# Backtesting a Keltner Channel Breakout Strategy with Backtrader

PyQuantLab Following 10 min read · Jun 5, 2025

Let’s try a Keltner Channel breakout trading strategy and backtest it using the `backtrader` library. We'll break down the code step-by-step, explaining the logic behind the custom Keltner Channel indicator, the strategy itself, and how to run a backtest to evaluate its performance.

Looking to supercharge your algorithmic trading research? The Ultimate Algorithmic Strategy Bundle has you covered with over 80 Python strategies, fully documented in comprehensive PDF

manuals:
Ultimate Algorithmic Strategy Bundle

## Keltner Channels and Breakout Strategies

Keltner Channels are volatility-based technical indicators that consist of three lines:

1. Middle Line: Typically an Exponential Moving Average (EMA) of the price.
2. Upper Band: The Middle Line plus a multiple of the Average True Range (ATR).
3. Lower Band: The Middle Line minus a multiple of the Average True Range (ATR).

These channels widen during periods of high volatility and narrow during low volatility.

Breakout strategies aim to capitalize on significant price movements that "break out" of a defined range or pattern. In the context of Keltner Channels, a breakout strategy might involve:

- Going Long (Buy): When the price closes above the upper Keltner Channel band, suggesting strong upward momentum.

- Going Short (Sell): When the price closes below the lower Keltner Channel band, suggesting strong downward momentum.

This particular strategy will use the previous day's close relative to the previous day's Keltner bands for entry signals and the current EMA for exit signals.

## Code Breakdown

Let's dive into the Python code.

### 1. Imports and Setup

First, we import the necessary libraries:

- `backtrader` (`bt`): The core backtesting framework.
- `yfinance` (`yf`): To download historical market data.
- `pandas` (`pd`): For data manipulation (though minimally used directly in this script, `yfinance` returns DataFrames).
- `numpy` (`np`): For numerical operations (also often used indirectly).
- `matplotlib.pyplot` (`plt`): For plotting the results.
- `datetime`: For handling dates (though `backtrader` often manages this internally for its data feeds).

- `warnings` : To suppress irrelevant warnings.

```
import backtrader as bt
import yfinance as yf
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from datetime import datetime
import warnings

warnings.filterwarnings("ignore") # Hide warnings for cleaner outpu
```

## 2. Custom Keltner Channel Indicator

`backtrader` has many built-in indicators, but creating a custom one provides flexibility. Here, `KeltnerChannel` is defined as a subclass of `bt.Indicator` .

```
# Custom Keltner Channel Indicator
class KeltnerChannel(bt.Indicator):
    """
    Keltner Channel indicator with EMA centerline and ATR-based ban
    """
    lines = ('mid', 'top', 'bot')  # Output lines: middle, top, bot
    params = (
        ('ema_period', 30),      # Period for the Exponential Movin
        ('atr_period', 14),      # Period for the Average True Rang
        ('atr_multiplier', 1.0), # Multiplier for ATR to set band w
    )

    def __init__(self):
        # EMA for centerline
```

```
            self.lines.mid = bt.indicators.EMA(self.data.close, period=

            # ATR for band width
            atr = bt.indicators.ATR(self.data, period=self.params.atr_p

            # Upper and lower bands
            self.lines.top = self.lines.mid + (atr * self.params.atr_mu
            self.lines.bot = self.lines.mid - (atr * self.params.atr_mu
```

- `lines = ('mid', 'top', 'bot')` : Defines the names of the lines this indicator will produce.
- `params` : A tuple of tuples defining configurable parameters for the indicator:
- `ema_period` : The lookback period for the EMA (centerline).
- `atr_period` : The lookback period for the ATR.
- `atr_multiplier` : How many ATRs to add/subtract from the EMA to form the bands.
- `__init__(self)` :
- `self.lines.mid` : Calculates the EMA of the closing price.
- `atr` : Calculates the ATR of the data.
- `self.lines.top` and `self.lines.bot` : Calculate the upper and lower bands by adding/subtracting the (ATR * multiplier) from the middle line.

