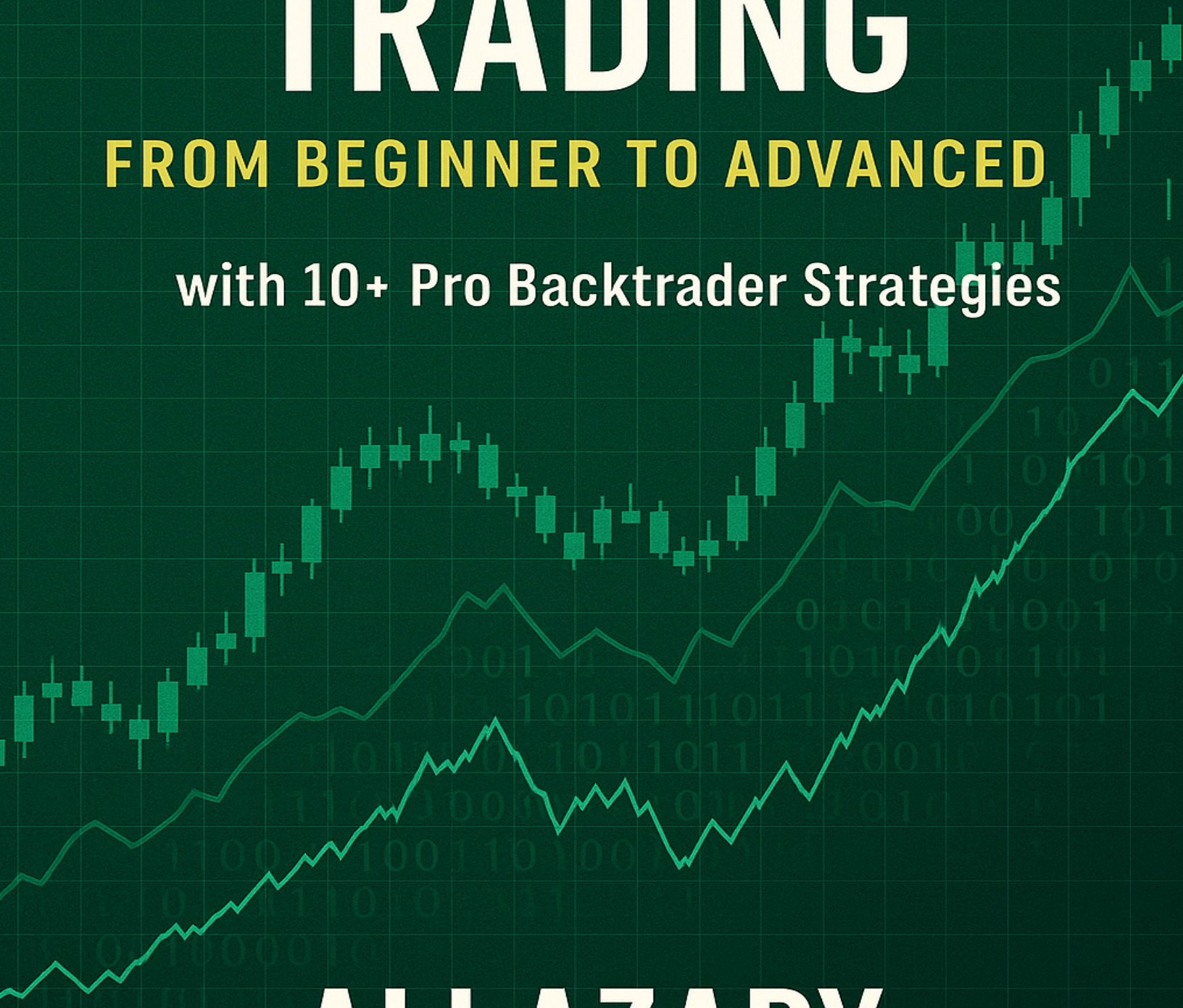
# ALGORITHMIC TRADING

## FROM BEGINNER TO ADVANCED

with 10+ Pro Backtrader Strategies



ALI AZARY

# Algorithmic Trading From Beginner to Advanced



# Preface

The Strategies in this book is a glimpse of the **Mega Backtrader Strategy Pack** — a collection of over **250 ready-to-use strategies** for your algorithmic trading with a **900+ pages manual**.

The **Mega Backtrader Strategy Pack** gives you **250+ fully-tested Python strategies** — adaptive filters, mean reversion, momentum, volatility breakouts, machine learning models & more — all ready to deploy in Backtrader.

### Why Traders Love This Pack

- Instant Impact — Deploy strategies and start testing today
- Zero Guesswork — Every strategy is documented, tested, and ready-to-run
- Future-Proof — Lifetime updates included
- Proven Variety — Adaptive, mean-reversion, momentum, volatility, and ML-driven models

What's Inside

✅ **250+ Strategies across multiple categories:**

- Adaptive Filters — Kalman, Hilbert, TRIX, Guppy MMA
- Mean Reversion — Bollinger, Keltner, pair trading, RSI/Stoch
- Momentum & Trend — MACD, ADX, Donchian, MA ribbons
- Volatility — ATR breakouts, Bollinger squeeze, VIX correlation
- Specialized ML & Advanced — HMM, decision trees, isolation forest, Hurst exponent

You Get

📂 Python `.py` scripts for direct Backtrader use
📄 PDF manuals detailing logic, parameters, and best practices
🎯 Get Instant Access — Download Today, Trade Tonight

👉 **Get it here**

# Chapter 1 — Introduction to Trading & Finance Concepts

Trading is both an art and a science.
While intuition and experience can help, modern trading — especially algorithmic trading — is grounded in **data, mathematics, and disciplined execution**.

This chapter covers the **core concepts, terminology, and quantitative measures** that will form the foundation for the strategies and Backtrader implementations you'll see later in this book.

---

## 1.1 What is Trading?

Trading is the act of **buying and selling assets** to profit from price movements.

Markets where trading takes place include:

- **Stock Market** — Shares of companies like Apple, Tesla.
- **Foreign Exchange (Forex)** — Currency pairs like EUR/USD, GBP/JPY.
- **Commodities** — Raw materials like gold, oil, wheat.
- **Cryptocurrencies** — Digital assets like Bitcoin, Ethereum.
- **Bonds** — Debt securities issued by governments or corporations.
- **Indices** — Market benchmarks like the S&P 500 or NASDAQ 100.

Algorithmic trading uses computer programs to execute trades based on predefined rules, often faster and more consistently than human traders.

---

# 1.2 Market Participants

Knowing who you are trading against helps you understand market behavior:

- **Retail Traders** — Individual traders managing their own accounts.
- **Institutional Investors** — Hedge funds, pension funds, asset managers.
- **Market Makers** — Firms that provide liquidity by posting continuous bid and ask prices.
- **High-Frequency Traders (HFTs)** — Algorithms exploiting microsecond opportunities.

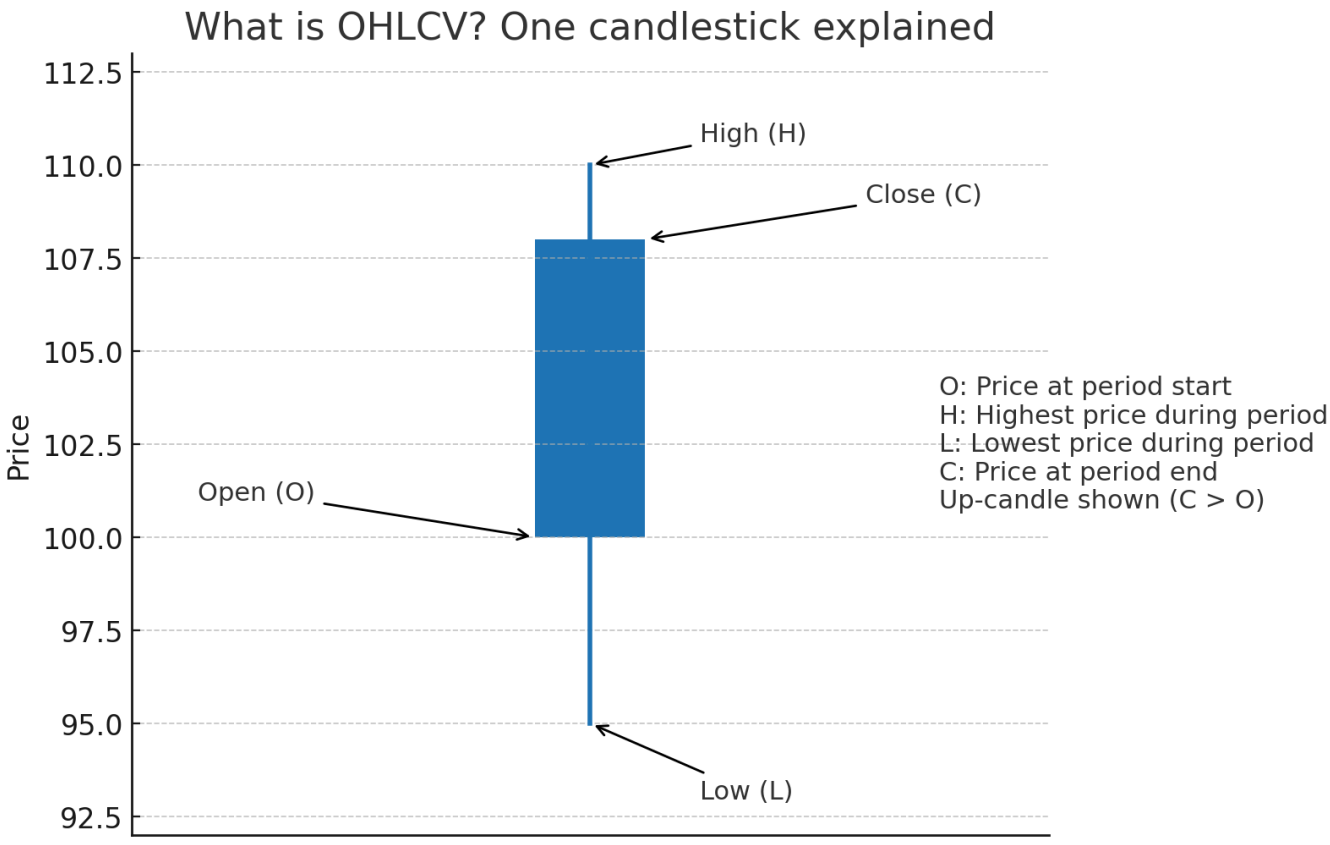
Institutions often have scale and speed advantages, but retail traders can be nimble, focus on niche markets, and use automation to level the playing field.

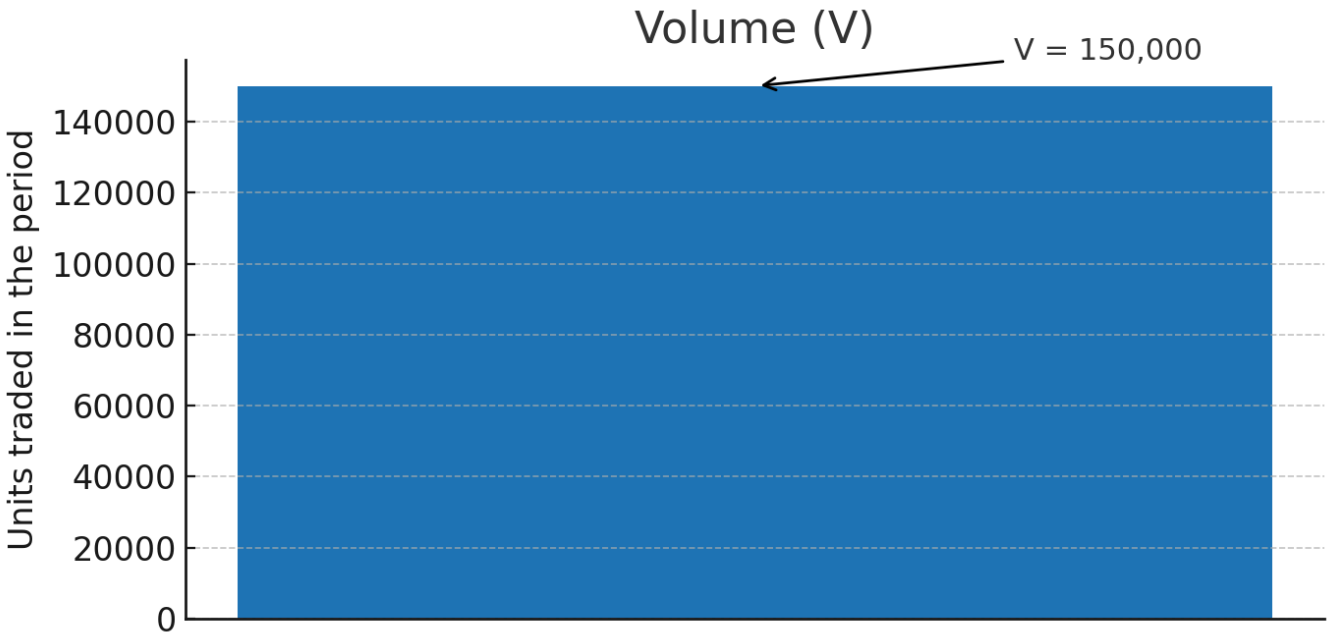
---

# 1.3 Understanding OHLCV Data

Trading platforms and APIs represent price movements with **OHLCV** data:

| Term | Definition | Example |
|---|---|---|
| **Open (O)** | Price at which the period starts | BTC opens at $29,500 at 09:00 |
| **High (H)** | Highest price reached in the period | BTC peaks at $29,700 |
| **Low (L)** | Lowest price reached in the period | BTC dips to $29,400 |
| **Close (C)** | Price at the end of the period | BTC closes at $29,650 |
| **Volume (V)** | Total traded quantity in the period | 120 BTC traded between 09:00–09:05 |




In **Backtrader**, OHLCV data is loaded into data feeds and accessed as:

```
self.data.open[0]
self.data.high[0]
self.data.low[0]
self.data.close[0]
self.data.volume[0]
```

# 1.4 Returns: Measuring Price Changes

Returns are the **percentage change in price** from one period to the next.

## 1.4.1 Simple Return

$$R_t = \frac{P_t - P_{t-1}}{P_{t-1}}$$

Example:
If yesterday's close was $30,000 and today's close is $31,000:

$$R_t = \frac{31,000 - 30,000}{30,000} = 0.0333 \ (3.33\%)$$

## 1.4.2 Logarithmic Return

$$r_t = \ln\left(\frac{P_t}{P_{t-1}}\right)$$

Log returns are preferred in many quantitative models because they are **time-additive**.

Python example:

```
import numpy as np
log_return = np.log(price_today / price_yesterday)
```

---

# 1.5 Volatility: Measuring Risk

Volatility is the **magnitude of price fluctuations** over time.
High volatility means larger and more frequent price swings.

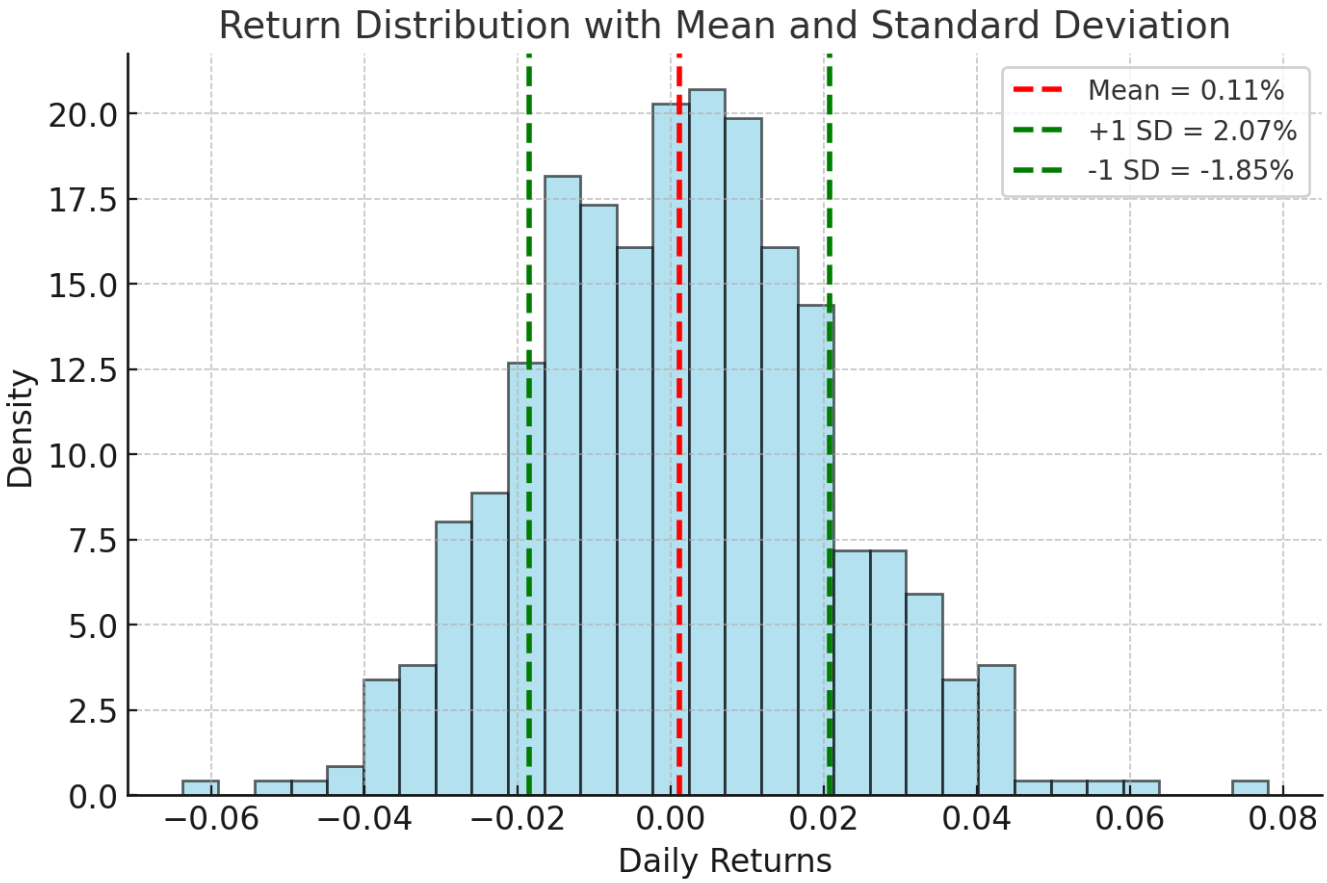
## 1.5.1 Standard Deviation

$$\sigma = \sqrt{\frac{\sum_{i=1}^{n}(R_i - \bar{R})^2}{n-1}}$$

Where:

- $R_i$ = individual returns
- $\bar{R}$ = average return

- $n$ = number of periods




Python example:

```
import numpy as np
volatility = np.std(returns)
```

---

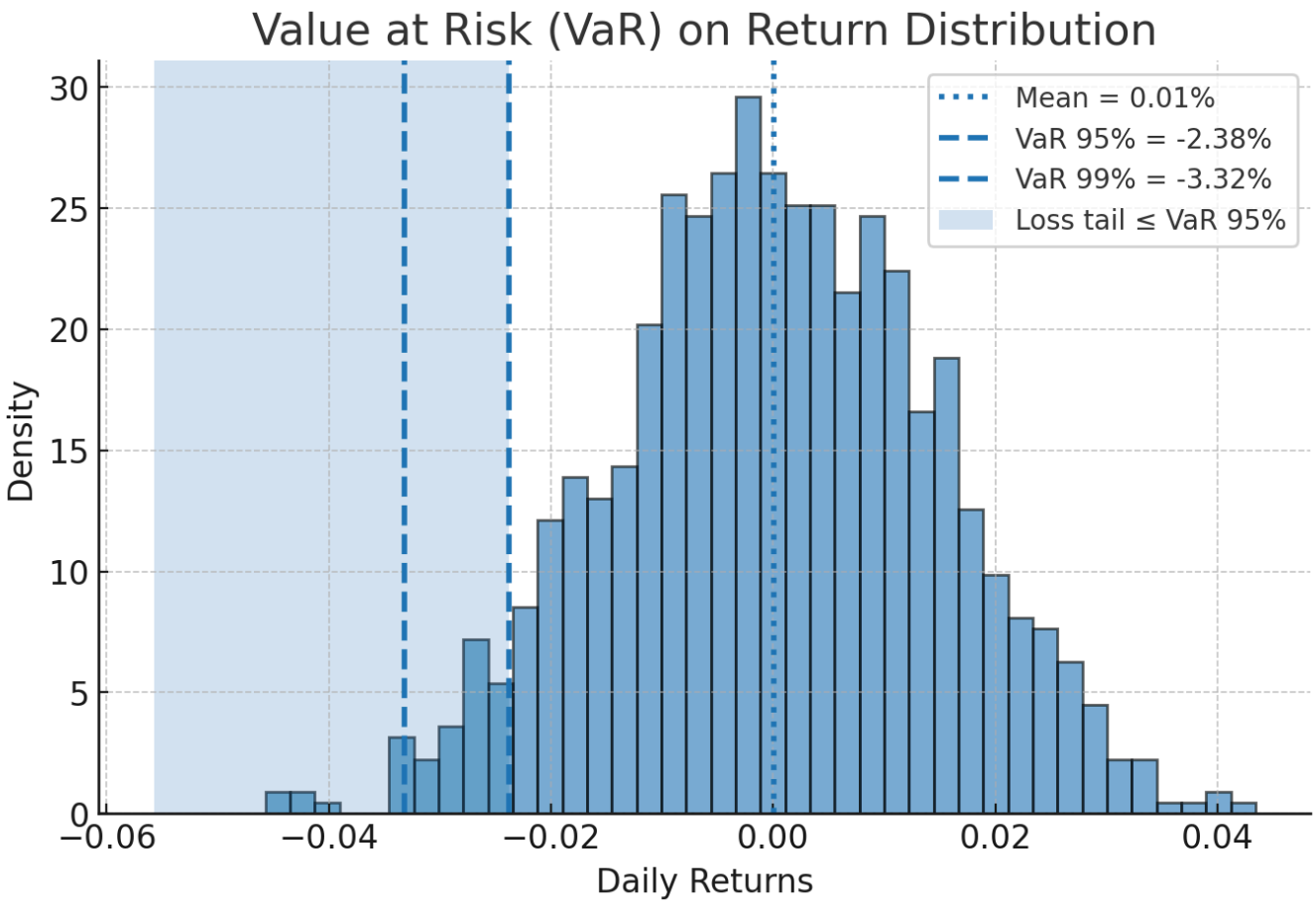
## 1.5.2 Value at Risk (VaR)

Value at Risk estimates the **maximum expected loss** over a given period at a certain confidence level.

Parametric VaR formula:

$$VaR_{95\%} = \bar{R} - z_{0.95} \cdot \sigma$$

Where $z_{0.95} \approx 1.65$ for 95% confidence.




Python example:

```
from scipy.stats import norm
confidence = 0.95
VaR_95 = mean_return - norm.ppf(confidence) * volatility
```

# 1.6 Key Risk and Performance Metrics

| Metric | Formula / Definition | Why It Matters |
|---|---|---|
| **Win Rate** | Winning trades / Total trades | Measures strategy accuracy |
| **Sharpe Ratio** | $(\bar{R} - R_f)/\sigma$ | Risk-adjusted performance |
| **Sortino Ratio** | $(\bar{R} - R_f)/\sigma_{down}$ | Penalizes only downside volatility |
| **Max Drawdown** | Largest % drop from equity peak | Measures worst historical loss |
| **Profit Factor** | Gross profit / Gross loss | >1.5 is desirable |

# 1.7 Order Types

- **Market Order** — Executes immediately at best price.
- **Limit Order** — Executes only at or better than a set price.
- **Stop Loss** — Closes position when price hits loss threshold.
- **Stop-Limit Order** — Triggers limit order after stop price reached.

Example in Backtrader:

```
self.buy(exectype=bt.Order.Market)
self.sell(exectype=bt.Order.Limit, price=target_price)
```

---

# 1.8 Types of Trading Strategies

## Trend Following

- **Goal**: Trade in the direction of the prevailing trend.
- **Example**: Buy when 50-day MA crosses above 200-day MA.

## Mean Reversion

- **Goal**: Bet that price will revert to its average.
- **Example**: Buy when RSI < 30, sell when RSI > 70.

## Breakout

- **Goal**: Enter when price breaks above resistance or below support.
- **Example**: Buy on breakout above 20-day high.

## Momentum

- **Goal**: Buy assets with strong performance, sell underperformers.
- **Example**: Rank assets by past 3-month returns.

---

# 1.9 Timeframes in Trading

| Timeframe | Style | Holding Period |
|---|---|---|
| 1m–15m | Scalping | Seconds–minutes |
| 15m–4h | Day Trading | Same day |
| 4h–Daily | Swing Trading | Days–weeks |
| Daily–Weekly | Position Trading | Weeks–months |
| Weekly–Monthly | Investing | Months–years |

Backtrader supports all timeframes, from tick data to monthly candles.

---

# 1.10 Algorithmic Trading: Advantages & Risks

**Advantages**

- Eliminates emotional bias
- Backtesting before risking capital
- Executes with speed & precision
- Can run 24/7 (especially in crypto)

**Risks**

- Overfitting to historical data
- Code bugs causing losses
- Market regime shifts
- Latency issues in live execution

---

# 1.11 How This Book Uses These Concepts

Later chapters will:

- Apply **OHLCV data** in Backtrader strategies
- Calculate **returns** for performance metrics
- Use **volatility** for position sizing
- Apply **VaR** and drawdown analysis for risk management
- Implement **order types** for precise execution

> **Pro Tip**: The *Mega Backtrader Pack* builds on these foundations with **250+ strategies**, volatility-adjusted position sizing, rolling backtests, and stress-testing frameworks.

🚀 **Trade Smarter — Without Wasting Years Coding From Scratch**

The **Mega Backtrader Strategy Pack** gives you **250+ fully-tested Python strategies** — adaptive filters, mean reversion, momentum, volatility breakouts, machine learning models & more — all ready to deploy in Backtrader.

**Why Traders Love This Pack**

- **Instant Impact** — Deploy strategies and start testing today
- **Zero Guesswork** — Every strategy is documented, tested, and ready-to-run
- **Future-Proof** — Lifetime updates included
- **Proven Variety** — Adaptive, mean-reversion, momentum, volatility, and ML-driven models

**What's Inside**
✅ 250+ Strategies across multiple categories:

- **Adaptive Filters** — Kalman, Hilbert, TRIX, Guppy MMA
- **Mean Reversion** — Bollinger, Keltner, pair trading, RSI/Stoch
- **Momentum & Trend** — MACD, ADX, Donchian, MA ribbons
- **Volatility** — ATR breakouts, Bollinger squeeze, VIX correlation
- **Specialized ML & Advanced** — HMM, decision trees, isolation forest, Hurst exponent

**You Get**
📂 Python `.py` scripts for direct Backtrader use
📄 PDF manuals detailing logic, parameters, and best practices
🎯 **Get Instant Access — Download Today, Trade Tonight**

👉 Get it here

# Chapter 2 — Python Quick Start for Trading

Algorithmic trading requires two things:

1. **Understanding markets** (covered in Chapter 1)
2. **Being able to code your strategy** so it can be tested and automated

Python is perfect for this — it's easy to learn, powerful, and has thousands of finance-related libraries.
This chapter is your crash course: we'll start **from zero**, then move to **real market data, indicators, and plotting**.

# 2.1 Python Basics: What You Need to Know Before You Start

## Why Python?

- **Simple** — readable syntax, minimal boilerplate
- **Powerful** — used by traders, hedge funds, and quants worldwide
- **Flexible** — works with market APIs, Backtrader, data analysis, and AI tools

---

# 2.2 Setting Up Your Coding Environment

You can write Python code **offline** or **online**.

## Option 1 — Spyder IDE (Offline)

Spyder comes with the Anaconda distribution, which is popular for data science and trading:

1. Download Anaconda: https://www.anaconda.com/download
2. Open **Spyder** from the Anaconda Navigator
3. You'll see an **editor** (left) and a **console** (bottom right) to run your code.

Your first code:

```
print("Hello, Algo Trading!")
```

Press **Run** and you'll see the output in the console.

---

## Option 2 — Google Colab (Online)

Google Colab is like an online Jupyter Notebook — no installation needed.

1. Go to https://colab.research.google.com
2. Sign in with your Google account
3. Click **New Notebook**
4. In the first cell, type:

```
print("Hello from Google Colab!")
```

Press **Shift + Enter** to run it.

> **Tip:** Colab is great for quick experiments, especially if you want to share your notebook or run code from your phone/tablet.

---

# 2.3 Writing Your First Trading-Oriented Python Code

## Variables

Variables store data so you can reuse it later in your code. In trading, you might store information like an asset's symbol, your available capital, or the current price.

```
symbol = "BTC-USD"   # Asset ticker (Bitcoin vs US Dollar)
capital = 10000      # Starting trading capital in USD
price = 29500.50     # Current price of Bitcoin in USD
```

Here:

- `symbol` holds the ticker name for Bitcoin in USD.
- `capital` is your starting budget for trading.
- `price` stores the current market price of the asset.

---

## Lists

Lists hold multiple values in one variable, often used for storing historical prices or trade records.

```
prices = [29500, 29600, 29750]
print(prices[0])  # First element
```

- Lists use square brackets `[]`.
- `prices[0]` gives the **first** price in the list (Python indexing starts at 0).
- In trading, lists are useful for storing sequences like price history or daily returns.

## Dictionaries

Dictionaries store **key-value pairs** — you give each value a name (the “key”) for easy access.

```
candle = {"open": 29500, "high": 29700, "low": 29400, "close": 29650}
```

- Each key ( `"open"` , `"high"` , `"low"` , `"close"` ) describes part of a candlestick.
- Values store the corresponding price for that key.
- This is useful when handling OHLC (Open-High-Low-Close) market data.

---

## Loops

Loops let you repeat actions for each item in a sequence, like going through a list of prices.

```
for p in prices:
    print("Price:", p)
```

- The loop runs once for each price in `prices` .
- `p` takes the value of each list element in order.
- In trading, you might loop through price history to apply calculations or strategies.

---

## Functions

Functions group code into reusable blocks. You can use them for calculations like moving averages.

```
def moving_average(data):
    return sum(data) / len(data)

print(moving_average([1, 2, 3, 4, 5]))
```

- `def` defines a function named `moving_average` .
- It takes `data` (a list of numbers) and returns the average.
- This is a simple version of the **Moving Average**, a common trading indicator.

