pysm Documentation

Release 0.3.8

Piotr Gularski

Mar 23, 2019

Contents

Github | PyPI

The State Pattern solves many problems, untangles the code and saves one's sanity. Yet.., it's a bit rigid and doesn't scale. The goal of this library is to give you a close to the State Pattern simplicity with much more flexibility. And, if needed, the full state machine functionality, including FSM, HSM, PDA and other tasty things.

Goals:

- Provide a State Pattern-like behavior with more flexibility
- Be explicit and don't add any code to objects
- Handle directly any kind of event (not only strings) - parsing strings is cool again!
- Keep it simple, even for someone who's not very familiar with the FSM terminology

CHAPTER 1

Module documentation

Python State Machine

The goal of this library is to give you a close to the State Pattern simplicity with much more flexibility. And, if needed, the full state machine functionality, including FSM, HSM, PDA and other tasty things.

Goals:

- Provide a State Pattern-like behavior with more flexibility
- Be explicit and don't add any code to objects
- Handle directly any kind of event (not only strings) - parsing strings is cool again!
- Keep it simple, even for someone who's not very familiar with the FSM terminology

---

class pysm.pysm.**AnyEvent**
Bases: object

hash(object()) doesn't work in MicroPython therefore the need for this class.

class pysm.pysm.**Event**(*name*, *input=None*, ***cargo*)
Bases: object

Triggers actions and transition in *StateMachine*.

Events are also used to control the flow of data propagated to states within the states hierarchy.

Event objects have the following attributes set after an event has been dispatched:

Attributes:

state_machine
A *StateMachine* instance that is handling the event (the one whose *pysm.pysm.StateMachine.dispatch()* method is called)

propagate
An event is propagated from a current leaf state up in the states hierarchy until it encounters a handler that can handle the event. To propagate it further, it has to be set to *True* in a handler.

Parameters

- **name** (Hashable) – Name of an event. It may be anything as long as it's hashable.
- **input** (Hashable) – Optional input. Anything hashable.
- ****cargo** – Keyword arguments for an event, used to transport data to handlers. It's added to an event as a *cargo* property of type *dict*. For *enter* and *exit* events, the original event that triggered a transition is passed in cargo as *source_event* entry.

Example Usage:

```
state_machine.dispatch(Event('start'))
state_machine.dispatch(Event('start', key='value'))
state_machine.dispatch(Event('parse', input='#', entity=my_object))
state_machine.dispatch(Event('%'))
state_machine.dispatch(Event(frozenset([1, 2])))
```

class pysm.pysm.**State**(*name*)

Bases: object

Represents a state in a state machine.

enter and *exit* handlers are called whenever a state is entered or exited respectively. These action names are reserved only for this purpose.

It is encouraged to extend this class to encapsulate a state behavior, similarly to the State Pattern.

Once it's extended, the preferred way of adding an event handlers is through the *register_handlers()* hook. Usually, there's no need to create the __init__() in a subclass.

Parameters **name** (*str*) – Human readable state name

Example Usage:

```
# Extending State to encapsulate state-related behavior. Similar to the
# State Pattern.
class Running(State):
    def on_enter(self, state, event):
        print('Running state entered')

    def on_jump(self, state, event):
        print('Jumping')

    def on_dollar(self, state, event):
        print('Dollar found!')

    def register_handlers(self):
        self.handlers = {
            'enter': self.on_enter,
            'jump': self.on_jump,
            '$': self.on_dollar
        }
```

```
# Different way of attaching handlers. A handler may be any function as
# long as it takes `state` and `event` args.
def another_handler(state, event):
    print('Another handler')

running = State('running')
```

(continues on next page)

(continued from previous page)

```
running.handlers = {
    'another_event': another_handler
}
```

is_substate(*state*)

Check whether the *state* is a substate of *self*.

Also *self* is considered a substate of *self*.

Parameters **state** (*State*) – State to verify

Returns *True* if *state* is a substate of *self*, *False* otherwise

Return type bool

register_handlers()

Hook method to register event handlers.

