OPTION TEMPLATE:

The provided code outlines a template for an automated trading strategy focusing on options trading with popular Indian indices like NIFTY and BANKNIFTY. It integrates with brokerage APIs to execute trades based on predefined criteria. Here's a detailed explanation of its components and functionalities:

Disclaimer Section

The disclaimer emphasizes the educational purpose of the script and cautions users about the risks associated with automated trading. It disclaims liability for any financial outcomes resulting from the use of the script.

Import Statements

The script imports necessary modules such as `datetime` for handling date and time operations, `time` for delays, `pandas` for data manipulation, and `requests` for HTTP requests. It also anticipates imports for broker-specific modules and helper functions, which would need to be defined elsewhere.

User-defined Inputs

Variables are defined to set up the strategy's parameters:

- **Entry time** (`entryHour`, `entryMinute`, `entrySecond`): When the strategy starts each day.
- **Stock index** (`stock`): The target index for trading (NIFTY or BANKNIFTY).
- **Option specifics** (`otm`, `SL_point`, `target_point`, `SL_percentage`, `target_percentage`): Define the strategy for selecting options and setting stop loss (SL) and targets.
- **Trade management** (`for_every_x_point`, `trail_by_y_point`): Parameters for trailing stop losses.
- **Trading volume and mode** (`qty`, `papertrading`): Quantity of options to trade and a flag for paper (simulated) vs. live trading.

Broker Flags

Flags (`shoonya_broker`, `nuvama_broker`, etc.) indicate which broker's API to use. The script includes placeholders for importing and initializing connections with various brokers. This is done using the helper function.

Core Functionality

- **findStrikePriceATM()**: Calculates the At The Money (ATM) strike price for options based on the current Last Traded Price (LTP) of the index. This involves finding the closest strike price and adjusting it for Out of The Money (OTM) options.

    - name: Initially we will find the name of the index (in the broker format).
    - intExpiry : Then we will find the current weekly expiry (using helper file).
    - ltp: We will get the current ltp of index.
    - Closest_Strike : This is the round off value of the ltp.
    - Closest_strike_ce and closest_strike_pe : These are the strike prices we want to trade in. It uses “otm” variable that we have defined earlier.
    - atmCE and atmPE : This is the option format that we are going to trade.
- **findStrikePricePremium()**: Finds the strike price based on the premium prices that we give.
    - name: Initially we will find the name of the index (in the broker format).
    - intExpiry : Then we will find the current weekly expiry (using helper file).
    - ltp: We will get the current ltp of index.
    - strikeList : We are creating a list of strikeprices, which can have the premium closest to the value we want.
    - In the loop (for strike in strikeList), the code goes through each strike that we add in strikeList. Then it find the current ltp of that strike (using findManualPrice() function). It calculates the ltp-premium = diff. We want this diff to be lowest. The lowest diff means that the strike price is closest to our intended premium.
    - Closest_strike_ce and closest_strike_pe : These are the strike prices we want to trade in.
    - atmCE and atmPE : This is the option format that we are going to trade.
- **takeEntry()**: Based on the calculated strike prices, it places entry orders for both Call (CE) and Put (PE) options. It logs entry prices and potential profits/losses (PnL).
    - Find the current ltp of atmCE and atmPE
    - Find ceSL and ceTarget if we want to put sl and target in points/percentage basis.
    - Place 2 orders using placeOrder1 function.
    - Call the exitPosition.
- **exitPosition()**: Monitors the positions for exit conditions, which could be based on SL, target achievements, or specific time criteria. Executes exit orders accordingly.
    - traded: this is a variable that will tell us whether our trades are exited or not. Initially its value is “No”
    - There is a while loop, which will keep running till trade == “No”. In this loop, we are getting the ltp of both CE and PE. And then we are checking if stoploss is hit, target is hit or time exit hit.
    - Once, any of the above is hit, that leg will be squared off. And traded = “CE” (if CE is hit) or trade = “PE” (if PE is hit).
    - Then we have 2 more while loops based on which leg is remaining. First we are making the SL to the entry price. And again we are checking if SL is hit, target is hit or time exit is hit. And it will take the second leg trade. And make traded = “Close”. Strategy is completed.
- **placeOrder1()**: A wrapper function that abstracts the order placement process, interfacing with the selected broker's API based on the paper trading flag and other parameters.
- **findManualPrice()**: This is a wrapper function to get the ltp of any symbol using direct broker function.

