Spectate Documentation

Release 1.0.0

Ryan Morshead

Jan 15, 2021

CONTENTS

A library that can track changes to mutable data types. With Spectate complicated protocols for managing updates, don't need to be the outward responsibility of a user, and can instead be done automagically in the background.

CHAPTER

ONE

INSTALL

Install `spectate` with pip:

```
pip install spectate
```

1.1 Development

If you'd like to work with the source code, then clone the repository from github:

```
git clone git@github.com:rmorshea/spectate.git && cd spectate
```

And do an editable install with pip that includes `requirements.txt`:

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

CHAPTER

TWO

USAGE

2.1 The Basics

Spectate defines three main constructs:

1. *models* - objects which get modified by the user.
2. *views* - functions which receives change events.
3. *controls* - private attributes of a model which produces change events.

Since the `mvc` module already provides some basic models for us you don't need to worry about *controls* yet. Let's begin by considering a builtin *Dict* model. We can instantiate this object just as we would with a standard `dict`:

```
from spectate import mvc

d = mvc.Dict(a=0)
```

Now though, we can now register a *view()* function with a decorator. This view function is called any time a change is made to the model `d` that causes its data to be mutated.

```
@mvc.view(d)  # <----- pass `d` in the decorator to observe its changes
def printer(
    model,  # <------- The model which experienced an event
    events,  #  <----- A tuple of event dictionaries
):
    print("model:", model)
    for e in events:
        print("event:", e)
```

Change events are passed into this function as a tuple of immutable dict-like objects containing change information. Each model has its own change event information. In the case of a *Dict* the event objects have the fields `key`, `old`, and `new`. So when we change a key in `d` we'll find that our `printer` view function is called and that it prints out an event object with the expected information:

```
d["a"] = 1
```

```
model: {'a': 1}
event: {'key': 'a', 'old': 0, 'new': 1}
```

In cases where a mutation would result in changes to multiple change, one or more event objects can be broadcast to the view function:

```
d.update(b=2, c=3)
```

```
model: {'a': 1, 'b': 2, 'c': 3}
event: {'key': 'b', 'old': Undefined, 'new': 2}
event: {'key': 'c', 'old': Undefined, 'new': 3}
```

2.1.1 Nesting Models

What if we want to observe changes to nested data structures though? Thankfuly all of Spectate's *Builtin Model Types* that inherit from `Structure` can handle this automatically whenevener another model is placed inside another:

```
from spectate import mvc

outer_dict = mvc.Dict()
inner_dict = mvc.Dict()

mvc.view(outer_dict, printer)

outer_dict["x"] = inner_dict
inner_dict["y"] = 1
```

```
model: {'x': {}}
event: {'key': 'x', 'old': Undefined, 'new': {}}
model: {'y': 1}
event: {'key': 'y', 'old': Undefined, 'new': 1}
```

This works just as well if you mix data types too:

```
from spectate import mvc

outer_dict = mvc.Dict()
middle_list = mvc.List()
inner_obj = mvc.Object()

mvc.view(outer_dict, printer)

outer_dict["x"] = middle_list
middle_list.append(inner_obj)
inner_obj.y = 1
```

```
model: {'x': []}
event: {'key': 'x', 'old': Undefined, 'new': []}
model: [<spectate.models.Object object at 0x7f8041ae9550>]
event: {'index': 0, 'old': Undefined, 'new': <spectate.models.Object object at␣
↪0x7f8041ae9550>}
model: <spectate.models.Object object at 0x7f8041ae9550>
event: {'attr': 'y', 'old': Undefined, 'new': 1}
```

However, note that events on nested data structures don't carry information about the location of the notifying model. For this you'll need to implement a *Custom Models* and add this information to the events manually.