# 2.4 Installing Trading Libraries

If you're in **Spyder/Anaconda**, open the terminal and run:

```
pip install backtrader pandas numpy matplotlib yfinance ta-lib
```

If you're in **Google Colab**, run in a code cell:

```
!pip install backtrader pandas numpy matplotlib yfinance ta-lib
```

---

# 2.5 Downloading Market Data with yfinance

```
import yfinance as yf

# Download BTC-USD data for 1 year, daily interval
df = yf.download("BTC-USD", period="1y", interval="1d").droplevel(1, 1)

print(df.head())
```

The result includes `Open`, `High`, `Low`, `Close`, `Adj Close`, and `Volume`.

```
                  open          high           low         close
volume
datetime
2024-08-13  59356.207031  61572.398438  58506.253906  60609.566406
30327698167
2024-08-14  60611.050781  61687.757812  58472.875000  58737.269531
29961696180
2024-08-15  58733.261719  59838.648438  56161.593750  57560.097656
35682112440
```

---

# 2.6 Calculating Technical Indicators with TA-Lib

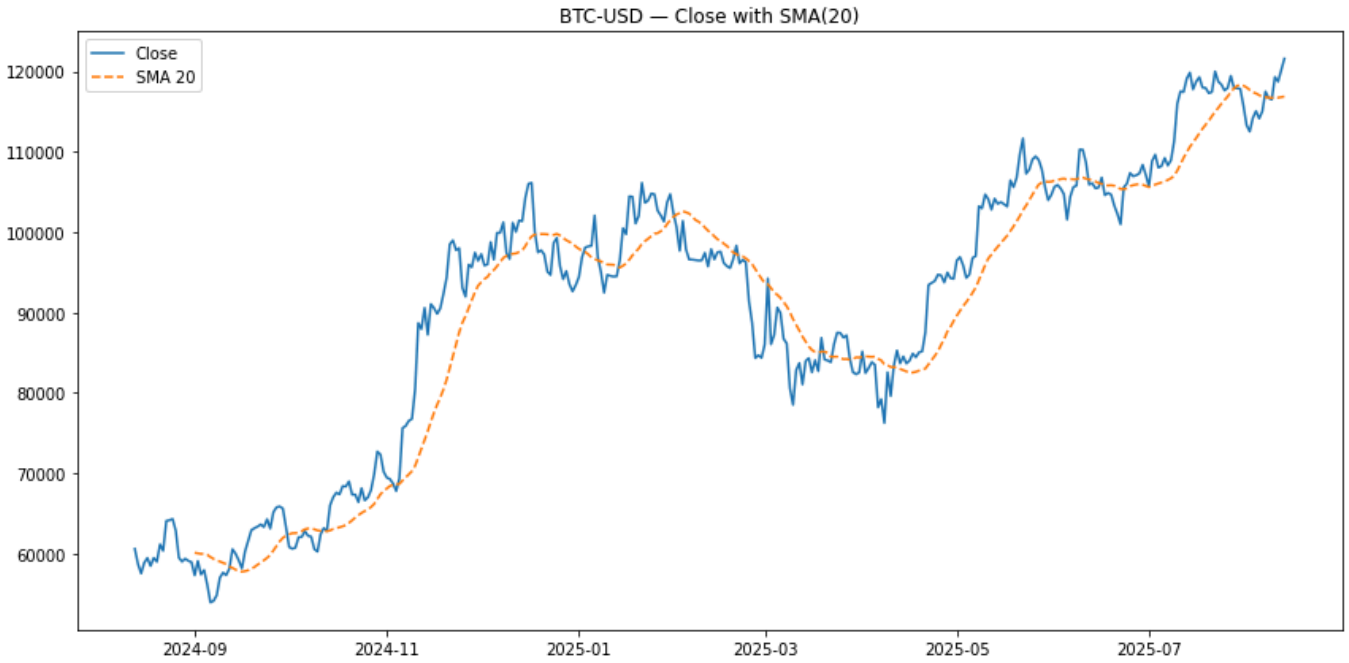
```
import talib

# RSI
df["RSI_14"] = talib.RSI(df["Close"], timeperiod=14)

# MACD
df["MACD"], df["MACD_signal"], df["MACD_hist"] = talib.MACD(
    df["Close"], fastperiod=12, slowperiod=26, signalperiod=9
)

# Bollinger Bands
df["BB_upper"], df["BB_middle"], df["BB_lower"] = talib.BBANDS(
    df["Close"], timeperiod=20, nbdevup=2, nbdevdn=2, matype=0
)

# ATR
df["ATR_14"] = talib.ATR(df["High"], df["Low"], df["Close"], timeperiod=14)
```

---

# 2.7 Plotting Price and Indicators

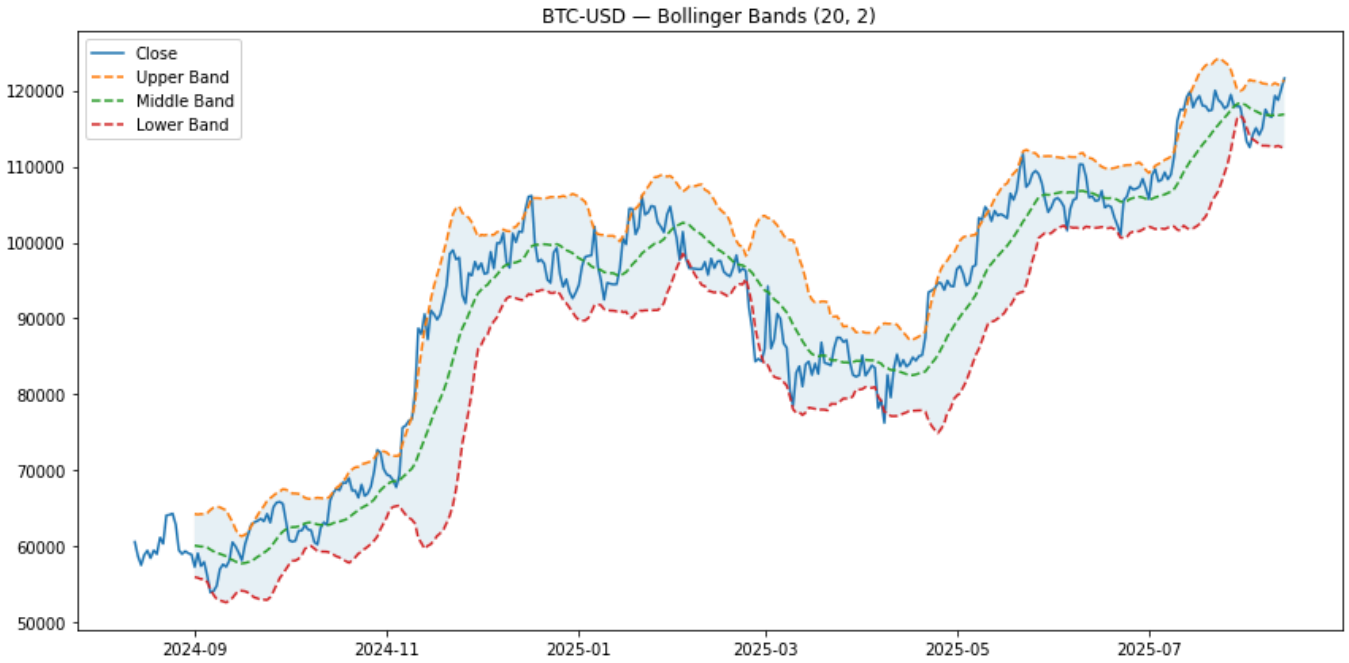
## Price + SMA

```
import matplotlib.pyplot as plt

df["SMA_20"] = df["Close"].rolling(window=20).mean()

plt.figure(figsize=(12,6))
plt.plot(df.index, df["Close"], label="Close Price", color="blue")
plt.plot(df.index, df["SMA_20"], label="SMA 20", color="red", linestyle="--")
plt.legend()
plt.title("BTC-USD with SMA 20")
plt.show()
```



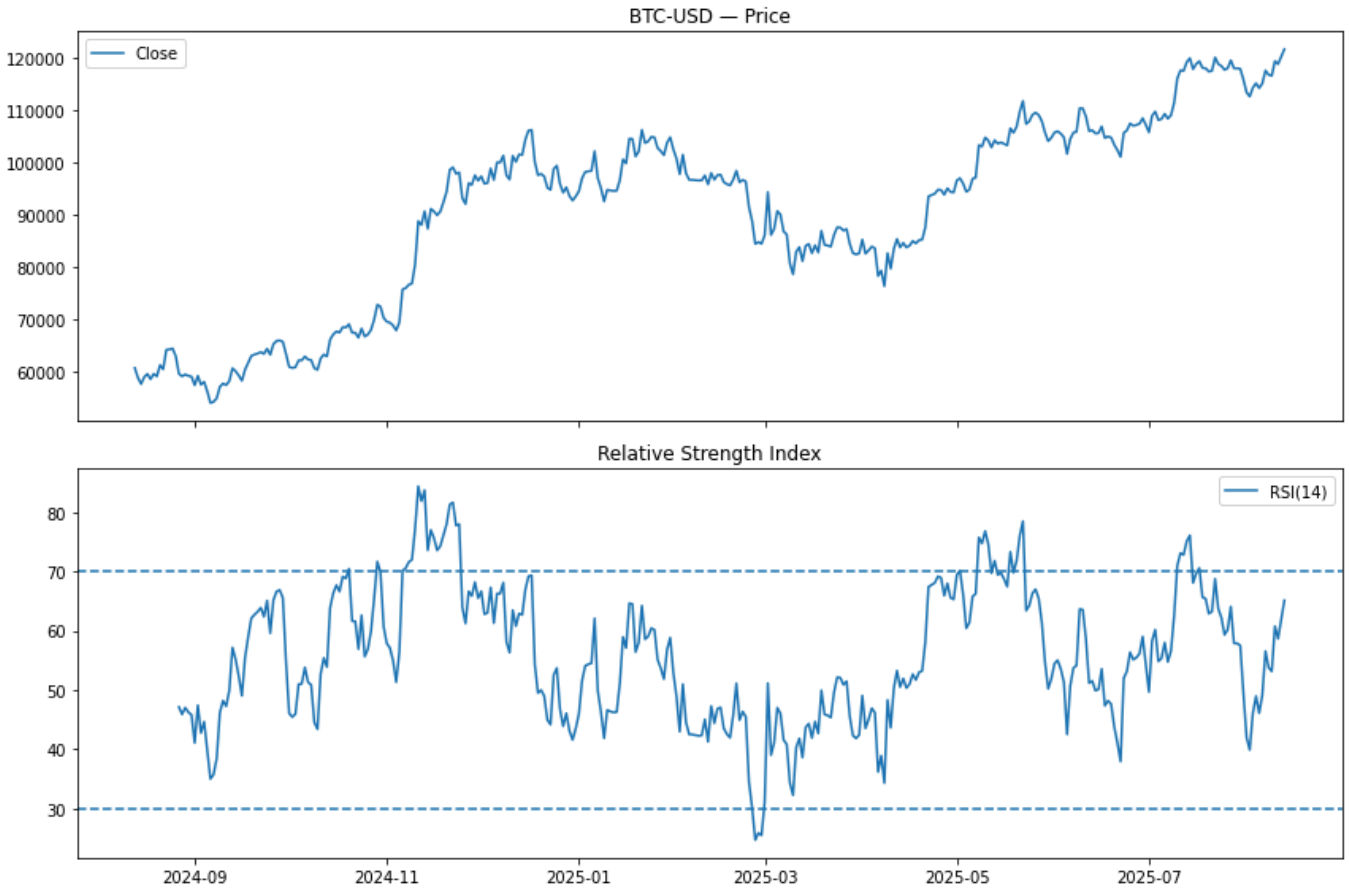

# Price + Bollinger Bands

```
plt.figure(figsize=(12,6))
plt.plot(df.index, df["Close"], label="Close Price", color="blue")
plt.plot(df.index, df["BB_upper"], label="Upper Band", color="red",
linestyle="--")
plt.plot(df.index, df["BB_middle"], label="Middle Band", color="black",
linestyle="--")
plt.plot(df.index, df["BB_lower"], label="Lower Band", color="green",
linestyle="--")
plt.fill_between(df.index, df["BB_lower"], df["BB_upper"], color="gray",
alpha=0.1)
plt.legend()
plt.title("Bollinger Bands on BTC-USD")
plt.show()
```




## Subplots: Price + RSI

```
fig, (ax1, ax2) = plt.subplots(2, 1, figsize=(12,8), sharex=True)

# Price
ax1.plot(df.index, df["Close"], label="Close Price")
ax1.set_title("BTC-USD Price")
ax1.legend()

# RSI
ax2.plot(df.index, df["RSI_14"], label="RSI 14", color="orange")
ax2.axhline(70, color="red", linestyle="--")
ax2.axhline(30, color="green", linestyle="--")
ax2.set_title("Relative Strength Index")
ax2.legend()

plt.tight_layout()
plt.show()
```




# 2.8 Preparing Data for Backtrader

Backtrader needs a specific feed format:

```
import backtrader as bt

# Rename columns to match Backtrader defaults
df_bt = df.rename(columns={
    "Open": "open", "High": "high", "Low": "low",
    "Close": "close", "Volume": "volume"
})
df_bt.index.name = "datetime"

data_feed = bt.feeds.PandasData(dataname=df_bt)
```

# 2.9 What You Can Do Now

By the end of this chapter, you can:

- Write and run Python code in Spyder or Google Colab
- Install and use finance-specific libraries
- Download real market data from Yahoo Finance
- Compute technical indicators with TA-Lib
- Visualize prices and indicators
- Prepare data for use in Backtrader

In **Chapter 3**, we'll bring it all together: building your first Backtrader strategy that uses these indicators for automated trading decisions.

---

> **Pro Tip:** The *Mega Backtrader Pack* includes **250+ pre-coded strategies** already integrated with TA-Lib indicators, so you can start modifying and backtesting from day one without writing code from scratch.

📄 PDF manuals detailing logic, parameters, and best practices
🎯 **Get Instant Access — Download Today, Trade Tonight**

👉 Get it here

# Chapter 3 — Getting Started with Backtrader

## 3.0 Quick mental model

- **Cerebro** is the engine: you add data, strategies, broker settings, analyzers, and press run.
- A **Strategy** is a class with lifecycle methods. You create indicators in `__init__` and make trading decisions in `next`.
- **Orders** are created by `buy`, `sell`, or `buy_bracket` / `sell_bracket` and filled by the simulated broker.
- **Analyzers** compute metrics like Sharpe, drawdown, and trade stats.

## 3.1 Barebones: the smallest possible strategy

Goal: prove your environment works and understand the minimal lifecycle.

### What each method is for

- `__init__(self)`: set up indicators and state variables. Runs once before the first bar.
- `next(self)`: called on every new bar (candle). Read data and place orders here.

### Code

```
import backtrader as bt
import yfinance as yf
import pandas as pd

class Barebones(bt.Strategy):
    def __init__(self):
        pass  # no indicators yet

    def next(self):
        pass  # no trading yet

def run_barebones(ticker="BTC-USD", period="6mo", interval="1d"):
    # 1) get data
    df = yf.download(ticker, period=period, interval=interval,
progress=False).droplevel(1, 1)
    if hasattr(df.index, "tz"):
```

```
        try: df.index = df.index.tz_localize(None)
        except: pass

    # 2) setup cerebro
    cerebro = bt.Cerebro()
    data = bt.feeds.PandasData(dataname=df)
    cerebro.adddata(data)
    cerebro.addstrategy(Barebones)
    cerebro.broker.setcash(10_000)

    print(f"Starting Portfolio Value: {cerebro.broker.getvalue():.2f}")
    cerebro.run()
    print(f"Final Portfolio Value:    {cerebro.broker.getvalue():.2f}")

run_barebones()
```

Checklist:

- You created a strategy class.
- You fed OHLCV data from `yfinance` into Backtrader.
- You executed a run without placing any orders.

# 3.2 First trade rule: minimal SMA crossover

Goal: create indicators in `__init__`, generate a signal in `next`.

## New pieces

- `bt.indicators.SMA(data.close, period=p)` creates a moving average.
- `bt.ind.CrossOver(ind1, ind2)` returns +1 on bullish cross, −1 on bearish.

## Code

```
class SMACrossMini(bt.Strategy):
    params = dict(fast=10, slow=30)

    def __init__(self):
        self.sma_fast = bt.indicators.SMA(self.data.close, period=self.p.fast)
        self.sma_slow = bt.indicators.SMA(self.data.close, period=self.p.slow)
        self.cross = bt.ind.CrossOver(self.sma_fast, self.sma_slow)

    def next(self):
        if not self.position:
            if self.cross > 0:
```

```
                self.buy()
        else:
            if self.cross < 0:
                self.sell()

def run_sma_mini(ticker="BTC-USD"):
    df = yf.download(ticker, period="1y", interval="1d",
progress=False).droplevel(1, 1)
    if hasattr(df.index, "tz"):
        try: df.index = df.index.tz_localize(None)
        except: pass
    cerebro = bt.Cerebro()
    cerebro.adddata(bt.feeds.PandasData(dataname=df))
    cerebro.addstrategy(SMACrossMini, fast=10, slow=30)
    cerebro.broker.setcash(10_000)
    cerebro.broker.setcommission(commission=0.001)  # 0.1%
    cerebro.addsizer(bt.sizers.PercentSizer, percents=95)
    cerebro.run()
    cerebro.plot()  # optional
```

Checklist:

- You created two indicators and a crossover signal.
- You bought and sold based on position state and signals.

# 3.3 Understanding the full lifecycle and logging

Goal: add logging and learn about order & trade notifications.

## New methods

- `log(self, txt)` : your helper for consistent prints.
- `notify_order(self, order)` : called when order status changes (Submitted → Completed, Canceled, Rejected).
- `notify_trade(self, trade)` : called when a trade closes (collect PnL).
- `start(self)` : optional hook at the beginning of the backtest.
- `stop(self)` : optional hook at the end for cleanup.

## Code

```
class SMACrossLogged(bt.Strategy):
    params = dict(fast=10, slow=30, printlog=True)

    def log(self, txt):
```

```
        if self.p.printlog:
            dt = self.datas[0].datetime.datetime(0)
            print(f"{dt} | {txt}")

    def __init__(self):
        self.order = None
        self.trades_log = []
        self.sma_fast = bt.indicators.SMA(self.data.close, period=self.p.fast)
        self.sma_slow = bt.indicators.SMA(self.data.close, period=self.p.slow)
        self.cross = bt.ind.CrossOver(self.sma_fast, self.sma_slow)

    def start(self):
        self.log("Backtest started")

    def notify_order(self, order):
        if order.status in [order.Submitted, order.Accepted]:
            return
        if order.status in [order.Completed]:
            side = "BUY" if order.isbuy() else "SELL"
            self.log(f"ORDER {side} EXECUTED @ {order.executed.price:.2f}, "
                     f"Value={order.executed.value:.2f}, Comm=
{order.executed.comm:.2f}")
        elif order.status in [order.Canceled, order.Margin, order.Rejected]:
            self.log("ORDER Canceled/Margin/Rejected")
        self.order = None

    def notify_trade(self, trade):
        if trade.isclosed:
            self.trades_log.append(dict(
                datetime=self.datas[0].datetime.datetime(0),
                size=trade.size,
                price=trade.price,
                pnl=trade.pnl,
                pnlcomm=trade.pnlcomm
            ))
            self.log(f"TRADE CLOSED PnL={trade.pnl:.2f} PnLcomm=
{trade.pnlcomm:.2f}")

    def next(self):
        if self.order:
            return
        if not self.position and self.cross > 0:
            self.order = self.buy()
        elif self.position and self.cross < 0:
            self.order = self.sell()
```

```
    def stop(self):
        self.log("Backtest finished")
        if self.trades_log:
            pd.DataFrame(self.trades_log).to_csv("trades_sma_logged.csv",
index=False)
            self.log("Saved trades_sma_logged.csv")
```

Checklist:

- You now see fills and PnL events.
- You saved a trade log to CSV.

# 3.4 Broker realism: commission, slippage, sizers

Goal: make results more realistic and control position size.

## Concepts

- `broker.setcommission(commission=0.001)` sets 0.1% per fill.
- `broker.set_slippage_perc(perc=0.0005)` simulates slippage (0.05%).
- `PercentSizer` allocates a percentage of equity into each trade.

## Code fragment

```
cerebro = bt.Cerebro()
cerebro.broker.setcash(10_000)
cerebro.broker.setcommission(commission=0.001)
# cerebro.broker.set_slippage_perc(perc=0.0005)  # optional
cerebro.addsizer(bt.sizers.PercentSizer, percents=95)
```

Checklist:

- You can model market frictions.
- You control capital allocation per trade.

# 3.5 Order types: Market, Limit, Stop, StopLimit, Bracket

Goal: place different order types and understand their use.

## Market and Limit

```
# in next()
if condition_market_buy:
```

```
    self.buy()  # market
if condition_limit_buy:
    self.buy(exectype=bt.Order.Limit, price=self.data.close[0] * 0.99)
```

## Stop and StopLimit

```
# stop (becomes market when stop price is hit)
self.sell(exectype=bt.Order.Stop, price=self.data.close[0] * 0.97)

# stop-limit (converts to a limit at stop)
self.sell(exectype=bt.Order.StopLimit, price=self.data.close[0] * 0.97,
plimit=self.data.close[0] * 0.965)
```

## Bracket orders (entry + stop-loss + take-profit in one go)

```
entry = self.data.close[0]
sl = entry * 0.95
tp = entry * 1.05
self.buy_bracket(price=entry, stopprice=sl, limitprice=tp)
```

Checklist:

- You understand execution control and risk automation via bracket orders.

# 3.6 Risk management with ATR sizing and dynamic stops

Goal: size positions by volatility and place adaptive stops.

## Concepts

- ATR measures typical bar range; higher ATR → more volatility → smaller position.
- Risk-per-trade model: risk 1% of equity, set stop at `k * ATR`, compute size = risk / (k*ATR).

## Code

```
class ATRRiskManaged(bt.Strategy):
    params = dict(atr_period=14, risk_pct=0.01, atr_mult=2.0)

    def __init__(self):
        self.atr = bt.indicators.ATR(self.data, period=self.p.atr_period)
        self.order = None

    def next(self):
```

```
        if self.order:
            return
        if not self.position:
            stop_distance = self.p.atr_mult * self.atr[0]
            cash = self.broker.getcash()
            equity = self.broker.getvalue()
            risk_amount = equity * self.p.risk_pct
            size = max(1, int(risk_amount / max(1e-6, stop_distance)))
            entry = self.data.close[0]
            stop = entry - stop_distance
            take = entry + 2 * stop_distance
            self.order = self.buy_bracket(size=size, price=entry,
stopprice=stop, limitprice=take)
```

Checklist:

- You converted volatility into a position size.
- Stops and targets adapt to market conditions.

# 3.7 Measuring performance with analyzers and plotting

Goal: compute Sharpe, drawdown, trade stats, and save charts.

## Analyzers you'll use

- `SharpeRatio` : risk-adjusted return
- `DrawDown` : max drawdown and length
- `TimeReturn` : bar-by-bar returns (build equity curve)
- `TradeAnalyzer` : win rate, counts, avg win/loss

## Code

```
def attach_analyzers(cerebro):
    cerebro.addanalyzer(bt.analyzers.SharpeRatio, riskfreerate=0.0,
                        timeframe=bt.TimeFrame.Days, annualize=True,
_name="sharpe")
    cerebro.addanalyzer(bt.analyzers.DrawDown, _name="dd")
    cerebro.addanalyzer(bt.analyzers.TimeReturn, timeframe=bt.TimeFrame.Days,
_name="timeret")
    cerebro.addanalyzer(bt.analyzers.TradeAnalyzer, _name="trades")

def summarize_analyzers(results):
    strat = results[0]
    sharpe = strat.analyzers.sharpe.get_analysis().get("sharperatio")
```

```
    dd = strat.analyzers.dd.get_analysis()
    tr = strat.analyzers.trades.get_analysis()
    timeret = strat.analyzers.timeret.get_analysis()

    print(f"Sharpe: {sharpe}")
    print(f"MaxDD: {dd.get('maxdrawdown'):.2f}% | MaxDD Len:
{dd.get('maxlen')}")
    ttot = getattr(getattr(tr, "total", None), "total", None)
    won = getattr(getattr(tr, "won", None), "total", None)
    lost = getattr(getattr(tr, "lost", None), "total", None)
    if ttot:
        wr = (won / ttot * 100) if won is not None else None
        print(f"Trades: {ttot} | Won: {won} | Lost: {lost} | Win rate:
{wr:.2f}%")

    return timeret

def save_equity(timereturn_dict, start_cash=10_000, name="sma"):
    if not timereturn_dict:
        return
    dates = sorted(timereturn_dict.keys())
    rets = pd.Series([timereturn_dict[d] for d in dates], index=dates)
    equity = start_cash * (1.0 + rets).cumprod()
    ax = equity.plot(figsize=(12,4), title=f"Equity Curve - {name}")
    fig = ax.get_figure()
    fig.tight_layout()
    fig.savefig(f"equity_curve_{name}.png", dpi=150)
    plt.show()
```

Checklist:

- You attached analyzers and printed key metrics.
- You exported a PNG equity curve.

# 3.8 Consolidated, ready-to-run script (toggle steps)

This script starts simple and grows with options. Change `STRAT_STAGE` to try each build stage.

```
"""
Chapter 3 consolidated runner
Stages:
  "bare"      -> Barebones (no trades)
  "sma_mini"  -> Minimal SMA crossover
  "sma_log"   -> SMA with lifecycle logging + CSV trades
  "atr_risk"  -> ATR risk-managed bracket orders
```

```
"""

import backtrader as bt
import yfinance as yf
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

TICKER = "BTC-USD"
PERIOD = "1y"
INTERVAL = "1d"
STARTING_CASH = 10_000.0
COMMISSION = 0.001
SLIPPAGE = 0.0
PERC_SIZE = 95
STRAT_STAGE = "sma_log"  # "bare", "sma_mini", "sma_log", "atr_risk"

# ---------- strategies ----------
class Barebones(bt.Strategy):
    def __init__(self): pass
    def next(self): pass

class SMACrossMini(bt.Strategy):
    params = dict(fast=10, slow=30)
    def __init__(self):
        self.sma_fast = bt.indicators.SMA(self.data.close, period=self.p.fast)
        self.sma_slow = bt.indicators.SMA(self.data.close, period=self.p.slow)
        self.cross = bt.ind.CrossOver(self.sma_fast, self.sma_slow)
    def next(self):
        if not self.position and self.cross > 0:
            self.buy()
        elif self.position and self.cross < 0:
            self.sell()

class SMACrossLogged(bt.Strategy):
    params = dict(fast=10, slow=30, printlog=True)
    def log(self, txt):
        if self.p.printlog:
            dt = self.datas[0].datetime.datetime(0)
            print(f"{dt} | {txt}")
    def __init__(self):
        self.order = None
        self.trades_log = []
        self.sma_fast = bt.indicators.SMA(self.data.close, period=self.p.fast)
        self.sma_slow = bt.indicators.SMA(self.data.close, period=self.p.slow)
        self.cross = bt.ind.CrossOver(self.sma_fast, self.sma_slow)
```

```
    def start(self): self.log("Backtest started")
    def notify_order(self, order):
        if order.status in [order.Submitted, order.Accepted]: return
        if order.status in [order.Completed]:
            side = "BUY" if order.isbuy() else "SELL"
            self.log(f"ORDER {side} EXECUTED @ {order.executed.price:.2f}, "
                     f"Value={order.executed.value:.2f}, Comm=
{order.executed.comm:.2f}")
        elif order.status in [order.Canceled, order.Margin, order.Rejected]:
            self.log("ORDER Canceled/Margin/Rejected")
        self.order = None
    def notify_trade(self, trade):
        if trade.isclosed:
            self.trades_log.append(dict(
                datetime=self.datas[0].datetime.datetime(0),
                size=trade.size, price=trade.price,
                pnl=trade.pnl, pnlcomm=trade.pnlcomm
            ))
            self.log(f"TRADE CLOSED PnL={trade.pnl:.2f} PnLcomm=
{trade.pnlcomm:.2f}")
    def next(self):
        if self.order: return
        if not self.position and self.cross > 0:
            self.order = self.buy()
        elif self.position and self.cross < 0:
            self.order = self.sell()
    def stop(self):
        self.log("Backtest finished")
        if self.trades_log:
            pd.DataFrame(self.trades_log).to_csv("trades_sma_logged.csv",
index=False)
            self.log("Saved trades_sma_logged.csv")

class ATRRiskManaged(bt.Strategy):
    params = dict(atr_period=14, risk_pct=0.01, atr_mult=2.0)
    def __init__(self):
        self.order = None
        self.atr = bt.indicators.ATR(self.data, period=self.p.atr_period)
    def next(self):
        if self.order: return
        if not self.position:
            stop_dist = self.p.atr_mult * float(self.atr[0])
            equity = float(self.broker.getvalue())
            risk_amt = equity * self.p.risk_pct
            size = max(1, int(risk_amt / max(stop_dist, 1e-6)))
            entry = float(self.data.close[0])
```

```
            stop = entry - stop_dist
            take = entry + 2*stop_dist
            self.order = self.buy_bracket(size=size, price=entry,
stopprice=stop, limitprice=take)

# ---------- analyzers ----------
def attach_analyzers(c):
    c.addanalyzer(bt.analyzers.SharpeRatio, riskfreerate=0.0,
                  timeframe=bt.TimeFrame.Days, annualize=True, _name="sharpe")
    c.addanalyzer(bt.analyzers.DrawDown, _name="dd")
    c.addanalyzer(bt.analyzers.TimeReturn, timeframe=bt.TimeFrame.Days,
_name="timeret")
    c.addanalyzer(bt.analyzers.TradeAnalyzer, _name="trades")

def summarize(results):
    s = results[0]
    sharpe = s.analyzers.sharpe.get_analysis().get("sharperatio")
    dd = s.analyzers.dd.get_analysis()
    tr = s.analyzers.trades.get_analysis()
    timeret = s.analyzers.timeret.get_analysis()
    print(f"Sharpe: {sharpe}")
    print(f"MaxDD: {dd.get('maxdrawdown'):.2f}% | MaxDD Len:
{dd.get('maxlen')}")
    total = getattr(getattr(tr, "total", None), "total", None)
    won = getattr(getattr(tr, "won", None), "total", None)
    lost = getattr(getattr(tr, "lost", None), "total", None)
    if total:
        wr = (won/total*100) if won is not None else None
        print(f"Trades: {total} | Won: {won} | Lost: {lost} | Win rate:
{wr:.2f}%")
    return timeret

def save_equity(timereturn_dict, start_cash=STARTING_CASH, name="stage"):
    if not timereturn_dict: return
    dates = sorted(timereturn_dict.keys())
    rets = pd.Series([timereturn_dict[d] for d in dates], index=dates)
    eq = start_cash * (1.0 + rets).cumprod()
    ax = eq.plot(figsize=(12,4), title=f"Equity Curve - {name}")
    fig = ax.get_figure()
    fig.tight_layout()
    fig.savefig(f"equity_curve_{name}.png", dpi=150)
    plt.show()

# ---------- data + run ----------
def fetch_df(ticker=TICKER, period=PERIOD, interval=INTERVAL):
    df = yf.download(ticker, period=period, interval=interval,
```

```
progress=False).droplevel(1, 1)
    if df.empty:
        raise RuntimeError("No data downloaded")
    if hasattr(df.index, "tz"):
        try: df.index = df.index.tz_localize(None)
        except: pass
    return df

def run_stage():
    df = fetch_df()
    cerebro = bt.Cerebro()
    cerebro.broker.setcash(STARTING_CASH)
    cerebro.broker.setcommission(commission=COMMISSION)
    if SLIPPAGE:
        cerebro.broker.set_slippage_perc(perc=SLIPPAGE)
    cerebro.adddata(bt.feeds.PandasData(dataname=df))
    cerebro.addsizer(bt.sizers.PercentSizer, percents=PERC_SIZE)

    if STRAT_STAGE == "bare":
        cerebro.addstrategy(Barebones)
        label = "bare"
    elif STRAT_STAGE == "sma_mini":
        cerebro.addstrategy(SMACrossMini, fast=10, slow=30)
        label = "sma_mini"
    elif STRAT_STAGE == "sma_log":
        cerebro.addstrategy(SMACrossLogged, fast=10, slow=30, printlog=True)
        label = "sma_log"
    else:
        cerebro.addstrategy(ATRRiskManaged, atr_period=14, risk_pct=0.01,
atr_mult=2.0)
        label = "atr_risk"

    attach_analyzers(cerebro)
    print(f"Starting Portfolio Value: {cerebro.broker.getvalue():.2f}")
    results = cerebro.run()
    print(f"Final Portfolio Value:    {cerebro.broker.getvalue():.2f}")

    timeret = summarize(results)

    try:
        figs = cerebro.plot(iplot=False, style="candlestick", volume=False)
        if figs and isinstance(figs, (list, tuple)):
            fig = figs[0][0] if isinstance(figs[0], (list, tuple)) else
figs[0]
            fig.savefig(f"backtrader_plot_{label}.png", dpi=150,
bbox_inches="tight")
```

```
            print(f"Saved backtrader_plot_{label}.png")
    except Exception as e:
        print(f"[WARN] plot not saved: {e}")

    save_equity(timeret, name=label)

if __name__ == "__main__":
    run_stage()
```

# Chapter 4 — Regime-Filtered Trend - Trade Only When Conditions Are Right

This chapter turns a simple trend-following idea (moving averages) into a **context-aware** system that selectively trades only when the market is likely trending. You'll learn the theory behind each "regime signal" (ADX, Bollinger Band width, volatility), how to **combine** them into a regime classifier with confidence, how to **smooth** regime changes, and how to map regime + confidence into **position sizing** and **trailing risk**. Then we'll implement everything in Backtrader, step by step, ending with a complete runnable strategy.

