It uses `__new__` to handle instance creation. If your subclasses use `__new__` for something, the call to `super()` is required, but you should probably check the code to determine how it fits with your own.

Properties

Property objects can be defined on subclasses of *Dispatcher* to create instance attributes that emit events when their values change. Binding and unbinding works exactly the same as with events. The callback signature is slightly different however. The first two arguments will be:

1. The instance object that generated the event
2. The Property value

Usage

```
from pydispatch import Dispatcher, Property

class MyEmitter(Dispatcher):
    name = Property()
    value = Property()

class MyListener(object):
    def on_name(self, instance, value, **kwargs):
        print('emitter name is {}'.format(value))
    def on_value(self, instance, value, **kwargs):
        print('emitter value is {}'.format(value))

emitter = MyEmitter()
listener = MyListener()

emitter.bind(name=listener.on_name, value=listener.on_value)

emitter.name = 'foo'
# >>> emitter name is foo
emitter.value = 42
# >>> emitter value is 42
```

If the attribute is set to the same value, an event is not dispatched:

```
emitter.value = 42
# No event
emitter.value = 43
# >>> emitter value is 43
```

Container Properties

`dict` and `list` objects are implemented as subclasses of *`Property`*:

- *`DictProperty`*
- *`ListProperty`*

They will emit events when their contents change. Nesting is also supported, so even the contents of a `list` or `dict` anywhere inside of the structure can trigger an event.

```
from pydispatch import Dispatcher, ListProperty, DictProperty

class MyEmitter(Dispatcher):
    values = ListProperty()
    data = DictProperty()

emitter = MyEmitter()

emitter.values.append('foo')
print(emitter.values)
# >>> ['foo']

emitter.values.extend(['bar', 'baz'])
print(emitter.values)
```

(continues on next page)

(continued from previous page)

```
# >>> ['foo', 'bar', 'baz']

emitter.data = {'foo':'bar'}
# or
emitter.data['foo'] = 'bar'
print(emitter.data)
# >>> {'foo':'bar'}

emitter.data['fruit'] = {'apple':'red'}
emitter.data['fruit']['banana'] = 'yellow'
# event would be dispatched to listeners
```

asyncio support

New in version 0.1.0.

Callbacks

Callbacks can also be coroutine functions (defined using `async def` or decorated with `@asyncio.coroutine`). For this to work properly, the `event loop` they belong to **must** be specified. This requirement is in place to prevent issues when multiple event loops are in use (via threads, etc).

The loop can be specified using the *`Dispatcher.bind_async`* method, or passed as a keyword argument to *`Dispatcher.bind`*.

Examples

bind_async method

```
import asyncio
from pydispatch import Dispatcher

class MyEmitter(Dispatcher):
    events = ['on_state']

class MyAsyncListener(object):
    def __init__(self):
        self.event_received = asyncio.Event()
    async def on_emitter_state(self, *args, **kwargs):
        self.event_received.set()

loop = asyncio.get_event_loop()
emitter = MyEmitter()
listener = MyAsyncListener()

# Pass the event loop as first argument to "bind_async"
emitter.bind_async(loop, on_state=listener.on_emitter_state)

async def trigger_and_wait_for_event():
    await asyncio.sleep(1)
```

(continues on next page)

(continued from previous page)

```
    emitter.emit('on_state')

    # listener.event_received should be set from "on_emitter_state"
    await listener.event_received.wait()

loop.run_until_complete(trigger_and_wait_for_event())
```

bind (with keyword argument)

```
import asyncio
from pydispatch import Dispatcher

class MyEmitter(Dispatcher):
    events = ['on_state']

class MyAsyncListener(object):
    def __init__(self):
        self.event_received = asyncio.Event()
    async def on_emitter_state(self, *args, **kwargs):
        self.event_received.set()

loop = asyncio.get_event_loop()
emitter = MyEmitter()
listener = MyAsyncListener()

# Pass the event loop using __aio_loop__
emitter.bind(on_state=listener.on_emitter_state, __aio_loop__=loop)

async def trigger_and_wait_for_event():
    await asyncio.sleep(1)

    emitter.emit('on_state')

    # listener.event_received should be set from "on_emitter_state"
    await listener.event_received.wait()

loop.run_until_complete(trigger_and_wait_for_event())
```