- **checkTime_tofindStrike()**: A loop that waits until the specified entry time to start the strategy. It checks continuously until the current time matches the `startTime` and then proceeds to execute `findStrikePriceATM()`.

Once the code runs fully, the paper trading results are stored in a csv file.

Execution Flow

The script starts by calling `checkTime_tofindStrike()`, initiating the strategy at the designated time. It includes an optional section for handling re-entry or additional checks after a brief pause, indicating the strategy might attempt to find new opportunities within the same trading day. Finally, it logs the day's trading activity, including PnL, to a CSV file for record-keeping.

Conclusion

This script provides a structured approach to automate options trading strategies with customizable parameters and integration with multiple brokerage platforms. However, the actual trading logic, broker API interactions, and error handling need to be implemented by the user based on their specific requirements, trading strategy, and the API documentation of the chosen broker. It's crucial to test the strategy extensively in a simulated environment before any live deployment to understand its behavior and potential financial implications.

INDICATOR TEMPLATE:

This Python script is designed for algorithmic trading, utilizing various technical indicators to make trading decisions. It's structured to run continuously, fetching market data at specified intervals, calculating indicators, and making trading decisions based on those indicators. Below is a detailed explanation and documentation of its components and functionality.

Disclaimer

The script begins with a disclaimer highlighting that it is for educational purposes only, emphasizing the risks involved in algorithmic trading and the importance of due diligence.

Imports and Library Initialization

- **Standard Libraries**: `time`, `math`, `datetime`, `os` for basic operations, timing, and date manipulations.
- **Third-Party Libraries**: `pandas` for data manipulation, `ta` (Technical Analysis Library in Python) for calculating technical indicators, `requests` for HTTP requests, `pandas_ta` as an additional technical analysis library.
- **Retry Mechanism**: `HTTPAdapter` and `Retry` from `requests` and `urllib3` for robust HTTP request handling.

Broker Integration

- The `importLibrary` function dynamically imports broker-specific modules and sets up connections based on the broker flags (`shoonya_broker`, `nuvama_broker`, etc.). It supports multiple brokers, indicating a versatile approach to trading across different platforms.

Configuration

- Global variables are defined for broker flags, indicating which brokers are to be used.
- `stock`, `checkInstrument`, `timeFrame`, `qty`, `tradeCEoption`, `tradePEoption`, `papertrading`, `otm`, `sl_point`, and `target_point` are configured for trading parameters like the instrument to trade, timeframe for analysis, quantity, and others.
- checkInstrument is the variable on which we will put all the indicators. So if you are looking to use equity, mcx, currency, then put the exact symbol (as per your broker format) in this checkInstrument.
- timeFrame is the time frame of the candles that we will be using.
- qty is quantity and not lot.
- papertrading 0 means its paper trading mode. If it is 1, then live trades will be placed.
- st variable tells us what is the current status of the strategy.
    - # 0 means no trde, but want to enter

- # 1 means buy trade.
- # 2 means sell trade.
- # -1 no trade and dont want to enter
- # 3 means check for stop and reverse

Main Functions

- `findStrikePriceATM` functions: This is similar to OPTION TEMLATE. Only difference is that it will take the entry in single direction now based on the input parameter passed (cepe). If cepe is "CE", then it will buy CE, otherwise it will but PE. The symbol that is traded is either saved as tradeCEoption or tradePEoption.
- `exitPosition` functions: We will pass which symbol we want to exit (either tradeCEoption or tradePEopion) and it will SELL that.
- placeOrder1: Wrapper function to place order in respective broker.
- getHistorical1: Wrapper function to get the OHLCV data of any symbol. It uses 3 parameters: the first parameter is the symbol, the second parameter is the timeframe and the third parameter is for how many historical days we want to get the data for.
- Calculate_heiken_ashi: This converts the normal OHLC data into heiken ashi OHLC.