This strategy is part of the **Mega Backtrader Strategy Pack** — a collection of over **250 ready-to-use strategies** for your algorithmic trading.

The **Mega Backtrader Strategy Pack** gives you **250+ fully-tested Python strategies** — adaptive filters, mean reversion, momentum, volatility breakouts, machine learning models & more — all ready to deploy in Backtrader.

**Why Traders Love This Pack**

- Instant Impact — Deploy strategies and start testing today
- Zero Guesswork — Every strategy is documented, tested, and ready-to-run
- Future-Proof — Lifetime updates included
- Proven Variety — Adaptive, mean-reversion, momentum, volatility, and ML-driven models

What's Inside

✅ **250+ Strategies across multiple categories:**

- Adaptive Filters — Kalman, Hilbert, TRIX, Guppy MMA
- Mean Reversion — Bollinger, Keltner, pair trading, RSI/Stoch
- Momentum & Trend — MACD, ADX, Donchian, MA ribbons
- Volatility — ATR breakouts, Bollinger squeeze, VIX correlation
- Specialized ML & Advanced — HMM, decision trees, isolation forest, Hurst exponent

You Get

📂 Python `.py` scripts for direct Backtrader use

📄 PDF manuals detailing logic, parameters, and best practices

🎯 Get Instant Access — Download Today, Trade Tonight

👉 **Get it here**

# 4.1 Why Regime Filtering?

Most simple trend strategies lose money in **chop** (sideways, noisy markets). A regime filter attempts to detect when the market is:

- **Trending**: directional persistence is present, breakouts follow through.
- **Ranging**: mean-reverting; breakouts fail; trend entries whipsaw.
- **Unknown**: not enough evidence; stay cautious.

Trade the trend logic **only** when the regime is trending. Otherwise, stand aside or use different logic (we'll stand aside in this chapter).

# 4.2 Ingredients and Definitions

We'll use three orthogonal signals. Each contributes one "vote" toward a trending regime.

## 4.2.1 Moving Averages (SMA)

We'll use **SMA fast** and **SMA slow** for the trend logic, not for regime classification.
SMA over window $n$ on closes $C_t$:

$$\mathrm{SMA}_n(t) = \frac{1}{n}\sum_{i=0}^{n-1} C_{t-i}$$

A **bullish cross** occurs when $\mathrm{SMA}_{\mathrm{fast}}$ crosses above $\mathrm{SMA}_{\mathrm{slow}}$; bearish is the opposite.

```
import backtrader as bt

class DemoSMAs(bt.Strategy):
    params = dict(ma_fast=7, ma_slow=30)

    def __init__(self):
        self.ma_fast = bt.indicators.SMA(self.data.close,
period=self.p.ma_fast)
        self.ma_slow = bt.indicators.SMA(self.data.close,
period=self.p.ma_slow)
        self.cross = bt.ind.CrossOver(self.ma_fast, self.ma_slow)
```

```
    def next(self):
        if not self.position and self.cross > 0:
            self.buy()
        elif self.position and self.cross < 0:
            self.close()
```

## 4.2.2 Average Directional Index (ADX)

ADX attempts to quantify trend **strength** (not direction). Internally it uses +DI and −DI built from **True Range** (TR):

$$TR_t = \max\{H_t - L_t,\ |H_t - C_{t-1}|,\ |L_t - C_{t-1}|\}$$

DI's measure directional movement; ADX smooths the **Directional Movement Index (DX)**:

$$DX_t = 100 \cdot \frac{|+DI_t - -DI_t|}{+DI_t + -DI_t}$$

$$ADX = \text{smoothed}(DX)$$

Heuristic: **ADX > 20–25** often signals trending conditions.

```
class DemoADX(bt.Strategy):
    params = dict(adx_period=14, threshold=20)
    def __init__(self):
        self.adx = bt.indicators.ADX(period=self.p.adx_period)
    def next(self):
        trending = self.adx[0] > self.p.threshold
        # use `trending` as a gate for entries
```

## 4.2.3 Bollinger Band Width

Bollinger Bands around a moving average (typically 20) with standard deviation $\sigma$:

$$\text{Upper} = MA + k\sigma, \quad \text{Lower} = MA - k\sigma$$

We use **band width** normalized by price as a proxy for expansion:

$$\text{BB\_width} = \frac{\text{Upper} - \text{Lower}}{MA}$$

Wider bands often accompany trends; extremely narrow bands suggest squeeze/range.

```
class DemoBBWidth(bt.Strategy):
    params = dict(bb_period=20, bb_width_threshold=0.01)
    def __init__(self):
        self.bb = bt.indicators.BollingerBands(self.data.close,
period=self.p.bb_period)
    def next(self):
        mid = float(self.bb.mid[0])
        bb_width = 0.0 if mid == 0 else (float(self.bb.top[0]) -
float(self.bb.bot[0])) / mid
        expansion = bb_width > self.p.bb_width_threshold
```

### 4.2.4 ATR-Normalized Volatility

Average True Range (ATR) is the moving average of TR. We normalize to price:

$$\text{NormVol}_t = \frac{\text{ATR}_t}{C_t}$$

If NormVol is above a threshold and/or rising relative to its recent average, the market is more **kinetic**, which is friendlier to trend-following.

```
class DemoNormATR(bt.Strategy):
    params = dict(atr_period=14, vol_threshold=0.01)
    def __init__(self):
        self.atr = bt.indicators.ATR(self.data, period=self.p.atr_period)
    def next(self):
        price = float(self.data.close[0]) or 1.0
        norm_vol = float(self.atr[0]) / price
        kinetic = norm_vol > self.p.vol_threshold
```

## 4.3 Classifying Regime

We compute three binary “trending” signals each bar:

1. **ADX trending** if $ADX_t > \theta_{ADX}$.
2. **BB trending** if $BB\_width_t > \theta_{BB}$.
3. **Vol trending** if $\text{NormVol}_t > \theta_{Vol}$.

Let $S$ be the number of signals triggered, $T$ the total (3 here). Define a **confidence** score:

$$\text{confidence}_t = \frac{S}{T}$$

- If $S \geq 2 \rightarrow$ regime candidate = **trending**
- If $S = 0 \rightarrow$ regime candidate = **ranging**
- Else $\rightarrow$ **unknown**

```
class DemoRegimeVotes(bt.Strategy):
    params = dict(adx_period=14, adx_thr=20, bb_period=20, bb_thr=0.01,
atr_period=14, vol_thr=0.01)
    def __init__(self):
        self.adx = bt.indicators.ADX(period=self.p.adx_period)
        self.bb  = bt.indicators.BollingerBands(self.data.close,
period=self.p.bb_period)
        self.atr = bt.indicators.ATR(self.data, period=self.p.atr_period)

    def classify(self):
        votes = 0
        if float(self.adx[0]) > self.p.adx_thr: votes += 1
        mid = float(self.bb.mid[0]); width = 0 if mid == 0 else
(float(self.bb.top[0]) - float(self.bb.bot[0]))/mid
        if width > self.p.bb_thr: votes += 1
        price = float(self.data.close[0]) or 1.0
        norm_vol = float(self.atr[0]) / price
        if norm_vol > self.p.vol_thr: votes += 1

        conf = votes/3.0
        if votes >= 2: return "trending", conf
        if votes == 0: return "ranging", conf
        return "unknown", conf
```

## Smoothing regime changes (confirmation)

Flip-flopping regimes is expensive. We require the candidate regime to be **consistent** for the last $K$ bars (e.g., 3 bars) to confirm a regime **change**. This provides hysteresis.

```
from collections import deque

class DemoRegimeSmooth(bt.Strategy):
    params = dict(confirm_k=3, adx_period=14, adx_thr=20, bb_period=20,
bb_thr=0.01, atr_period=14, vol_thr=0.01)
    def __init__(self):
        self.adx = bt.indicators.ADX(period=self.p.adx_period)
        self.bb  = bt.indicators.BollingerBands(self.data.close,
period=self.p.bb_period)
        self.atr = bt.indicators.ATR(self.data, period=self.p.atr_period)
```

```
        self.reg_hist = deque(maxlen=self.p.confirm_k)
        self.regime = "unknown"
        self.confidence = 0.0

    def classify(self):
        # same as above
        votes = 0
        if float(self.adx[0]) > self.p.adx_thr: votes += 1
        mid = float(self.bb.mid[0]); width = 0 if mid == 0 else
(float(self.bb.top[0]) - float(self.bb.bot[0]))/mid
        if width > self.p.bb_thr: votes += 1
        price = float(self.data.close[0]) or 1.0
        norm_vol = float(self.atr[0]) / price
        if norm_vol > self.p.vol_thr: votes += 1
        conf = votes/3.0
        if votes >= 2: cand = "trending"
        elif votes == 0: cand = "ranging"
        else: cand = "unknown"
        return cand, conf

    def next(self):
        cand, conf = self.classify()
        self.reg_hist.append(cand)
        if len(self.reg_hist) == self.p.confirm_k and all(r == cand for r in
self.reg_hist):
            self.regime = cand
        self.confidence = conf
```

# 4.4 Position Sizing by Regime Confidence

Map regime + confidence to a size between $[\text{min\_pct}, \text{max\_pct}]$:

- **Trending**: interpolate linearly by confidence
- **Ranging**: return 0 (we stand aside)
- **Unknown**: only allow size if confidence exceeds a high bar (e.g., 0.7)

```
class DemoSizing(bt.Strategy):
    params = dict(min_pct=0.20, max_pct=0.80)
    def size_from_conf(self, regime, confidence):
        if regime == "trending":
            c = max(0.0, min(1.0, confidence))
            return self.p.min_pct + c * (self.p.max_pct - self.p.min_pct)
        if regime == "ranging":
            return 0.0
```

```
        return self.p.min_pct if confidence > 0.7 else 0.0
```

# 4.5 Trailing Risk by Regime

Set trailing stop as multiple of ATR:

- **Trending**: looser trail $= \text{trail\_atr\_mult} \times ATR$ and optionally modulate by recent volatility (e.g., increase trail in high vol, tighten in low vol).
- **Ranging**: tighter trail $= \text{range\_atr\_mult} \times ATR$.

```
import numpy as np

class DemoTrail(bt.Strategy):
    params = dict(atr_period=14, trail_mult=3.0, range_mult=1.0)
    def __init__(self):
        self.atr = bt.indicators.ATR(self.data, period=self.p.atr_period)
        self.vol_hist = []

    def normalized_vol(self):
        price = float(self.data.close[0]) or 1.0
        return float(self.atr[0]) / price

    def trail_multiplier(self, regime):
        if regime == "trending":
            mult = self.p.trail_mult
            self.vol_hist.append(self.normalized_vol())
            hist = self.vol_hist[-10:]
            if len(hist) == 10:
                cur, avg = hist[-1], float(np.mean(hist))
                if cur > 1.2 * avg: mult *= 1.3
                elif cur < 0.8 * avg: mult *= 0.8
            return mult
        return self.p.range_mult
```

# 4.6 Entry/Exit Logic

- Engage the trend logic only if regime says **trade trend**.
- **Entry long**: $\text{SMA}_{\text{fast}}$ crosses above $\text{SMA}_{\text{slow}}$.
- **Entry short**: $\text{SMA}_{\text{fast}}$ crosses below $\text{SMA}_{\text{slow}}$ (optional: if your broker settings allow shorting).

- Optionally **add** a small incremental position on strong trend continuation (e.g., MA spread > 3%).
- **Exit**: ATR trailing stop or cross in the opposite direction.

```
class DemoEntries(bt.Strategy):
    params = dict(ma_fast=7, ma_slow=30, atr_period=14, trail_mult=3.0)
    def __init__(self):
        self.ma_fast = bt.indicators.SMA(self.data.close,
period=self.p.ma_fast)
        self.ma_slow = bt.indicators.SMA(self.data.close,
period=self.p.ma_slow)
        self.cross = bt.ind.CrossOver(self.ma_fast, self.ma_slow)
        self.atr = bt.indicators.ATR(self.data, period=self.p.atr_period)
        self.trail_order = None
        self.regime = "trending"  # assume allowed for this demo

    def cancel_trail(self):
        if self.trail_order:
            try: self.cancel(self.trail_order)
            except: pass
            self.trail_order = None

    def next(self):
        price = float(self.data.close[0])
        bullish = self.cross > 0
        bearish = self.cross < 0

        # entry
        if not self.position and self.regime == "trending":
            if bullish:
                size = max(1, int(self.broker.getcash() * 0.5 / price))
                self.buy(size=size)

        # trailing management
        if self.position.size > 0:
            stop_price = price - self.p.trail_mult * float(self.atr[0])
            self.cancel_trail()
            self.trail_order = self.sell(exectype=bt.Order.Stop,
price=stop_price)
            if bearish:
                self.close()
```

# 4.7 Backtrader Implementation: Step-by-Step

We'll mirror the parameters used in your strategy files so your readers can swap them seamlessly.

### 4.7.1 Parameters

- `ma_fast=7`, `ma_slow=30`
- `adx_period=14`, `adx_trending_threshold=20`
- `bb_period=7`, `bb_width_threshold=0.01`
- `volatility_lookback=7`, `vol_trending_threshold=0.01`
- `atr_period=14`, `trail_atr_mult=3.0`, `range_atr_mult=1.0`
- `max_position_pct=0.80`, `min_position_pct=0.20`
- `regime_confirmation=3`

### 4.7.2 Build indicators

- SMA fast/slow for signals
- ADX, BB, ATR for regime
- Maintain a short **volatility history** of NormVol for adaptive trail

### 4.7.3 Classify regime each bar

- Compute the three binary signals
- Turn votes into confidence $[0, 1]$
- Push candidate regime into a short **history deque**
- If last `K` entries agree, **confirm** change

### 4.7.4 Compute position size and trailing multiplier

- Size = function(regime, confidence) clamped to $[min, max]$
- Trail multiplier depends on regime and recent NormVol

### 4.7.5 Entries, exits, and trailing

- Only trade if `should_trade_trend_following()` is true
- On cross, size position by `position_size_pct * cash / price`
- Maintain/update a trailing stop order based on ATR

## 4.8 Complete Backtrader Code

This is a **self-contained** strategy class you can drop into your project. It follows the structure described above and matches your parameter names so the chapter and your codebase stay in

sync.

```
import backtrader as bt
import numpy as np

class RegimeFilteredTrendStrategy(bt.Strategy):
    params = (
        ('ma_fast', 7),            # Fast moving average
        ('ma_slow', 30),            # Slow moving average
        ('adx_period', 14),         # ADX period
        ('adx_trending_threshold', 20), # ADX threshold for trending regime
        ('bb_period', 7),          # Bollinger Bands period
        ('bb_width_threshold', 0.01), # BB width threshold for trending (3%)
        ('volatility_lookback', 7), # Volatility measurement period
        ('vol_trending_threshold', 0.01), # Volatility threshold for trending
        ('atr_period', 14),         # ATR period
        ('trail_atr_mult', 3.0),    # Trailing stop multiplier (trending)
        ('range_atr_mult', 1.),    # Trailing stop multiplier (ranging)
        ('max_position_pct', 0.80), # Maximum position size
        ('min_position_pct', 0.20), # Minimum position size
        ('regime_confirmation', 3), # Bars to confirm regime change
    )

    def __init__(self):
        self.dataclose = self.datas[0].close

        # Moving averages for trend following
        self.ma_fast = bt.indicators.SMA(period=self.params.ma_fast)
        self.ma_slow = bt.indicators.SMA(period=self.params.ma_slow)

        # Regime classification indicators
        self.adx = bt.indicators.ADX(period=self.params.adx_period)
        self.bb = bt.indicators.BollingerBands(period=self.params.bb_period)
        self.atr = bt.indicators.ATR(period=self.params.atr_period)

        # Track orders and regime state
        self.order = None
        self.trail_order = None

        # Regime tracking
        self.current_regime = "unknown"  # "trending", "ranging", "unknown"
        self.regime_history = []
        self.regime_confidence = 0

        # Volatility tracking
        self.volatility_history = []
```

```
    def cancel_trail(self):
        if self.trail_order:
            self.cancel(self.trail_order)
            self.trail_order = None

    def calculate_volatility(self):
        """Calculate normalized volatility measure"""
        if len(self.atr) == 0 or self.dataclose[0] <= 0:
            return 0

        try:
            return self.atr[0] / self.dataclose[0]
        except:
            return 0

    def classify_market_regime(self):
        """Classify current market regime using multiple indicators"""
        if (len(self.adx) == 0 or len(self.bb) == 0 or
            len(self.volatility_history) < 5):
            return "unknown", 0

        try:
            trending_signals = 0
            total_signals = 0

            # ADX Signal
            total_signals += 1
            if self.adx[0] > self.params.adx_trending_threshold:
                trending_signals += 1

            # Bollinger Band Width Signal
            total_signals += 1
            bb_width = (self.bb.top[0] - self.bb.bot[0]) / self.bb.mid[0]
            if bb_width > self.params.bb_width_threshold:
                trending_signals += 1

            # Volatility Signal
            total_signals += 1
            current_vol = self.volatility_history[-1]
            if current_vol > self.params.vol_trending_threshold:
                trending_signals += 1

            # Moving Average Separation Signal
            total_signals += 1
            ma_separation = abs(self.ma_fast[0] - self.ma_slow[0]) /
```

```
self.ma_slow[0]
            if ma_separation > 0.02:  # 2% separation
                trending_signals += 1

            # Calculate confidence
            confidence = trending_signals / total_signals

            # Classify regime
            if confidence >= 0.75:  # 3+ out of 4 signals
                regime = "trending"
            elif confidence <= 0.25:  # 1 or fewer signals
                regime = "ranging"
            else:
                regime = "uncertain"

            return regime, confidence

        except Exception as e:
            return "unknown", 0

    def update_regime_state(self):
        """Update regime state with confirmation logic"""
        new_regime, confidence = self.classify_market_regime()

        # Add to regime history
        self.regime_history.append(new_regime)
        if len(self.regime_history) > self.params.regime_confirmation * 2:
            self.regime_history = self.regime_history[-
self.params.regime_confirmation * 2:]

        # Confirm regime change only if consistent over multiple bars
        if len(self.regime_history) >= self.params.regime_confirmation:
            recent_regimes = self.regime_history[-
self.params.regime_confirmation:]

            # Check for consistency
            if all(r == new_regime for r in recent_regimes):
                if self.current_regime != new_regime:
                    # Regime change confirmed
                    self.current_regime = new_regime
                    self.regime_confidence = confidence
            else:
                # Mixed signals, keep current regime but update confidence
                self.regime_confidence = confidence

    def calculate_regime_position_size(self):
```

```
        """Calculate position size based on regime and confidence"""
        try:
            base_size = self.params.max_position_pct

            if self.current_regime == "trending":
                # Full size in trending markets
                size_factor = 1.0
            elif self.current_regime == "ranging":
                # Reduced size in ranging markets
                size_factor = 0.3
            else:  # uncertain or unknown
                # Minimal size in uncertain markets
                size_factor = 0.1

            # Adjust by confidence
            confidence_factor = max(0.5, self.regime_confidence)

            final_size = base_size * size_factor * confidence_factor

            return max(self.params.min_position_pct,
                       min(self.params.max_position_pct, final_size))

        except Exception as e:
            return self.params.min_position_pct

    def get_adaptive_stop_multiplier(self):
        """Get ATR multiplier based on regime and volatility"""
        if self.current_regime == "trending":
            # Wider stops in trending markets
            base_mult = self.params.trail_atr_mult

            # Adjust for volatility
            if len(self.volatility_history) >= 5:
                current_vol = self.volatility_history[-1]
                avg_vol = np.mean(self.volatility_history[-10:])

                if current_vol > avg_vol * 1.2:  # High volatility
                    return base_mult * 1.3
                elif current_vol < avg_vol * 0.8:  # Low volatility
                    return base_mult * 0.8

            return base_mult
        else:
            # Tighter stops in ranging markets
            return self.params.range_atr_mult
```

```
    def should_trade_trend_following(self):
        """Determine if trend following should be active"""
        if self.current_regime == "trending":
            return True
        elif self.current_regime == "ranging":
            return False  # No trend following in ranging markets
        else:
            # Uncertain regime - very cautious
            return self.regime_confidence > 0.7

    def notify_order(self, order):
        if order.status in [order.Submitted, order.Accepted]:
            return
        # log optional
        # print(self.data.datetime.date(0), order.getstatusname())
        self.order = None  # <- CRITICAL: free the lock when
done/rejected/canceled

    def notify_trade(self, trade):
        if trade.isclosed:
            # optional: print realized PnL
            # print(f"{self.data.datetime.date(0)} | PnL:
{trade.pnlcomm:.2f}")
            pass




    def next(self):
        # Skip if order is pending
        if self.order:
            return

        # Update volatility tracking
        current_vol = self.calculate_volatility()
        if current_vol > 0:
            self.volatility_history.append(current_vol)
            if len(self.volatility_history) > self.params.volatility_lookback:
                self.volatility_history = self.volatility_history[-
self.params.volatility_lookback:]

        # Update regime classification
        self.update_regime_state()

        # Handle existing positions with adaptive stops
        if self.position:
```

```
            stop_multiplier = self.get_adaptive_stop_multiplier()

            # (re)arm/refresh trailing stop if missing
            if not self.trail_order:
                if self.position.size > 0:
                    self.trail_order = self.sell(
                        exectype=bt.Order.StopTrail,
                        trailamount=self.atr[0] * stop_multiplier,
                        size=self.position.size              # <- add size
                    )
                else:  # short
                    self.trail_order = self.buy(
                        exectype=bt.Order.StopTrail,
                        trailamount=self.atr[0] * stop_multiplier,
                        size=abs(self.position.size)         # <- add size
                    )

            # ALSO exit on signal flip or regime change (don't rely only on
stop)
            if self.position.size > 0:
                if (self.ma_fast[0] < self.ma_slow[0]) or (self.current_regime
!= "trending"):
                    self.cancel_trail()
                    self.order = self.close()                # <- create
explicit exit
            else:  # short
                if (self.ma_fast[0] > self.ma_slow[0]) or (self.current_regime
!= "trending"):
                    self.cancel_trail()
                    self.order = self.close()

            return

        # Ensure sufficient data
        required_bars = max(self.params.ma_slow, self.params.adx_period,
self.params.bb_period)
        if len(self) < required_bars:
            return

        # Check if we should engage in trend following
        if not self.should_trade_trend_following():
            return  # Stay out during non-trending regimes

        # Moving average crossover signals (only in trending regimes)
        ma_bullish_cross = (self.ma_fast[0] > self.ma_slow[0] and
self.ma_fast[-1] <= self.ma_slow[-1])
```

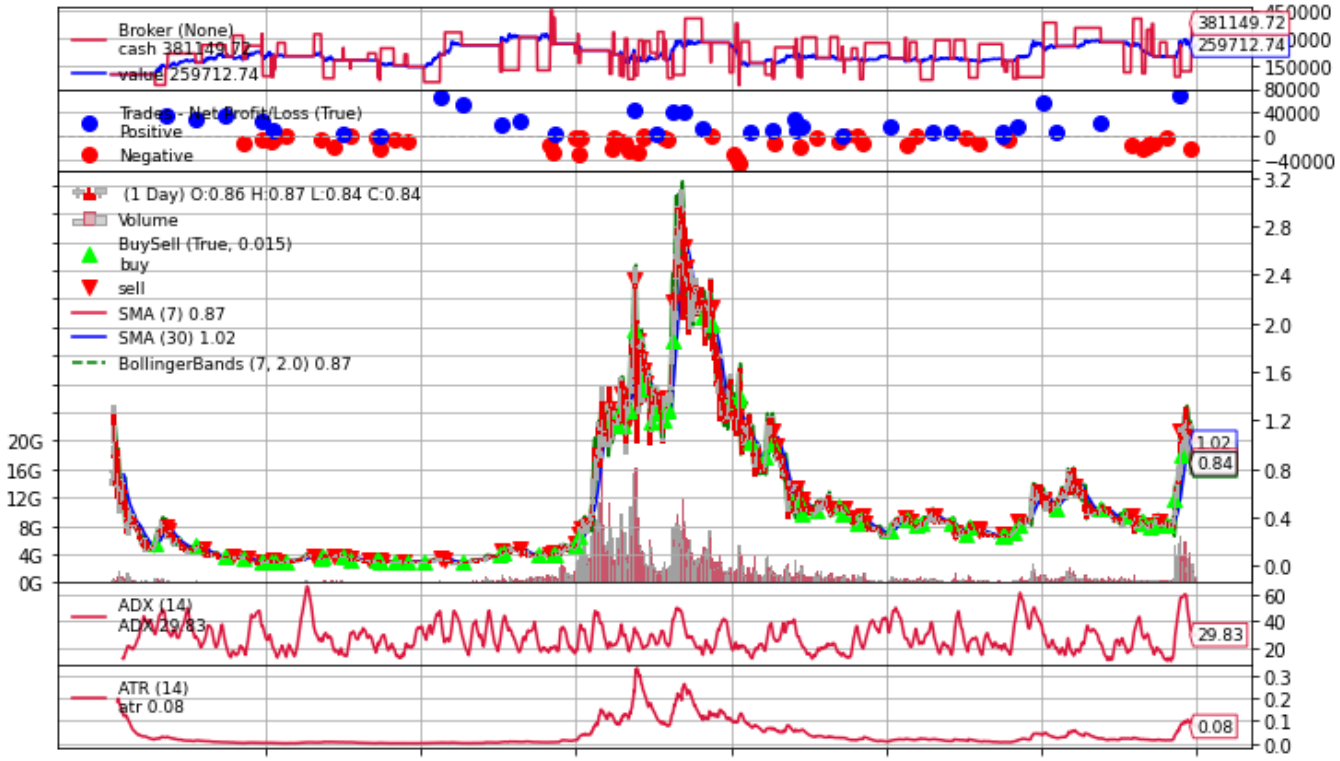
```
        ma_bearish_cross = (self.ma_fast[0] < self.ma_slow[0] and
self.ma_fast[-1] >= self.ma_slow[-1])

        # Position sizing based on regime
        position_size_pct = self.calculate_regime_position_size()
        current_price = float(self.dataclose[0])

        # LONG ENTRY
        if ma_bullish_cross:
            self.cancel_trail()
            cash = float(self.broker.getcash())
            target_value = cash * position_size_pct
            shares = target_value / max(current_price, 1e-12)
            self.order = self.buy(size=shares)               # keep float
sizing for crypto

        # SHORT ENTRY
        elif ma_bearish_cross:
            self.cancel_trail()
            cash = float(self.broker.getcash())
            target_value = cash * position_size_pct
            shares = target_value / max(current_price, 1e-12)
            self.order = self.sell(size=shares)

        # Alternative entry: strong continuation
        elif (self.current_regime == "trending" and self.regime_confidence >
0.8):
            ma_spread = (self.ma_fast[0] - self.ma_slow[0]) / self.ma_slow[0]
            if ma_spread > 0.03:
                cash = float(self.broker.getcash())
                target_value = cash * (position_size_pct * 0.7)
                shares = target_value / max(current_price, 1e-12)
                self.order = self.buy(size=shares)
            elif ma_spread < -0.03:
                cash = float(self.broker.getcash())
                target_value = cash * (position_size_pct * 0.7)
                shares = target_value / max(current_price, 1e-12)
                self.order = self.sell(size=shares)
```

## 4.9 Backtesting

Run this as a separate script (or notebook cell) to fetch data with `yfinance`, set basic broker settings, and execute the regime-filtered trend strategy. You'll get a **plot** and can export metrics with Backtrader analyzers (see Chapter 3 analyzers section).

```
import backtrader as bt
import yfinance as yf
import pandas as pd
import numpy as np
import dateutil.relativedelta as rd
import matplotlib.pyplot as plt
%matplotlib inline
import seaborn as sns
import importlib

def load_strategy(class_name):
    module = importlib.import_module(class_name)
    strategy = getattr(module, class_name)
    return strategy

ticker = "ADA-USD"
start = "2018-01-01"
end = "2025-01-01"

start_dt = pd.to_datetime(start)
end_dt = pd.to_datetime(end)


strategy = load_strategy("RegimeFilteredTrendStrategy")

data = yf.download(ticker, start=start, end=end, progress=False)


data = data.droplevel(1, 1) if isinstance(data.columns, pd.MultiIndex) else
data

feed = bt.feeds.PandasData(dataname=data)
cerebro = bt.Cerebro()
cerebro.addstrategy(strategy)
cerebro.adddata(feed)
cerebro.broker.setcash(100000)
cerebro.broker.setcommission(commission=0.001)
# cerebro.addsizer(bt.sizers.PercentSizer, percents=95)

cerebro.addanalyzer(bt.analyzers.SharpeRatio, _name='sharpe',
timeframe=bt.TimeFrame.Days, riskfreerate=0.0)
cerebro.addanalyzer(bt.analyzers.DrawDown,    _name='dd')
cerebro.addanalyzer(bt.analyzers.Returns,     _name='rets')
cerebro.addanalyzer(bt.analyzers.TradeAnalyzer, _name='trades')

start_val = cerebro.broker.getvalue()
```

```
results = cerebro.run()
strat = results[0]
final_val = cerebro.broker.getvalue()
ret = (final_val - start_val) / start_val * 100


print(f"Return: {ret:.2f}% | Final Value: {final_val:.2f}")



sh = strat.analyzers.sharpe.get_analysis().get('sharperatio', None)
dd = strat.analyzers.dd.get_analysis()
rets = strat.analyzers.rets.get_analysis()
tr = strat.analyzers.trades.get_analysis()

print(f"Sharpe: {sh}")
print(f"Max DD: {dd.get('max', {}).get('drawdown', 0):.2f}%  Len:
{dd.get('max', {}).get('len', 0)}")
print(f"CAGR:   {rets.get('rnorm100', 0):.2f}%  Total: {rets.get('rtot',
0)*100:.2f}%")
print(f"Trades: {tr.get('total', {}).get('total', 0)}  Won: {tr.get('won',
{}).get('total', 0)}  Lost: {tr.get('lost', {}).get('total', 0)}")

plt.rcParams['figure.figsize'] = [10, 6]
plt.rcParams['font.size'] = 10
cerebro.plot(iplot=False)
```



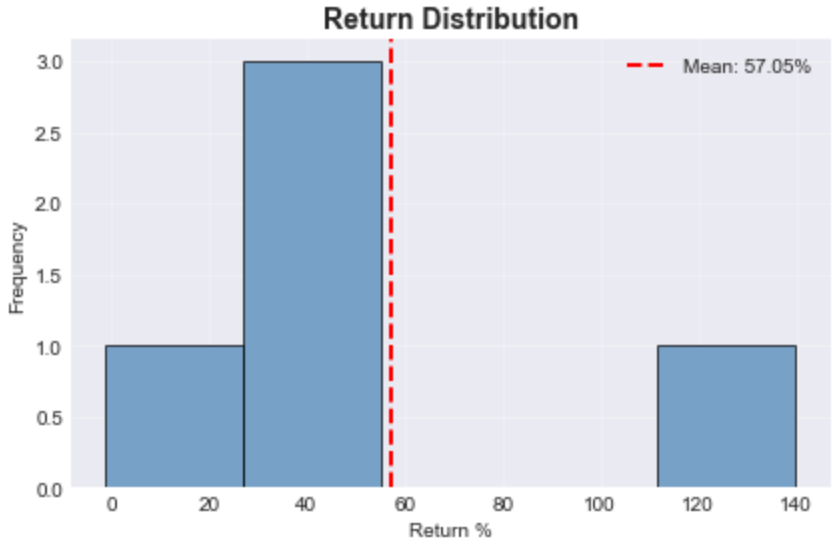

- **Thresholds**: Start with `ADX>20`, `BB_width>1%`, `NormVol>1%`, then tune by asset and timeframe.
- **Confirmation**: `regime_confirmation=3` is a good default; raise to reduce false flips, lower to react faster.
- **Sizing**: If results are too volatile, reduce `max_position_pct` and/or use **PercentSizer** at the Cerebro level.
- **Shorts**: If your venue disallows shorting, set `allow_short=False` and delete the short branch.
- **Continuation adds**: Small, infrequent adds based on MA spread can help ride strong trends, but cap their size.