2.1.2 Custom Models

To create a custom model all you have to do is inherit from *Model* and broadcast events with a *notifier()*. To get the idea across, lets implement a simple counter object that notifies when a value is incremented or decremented.

```
from spectate import mvc

class Counter(mvc.Model):

    def __init__(self):
        self.value = 0

    def increment(self):
        self.value += 1
        with mvc.notifier(self) as notify:
            notify(new=self.value)

    def decrement(self):
        self.value -= 1
        with mvc.notifier(self) as notify:
            notify(new=self.value)

counter = Counter()

@mvc.view(counter)
def printer(model, events):
    for e in events:
        print(e)

counter.increment()
counter.increment()
counter.decrement()
```

```
{'new': 1}
{'new': 2}
{'new': 1}
```

To share or unshare the view functions between two models using the *link()* and *unlink()* functions respectively. This is especially useful when creating nested data structures. For example we can use it to create an observable binary tree:

```
class Node(mvc.Model):

    def __init__(self, data, parent=None):
        if parent is not None:
            mvc.link(parent, self)
        self.parent = parent
        self.left = None
        self.right = None
        self.data = data

    def add(self, data):
        if data <= self.data:
            if self.left is None:
                self.left = Node(data, self)
                with mvc.notifier(self) as notify:
                    notify(left=self.left, path=self.path())
```

(continues on next page)

(continued from previous page)

```
            else:
                self.left.add(data)
        else:
            if self.right is None:
                self.right = Node(data, self)
                with mvc.notifier(self) as notify:
                    notify(right=self.right, path=self.path())
            else:
                self.right.add(data)

    def path(self):
        n = self
        path = []
        while n is not None:
            path.insert(0, n)
            n = n.parent
        return path

    def __repr__(self):
        return f"Node({self.data})"

root = Node(0)

mvc.view(root, printer)

root.add(1)
root.add(0)
root.add(5)
root.add(2)
root.add(4)
root.add(3)
```

```
model: Node(0)
event: {'right': Node(1), 'path': [Node(0)]}
model: Node(0)
event: {'left': Node(0), 'path': [Node(0)]}
model: Node(1)
event: {'right': Node(5), 'path': [Node(0), Node(1)]}
model: Node(5)
event: {'left': Node(2), 'path': [Node(0), Node(1), Node(5)]}
model: Node(2)
event: {'right': Node(4), 'path': [Node(0), Node(1), Node(5), Node(2)]}
model: Node(4)
event: {'left': Node(3), 'path': [Node(0), Node(1), Node(5), Node(2), Node(4)]}
```

2.2 Handling Events

Spectate provides a series of context managers which allow you to capture and then modify events before they are distributed to views. This allows you to *hold*, *rollback*, and even *mute* events. These context managers are useful for handling edge cases in your code, improving performance by *merging* events, or *undo* unwanted changes.

2.2.1 Holding Events

It's often useful to withhold sending notifications until all your changes are complete. Using the `hold()` context manager, events created when modifying a model won't be distributed until we exit the context:

```
d = mvc.Dict()

# effectively the same as the printer view above
mvc.view(d, lambda d, e: list(map(print, e)))

print("before")
with mvc.hold(d):
    d["a"] = 1
    print("during")
# notifications are sent upon exiting
print("after")
```

```
before
during
{'key': 'a', 'old': Undefined, 'new': 1}
after
```

Merging Events

Sometimes there is a block of code in which it's possible to produce duplicate events or events which could be merged into one. By passing in a `reducer` to `hold()` you can change the list of events just before they are distributed. This is done by having the `reducer` return or yield the new events.

```
from spectate import mvc

d = mvc.Dict()

mvc.view(d, lambda _, es: list(map(print, es)))

def merge_dict_events(model, events):
    changes = {}

    for e in events:
        if e.key in changes:
            changes[e.key][1] = e.new
        else:
            changes[e.key] = [e.old, e.new]

    for key, (old, new) in changes.items():
        yield {"key": key, "new": new, "old": old}

with mvc.hold(d, reducer=merge_dict_events):
```

(continues on next page)

(continued from previous page)

```
    for i in range(5):
        # this loop would normally produce 5 different events
        d["a"] = i
```

```
{'key': 'a', 'new': Undefined, 'old': 4}
```

2.2.2 Rolling Back Events

When an error occurs while modifying a model you may not want to distribute events. Using *rollback()* you can suppress events that were produced in the same context as an error:

```
from spectate import mvc

d = mvc.Dict()

@mvc.view(d)
def should_not_be_called(d, events):
    # we never call this view
    assert False

try:
    with mvc.rollback(d):
        d["a"] = 1
        d["b"]  # key doesn't exist
except KeyError:
    pass
```

