```python
from .environment_base import SimpleTrader
import numpy as np
import random
import yfinance as yf

# Ryan's implementation of the environment


class Trader(SimpleTrader):

    def __init__(self, ticker_list, observation_metrics=3, initial_funds=2000, starting_date="2023-04-05", ending_date="2023-10-05"):
        super().__init__(ticker_list, initial_funds=initial_funds, observation_metrics=observation_metrics,
                         starting_date=starting_date, ending_date=ending_date)

        self.epsilon = 1
        self.vix = yf.download(
            "^VIX", start=self.starting_date, end=self.ending_date)
        self.max_portfolio = self.initial_funds

    def reset(self, render=False, seed=None):
        """
        Resets the reinforcement learning environment

        Parameters:
            render (boolean): Whether to display the episodic performance of the
        agent
            seed (int): Seed for predictable behaviour (NOT USED)

        Returns:
            observation (float): The observations for the initial trading day
            info (dict): A dictionary containing additional information about
            the environment (NOT USED)
        """

        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)

        self.returns = []

        self.epsilon = 1
        self.max_portfolio = self.initial_funds

        observation = self._get_observation_ryan_0()
        info = {}

        return observation, info

    def step(self, action_list):
        """
        Driving logic for updating the reinformcement learning environment on
        each trading day

        Parameters:
            action_list (list): A list of actions for each stock in the portfolio

        Returns:
            observation (float): The observations for the current trading day
            reward (float): The reward signal for a given series of actions
            done (boolean): A flag to check whether the current episode has
            terminated
            terminated (boolean): A flag to check whether early stopping of the
            episode has occurred (NOT USED)
            info (dict): A dictionary containing additional information about
            the environment (NOT USED)
        """

        self.curr_step += 1

        done = self.curr_step >= self.num_trading_days

        reward = self._perform_action_ryan_0(action_list)
```

```python
        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_ryan_0()
        else:
            observation = None

        terminated = False
        info = {}

        return observation, reward, done, terminated, info

    def _perform_action_ryan_0(self, action_list):
        """
        Base action function:
            - Retrieves the opening price for a given trading day
            - Performs the actual buy / sell actions from a given action list
            - Updates the current funds, shares held and new portfolio valuation
            - Calculates and returns reward

        Parameters:
            action_list (list): A list of actions for each stock in the portfolio

        Returns:
            reward (float): The reward signal for a given series of actions
        """

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

        stockVal = 0

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

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

        buy_reward = 0
        sell_reward = 0

        money_change = []

        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 stocks that don't exist
                    sell_reward -= abs(action * 10) ** 2

        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:
```

```python
                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_ryan_7(
            action_list)

        self.previous_portfolio = self.portfolio_value

        return reward

    def _perform_action_ryan_1(self, action_list):
        """
        Second version of action function:
            - Retrieves the opening price for a given trading day
            - Performs the actual buy / sell actions from a given action list
            - Updates the current funds, shares held and new portfolio valuation
            - Calculates and returns reward
            - Includes epsilon greedy exploration (if below threshold, random actions are taken)

        Parameters:
            action_list (list): A list of actions for each stock in the portfolio

        Returns:
            reward (float): The reward signal for a given series of actions