### 3. Keltner Channel Breakout Strategy

This class, `KeltnerBreakoutStrategy`, inherits from `bt.Strategy` and implements the trading logic.

```
# Keltner Channel Breakout Strategy
class KeltnerBreakoutStrategy(bt.Strategy):
    params = (
        ('ema_period', 30),         # EMA period for Keltner Channel
        ('atr_period', 14),         # ATR period for Keltner Channel
        ('atr_multiplier', 1.0),    # ATR multiplier for Keltner Cha
        ('printlog', True),         # Whether to print log messages
    )

    def log(self, txt, dt=None):
        """Logging utility"""
        if self.params.printlog:
            dt = dt or self.datas[0].datetime.date(0) # Use current
            print(f'{dt.isoformat()}: {txt}')

    def __init__(self):
        # Keep reference to the "close" and "open" lines in the dat
        self.dataclose = self.datas[0].close
        self.dataopen = self.datas[0].open

        # Initialize Keltner Channel using parameters passed to the
        self.keltner = KeltnerChannel(
            self.data, # Pass the main data feed to the indicator
            ema_period=self.params.ema_period,
            atr_period=self.params.atr_period,
            atr_multiplier=self.params.atr_multiplier
        )

        # Track order and buy price/commission
        self.order = None
        self.buyprice = None
        self.buycomm = None

        # Add indicators to plot (ensure they are plotted on the ma
        self.keltner.plotinfo.subplot = False # Plot on same chart
        self.keltner.plotlines.mid._plotskip = False # Ensure mid l
```

```
        self.keltner.plotlines.top._plotskip = False # Ensure top l
        self.keltner.plotlines.bot._plotskip = False # Ensure bot l

    def notify_order(self, order):
        """Track order execution status"""
        if order.status in [order.Submitted, order.Accepted]:
            # Broker has accepted/submitted the order - Nothing to
            return

        if order.status in [order.Completed]:
            if order.isbuy():
                self.log(f'BUY EXECUTED: Price: {order.executed.pri
                         f'Cost: {order.executed.value:.2f}, Comm:
                self.buyprice = order.executed.price
                self.buycomm = order.executed.comm
            elif order.issell(): # Sell
                self.log(f'SELL EXECUTED: Price: {order.executed.pr
                         f'Cost: {order.executed.value:.2f}, Comm:

            self.bar_executed = len(self) # Record bar number when

        elif order.status in [order.Canceled, order.Margin, order.R
            self.log('Order Canceled/Margin/Rejected')

        self.order = None # Reset order status after handling

    def notify_trade(self, trade):
        """Track completed trades"""
        if not trade.isclosed:
            return # Trade is not closed yet

        self.log(f'OPERATION PROFIT: Gross {trade.pnl:.2f}, Net {tr

    def next(self):
        """Main strategy logic, called for each bar of data"""
        # Skip if we don't have enough data for indicators
        if len(self.data) < max(self.params.ema_period, self.params
            return

        # Check if we have a pending order
        if self.order:
            return # Do nothing if an order is pending

        # Get previous day's values for signal generation
```

```
        # Ensure there's at least one previous bar available (index
        if len(self.data) < 2: # Needs current bar (0) and previous
            return

        prev_close = self.dataclose[-1]      # Previous bar's close
        prev_upper = self.keltner.top[-1]    # Previous bar's Keltn
        prev_lower = self.keltner.bot[-1]    # Previous bar's Keltn
        current_ema = self.keltner.mid[0]    # Current bar's Keltne
        current_close = self.dataclose[0]    # Current bar's close

        if not self.position:  # Not in the market
            # Long entry: Previous close broke above Previous Keltn
            if prev_close > prev_upper:
                self.log(f'BUY CREATE: Price {self.dataopen[0]:.2f}
                self.order = self.buy() # Place buy order at curren

            # Short entry: Previous close broke below Previous Kelt
            elif prev_close < prev_lower:
                self.log(f'SELL CREATE: Price {self.dataopen[0]:.2f
                self.order = self.sell() # Place sell order at curr

        else:  # Already in the market, check for exit conditions
            # Exit conditions based on current close vs EMA (Keltne
            if self.position.size > 0:  # If in a Long position
                if current_close < current_ema:
                    self.log(f'CLOSE LONG: Price {current_close:.2f
                    self.order = self.close() # Close long position

            elif self.position.size < 0:  # If in a Short position
                if current_close > current_ema:
                    self.log(f'CLOSE SHORT: Price {current_close:.2
                    self.order = self.close() # Close short positio
```

- `params` : Strategy-specific parameters, including those for the Keltner Channel and a `printlog` flag.
- `log()` : A helper function for printing messages with timestamps.