Rolling Back Changes

Suppressing events after an error may not be enough. You can pass *rollback()* an `undo` function which gives you a chances to analyze the events in order to determine and then return a model to its original state. Any events that you might produce while modifying a model within the `undo` function will be *muted*.

```
d = mvc.Dict()

def undo_dict_changes(model, events, error):
    seen = set()
    for e in reversed(events):
        if e.old is mvc.Undefined:
            del model[e.key]
        else:
            model[e.key] = e.old

try:
    with mvc.rollback(d, undo=undo_dict_changes):
        d["a"] = 1
        d["b"] = 2
        print(d)
        d["c"]
except KeyError:
    pass
print(d)
```

```
{'a': 1, 'b': 2}
{}
```

2.2.3 Muting Events

If you are setting a default state, or returning to one, it may be useful to withhold events completely. This one's pretty simple compared to the context managers above. Just use *mute()* and within its context, no events will be distributed:

```
from spectate import mvc

l = mvc.List()

@mvc.view(l)
def raises(events):
    # this won't ever happen
    raise ValueError("Events occured!")

with mvc.mute(l):
    l.append(1)
```

2.2.4 Manually Notifying

At times, and more likely when writing tests, you may need to forcefully send an event to a model. This can be achieved using the *notifier()* context manager which provides a `notify()` function identical to the one seen in *Control Callbacks*.

Warning: While you could use *notifier()* instead of adding *Adding Model Controls* to your custom models, this is generall discouraged because the resulting implementation is resistent to extension in subclasses.

```
from spectate import mvc

m = mvc.Model()

@mvc.view(m)
def printer(m, events):
    for e in events:
        print(e)

with mvc.notifier(m) as notify:
    # the view should print out this event
    notify(x=1, y=2)
```

2.3 Builtin Model Types

Spectate provides a number of builtin model types that you can use out of the box. For most users these built-in types should be enough, however if you're adventurous, then you can create your own *Custom Models*.

2.3.1 Dictionary

The *Dict* model is a subclass of Python's standard `dict`. This will produce events when the value of a key in the dictionary changes or is deleted. This will result when calling methods like `dict.update()` and `dict.pop()`, but also when using the normal syntax to set or delete an item. Events produced by *Dict* have the following fields:

Field	Description
`key`	The key in the dict model that changed.
`old`	• The value that was present in the key before the change • Is `Undefined` if the index was not present.
`new`	• The value that this is now present after the change • Is `Undefined` if the index was deleted.

2.3.2 List

The *List* model is a subclass of Python's standard `list`. This model will produce events when an element of the list changes or an element changes from one position to another. This may happen when calling methods like `list.append()` or `list.remove()`, but also when using the normal syntax to set or delete an item. Events produced by *List* have the following keys:

Field	Description
`index`	The index in the dict model that changed.
`old`	• The value that was present before the change • Is `Undefined` if the key was not present.
`new`	• The value that this is now present after the change • Is `Undefined` if the key was deleted.

2.3.3 Set

The *Set* model is a subclass of Python's standard `set`. This model will produce events when an element of the set changes. This may happen when calling methods like `set.add()` or `set.discard()`. Events produced by *Set* have the following keys:

Field	Description
`old`	A set of values that were removed due to the change.
`new`	A set of the values that were added due to the change.

2.3.4 Object

The *Object* model is a subclass of Python's standard `object`. This model will produce events when an attribute of the object changes or is deleted. This may happen when using `setattr()` or `delattr()`, but also when using the normal syntax to set or delete attributes. Events produced by *Object* have the following keys:

Field	Description
`attr`	The attribute in the model that changed.
`old`	• The value that was present before the change • Is `Undefined` if the attribute was not present.
`new`	• The value that this is now present after the change • Is `Undefined` if the key was deleted.

2.4 Spectating Other Types

In a prior example demonstrating how to create *a custom model* we used *notifier()* to produce events. This is sufficient in most cases, but sometimes you aren't able to manually trigger events from within a method. This might occur when inheriting from a builtin type (e.g. `list`, `dict`, etc) that is implemented in C or a third party package that doesn't use `spectate`. In those cases, you must wrap an existing method and are religated to producing events before and/or after it gets called.

In these scenarios you must define a *Model* subclass which has *Control* objects assigned to it. Each control object is responsible for observing calls to particular methods on the model class. For example, if you wanted to know when an element was appended to a list you might observe the `append` method.