        """

        epsilon = 0.1

        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 = []

        x = random.uniform(0, 1)

        if x < epsilon:
            action_list = np.random.uniform(-1, 1, size=(len(action_list),))
            print(f"Random action: {action_list}")

        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 stocks that don't exist
                    sell_reward -= abs(action * 10) ** 2

        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_ryan_0(
        action_list, buy_reward, sell_reward, money_change, startPortfolio)

    self.previous_portfolio = self.portfolio_value

    return reward

def _perform_action_ryan_2(self, action_list):
    """
    Third version of action function:
        - Retrieves the opening price for a given trading day
        - Performs the actual buy / sell actions from a given action list
        - Updates the current funds, shares held and new portfolio valuation
        - Calculates and returns reward
        - Includes decaying epsilon greedy exploration to be passed to reward function

    Parameters:
        action_list (list): A list of actions for each stock in the portfolio

    Returns:
        reward (float): The reward signal for a given series of actions
    """

    self.epsilon *= 0.5

    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 = []

    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 stocks that don't exist
                sell_reward -= abs(action * 10) ** 2

    for ii, action in enumerate(action_list):
```

```python
            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_ryan_4(
            action_list, buy_reward, sell_reward, money_change, startPortfolio)

        self.previous_portfolio = self.portfolio_value

        return reward

    def _get_reward_ryan_0(self, action_list, buy_reward, sell_reward, money_change):
        """
        Reward function for stock trading environment:
            - Increase portfolio value from initial value

        Parameters:
            action_list (list): A list of actions for each stock in the
            portfolio
            buy_reward (float): Reward for buying stocks
            sell_reward (float): Reward for selling stocks
            money_change (list): A list of the amount of money gained / lost

        Returns:
            reward (float): The reward signal for a given series of actions
        """

        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]

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

        reward = (100 * (next_portfolio - self.portfolio_value) / next_portfolio)

        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]

        for ii, action in enumerate(action_list):
            reward += (money_change[ii] * action * (next_opening_price[ii] -
                       opening_price[ii])/next_opening_price[ii])/10

        reward += (buy_reward + sell_reward)

        return reward

    def _get_reward_ryan_1(self, action_list, buy_reward, sell_reward, money_change):
        """
        Reward function for stock trading environment:
            - Increase portfolio value from initial value
            - Decaying Epsilon greedy exploration (gradually shrinking reward
            component to encourage a learned strategy)

        Parameters:
```

```
            action_list (list): A list of actions for each stock in the
            portfolio
            buy_reward (float): Reward for buying stocks
            sell_reward (float): Reward for selling stocks
            money_change (list): A list of the amount of money gained / lost

        Returns:
            reward (float): The reward signal for a given series of actions
        """

        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]

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

        reward = (100 * (next_portfolio - self.portfolio_value) / next_portfolio)

        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]

        for ii, action in enumerate(action_list):
            reward += (money_change[ii] * action * (next_opening_price[ii] -
                       opening_price[ii])/next_opening_price[ii])/10

        reward += (buy_reward + sell_reward)

        reward += (self.epsilon * 10)

        return reward

    def _get_reward_ryan_2(self):
        """
        Reward function for stock trading environment:
            - Calculate daily return (ratio of portfolio value to previous portfolio
            value)
            - Calculate short-term Sharpe ratio over 20 day window

