```
import time
import random
from collections import OrderedDict

from simulator import Simulator

class TrafficLight(object):
    """A traffic light that switches periodically."""

    valid_states = [True, False]  # True = NS open, False = EW open

    def __init__(self, state=None, period=None):
        self.state = state if state is not None else random.choice(self.valid_states)
        self.period = period if period is not None else random.choice([3, 4, 5])
        self.last_updated = 0

    def reset(self):
        self.last_updated = 0

    def update(self, t):
        if t - self.last_updated >= self.period:
            self.state = not self.state  # assuming state is boolean
            self.last_updated = t


class Environment(object):
    """Environment within which all agents operate."""

    valid_actions = [None, 'forward', 'left', 'right']
    valid_inputs = {'light': TrafficLight.valid_states, 'oncoming': valid_actions, 'left':
valid_actions, 'right': valid_actions}
    valid_headings = [(1, 0), (0, -1), (-1, 0), (0, 1)]  # ENWS
    hard_time_limit = -100  # even if enforce_deadline is False, end trial when deadline
reaches this value (to avoid deadlocks)

    def __init__(self, num_dummies=3):
        self.num_dummies = num_dummies  # no. of dummy agents

        # Initialize simulation variables
        self.done = False
        self.t = 0
        self.agent_states = OrderedDict()
        self.status_text = ""

        # Road network
        self.grid_size = (8, 6)  # (cols, rows)
        self.bounds = (1, 1, self.grid_size[0], self.grid_size[1])
```

```
        self.block_size = 100
        self.intersections = OrderedDict()
        self.roads = []
        for x in xrange(self.bounds[0], self.bounds[2] + 1):
            for y in xrange(self.bounds[1], self.bounds[3] + 1):
                self.intersections[(x, y)] = TrafficLight()  # a traffic light at each intersection

        for a in self.intersections:
            for b in self.intersections:
                if a == b:
                    continue
                if (abs(a[0] - b[0]) + abs(a[1] - b[1])) == 1:  # L1 distance = 1
                    self.roads.append((a, b))

        # Dummy agents
        for i in xrange(self.num_dummies):
            self.create_agent(DummyAgent)

        # Primary agent and associated parameters
        self.primary_agent = None  # to be set explicitly
        self.enforce_deadline = False

    def create_agent(self, agent_class, *args, **kwargs):
        agent = agent_class(self, *args, **kwargs)
        self.agent_states[agent] = {'location': random.choice(self.intersections.keys()),
'heading': (0, 1)}
        return agent

    def set_primary_agent(self, agent, enforce_deadline=False):
        self.primary_agent = agent
        self.enforce_deadline = enforce_deadline

    def reset(self):
        self.done = False
        self.t = 0

        # Reset traffic lights
        for traffic_light in self.intersections.itervalues():
            traffic_light.reset()

        # Pick a start and a destination
        start = random.choice(self.intersections.keys())
        destination = random.choice(self.intersections.keys())

        # Ensure starting location and destination are not too close
        while self.compute_dist(start, destination) < 4:
            start = random.choice(self.intersections.keys())
```

```
            destination = random.choice(self.intersections.keys())

        start_heading = random.choice(self.valid_headings)
        deadline = self.compute_dist(start, destination) * 5
        print "Environment.reset(): Trial set up with start = {}, destination = {}, deadline =
{}".format(start, destination, deadline)

        # Initialize agent(s)
        for agent in self.agent_states.iterkeys():
            self.agent_states[agent] = {
                'location': start if agent is self.primary_agent else
random.choice(self.intersections.keys()),
                'heading': start_heading if agent is self.primary_agent else
random.choice(self.valid_headings),
                'destination': destination if agent is self.primary_agent else None,
                'deadline': deadline if agent is self.primary_agent else None}
            agent.reset(destination=(destination if agent is self.primary_agent else None))

    def step(self):
        #print "Environment.step(): t = {}".format(self.t)  # [debug]

        # Update traffic lights
        for intersection, traffic_light in self.intersections.iteritems():
            traffic_light.update(self.t)

        # Update agents
        for agent in self.agent_states.iterkeys():
            agent.update(self.t)

        if self.done:
            return  # primary agent might have reached destination

        if self.primary_agent is not None:
            agent_deadline = self.agent_states[self.primary_agent]['deadline']
            if agent_deadline <= self.hard_time_limit:
                self.done = True
                print "Environment.step(): Primary agent hit hard time limit ({})! Trial
aborted.".format(self.hard_time_limit)
            elif self.enforce_deadline and agent_deadline <= 0:
                self.done = True
                print "Environment.step(): Primary agent ran out of time! Trial aborted."
            self.agent_states[self.primary_agent]['deadline'] = agent_deadline - 1

        self.t += 1

    def sense(self, agent):
        assert agent in self.agent_states, "Unknown agent!"