To show how this works we will implement a simple counter with the goal of knowing when the value in the counter has incremented or decremented. To get started we should create a `Counter` class which inherits from *Model* and define its `increment` and `decrement` methods normally:

Note: Usually if you're using *Control* objects you'd do it with multiple inheritance, but to keep things simple we aren't doing that in the following examples.

```
from spectate import mvc

class Counter(mvc.Model):

    def __init__(self):
        self.value = 0

    def increment(self, amount):
        self.value += amount

    def decrement(self, amount):
        self.value -= amount
```

```
c = Counter()
c.increment(1)
c.increment(1)
```

(continues on next page)

(continued from previous page)

```
c.decrement(1)
assert c.value == 1
```

2.4.1 Adding Model Controls

Because we know that the value within the `Counter` changes whenever `increment` or `decrement` is called these are the methods that we must observe in order to determine whether, and by how much it changes. Do do this we should add a *`Control`* to the `Counter` and pass in the names of the methods it should be tracking.

```
from spectate import mvc

class Counter(mvc.Model):

    def __init__(self):
        self.value = 0

    def increment(self, amount):
        self.value += amount

    def decrement(self, amount):
        self.value -= amount

    _control_change = mvc.Control('increment', 'decrement')
```

We define the behavior of `_control_change` with methods that are triggered before and/or after the ones being observed. We register these with `Control.before()` and `Control.after()`. For now our beforeback and afterback will just contain print statements so we can see what they receive when they are called.

```
from spectate import mvc

class Counter(mvc.Model):

    def __init__(self):
        self.value = 0

    def increment(self, amount):
        self.value += amount

    def decrement(self, amount):
        self.value -= amount

    _control_change = mvc.Control(
        ["increment", "decrement"],
        before="_before_change",
        after="_after_change",
    )

    def _before_change(self, call, notify):
        print("BEFORE")
        print(call)
        print(notify)
        print()
        return "result-from-before"

    def _after_change(self, answer, notify):
```

(continues on next page)

(continued from previous page)

```
        print("AFTER")
        print(answer)
        print(notify)
        print()
```

No lets see what happens we can call `increment` or `decrement`:

```
c = Counter()
c.increment(1)
c.decrement(1)
```

```
BEFORE
{'name': 'increment', 'kwargs': {}, 'args': (1,), 'parameters': <function␣
↪BoundControl.before.<locals>.beforeback.<locals>.parameters at 0x7f9ce57e8a60>}
<function BoundControl.before.<locals>.beforeback.<locals>.notify at 0x7f9ce57e89d8>

AFTER
{'before': 'result-from-before', 'name': 'increment'}
<function BoundControl.after.<locals>.afterback.<locals>.notify at 0x7f9ce57e89d8>

BEFORE
{'name': 'decrement', 'kwargs': {}, 'args': (1,), 'parameters': <function␣
↪BoundControl.before.<locals>.beforeback.<locals>.parameters at 0x7f9ce57f2400>}
<function BoundControl.before.<locals>.beforeback.<locals>.notify at 0x7f9ce57e89d8>

AFTER
{'before': 'result-from-before', 'name': 'decrement'}
<function BoundControl.after.<locals>.afterback.<locals>.notify at 0x7f9ce57e89d8>
```

2.4.2 Control Callbacks

The callback pair we registered to our `Counter` when learning how to *define controls*, hereafter referred to as *"beforebacks"* and *"afterbacks"* are how event information is communicated to views. Defining both a beforeback and an afterback is not required, but doing so allows for a beforeback to pass data to its corresponding afterback which in turn makes it possible to compute the difference between the state before and the state after a change takes place:

```
from spectate import mvc

class Counter(mvc.Model):

    def __init__(self):
        self.value = 0

    def increment(self, amount):
        self.value += amount

    def decrement(self, amount):
        self.value -= amount

    _control_change = mvc.Control(
        ["increment", "decrement"],
        before="_before_change",
        after="_after_change",
    )
```

(continues on next page)

(continued from previous page)

```
    def _before_change(self, call, notify):
        amount = call.parameters()["amount"]
        print(f"value will {call['name']} by {amount}")
        old_value = self.value
        return old_value

    def _after_change(self, answer, notify):
        old_value = answer["before"]  # this was returned by `_before_change`
        new_value = self.value
        print(f"the old value was {old_value})
        print(f"the new value is {new_value})
        print(f"the value changed by {new_value - old_value}")
```

