```python
# import sys
# import os
# sys.path.insert(1, os.path.join(sys.path[0], '..'))

from .environment_base import SimpleTrader
import numpy as np

# Julian's implementation of the environment
class Trader(SimpleTrader):

    def __init__(self, ticker_list, observation_metrics, initial_funds=2000, starting_date="2023-04-05", ending_date="2023-10-05"):
        super().__init__(ticker_list, initial_funds=initial_funds, starting_date=starting_date, ending_date=ending_date, observation_metrics=observation_metrics)

    def reset(self, render=False, seed=None):

        if seed != None:
            np.random.seed(seed)

        if render == True:
            self._render_on_completion()

        if not hasattr(self, "funds_history"):
            self.funds_history, self.portfolio_history = [], []
            self.episode_funds, self.episode_portfolio = [], []
            self.render_episodes = False

        self.curr_step = 0
        self.curr_funds = self.initial_funds
        self.portfolio_value = self.initial_funds

        self.num_buys, self.num_sells = 0, 0
        self.buy_percents, self.sell_percents = 0.0, 0.0
        self.owned_shares = np.zeros(self.num_stocks)

        observation = self._get_observation_julian_0()
        #observation = self._get_observation_julian_1()

        return observation, {}

    def step(self, action_list):

        self.curr_step += 1

        done = self.curr_step >= self.num_trading_days

        reward = self._perform_action_julian_0(action_list)

        if self.render_episodes == True:
            self.episode_funds.append(self.curr_funds - self.initial_funds)
            self.episode_portfolio.append(self.portfolio_value)

            if done:
                self.funds_history.append(self.episode_funds)
                self.portfolio_history.append(
                    self.episode_portfolio)
                self.episode_funds, self.episode_portfolio = [], []
                self._get_total_action_count()
                self._get_total_buy_sell_percents()

        if not done:
            observation = self._get_observation_julian_0()
            #observation = self._get_observation_julian_1()
        else:
            observation = None

        return observation, reward, done, False, {}

    def _perform_action_julian_0(self, action_list):

        curr_date = self.trading_days[self.curr_step - 1]

        opening_price = [self.stock_data.loc[(ticker, curr_date), "Open"] for ticker in self.ticker_list]

        stockVal = 0

        for ii in range(len(self.owned_shares)):
            stockVal += (self.owned_shares[ii] * opening_price[ii])

        startPortfolio = self.curr_funds + stockVal

        #print(f"action: {action_list}")

        buy_reward = 0
        sell_reward = 0

        money_change = []

        #Check all actions in list for sell actions
        for ii, action in enumerate(action_list):

            if action < 0:  # sell signal
                max_shares = self.owned_shares[ii]
                num_shares = int(abs(action) * max_shares)

                if self.owned_shares[ii] >= num_shares:
                    self.owned_shares[ii] -= num_shares
                    money_change.append(num_shares * opening_price[ii])
                    self.curr_funds += num_shares * opening_price[ii]

                    self.num_sells += 1
                    self.sell_percents += abs(action)

                if num_shares == 0:
                    # Reduce reward if trying to sell stock that don't exist
                    sell_reward -= abs(action * 10) ** 2
        #Check all actions in list for buy actions
        for ii, action in enumerate(action_list):
            if action > 0:  # buy signal
                max_investment = self.curr_funds
                investment = action * max_investment
                num_shares = int(investment / opening_price[ii])
                investment = num_shares * opening_price[ii]

                if self.curr_funds >= investment:
                    self.owned_shares[ii] += num_shares
                    self.curr_funds -= investment
                    money_change.append(investment)

                    self.num_buys += 1
                    self.buy_percents += action

                if investment == 0:
                    # Reduce reward if trying to buy stock without money
                    buy_reward -= abs(action * 10) ** 2

        # Prevents crashes in the rare event of an action for a stock being equal to 0
        for ii, action in enumerate(action_list):
            if action == 0.0:  # Hold stock
                money_change.append(0.0)