- `__init__()`:
- Stores references to `self.datas[0].close` (closing price of the primary data feed) and `self.datas[0].open`.
- Instantiates the custom `KeltnerChannel` indicator.
- Initializes `self.order` to `None` to track pending orders.
- Configures plotting options for the Keltner Channel lines so they appear on the main price chart.
- `notify_order()`: Called by `backtrader` when an order's status changes. It logs executed buy/sell orders and resets `self.order`.
- `notify_trade()`: Called by `backtrader` when a trade is completed (closed). It logs the profit or loss.
- `next()`: This is the heart of the strategy, executed on each new bar of data.
- Data Check: Ensures enough data is available for indicator calculation.
- Pending Order Check: If an order is already pending, it does nothing.
- Signal Generation:
- It uses the previous bar's close (`prev_close`) and the previous bar's Keltner bands (`prev_upper`, `prev_lower`) for entry

signals. This is a common approach to avoid look-ahead bias, ensuring decisions are based on completed information.

- Entry Logic (if not in market):
- Long Entry: If `prev_close` was greater than `prev_upper`, a buy order is created for the current bar (typically executed at the open of the current bar).
- Short Entry: If `prev_close` was less than `prev_lower`, a sell order is created.
- Exit Logic (if in market):
- It uses the current bar's close (`current_close`) and the current bar's EMA (`current_ema`) for exit signals.
- Exit Long: If in a long position and `current_close` falls below `current_ema`, the position is closed.
- Exit Short: If in a short position and `current_close` rises above `current_ema`, the position is closed.

### 4. Custom Trade Analyzer (Optional)

This is a simple custom analyzer to show how one might track specific metrics. `backtrader` already provides many comprehensive analyzers.

```
# Analyzer to track additional metrics (example)
class TradeAnalyzer(bt.Analyzer):
```

```
    def create_analysis(self):
        self.rets = {} # Placeholder for returns, not fully utilize
        self.vals = [] # List to store portfolio values over time

    def notify_cashvalue(self, cash, value):
        """Called on each bar with current cash and portfolio value
        self.vals.append(value) # Store the current portfolio value

    def get_analysis(self):
        """Return the analysis results"""
        return {'final_value': self.vals[-1] if self.vals else 0, #
                'max_value': max(self.vals) if self.vals else 0} #
```

This analyzer primarily tracks the portfolio value at each bar to find the final and maximum values.