Now we can try incrementing and decrementing as before:

```
c = Counter()
c.increment(1)
c.decrement(1)
```

```
value will increment by 1
the old value was 0
the new value is 1
the value changed by 1
value will decrement by 1
the old value was 1
the new value is 0
the value changed by -1
```

2.4.3 Control Event Notifications

We're now able to use *"beforebacks"* and *"afterbacks"* to print out information about a model before and after a change occures, but what we actually want is to send this same information to `views` as we did when we learned *The Basics*. To accomplish this we use the `notify` function passed into the beforeback and afterback and pass it keyword parameters that can be consumed by views. To keep things simple we'll just replace our `print` statements with calls to `notify`:

```
from spectate import mvc

class Counter(mvc.Model):

    def __init__(self):
        self.value = 0

    def increment(self, amount):
        self.value += amount

    def decrement(self, amount):
        self.value -= amount

    _control_change = (
        mvc.Control('increment', 'decrement')
        .before("_before_change")
        .after("_after_change")
```

(continues on next page)

(continued from previous page)

```
    )

    def _before_change(self, call, notify):
        amount = call.parameters()["amount"]
        notify(message="value will %s by %s" % (call["name"], amount))
        old_value = self.value
        return old_value

    def _after_change(self, answer, notify):
        old_value = answer["before"]  # this was returned by `_before_change`
        new_value = self.value
        notify(message="the old value was %r" % old_value)
        notify(message="the new value is %r" % new_value)
        notify(message="the value changed by %r" % (new_value - old_value))
```

To print out the same messages as before we'll need to register a view with out counter:

```
c = Counter()

@mvc.view(c)
def print_messages(c, events):
    for e in events:
        print(e["message"])

c.increment(1)
c.decrement(1)
```

```
value will increment by 1
the old value was 0
the new value is 1
the value changed by 1
value will decrement by 1
the old value was 1
the new value is 0
the value changed by -1
```

2.4.4 Control Beforebacks

Have a signature of `(call, notify) -> before`

- `call` is a `dict` with the keys
  - `'name'` - the name of the method which was called
  - `'args'` - the arguments which that method will call
  - `'kwargs'` - the keywords which tCallbacks are registered to specific methods in pairs - one will be triggered before, and the other after, a call to that method is made. These two callbacks are referred to as "beforebacks" and "afterbacks" respectively. Defining both a beforeback and an afterback in each pair is not required, but doing so allows a beforeback to pass data to its corresponding afterback.
  - `parameters` a function which returns a dictionary where the `args` and `kwargs` passed to the method have been mapped to argument names. This won't work for builtin method like `dict.get()` since they're implemented in C.
- `notify` is a function which will distribute an event to *views*

- `before` is a value which gets passed on to its respective *afterback*.

2.4.5 Control Afterbacks

Have a signature of `(answer, notify)`

- `answer` is a `dict` with the keys
  - `'name'` - the name of the method which was called
  - `'value'` - the value returned by the method
  - `'before'` - the value returned by the respective beforeback
- `notify` is a function which will distribute an event to *views*

2.5 Spectate in Traitlets

The inspiration for Spectate originally came from difficulties encountered while working with mutable data types in IPython's Traitlets. Unfortunately Traitlets does not natively allows you to track changes to mutable data types.

Now though, with Spectate, we can add this functionality to traitlets using a custom `TraitType` that can act as a base class for all mutable traits.

```
from spectate import mvc
from traitlets import TraitType


class Mutable(TraitType):
    """A base class for mutable traits using Spectate"""

    # Overwrite this in a subclass.
    _model_type = None

    # The event type observers must track to spectate changes to the model
    _event_type = "mutation"

    # You can dissallow attribute assignment to avoid discontinuities in the
    # knowledge observers have about the state of the model. Removing the line below
    # will enable attribute assignment and require observers to track 'change'
    # events as well as 'mutation' events in to avoid such discontinuities.
    __set__ = None

    def default(self, obj):
        """Create the initial model instance

        The value returned here will be mutated by users of the HasTraits object
        it is assigned to. The resulting events will be tracked in the ``callback``
        defined below and distributed to event observers.
        """
        model = self._model_type()

        @mvc.view(model)
        def callback(model, events):
            obj.notify_change(
                dict(
```

(continues on next page)

(continued from previous page)

```
                    self._make_change(model, events),
                    name=self.name,
                    type=self._event_type,
                )
            )

        return model

    def _make_change(self, model, events):
        """Construct a dictionary describing the change"""
        raise NotImplementedError()
```