Async (awaitable) Events

Event (and *Property*) objects are awaitable. This allows event subscription without callbacks in an async environment. The *Event* instance itself must first be obtained using the *Dispatcher.get_dispatcher_event* method. Any positional and keyword arguments from the event are returned as a two-tuple:

```
async def wait_for_event(event_name):
    event = emitter.get_dispatcher_event(event_name)
    args, kwargs = await event
    return args, kwargs

loop.run_until_complete(wait_for_event('on_state'))
```

This can also be done with *Property* objects:

```
import asyncio
from pydispatch import Dispatcher, Property


class MyEmitter(Dispatcher):
    value = Property()
    async def change_values(self):
        for i in range(10):
            await asyncio.sleep(.1)
            self.value = i

class MyAsyncListener(object):
    async def wait_for_value(self, emitter):
        event = emitter.get_dispatcher_event('value')
        while True:
            args, kwargs = await event
            instance, value = args
            print(value)
            if value >= 9:
                break

loop = asyncio.get_event_loop()
emitter = MyEmitter()
listener = MyAsyncListener()

# Make the emitter value iterate from 0-9
asyncio.ensure_future(emitter.change_values())

# Listens for changes, then exits after value reaches 9
loop.run_until_complete(listener.wait_for_value(emitter))
```

1.6.2 Reference

pydispatch.dispatch module

Dispatcher class

class Dispatcher(**args*, ***kwargs*)

Core class used to enable all functionality in the library

Interfaces with *Event* and *Property* objects upon instance creation.

Events can be created by calling *register_event()* or by the subclass definition:

```
class Foo(Dispatcher):
    _events_ = ['awesome_event', 'on_less_awesome_event']
```

Once defined, an event can be dispatched to listeners by calling *emit()*.

bind(***kwargs*)

Subscribes to events or to *Property* updates

Keyword arguments are used with the Event or Property names as keys and the callbacks as values:

```
class Foo(Dispatcher):
    name = Property()
```

(continues on next page)

(continued from previous page)

```
foo = Foo()

foo.bind(name=my_listener.on_foo_name_changed)
foo.bind(name=other_listener.on_name,
         value=other_listener.on_value)
```

The callbacks are stored as weak references and their order is not maintained relative to the order of binding.

Async Callbacks:

Callbacks may be coroutine functions (defined using `async def` or decorated with `@asyncio.coroutine`), but an event loop must be explicitly provided with the keyword argument `"__aio_loop__"` (an instance of `asyncio.BaseEventLoop`):

```
import asyncio
from pydispatch import Dispatcher

class Foo(Dispatcher):
    _events_ = ['test_event']

class Bar(object):
    def __init__(self):
        self.got_foo_event = asyncio.Event()
    async def wait_for_foo(self):
        await self.got_foo_event.wait()
        print('got foo!')
    async def on_foo_test_event(self, *args, **kwargs):
        self.got_foo_event.set()

foo = Foo()
bar = Bar()

loop = asyncio.get_event_loop()
foo.bind(test_event=bar.on_foo_test_event, __aio_loop__=loop)

loop.run_until_complete(bar.wait_for_foo())
```

This can also be done using *bind_async()*.

New in version 0.1.0.

bind_async(*loop*, ***kwargs*)

Subscribes to events with async callbacks

Functionality is matches the *bind()* method, except the provided callbacks should be coroutine functions. When the event is dispatched, callbacks will be placed on the given event loop.

For keyword arguments, see *bind()*.