# 4.10 Rolling Backtest

The `run_rolling_backtest` function tests the strategy over 12-month windows for BTC-USD, fetching data via `yfinance`, with $100,000 initial capital, 0.1% commission, and a 95% sizer (overridden by the strategy).

- **Data**: Uses `yfinance` to fetch BTC-USD daily data, processed into a Backtrader-compatible `PandasData` feed.
- **Broker Settings**: Starts with $100,000 and a 0.1% commission, with position sizing handled dynamically by the strategy.
- **Regime Stability**: The 3-bar confirmation period reduces false regime switches, enhancing reliability.

- **Risk Management**: Combines regime-based position sizing, adaptive stops, and trend filters for robust risk control.


The Regime Filtered Trend Strategy excels at adapting to market conditions through multi-indicator regime classification, ensuring trades are taken in favorable trending environments. Its use of SMAs for trend detection, combined with dynamic position sizing and adaptive stops, makes it well-suited for volatile markets like cryptocurrencies. The rolling backtest framework provides a comprehensive evaluation across diverse market conditions, enabling traders to assess consistency and optimize parameters (e.g., MA periods, regime thresholds) for specific assets or timeframes.

# Chapter 5 — Relative Momentum Acceleration - Thrust Oscillator Breakouts

This strategy looks for bursts of acceleration away from an adaptive baseline. We build a custom **thrust oscillator** from a fast EMA relative to **KAMA** (Kaufman's Adaptive Moving Average), then trigger entries on **statistical breakouts** of that oscillator using **Bollinger Bands**. Risk is managed with an **ATR-based trailing stop** that updates off the highest/lowest price since entry.



# 5.1 Intuition

1. Use KAMA as the slow, adaptive baseline (it "speeds up" in trends and "slows down" in chop).
2. Use a fast EMA as the quick momentum probe.
3. Define a normalized thrust oscillator:

$$\text{thrust}_t \;=\; \frac{\text{EMA}_{\text{fast}}(t) - \text{KAMA}(t)}{\text{KAMA}(t) + \varepsilon}$$

4. Put **Bollinger Bands** on that oscillator; breakouts above/below its band mean “unusual acceleration” relative to recent behavior.
5. Trail the position with an **ATR** multiple from the running high/low since entry.

Notes

- $\varepsilon = 10^{-6}$ avoids divide-by-zero.
- Using bands on the oscillator (instead of price) adapts to the instrument’s recent “relative acceleration” regime.

# 5.2 Components and formulas

KAMA (conceptual)

- Efficiency ratio $ER = \frac{|P_t - P_{t-n}|}{\sum_{i=0}^{n-1} |P_{t-i} - P_{t-i-1}|}$
- Smoothing constant $SC = [ER \cdot (FC - SC_{slow}) + SC_{slow}]^2$
- Update $KAMA_t = KAMA_{t-1} + SC \cdot (P_t - KAMA_{t-1})$
  Backtrader provides `bt.ind.KAMA(period=...)`.

EMA (fast)

$$EMA_n(t) = \alpha P_t + (1-\alpha) EMA_n(t-1), \quad \alpha = \frac{2}{n+1}$$

Bollinger Bands (on oscillator)

$$\mu_t = \text{MA}_m(\text{thrust}), \quad \sigma_t = \text{stdev}_m(\text{thrust})$$
$$\text{Upper} = \mu_t + k\sigma_t, \quad \text{Lower} = \mu_t - k\sigma_t$$

ATR trailing stop

- Long: $\text{stop}_t = \max(\text{stop}_{t-1}, \text{HH}_t - m \cdot ATR_t)$ with $\text{HH}_t = \max(\text{HH}_{t-1}, High_t)$
- Short: $\text{stop}_t = \min(\text{stop}_{t-1}, \text{LL}_t + m \cdot ATR_t)$ with $\text{LL}_t = \min(\text{LL}_{t-1}, Low_t)$

# 5.3 Step-by-step code build

## 5.3.1 Baseline and momentum probes

```
import backtrader as bt

class _RMA_Part1(bt.Strategy):
```

```
    params = dict(kama_period=30, fast_ema_period=7)

    def __init__(self):
        self.kama = bt.ind.KAMA(self.data.close, period=self.p.kama_period)
        self.fast_ema = bt.ind.EMA(self.data.close,
period=self.p.fast_ema_period)
```

## 5.3.2 Thrust oscillator (normalized EMA–KAMA)

```
class _RMA_Part2(bt.Strategy):
    params = dict(kama_period=30, fast_ema_period=7)
    def __init__(self):
        self.kama = bt.ind.KAMA(self.data.close, period=self.p.kama_period)
        self.fast_ema = bt.ind.EMA(self.data.close,
period=self.p.fast_ema_period)
        self.thrust_osc = (self.fast_ema - self.kama) / (self.kama + 1e-6)
```

## 5.3.3 Bollinger Bands on the oscillator

```
class _RMA_Part3(bt.Strategy):
    params = dict(kama_period=30, fast_ema_period=7, thrust_bb_period=7,
thrust_bb_devfactor=1.0)
    def __init__(self):
        self.kama = bt.ind.KAMA(self.data.close, period=self.p.kama_period)
        self.fast_ema = bt.ind.EMA(self.data.close,
period=self.p.fast_ema_period)
        self.thrust_osc = (self.fast_ema - self.kama) / (self.kama + 1e-6)

        self.thrust_bbands = bt.ind.BollingerBands(
            self.thrust_osc, period=self.p.thrust_bb_period,
devfactor=self.p.thrust_bb_devfactor
        )
```

## 5.3.4 ATR and trailing state

```
class _RMA_Part4(bt.Strategy):
    params = dict(atr_period=7)
    def __init__(self):
        self.atr = bt.ind.ATR(self.data, period=self.p.atr_period)
        self.stop_price = None
        self.highest_price_since_entry = None
        self.lowest_price_since_entry = None
```

## 5.3.5 Orders: entries on oscillator breakouts

```
class _RMA_Part5(bt.Strategy):
    params = dict(kama_period=30, fast_ema_period=7, thrust_bb_period=7,
thrust_bb_devfactor=1.0)
    def __init__(self):
        self.kama = bt.ind.KAMA(self.data.close, period=self.p.kama_period)
        self.fast_ema = bt.ind.EMA(self.data.close,
period=self.p.fast_ema_period)
        self.thrust_osc = (self.fast_ema - self.kama) / (self.kama + 1e-6)
        self.thrust_bbands = bt.ind.BollingerBands(self.thrust_osc,

period=self.p.thrust_bb_period,

devfactor=self.p.thrust_bb_devfactor)
        self.order = None

    def next(self):
        if self.order: return
        if not self.position:
            if self.thrust_osc[0] > self.thrust_bbands.top[0]:
                self.order = self.buy()
            elif self.thrust_osc[0] < self.thrust_bbands.bot[0]:
                self.order = self.sell()
```

## 5.3.6 Manual ATR trailing stop logic

```
class _RMA_Part6(bt.Strategy):
    params = dict(atr_period=7, atr_stop_multiplier=3.0)
    def __init__(self):
        self.atr = bt.ind.ATR(self.data, period=self.p.atr_period)
        self.stop_price = None
        self.highest_price_since_entry = None
        self.lowest_price_since_entry = None
        self.order = None

    def notify_order(self, order):
        if order.status in [order.Submitted, order.Accepted]:
            return
        if order.status in [order.Completed]:
            if self.position and self.stop_price is None:
                if order.isbuy():
                    self.highest_price_since_entry = self.data.high[0]
                    self.stop_price = self.highest_price_since_entry -
```

```
(self.atr[0] * self.p.atr_stop_multiplier)
                elif order.issell():
                    self.lowest_price_since_entry = self.data.low[0]
                    self.stop_price = self.lowest_price_since_entry +
(self.atr[0] * self.p.atr_stop_multiplier)
            elif not self.position:
                self.stop_price = None
                self.highest_price_since_entry = None
                self.lowest_price_since_entry = None
        self.order = None

    def next(self):
        if self.order: return
        if self.position:
            if self.position.size > 0:
                self.highest_price_since_entry =
max(self.highest_price_since_entry, self.data.high[0])
                new_stop = self.highest_price_since_entry - (self.atr[0] *
self.p.atr_stop_multiplier)
                self.stop_price = max(self.stop_price, new_stop)
                if self.data.close[0] < self.stop_price:
                    self.order = self.close()
            elif self.position.size < 0:
                self.lowest_price_since_entry =
min(self.lowest_price_since_entry, self.data.low[0])
                new_stop = self.lowest_price_since_entry + (self.atr[0] *
self.p.atr_stop_multiplier)
                self.stop_price = min(self.stop_price, new_stop)
                if self.data.close[0] > self.stop_price:
                    self.order = self.close()
```

# 5.4 Complete Backtrader strategy

Exact class as you specified.

```
import backtrader as bt

class RelativeMomentumAccel(bt.Strategy):
    """
    A strategy that enters on a statistical breakout of a custom oscillator
    measuring the acceleration of price away from its adaptive baseline trend.
    """
    params = (
        # Baseline Trend and Momentum
        ('kama_period', 30),
```

```
        ('fast_ema_period', 7),
        # Thrust Oscillator Breakout
        ('thrust_bb_period', 7),
        ('thrust_bb_devfactor', 1.),
        # Risk Management
        ('atr_period', 7),
        ('atr_stop_multiplier', 3.0),
    )

    def __init__(self):
        self.order = None

        # --- Indicators ---
        self.kama = bt.indicators.KAMA(self.data.close,
period=self.p.kama_period)
        self.fast_ema =
bt.indicators.ExponentialMovingAverage(self.data.close,
period=self.p.fast_ema_period)

        # --- Custom Thrust Oscillator ---
        # Note: A small number is added to self.kama to prevent division by
zero in rare cases
        self.thrust_osc = (self.fast_ema - self.kama) / (self.kama + 1e-6)

        # --- Breakout Bands on the Oscillator ---
        self.thrust_bbands = bt.indicators.BollingerBands(
            self.thrust_osc,
            period=self.p.thrust_bb_period,
            devfactor=self.p.thrust_bb_devfactor
        )

        self.atr = bt.indicators.AverageTrueRange(self.data,
period=self.p.atr_period)

        # --- Trailing Stop State ---
        self.stop_price = None
        self.highest_price_since_entry = None
        self.lowest_price_since_entry = None

    def notify_order(self, order):
        if order.status in [order.Submitted, order.Accepted]: return
        if order.status in [order.Completed]:
            if self.position and self.stop_price is None:
                if order.isbuy():
                    self.highest_price_since_entry = self.data.high[0]
                    self.stop_price = self.highest_price_since_entry -
```

```
(self.atr[0] * self.p.atr_stop_multiplier)
                elif order.issell():
                    self.lowest_price_since_entry = self.data.low[0]
                    self.stop_price = self.lowest_price_since_entry +
(self.atr[0] * self.p.atr_stop_multiplier)
            elif not self.position:
                self.stop_price = None; self.highest_price_since_entry = None;
self.lowest_price_since_entry = None
        self.order = None

    def next(self):
        if self.order: return

        if not self.position:
            # --- Entry Logic ---
            # Buy when the thrust oscillator breaks above its upper band
            if self.thrust_osc[0] > self.thrust_bbands.top[0]:
                self.order = self.buy()
            # Sell when the thrust oscillator breaks below its lower band
            elif self.thrust_osc[0] < self.thrust_bbands.bot[0]:
                self.order = self.sell()

        elif self.position:
            # --- Manual ATR Trailing Stop Logic ---
            if self.position.size > 0: # Long
                self.highest_price_since_entry =
max(self.highest_price_since_entry, self.data.high[0])
                new_stop = self.highest_price_since_entry - (self.atr[0] *
self.p.atr_stop_multiplier)
                self.stop_price = max(self.stop_price, new_stop)
                if self.data.close[0] < self.stop_price: self.order =
self.close()
            elif self.position.size < 0: # Short
                self.lowest_price_since_entry =
min(self.lowest_price_since_entry, self.data.low[0])
                new_stop = self.lowest_price_since_entry + (self.atr[0] *
self.p.atr_stop_multiplier)
                self.stop_price = min(self.stop_price, new_stop)
                if self.data.close[0] > self.stop_price: self.order =
self.close()
```

# 5.5 Backtesting

Use this to execute and generate your own plots afterward. No extra sections.

```
import backtrader as bt
import yfinance as yf
import pandas as pd
import numpy as np
import dateutil.relativedelta as rd
import matplotlib.pyplot as plt
%matplotlib inline
import seaborn as sns
import importlib

def load_strategy(class_name):
    module = importlib.import_module(class_name)
    strategy = getattr(module, class_name)
    return strategy

ticker = "BTC-USD"
start = "2018-01-01"
end = "2025-01-01"

start_dt = pd.to_datetime(start)
end_dt = pd.to_datetime(end)


strategy = load_strategy("RelativeMomentumAccel")

data = yf.download(ticker, start=start, end=end, progress=False)


data = data.droplevel(1, 1) if isinstance(data.columns, pd.MultiIndex) else
data

feed = bt.feeds.PandasData(dataname=data)
cerebro = bt.Cerebro()
cerebro.addstrategy(strategy)
cerebro.adddata(feed)
cerebro.broker.setcash(100000)
cerebro.broker.setcommission(commission=0.001)
# cerebro.addsizer(bt.sizers.PercentSizer, percents=95)

cerebro.addanalyzer(bt.analyzers.SharpeRatio, _name='sharpe',
timeframe=bt.TimeFrame.Days, riskfreerate=0.0)
cerebro.addanalyzer(bt.analyzers.DrawDown,    _name='dd')
cerebro.addanalyzer(bt.analyzers.Returns,     _name='rets')
cerebro.addanalyzer(bt.analyzers.TradeAnalyzer, _name='trades')

start_val = cerebro.broker.getvalue()
```

```
results = cerebro.run()
strat = results[0]
final_val = cerebro.broker.getvalue()
ret = (final_val - start_val) / start_val * 100


print(f"Return: {ret:.2f}% | Final Value: {final_val:.2f}")



sh = strat.analyzers.sharpe.get_analysis().get('sharperatio', None)
dd = strat.analyzers.dd.get_analysis()
rets = strat.analyzers.rets.get_analysis()
tr = strat.analyzers.trades.get_analysis()

print(f"Sharpe: {sh}")
print(f"Max DD: {dd.get('max', {}).get('drawdown', 0):.2f}%  Len:
{dd.get('max', {}).get('len', 0)}")
print(f"CAGR:   {rets.get('rnorm100', 0):.2f}%  Total: {rets.get('rtot',
0)*100:.2f}%")
print(f"Trades: {tr.get('total', {}).get('total', 0)}  Won: {tr.get('won',
{}).get('total', 0)}  Lost: {tr.get('lost', {}).get('total', 0)}")

plt.rcParams['figure.figsize'] = [10, 6]
plt.rcParams['font.size'] = 10
cerebro.plot(iplot=False)
```

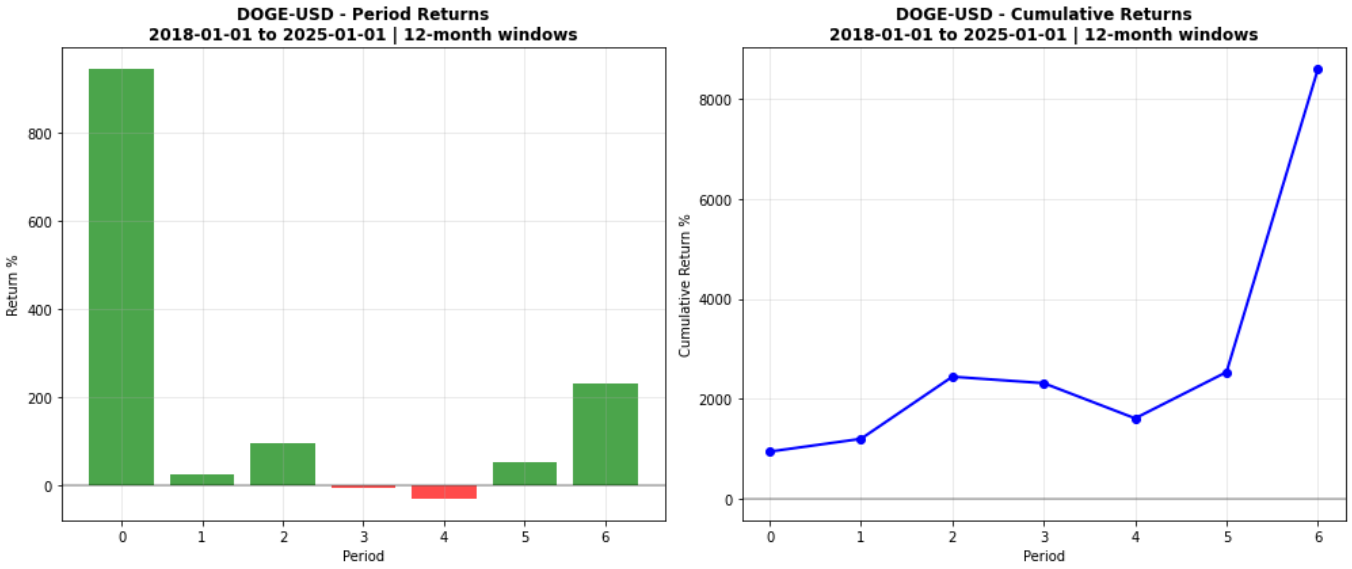

# 5.6 Rolling Backtest

Here's a rolling backtest and results:

```
strategy = load_strategy("RelativeMomentumAccel")

    ticker = "DOGE-USD"
    start = "2018-01-01"
    end = "2025-01-01"
    window_months = 12

    df = run_rolling_backtest(ticker=ticker, start=start, end=end,
window_months=window_months)
```

```
=== ROLLING BACKTEST RESULTS ===
        start         end  return_pct   final_value
0  2018-01-01  2019-01-01  942.746130  1.042746e+06
1  2019-01-01  2020-01-01   24.553182  1.245532e+05
2  2020-01-01  2021-01-01   95.744318  1.957443e+05
3  2021-01-01  2022-01-01   -5.070911  9.492909e+04
4  2022-01-01  2023-01-01  -29.193316  7.080668e+04
5  2023-01-01  2024-01-01   54.207870  1.542079e+05
6  2024-01-01  2025-01-01  230.066964  3.300670e+05
```

```
=== ROLLING BACKTEST STATISTICS ===
Mean Return %: 187.58
Median Return %: 54.21
Std Dev %: 318.25
Min Return %: -29.19
Max Return %: 942.75
Sharpe Ratio: 0.59
```




### Key Observations

- **Explosive upside capture** in strong trends (2018 rally, 2024 rebound).
- **Highly skewed distribution** — a handful of massive years drive the mean.
- **ATR trailing stop** locks in large wins but still suffers in range-bound conditions.
- **Volatility profile** means significant drawdowns are possible — risk sizing is critical.

### Potential Refinements

- Introduce a **trend regime filter** (e.g., ADX > 20 or price > long SMA) before allowing breakouts.
- Increase `thrust_bb_period` or `devfactor` to reduce false breakouts.
- Position sizing based on **current ATR** to normalize risk exposure.

# Chapter 6 — Ichimoku Cloud Strategy - Kumo Breakout + TK Cross + Chikou

## 6.1 Intuition

Trade only when price breaks out of the **Kumo (cloud)** with **Tenkan/Kijun** momentum confirmation and **Chikou** agreement. Manage risk with a simple trailing stop expressed as a fraction of price.

This strategy is part of the **Mega Backtrader Strategy Pack** — a collection of over **250 ready-to-use strategies** for your algorithmic trading.

The **Mega Backtrader Strategy Pack** gives you **250+ fully-tested Python strategies** — adaptive filters, mean reversion, momentum, volatility breakouts, machine learning models & more — all ready to deploy in Backtrader.

**Why Traders Love This Pack**

- Instant Impact — Deploy strategies and start testing today
- Zero Guesswork — Every strategy is documented, tested, and ready-to-run
- Future-Proof — Lifetime updates included
- Proven Variety — Adaptive, mean-reversion, momentum, volatility, and ML-driven models

What's Inside
✅ **250+ Strategies across multiple categories:**

- Adaptive Filters — Kalman, Hilbert, TRIX, Guppy MMA
- Mean Reversion — Bollinger, Keltner, pair trading, RSI/Stoch
- Momentum & Trend — MACD, ADX, Donchian, MA ribbons
- Volatility — ATR breakouts, Bollinger squeeze, VIX correlation
- Specialized ML & Advanced — HMM, decision trees, isolation forest, Hurst exponent

You Get
📂 Python `.py` scripts for direct Backtrader use
📄 PDF manuals detailing logic, parameters, and best practices
🎯 Get Instant Access — Download Today, Trade Tonight

👉 **Get it here**

# 6.2 Components & formulas

Tenkan-sen (conversion line) over `tenkan` bars

$$\text{Tenkan}(t) = \frac{\max(H_{t-\ell..t}) + \min(L_{t-\ell..t})}{2}, \quad \ell = \text{tenkan}$$

Kijun-sen (base line) over `kijun` bars

$$\text{Kijun}(t) = \frac{\max(H_{t-\ell..t}) + \min(L_{t-\ell..t})}{2}, \quad \ell = \text{kijun}$$

Senkou Span A (cloud top/bottom piece)

$$\text{SpanA}(t) = \frac{\text{Tenkan}(t) + \text{Kijun}(t)}{2} \text{ shifted } + \text{senkou\_lead}$$

Senkou Span B (other cloud edge) over `senkou` bars

$$\text{SpanB}(t) = \frac{\max(H_{t-\ell..t}) + \min(L_{t-\ell..t})}{2} \text{ shifted } + \text{senkou\_lead}$$

Chikou Span (lagging line)

$$\text{Chikou}(t) = \text{Close}(t) \text{ shifted } - \text{chikou}$$

Bullish bias: price above both SpanA and SpanB; Tenkan > Kijun; Chikou above past price. Bearish is the mirror.

got it. here's Chapter 6's snippets again with **line-by-line**, practical explanations so readers can follow exactly what each piece does. no results/plots sections—just code + why.

# 6.3 Step-by-step build (with detailed explanations)

## 6.3.1 Add Ichimoku with your parameters

```
import backtrader as bt

class _Ichi_Part1(bt.Strategy):
    params = dict(tenkan=7, kijun=14, senkou=30, senkou_lead=14, chikou=14)
    def __init__(self):
        # Construct the full Ichimoku indicator on the primary data feed
        self.ichimoku = bt.ind.Ichimoku(
            self.datas[0],                  # use the first (main) data feed
            tenkan=self.p.tenkan,          # Tenkan lookback (conversion line)
            kijun=self.p.kijun,            # Kijun lookback (base line)
            senkou=self.p.senkou,          # Span B lookback (cloud breadth)
            senkou_lead=self.p.senkou_lead,# how far forward Span A/B are
plotted
            chikou=self.p.chikou           # how far back the Chikou span is
plotted
        )
```

What each line means

- `params = dict(...)` defines tunable inputs you can override in `cerebro.addstrategy(...)`.
- `bt.ind.Ichimoku(...)` builds **all five** components at once:

  - `tenkan_sen` = mid of highest high / lowest low over `tenkan` bars.
  - `kijun_sen` = mid over `kijun` bars.
  - `senkou_span_a` = (Tenkan + Kijun)/2 shifted forward.
  - `senkou_span_b` = mid of highs/lows over `senkou` bars, shifted forward.
  - `chikou_span` = current close shifted **back** by `chikou` bars.
- In Backtrader, the **shift** affects plotting; you can still read `[0]` for logical comparisons at the current bar.

## 6.3.2 Cloud breakout conditions

```
class _Ichi_Part2(bt.Strategy):
    params = dict(tenkan=7, kijun=14, senkou=30, senkou_lead=14, chikou=14)
    def __init__(self):
        self.ichimoku = bt.ind.Ichimoku(self.datas[0],
                                        tenkan=self.p.tenkan,
kijun=self.p.kijun,
                                        senkou=self.p.senkou,
senkou_lead=self.p.senkou_lead,
                                        chikou=self.p.chikou)
    def next(self):
        price = self.data.close[0]  # current close
        # Bullish bias if price is above BOTH cloud edges
        above_cloud = (price > self.ichimoku.senkou_span_a[0] and
                       price > self.ichimoku.senkou_span_b[0])
        # Bearish bias if price is below BOTH cloud edges
        below_cloud = (price < self.ichimoku.senkou_span_a[0] and
                       price < self.ichimoku.senkou_span_b[0])
```

Why this matters

- Requiring price above **both** Span A and Span B avoids “half-breakouts” that sit inside the cloud.
- The cloud adapts to volatility; being clear of it is a stronger regime signal than a simple MA cross.

## 6.3.3 Tenkan/Kijun cross and Chikou confirmation

```
class _Ichi_Part3(bt.Strategy):
    params = dict(tenkan=7, kijun=14, senkou=30, senkou_lead=14, chikou=14)
    def __init__(self):
        self.ichimoku = bt.ind.Ichimoku(self.datas[0],
                                        tenkan=self.p.tenkan,
```

```
kijun=self.p.kijun,
                                        senkou=self.p.senkou,
senkou_lead=self.p.senkou_lead,
                                        chikou=self.p.chikou)
    def next(self):
        # TK cross gives direction and momentum
        tk_bull = self.ichimoku.tenkan_sen[0] > self.ichimoku.kijun_sen[0]
        tk_bear = self.ichimoku.tenkan_sen[0] < self.ichimoku.kijun_sen[0]

        # Chikou confirms breadth by comparing to past price
        chikou_bull = self.ichimoku.chikou_span[0] > self.data.high[-
self.p.chikou]
        chikou_bear = self.ichimoku.chikou_span[0] < self.data.low[-
self.p.chikou]
```

Why this matters

- **Tenkan > Kijun**: faster line above slower line → supportive momentum.
- **Chikou > past price**: current price (shifted back) clears prior resistance; the mirror for shorts.
- Using `high[-chikou]` / `low[-chikou]` is stricter than comparing to `close`—fewer false confirms.
- Warm-up: these references require at least `chikou` bars of history; Backtrader delays indicator output automatically, but if you hand-roll, guard with `if len(self) > self.p.chikou:`.

# 6.3.4 Trailing stop with StopTrail

```
class _Ichi_Part4(bt.Strategy):
    params = dict(trail_percent=0.02)  # NOTE: 0.02 = 2% trail
    def notify_order(self, order):
        if order.status in [order.Submitted, order.Accepted]:
            return  # wait for fill

        if order.status in [order.Completed]:
            if order.isbuy():
                # Immediately attach a trailing SELL that follows price up
                self.sell(exectype=bt.Order.StopTrail,
                          trailpercent=self.p.trail_percent)
            elif order.issell():
                # Immediately attach a trailing BUY that follows price down
                self.buy(exectype=bt.Order.StopTrail,
                         trailpercent=self.p.trail_percent)
        # Clear the pending-order lock so next() may submit new orders later
```

```
        # (Backtrader pattern—prevents stacking unintended duplicates)
        self.order = None
```

Why this matters

- `StopTrail` moves with favorable price; when price reverses by `trailpercent`, it triggers.
- `trailpercent=0.02` means **2%**—if you intended 4%, set `0.04` (the comment in your original text said “4%”).
- Many users also pass `size=abs(self.position.size)` to tie the protective order exactly to the current exposure. Your version will work as is, but adding size is more explicit.

# 6.3.5 Entry logic joins the pieces

```
class _Ichi_Part5(bt.Strategy):
    params = dict(tenkan=7, kijun=14, senkou=30, senkou_lead=14, chikou=14,
trail_percent=0.02)
    def __init__(self):
        self.order = None
        self.ichimoku = bt.ind.Ichimoku(self.datas[0],
                                        tenkan=self.p.tenkan,
kijun=self.p.kijun,
                                        senkou=self.p.senkou,
senkou_lead=self.p.senkou_lead,
                                        chikou=self.p.chikou)
    def notify_order(self, order):
        if order.status in [order.Submitted, order.Accepted]:
            return
        if order.status in [order.Completed]:
            if order.isbuy():
                self.sell(exectype=bt.Order.StopTrail,
trailpercent=self.p.trail_percent)
            elif order.issell():
                self.buy(exectype=bt.Order.StopTrail,
trailpercent=self.p.trail_percent)
        self.order = None

    def next(self):
        if self.order:              # don’t place a new order until prior one
resolves
            return
        if self.position:           # entries only when flat (your chosen
style)
            return
```

```
        price = self.data.close[0]

        # 1) Cloud filter
        is_above_cloud = (price > self.ichimoku.senkou_span_a[0] and
                          price > self.ichimoku.senkou_span_b[0])
        is_below_cloud = (price < self.ichimoku.senkou_span_a[0] and
                          price < self.ichimoku.senkou_span_b[0])

        # 2) Tenkan/Kijun alignment
        is_tk_cross_bullish = self.ichimoku.tenkan_sen[0] >
self.ichimoku.kijun_sen[0]
        is_tk_cross_bearish = self.ichimoku.tenkan_sen[0] <
self.ichimoku.kijun_sen[0]

        # 3) Chikou confirmation vs past price
        is_chikou_bullish = self.ichimoku.chikou_span[0] > self.data.high[-
self.p.chikou]
        is_chikou_bearish = self.ichimoku.chikou_span[0] < self.data.low[-
self.p.chikou]

        # Final composite entry
        if is_above_cloud and is_tk_cross_bullish and is_chikou_bullish:
            self.order = self.buy()
        elif is_below_cloud and is_tk_cross_bearish and is_chikou_bearish:
            self.order = self.sell()
```

Why this matters

- You're "AND-ing" three confirmations: regime (cloud), momentum (TK), and breadth (Chikou). This reduces whipsaws at the cost of fewer trades.
- Entries only when flat keeps the system simple. If you want pyramids later, you'd remove the `if self.position: return` guard and add rules for adds/reductions.