With this in place we can then subclass our base `Mutable` class and use it to create a `MutableDict`:

```
class MutableDict(Mutable):
    """A mutable dictionary trait"""

    _model_type = mvc.Dict

    def _make_change(self, model, events):
        old, new = {}, {}
        for e in events:
            old[e["key"]] = e["old"]
            new[e["key"]] = e["new"]
        return {"value": model, "old": old, "new": new}
```

An example usage of this trait would then look like:

```
from traitlets import HasTraits, observe


class MyObject(HasTraits):
    mutable_dict = MutableDict()

    @observe("mutable_dict", type="mutation")
    def track_mutations_from_method(self, change):
        print("method observer:", change)


def track_mutations_from_function(change):
    print("function observer:", change)


my_object = MyObject()
my_object.observe(track_mutations_from_function, "mutable_dict", type="mutation")


my_object.mutable_dict["x"] = 1
my_object.mutable_dict.update(x=2, y=3)
```

```
method observer: {'old': {'x': Undefined}, 'new': {'x': 1}, 'name': 'mutable_dict',
↪'type': 'mutation'}
function observer: {'old': {'x': Undefined}, 'new': {'x': 1}, 'name': 'mutable_dict',
↪'type': 'mutation'}
method observer: {'old': {'x': 1, 'y': Undefined}, 'new': {'x': 2, 'y': 3}, 'name':
↪'mutable_dict', 'type': 'mutation'}
```

(continues on next page)

(continued from previous page)

```
function observer: {'old': {'x': 1, 'y': Undefined}, 'new': {'x': 2, 'y': 3}, 'name':
↪'mutable_dict', 'type': 'mutation'}
```

CHAPTER

THREE

API

class spectate.models.**Dict**(**args: Any*, ***kwargs: Any*)
A *spectate.mvc* enabled `dict`.

clear() → None. Remove all items from D.

pop(*k*[, *d*]) → v, remove specified key and return the corresponding value.
If key is not found, d is returned if given, otherwise KeyError is raised

setdefault(*key*, *default=None*, */*)
Insert key with a value of default if key is not in the dictionary.

Return the value for key if key is in the dictionary, else default.

update([*E*], ***F*) → None. Update D from dict/iterable E and F.
If E is present and has a .keys() method, then does: for k in E: D[k] = E[k] If E is present and lacks a .keys() method, then does: for k, v in E: D[k] = v In either case, this is followed by: for k in F: D[k] = F[k]

class spectate.models.**List**(**args: Any*, ***kwargs: Any*)
A *spectate.mvc* enabled `list`.

append(*object*, */*)
Append object to the end of the list.

clear()
Remove all items from list.

extend(*iterable*, */*)
Extend list by appending elements from the iterable.

insert(*index*, *object*, */*)
Insert object before index.

pop(*index=-1*, */*)
Remove and return item at index (default last).

Raises IndexError if list is empty or index is out of range.

remove(*value*, */*)
Remove first occurrence of value.

Raises ValueError if the value is not present.

reverse()
Reverse *IN PLACE*.

sort(*, *key=None*, *reverse=False*)
Stable sort *IN PLACE*.

class spectate.models.**Object**(**args: Any*, ***kwargs: Any*)
A spectat.mvc enabled object.

class spectate.models.**Set**(**args: Any*, ***kwargs: Any*)
A *spectate.mvc* enabled set.

add()
Add an element to a set.

This has no effect if the element is already present.

clear()
Remove all elements from this set.

difference_update()
Remove all elements of another set from this set.

discard()
Remove an element from a set if it is a member.

If the element is not a member, do nothing.

intersection_update()
Update a set with the intersection of itself and another.

pop()
Remove and return an arbitrary set element. Raises KeyError if the set is empty.

remove()
Remove an element from a set; it must be a member.

If the element is not a member, raise a KeyError.

symmetric_difference_update()
Update a set with the symmetric difference of itself and another.

update()
Update a set with the union of itself and others.

class spectate.models.**Structure**(**args: Any*, ***kwargs: Any*)

class spectate.base.**Control**(*methods*, ***, *before=None*, *after=None*)
An object used to define control methods on a *Model*

A "control" method on a *Model* is one which reacts to another method being called. For example there is a control method on the List which responds when append() is called.