        closing_price = np.array([self.stock_data.loc[(ticker, curr_date), "Adj Close"] for ticker in self.ticker_list])

        self.portfolio_value = self.curr_funds + \
                               sum(self.owned_shares * closing_price)

        #reward = self._get_reward_julian_0(action_list, buy_reward, sell_reward, money_change, startPortfolio)
        #reward = self._get_reward_julian_1(action_list, closing_price, opening_price, money_change)
        #reward = self._get_reward_julian_2(action_list, closing_price, opening_price, buy_reward, sell_reward, money_change)
        reward = self._get_reward_julian_3(action_list, opening_price, buy_reward, sell_reward, money_change)

        # print(f"Reward: {reward}\n")

        self.previous_portfolio = self.portfolio_value

        return reward

    '''Adds reward for increasing value of portfolio into next state
       Penalty for buying without enough money and selling without enough stock'''
    """def _get_reward_julian_0(self, action_list, buy_reward, sell_reward, money_change, startPortfolio):
        # Reward based on increasing portfolio value from initial value
        if self.curr_step != len(self.trading_days):
            next_date = self.trading_days[self.curr_step]
        else:
            next_date = self.trading_days[self.curr_step - 1]

        next_opening_price = [self.stock_data.loc[(
            ticker, next_date), "Open"] for ticker in self.ticker_list]

        next_closing_price = [self.stock_data.loc[(
            ticker, next_date), "Adj Close"] for ticker in self.ticker_list]

        stockVal = 0
        for ii in range(len(self.owned_shares)):
            stockVal += (self.owned_shares[ii] * next_closing_price[ii])
        next_portfolio = self.curr_funds + stockVal

        #Reward based on change in portfolio value into the next state
        reward = (100 * (next_portfolio - self.portfolio_value) / next_portfolio)

        curr_date = self.trading_days[self.curr_step - 1]
        # prev_date = self.trading_days[self.curr_step - 2]

        opening_price = [self.stock_data.loc[
            (ticker, curr_date), "Open"] for ticker in self.ticker_list]

        # Adds reward for buying when a price is going up and selling when a price is going down
        for ii, action in enumerate(action_list):
            reward += (money_change[ii] * action * (next_opening_price[ii] - opening_price[ii])/next_opening_price[ii])/10

        # Penalty for buying or selling when unable to buy or sell
        reward += (buy_reward + sell_reward)

        #print(f"Portfolios: {self.portfolio_value} : {next_portfolio}")

        #print(f"reward :{reward}\n")

        return reward"""

    '''Adds reward for increasing reward from previous portfolio (outdated)
       Adds reward when buying when closing price is higher than opening price and selling when closing price is lower,
       Subtracts reward when buying when closing price is lower and selling when closing price is higher'''
    def _get_reward_julian_1(self, action_list, closing_price, opening_price, money_change):
        reward = (self.portfolio_value - self.previous_portfolio)/self.previous_portfolio

        #Adds reward for buying before a price goes up and selling before a price goes down
        for ii, action in enumerate(action_list):
            reward += (money_change[ii] * action * (closing_price[ii] - opening_price[ii])/closing_price[ii])

            #Penalise if selling when the closing price is higher than the opening price
            if closing_price[ii] > opening_price[ii] and action < 0:
                reward -= abs(action) * (closing_price[ii] - opening_price[ii])/closing_price[ii]

        return reward

    '''Adds reward for increasing reward from previous portfolio (outdated)
       Adds reward when buying when closing price is higher than opening price and selling when closing price is lower,
       Subtracts reward when buying when closing price is lower and selling when closing price is higher
       Penalty for buying without enough money and selling without enough stock'''
    def _get_reward_julian_2(self, action_list, closing_price, opening_price, buy_reward, sell_reward, money_change):
        reward = (self.portfolio_value - self.previous_portfolio)/self.previous_portfolio

        #Adds reward for buying before a price goes up and selling before a price goes down
        for ii, action in enumerate(action_list):
            reward += (money_change[ii] * action * (closing_price[ii] - opening_price[ii])/closing_price[ii])

            #Penalise if selling when the closing price is higher than the opening price