### 5. `run_backtest()` Function

This function sets up and executes the backtest.

```
def run_backtest():
    """Main function to run the backtest"""

    # Parameters for the backtest
    TICKER = "BTC-USD"        # Asset to backtest
    START_DATE = "2023-01-01" # Start date for data
    END_DATE = "2024-12-31"   # End date for data
    INITIAL_CASH = 100000     # Starting portfolio value
    COMMISSION = 0.001        # Commission per trade (0.1%)

    # Keltner Channel Parameters (can be tuned)
    EMA_WINDOW = 30
    ATR_WINDOW = 14
    ATR_MULTIPLIER = 1.0
```

```
print(f"Downloading data for {TICKER}...")

# Download data using yfinance
# Applying user preference: auto_adjust=False and droplevel(1,
data = yf.download(TICKER, start=START_DATE, end=END_DATE, auto
data = data.droplevel(axis=1, level=1) # Simplify MultiIndex co

if data.empty:
    raise ValueError("No data downloaded. Check ticker or date

print(f"Data downloaded. Shape: {data.shape}")

# Create Backtrader data feed from the pandas DataFrame
data_bt = bt.feeds.PandasData(dataname=data)

# Create cerebro engine (the brain of backtrader)
cerebro = bt.Cerebro()

# Add data to cerebro
cerebro.adddata(data_bt)

# Add strategy to cerebro
cerebro.addstrategy(
    KeltnerBreakoutStrategy,
    ema_period=EMA_WINDOW,
    atr_period=ATR_WINDOW,
    atr_multiplier=ATR_MULTIPLIER,
    printlog=False  # Set to True for detailed logs during stra
)

# Set initial cash
cerebro.broker.setcash(INITIAL_CASH)

# Set commission
cerebro.broker.setcommission(commission=COMMISSION)

# Add analyzers to evaluate performance
cerebro.addanalyzer(bt.analyzers.TradeAnalyzer, _name='trades')
cerebro.addanalyzer(bt.analyzers.SharpeRatio, _name='sharpe')
cerebro.addanalyzer(bt.analyzers.DrawDown, _name='drawdown')
cerebro.addanalyzer(bt.analyzers.Returns, _name='returns')
cerebro.addanalyzer(TradeAnalyzer, _name='custom') # Add our cu

print(f'Starting Portfolio Value: ${cerebro.broker.getvalue():.
```

```
# Run backtest
results = cerebro.run() # This executes the backtest
strat = results[0]      # Get the executed strategy instance

print(f'Final Portfolio Value: ${cerebro.broker.getvalue():.2f}

# Print analysis results
print('\n--- PERFORMANCE ANALYSIS ---')

# Trade Analysis (wins, losses, etc.)
trade_analysis = strat.analyzers.trades.get_analysis()
if 'total' in trade_analysis and 'closed' in trade_analysis['to
    total_trades = trade_analysis['total']['closed']
    won_trades = trade_analysis.get('won', {}).get('total', 0)
    lost_trades = trade_analysis.get('lost', {}).get('total', 0

    print(f'Total Closed Trades: {total_trades}')
    print(f'Winning Trades: {won_trades}')
    print(f'Losing Trades: {lost_trades}')

    if total_trades > 0:
        win_rate = (won_trades / total_trades) * 100
        print(f'Win Rate: {win_rate:.2f}%')

    if 'won' in trade_analysis and 'pnl' in trade_analysis['won
        avg_win = trade_analysis['won']['pnl']['average']
        print(f'Average Win: ${avg_win:.2f}')

    if 'lost' in trade_analysis and 'pnl' in trade_analysis['lo
        avg_loss = trade_analysis['lost']['pnl']['average']
        print(f'Average Loss: ${avg_loss:.2f}')
else:
    print('No completed trades found')

# Returns Analysis
returns_analysis = strat.analyzers.returns.get_analysis()
if 'rtot' in returns_analysis: # Total return over the period
    total_return = returns_analysis['rtot'] * 100
    print(f'Total Return: {total_return:.2f}%')

# Sharpe Ratio (annualized)
sharpe_analysis = strat.analyzers.sharpe.get_analysis()
# The SharpeRatio analyzer might return None if there are not e
```

```
    if 'sharperatio' in sharpe_analysis and sharpe_analysis['sharpe
        sharpe_ratio = sharpe_analysis['sharperatio']
        print(f'Sharpe Ratio: {sharpe_ratio:.3f}')

    # Drawdown Analysis
    drawdown_analysis = strat.analyzers.drawdown.get_analysis()
    if 'max' in drawdown_analysis and 'drawdown' in drawdown_analys
        max_drawdown = drawdown_analysis['max']['drawdown']
        print(f'Max Drawdown: {max_drawdown:.2f}%')

    # Calculate Buy & Hold comparison
    if not data.empty:
        initial_price = data['Close'].iloc[0]
        final_price = data['Close'].iloc[-1]
        buy_hold_return = ((final_price / initial_price) - 1) * 100
        print(f'Buy & Hold Return for {TICKER}: {buy_hold_return:.2

    # Plot results
    print('\nGenerating plots...')

    # Configure plot size and style
    plt.rcParams['figure.figsize'] = [10, 6] # Set default figure s
    cerebro.plot(
        style='candlestick', # Use candlestick chart
        barup='green',       # Color for upward bars
        bardown='red',       # Color for downward bars
        volume=False,        # Do not plot volume subplot by defaul
        iplot=False          # Set to True if running in Jupyter no
    )

    plt.show() # Display the plot

    return cerebro, results
```

- Parameters: Defines the ticker, date range, initial capital, commission, and Keltner Channel parameters.

- Data Download: Fetches historical data using `yfinance`. Note: It correctly uses `auto_adjust=False` and `droplevel(axis=1, level=1)` as per your saved preferences.
- Cerebro Setup:
- `bt.Cerebro()`: Creates the main backtesting engine.
- `cerebro.adddata()`: Adds the historical data.
- `cerebro.addstrategy()`: Adds the `KeltnerBreakoutStrategy` with its parameters.
- `cerebro.broker.setcash()`: Sets the initial portfolio cash.
- `cerebro.broker.setcommission()`: Sets the trading commission.
- Analyzers: Adds various built-in analyzers (`TradeAnalyzer`, `SharpeRatio`, `DrawDown`, `Returns`) and the custom `TradeAnalyzer` to gather performance statistics.
- Run Backtest: `cerebro.run()` executes the strategy over the historical data.
- Performance Analysis: Extracts and prints key metrics from the analyzers:
- Total Trades, Winning/Losing Trades, Win Rate
- Average Win/Loss amounts
- Total Return
- Sharpe Ratio

- Maximum Drawdown
- Buy & Hold Comparison: Calculates the return of a simple buy-and-hold strategy for the same period as a benchmark.
- Plotting: `cerebro.plot()` generates a visual chart of the trades, price action, and indicators.