        Parameters:

        Returns:
            reward (float): The reward signal for a given series of actions
        """

        SHARPE_WINDOW = 20

        daily_return = (self.portfolio_value -
                        self.previous_portfolio) / self.previous_portfolio

        self.returns.append(daily_return)

        if len(self.returns) < SHARPE_WINDOW:
            reward = 0
        else:
            ret = np.array(self.returns[-SHARPE_WINDOW:])
            sharpe_ratio = np.sqrt(SHARPE_WINDOW) * np.mean(ret) / np.std(ret)

            reward = sharpe_ratio

        return reward

    def _get_reward_ryan_3(self):
        """
        Reward function for stock trading environment:
            - Calculate daily return (ratio of portfolio value to previous portfolio
            value)
            - Calculate short-term Sharpe ratio over 20 day window
            - Dynamic volatility weighting based on VIX index

        Parameters:

        Returns:
            reward (float): The reward signal for a given series of actions
        """

        SHARPE_WINDOW = 20

        daily_return = (self.portfolio_value -
```

```python
                    self.previous_portfolio) / self.previous_portfolio

        self.returns.append(daily_return)

        reward = 0

        if len(self.returns) >= SHARPE_WINDOW:
            ret = np.array(self.returns[-SHARPE_WINDOW:])
            sharpe_ratio = np.sqrt(SHARPE_WINDOW) * np.mean(ret) / np.std(ret)

            reward += sharpe_ratio

        if len(self.returns) > 0:
            curr_volatility = np.std(
                np.array(self.returns[-min(SHARPE_WINDOW, len(self.returns)):]))

            curr_vix = self._get_vix()
            volatility_weight = curr_vix * 0.01

            volatility_penalty = curr_volatility * volatility_weight

            reward -= volatility_penalty

        return reward

    def _get_reward_ryan_4(self):
        """
        Reward function for stock trading environment:
            - Calculate daily return (ratio of portfolio value to previous portfolio
            value)
            - Calculate short-term Sharpe ratio over 20 day window
            - Dynamic volatility weighting based on VIX index
            - Dynamically adjusted risk aversion (drawdown penalty)

        Parameters:

        Returns:
            reward (float): The reward signal for a given series of actions
        """

        SHARPE_WINDOW = 20

        daily_return = (self.portfolio_value -
                        self.previous_portfolio) / self.previous_portfolio

        self.returns.append(daily_return)
        self.max_portfolio = max(self.max_portfolio, self.portfolio_value)

        reward = 0

        curr_vix = self._get_vix()

        drawdown_threshold = curr_vix * 0.005

        drawdown = (self.max_portfolio - self.portfolio_value) / \
            self.max_portfolio

        if drawdown > drawdown_threshold:
            reward -= drawdown

        if len(self.returns) >= SHARPE_WINDOW:
            ret = np.array(self.returns[-SHARPE_WINDOW:])
            sharpe_ratio = np.sqrt(SHARPE_WINDOW) * np.mean(ret) / np.std(ret)

            reward += sharpe_ratio

        if len(self.returns) > 0:
            curr_volatility = np.std(
                np.array(self.returns[-min(SHARPE_WINDOW, len(self.returns)):]))

            volatility_weight = curr_vix * 0.01

            volatility_penalty = curr_volatility * volatility_weight

            reward -= volatility_penalty

        return reward

    def _get_reward_ryan_5(self, action_list):
        """
        Reward function for stock trading environment:
            - Calculate daily return (ratio of portfolio value to previous portfolio
            value)
            - Calculate short-term Sharpe ratio over 20 day window
            - Dynamic volatility weighting based on VIX index
            - Dynamically adjusted risk aversion (drawdown penalty)
            - Trend following based on adjustable weighting
```

```
    Parameters:
        action_list (list): A list of actions for each stock in the portfolio

    Returns:
        reward (float): The reward signal for a given series of actions
    """

    SHARPE_WINDOW = 20
    TREND_REWARD = 0.1

    daily_return = (self.portfolio_value -
                    self.previous_portfolio) / self.previous_portfolio

    self.returns.append(daily_return)
    self.max_portfolio = max(self.max_portfolio, self.portfolio_value)

    reward = 0

    curr_vix = self._get_vix()