It is used to easily extend *State* class. The hook is called from within the base `State.__init__()`. Usually, the `__init__()` doesn't have to be created in a subclass.

Event handlers are kept in a *dict*, with events' names as keys, therefore registered events may be of any hashable type.

Handlers take two arguments:

- **state: The current state that is handling an event. The same** handler function may be attached to many states, therefore it is helpful to get the handling state's instance.
- **event: An event that triggered the handler call. If it is an** *enter* or *exit* event, then the source event (the one that triggered the transition) is passed in *event*'s cargo property as *cargo.source_event*.

Example Usage:

```
class On(State):
    def handle_my_event(self, state, event):
        print('Handling an event')

    def register_handlers(self):
        self.handlers = {
            'my_event': self.handle_my_event,
            '&': self.handle_my_event,
            frozenset([1, 2]): self.handle_my_event
        }
```

class pysm.pysm.**StateMachine**(*name*)

Bases: *pysm.pysm.State*

State machine controls actions and transitions.

To provide the State Pattern-like behavior, the formal state machine rules may be slightly broken, and instead of creating an internal transition for every action that doesn't require a state change, event handlers may be added to states. These are handled first when an event occurs. After that the actual transition is called, calling *enter*/*exit* actions and other transition actions. Nevertheless, internal transitions are also supported.

So the order of calls on an event is as follows:

1. State's event handler
2. *condition* callback
3. *before* callback

4. *exit* handlers
5. *action* callback
6. *enter* handlers
7. *after* callback

If there's no handler in states or transition for an event, it is silently ignored.

If using nested state machines, all events should be sent to the root state machine.

Attributes:

state
Current, local state (instance of *State*) in a state machine.

stack
Stack that can be used if the Pushdown Automaton (PDA) functionality is needed.

state_stack
Stack of previous local states in a state machine. With every transition, a previous state (instance of *State*) is pushed to the *state_stack*. Only `StateMachine.STACK_SIZE` (32 by default) are stored and old values are removed from the stack.

leaf_state_stack
Stack of previous leaf states in a state machine. With every transition, a previous leaf state (instance of *State*) is pushed to the *leaf_state_stack*. Only `StateMachine.STACK_SIZE` (32 by default) are stored and old values are removed from the stack.

leaf_state See the *leaf_state* property.

root_machine See the *root_machine* property.

Parameters **name** (*str*) – Human readable state machine name

---

Note: *StateMachine* extends *State* and therefore it is possible to always use a *StateMachine* instance instead of the *State*. This wouldn't be a good practice though, as the *State* class is designed to be as small as possible memory-wise and thus it's more memory efficient. It is valid to replace a *State* with a *StateMachine* later on if there's a need to extend a state with internal states.

Note: For the sake of speed thread safety isn't guaranteed.

Example Usage:

```
state_machine = StateMachine('root_machine')
state_on = State('On')
state_off = State('Off')
state_machine.add_state('Off', initial=True)
state_machine.add_state('On')
state_machine.add_transition(state_on, state_off, events=['off'])
state_machine.add_transition(state_off, state_on, events=['on'])
state_machine.initialize()
state_machine.dispatch(Event('on'))
```

add_state(*state*, *initial=False*)
Add a state to a state machine.

If states are added, one (and only one) of them has to be declared as *initial*.

Parameters

- **state** (*State*) – State to be added. It may be an another *StateMachine*
- **initial** (*bool*) – Declare a state as initial

add_states(**states*)
Add *states* to the *StateMachine*.

To set the initial state use *set_initial_state()*.

Parameters **states** (*State*) – A list of states to be added

add_transition(*from_state*, *to_state*, *events*, *input=None*, *action=None*, *condition=None*, *before=None*, *after=None*)
Add a transition to a state machine.

All callbacks take two arguments - *state* and *event*. See parameters description for details.

It is possible to create conditional if/elif/else-like logic for transitions. To do so, add many same transition rules with different condition callbacks. First met condition will trigger a transition, if no condition is met, no transition is performed.