### 6. Main Execution Block

This standard Python construct ensures `run_backtest()` is called only when the script is executed directly.

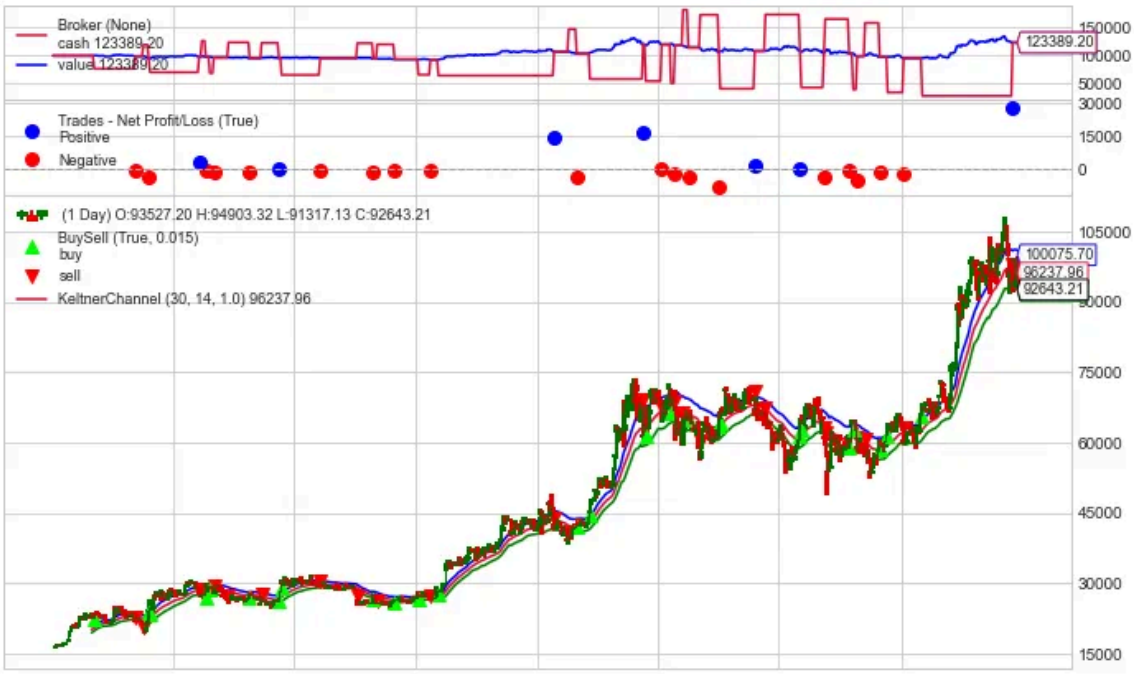
```
if __name__ == '__main__':
    try:
        cerebro, results = run_backtest()
        print("Backtest completed successfully!")

    except Exception as e:
        print(f"An error occurred: {e}")
        import traceback
        traceback.print_exc() # Print full traceback for debugging
```

It includes basic error handling to catch and display any exceptions during the backtest.

## Strategy Explanation Recap

- Indicator: Keltner Channel (EMA centerline, ATR-based upper/lower bands).
- Entry Signals (based on previous bar's data):
- Long: Previous day's close > Previous day's Keltner upper band.
- Short: Previous day's close < Previous day's Keltner lower band.
- Exit Signals (based on current bar's data):
- Exit Long: Current day's close < Current day's Keltner middle line (EMA).
- Exit Short: Current day's close > Current day's Keltner middle line (EMA).
- Order Execution: Orders are typically placed at the open of the current bar after a signal on the previous bar.




## Conclusion

This code provides a solid foundation for backtesting a Keltner Channel breakout strategy. By understanding its components, you can further experiment with different parameters, assets, and modifications to the trading logic itself. `backtrader` is a powerful tool that allows for complex strategy development and rigorous performance evaluation. Remember that past performance is not indicative of future results, and thorough testing and risk management are crucial in trading.