Main Trading Logic

- A loop runs continuously, checking the current time and executing its core logic at the start of new timeframes based on the `timeFrame` setting.
- `getHistorical1` fetches historical OHLC data for the specified instrument and timeframe. This data is crucial for calculating technical indicators.
- The historical data (timeframe) is converted to lists (open, high, low, close, volume).
- If needed, we can find the heikenashi OHLC using the calculate_heiken_ashi function.
- Technical indicators such as MACD, SMA, RSI, and Supertrend are calculated using the fetched data. These indicators inform the trading decisions. The way to get the indicator value is by using the pdf file: https://buildmedia.readthedocs.org/media/pdf/technical-analysis-library-in-python/latest/technical-analysis-library-in-python.pdf
- At the end of the timeframe, the code will go into elif part. This is where it is going to go for trading logic. Initially since we are not in trade, we will be checking st == 0. Now we can add the logic based on which we want to take the entry (both buy and sell). If buy entry triggers, we will make st = 1, find sl, target, and take trade. If sell entry triggers, we will make st = 2, find sl, target, and take trade.
- Then we will check for st == 1 (that means we are in buy trade). This code is doing everything on closing basis. It will check if sl hit/target hit/indicator exit. And consequently it will take exit trade (using exitPosition). We also update the value of st accordingly.
- Same thing is checked for st == 2 (that means we are in sell trade) and for st == 3 (that means we want to go for stop and reverse).

- Finally we check for time exit. It will exit the open position based on the st value.

Saving Data and End-of-Day Logic

- Market data is saved to a CSV file for record-keeping, allowing for offline analysis and backtesting.
- The script includes logic for handling trading positions at the end of the trading day (EOD Exit).

Notes for Use

- The code is a framework for developing an algorithmic trading strategy. It requires filling in specific functionalities, such as data fetching (`getHistorical1`), order placement (`placeOrder1`), and defining the broker-specific logic within `importLibrary`.
- The script must be customized with actual trading logic, API keys, and broker details. Extensive testing and validation in a simulated environment are recommended before any live trading.
- Proper error handling, logging, and risk management strategies should be implemented to safeguard against unforeseen issues.

This script demonstrates the structure and components necessary for building an algorithmic trading bot but requires significant customization and completion to function with live market data and execute actual trades.

LEVEL BASED TEMPLATE:

This Python script is designed for algorithmic trading, utilizing levels to make trading decisions. It's structured to run continuously, fetching market data at specified intervals, and making trading decisions based on those levels. Below is a detailed explanation and documentation of its components and functionality.

Disclaimer

The script begins with a disclaimer highlighting that it is for educational purposes only, emphasizing the risks involved in algorithmic trading and the importance of due diligence.

Imports and Library Initialization

- **Standard Libraries**: `time`, `math`, `datetime`, `os` for basic operations, timing, and date manipulations.
- **Third-Party Libraries**: `pandas` for data manipulation, `ta` (Technical Analysis Library in Python) for calculating technical indicators, `requests` for HTTP requests, `pandas_ta` as an additional technical analysis library.
- **Retry Mechanism**: `HTTPAdapter` and `Retry` from `requests` and `urllib3` for robust HTTP request handling.

Broker Integration

- The `importLibrary` function dynamically imports broker-specific modules and sets up connections based on the broker flags (`shoonya_broker`, `nuvama_broker`, etc.). It supports multiple brokers, indicating a versatile approach to trading across different platforms.