Parameters

- **from_state** (*State*) – Source state
- **to_state** (*State*, *None*) – Target state. If *None*, then it's an internal transition
- **events** (`Iterable` of `Hashable`) – List of events that trigger the transition
- **input** (*None*, `Iterable` of `Hashable`) – List of inputs that trigger the transition. A transition event may be associated with a specific input. i.e.: An event may be `parse` and an input associated with it may be `$`. May be *None* (default), then every matched event name triggers a transition.
- **action** (`Callable`) – Action callback that is called during the transition after all states have been left but before the new one is entered.

  action callback takes two arguments:

  – state: Leaf state before transition

  – event: Event that triggered the transition
- **condition** (`Callable`) – Condition callback - if returns *True* transition may be initiated.

  condition callback takes two arguments:

  – state: Leaf state before transition

  – event: Event that triggered the transition
- **before** (`Callable`) – Action callback that is called right before the transition.

  before callback takes two arguments:

  – state: Leaf state before transition

  – event: Event that triggered the transition
- **after** (`Callable`) – Action callback that is called just after the transition

  after callback takes two arguments:

  – state: Leaf state after transition

- event: Event that triggered the transition

dispatch(*event*)
Dispatch an event to a state machine.

If using nested state machines (HSM), it has to be called on a root state machine in the hierarchy.

Parameters **event** (*Event*) – Event to be dispatched

initial_state
Get the initial state in a state machine.

Returns Initial state in a state machine

Return type *State*

initialize()
Initialize states in the state machine.

After a state machine has been created and all states are added to it, *initialize()* has to be called.

If using nested state machines (HSM), *initialize()* has to be called on a root state machine in the hierarchy.

is_substate(*state*)
Check whether the *state* is a substate of *self*.

Also *self* is considered a substate of *self*.

Parameters **state** (*State*) – State to verify

Returns *True* if *state* is a substate of *self*, *False* otherwise

Return type bool

leaf_state
Get the current leaf state.

The *state* property gives the current, local state in a state machine. The *leaf_state* goes to the bottom in a hierarchy of states. In most cases, this is the property that should be used to get the current state in a state machine, even in a flat FSM, to keep the consistency in the code and to avoid confusion.

Returns Leaf state in a hierarchical state machine

Return type *State*

register_handlers()
Hook method to register event handlers.

It is used to easily extend *State* class. The hook is called from within the base `State.__init__()`. Usually, the `__init__()` doesn't have to be created in a subclass.

Event handlers are kept in a *dict*, with events' names as keys, therefore registered events may be of any hashable type.

Handlers take two arguments:

- **state: The current state that is handling an event. The same** handler function may be attached to many states, therefore it is helpful to get the handling state's instance.
- **event: An event that triggered the handler call. If it is an** *enter* or *exit* event, then the source event (the one that triggered the transition) is passed in *event*'s cargo property as *cargo.source_event*.

Example Usage:

```
class On(State):
    def handle_my_event(self, state, event):
        print('Handling an event')

    def register_handlers(self):
        self.handlers = {
            'my_event': self.handle_my_event,
            '&': self.handle_my_event,
            frozenset([1, 2]): self.handle_my_event
        }
```

revert_to_previous_leaf_state(*event=None*)
Similar to *set_previous_leaf_state()* but the current leaf_state is not saved on the stack of states. It allows to perform transitions further in the history of states.

root_machine
Get the root state machine in a states hierarchy.

Returns Root state in the states hierarchy

Return type *StateMachine*

set_initial_state(*state*)
Set an initial state in a state machine.

Parameters **state** (*State*) – Set this state as initial in a state machine

set_previous_leaf_state(*event=None*)
Transition to a previous leaf state. This makes a dynamic transition to a historical state. The current *leaf_state* is saved on the stack of historical leaf states when calling this method.

Parameters **event** (*Event*) – (Optional) event that is passed to states involved in the transition

exception pysm.pysm.**StateMachineException**
Bases: exceptions.Exception

All *StateMachine* exceptions are of this type.