            if closing_price[ii] > opening_price[ii] and action < 0:
                reward -= abs(action) * (closing_price[ii] - opening_price[ii])/closing_price[ii]

        #Penalty for selling or buying stock when action is not available
        reward += (buy_reward + sell_reward)

        return reward

    '''Adds reward for increasing value of portfolio into next state
       Adds reward when buying when closing price is higher than opening price and selling when closing price is lower,
       Subtracts reward when buying when closing price is lower and selling when closing price is higher
       Penalty for buying without enough money and selling without enough stock'''
    def _get_reward_julian_3(self, action_list, opening_price, buy_reward, sell_reward, money_change):
        if self.curr_step != len(self.trading_days):
            next_date = self.trading_days[self.curr_step]
        else:
            next_date = self.trading_days[self.curr_step - 1]

        next_opening_price = [self.stock_data.loc[(
            ticker, next_date), "Open"] for ticker in self.ticker_list]

        next_closing_price = [self.stock_data.loc[(
            ticker, next_date), "Adj Close"] for ticker in self.ticker_list]

        stockVal = 0
        for ii in range(len(self.owned_shares)):
            stockVal += (self.owned_shares[ii] * next_closing_price[ii])
        next_portfolio = self.curr_funds + stockVal

        #Reward based on change in portfolio value into the next state
        reward = (100 * (next_portfolio - self.portfolio_value) / next_portfolio)

        #Adds reward for buying before a price goes up and selling before a price goes down
        for ii, action in enumerate(action_list):
            reward += (money_change[ii] * action * (next_opening_price[ii] - opening_price[ii])/next_opening_price[ii])

            #Penalise if selling when the next opening price is higher than the opening price
            if next_opening_price[ii] > opening_price[ii] and action < 0:
                reward -= abs(action) * (next_opening_price[ii] - opening_price[ii])/next_opening_price[ii]

        #Penalty for selling or buying stock when action is not available
        reward += (buy_reward + sell_reward)

        return reward

    #Does not include volume
    def _get_observation_julian_0(self):

        curr_date = self.trading_days[self.curr_step - 1]
        if self.curr_step != 1:
            prev_date = self.trading_days[self.curr_step - 2]
        else:
            prev_date = self.trading_days[self.curr_step - 1]

        opening_price = np.array([self.stock_data.loc[(
                                                          ticker, curr_date), "Open"] for ticker in self.ticker_list])
        opening_price_prev = np.array([self.stock_data.loc[(
                                                               ticker, prev_date), "Open"] for ticker in
                                       self.ticker_list])

        volume = np.array([self.stock_data.loc[(ticker, curr_date), "Volume"] for ticker in self.ticker_list])

        observation = [round(self.curr_funds, 3)]
        for ii in range(self.num_stocks):
            observation.append(int(self.owned_shares[ii]))
            observation.append(round(opening_price[ii], 3))
            observation.append(round(opening_price_prev[ii], 3))

        #print(f"observation: {observation}")

        return observation

    #Includes Volume
    def _get_observation_julian_1(self):

        curr_date = self.trading_days[self.curr_step - 1]
        if self.curr_step != 1:
            prev_date = self.trading_days[self.curr_step - 2]
        else:
            prev_date = self.trading_days[self.curr_step - 1]

        opening_price = np.array([self.stock_data.loc[(
                                                          ticker, curr_date), "Open"] for ticker in self.ticker_list])
        opening_price_prev = np.array([self.stock_data.loc[(
                                                               ticker, prev_date), "Open"] for ticker in
                                       self.ticker_list])

        volume = np.array([self.stock_data.loc[(ticker, curr_date), "Volume"] for ticker in self.ticker_list])

        observation = [round(self.curr_funds, 3)]
        for ii in range(self.num_stocks):
            observation.append(int(self.owned_shares[ii]))
            observation.append(round(opening_price[ii], 3))
            observation.append(round(opening_price_prev[ii], 3))
            observation.append(round(volume[ii], 3))

        #print(f"observation: {observation}")
        return observation
```