Configuration

- Global variables are defined for broker flags, indicating which brokers are to be used.
- `stock`, `checkInstrument`, `timeFrame`, `qty`, `tradeCEoption`, `tradePEoption`, `papertrading`, `otm`, `sl_percentage`, and `target_percentage` are configured for trading parameters like the instrument to trade, timeframe for analysis, quantity, and others.
- entryHour, entryMinute, entrySecond is used to specify the strategy starting time.
- checkInstrument is the variable on which we will put all the indicators. So if you are looking to use equity, mcx, currency, then put the exact symbol (as per your broker format) in this checkInstrument.
- timeFrame is the time frame of the candles that we will be using.
- qty is quantity and not lot.
- papertrading 0 means its paper trading mode. If it is 1, then live trades will be placed.

- st variable tells us what is the current status of the strategy.
    - # 0 means no trde, but want to enter
    - # 1 means buy trade.
    - # 2 means sell trade.
    - # -1 no trade and dont want to enter
    - # 3 means check for stop and reverse
- Upper level and lowerlevel are the variables which will tell the price at which we will take the trade.
- Maxtrade tells the max number of trades that the strategy will take.
- number_of_trade shows the current number of trades that have been executed.

Main Functions

- `findStrikePriceATM` functions: This is similar to OPTION TEMLATE. Only difference is that it will take the entry in single direction now based on the input parameter passed (cepe). If cepe is "CE", then it will buy CE, otherwise it will but PE. The symbol that is traded is either saved as tradeCEoption or tradePEoption.
- `exitPosition` functions: We will pass which symbol we want to exit (either tradeCEoption or tradePEopion) and it will SELL that.
- placeOrder1: Wrapper function to place order in respective broker.
- getHistorical1: Wrapper function to get the OHLCV data of any symbol. It uses 3 parameters: the first parameter is the symbol, the second parameter is the timeframe and the third parameter is for how many historical days we want to get the data for.
- Calculate_heiken_ashi: This converts the normal OHLC data into heiken ashi OHLC.

Main Trading Logic

- An infinite loops runs in the beginning waiting for the start to be reached.
- A loop runs continuously, checking the current time and executing its core logic at the start of new timeframes based on the `timeFrame` setting.
- `getHistorical1` fetches historical OHLC data for the specified instrument and timeframe. This data is crucial for calculating technical indicators.
- The historical data (timeframe) is converted to lists (open, high, low, close, volume).
- This is where we are checking if ltp has broker the upper level or lower level. Before we take the entry, we also check that the number_of_trade is less than max_trades. If buy entry triggers, we will make st = 1, find sl, target, and take trade. If sell entry triggers, we will make st = 2, find sl, target, and take trade.
- Then we will check for st == 1 (that means we are in buy trade). This code is doing everything on ltp basis. It will get the ltp and then it will check if sl hit/target hit/indicator exit. And consequently it will take exit trade (using exitPosition). We also update the value of st accordingly.
- Same thing is checked for st == 2 (that means we are in sell trade).

- Finally we check for time exit. It will exit the open position based on the st value.

Saving Data and End-of-Day Logic

- Market data is saved to a CSV file for record-keeping, allowing for offline analysis and backtesting.
- The script includes logic for handling trading positions at the end of the trading day (EOD Exit).

Notes for Use

- The code is a framework for developing an algorithmic trading strategy. It requires filling in specific functionalities, such as data fetching (`getHistorical1`), order placement (`placeOrder1`), and defining the broker-specific logic within `importLibrary`.
- The script must be customized with actual trading logic, API keys, and broker details. Extensive testing and validation in a simulated environment are recommended before any live trading.
- Proper error handling, logging, and risk management strategies should be implemented to safeguard against unforeseen issues.

This script demonstrates the structure and components necessary for building an algorithmic trading bot but requires significant customization and completion to function with live market data and execute actual trades.