CHAPTER 2

Installation

Install pysm from PyPI:

```
pip install pysm
```

or clone the Github pysm repository:

```
git clone https://github.com/pgularski/pysm
cd pysm
python setup.py install
```

CHAPTER 3

Examples

3.1 Simple state machine

This is a simple state machine with only two states - *on* and *off*.

```
from pysm import State, StateMachine, Event

on = State('on')
off = State('off')

sm = StateMachine('sm')
sm.add_state(on, initial=True)
sm.add_state(off)

sm.add_transition(on, off, events=['off'])
sm.add_transition(off, on, events=['on'])

sm.initialize()

def test():
    assert sm.state == on
    sm.dispatch(Event('off'))
    assert sm.state == off
    sm.dispatch(Event('on'))
```

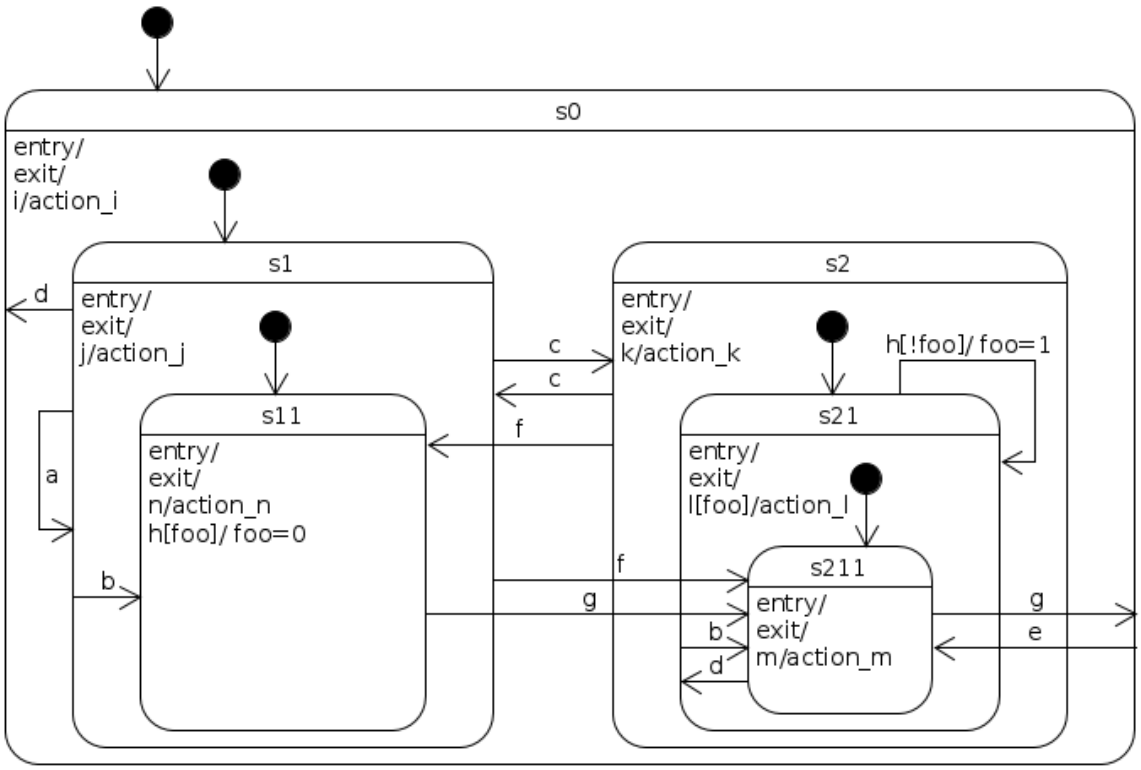
(continues on next page)

(continued from previous page)

```
    assert sm.state == on


if __name__ == '__main__':
    test()
```