# 6.4 Complete Backtrader strategy (your exact class)

```
import backtrader as bt

class IchimokuCloudStrategy(bt.Strategy):
    """
    Trades on a confirmed breakout from the Ichimoku Cloud (Kumo).
    1.  Price breaks out of the Kumo.
    2.  Tenkan/Kijun cross confirms momentum.
    3.  Chikou Span confirms the trend.
    4.  Exit is managed with a trailing stop-loss.
    """
```

```
    params = (
        # Default Ichimoku parameters
        ('tenkan', 7),
        ('kijun', 14),
        ('senkou', 30),
        ('senkou_lead', 14),  # How far forward to plot the cloud
        ('chikou', 14),       # How far back to plot the lagging span
        # Strategy parameters
        ('trail_percent', 0.02), # Trailing stop loss of 4%
    )

    def __init__(self):
        self.order = None

        # Add the Ichimoku indicator with its parameters
        self.ichimoku = bt.indicators.Ichimoku(
            self.datas[0],
            tenkan=self.p.tenkan,
            kijun=self.p.kijun,
            senkou=self.p.senkou,
            senkou_lead=self.p.senkou_lead,
            chikou=self.p.chikou
        )

    def notify_order(self, order):
        if order.status in [order.Submitted, order.Accepted]:
            return

        if order.status in [order.Completed]:
            if order.isbuy():
                # Place a trailing stop for the long position
                self.sell(exectype=bt.Order.StopTrail,
trailpercent=self.p.trail_percent)
            elif order.issell():
                # Place a trailing stop for the short position
                self.buy(exectype=bt.Order.StopTrail,
trailpercent=self.p.trail_percent)

        self.order = None

    def next(self):
        # Check for pending orders
        if self.order:
            return

        # Check if we are in a position
```

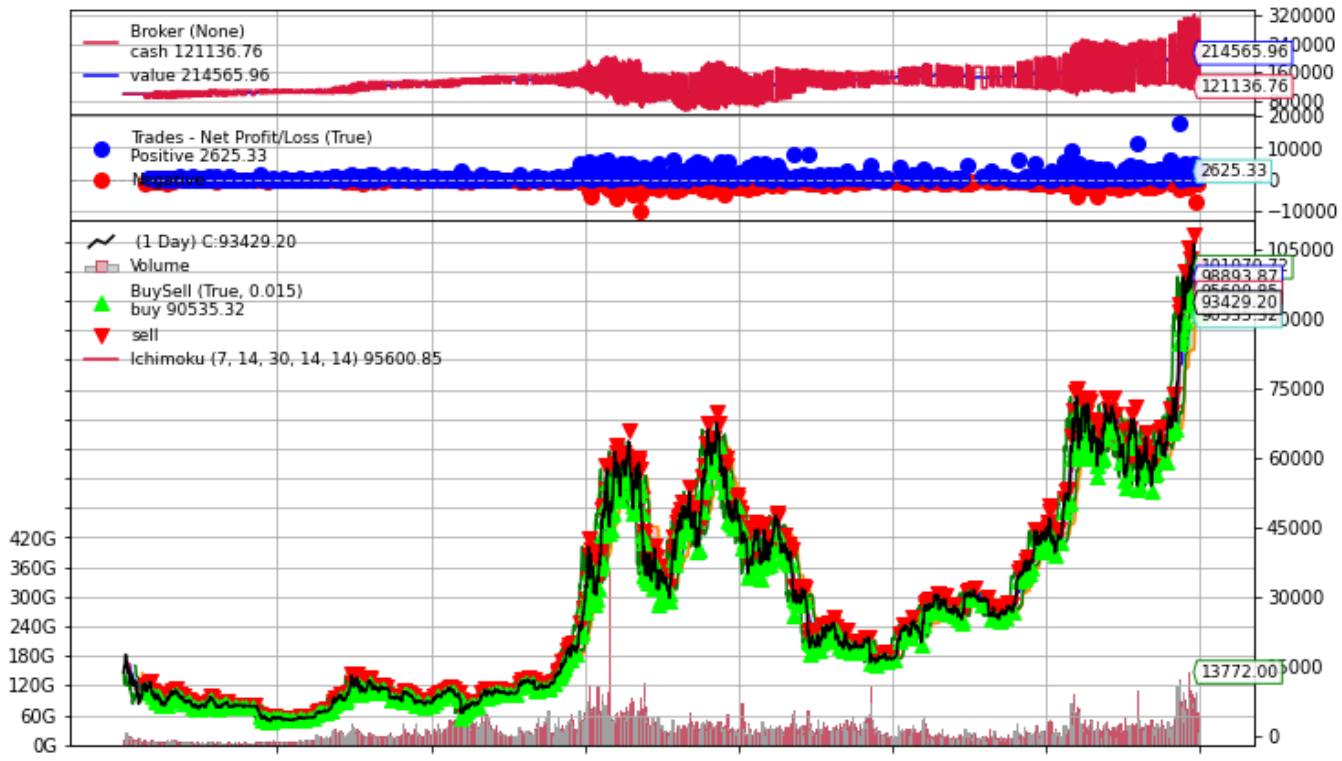
```
        if not self.position:
            # --- Bullish Entry Conditions ---
            # 1. Price is above both lines of the Kumo cloud
            is_above_cloud = (self.data.close[0] >
self.ichimoku.senkou_span_a[0] and
                              self.data.close[0] >
self.ichimoku.senkou_span_b[0])

            # 2. Tenkan-sen is above Kijun-sen
            is_tk_cross_bullish = self.ichimoku.tenkan_sen[0] >
self.ichimoku.kijun_sen[0]

            # 3. Chikou Span is above the price from 26 periods ago
            is_chikou_bullish = self.ichimoku.chikou_span[0] >
self.data.high[-self.p.chikou]

            if is_above_cloud and is_tk_cross_bullish and is_chikou_bullish:
                self.order = self.buy()

            # --- Bearish Entry Conditions ---
            # 1. Price is below both lines of the Kumo cloud
            is_below_cloud = (self.data.close[0] <
self.ichimoku.senkou_span_a[0] and
                              self.data.close[0] <
self.ichimoku.senkou_span_b[0])

            # 2. Tenkan-sen is below Kijun-sen
            is_tk_cross_bearish = self.ichimoku.tenkan_sen[0] <
self.ichimoku.kijun_sen[0]

            # 3. Chikou Span is below the price from 26 periods ago
            is_chikou_bearish = self.ichimoku.chikou_span[0] < self.data.low[-
self.p.chikou]

            if is_below_cloud and is_tk_cross_bearish and is_chikou_bearish:
                self.order = self.sell()
```

# 6.5 Backtesting

```
import backtrader as bt
import yfinance as yf
import pandas as pd
import numpy as np
import dateutil.relativedelta as rd
import matplotlib.pyplot as plt
```

```
%matplotlib inline
import seaborn as sns
import importlib

def load_strategy(class_name):
    module = importlib.import_module(class_name)
    strategy = getattr(module, class_name)
    return strategy

ticker = "BTC-USD"
start = "2018-01-01"
end = "2025-01-01"

start_dt = pd.to_datetime(start)
end_dt = pd.to_datetime(end)


strategy = load_strategy("IchimokuCloudStrategy")

data = yf.download(ticker, start=start, end=end, progress=False)


data = data.droplevel(1, 1) if isinstance(data.columns, pd.MultiIndex) else
data

feed = bt.feeds.PandasData(dataname=data)
cerebro = bt.Cerebro()
cerebro.addstrategy(strategy)
cerebro.adddata(feed)
cerebro.broker.setcash(100000)
cerebro.broker.setcommission(commission=0.001)
# cerebro.addsizer(bt.sizers.PercentSizer, percents=95)

cerebro.addanalyzer(bt.analyzers.SharpeRatio, _name='sharpe',
timeframe=bt.TimeFrame.Days, riskfreerate=0.0)
cerebro.addanalyzer(bt.analyzers.DrawDown,    _name='dd')
cerebro.addanalyzer(bt.analyzers.Returns,     _name='rets')
cerebro.addanalyzer(bt.analyzers.TradeAnalyzer, _name='trades')

start_val = cerebro.broker.getvalue()
results = cerebro.run()
strat = results[0]
final_val = cerebro.broker.getvalue()
ret = (final_val - start_val) / start_val * 100
```

```
print(f"Return: {ret:.2f}% | Final Value: {final_val:.2f}")



sh = strat.analyzers.sharpe.get_analysis().get('sharperatio', None)
dd = strat.analyzers.dd.get_analysis()
rets = strat.analyzers.rets.get_analysis()
tr = strat.analyzers.trades.get_analysis()

print(f"Sharpe: {sh}")
print(f"Max DD: {dd.get('max', {}).get('drawdown', 0):.2f}%  Len:
{dd.get('max', {}).get('len', 0)}")
print(f"CAGR:   {rets.get('rnorm100', 0):.2f}%  Total: {rets.get('rtot',
0)*100:.2f}%")
print(f"Trades: {tr.get('total', {}).get('total', 0)}  Won: {tr.get('won',
{}).get('total', 0)}  Lost: {tr.get('lost', {}).get('total', 0)}")

plt.rcParams['figure.figsize'] = [10, 6]
plt.rcParams['font.size'] = 10
cerebro.plot(iplot=False)
```




# 6.6 Rolling Backtest

To comprehensively evaluate the strategy's performance, a **rolling backtest** is used. This method assesses the strategy over multiple, successive time windows, providing a more robust

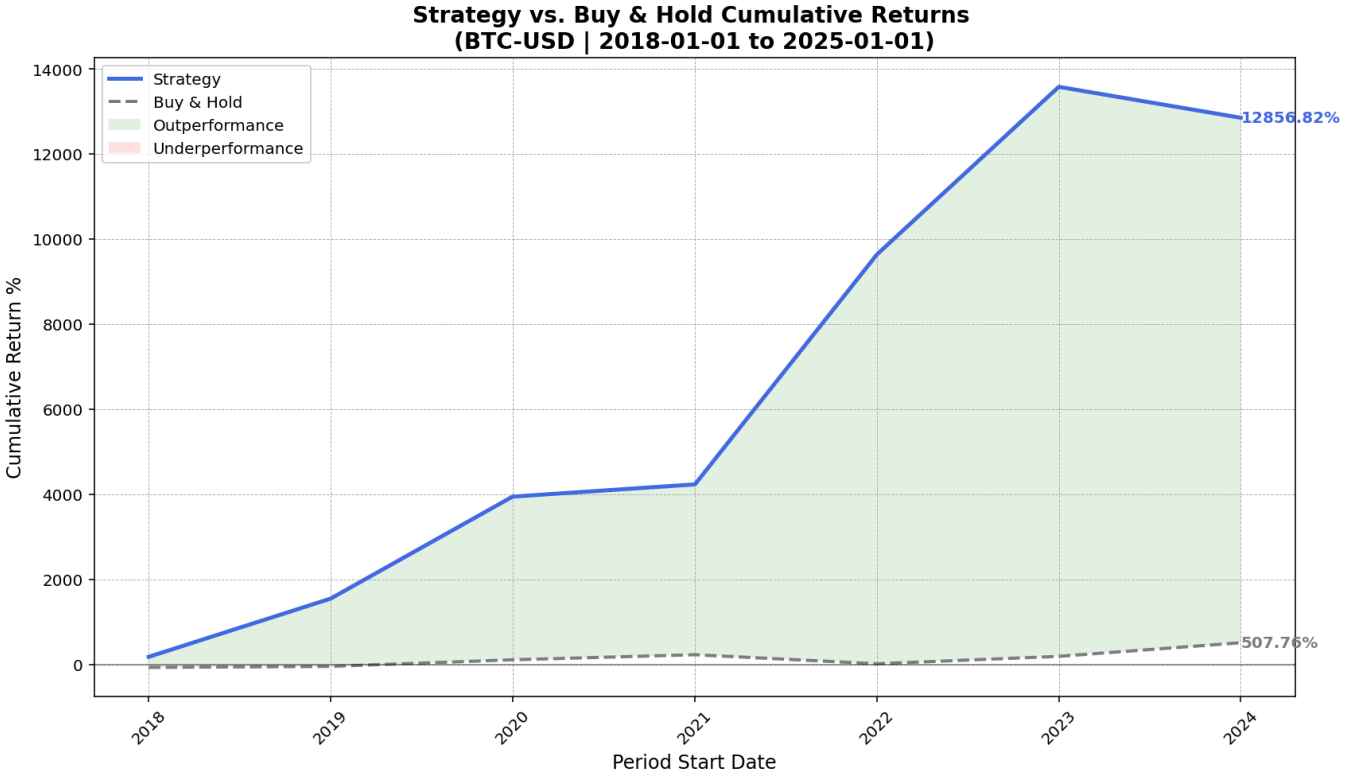
view of its consistency compared to a single, fixed backtest. Here we tried an **ATR Threshold**: The current Average True Range (ATR) must be **above its N-period Simple Moving Average**. This ensures that the market has sufficient volatility to support a trend and that the breakout isn't occurring in a flat or consolidating market. Trading is avoided in low-volatility environments where false breakouts are common. Let's see how the enhanced version performs.

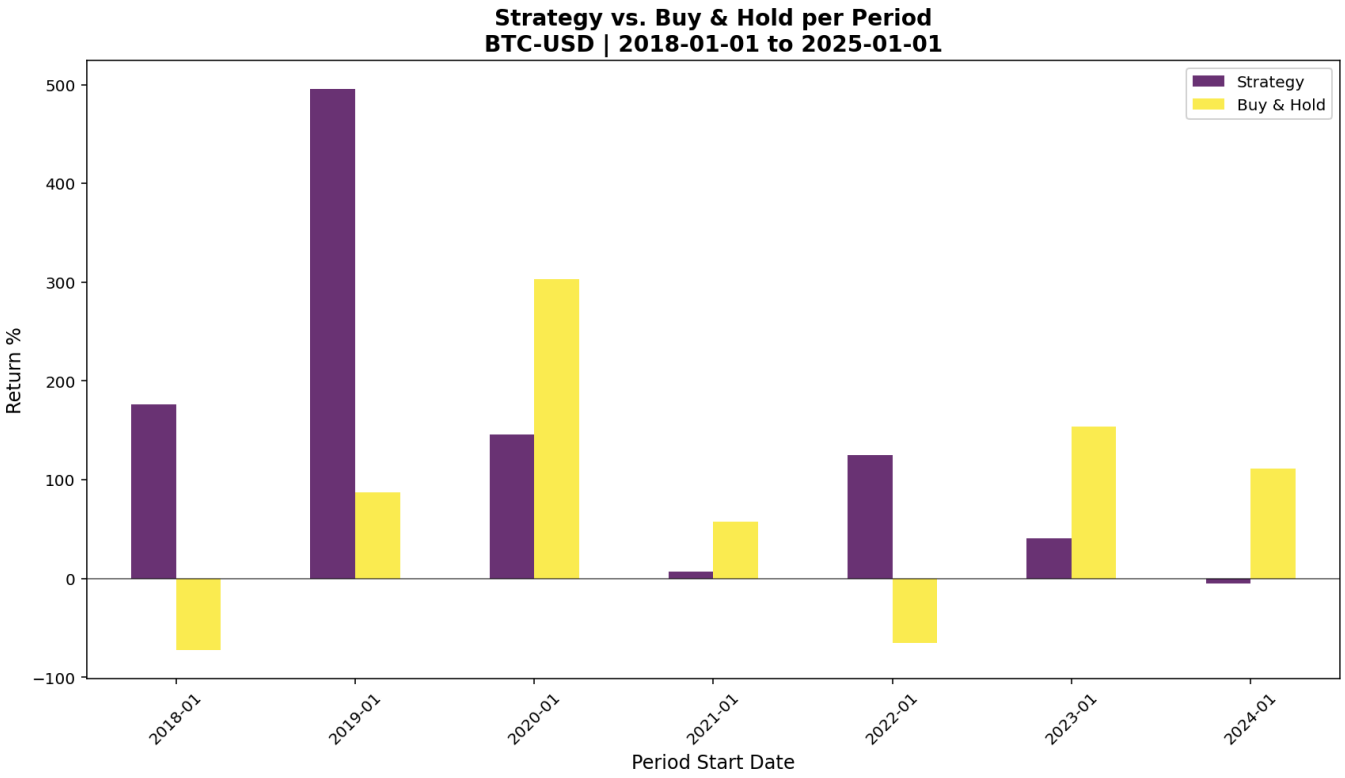
```
from collections import deque
import backtrader as bt
import numpy as np
import pandas as pd
import yfinance as yf
import dateutil.relativedelta as rd

# Assuming VolatilityFilteredIchimokuStrategy class is defined above this
section.

def run_rolling_backtest(
    ticker,
    start,
    end,
    window_months,
    strategy_class,
    strategy_params=None
):
    strategy_params = strategy_params or {}
    all_results = []
    start_dt = pd.to_datetime(start)
    end_dt = pd.to_datetime(end)
    current_start = start_dt

    while True:
        current_end = current_start + rd.relativedelta(months=window_months)
        if current_end > end_dt:
            current_end = end_dt
            if current_start >= current_end:
                break

        print(f"\nROLLING BACKTEST: {current_start.date()} to
{current_end.date()}")

        data = yf.download(ticker, start=current_start, end=current_end,
auto_adjust=False, progress=False)

        if data.empty or len(data) < 90:
            print("Not enough data for this period. Skipping.")
```

```
            current_start += rd.relativedelta(months=window_months)
            continue

        if isinstance(data.columns, pd.MultiIndex):
            data = data.droplevel(1, 1)

        start_price = data['Close'].iloc[0]
        end_price = data['Close'].iloc[-1]
        benchmark_ret = (end_price - start_price) / start_price * 100

        feed = bt.feeds.PandasData(dataname=data)
        cerebro = bt.Cerebro()

        cerebro.addstrategy(strategy_class, **strategy_params)
        cerebro.adddata(feed)
        cerebro.broker.setcash(100000)
        cerebro.broker.setcommission(commission=0.001)
        cerebro.addsizer(bt.sizers.PercentSizer, percents=95)

        start_val = cerebro.broker.getvalue()
        try:
            cerebro.run()
        except Exception as e:
            print(f"Error running backtest for {current_start.date()} to
{current_end.date()}: {e}")
            current_start += rd.relativedelta(months=window_months)
            continue

        final_val = cerebro.broker.getvalue()
        strategy_ret = (final_val - start_val) / start_val * 100

        all_results.append({
            'start': current_start.date(),
            'end': current_end.date(),
            'strategy_return_pct': strategy_ret,
            'benchmark_return_pct': benchmark_ret,
            'final_value': final_val,
        })

        print(f"Strategy Return: {strategy_ret:.2f}% | Buy & Hold Return:
{benchmark_ret:.2f}%")

        current_start += rd.relativedelta(months=window_months)

        if current_start > end_dt:
            break
```

```
    return pd.DataFrame(all_results)
```

- `ticker`, `start`, `end`: The asset symbol and the overall historical period for the backtest.
- `window_months`: The duration of each individual backtesting window in months.
- `strategy_class`: The `backtrader.Strategy` class to be tested (e.g., `VolatilityFilteredIchimokuStrategy`).
- `strategy_params`: A dictionary to pass specific parameters to the chosen strategy for each run.
- The function iterates through the overall timeframe, creating consecutive time windows. For each window, it:
  - Downloads historical data using `yfinance` with `auto_adjust=False` and `droplevel` applied.
  - Initializes a fresh `backtrader.Cerebro` instance.
  - Adds the specified `strategy_class` and its parameters.
  - Executes the backtest for that specific window.
  - Calculates and records the strategy's return and a simple buy-and-hold benchmark return for that period.
- The function returns a `Pandas DataFrame` containing the comprehensive results for each rolling window.


Strategy vs. Buy & Hold Cumulative Returns
(BTC-USD | 2018-01-01 to 2025-01-01)
Strategy
Buy & Hold
Outperformance
Underperformance
12856.82%
507.76%
Cumulative Return %
Period Start Date


Strategy vs. Buy & Hold per Period
BTC-USD | 2018-01-01 to 2025-01-01
Strategy
Buy & Hold
Return %
Period Start Date

**Key Performance Indicators**
**BTC-USD | 2018-01-01 to 2025-01-01**

| Metric | Value |
|---|---|
| Total Periods | 7 |
| Winning Periods | 6 |
| Losing Periods | 1 |
| Mean Return % | 140.71 |
| Median Return % | 124.79 |
| Std Dev % | 158.98 |
| Win Rate % | 85.71 |
| Sharpe Ratio | 0.89 |
| Min Return % | -5.32 |
| Max Return % | 495.92 |

The `VolatilityFilteredIchimokuStrategy` offers a sophisticated approach to trend trading by leveraging the comprehensive nature of the **Ichimoku Kinko Hyo** and enhancing it with a **dynamic volatility filter**. By ensuring trades are only entered when volatility is conducive to strong directional moves, it aims to reduce false signals and improve overall strategy robustness. The implementation of a **trailing stop-loss** provides essential risk management. The use of a **rolling backtest** is vital for thoroughly evaluating the strategy's consistency and adaptability across various market conditions, providing a more reliable assessment of its performance.

# Chapter 7 — Keltner Channel Breakout

Idea: detect **range expansion** by comparing yesterday’s close to yesterday’s Keltner bands (EMA centerline ± ATR·multiplier). Enter at today’s open after a confirmed breakout; exit when price reverts back through the EMA midline. Using yesterday’s signal avoids look-ahead bias.

This strategy is part of the **Mega Backtrader Strategy Pack** — a collection of over **250 ready-to-use strategies** for your algorithmic trading.

The **Mega Backtrader Strategy Pack** gives you **250+ fully-tested Python strategies** — adaptive filters, mean reversion, momentum, volatility breakouts, machine learning models & more — all ready to deploy in Backtrader.

**Why Traders Love This Pack**

- Instant Impact — Deploy strategies and start testing today

- Zero Guesswork — Every strategy is documented, tested, and ready-to-run
- Future-Proof — Lifetime updates included
- Proven Variety — Adaptive, mean-reversion, momentum, volatility, and ML-driven models

What's Inside

✅ **250+ Strategies across multiple categories:**

- Adaptive Filters — Kalman, Hilbert, TRIX, Guppy MMA
- Mean Reversion — Bollinger, Keltner, pair trading, RSI/Stoch
- Momentum & Trend — MACD, ADX, Donchian, MA ribbons
- Volatility — ATR breakouts, Bollinger squeeze, VIX correlation
- Specialized ML & Advanced — HMM, decision trees, isolation forest, Hurst exponent

You Get

📂 Python `.py` scripts for direct Backtrader use
📄 PDF manuals detailing logic, parameters, and best practices
🎯 Get Instant Access — Download Today, Trade Tonight

👉 **Get it here**

# 7.1 Components and formulas

Keltner Channel with EMA centerline and ATR-based bands

- Centerline $\mathrm{mid}_t = \mathrm{EMA}_n(\mathrm{Close})$
- Width $w_t = \mathrm{ATR}_m$
- Upper/Lower $\mathrm{top}_t = \mathrm{mid}_t + k \cdot w_t,\ \mathrm{bot}_t = \mathrm{mid}_t - k \cdot w_t$

Breakout signal (evaluated on the **previous bar**)

- Long setup when $\mathrm{Close}_{t-1} > \mathrm{top}_{t-1}$
- Short setup when $\mathrm{Close}_{t-1} < \mathrm{bot}_{t-1}$

Exit rule (evaluated on current bar)

- For longs, exit when $\mathrm{Close}_t < \mathrm{mid}_t$
- For shorts, exit when $\mathrm{Close}_t > \mathrm{mid}_t$

# 7.2 Step-by-step build with explanations

## 7.2.1 Custom Keltner Channel indicator

```
import backtrader as bt

class KeltnerChannel(bt.Indicator):
    """
    Keltner Channel indicator with EMA centerline and ATR-based bands
    """
    lines = ('mid', 'top', 'bot')
    params = (
        ('ema_period', 30),
        ('atr_period', 14),
        ('atr_multiplier', 1.0),
    )

    def __init__(self):
        # EMA centerline on the instrument's Close
        self.lines.mid = bt.indicators.EMA(self.data.close,
period=self.params.ema_period)

        # ATR measures typical bar range; used to scale band width
        atr = bt.indicators.ATR(self.data, period=self.params.atr_period)

        # Upper / lower bands = EMA ± ATR × multiplier
        self.lines.top = self.lines.mid + (atr * self.params.atr_multiplier)
        self.lines.bot = self.lines.mid - (atr * self.params.atr_multiplier)
```

What's happening

- `lines = ('mid','top','bot')` declares three output series the rest of the code can read.
- `EMA` is the midline (trend proxy). `ATR` provides volatility-adaptive width.
- Multiplying ATR by `atr_multiplier` lets you tighten/loosen the breakout threshold.

## 7.2.2 Strategy skeleton, logging, and indicator wiring

```
class KeltnerBreakoutStrategy(bt.Strategy):
    params = (
        ('ema_period', 30),
        ('atr_period', 7),
        ('atr_multiplier', 1.0),
        ('printlog', True),
    )

    def log(self, txt, dt=None):
        if self.params.printlog:
            dt = dt or self.datas[0].datetime.date(0)
```

```
            print(f'{dt.isoformat()}: {txt}')

    def __init__(self):
        # Handy references for readability
        self.dataclose = self.datas[0].close
        self.dataopen  = self.datas[0].open

        # Instantiate the custom Keltner channel with your chosen params
        self.keltner = KeltnerChannel(
            self.data,
            ema_period=self.params.ema_period,
            atr_period=self.params.atr_period,
            atr_multiplier=self.params.atr_multiplier
        )

        # Order state and bookkeeping
        self.order = None
        self.buyprice = None
        self.buycomm = None

        # Optional plotting flags for nicer chart overlays
        self.keltner.plotinfo.subplot = False
        self.keltner.plotlines.mid._plotskip = False
        self.keltner.plotlines.top._plotskip = False
        self.keltner.plotlines.bot._plotskip = False
```

Why this matters

- `self.dataclose` / `self.dataopen` make the logic more readable in `next`.
- Wiring the indicator once in `__init__` makes its lines available every bar.
- The plotting flags don't affect logic—they just tell Backtrader to overlay bands on price.

## 7.2.3 Order notifications and trade PnL

```
    def notify_order(self, order):
        if order.status in [order.Submitted, order.Accepted]:
            return  # pending; nothing to do yet

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

```
            elif order.issell():
                self.log(f'SELL EXECUTED: Price: {order.executed.price:.2f}, '
                         f'Cost: {order.executed.value:.2f}, Comm:
{order.executed.comm:.2f}')
            self.bar_executed = len(self)

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

        # Always clear the order lock so next() can place new orders later
        self.order = None

    def notify_trade(self, trade):
        if not trade.isclosed:
            return
        self.log(f'OPERATION PROFIT: Gross {trade.pnl:.2f}, Net
{trade.pnlcomm:.2f}')
```

Why this matters

- `notify_order` is called when orders change state; logging here shows fills and costs.
- Clearing `self.order` after completion ensures you don't block future signals.
- `notify_trade` fires when a position round-trip closes; good for quick sanity checks.

## 7.2.4 The trading logic in `next`

```
    def next(self):
        # 1) Safety rails: enough data and no pending orders
        if len(self.data) < max(self.params.ema_period,
self.params.atr_period):
            return
        if self.order:
            return
        if len(self.data) < 2:
            return  # need yesterday's values for a non-look-ahead signal

        # 2) Read yesterday's breakout condition (no look-ahead)
        prev_close = self.dataclose[-1]
        prev_upper = self.keltner.top[-1]
        prev_lower = self.keltner.bot[-1]

        # 3) Read today's midline and close for exits
        current_ema   = self.keltner.mid[0]
        current_close = self.dataclose[0]
```

```
        # 4) Entries only when flat
        if not self.position:
            # Long: yesterday closed above yesterday's upper band → expansion
up
            if prev_close > prev_upper:
                self.log(f'BUY CREATE: Price {self.dataopen[0]:.2f} '
                         f'(Prev Close: {prev_close:.2f} > Upper:
{prev_upper:.2f})')
                self.order = self.buy()  # will execute at the broker's next
bar logic

            # Short: yesterday closed below yesterday's lower band → expansion
down
            elif prev_close < prev_lower:
                self.log(f'SELL CREATE: Price {self.dataopen[0]:.2f} '
                         f'(Prev Close: {prev_close:.2f} < Lower:
{prev_lower:.2f})')
                self.order = self.sell()

        # 5) Exits when in a position: cross back through EMA midline
        else:
            if self.position.size > 0:  # long
                if current_close < current_ema:
                    self.log(f'CLOSE LONG: Price {current_close:.2f} < EMA
{current_ema:.2f}')
                    self.order = self.close()
            elif self.position.size < 0:  # short
                if current_close > current_ema:
                    self.log(f'CLOSE SHORT: Price {current_close:.2f} > EMA
{current_ema:.2f}')
                    self.order = self.close()
```

Why this matters

- Using `[-1]` for bands and close means the **signal is formed on the completed bar**, and the order is placed for the next bar → correct backtesting practice.
- Entries are symmetric (long and short); delete the short branch if your venue is long-only.
- Exits are simple and consistent: revert to trend midline = give back some gains, avoid flip-flops.

## 7.2.5 Optional: tiny analyzer to track account value over time

Your class is named `TradeAnalyzer`, but Backtrader already has a built-in `TradeAnalyzer`. To avoid confusion, rename this to something like `EquityTracker`.