    drawdown_threshold = curr_vix * 0.005

    drawdown = (self.max_portfolio - self.portfolio_value) / \
        self.max_portfolio

    if drawdown > drawdown_threshold:
        reward -= drawdown

    if len(self.returns) >= SHARPE_WINDOW:
        ret = np.array(self.returns[-SHARPE_WINDOW:])
        sharpe_ratio = np.sqrt(SHARPE_WINDOW) * np.mean(ret) / np.std(ret)

        reward += sharpe_ratio

    if len(self.returns) > 0:
        curr_volatility = np.std(
            np.array(self.returns[-min(SHARPE_WINDOW, len(self.returns)):]))

        volatility_weight = curr_vix * 0.01

        volatility_penalty = curr_volatility * volatility_weight

        reward -= volatility_penalty

    if self.curr_step > 0:
        curr_date = self.trading_days[self.curr_step - 1]

        for ii, action in enumerate(action_list):

            if self.curr_step >= SHARPE_WINDOW:
                prev_date = self.trading_days[self.curr_step - SHARPE_WINDOW]
                stock_price = self.stock_data.xs(
                    self.ticker_list[ii], level="Ticker", axis=0).loc[prev_date:curr_date, "Adj Close"].values

                if len(stock_price) > 0:
                    ema = np.mean(stock_price)

                    curr_price = self.stock_data.loc[(
                        self.ticker_list[ii], curr_date), "Adj Close"]

                    if (action > 0 and curr_price > ema) or (action < 0 and curr_price < ema):
                        reward += TREND_REWARD * abs(action)

    return

def _get_reward_ryan_6(self, action_list):
    """
    Reward function for stock trading environment:
        - Calculate daily return (ratio of portfolio value to previous portfolio
        value)
        - Calculate short-term Sharpe ratio over 20 day window
        - Dynamic volatility weighting based on VIX index
        - Dynamically adjusted risk aversion (drawdown penalty)
        - Dynamic trend following with adjustable base weight

    Parameters:
        action_list (list): A list of actions for each stock in the portfolio

    Returns:
        reward (float): The reward signal for a given series of actions
    """

    SHARPE_WINDOW = 20
    BASE_TREND_REWARD = 0.1
    TREND_WINDOW = SHARPE_WINDOW
```

```
        daily_return = (self.portfolio_value -
                        self.previous_portfolio) / self.previous_portfolio

        self.returns.append(daily_return)
        self.max_portfolio = max(self.max_portfolio, self.portfolio_value)

        reward = 0

        curr_vix = self._get_vix()

        drawdown_threshold = curr_vix * 0.005

        drawdown = (self.max_portfolio - self.portfolio_value) / \
            self.max_portfolio

        if drawdown > drawdown_threshold:
            reward -= drawdown

        if len(self.returns) >= SHARPE_WINDOW:
            ret = np.array(self.returns[-SHARPE_WINDOW:])
            sharpe_ratio = np.sqrt(SHARPE_WINDOW) * np.mean(ret) / np.std(ret)

            reward += sharpe_ratio

        if len(self.returns) > 0:
            curr_volatility = np.std(
                np.array(self.returns[-min(SHARPE_WINDOW, len(self.returns)):]))

            volatility_weight = curr_vix * 0.01

            volatility_penalty = curr_volatility * volatility_weight

            reward -= volatility_penalty

        if self.curr_step > 0:
            curr_date = self.trading_days[self.curr_step - 1]

            for ii, action in enumerate(action_list):

                if self.curr_step >= SHARPE_WINDOW:
                    prev_date = self.trading_days[self.curr_step - SHARPE_WINDOW]
                    stock_price = self.stock_data.xs(
                        self.ticker_list[ii], level="Ticker", axis=0).loc[prev_date:curr_date, "Adj Close"].values

                    if len(stock_price) > 0:
                        ema = np.mean(stock_price)

                        curr_price = self.stock_data.loc[(
                            self.ticker_list[ii], curr_date), "Adj Close"]

                        if len(stock_price) >= TREND_WINDOW:
                            returns = np.diff(stock_price) / stock_price[:-1]
                            volatility = np.std(returns[-TREND_WINDOW:])
                            trend_reward = BASE_TREND_REWARD * (1 + volatility)
                        else:
                            trend_reward = BASE_TREND_REWARD

                        if (action > 0 and curr_price > ema) or (action < 0 and curr_price < ema):
                            reward += trend_reward * abs(action)

        return reward