3.2 Complex hierarchical state machine

A Hierarchical state machine similar to the one from Miro Samek's book[1], page 95. *It is a state machine that contains all possible state transition topologies up to four levels of state nesting*[2]




```
from pysm import State, StateMachine, Event

foo = True

def on_enter(state, event):
    print('enter state {0}'.format(state.name))

def on_exit(state, event):
    print('exit state {0}'.format(state.name))

def set_foo(state, event):
```

(continues on next page)

[1] Miro Samek, Practical Statecharts in C/C++, CMP Books 2002.
[2] http://www.embedded.com/print/4008251 (visited on 07.06.2016)

(continued from previous page)

```
    global foo
    print('set foo')
    foo = True

def unset_foo(state, event):
    global foo
    print('unset foo')
    foo = False

def action_i(state, event):
    print('action_i')

def action_j(state, event):
    print('action_j')

def action_k(state, event):
    print('action_k')

def action_l(state, event):
    print('action_l')

def action_m(state, event):
    print('action_m')

def action_n(state, event):
    print('action_n')

def is_foo(state, event):
    return foo is True

def is_not_foo(state, event):
    return foo is False


m = StateMachine('m')
s0 = StateMachine('s0')
s1 = StateMachine('s1')
s2 = StateMachine('s2')
s11 = State('s11')
s21 = StateMachine('s21')
s211 = State('s211')

m.add_state(s0, initial=True)
s0.add_state(s1, initial=True)
s0.add_state(s2)
s1.add_state(s11, initial=True)
s2.add_state(s21, initial=True)
s21.add_state(s211, initial=True)

# Internal transitions
m.add_transition(s0, None, events='i', action=action_i)
s0.add_transition(s1, None, events='j', action=action_j)
s0.add_transition(s2, None, events='k', action=action_k)
s1.add_transition(s11, None, events='h', condition=is_foo, action=unset_foo)
s1.add_transition(s11, None, events='n', action=action_n)
s21.add_transition(s211, None, events='m', action=action_m)
s2.add_transition(s21, None, events='l', condition=is_foo, action=action_l)
```

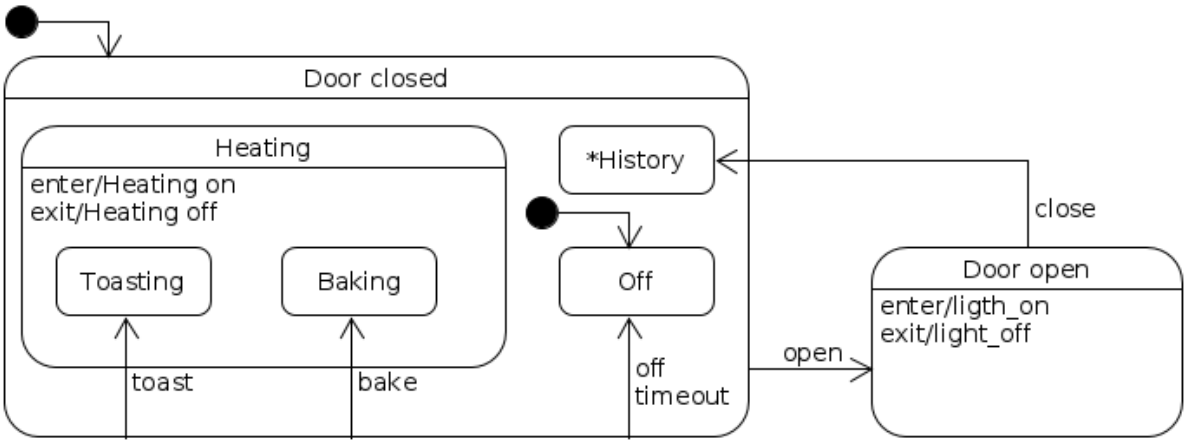
(continues on next page)

