pystate Documentation

Release latest

Aug 19, 2017

Contents

FiniteStateMachine is a class representing a finite state machine. Each state is represented by an instance of the State class. Each state also has a state handler function defined for it. The handler function is a co-routine that excepts an event and performs an action based on it.

To define the state machine:

1. define the states (e.g. STATE_A = State('STATE_A'))
2. define the state handler functions. See below for the structure of a state handling function.
3. create an instance of the state machine (e.g. fsm = Fsm())
4. **add states to the state machine, including exactly one state marked as the initial state. Each state also takes** a sequence of states that it can be transitioned from (from_states).
5. call the start function on the FSM (e.g. fsm.start())

For each event you will need to call the dispatch_event function (e.g. fsm.dispatch_event()) to route the event to the co-routine. An event can be anything you want (e.g. a tuple with event_id and arguments). The main loops generally looks like:

try:

while True: event = get_next_event() fsm.dispatch_event(event)

except ExpectedExit as e: pass

The basic structure of a state handler is:

def state_handler_<state name>(fsm): # Enter the main loop for the co-routine while True:

event = yield

if event == 'EVENT_1': # Transition to another state fsm.transition_to(STATE_X)

elif event == 'EVENT_2': # Do some processing but stay in this state print('Got EVENT_2')

elif event == 'TERMINATING_EVENT': raise FsmExit

else: print('Unrecognized event (%s)' % event)

A simple example of this is shown in the turnstile_test.py test case.

For convenience this can be wrapped with a @state_handler decorator. The decorator takes care of the co-routine boiler plate and hands the handler function an fsm and event. This would look like:

```
@pystate.state_handler
```

def state_locked_handler(event, fsm):

if event == 'EVENT_1': # Transition to another state fsm.transition_to(STATE_X)

elif event == 'EVENT_2': # Do some processing but stay in this state print('Got EVENT_2')

elif event == 'TERMINATING_EVENT': raise FsmExit

else: print('Unrecognized event (%s)' % event)

There are two ways to handle a state that needs to keep persistant data. You can create a callable clas (i.e. define the __call__ dunder method to call as the state handler.) This allows you to use the state_handler decorator around the __call__ method. Alternatively, you can set the state data above the while loop if you define the co-routine by hand, however, this precludes using the decorator. See the callable_test.py test case for an example.