    def _get_reward_ryan_7(self, action_list):
        """
        Reward function for stock trading environment:
            - Calculate daily return (ratio of portfolio value to previous portfolio
            value)
            - Calculate short-term Sharpe ratio over three different windows with
            different adjustable weighting (short, medium and long)
            - Dynamic volatility weighting based on VIX index
            - Dynamically adjusted risk aversion (drawdown penalty)
            - Dynamic trend following with adjustable base weight

        Parameters:
            action_list (list): A list of actions for each stock in the portfolio

        Returns:
            reward (float): The reward signal for a given series of actions
        """

        SHARPE_WINDOW_SHORT = 20
        SHARPE_WINDOW_MED = 60
        SHARPE_WINDOW_LONG = 100
        BASE_TREND_REWARD = 0.1
        TREND_WINDOW = 20

        daily_return = (self.portfolio_value -
```

```python
                        self.previous_portfolio) / self.previous_portfolio

        self.returns.append(daily_return)
        self.max_portfolio = max(self.max_portfolio, self.portfolio_value)

        reward = 0

        curr_vix = self._get_vix()

        drawdown_threshold = curr_vix * 0.005

        drawdown = (self.max_portfolio - self.portfolio_value) / \
            self.max_portfolio

        if drawdown > drawdown_threshold:
            reward -= drawdown

        if len(self.returns) >= SHARPE_WINDOW_SHORT:
            ret = np.array(self.returns[-SHARPE_WINDOW_SHORT:])
            sharpe_ratio = np.sqrt(SHARPE_WINDOW_SHORT) * np.mean(ret) / np.std(ret)

            reward += sharpe_ratio

        if len(self.returns) >= SHARPE_WINDOW_MED:
            ret = np.array(self.returns[-SHARPE_WINDOW_MED:])
            sharpe_ratio = np.sqrt(SHARPE_WINDOW_MED) * np.mean(ret) / np.std(ret)

            reward += sharpe_ratio * 0.5

        if len(self.returns) >= SHARPE_WINDOW_LONG:
            ret = np.array(self.returns[-SHARPE_WINDOW_LONG:])
            sharpe_ratio = np.sqrt(SHARPE_WINDOW_LONG) * np.mean(ret) / np.std(ret)

            reward += sharpe_ratio * 0.25

        if len(self.returns) > 0:
            curr_volatility = np.std(
                np.array(self.returns[-min(SHARPE_WINDOW_SHORT, len(self.returns)):]))

            volatility_weight = curr_vix * 0.01

            volatility_penalty = curr_volatility * volatility_weight

            reward -= volatility_penalty

        if self.curr_step > 0:
            curr_date = self.trading_days[self.curr_step - 1]

            for ii, action in enumerate(action_list):

                if self.curr_step >= SHARPE_WINDOW_SHORT:
                    prev_date = self.trading_days[self.curr_step - SHARPE_WINDOW_SHORT]
                    stock_price = self.stock_data.xs(
                        self.ticker_list[ii], level="Ticker", axis=0).loc[prev_date:curr_date, "Adj Close"].values

                    if len(stock_price) > 0:
                        ema = np.mean(stock_price)

                        curr_price = self.stock_data.loc[(
                            self.ticker_list[ii], curr_date), "Adj Close"]

                        if len(stock_price) >= TREND_WINDOW:
                            returns = np.diff(stock_price) / stock_price[:-1]
                            volatility = np.std(returns[-TREND_WINDOW:])
                            trend_reward = BASE_TREND_REWARD * (1 + volatility)
                        else:
                            trend_reward = BASE_TREND_REWARD

                        if (action > 0 and curr_price > ema) or (action < 0 and curr_price < ema):
                            reward += trend_reward * abs(action)

        return reward

    def _get_observation_ryan_0(self):
        """
        Returns observations for current time step:

        Parameters:

        Returns:
            observation (list): The owned shares, current and previous opening
            prices, and volume for each stock in the portfolio
        """

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

```python
        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))

        return observation

    def _get_vix(self):
        """
        Retrieves the VIX index for the current trading day:

        Parameters:

        Returns:
            curr_vix (float): The VIX opening price for the current trading day
        """

        curr_vix = self.vix["Open"].iloc[-1]
        return curr_vix
```