(continued from previous page)

```
# External transition
m.add_transition(s0, s211, events='e')
s0.add_transition(s1, s0, events='d')
s0.add_transition(s1, s11, events='b')
s0.add_transition(s1, s1, events='a')
s0.add_transition(s1, s211, events='f')
s0.add_transition(s1, s2, events='c')
s0.add_transition(s2, s11, events='f')
s0.add_transition(s2, s1, events='c')
s1.add_transition(s11, s211, events='g')
s21.add_transition(s211, s0, events='g')
s21.add_transition(s211, s21, events='d')
s2.add_transition(s21, s211, events='b')
s2.add_transition(s21, s21, events='h', condition=is_not_foo, action=set_foo)

# Attach enter/exit handlers
states = [m, s0, s1, s11, s2, s21, s211]
for state in states:
    state.handlers = {'enter': on_enter, 'exit': on_exit}

m.initialize()


def test():
    assert m.leaf_state == s11
    m.dispatch(Event('a'))
    assert m.leaf_state == s11
    # This transition toggles state between s11 and s211
    m.dispatch(Event('c'))
    assert m.leaf_state == s211
    m.dispatch(Event('b'))
    assert m.leaf_state == s211
    m.dispatch(Event('i'))
    assert m.leaf_state == s211
    m.dispatch(Event('c'))
    assert m.leaf_state == s11
    assert foo is True
    m.dispatch(Event('h'))
    assert foo is False
    assert m.leaf_state == s11
    # Do nothing if foo is False
    m.dispatch(Event('h'))
    assert m.leaf_state == s11
    # This transition toggles state between s11 and s211
    m.dispatch(Event('c'))
    assert m.leaf_state == s211
    assert foo is False
    m.dispatch(Event('h'))
    assert foo is True
    assert m.leaf_state == s211
    m.dispatch(Event('h'))
    assert m.leaf_state == s211


if __name__ == '__main__':
    test()
```

3.3 Different ways to attach event handlers

A state machine and states may be created in many ways. The code below mixes many styles to demonstrate it (In production code you'd rather keep your code style consistent). One way is to subclass the *State* class and attach event handlers to it. This resembles the State Pattern way of writing a state machine. But handlers may live anywhere, really, and you can attach them however you want. You're free to chose your own style of writing state machines with pysm. Also in this example a transition to a historical state is used.




```
import threading
import time
from pysm import StateMachine, State, Event


# It's possible to encapsulate all state related behaviour in a state class.
class HeatingState(StateMachine):
    def on_enter(self, state, event):
        oven = event.cargo['source_event'].cargo['oven']
        if not oven.timer.is_alive():
            oven.start_timer()
        print('Heating on')

    def on_exit(self, state, event):
        print('Heating off')

    def register_handlers(self):
        self.handlers = {
            'enter': self.on_enter,
            'exit': self.on_exit,
        }


class Oven(object):
    TIMEOUT = 0.1

    def __init__(self):
        self.sm = self._get_state_machine()
        self.timer = threading.Timer(Oven.TIMEOUT, self.on_timeout)
```

(continues on next page)

(continued from previous page)

```
    def _get_state_machine(self):
        oven = StateMachine('Oven')
        door_closed = StateMachine('Door closed')
        door_open = State('Door open')
        heating = HeatingState('Heating')
        toasting = State('Toasting')
        baking = State('Baking')
        off = State('Off')

        oven.add_state(door_closed, initial=True)
        oven.add_state(door_open)
        door_closed.add_state(off, initial=True)
        door_closed.add_state(heating)
        heating.add_state(baking, initial=True)
        heating.add_state(toasting)

        oven.add_transition(door_closed, toasting, events=['toast'])
        oven.add_transition(door_closed, baking, events=['bake'])
        oven.add_transition(door_closed, off, events=['off', 'timeout'])
        oven.add_transition(door_closed, door_open, events=['open'])

        # This time, a state behaviour is handled by Oven's methods.
        door_open.handlers = {
            'enter': self.on_open_enter,
            'exit': self.on_open_exit,
            'close': self.on_door_close
        }

        oven.initialize()
        return oven

    @property
    def state(self):
        return self.sm.leaf_state.name

    def light_on(self):
        print('Light on')

    def light_off(self):
        print('Light off')

    def start_timer(self):
        self.timer.start()

    def bake(self):
        self.sm.dispatch(Event('bake', oven=self))

    def toast(self):
        self.sm.dispatch(Event('toast', oven=self))

    def open_door(self):
        self.sm.dispatch(Event('open', oven=self))

    def close_door(self):
        self.sm.dispatch(Event('close', oven=self))

    def on_timeout(self):
```