Parameters **loop** – The `EventLoop` to use when events are dispatched

Availability: Python>=3.5

New in version 0.1.0.

emission_lock(*name*)

Holds emission of events and dispatches the last event on release

The context manager returned will store the last event data called by *emit()* and prevent callbacks until it exits. On exit, it will dispatch the last event captured (if any):

```
class Foo(Dispatcher):
    _events_ = ['my_event']

def on_my_event(value):
    print(value)

foo = Foo()
foo.bind(my_event=on_my_event)

with foo.emission_lock('my_event'):
    foo.emit('my_event', 1)
    foo.emit('my_event', 2)

>>> 2
```

Parameters **name** (*str*) – The name of the *Event* or *Property*

Returns

A context manager to be used by the `with` statement.

If available, this will also be an async context manager to be used with the `async with` statement (see PEP 492).

Note: The context manager is re-entrant, meaning that multiple calls to this method within nested context scopes are possible.

---

emit (*name*, **args*, ***kwargs*)
Dispatches an event to any subscribed listeners

Note: If a listener returns `False`, the event will stop dispatching to other listeners. Any other return value is ignored.

Parameters

- **name** (*str*) – The name of the *Event* to dispatch
- ***args** (*Optional*) – Positional arguments to be sent to listeners
- ****kwargs** (*Optional*) – Keyword arguments to be sent to listeners

get_dispatcher_event (*name*)
Retrieves an Event object by name

Parameters **name** (*str*) – The name of the *Event* or *Property* object to retrieve

Returns The *Event* instance for the event or property definition

New in version 0.1.0.

register_event (**names*)
Registers new events after instance creation

Parameters ***names** (*str*) – Name or names of the events to register

unbind(**args*)
Unsubscribes from events or *Property* updates

Multiple arguments can be given. Each of which can be either the method that was used for the original call to *bind()* or an instance object.

If an instance of an object is supplied, any previously bound Events and Properties will be 'unbound'.

Event class

class Event(*name*)
Holds references to event names and subscribed listeners

This is used internally by *Dispatcher*.

__call__(**args*, ***kwargs*)
Dispatches the event to listeners

Called by *emit()*

pydispatch.properties module

Property objects can be defined on subclasses of *Dispatcher* to create instance attributes that act as events when their values change:

```
from pydispatch import Dispatcher, Property

class Foo(Dispatcher):
    name = Property()
    value = Property()
def __str__(self):
    return self.__class__.__name__

class Listener(object):
    def on_foo_name(self, instance, value, **kwargs):
        print("{}'s name is {}".format(instance, value))
    def on_foo_value(self, instance, value, **kwargs):
        print('{} = {}'.format(instance, value))

foo_obj = Foo()
listener_obj = Listener()

foo_obj.bind(name=listener_obj.on_foo_name, value=listener_obj.on_foo_value)

foo_obj.name = 'bar'
# Foo's name is bar

foo_obj.value = 42
# Foo = 42
```

Type checking is not enforced, so values can be any valid python type. Values are however checked for equality to avoid dispatching events for no reason. If custom objects are used as values, they must be able to support equality checking. In most cases, this will be handled automatically.

Property class

class Property(*default=None*)
Defined on the class level to create an observable attribute

Parameters **default** (*Optional*) – If supplied, this will be the default value of the Property for all instances of the class. Otherwise None

name
The name of the Property as defined in the class definition. This will match the attribute name for the *Dispatcher* instance.

Type str

_on_change(*obj*, *old*, *value*, ***kwargs*)
Called internally to emit changes from the instance object

The keyword arguments here will be passed to callbacks through the instance object's *emit()* method.

Keyword Arguments

- **property** – The *Property* instance. This is useful if multiple properties are bound to the same callback. The attribute name
- **keys** (*optional*) – If the *Property* is a container type (*ListProperty* or *DictProperty*), the changes may be found here. This is not implemented for nested containers and will only be available for operations that do not alter the size of the container.

ListProperty class

class ListProperty(*default=None*, *copy_on_change=False*)
Bases: *pydispatch.properties.Property*

Property with a list type value

Parameters

- **default** (*Optional*) – If supplied, this will be the default value of the Property for all instances of the class. Otherwise None
- **copy_on_change** (*bool, optional*) – If True, the list will be copied when contents are modified. This can be useful for observing the original state of the list from within callbacks. The copied (original) state will be available from the keyword argument 'old'. The default is False (for performance and memory reasons).