```
class EquityTracker(bt.Analyzer):
    def create_analysis(self):
        self.values = []

    def notify_cashvalue(self, cash, value):
        self.values.append(value)

    def get_analysis(self):
        return {
            'final_value': self.values[-1] if self.values else 0,
            'max_value': max(self.values) if self.values else 0
        }
```

Why this matters

- Analyzers are easy hooks for collecting metrics without cluttering your strategy.
- Renaming prevents accidentally shadowing `bt.analyzers.TradeAnalyzer`.

# 7.3 Complete code (indicator + strategy + analyzer)

```
import backtrader as bt

class KeltnerChannel(bt.Indicator):
    """
    Keltner Channel indicator with EMA centerline and ATR-based bands
    """
    lines = ('mid', 'top', 'bot')
    params = (
        ('ema_period', 30),
        ('atr_period', 14),
        ('atr_multiplier', 1.0),
    )

    def __init__(self):
        self.lines.mid = bt.indicators.EMA(self.data.close,
period=self.params.ema_period)
        atr = bt.indicators.ATR(self.data, period=self.params.atr_period)
        self.lines.top = self.lines.mid + (atr * self.params.atr_multiplier)
        self.lines.bot = self.lines.mid - (atr * self.params.atr_multiplier)


class KeltnerBreakoutStrategy(bt.Strategy):
    params = (
        ('ema_period', 30),
```

```
        ('atr_period', 7),
        ('atr_multiplier', 1.0),
        ('printlog', True),
    )

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

    def __init__(self):
        self.dataclose = self.datas[0].close
        self.dataopen  = self.datas[0].open

        self.keltner = KeltnerChannel(
            self.data,
            ema_period=self.params.ema_period,
            atr_period=self.params.atr_period,
            atr_multiplier=self.params.atr_multiplier
        )

        self.order = None
        self.buyprice = None
        self.buycomm = None

        # plotting (optional)
        self.keltner.plotinfo.subplot = False
        self.keltner.plotlines.mid._plotskip = False
        self.keltner.plotlines.top._plotskip = False
        self.keltner.plotlines.bot._plotskip = False

    def notify_order(self, order):
        if order.status in [order.Submitted, order.Accepted]:
            return

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

```
            self.bar_executed = len(self)

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

        self.order = None

    def notify_trade(self, trade):
        if not trade.isclosed:
            return
        self.log(f'OPERATION PROFIT: Gross {trade.pnl:.2f}, Net
{trade.pnlcomm:.2f}')

    def next(self):
        if len(self.data) < max(self.params.ema_period,
self.params.atr_period):
            return
        if self.order:
            return
        if len(self.data) < 2:
            return

        prev_close = self.dataclose[-1]
        prev_upper = self.keltner.top[-1]
        prev_lower = self.keltner.bot[-1]
        current_ema   = self.keltner.mid[0]
        current_close = self.dataclose[0]

        if not self.position:
            if prev_close > prev_upper:
                self.log(f'BUY CREATE: Price {self.dataopen[0]:.2f} '
                         f'(Prev Close: {prev_close:.2f} > Upper:
{prev_upper:.2f})')
                self.order = self.buy()
            elif prev_close < prev_lower:
                self.log(f'SELL CREATE: Price {self.dataopen[0]:.2f} '
                         f'(Prev Close: {prev_close:.2f} < Lower:
{prev_lower:.2f})')
                self.order = self.sell()
        else:
            if self.position.size > 0:
                if current_close < current_ema:
                    self.log(f'CLOSE LONG: Price {current_close:.2f} < EMA
{current_ema:.2f}')
                    self.order = self.close()
            elif self.position.size < 0:
```

```
                if current_close > current_ema:
                    self.log(f'CLOSE SHORT: Price {current_close:.2f} > EMA
{current_ema:.2f}')
                    self.order = self.close()


class EquityTracker(bt.Analyzer):
    def create_analysis(self):
        self.values = []
    def notify_cashvalue(self, cash, value):
        self.values.append(value)
    def get_analysis(self):
        return {
            'final_value': self.values[-1] if self.values else 0,
            'max_value': max(self.values) if self.values else 0
        }
```

# 7.4 Backtesting

```
import backtrader as bt
import yfinance as yf
import pandas as pd
import numpy as np
import dateutil.relativedelta as rd
import matplotlib.pyplot as plt
%matplotlib inline
import seaborn as sns
import importlib

def load_strategy(class_name):
    module = importlib.import_module(class_name)
    strategy = getattr(module, class_name)
    return strategy

ticker = "BTC-USD"
start = "2018-01-01"
end = "2025-01-01"

start_dt = pd.to_datetime(start)
end_dt = pd.to_datetime(end)


strategy = load_strategy("KeltnerBreakoutStrategy")

data = yf.download(ticker, start=start, end=end, progress=False)
```

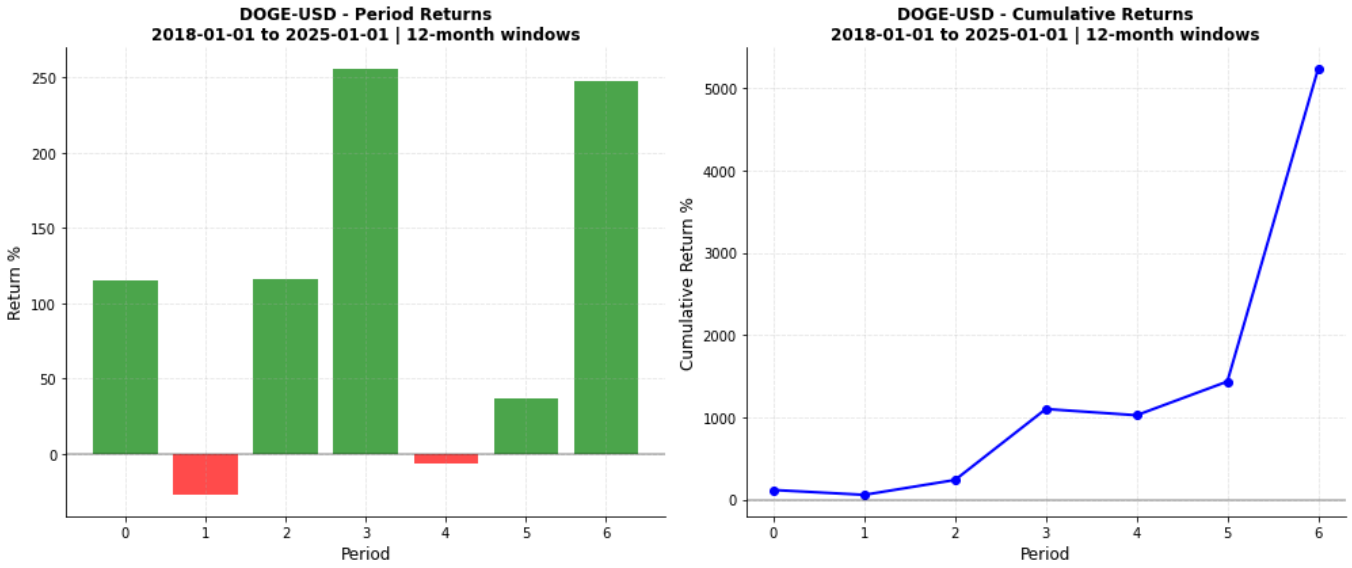
```
data = data.droplevel(1, 1) if isinstance(data.columns, pd.MultiIndex) else
data

feed = bt.feeds.PandasData(dataname=data)
cerebro = bt.Cerebro()
cerebro.addstrategy(strategy)
cerebro.adddata(feed)
cerebro.broker.setcash(100000)
cerebro.broker.setcommission(commission=0.001)
# cerebro.addsizer(bt.sizers.PercentSizer, percents=95)

cerebro.addanalyzer(bt.analyzers.SharpeRatio, _name='sharpe',
timeframe=bt.TimeFrame.Days, riskfreerate=0.0)
cerebro.addanalyzer(bt.analyzers.DrawDown,    _name='dd')
cerebro.addanalyzer(bt.analyzers.Returns,     _name='rets')
cerebro.addanalyzer(bt.analyzers.TradeAnalyzer, _name='trades')

start_val = cerebro.broker.getvalue()
results = cerebro.run()
strat = results[0]
final_val = cerebro.broker.getvalue()
ret = (final_val - start_val) / start_val * 100


print(f"Return: {ret:.2f}% | Final Value: {final_val:.2f}")



sh = strat.analyzers.sharpe.get_analysis().get('sharperatio', None)
dd = strat.analyzers.dd.get_analysis()
rets = strat.analyzers.rets.get_analysis()
tr = strat.analyzers.trades.get_analysis()

print(f"Sharpe: {sh}")
print(f"Max DD: {dd.get('max', {}).get('drawdown', 0):.2f}%  Len:
{dd.get('max', {}).get('len', 0)}")
print(f"CAGR:   {rets.get('rnorm100', 0):.2f}%  Total: {rets.get('rtot',
0)*100:.2f}%")
print(f"Trades: {tr.get('total', {}).get('total', 0)}  Won: {tr.get('won',
{}).get('total', 0)}  Lost: {tr.get('lost', {}).get('total', 0)}")

plt.rcParams['figure.figsize'] = [10, 6]
plt.rcParams['font.size'] = 10
```

```
cerebro.plot(iplot=False)
```


# 7.5 Rolling Backtest

Rolling 12-Month Windows (DOGE-USD)

```
if __name__ == '__main__':
    df = run_rolling_backtest(
        strategy_cls=KeltnerBreakoutStrategy,
        ticker="DOGE-USD",
        start="2018-01-01",
        end="2025-01-01",
        window_months=12,
        strategy_params=dict(ema_period=30, atr_period=14,
atr_multiplier=1.0),
        initial_cash=100_000,
        commission=0.001,
        stake_pct=95,
        min_bars=90
    )
    print("\nROLLING SUMMARY")
    print(df)
```




- **ATR window & multiplier:** Wider bands (larger ATR multiplier) reduce false breaks in choppy regimes; narrower bands increase frequency.
- **EMA period:** Longer midlines favor trend capture and delay exits; shorter midlines switch faster but may whipsaw.
- **Exit model:** Swap EMA exit for an ATR-based trailing stop in high-vol assets, or a hybrid: exit on EMA breach or trail hit, whichever comes first.
- **Regime filter:** Gate entries with a rising long SMA slope or ADX>20 to avoid low-energy expansions.

# Chapter 8 — Hybrid Keltner + RSI Breakout

**Idea**
Use a Keltner Channel (EMA ± ATR×multiplier) to detect **range expansion** and gate entries with a lightweight **RSI confirmation** so we don’t buy the weakest spikes or short the weakest dips. Risk is managed by a **manual ATR trailing stop**.

- Proven Variety — Adaptive, mean-reversion, momentum, volatility, and ML-driven models

What's Inside

✅ **250+ Strategies across multiple categories:**

- Adaptive Filters — Kalman, Hilbert, TRIX, Guppy MMA
- Mean Reversion — Bollinger, Keltner, pair trading, RSI/Stoch
- Momentum & Trend — MACD, ADX, Donchian, MA ribbons
- Volatility — ATR breakouts, Bollinger squeeze, VIX correlation
- Specialized ML & Advanced — HMM, decision trees, isolation forest, Hurst exponent

You Get

📂 Python `.py` scripts for direct Backtrader use
📄 PDF manuals detailing logic, parameters, and best practices
🎯 Get Instant Access — Download Today, Trade Tonight

👉 **Get it here**

# 8.1 Components and formulas

Keltner

- Midline $\mathrm{EMA}_n(\mathrm{Close})$
- Width $\mathrm{ATR}_m$
- Bands $\mathrm{Upper} = \mathrm{EMA} + k \cdot \mathrm{ATR},\ \mathrm{Lower} = \mathrm{EMA} - k \cdot \mathrm{ATR}$

RSI (Wilder)

- Scaled 0–100; you're using simple thresholds as confirmation (kept exactly).

Signal logic

- Long if Close pierces **Upper** and RSI is above `rsi_low`
- Short if Close pierces **Lower** and RSI is below `rsi_high`
- Exit via **ATR trailing stop** updated each bar

Backtrader execution model note

- Orders placed in `next()` execute **on the next bar** (unless you enable cheat-on-open). So using current-bar conditions is **not** look-ahead because fills occur next bar by default.

# 8.2 Step-by-step build with explanations

## 8.2.1 Wire up EMA, ATR, RSI

```
import backtrader as bt

class _KC_RSI_Part1(bt.Strategy):
    params = dict(ema_period=30, atr_period=7, rsi_period=14)

    def __init__(self):
        # Trend proxy and volatility
        self.ema = bt.ind.EMA(self.data.close, period=self.p.ema_period)
        self.atr = bt.ind.ATR(self.data, period=self.p.atr_period)

        # Momentum confirmation
        self.rsi = bt.ind.RSI(self.data.close, period=self.p.rsi_period)
```

Why

- EMA is your centerline; ATR scales expansion.
- RSI is a low-cost momentum sanity check.

## 8.2.2 Build Keltner bands from EMA and ATR

```
class _KC_RSI_Part2(bt.Strategy):
    params = dict(ema_period=30, atr_period=7, atr_multiplier=1.0,
rsi_period=14)

    def __init__(self):
        self.ema = bt.ind.EMA(self.data.close, period=self.p.ema_period)
        self.atr = bt.ind.ATR(self.data, period=self.p.atr_period)
        self.rsi = bt.ind.RSI(self.data.close, period=self.p.rsi_period)

        # Volatility-adaptive envelopes
        self.upper_band = self.ema + (self.atr * self.p.atr_multiplier)
        self.lower_band = self.ema - (self.atr * self.p.atr_multiplier)
```

Why

- Keltner responds to current volatility; higher ATR → wider bands → fewer, cleaner breaks.

## 8.2.3 Entry rules: breakout + RSI confirmation

```
class _KC_RSI_Part3(bt.Strategy):
    params = dict(ema_period=30, atr_period=7, atr_multiplier=1.0,
```

```
                    rsi_period=14, rsi_low=30, rsi_high=70)

    def __init__(self):
        self.ema = bt.ind.EMA(self.data.close, period=self.p.ema_period)
        self.atr = bt.ind.ATR(self.data, period=self.p.atr_period)
        self.rsi = bt.ind.RSI(self.data.close, period=self.p.rsi_period)
        self.upper_band = self.ema + (self.atr * self.p.atr_multiplier)
        self.lower_band = self.ema - (self.atr * self.p.atr_multiplier)
        self.order = None

    def next(self):
        if self.order:
            return

        if not self.position:
            long_break  = self.data.close[0] > self.upper_band[0]
            short_break = self.data.close[0] < self.lower_band[0]

            # Your confirmation rules (kept exactly):
            long_ok  = self.rsi[0] > self.p.rsi_low
            short_ok = self.rsi[0] < self.p.rsi_high

            if long_break and long_ok:
                self.order = self.buy()
            elif short_break and short_ok:
                self.order = self.sell()
```

Why

- Breakout proves range expansion; RSI simply prevents the weakest extremes (per your thresholds).
- Optional refinement (comment only): many traders also require RSI slope confirmation, e.g. `self.rsi[0] > self.rsi[-1]` for longs.

## 8.2.4 Manual ATR trailing stop (long and short)

```
class _KC_RSI_Part4(bt.Strategy):
    params = dict(atr_period=7, trailing_stop_atr=1.0)

    def __init__(self):
        self.atr = bt.ind.ATR(self.data, period=self.p.atr_period)
        self.trailing_stop_price = None
        self.order = None

    def next(self):
```

```
        if self.order:
            return

        if self.position:
            if self.position.size > 0:
                # For longs: trail below price by ATR×mult
                new_stop = self.data.close[0] - (self.atr[0] *
self.p.trailing_stop_atr)
                if self.trailing_stop_price is None or new_stop >
self.trailing_stop_price:
                    self.trailing_stop_price = new_stop
                if self.data.close[0] <= self.trailing_stop_price:
                    self.order = self.close()
                    self.trailing_stop_price = None
            else:
                # For shorts: trail above price by ATR×mult
                new_stop = self.data.close[0] + (self.atr[0] *
self.p.trailing_stop_atr)
                if self.trailing_stop_price is None or new_stop <
self.trailing_stop_price:
                    self.trailing_stop_price = new_stop
                if self.data.close[0] >= self.trailing_stop_price:
                    self.order = self.close()
                    self.trailing_stop_price = None
```

Why

- Manual trailing lets you tune the update rule precisely (e.g., you're basing it on Close).
- If you prefer broker-managed trailing, you could use `StopTrail`, but your manual approach is fine and easy to reason about.

## 8.2.5 Clear the order lock when fills resolve

```
class _KC_RSI_Part5(bt.Strategy):
    def __init__(self):
        self.order = None
    def notify_order(self, order):
        if order.status in [order.Completed, order.Canceled, order.Rejected]:
            self.order = None
```

Why

- Standard Backtrader pattern: never stack orders while one is pending.

# 8.3 Complete Backtrader strategy (your exact class)

```
import backtrader as bt

class KeltnerChannelRSIBreakoutStrategy(bt.Strategy):
    params = (
        ('ema_period', 30),
        ('atr_period', 7),
        ('atr_multiplier', 1.),
        ('rsi_period', 14),
        ('rsi_low', 30),
        ('rsi_high', 70),
        ('trailing_stop_atr', 1.0),
    )

    def __init__(self):
        self.ema = bt.indicators.EMA(self.data.close,
period=self.params.ema_period)
        self.atr = bt.indicators.ATR(self.data, period=self.params.atr_period)
        self.rsi = bt.indicators.RSI(self.data.close,
period=self.params.rsi_period)

        self.upper_band = self.ema + (self.atr * self.params.atr_multiplier)
        self.lower_band = self.ema - (self.atr * self.params.atr_multiplier)

        self.order = None
        self.trailing_stop_price = None

    def next(self):
        if self.order:
            return

        if not self.position:
            # Long: price above upper band + RSI confirmation
            if (self.data.close[0] > self.upper_band[0] and
                self.rsi[0] > self.params.rsi_low):
                self.order = self.buy()

            # Short: price below lower band + RSI confirmation
            elif (self.data.close[0] < self.lower_band[0] and
                  self.rsi[0] < self.params.rsi_high):
                self.order = self.sell()
        else:
            # Update trailing stop
            if self.position.size > 0:  # Long position
```

```
                new_stop = self.data.close[0] - (self.atr[0] *
self.params.trailing_stop_atr)
                if self.trailing_stop_price is None or new_stop >
self.trailing_stop_price:
                    self.trailing_stop_price = new_stop
                if self.data.close[0] <= self.trailing_stop_price:
                    self.order = self.close()
                    self.trailing_stop_price = None

            elif self.position.size < 0:  # Short position
                new_stop = self.data.close[0] + (self.atr[0] *
self.params.trailing_stop_atr)
                if self.trailing_stop_price is None or new_stop <
self.trailing_stop_price:
                    self.trailing_stop_price = new_stop
                if self.data.close[0] >= self.trailing_stop_price:
                    self.order = self.close()
                    self.trailing_stop_price = None

    def notify_order(self, order):
        if order.status in [order.Completed, order.Canceled, order.Rejected]:
            self.order = None
```

# 8.4 Minimal runner

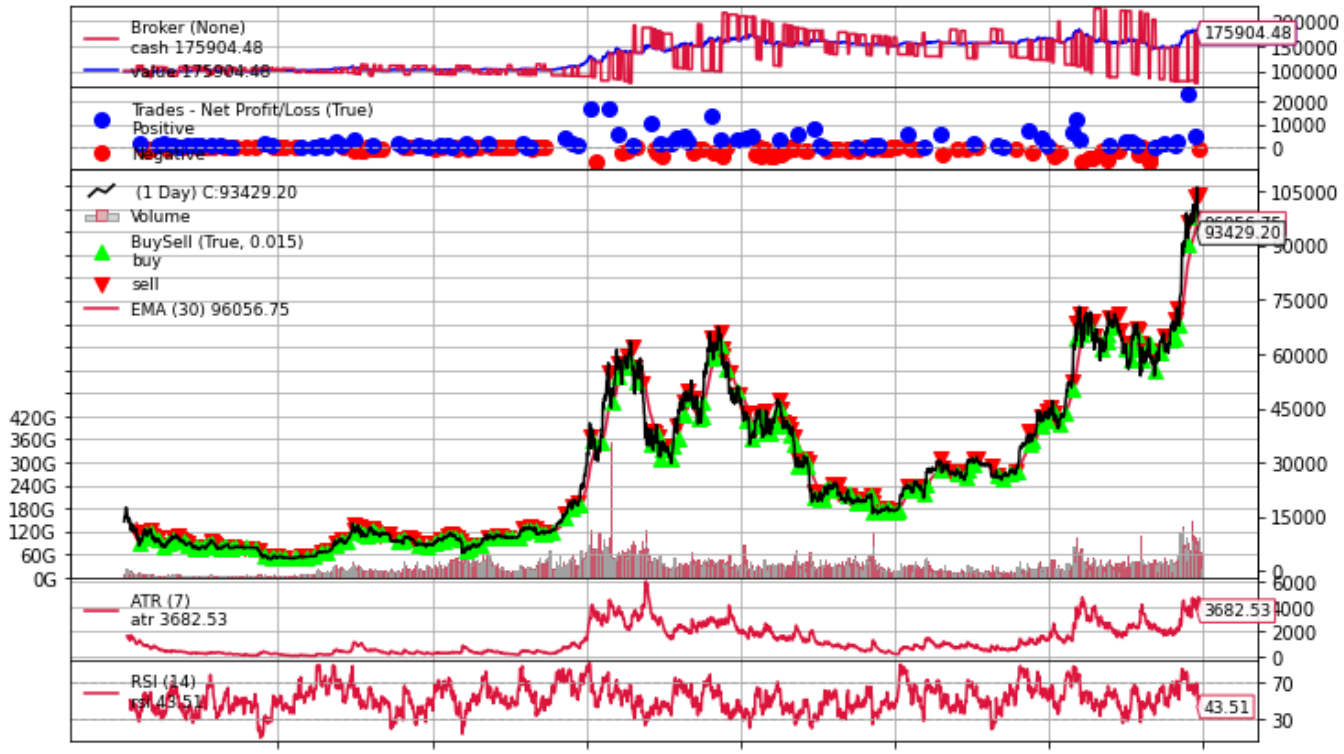
```
import backtrader as bt
import yfinance as yf
import pandas as pd
import numpy as np
import dateutil.relativedelta as rd
import matplotlib.pyplot as plt
%matplotlib inline
import seaborn as sns
import importlib

def load_strategy(class_name):
    module = importlib.import_module(class_name)
    strategy = getattr(module, class_name)
    return strategy

ticker = "BTC-USD"
start = "2018-01-01"
end = "2025-01-01"

start_dt = pd.to_datetime(start)
```

```
end_dt = pd.to_datetime(end)


strategy = load_strategy("KeltnerChannelRSIBreakoutStrategy")

data = yf.download(ticker, start=start, end=end, progress=False)


data = data.droplevel(1, 1) if isinstance(data.columns, pd.MultiIndex) else
data

feed = bt.feeds.PandasData(dataname=data)
cerebro = bt.Cerebro()
cerebro.addstrategy(strategy)
cerebro.adddata(feed)
cerebro.broker.setcash(100000)
cerebro.broker.setcommission(commission=0.001)
# cerebro.addsizer(bt.sizers.PercentSizer, percents=95)

cerebro.addanalyzer(bt.analyzers.SharpeRatio, _name='sharpe',
timeframe=bt.TimeFrame.Days, riskfreerate=0.0)
cerebro.addanalyzer(bt.analyzers.DrawDown,    _name='dd')
cerebro.addanalyzer(bt.analyzers.Returns,     _name='rets')
cerebro.addanalyzer(bt.analyzers.TradeAnalyzer, _name='trades')

start_val = cerebro.broker.getvalue()
results = cerebro.run()
strat = results[0]
final_val = cerebro.broker.getvalue()
ret = (final_val - start_val) / start_val * 100


print(f"Return: {ret:.2f}% | Final Value: {final_val:.2f}")



sh = strat.analyzers.sharpe.get_analysis().get('sharperatio', None)
dd = strat.analyzers.dd.get_analysis()
rets = strat.analyzers.rets.get_analysis()
tr = strat.analyzers.trades.get_analysis()

print(f"Sharpe: {sh}")
print(f"Max DD: {dd.get('max', {}).get('drawdown', 0):.2f}%  Len:
{dd.get('max', {}).get('len', 0)}")
print(f"CAGR:   {rets.get('rnorm100', 0):.2f}%  Total: {rets.get('rtot',
0)*100:.2f}%")
```

```
print(f"Trades: {tr.get('total', {}).get('total', 0)}  Won: {tr.get('won',
{}).get('total', 0)}  Lost: {tr.get('lost', {}).get('total', 0)}")

plt.rcParams['figure.figsize'] = [10, 6]
plt.rcParams['font.size'] = 10
cerebro.plot(iplot=False)
```

Optional refinements:

- For **tighter confirmation**, many traders also require RSI to be **rising** for longs ( `self.rsi[0] > self.rsi[-1]` ) and **falling** for shorts.
- To avoid **same-bar re-entries** after immediate exits, reset `self.trailing_stop_price = None` on every new entry in `notify_order` (after a completed fill).



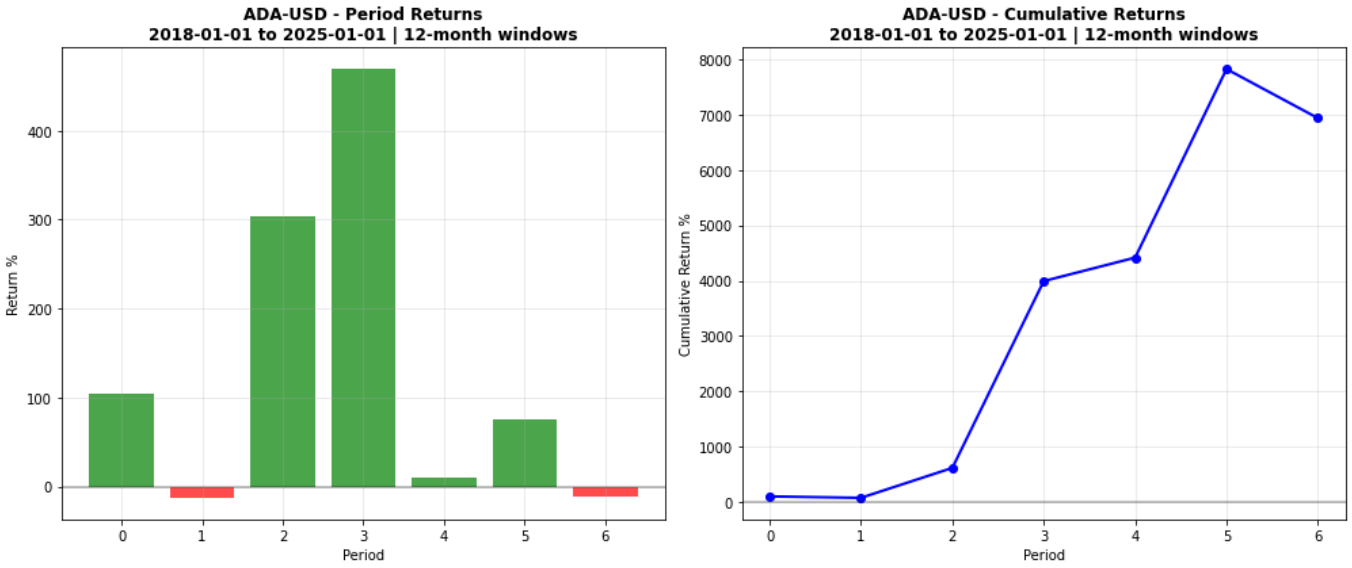

# 8.5 Rolling Backtest

Again for a better analysis of the strategy's performance over time let's try a rolling backtest.

```
strategy = load_strategy("KeltnerChannelRSIBreakoutStrategy")

    ticker = "ADA-USD"
    start = "2018-01-01"
    end = "2025-01-01"
    window_months = 12
```

```
    df = run_rolling_backtest(ticker=ticker, start=start, end=end,
window_months=window_months)
```

```
=== ROLLING BACKTEST RESULTS ===
        start         end  return_pct    final_value
0  2018-01-01  2019-01-01  104.134899  204134.898511
1  2019-01-01  2020-01-01  -12.749410   87250.589894
2  2020-01-01  2021-01-01  303.583149  403583.149018
3  2021-01-01  2022-01-01  469.340567  569340.567316
4  2022-01-01  2023-01-01   10.358072  110358.071974
5  2023-01-01  2024-01-01   75.401604  175401.603631
6  2024-01-01  2025-01-01  -11.184511   88815.489427

=== ROLLING BACKTEST STATISTICS ===
Mean Return %: 134.13
Median Return %: 75.40
Std Dev %: 170.48
Min Return %: -12.75
Max Return %: 469.34
Sharpe Ratio: 0.79
```




The Keltner–RSI Breakout strategy's rolling backtest shows **high upside bursts** with a few standout years (e.g., 2021 hitting +469%), but also **occasional drawdowns** like -12.7% and -11.18%.

On average, returns are strong (**134% mean, 75% median**), but the high standard deviation (**170%**) reflects the strategy's exposure to volatility.
The **Sharpe ratio of 0.79** suggests decent risk-adjusted performance, driven largely by big trend years offsetting quieter or losing periods.

# Chapter 9 — ML-Enhanced ADX Strategy - Random Forest + Technical Gate

**Idea**
Blend a classic **ADX trend filter** (+DI/−DI direction) with a lightweight **machine-learning classifier** that reads a compact feature set (returns, DI spread, ATR%, BB width/position, RSI, volume ratio, MA gaps, recent range). Trades only when **both**: (1) ADX regime is trending **and** (2) ML predicts the move with enough confidence.

**Risk**
Use ATR-scaled trailing stops managed by the broker ( `StopTrail` ) so the exit follows price automatically.

This strategy is part of the **Mega Backtrader Strategy Pack** — a collection of over **250 ready-to-use strategies** for your algorithmic trading.

The **Mega Backtrader Strategy Pack** gives you **250+ fully-tested Python strategies** — adaptive filters, mean reversion, momentum, volatility breakouts, machine learning models & more — all ready to deploy in Backtrader.

**Why Traders Love This Pack**

- Instant Impact — Deploy strategies and start testing today
- Zero Guesswork — Every strategy is documented, tested, and ready-to-run
- Future-Proof — Lifetime updates included
- Proven Variety — Adaptive, mean-reversion, momentum, volatility, and ML-driven models

What's Inside
✅ **250+ Strategies across multiple categories:**

- Adaptive Filters — Kalman, Hilbert, TRIX, Guppy MMA
- Mean Reversion — Bollinger, Keltner, pair trading, RSI/Stoch
- Momentum & Trend — MACD, ADX, Donchian, MA ribbons
- Volatility — ATR breakouts, Bollinger squeeze, VIX correlation
- Specialized ML & Advanced — HMM, decision trees, isolation forest, Hurst exponent

You Get

📂 Python `.py` scripts for direct Backtrader use

📄 PDF manuals detailing logic, parameters, and best practices

🎯 Get Instant Access — Download Today, Trade Tonight

👉 **Get it here**

---

# 9.1 Indicators and gating (why this combo)

- **ADX & DI (+DI / −DI)**: gate trades to trending regimes; direction from DI dominance.
- **RSI**: “sentiment/breadth” proxy; also used as a raw feature.
- **Bollinger Bands (mid/width/position)**: volatility state and where price sits inside the band.
- **ATR**: risk and volatility magnitude (as % of price).
- **SMA short/long & Volume SMA**: basic trend vs. noise; volume ratio for participation.

The ADX gate reduces false ML triggers in chop; the ML layer helps **rank** which trend signals to take.

---

# 9.2 Feature engineering (what each input conveys)

We assemble one feature vector per bar:

- **Trend strength/direction**: `ADX`, `+DI`, `−DI`, `DI spread = +DI − −DI`.
- **Returns**: 5-day and 10-day pct returns (momentum window).
- **MA gaps**: `Close / SMA(10) − 1`, `Close / SMA(30) − 1` (distance from trend).
- **Volatility size**: `ATR / Close * 100` (% of price).
- **BB width**: `(BB.top − BB.bot)/BB.mid * 100` (state tight/wide).
- **BB position**: `(Close − BB.mid)/(BB.top − BB.bot)` (where we sit inside the band).
- **Volume ratio**: `Volume / SMA(20)` (participation).
- **RSI**, and `50 − RSI` (distance from neutral from the other side).
- **Recent realized range** (last ~5 bars): `(max(Highs) − min(Lows)) / Close * 100`.

> Tip: Most features are **scale-dependent** → we standardize with `StandardScaler` before training.

---

# 9.3 Targets and training cadence (how labels are formed)

- The code labels with a **3-class** scheme: `+1` up, `-1` down, `0` flat based on future % move (thresholds ±1%).
- Retrains every `retrain_freq` bars (e.g., 30): rolling online learning feel with a fresh Random Forest.
- Uses **probability** and only acts on **high-confidence** predictions (max class prob > 0.6).

> Important note about **forward labels** in backtests: you cannot "see" the future bar inside `next()`. A common pattern is to **enqueue features now** and **append the label only after N bars have passed** (see optional snippet below). Your class keeps it simple; I'll keep your code unchanged in the full listing and show the safer alignment pattern as a commented alternative.

---

# 9.4 Risk: ATR trailing stop logic

- On entry (once a position exists), we arm a broker-managed **StopTrail** using `trailamount = ATR × atr_multiplier`.
- The order follows favorable price; if price retraces by the trail amount, we exit.