```

```
        state = self.agent_states[agent]
        location = state['location']
        heading = state['heading']
        light = 'green' if (self.intersections[location].state and heading[1] != 0) or ((not
self.intersections[location].state) and heading[0] != 0) else 'red'

        # Populate oncoming, left, right
        oncoming = None
        left = None
        right = None
        for other_agent, other_state in self.agent_states.iteritems():
            if agent == other_agent or location != other_state['location'] or (heading[0] ==
other_state['heading'][0] and heading[1] == other_state['heading'][1]):
                continue
            other_heading = other_agent.get_next_waypoint()
            if (heading[0] * other_state['heading'][0] + heading[1] * other_state['heading'][1])
== -1:
                if oncoming != 'left':  # we don't want to override oncoming == 'left'
                    oncoming = other_heading
            elif (heading[1] == other_state['heading'][0] and -heading[0] ==
other_state['heading'][1]):
                if right != 'forward' and right != 'left':  # we don't want to override right ==
'forward or 'left'
                    right = other_heading
            else:
                if left != 'forward':  # we don't want to override left == 'forward'
                    left = other_heading

        return {'light': light, 'oncoming': oncoming, 'left': left, 'right': right}

    def get_deadline(self, agent):
        return self.agent_states[agent]['deadline'] if agent is self.primary_agent else None

    def act(self, agent, action):
        assert agent in self.agent_states, "Unknown agent!"
        assert action in self.valid_actions, "Invalid action!"

        state = self.agent_states[agent]
        location = state['location']
        heading = state['heading']
        light = 'green' if (self.intersections[location].state and heading[1] != 0) or ((not
self.intersections[location].state) and heading[0] != 0) else 'red'
        inputs = self.sense(agent)

        # Move agent if within bounds and obeys traffic rules
        reward = 0  # reward/penalty
```

```
        move_okay = True
        if action == 'forward':
            if light != 'green':
                move_okay = False
        elif action == 'left':
            if light == 'green' and (inputs['oncoming'] == None or inputs['oncoming'] == 'left'):
                heading = (heading[1], -heading[0])
            else:
                move_okay = False
        elif action == 'right':
            if light == 'green' or inputs['left'] != 'forward':
                heading = (-heading[1], heading[0])
            else:
                move_okay = False

        if move_okay:
            # Valid move (could be null)
            if action is not None:
                # Valid non-null move
                location = ((location[0] + heading[0] - self.bounds[0]) % (self.bounds[2] -
self.bounds[0] + 1) + self.bounds[0],
                            (location[1] + heading[1] - self.bounds[1]) % (self.bounds[3] -
self.bounds[1] + 1) + self.bounds[1])  # wrap-around
                #if self.bounds[0] <= location[0] <= self.bounds[2] and self.bounds[1] <=
location[1] <= self.bounds[3]:  # bounded
                state['location'] = location
                state['heading'] = heading
                reward = 2.0 if action == agent.get_next_waypoint() else -0.5  # valid, but is it
correct? (as per waypoint)
            else:
                # Valid null move
                reward = 0.0
        else:
            # Invalid move
            reward = -1.0

        if agent is self.primary_agent:
            if state['location'] == state['destination']:
                if state['deadline'] >= 0:
                    reward += 10  # bonus
                self.done = True
                print "Environment.act(): Primary agent has reached destination!"  # [debug]
            self.status_text = "state: {}\naction: {}\nreward: {}".format(agent.get_state(),
action, reward)
            #print "Environment.act() [POST]: location: {}, heading: {}, action: {}, reward:
{}".format(location, heading, action, reward)  # [debug]
```

```
        return reward

    def compute_dist(self, a, b):
        """L1 distance between two points."""
        return abs(b[0] - a[0]) + abs(b[1] - a[1])


class Agent(object):
    """Base class for all agents."""

    def __init__(self, env):
        self.env = env
        self.state = None
        self.next_waypoint = None
        self.color = 'cyan'

    def reset(self, destination=None):
        pass

    def update(self, t):
        pass

    def get_state(self):
        return self.state

    def get_next_waypoint(self):
        return self.next_waypoint


class DummyAgent(Agent):
    color_choices = ['blue', 'cyan', 'magenta', 'orange']

    def __init__(self, env):
        super(DummyAgent, self).__init__(env)  # sets self.env = env, state = None,
next_waypoint = None, and a default color
        self.next_waypoint = random.choice(Environment.valid_actions[1:])
        self.color = random.choice(self.color_choices)

    def update(self, t):
        inputs = self.env.sense(self)

        action_okay = True
        if self.next_waypoint == 'right':
            if inputs['light'] == 'red' and inputs['left'] == 'forward':
                action_okay = False
        elif self.next_waypoint == 'forward':
            if inputs['light'] == 'red':
```

```
                action_okay = False
        elif self.next_waypoint == 'left':
            if inputs['light'] == 'red' or (inputs['oncoming'] == 'forward' or inputs['oncoming']
== 'right'):
                action_okay = False

        action = None
        if action_okay:
            action = self.next_waypoint
            self.next_waypoint = random.choice(Environment.valid_actions[1:])
        reward = self.env.act(self, action)
        #print "DummyAgent.update(): t = {}, inputs = {}, action = {}, reward = {}".format(t,
inputs, action, reward)  # [debug]
        #print "DummyAgent.update(): next_waypoint = {}".format(self.next_waypoint)  #
[debug]
```