Changes to the contents of the list are able to be observed through *ObservableList*.

DictProperty class

class DictProperty(*default=None*, *copy_on_change=False*)
Bases: *pydispatch.properties.Property*

Property with a dict type value

Parameters

- **default** (*Optional*) – If supplied, this will be the default value of the Property for all instances of the class. Otherwise None

- **copy_on_change** (*bool, optional*) – If `True`, the dict will be copied when contents are modified. This can be useful for observing the original state of the dict from within callbacks. The copied (original) state will be available from the keyword argument 'old'. The default is `False` (for performance and memory reasons).

Changes to the contents of the dict are able to be observed through *ObservableDict*.

Observable class

class Observable

Mixin used by *ObservableList* and *ObservableDict* to emit changes and build other observables

When an item is added to an observable container (a subclass of Observable) it is type-checked and, if possible replaced by an observable version of it.

In other words, if a dict is added to a *ObservableDict*, it is copied and replaced by another *ObservableDict*. This allows nested containers to be observed and their changes to be tracked.

ObservableList class

class ObservableList(*initlist=None*, ***kwargs*)

Bases: `list`, *pydispatch.properties.Observable*

A `list` subclass that tracks changes to its contents

Note: This class is for internal use and not intended to be used directly

ObservableDict class

class ObservableDict(*initdict=None*, ***kwargs*)

Bases: `dict`, *pydispatch.properties.Observable*

A `dict` subclass that tracks changes to its contents

Note: This class is for internal use and not intended to be used directly

pydispatch.utils module

WeakMethodContainer class

class WeakMethodContainer(***kw*)

Container to store weak references to callbacks

Instance methods are stored using the underlying function object and the instance id (using `id(obj)`) as the key (a two-tuple) and the object itself as the value. This ensures proper weak referencing.

Functions are stored using the string "function" and the id of the function as the key (a two-tuple).

add_method(*m*, ***kwargs*)

Add an instance method or function

Parameters **m** – The instance method or function to store

del_instance(*obj*)
Remove any stored instance methods that belong to an object

Parameters **obj** – The instance object to remove

del_method(*m*)
Remove an instance method or function if it exists

Parameters **m** – The instance method or function to remove

iter_instances()
Iterate over the stored objects

Yields *wrkey* – The two-tuple key used to store the object obj: The instance or function object

iter_methods()
Iterate over stored functions and instance methods

Yields Instance methods or function objects

keys() → a set-like object providing a view on D's keys

InformativeWVDict class

class InformativeWVDict(***kwargs*)
Bases: weakref.WeakValueDictionary

A WeakValueDictionary providing a callback for deletion

Keyword Arguments **del_callback** – A callback function that will be called when an item is either deleted or dereferenced. It will be called with the key as the only argument.

EmissionHoldLock class

class EmissionHoldLock(*event_instance*)
Bases: *pydispatch.utils.EmissionHoldLock_*, *pydispatch.aioutils.AioEmissionHoldLock*

EmissionHoldLock_ class

class EmissionHoldLock_(*event_instance*)
Context manager used for *pydispatch.dispatch.Dispatcher.emission_lock()*

Parameters **event_instance** – The *Event* instance associated with the lock

event_instance
The *Event* instance associated with the lock

last_event
The positional and keyword arguments from the event's last emission as a two-tuple. If no events were triggered while the lock was held, None.

held
The internal state of the lock

Type bool

AioEmissionHoldLock class

class AioEmissionHoldLock
Bases: object

Async context manager mixin for *pydispatch.utils.EmissionHoldLock_*

Supports use in `async with` statements

pydispatch.aioutils module

AioWeakMethodContainer class

class AioWeakMethodContainer
Bases: *pydispatch.utils.WeakMethodContainer*

Storage for coroutine functions as weak references

New in version 0.1.0.