---

# 9.5 Step-by-step build with explanations

## 9.5.1 Wire up indicators and DI/ADX gate

```
import backtrader as bt

class _MLADX_Part1(bt.Strategy):
    params = dict(adx_period=14, rsi_period=14, bb_period=7, atr_period=14)

    def __init__(self):
        d = self.datas[0]
        self.close = d.close; self.high = d.high; self.low = d.low; self.vol =
d.volume

        # Trend & direction
        self.adx    = bt.ind.ADX(d, period=self.p.adx_period)
        self.plusdi = bt.ind.PlusDI(d, period=self.p.adx_period)
        self.minusdi= bt.ind.MinusDI(d, period=self.p.adx_period)
```

```
        # Vol & sentiment
        self.atr = bt.ind.ATR(d, period=self.p.atr_period)
        self.bb  = bt.ind.BollingerBands(self.close, period=self.p.bb_period)
        self.rsi = bt.ind.RSI(self.close, period=self.p.rsi_period)

        # Trend baselines / volume baseline
        self.sma_short = bt.ind.SMA(self.close, period=10)
        self.sma_long  = bt.ind.SMA(self.close, period=30)
        self.vsma      = bt.ind.SMA(self.vol, period=20)
```

**Why**: These become both **filters** and **features**. ADX + DI give the “green light”; others paint the market state for ML.

---

## 9.5.2 Feature vector (one row per bar)

```
import numpy as np

class _MLADX_Part2(_MLADX_Part1):
    params = dict(lookback=7)

    def get_features(self):
        if len(self) < self.p.lookback:
            return None

        feats = []
        # ADX block
        feats += [ self.adx[0], self.plusdi[0], self.minusdi[0],
self.plusdi[0] - self.minusdi[0] ]
        # Returns / trend distance
        feats += [
            (self.close[0] - self.close[-5]) / self.close[-5] * 100,
            (self.close[0] - self.close[-10]) / self.close[-10] * 100,
            self.close[0] / self.sma_short[0] - 1,
            self.close[0] / self.sma_long[0]  - 1
        ]
        # Volatility and BB
        feats += [
            self.atr[0] / self.close[0] * 100,
            (self.bb.top[0] - self.bb.bot[0]) / self.bb.mid[0] * 100,
            (self.close[0] - self.bb.mid[0]) / max(1e-9, (self.bb.top[0] -
self.bb.bot[0]))
        ]
```

```
        # Volume & RSI
        feats += [ self.vol[0] / max(1e-9, self.vsma[0]), self.rsi[0], 50 -
self.rsi[0] ]
        # Recent realized range
        n = min(5, len(self))
        highs = [self.high[-i] for i in range(n)]
        lows  = [self.low[-i]  for i in range(n)]
        feats += [ (max(highs) - min(lows)) / self.close[0] * 100 ]

        return np.array(feats).reshape(1, -1)
```

**Why**: Every term encodes a **dimension** of state (trend, speed, stretch, vol, participation). Note the `max(1e-9, …)` guards to avoid divide-by-zero.

---

## 9.5.3 Targets (3-class) and safe alignment (optional pattern)

Your code computes a label based on a "future return." A robust pattern is to **buffer** feature rows and assign the label **N bars later**, once the outcome is known.

```
# OPTIONAL: safer label alignment pattern (keep your class as-is if you
prefer)
from collections import deque

class _MLADX_LabelBuffer(_MLADX_Part2):
    params = dict(future_bars=5)
    def __init__(self):
        super().__init__()
        self.feature_q = deque()   # store (index, features, entry_price)
        self.X, self.y = [], []

    def next(self):
        feats = self.get_features()
        if feats is not None:
            self.feature_q.append((len(self), feats, float(self.close[0])))

        # When enough bars have elapsed, pop and create label
        while self.feature_q and len(self) - self.feature_q[0][0] >=
self.p.future_bars:
            _, f, price_then = self.feature_q.popleft()
            fwd_ret = (self.close[0] - price_then)/price_then * 100
            label = 1 if fwd_ret > 1.0 else (-1 if fwd_ret < -1.0 else 0)
            self.X.append(f[0]); self.y.append(label)
```

**Why**: Prevents accidental look-ahead and aligns “feature at t” with “outcome by t+N”.

---

## 9.5.4 Train the model on a rolling basis

```
from sklearn.ensemble import RandomForestClassifier
from sklearn.preprocessing import StandardScaler

class _MLADX_Part3(_MLADX_Part2):
    params = dict(retrain_freq=30)
    def __init__(self):
        super().__init__()
        self.model = None
        self.scaler = StandardScaler()
        self.feature_history, self.target_history = [], []
        self.model_trained = False
        self.last_retrain = 0

    def train_model(self):
        if len(self.feature_history) < 50:
            return False
        X = np.vstack(self.feature_history)
        y = np.array(self.target_history)
        Xs = self.scaler.fit_transform(X)
        self.model = RandomForestClassifier(
            n_estimators=50, max_depth=10, min_samples_split=5,
random_state=42
        ).fit(Xs, y)
        self.model_trained = True
        return True
```

**Why**: 50+ samples is a minimal baseline for a 3-class RF. Standardization helps distance-based splits; it certainly helps many other models.

---

## 9.5.5 Get ML prediction with confidence gate

```
class _MLADX_Part4(_MLADX_Part3):
    def get_ml_signal(self):
        if not self.model_trained:
            return 0
        feats = self.get_features()
```

```
        if feats is None:
            return 0
        Xs = self.scaler.transform(feats)
        pred = self.model.predict(Xs)[0]
        proba = self.model.predict_proba(Xs)[0]
        if max(proba) > 0.6:
            return int(pred)
        return 0
```

**Why**: Only act on higher-confidence calls. Low-confidence ML is worse than no ML.

---

## 9.5.6 Order logic: ADX gate × ML signal, plus ATR StopTrail

```
class _MLADX_Part5(_MLADX_Part4):
    params = dict(adx_threshold=20, atr_multiplier=3.0, retrain_freq=30)

    def __init__(self):
        super().__init__()
        self.order = None
        self.trail_order = None

    def cancel_trail(self):
        if self.trail_order:
            self.cancel(self.trail_order); self.trail_order = None

    def next(self):
        if self.order:
            return

        # Manage trailing stop if in position
        if self.position:
            if not self.trail_order:
                if self.position.size > 0:
                    self.trail_order = self.sell(exectype=bt.Order.StopTrail,

trailamount=self.atr[0]*self.p.atr_multiplier)
                else:
                    self.trail_order = self.buy(exectype=bt.Order.StopTrail,

trailamount=self.atr[0]*self.p.atr_multiplier)
            return

        # Need enough history for features
```

```
        if len(self) < self.p.lookback:
            return

        # Collect feature/label samples for later training
        feats = self.get_features()
        if feats is not None and len(self) > 10:
            # This mirrors your simple buffer; see optional alignment above
for stricter approach
            self.feature_history.append(feats[0])
            self.target_history.append(0)  # placeholder, or your get_target()

            if len(self.feature_history) > 500:
                self.feature_history = self.feature_history[-400:]
                self.target_history = self.target_history[-400:]

        # Periodic retrain
        if (len(self) - self.last_retrain) > self.p.retrain_freq:
            if self.train_model():
                self.last_retrain = len(self)

        # ADX gate
        if self.adx[0] < self.p.adx_threshold:
            return

        # Direction from DI; decision from ML
        adx_long  = self.plusdi[0] > self.minusdi[0]
        adx_short = self.minusdi[0] > self.plusdi[0]
        ml = self.get_ml_signal()

        if adx_long and ml == 1:
            self.cancel_trail(); self.order = self.buy()
        elif adx_short and ml == -1:
            self.cancel_trail(); self.order = self.sell()
```

**Why**: Two keys must turn: **trend gate** and **ML classification**. Trailing stop arms only **after** a position exists.

---

# 9.6 Full strategy class (your code unchanged)

```
import backtrader as bt
import datetime
import yfinance as yf
import pandas as pd
```

```
import matplotlib.pyplot as plt
import numpy as np
from sklearn.ensemble import RandomForestClassifier
from sklearn.preprocessing import StandardScaler
import warnings
warnings.filterwarnings('ignore')

class MLEnhancedADXStrategy(bt.Strategy):
    params = (
        ('adx_period', 14),          # ADX period
        ('adx_threshold', 20),       # ADX threshold for trend strength
        ('rsi_period', 14),          # RSI period (sentiment proxy)
        ('bb_period', 7),           # Bollinger Bands period (volatility)
        ('atr_period', 14),          # ATR period
        ('lookback', 7),            # Lookback period for ML features
        ('retrain_freq', 30),        # Retrain model every N bars
        ('atr_multiplier', 3.0),     # ATR multiplier for trailing stops
        ('printlog', False),
    )

    def log(self, txt, dt=None, doprint=False):
        if self.params.printlog or doprint:
            dt = dt or self.datas[0].datetime.datetime(0)
            print(f"{dt.isoformat()} - {txt}")

    def __init__(self):
        self.dataclose = self.datas[0].close
        self.datahigh = self.datas[0].high
        self.datalow = self.datas[0].low
        self.datavolume = self.datas[0].volume

        # Traditional indicators
        self.adx = bt.indicators.ADX(self.datas[0],
period=self.params.adx_period)
        self.plusdi = bt.indicators.PlusDI(self.datas[0],
period=self.params.adx_period)
        self.minusdi = bt.indicators.MinusDI(self.datas[0],
period=self.params.adx_period)

        # Sentiment and volatility indicators
        self.rsi = bt.indicators.RSI(self.dataclose,
period=self.params.rsi_period)
        self.bb = bt.indicators.BollingerBands(self.dataclose,
period=self.params.bb_period)
        self.atr = bt.indicators.ATR(self.datas[0],
period=self.params.atr_period)
```

```
        # Additional features
        self.sma_short = bt.indicators.SMA(self.dataclose, period=10)
        self.sma_long = bt.indicators.SMA(self.dataclose, period=30)
        self.volume_sma = bt.indicators.SMA(self.datavolume, period=20)

        # ML components
        self.model = None
        self.scaler = StandardScaler()
        self.feature_history = []
        self.target_history = []
        self.model_trained = False
        self.last_retrain = 0

        # Track orders
        self.order = None
        self.trail_order = None

    def notify_order(self, order):
        if order.status in [order.Submitted, order.Accepted]:
            return

        if order.status in [order.Completed]:
            if order.isbuy():
                self.log(f"BUY EXECUTED at {order.executed.price:.2f}")
            elif order.issell():
                self.log(f"SELL EXECUTED at {order.executed.price:.2f}")
        elif order.status in [order.Canceled, order.Margin, order.Rejected]:
            self.log(f"Order Canceled/Margin/Rejected:
{order.getstatusname()}")

        if order == self.order:
            self.order = None
        if order == self.trail_order:
            self.trail_order = None

    def notify_trade(self, trade):
        if not trade.isclosed:
            return
        self.log(f"Trade Profit: GROSS {trade.pnl:.2f}, NET
{trade.pnlcomm:.2f}")

    def cancel_trail(self):
        if self.trail_order:
            self.cancel(self.trail_order)
            self.trail_order = None
```

```
    def get_features(self):
        """Extract features for ML model"""
        if len(self) < self.params.lookback:
            return None

        features = []

        # Traditional ADX features
        features.append(self.adx[0])
        features.append(self.plusdi[0])
        features.append(self.minusdi[0])
        features.append(self.plusdi[0] - self.minusdi[0])  # DI spread

        # Price action features
        features.append((self.dataclose[0] - self.dataclose[-5]) /
self.dataclose[-5] * 100)  # 5-day return
        features.append((self.dataclose[0] - self.dataclose[-10]) /
self.dataclose[-10] * 100)  # 10-day return
        features.append(self.dataclose[0] / self.sma_short[0] - 1)  # Price vs
short MA
        features.append(self.dataclose[0] / self.sma_long[0] - 1)   # Price vs
long MA

        # Volatility features (ML-based volatility forecasting)
        features.append(self.atr[0] / self.dataclose[0] * 100)  # ATR as % of
price
        bb_width = (self.bb.top[0] - self.bb.bot[0]) / self.bb.mid[0] * 100
        features.append(bb_width)  # BB width
        features.append((self.dataclose[0] - self.bb.mid[0]) / (self.bb.top[0]
- self.bb.bot[0]))  # BB position

        # Volume features (sentiment proxy)
        features.append(self.datavolume[0] / self.volume_sma[0])  # Volume
ratio

        # Sentiment indicators (RSI as VIX proxy)
        features.append(self.rsi[0])
        features.append(50 - self.rsi[0])  # RSI divergence from neutral

        # Recent volatility
        recent_highs = [self.datahigh[-i] for i in range(min(5, len(self)))]
        recent_lows = [self.datalow[-i] for i in range(min(5, len(self)))]
        if len(recent_highs) > 1:
            volatility = (max(recent_highs) - min(recent_lows)) /
self.dataclose[0] * 100
```

```
            features.append(volatility)
        else:
            features.append(0)

        return np.array(features).reshape(1, -1)

    def get_target(self, future_bars=5):
        """Get target for training (future return)"""
        if len(self) < future_bars:
            return 0

        current_price = self.dataclose[0]
        future_price = self.dataclose[-future_bars] if len(self) > future_bars
else current_price
        return_pct = (future_price - current_price) / current_price * 100

        # Convert to classification: 1 for up, -1 for down, 0 for sideways
        if return_pct > 1.0:
            return 1
        elif return_pct < -1.0:
            return -1
        else:
            return 0

    def train_model(self):
        """Train the ML model"""
        if len(self.feature_history) < 50:  # Need minimum data
            return False

        try:
            X = np.vstack(self.feature_history)
            y = np.array(self.target_history)

            # Scale features
            X_scaled = self.scaler.fit_transform(X)

            # Train Random Forest
            self.model = RandomForestClassifier(
                n_estimators=50,
                max_depth=10,
                random_state=42,
                min_samples_split=5
            )
            self.model.fit(X_scaled, y)
            self.model_trained = True
            self.log(f"ML Model trained with {len(X)} samples")
```

```
            return True

        except Exception as e:
            self.log(f"Model training error: {e}")
            return False

    def get_ml_signal(self):
        """Get ML prediction"""
        if not self.model_trained:
            return 0

        features = self.get_features()
        if features is None:
            return 0

        try:
            features_scaled = self.scaler.transform(features)
            prediction = self.model.predict(features_scaled)[0]

            # Get prediction probability for confidence
            proba = self.model.predict_proba(features_scaled)[0]
            confidence = max(proba)

            # Only use high-confidence predictions
            if confidence > 0.6:
                return prediction
            else:
                return 0

        except Exception as e:
            self.log(f"Prediction error: {e}")
            return 0

    def next(self):
        # Skip if order is pending
        if self.order:
            return

        # Handle trailing stops for existing positions
        if self.position:
            if not self.trail_order:
                if self.position.size > 0:
                    self.trail_order = self.sell(
                        exectype=bt.Order.StopTrail,
                        trailamount=self.atr[0] * self.params.atr_multiplier)
                elif self.position.size < 0:
```

```
                    self.trail_order = self.buy(
                        exectype=bt.Order.StopTrail,
                        trailamount=self.atr[0] * self.params.atr_multiplier)
            return

        # Ensure sufficient data
        if len(self) < self.params.lookback:
            return

        # Collect features for ML
        features = self.get_features()
        if features is not None and len(self) > 10:  # Need some history for
target
            target = self.get_target()
            self.feature_history.append(features[0])
            self.target_history.append(target)

            # Keep only recent history
            if len(self.feature_history) > 500:
                self.feature_history = self.feature_history[-400:]
                self.target_history = self.target_history[-400:]

        # Retrain model periodically
        if (len(self) - self.last_retrain) > self.params.retrain_freq:
            if self.train_model():
                self.last_retrain = len(self)

        # Traditional ADX signal
        if self.adx[0] < self.params.adx_threshold:
            return

        # Traditional directional signals
        adx_long = self.plusdi[0] > self.minusdi[0]
        adx_short = self.minusdi[0] > self.plusdi[0]

        # Get ML ensemble signal
        ml_signal = self.get_ml_signal()

        # Combine traditional and ML signals
        long_signal = adx_long and ml_signal == 1
        short_signal = adx_short and ml_signal == -1

        if long_signal:
            self.log(f"ML ENHANCED LONG signal at {self.dataclose[0]:.2f}")
            self.log(f"ADX: {self.adx[0]:.2f}, +DI: {self.plusdi[0]:.2f}, ML:
{ml_signal}")
```

```
            self.cancel_trail()
            if self.position and self.position.size < 0:
                self.order = self.buy()
            elif not self.position:
                self.order = self.buy()

        elif short_signal:
            self.log(f"ML ENHANCED SHORT signal at {self.dataclose[0]:.2f}")
            self.log(f"ADX: {self.adx[0]:.2f}, -DI: {self.minusdi[0]:.2f}, ML:
{ml_signal}")

            self.cancel_trail()
            if self.position and self.position.size > 0:
                self.order = self.sell()
            elif not self.position:
                self.order = self.sell()

    def stop(self):
        self.log(f"Ending Portfolio Value: {self.broker.getvalue():.2f}",
doprint=True)
        if self.model_trained:
            self.log(f"Model trained on {len(self.feature_history)} samples",
doprint=True)
```

---

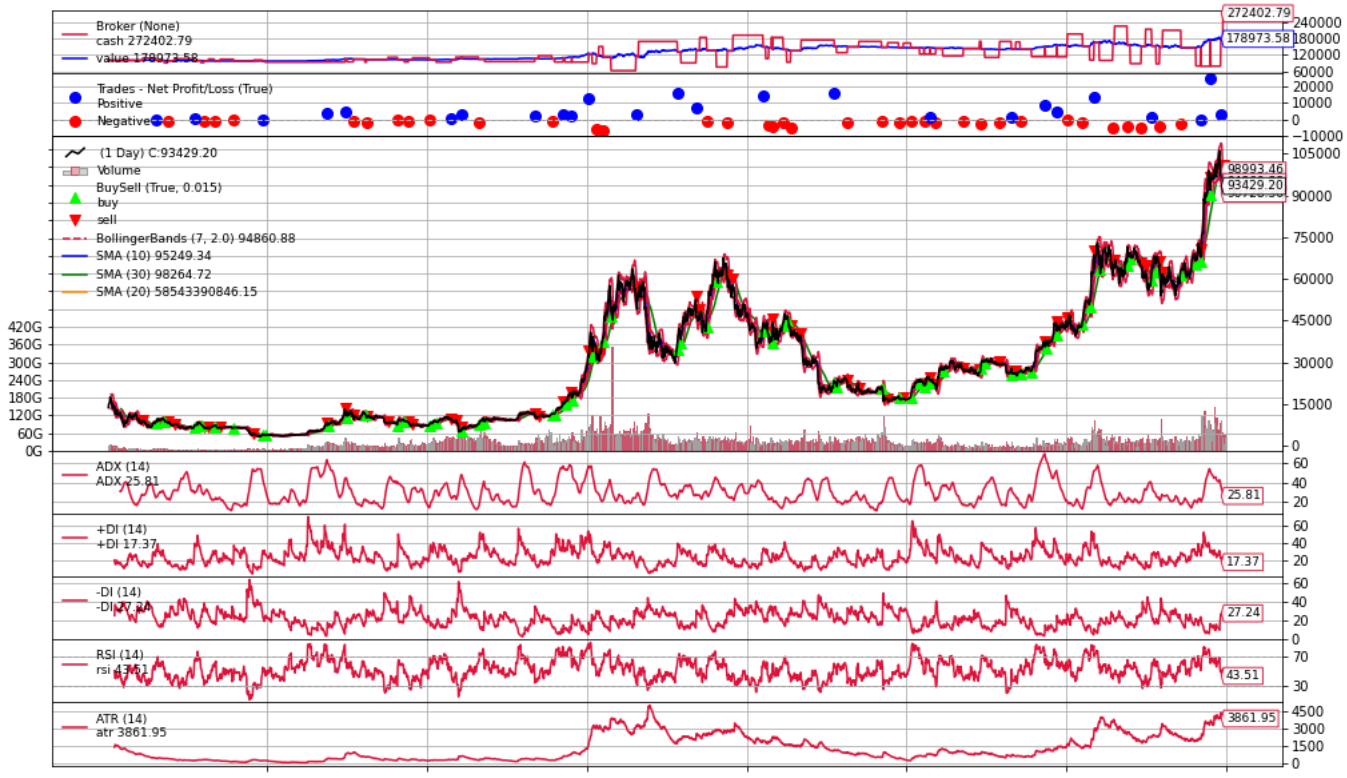
# 9.7 Backtesting

```
import backtrader as bt
import yfinance as yf
import pandas as pd
import numpy as np
import dateutil.relativedelta as rd
import matplotlib.pyplot as plt
%matplotlib inline
import seaborn as sns
import importlib

def load_strategy(class_name):
    module = importlib.import_module(class_name)
    strategy = getattr(module, class_name)
    return strategy

ticker = "BTC-USD"
```

```
start = "2018-01-01"
end = "2025-01-01"

start_dt = pd.to_datetime(start)
end_dt = pd.to_datetime(end)


strategy = load_strategy("MLEnhancedADXStrategy")

data = yf.download(ticker, start=start, end=end, progress=False)


data = data.droplevel(1, 1) if isinstance(data.columns, pd.MultiIndex) else
data

feed = bt.feeds.PandasData(dataname=data)
cerebro = bt.Cerebro()
cerebro.addstrategy(strategy)
cerebro.adddata(feed)
cerebro.broker.setcash(100000)
cerebro.broker.setcommission(commission=0.001)
# cerebro.addsizer(bt.sizers.PercentSizer, percents=95)

cerebro.addanalyzer(bt.analyzers.SharpeRatio, _name='sharpe',
timeframe=bt.TimeFrame.Days, riskfreerate=0.0)
cerebro.addanalyzer(bt.analyzers.DrawDown,    _name='dd')
cerebro.addanalyzer(bt.analyzers.Returns,     _name='rets')
cerebro.addanalyzer(bt.analyzers.TradeAnalyzer, _name='trades')

start_val = cerebro.broker.getvalue()
results = cerebro.run()
strat = results[0]
final_val = cerebro.broker.getvalue()
ret = (final_val - start_val) / start_val * 100


print(f"Return: {ret:.2f}% | Final Value: {final_val:.2f}")



sh = strat.analyzers.sharpe.get_analysis().get('sharperatio', None)
dd = strat.analyzers.dd.get_analysis()
rets = strat.analyzers.rets.get_analysis()
tr = strat.analyzers.trades.get_analysis()

print(f"Sharpe: {sh}")
```

```
print(f"Max DD: {dd.get('max', {}).get('drawdown', 0):.2f}%  Len:
{dd.get('max', {}).get('len', 0)}")
print(f"CAGR:   {rets.get('rnorm100', 0):.2f}%  Total: {rets.get('rtot',
0)*100:.2f}%")
print(f"Trades: {tr.get('total', {}).get('total', 0)}  Won: {tr.get('won',
{}).get('total', 0)}  Lost: {tr.get('lost', {}).get('total', 0)}")

plt.rcParams['figure.figsize'] = [10, 6]
plt.rcParams['font.size'] = 10
cerebro.plot(iplot=False)
```

- If you see **feature scaling errors** (NaNs/infs), clamp denominators.
- If the classifier gets too many **zeros** (flat class), consider dropping the 0 class or lowering the ±1% threshold.
- For **faster warm-up**, start with a longer historical window so you hit the 50-sample minimum quickly.




# Chapter 10 — Momentum Ignition Strategy

This system hunts for **energy builds** during consolidation, then enters on a **statistical breakout in momentum** (ROC) aligned with the long-term trend. Risk is managed by a tight **ATR-based manual trailing stop**.

This strategy is part of the **Mega Backtrader Strategy Pack** — a collection of over **250 ready-to-use strategies** for your algorithmic trading.

The **Mega Backtrader Strategy Pack** gives you **250+ fully-tested Python strategies** — adaptive filters, mean reversion, momentum, volatility breakouts, machine learning models & more — all ready to deploy in Backtrader.

**Why Traders Love This Pack**

- Instant Impact — Deploy strategies and start testing today
- Zero Guesswork — Every strategy is documented, tested, and ready-to-run
- Future-Proof — Lifetime updates included
- Proven Variety — Adaptive, mean-reversion, momentum, volatility, and ML-driven models

What's Inside
✅ **250+ Strategies across multiple categories:**

- Adaptive Filters — Kalman, Hilbert, TRIX, Guppy MMA
- Mean Reversion — Bollinger, Keltner, pair trading, RSI/Stoch
- Momentum & Trend — MACD, ADX, Donchian, MA ribbons
- Volatility — ATR breakouts, Bollinger squeeze, VIX correlation
- Specialized ML & Advanced — HMM, decision trees, isolation forest, Hurst exponent

You Get
📂 Python `.py` scripts for direct Backtrader use
📄 PDF manuals detailing logic, parameters, and best practices
🎯 Get Instant Access — Download Today, Trade Tonight

👉 **Get it here**

# 10.1 Concepts and formulas

Consolidation filter (price quietude)
Let $C_t$ be close. Over window $n$,

$$\sigma_C(t) = \mathrm{StdDev}\big(C_{t-n+1..t}\big), \qquad \mathrm{NormStd}(t) = \frac{\sigma_C(t)}{C_t}$$

We call it "consolidating" when $\mathrm{NormStd}(t) < \theta_{\text{cons}}$.

Momentum ignition via ROC breakout

$$\mathrm{ROC}_n(t) = \frac{C_t - C_{t-n}}{C_{t-n}} \quad (\text{or } 100 \times \text{ROC for pct})$$

Put a moving average and standard deviation on ROC over window $m$:

$$\mu_{\mathrm{ROC}}(t) = \mathrm{MA}_m(\mathrm{ROC}), \qquad \sigma_{\mathrm{ROC}}(t) = \mathrm{StdDev}_m(\mathrm{ROC})$$

Breakout bands:

$$\mathrm{Upper} = \mu_{\mathrm{ROC}} + k\,\sigma_{\mathrm{ROC}}, \qquad \mathrm{Lower} = \mu_{\mathrm{ROC}} - k\,\sigma_{\mathrm{ROC}}$$

Go long if ROC > Upper; short if ROC < Lower.

Trend filter

$$\text{Uptrend if } C_t > \mathrm{SMA}_T(C), \quad \text{Downtrend if } C_t < \mathrm{SMA}_T(C)$$

Risk (manual trailing stop)
Long: keep $\mathrm{stop} = \max(\mathrm{stop}, \mathrm{HH} - m \cdot ATR)$.
Short: keep $\mathrm{stop} = \min(\mathrm{stop}, \mathrm{LL} + m \cdot ATR)$.

# 10.2 Step-by-step build with explanations

## 10.2.1 Wire indicators

```
import backtrader as bt

class _MI_Part1(bt.Strategy):
    params = dict(
        consolidation_period=30, consolidation_threshold=0.1,
        roc_period=7, roc_ma_period=30, roc_breakout_std=1.0,
        trend_period=30, atr_period=7
    )
    def __init__(self):
        d = self.datas[0]
        # Volatility of price (used as % of current price)
        self.price_stddev = bt.ind.StandardDeviation(d.close,
period=self.p.consolidation_period)
        # Momentum (rate of change)
        self.roc       = bt.ind.RateOfChange(d.close,
period=self.p.roc_period)
        # Baseline and dispersion of momentum itself
        self.roc_ma    = bt.ind.SMA(self.roc, period=self.p.roc_ma_period)
        self.roc_std   = bt.ind.StandardDeviation(self.roc,
period=self.p.roc_ma_period)
        # Macro trend
        self.trend_sma = bt.ind.SMA(d.close, period=self.p.trend_period)
        # Risk
        self.atr       = bt.ind.ATR(d, period=self.p.atr_period)
```

Why: we explicitly separate **quietude** (std of price) from **ignition** (ROC breakout) and align with a slow **trend SMA**.

## 10.2.2 Consolidation test (normalize std by price)

```
class _MI_Part2(_MI_Part1):
    def is_consolidating(self):
        # Guard tiny denominators
        close = float(self.datas[0].close[0]) or 1.0
        norm_std = float(self.price_stddev[0]) / close
        return norm_std < self.p.consolidation_threshold
```

Why: comparing std to current price makes the threshold dimensionless and portable across assets/timeframes.

## 10.2.3 ROC breakout bands and tests

```
class _MI_Part3(_MI_Part2):
    def roc_bands(self):
        mu   = float(self.roc_ma[0])
        sig  = float(self.roc_std[0])
        k    = self.p.roc_breakout_std
        return mu + k*sig, mu - k*sig

    def momentum_breakout(self):
        up, down = self.roc_bands()
        r = float(self.roc[0])
        return (r > up), (r < down)
```

Why: a **statistical** ignition (ROC exceeds its own recent distribution) is stronger than raw ROC > 0.

## 10.2.4 Trend alignment and entries

```
class _MI_Part4(_MI_Part3):
    def trend_state(self):
        close = float(self.datas[0].close[0]); base = float(self.trend_sma[0])
        return (close > base), (close < base)  # uptrend, downtrend

    def next(self):
        if self.position: return  # entries only from flat (your style)
        if not self.is_consolidating(): return
```

```
        mom_up, mom_down = self.momentum_breakout()
        uptrend, downtrend = self.trend_state()

        if uptrend and mom_up:
            self.buy()
        elif downtrend and mom_down:
            self.sell()
```

Why: require **all three**: quiet → ignition → aligned regime.

## 10.2.5 Manual ATR trailing stop

```
class _MI_Part5(_MI_Part4):
    params = dict(atr_stop_multiplier=1.0)
    def __init__(self):
        super().__init__()
        self.stop_price = None
        self.highest = None
        self.lowest  = None
        self.order = None

    def notify_order(self, order):
        if order.status in [order.Submitted, order.Accepted]: return
        if order.status in [order.Completed]:
            if self.position and self.stop_price is None:
                if order.isbuy():
                    self.highest = float(self.datas[0].high[0])
                    self.stop_price = self.highest - float(self.atr[0]) *
self.p.atr_stop_multiplier
                elif order.issell():
                    self.lowest = float(self.datas[0].low[0])
                    self.stop_price = self.lowest + float(self.atr[0]) *
self.p.atr_stop_multiplier
            elif not self.position:
                self.stop_price = self.highest = self.lowest = None
        self.order = None

    def next(self):
        if self.order: return
        # manage trailing when in position
        if self.position:
            if self.position.size > 0:
                self.highest = max(self.highest, float(self.datas[0].high[0]))
                new_stop = self.highest - float(self.atr[0]) *
self.p.atr_stop_multiplier
```

```
                self.stop_price = max(self.stop_price, new_stop)
                if float(self.datas[0].close[0]) < self.stop_price:
                    self.order = self.close()
            else:
                self.lowest = min(self.lowest, float(self.datas[0].low[0]))
                new_stop = self.lowest + float(self.atr[0]) *
self.p.atr_stop_multiplier
                self.stop_price = min(self.stop_price, new_stop)
                if float(self.datas[0].close[0]) > self.stop_price:
                    self.order = self.close()
        else:
            # entry block from Part4 can be merged here (kept separate above
for clarity)
            pass
```

Why: the stop **only ever tightens** in the favorable direction (monotone). You anchor to extreme since entry and offset by ATR×mult.