Algorithmic Trading | Python | Quantitative Finance | Backtrader | Backtesting

## Written by PyQuantLab

655 followers · 6 following

Following

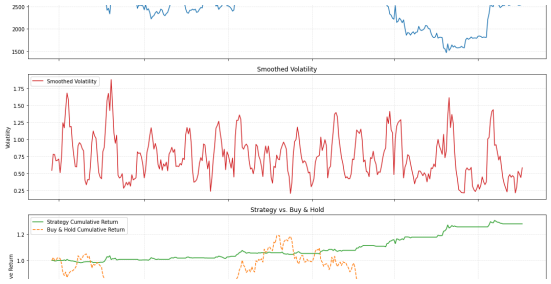
Your go-to place for Python-based quant tutorials, strategy deep-dives, and reproducible code. For more visit our website: www.pyquantlab.com

## No responses yet

Steven Feng CAI

What are your thoughts?

## More from PyQuantLab




PyQuantLab

## Volatility Clustering Trading Strategy with Python

Ultimate Algorithmic Strategy Bundle has you covered with over 80 Python...

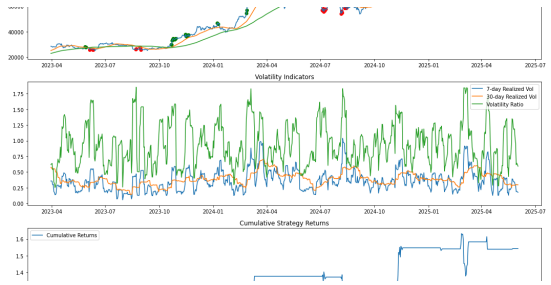
Jun 3 32 3


PyQuantLab

## An Algorithmic Exploration of Volume Spread Analysis...

Note: You can read all articles for free on our website: pyquantlab.com

Jun 9 54 1

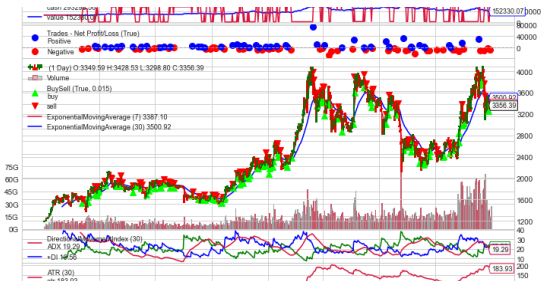



PyQuantLab

## Trend-Volatility Confluence Trading Strategy

The Ultimate Algorithmic Strategy Bundle has you covered with over 80...

Jun 3 60




PyQuantLab

## Building an Adaptive Trading Strategy with Backtrader: A...

Note: You can read all articles for free on our website: pyquantlab.com

Jun 4 64

See all from PyQuantLab

## Recommended from Medium

MarketMuse

### “I Let an AI Bot Trade for Me for 7 Days—It Made $8,000...

Subtitle: While you’re analyzing candlestick patterns, AI bots are fron...

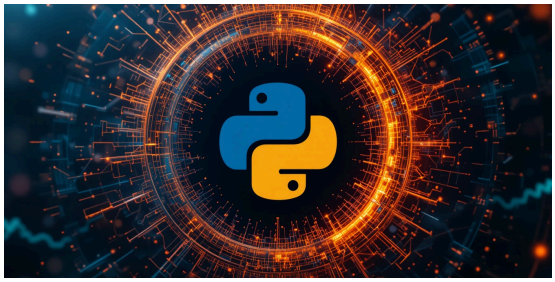
Jun 3 75 3

Swapnilphutane

### How I Built a Multi-Market Trading Strategy That Pass...

When I first got into trading, I had no plans of building a full-blown system....

6d ago 16 1



Andrew Rul Trading

## I Tested 5 Free Trading Bots. Here Are My Results

We've all seen them: "100% FREE Trading Bot!". "Automate your trades...

4d ago 8 2

Unicorn Day

## The Quest for the Perfect Trading Score: Turning...

Navigating the financial markets... it feels like being hit by a tsunami of da...

3d ago 43


In InsiderFinance Wire by Nayab Bhutta

## Don't Trade Without This! Why I Switched from...

The Indicator Addiction (That Almost Ruined My Trading)

Jun 12 152 3

FMZQuant

## Multi-Timeframe Dynamic Trend Detection System: EM...

Strategy Overview

Jun 4 1

See more recommendations