__call__(**args*, ***kwargs*)
Triggers all stored callbacks (coroutines)

Parameters

- ***args** – Positional arguments to pass to callbacks
- ****kwargs** – Keyword arguments to pass to callbacks

add_method(*loop*, *callback*)
Add a coroutine function

Parameters

- **loop** – The `event loop` instance on which to schedule callbacks
- **callback** – The coroutine function to add

iter_instances()
Iterate over the stored objects

See also:

pydispatch.utils.WeakMethodContainer.iter_instances()

iter_methods()
Iterate over stored coroutine functions

Yields Stored coroutine function objects

See also:

pydispatch.utils.WeakMethodContainer.iter_instances()

submit_coroutine(*coro*, *loop*)
Schedule and await a coroutine on the specified loop

The coroutine is wrapped and scheduled using `asyncio.run_coroutine_threadsafe()`. While the coroutine is "awaited", the result is not available as method returns immediately.

Parameters

- **coro** – The coroutine to schedule
- **loop** – The `event loop` on which to schedule the coroutine

Note: This method is used internally by *__call__()* and is not meant to be called directly.

AioEventWaiters class

class AioEventWaiters
Container used to manage `await` use with events

Used by *pydispatch.dispatch.Event* when it is `awaited`

waiters
Instances of *AioEventWaiter* currently "awaiting" the event

Type set

lock
A sync/async lock to guard modification to the *waiters* container during event emission

Type *AioSimpleLock*

New in version 0.1.0.

__call__(**args*, ***kwargs*)
Triggers any stored *waiters*

Calls *AioEventWaiter.trigger()* method on all instances stored in *waiters*. After completion, the *waiters* are removed.

Parameters

- ***args** – Positional arguments to pass to *AioEventWaiter.trigger()*
- ****kwargs** – Keyword arguments to pass to *AioEventWaiter.trigger()*

add_waiter()
Add a *AioEventWaiter* to the *waiters* container

The event loop to use for *AioEventWaiter.loop* is found in the current context using `asyncio.get_event_loop()`

Returns The created *AioEventWaiter* instance

Return type waiter

wait()
Creates a *waiter* and "awaits" its result

This method is used by *pydispatch.dispatch.Event* instances when they are "awaited" and is the primary functionality of *AioEventWaiters* and *AioEventWaiter*.

Returns Positional arguments attached to the event kwargs (dict): Keyword arguments attached to the event

Return type args (list)

AioEventWaiter class

class AioEventWaiter(*loop*)
Stores necessary information for a single "waiter"

Used by *AioEventWaiters* to handle `awaiting` an *Event* on a specific `event loop`

loop
The `EventLoop` instance

aio_event
An `asyncio.Event` used to track event emission

args
The positional arguments attached to the event

Type list

kwargs
The keyword arguments attached to the event

Type dict

New in version 0.1.0.

trigger(**args*, ***kwargs*)
Called on event emission and notifies the *`wait()`* method

Called by *`AioEventWaiters`* when the *`Event`* instance is dispatched.

Positional and keyword arguments are stored as instance attributes for use in the *`wait()`* method and *`aio_event`* is set.

wait()
Waits for event emission and returns the event parameters

Returns Positional arguments attached to the event kwargs (dict): Keyword arguments attached to the event

Return type args (list)

AioSimpleLock class

class AioSimpleLock
`asyncio.Lock` alternative backed by a `threading.Lock`

This is a context manager that supports use in both `with` and `async with` context blocks.

lock
Instance of `threading.Lock`

New in version 0.1.0.

acquire(*blocking=True*, *timeout=-1*)
Acquire the *`lock`*

Parameters

- **blocking** (*bool*) – See `threading.Lock.acquire()`
- **timeout** (*float*) – See `threading.Lock.acquire()`

Returns `True` if the lock was acquired, otherwise `False`

Return type bool

acquire_async()
Acquire the *`lock`* asynchronously

release()
Release the *`lock`*

Python Module Index

p

Index

Symbols

A

B

D

E

G

H

I

K

L

N

O

P

R

S

T

U

W