(continues on next page)

(continued from previous page)

```
        print('Timeout...')
        self.sm.dispatch(Event('timeout', oven=self))
        self.timer = threading.Timer(Oven.TIMEOUT, self.on_timeout)

    def on_open_enter(self, state, event):
        print('Opening door')
        self.light_on()

    def on_open_exit(self, state, event):
        print('Closing door')
        self.light_off()

    def on_door_close(self, state, event):
        # Transition to a history state
        self.sm.set_previous_leaf_state(event)


def test_oven():
    oven = Oven()
    print(oven.state)
    assert oven.state == 'Off'
    oven.bake()
    print(oven.state)
    assert oven.state == 'Baking'
    oven.open_door()
    print(oven.state)
    assert oven.state == 'Door open'
    oven.close_door()
    print(oven.state)
    assert oven.state == 'Baking'
    time.sleep(0.2)
    print(oven.state)
    assert oven.state == 'Off'


if __name__ == '__main__':
    test_oven()
```

3.4 Reverse Polish notation calculator

A state machine is used in the Reverse Polish notation (RPN) calculator as a parser. A single event name (*parse*) is used along with specific *inputs* (See `pysm.pysm.StateMachine.add_transition()`).

This example also demonstrates how to use the stack of a state machine, so it behaves as a Pushdown Automaton (PDA)

```
import string as py_string
from pysm import StateMachine, Event, State


class Calculator(object):
    def __init__(self):
        self.sm = self.get_state_machine()
        self.result = None
```

(continues on next page)

(continued from previous page)

```
    def get_state_machine(self):
        sm = StateMachine('sm')
        initial = State('Initial')
        number = State('BuildingNumber')
        sm.add_state(initial, initial=True)
        sm.add_state(number)
        sm.add_transition(initial, number,
                          events=['parse'], input=py_string.digits,
                          action=self.start_building_number)
        sm.add_transition(number, None,
                          events=['parse'], input=py_string.digits,
                          action=self.build_number)
        sm.add_transition(number, initial,
                          events=['parse'], input=py_string.whitespace)
        sm.add_transition(initial, None,
                          events=['parse'], input='+-*/',
                          action=self.do_operation)
        sm.add_transition(initial, None,
                          events=['parse'], input='=',
                          action=self.do_equal)
        sm.initialize()
        return sm

    def parse(self, string):
        for char in string:
            self.sm.dispatch(Event('parse', input=char))

    def calculate(self, string):
        self.parse(string)
        return self.result

    def start_building_number(self, state, event):
        digit = event.input
        self.sm.stack.push(int(digit))
        return True

    def build_number(self, state, event):
        digit = event.input
        number = str(self.sm.stack.pop())
        number += digit
        self.sm.stack.push(int(number))
        return True

    def do_operation(self, state, event):
        operation = event.input
        y = self.sm.stack.pop()
        x = self.sm.stack.pop()
        # eval is evil
        result = eval('float({0}) {1} float({2})'.format(x, operation, y))
        self.sm.stack.push(result)
        return True

    def do_equal(self, state, event):
        operation = event.input
        number = self.sm.stack.pop()
        self.result = number
```

(continues on next page)

(continued from previous page)

```
        return True


def test_calc_callbacks():
    calc = Calculator()
    assert calc.calculate(' 167 3 2 2 * * * 1 - =') == 2003
    assert calc.calculate('    167 3 2 2 * * * 1 - 2 / =') == 1001.5
    assert calc.calculate('    3   5 6 +  * =') == 33
    assert calc.calculate('        3    4       +     =') == 7
    assert calc.calculate('2 4 / 5 6 - * =') == -0.5

if __name__ == '__main__':
    test_calc_callbacks()
```

Python Module Index

p

Index