A control method is a slightly modified *beforeback* or *afterback* that accepts an extra notify argument. These are added to a control object by calling Control.before() or Control.after() respectively. The notify arugment is a function which allows a control method to send messages to *views* that are registered to a *Model*.

Parameters

- **methods** (Union[list, tuple, str]) – The names of the methods on the model which this control will react to When they are calthrough the Nodeled. This is either a comma seperated string, or a list of strings.
- **before** (Union[Callable, str, None]) – A control method that reacts before any of the given methods are called. If given as a callable, then that function will be used as the callback. If given as a string, then the control will look up a method with that name when reacting (useful when subclassing).

- **after** (Union[Callable, str, None]) – A control method that reacts after any of the given methods are alled. If given as a callable, then that function will be used as the callback. If given as a string, then the control will look up a method with that name when reacting (useful when subclassing).

Examples

Control methods are registered to a *Control* with a str or function. A string may refer to the name of a method on a *Model* while a function should be decorated under the same name as the *Control* object to preserve the namespace.

```
from spectate import mvc

class X(mvc.Model):

    _control_method = mvc.Control("method").before("_control_before_method")

    def _control_before_method(self, call, notify):
        print("before")

    # Note how the method uses the same name. It
    # would be redundant to use a different one.
    @_control_a.after
    def _control_method(self, answer, notify):
        print("after")

    def method(self):
        print("during")

x = X()
x.method()
```

```
before
during
after
```

class spectate.base.**Model**(**args: Any*, ***kwargs: Any*)

An object that can be *controlled* and *viewed*.

Users should define *Control* methods and then *view()* the change events those controls emit. This process starts by defining controls on a subclass of *Model*.

Examples

```
from specate import mvc

class Object(Model):

    _control_attr_change = Control(
        "__setattr__, __delattr__",
        before="_control_before_attr_change",
        after="_control_after_attr_change",
    )

    def __init__(self, *args, **kwargs):
```

(continues on next page)

(continued from previous page)

```
        for k, v in dict(*args, **kwargs).items():
            setattr(self, k, v)

    def _control_before_attr_change(self, call, notify):
        return call["args"][0], getattr(self, call["args"][0], Undefined)

    def _control_after_attr_change(self, answer, notify):
        attr, old = answer["before"]
        new = getattr(self, attr, Undefined)
        if new != old:
            notify(attr=attr, old=old, new=new)

o = Object()

@mvc.view(o)
def printer(o, events):
    for e in events:
        print(e)
```

spectate.base.**link**(*source*, **targets*)

Attach all of the source's present and future view functions to the targets.

Parameters

- **source** (*Model*) – The model whose view functions will be attached to the targets.
- **targets** (*Model*) – The models that will acquire the source's view functions.

Return type None

spectate.base.**notifier**(*model*)

Manually send notifications to the given model.

Parameters **model** (*Model*) – The model whose views will recieve notifications

Return type Iterator[Callable[..., None]]

Returns A function whose keyword arguments become event data.

Example

```
m = Model()

@view(m)
def printer(m, events):
    for e in events:
        print(e)

with notifier(m) as notify:
    # the view should print out this event
    notify(x=1, y=2)
```

spectate.base.**unlink**(*source*, **targets*)

Remove all of the source's present and future view functions from the targets.

Parameters

- **source** (*Model*) – The model whose view functions will be removed from the targets.
- **targets** (*Model*) – The models that will no longer share view functions with the source.

Return type None

spectate.base.**unview**(*model*, *function*)
Remove a view callbcak from a model.

Parameters

- **model** (*Model*) – The model which contains the view function.
- **function** (Callable[[*Model*, Tuple[Dict[str, Any], ...]], None]) – The callable which was registered to the model as a view.

Raises **ValueError** – If the given function is not a view of the given model.

Return type None

spectate.base.**view**(*model: Model*) → Callable[[_F], _F]
spectate.base.**view**(*model: Model*, *function: Callable[[Model, Tuple[Dict[str, Any], ...]], None]*) → None
A decorator for registering a callback to a model

Parameters **model** (*Model*) – the model object whose changes the callback should respond to.

Examples

```
from spectate import mvc

items = mvc.List()

@mvc.view(items)
def printer(items, events):
    for e in events:
        print(e)

items.append(1)
```

Return type Optional[Callable[[~_F], ~_F]]

spectate.base.**views**(*model*)
Return a model's views keyed on what events they respond to.

Model views are added by calling *view()* on a model.

Return type List[Callable[[*Model*, Tuple[Dict[str, Any], ...]], None]]

spectate.events.**hold**(*model*, *reducer=None*)
Temporarilly withold change events in a modifiable list.

All changes that are captured within a "hold" context are forwarded to a list which is yielded to the user before being sent to views of the given model. If desired, the user may modify the list of events before the context is left in order to change the events that are ultimately sent to the model's views.

Parameters

- **model** (*Model*) – The model object whose change events will be temporarilly witheld.
- **reducer** (Optional[Callable[[*Model*, List[Dict[str, Any]]], List[Dict[str, Any]]]]) – A function for modifying the events list at the end of the context. Its signature is `(model, events) -> new_events` where model is the given model, events is the complete list of events produced in the context, and the returned new_events is a list of events that will actuall be distributed to views.

Notes

All changes witheld from views will be sent as a single notification. For example if you view a `specate.mvc.models.List` and its `append()` method is called three times within a *`hold()`* context,

Examples

Note how the event from `l.append(1)` is omitted from the printed statements.

```
from spectate import mvc

l = mvc.List()

mvc.view(d, lambda d, e: list(map(print, e)))

with mvc.hold(l) as events:
    l.append(1)
    l.append(2)

    del events[0]
```

```
{'index': 1, 'old': Undefined, 'new': 2}
```

Return type `Iterator`[`List`[`Dict`[`str`, `Any`]]]

`spectate.events.`**`mute`**(*model*)

Block a model's views from being notified.

All changes within a "mute" context will be blocked. No content is yielded to the user as in *`hold()`*, and the views of the model are never notified that changes took place.

Parameters **`mode`** – The model whose change events will be blocked.

Examples

The view is never called due to the *`mute()`* context:

```
from spectate import mvc

l = mvc.List()

@mvc.view(l)
def raises(events):
    raise ValueError("Events occured!")

with mvc.mute(l):
    l.append(1)
```

Return type `Iterator`[`None`]

`spectate.events.`**`rollback`**(*model*, *undo=None*, *reducer=None*)

Withold events if an error occurs.

Generall operate

Parameters

- **model** (*Model*) – The model object whose change events may be witheld.
- **undo** (Optional[Callable[[*Model*, Tuple[Dict[str, Any], ...], Exception], None]]) – An optional function for reversing any changes that may have taken place. Its signature is `(model, events, error)` where `model` is the given model, `events` is a list of all the events that took place, and `error` is the exception that was riased. Any changes that you make to the model within this function will not produce events.

Examples

Simple supression of events:

```
from spectate import mvc

d = mvc.Dict()

@mvc.view(d)
def should_not_be_called(d, events):
    # we never call this view
    assert False

try:
    with mvc.rollback(d):
        d["a"] = 1
        d["b"]  # key doesn't exist
except KeyError:
    pass
```

Undo changes for a dictionary:

```
from spectate import mvc

def undo_dict_changes(model, events, error):
    seen = set()
    for e in reversed(events):
        if e.old is mvc.Undefined:
            del model[e.key]
        else:
            model[e.key] = e.old

try:
    with mvc.rollback(d, undo=undo_dict_changes):
        d["a"] = 1
        d["b"] = 2
        print(d)
        d["c"]
except KeyError:
    pass
print(d)
```

```
{'a': 1, 'b': 2}
{}
```

Return type Iterator[None]

A modules which exports specate's Model-View-Controller utilities in a common namespace

For more info:

- *spectate.base*
- *spectate.events*
- *spectate.models*

CHAPTER

FOUR

AT A GLANCE

If you're using Python 3.6 and above, create a *spectate.mvc* object

```
from spectate import mvc

l = mvc.List()
```

Register a view function to it that observes changes

```
@mvc.view(l)
def printer(l, events):
    for e in events:
        print(e)
```

Then modify your object and watch the view function react

```
l.append(0)
l[0] = 1
l.extend([2, 3])
```

```
{'index': 0, 'old': Undefined, 'new': 0}
{'index': 0, 'old': 0, 'new': 1}
{'index': 1, 'old': Undefined, 'new': 2}
{'index': 2, 'old': Undefined, 'new': 3}
```

PYTHON MODULE INDEX

S

INDEX

V