# 10.3 Complete Backtrader strategy (your code unchanged)

```
import backtrader as bt

class MomentumIgnitionStrategy(bt.Strategy):
    """
    Identifies periods of low price volatility and enters on a statistical
    breakout in the Rate of Change (ROC) momentum indicator, aligned
    with the long-term trend.
    """
    params = (
        # Volatility Filter
        ('consolidation_period', 30),
        ('consolidation_threshold', 0.1), # Max StdDev as % of price
        # Momentum Breakout
        ('roc_period', 7),
        ('roc_ma_period', 30),
        ('roc_breakout_std', 1.0), # ROC must exceed N StdDevs of its MA
        # Trend Filter
        ('trend_period', 30),
        # Risk Management
        ('atr_period', 7),
        ('atr_stop_multiplier', 1.0),
    )

    def __init__(self):
```

```
        self.order = None

        # --- Indicators ---
        self.price_stddev = bt.indicators.StandardDeviation(self.data.close,
period=self.p.consolidation_period)
        self.roc = bt.indicators.RateOfChange(self.data.close,
period=self.p.roc_period)
        self.roc_ma = bt.indicators.SimpleMovingAverage(self.roc,
period=self.p.roc_ma_period)
        self.roc_stddev = bt.indicators.StandardDeviation(self.roc,
period=self.p.roc_ma_period)
        self.trend_sma = bt.indicators.SimpleMovingAverage(self.data.close,
period=self.p.trend_period)
        self.atr = bt.indicators.AverageTrueRange(self.data,
period=self.p.atr_period)

        # --- Trailing Stop State ---
        self.stop_price = None
        self.highest_price_since_entry = None
        self.lowest_price_since_entry = None

    def notify_order(self, order):
        if order.status in [order.Submitted, order.Accepted]: return
        if order.status in [order.Completed]:
            if self.position and self.stop_price is None:
                if order.isbuy():
                    self.highest_price_since_entry = self.data.high[0]
                    self.stop_price = self.highest_price_since_entry -
(self.atr[0] * self.p.atr_stop_multiplier)
                elif order.issell():
                    self.lowest_price_since_entry = self.data.low[0]
                    self.stop_price = self.lowest_price_since_entry +
(self.atr[0] * self.p.atr_stop_multiplier)
            elif not self.position:
                self.stop_price = None; self.highest_price_since_entry = None;
self.lowest_price_since_entry = None
        self.order = None

    def next(self):
        if self.order: return

        if not self.position:
            # --- Filter Conditions ---
            # 1. Is the market consolidating (low price volatility)?
            is_consolidating = (self.price_stddev[0] / self.data.close[0]) <
self.p.consolidation_threshold
```

```
            # 2. Is the macro trend aligned?
            is_macro_uptrend = self.data.close[0] > self.trend_sma[0]
            is_macro_downtrend = self.data.close[0] < self.trend_sma[0]

            if is_consolidating:
                # 3. Has momentum "ignited" with a statistical breakout?
                roc_upper_band = self.roc_ma[0] + (self.roc_stddev[0] *
self.p.roc_breakout_std)
                roc_lower_band = self.roc_ma[0] - (self.roc_stddev[0] *
self.p.roc_breakout_std)

                is_mom_breakout_up = self.roc[0] > roc_upper_band
                is_mom_breakout_down = self.roc[0] < roc_lower_band

                # --- Entry Logic ---
                if is_macro_uptrend and is_mom_breakout_up:
                    self.order = self.buy()
                elif is_macro_downtrend and is_mom_breakout_down:
                    self.order = self.sell()

        elif self.position:
            # --- Manual ATR Trailing Stop Logic ---
            if self.position.size > 0: # Long
                self.highest_price_since_entry =
max(self.highest_price_since_entry, self.data.high[0])
                new_stop = self.highest_price_since_entry - (self.atr[0] *
self.p.atr_stop_multiplier)
                self.stop_price = max(self.stop_price, new_stop)
                if self.data.close[0] < self.stop_price: self.order =
self.close()
            elif self.position.size < 0: # Short
                self.lowest_price_since_entry =
min(self.lowest_price_since_entry, self.data.low[0])
                new_stop = self.lowest_price_since_entry + (self.atr[0] *
self.p.atr_stop_multiplier)
                self.stop_price = min(self.stop_price, new_stop)
                if self.data.close[0] > self.stop_price: self.order =
self.close()
```

# 10.4 Backtesting

```
import backtrader as bt
import yfinance as yf
import pandas as pd
```

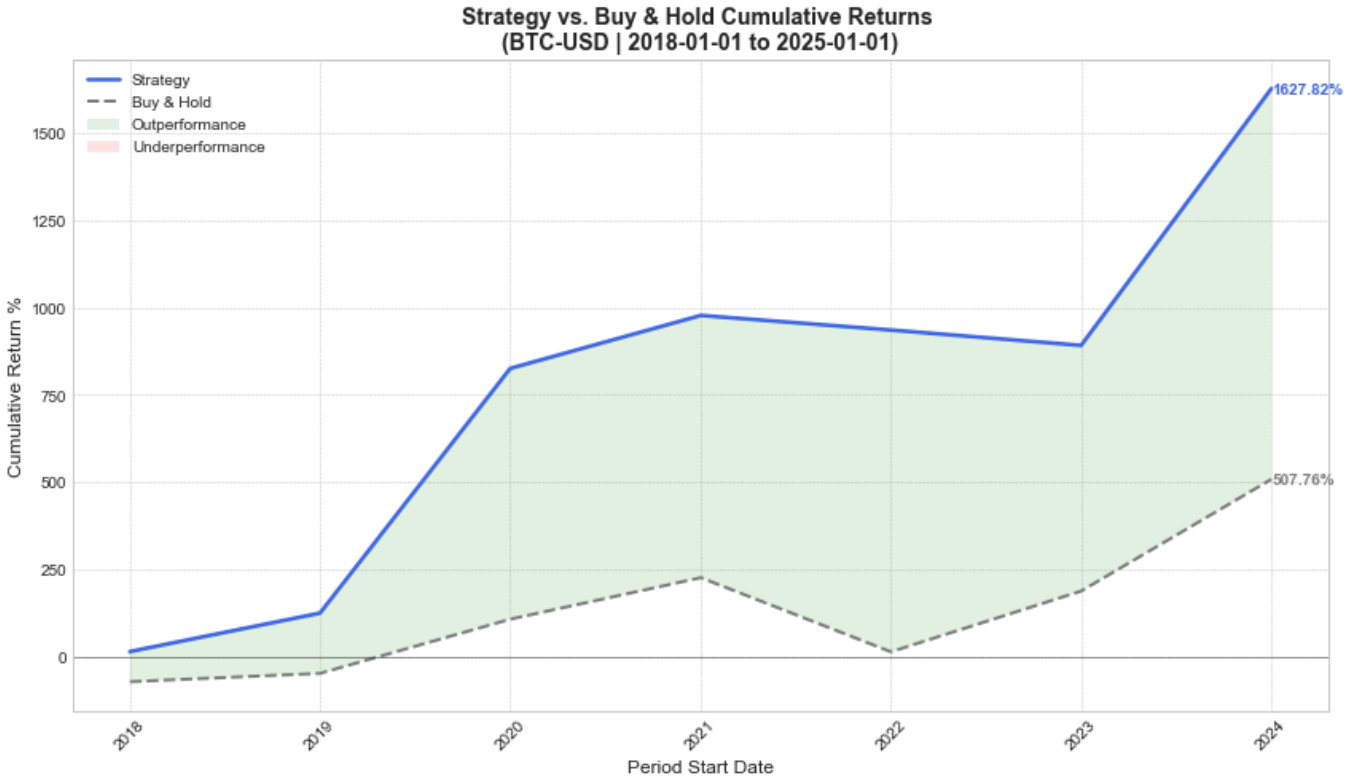
```
import numpy as np
import dateutil.relativedelta as rd
import matplotlib.pyplot as plt
%matplotlib inline
import seaborn as sns
import importlib

def load_strategy(class_name):
    module = importlib.import_module(class_name)
    strategy = getattr(module, class_name)
    return strategy

ticker = "BTC-USD"
start = "2018-01-01"
end = "2025-01-01"

start_dt = pd.to_datetime(start)
end_dt = pd.to_datetime(end)


strategy = load_strategy("MomentumIgnitionStrategy")

data = yf.download(ticker, start=start, end=end, progress=False)


data = data.droplevel(1, 1) if isinstance(data.columns, pd.MultiIndex) else
data

feed = bt.feeds.PandasData(dataname=data)
cerebro = bt.Cerebro()
cerebro.addstrategy(strategy)
cerebro.adddata(feed)
cerebro.broker.setcash(100000)
cerebro.broker.setcommission(commission=0.001)
# cerebro.addsizer(bt.sizers.PercentSizer, percents=95)

cerebro.addanalyzer(bt.analyzers.SharpeRatio, _name='sharpe',
timeframe=bt.TimeFrame.Days, riskfreerate=0.0)
cerebro.addanalyzer(bt.analyzers.DrawDown,    _name='dd')
cerebro.addanalyzer(bt.analyzers.Returns,     _name='rets')
cerebro.addanalyzer(bt.analyzers.TradeAnalyzer, _name='trades')

start_val = cerebro.broker.getvalue()
results = cerebro.run()
strat = results[0]
final_val = cerebro.broker.getvalue()
```

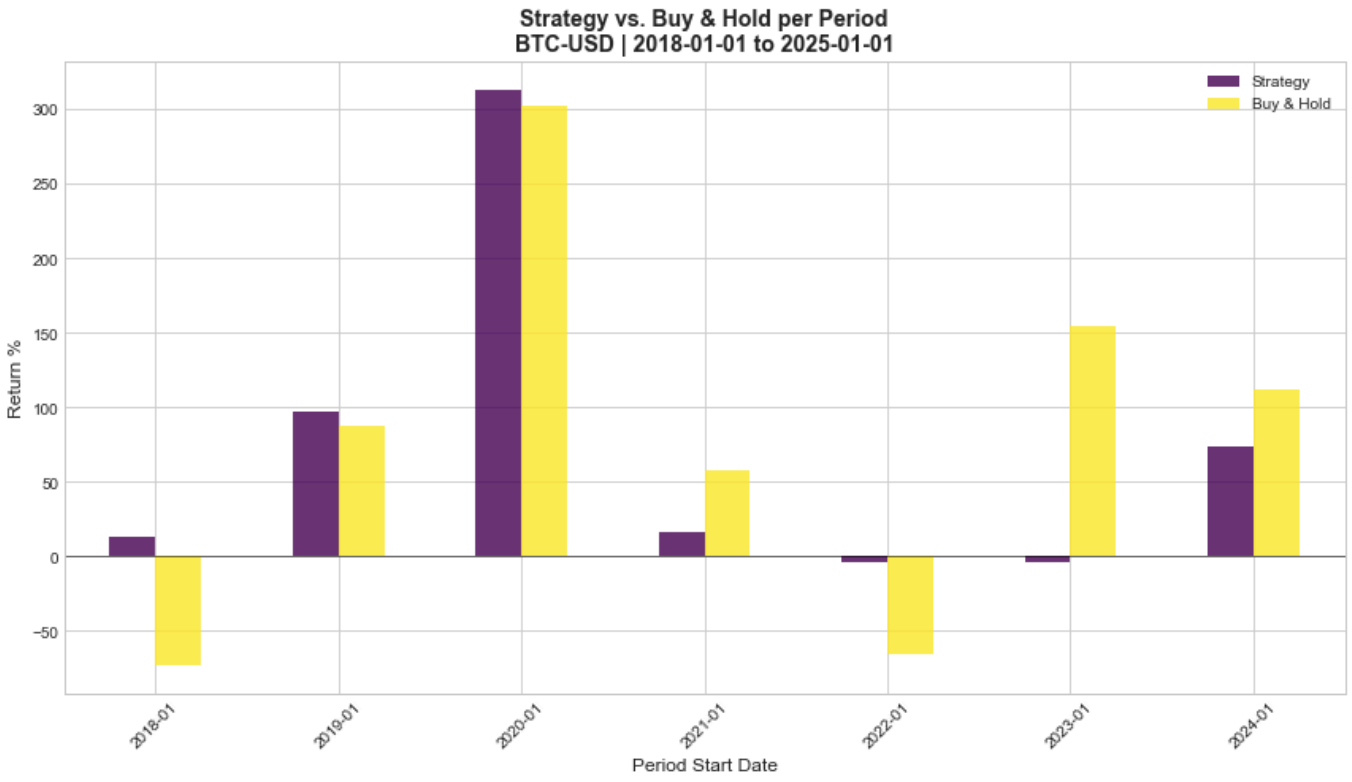
```
ret = (final_val - start_val) / start_val * 100


print(f"Return: {ret:.2f}% | Final Value: {final_val:.2f}")



sh = strat.analyzers.sharpe.get_analysis().get('sharperatio', None)
dd = strat.analyzers.dd.get_analysis()
rets = strat.analyzers.rets.get_analysis()
tr = strat.analyzers.trades.get_analysis()

print(f"Sharpe: {sh}")
print(f"Max DD: {dd.get('max', {}).get('drawdown', 0):.2f}%  Len:
{dd.get('max', {}).get('len', 0)}")
print(f"CAGR:   {rets.get('rnorm100', 0):.2f}%  Total: {rets.get('rtot',
0)*100:.2f}%")
print(f"Trades: {tr.get('total', {}).get('total', 0)}  Won: {tr.get('won',
{}).get('total', 0)}  Lost: {tr.get('lost', {}).get('total', 0)}")

plt.rcParams['figure.figsize'] = [10, 6]
plt.rcParams['font.size'] = 10
cerebro.plot(iplot=False)
```

- `consolidation_threshold=0.1` means “std dev < 10% of price.” On daily data for majors, you may want a smaller value; intra-day can tolerate higher.
- If you want **same-bar execution** at the next open with yesterday’s signal, switch the entry check to use `[-1]` values and place the order; the current version uses current bar values but still executes next bar by Backtrader’s default.


# 10.5 Rolling Backtest

Let's see a proper rolling backtest results.


Strategy vs. Buy & Hold Cumulative Returns
(BTC-USD | 2018-01-01 to 2025-01-01)
Strategy
Buy & Hold
Outperformance
Underperformance
1627.82%
507.76%
Cumulative Return %
1500
1250
1000
750
500
250
0
2018
2019
2020
2021
2022
2023
2024
Period Start Date


Strategy vs. Buy & Hold per Period
BTC-USD | 2018-01-01 to 2025-01-01
Strategy
Buy & Hold
Return %
300
250
200
150
100
50
0
−50
2018-01
2019-01
2020-01
2021-01
2022-01
2023-01
2024-01
Period Start Date

**Key Performance Indicators**
**BTC-USD | 2018-01-01 to 2025-01-01**

| Metric | Value |
|---|---|
| Total Periods | 7 |
| Winning Periods | 5 |
| Losing Periods | 2 |
| Mean Return % | 72.35 |
| Median Return % | 16.43 |
| Std Dev % | 104.79 |
| Win Rate % | 71.43 |
| Sharpe Ratio | 0.69 |
| Min Return % | -4.25 |
| Max Return % | 313.18 |

# Potential Improvements

- **Parameter Tuning**: Adjust `consolidation_period`, `consolidation_threshold`, `roc_period`, `roc_breakout_std`, or `atr_stop_multiplier` to optimize for specific assets or market conditions.
- **Additional Filters**: Incorporate volume or other momentum indicators (e.g., ADX) to further confirm breakout signals.
- **Fixed Stop-Loss**: Add a fixed stop-loss alongside trailing stops for additional risk control.
- **Performance Metrics**: Include analyzers for returns, Sharpe ratio, drawdown, and trade statistics to evaluate strategy effectiveness.
- **Multi-Asset Testing**: Apply the strategy to multiple assets to assess performance across different markets.

This strategy is designed for markets with periodic low-volatility consolidations followed by strong momentum-driven breakouts, suitable for assets like forex, stocks, or cryptocurrencies, and can be backtested to evaluate its effectiveness across various timeframes and assets.

# Chapter 11 — OBV Market Regime Breakout

**Idea**
Use **On-Balance Volume (OBV)** to catch accumulation/distribution turns, but only trade when the backdrop is supportive:

1. **Trend strength** via ADX,
2. **Price confirmation** via a breakout of recent highs/lows,
3. **Participation** via volume > its MA,
4. Momentum sanity check with RSI (avoid overbought/oversold extremes),
5. Risk handled by a **trailing percent stop**.

This strategy is part of the **Mega Backtrader Strategy Pack** — a collection of over **250 ready-to-use strategies** for your algorithmic trading.

The **Mega Backtrader Strategy Pack** gives you **250+ fully-tested Python strategies** — adaptive filters, mean reversion, momentum, volatility breakouts, machine learning models & more — all ready to deploy in Backtrader.

**Why Traders Love This Pack**

- Instant Impact — Deploy strategies and start testing today
- Zero Guesswork — Every strategy is documented, tested, and ready-to-run
- Future-Proof — Lifetime updates included
- Proven Variety — Adaptive, mean-reversion, momentum, volatility, and ML-driven models

What's Inside
✅ **250+ Strategies across multiple categories:**

- Adaptive Filters — Kalman, Hilbert, TRIX, Guppy MMA
- Mean Reversion — Bollinger, Keltner, pair trading, RSI/Stoch
- Momentum & Trend — MACD, ADX, Donchian, MA ribbons
- Volatility — ATR breakouts, Bollinger squeeze, VIX correlation
- Specialized ML & Advanced — HMM, decision trees, isolation forest, Hurst exponent

You Get
📂 Python `.py` scripts for direct Backtrader use
📄 PDF manuals detailing logic, parameters, and best practices
🎯 Get Instant Access — Download Today, Trade Tonight

👉 **Get it here**

# 11.1 Building blocks (quick math)

- **OBV**: adds volume when price up, subtracts when price down. Cumulative: $\mathrm{OBV}_t = \mathrm{OBV}_{t-1} + \mathrm{sign}(C_t - C_{t-1}) \cdot V_t$
- **OBV trend**: OBV vs. its SMA and their **crossover**.

- **Price breakout**: Close > previous $N$-bar highest high (long) or < previous $N$-bar lowest low (short).
- **Trend strength**: ADX > threshold.
- **Participation**: Volume > SMA(volume).
- **Momentum sanity**: RSI not stretched against the trade (longs avoid >70, shorts avoid <30).
- **Exit**: trailing stop as a fraction of price (broker-managed StopTrail).

---

# 11.2 Step-by-step with explanations

## 11.2.1 Custom OBV (cumulative)

```
import backtrader as bt

class CustomOBV(bt.Indicator):
    lines = ('obv',)
    plotinfo = dict(subplot=True)

    def next(self):
        # Initialize on the first calculable bar, then accumulate
        if len(self) == 1:
            if self.data.close[0] > self.data.close[-1]:
                self.lines.obv[0] = self.data.volume[0]
            elif self.data.close[0] < self.data.close[-1]:
                self.lines.obv[0] = -self.data.volume[0]
            else:
                self.lines.obv[0] = 0
        else:
            prev_obv = self.lines.obv[-1]
            if self.data.close[0] > self.data.close[-1]:
                self.lines.obv[0] = prev_obv + self.data.volume[0]
            elif self.data.close[0] < self.data.close[-1]:
                self.lines.obv[0] = prev_obv - self.data.volume[0]
            else:
                self.lines.obv[0] = prev_obv
```

### Why this way

- OBV measures whether volume is **mostly buying or selling pressure**.
- We compare today's close to yesterday's to decide the volume sign, then **accumulate**.

## 11.2.2 Regime, momentum, and participation filters

```
class _OBV_BuildFilters(bt.Strategy):
    params = dict(
        obv_ma_period=7,
        rsi_period=14,
        volume_ma_period=7,
        adx_period=7,
        adx_threshold=20,
        breakout_lookback=7,
        trail_percent=0.03,
    )

    def __init__(self):
        d = self.datas[0]
        self.close, self.high, self.low, self.vol = d.close, d.high, d.low,
d.volume

        # OBV + its SMA, and a crossover detector (OBV crossing up = bullish)
        self.obv     = CustomOBV(d)
        self.obv_ma  = bt.ind.SMA(self.obv.lines.obv,
period=self.p.obv_ma_period)
        self.obv_x   = bt.ind.CrossOver(self.obv.lines.obv, self.obv_ma)

        # Momentum sanity gate (explicitly on price close)
        self.rsi     = bt.ind.RSI(self.close, period=self.p.rsi_period)

        # Participation: raw volume vs its own SMA
        self.vsma    = bt.ind.SMA(self.vol, period=self.p.volume_ma_period)

        # Trend strength (not direction) - only take trades when meaningful
trend exists
        self.adx     = bt.ind.ADX(d, period=self.p.adx_period)

        # Price breakout rails (use previous N bars for signal; today must
exceed yesterday's rail)
        self.hhN     = bt.ind.Highest(self.high,
period=self.p.breakout_lookback)
        self.llN     = bt.ind.Lowest(self.low,
period=self.p.breakout_lookback)

        self.order = None
```

### Why these filters

- **ADX>threshold**: filters choppy regimes where OBV crosses whipsaw.
- **Volume>MA**: demand confirmation—breakouts with weak participation are less reliable.

- **RSI**: avoid late longs when already overbought (>70) and late shorts when already oversold (<30).
- **Breakout rails**: price should **confirm** the OBV thrust by clearing recent extremes.

## 11.2.3 Entry logic (all conditions align)

```
class _OBV_Entries(_OBV_BuildFilters):
    def next(self):
        if self.order:
            return

        # Ensure warm-up: indicators need enough bars; Highest/Lowest uses
past N bars
        if len(self) < max(self.p.obv_ma_period, self.p.rsi_period,
                           self.p.volume_ma_period, self.p.adx_period,
                           self.p.breakout_lookback):
            return

        # Regime strength
        is_trending = self.adx.adx[0] > self.p.adx_threshold

        # Price breakout uses PREVIOUS rail: close > yesterday's N-bar high,
or < yesterday's N-bar low
        bull_break = self.close[0] > self.hhN[-1]
        bear_break = self.close[0] < self.llN[-1]

        if not self.position:
            # Bullish composite: OBV crosses UP, RSI not overbought, volume
strong, ADX strong, price breaks out
            if (self.obv_x[0] > 0 and
                self.rsi[0] < 70 and
                self.vol[0] > self.vsma[0] and
                is_trending and
                bull_break):
                self.order = self.buy()

            # Bearish composite: OBV crosses DOWN, RSI not oversold, volume
strong, ADX strong, price breaks down
            elif (self.obv_x[0] < 0 and
                  self.rsi[0] > 30 and
                  self.vol[0] > self.vsma[0] and
                  is_trending and
                  bear_break):
                self.order = self.sell()
```

**Why this combination**

- We require **signal confluence** (OBV thrust + price breakout) **inside a trending regime** with **healthy participation** and **not extremely stretched momentum**.

## 11.2.4 Trailing risk (broker-managed)

```
class _OBV_Trailing(_OBV_Entries):
    def notify_order(self, order):
        if order.status in [order.Submitted, order.Accepted]:
            return
        if order.status in [order.Completed]:
            if order.isbuy():
                # Attach a 3% trailing stop for longs (follows price up,
triggers on pullback)
                self.sell(exectype=bt.Order.StopTrail,
trailpercent=self.p.trail_percent)
            elif order.issell():
                # Mirror for shorts
                self.buy(exectype=bt.Order.StopTrail,
trailpercent=self.p.trail_percent)
        self.order = None
```

**Why**

- `StopTrail` moves automatically with favorable price, simplifying exit management.
- `trailpercent=0.03` means **3%**—tight for daily bars on volatile assets; feel free to widen.

---

# 11.3 Complete strategy (your code unchanged)

```
import backtrader as bt

# Custom On-Balance Volume Indicator (remains the same)
class CustomOBV(bt.Indicator):
    lines = ('obv',)
    plotinfo = dict(subplot=True)
    def next(self):
        if len(self) == 1:
            if self.data.close[0] > self.data.close[-1]:
                self.lines.obv[0] = self.data.volume[0]
            elif self.data.close[0] < self.data.close[-1]:
                self.lines.obv[0] = -self.data.volume[0]
```

```
            else:
                self.lines.obv[0] = 0
        else:
            prev_obv = self.lines.obv[-1]
            if self.data.close[0] > self.data.close[-1]:
                self.lines.obv[0] = prev_obv + self.data.volume[0]
            elif self.data.close[0] < self.data.close[-1]:
                self.lines.obv[0] = prev_obv - self.data.volume[0]
            else:
                self.lines.obv[0] = prev_obv


class OBVMarketRegimeStrategyBreakout(bt.Strategy):
    params = (
        ('obv_ma_period', 7),
        ('rsi_period', 14),
        ('volume_ma_period', 7),
        ('adx_period', 7),
        ('adx_threshold', 20),
        ('breakout_lookback', 7), # Lookback for highest high/lowest low
        ('trail_percent', 0.03),
    )

    def __init__(self):
        self.order = None
        self.dataclose = self.datas[0].close
        self.datahigh = self.datas[0].high
        self.datalow = self.datas[0].low

        self.obv = CustomOBV(self.datas[0])
        self.obv_ma = bt.indicators.SMA(self.obv.lines.obv,
period=self.p.obv_ma_period)
        self.obv_cross = bt.indicators.CrossOver(self.obv.lines.obv,
self.obv_ma)

        self.rsi = bt.indicators.RSI(period=self.p.rsi_period)
        self.volume_ma = bt.indicators.SMA(self.data.volume,
period=self.p.volume_ma_period)

        self.adx = bt.indicators.ADX(self.datas[0], period=self.p.adx_period)

        # Highest high and Lowest low for breakout
        self.highest_high = bt.indicators.Highest(self.datahigh,
period=self.p.breakout_lookback)
        self.lowest_low = bt.indicators.Lowest(self.datalow,
period=self.p.breakout_lookback)
```

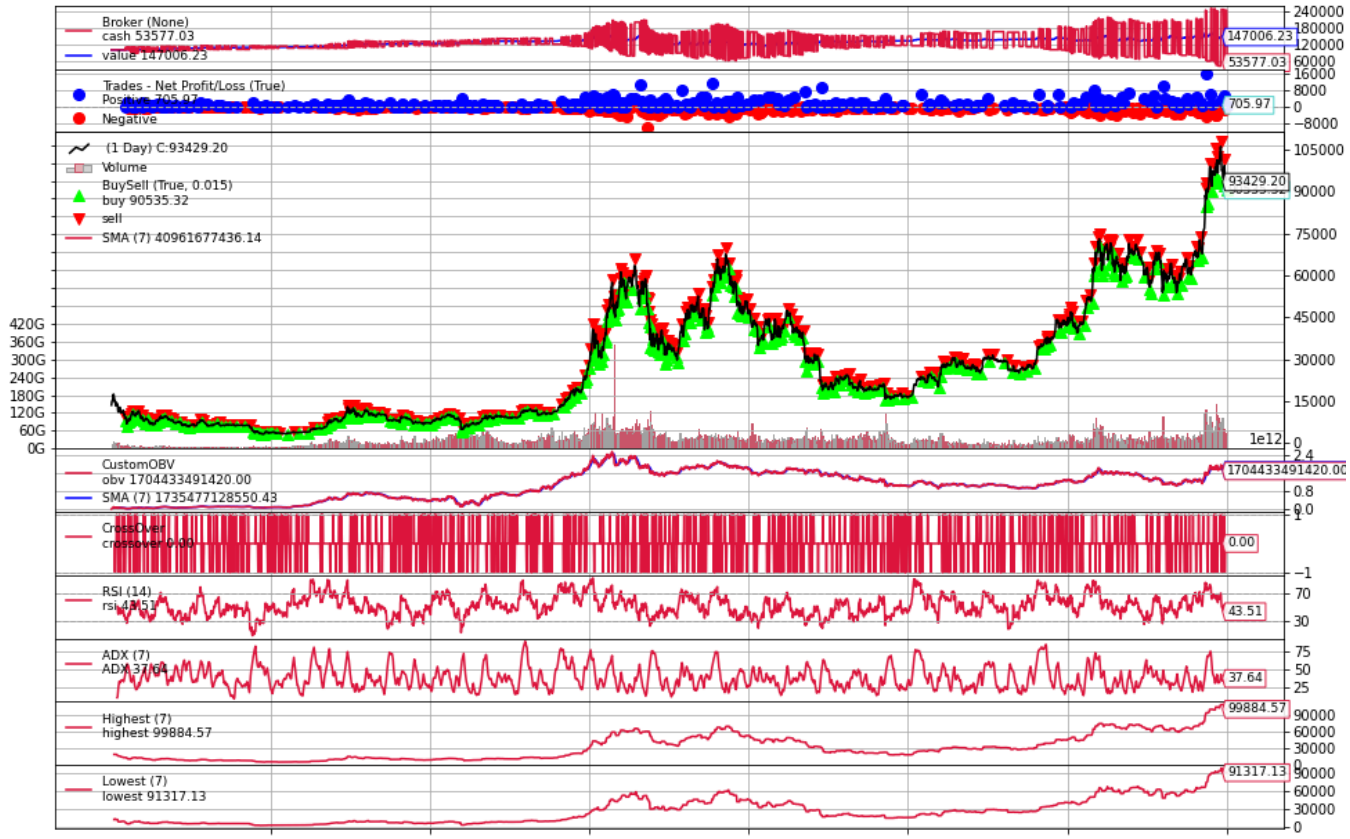
```
    def notify_order(self, order):
        if order.status in [order.Submitted, order.Accepted]:
            return
        if order.status in [order.Completed]:
            if order.isbuy():
                self.sell(exectype=bt.Order.StopTrail,
trailpercent=self.p.trail_percent)
            elif order.issell():
                self.buy(exectype=bt.Order.StopTrail,
trailpercent=self.p.trail_percent)
        self.order = None

    def next(self):
        if self.order:
            return

        if len(self) < max(self.p.obv_ma_period, self.p.rsi_period,
self.p.volume_ma_period, self.p.adx_period, self.p.breakout_lookback):
            return

        is_trending = self.adx.adx[0] > self.p.adx_threshold

        # Price Breakout Confirmation
        is_bullish_breakout = self.dataclose[0] > self.highest_high[-1] #
Current close above previous N-period high
        is_bearish_breakout = self.dataclose[0] < self.lowest_low[-1]   #
Current close below previous N-period low

        if not self.position:
            if (self.obv_cross[0] > 0.0 and
                self.rsi[0] < 70 and
                self.data.volume[0] > self.volume_ma[0] and
                is_trending and
                is_bullish_breakout): # Added price breakout
                self.order = self.buy()

            elif (self.obv_cross[0] < 0.0 and
                  self.rsi[0] > 30 and
                  self.data.volume[0] > self.volume_ma[0] and
                  is_trending and
                  is_bearish_breakout): # Added price breakout
                self.order = self.sell()
```

# 11.4 Backtesting

```
import backtrader as bt
import yfinance as yf
import pandas as pd
import numpy as np
import dateutil.relativedelta as rd
import matplotlib.pyplot as plt
%matplotlib inline
import seaborn as sns
import importlib

def load_strategy(class_name):
    module = importlib.import_module(class_name)
    strategy = getattr(module, class_name)
    return strategy

ticker = "BTC-USD"
start = "2018-01-01"
end = "2025-01-01"

start_dt = pd.to_datetime(start)
end_dt = pd.to_datetime(end)


strategy = load_strategy("OBVMarketRegimeStrategyBreakout")

data = yf.download(ticker, start=start, end=end, progress=False)


data = data.droplevel(1, 1) if isinstance(data.columns, pd.MultiIndex) else
data

feed = bt.feeds.PandasData(dataname=data)
cerebro = bt.Cerebro()
cerebro.addstrategy(strategy)
cerebro.adddata(feed)
cerebro.broker.setcash(100000)
cerebro.broker.setcommission(commission=0.001)
# cerebro.addsizer(bt.sizers.PercentSizer, percents=95)

cerebro.addanalyzer(bt.analyzers.SharpeRatio, _name='sharpe',
timeframe=bt.TimeFrame.Days, riskfreerate=0.0)
cerebro.addanalyzer(bt.analyzers.DrawDown,    _name='dd')
cerebro.addanalyzer(bt.analyzers.Returns,     _name='rets')
```

```
cerebro.addanalyzer(bt.analyzers.TradeAnalyzer, _name='trades')

start_val = cerebro.broker.getvalue()
results = cerebro.run()
strat = results[0]
final_val = cerebro.broker.getvalue()
ret = (final_val - start_val) / start_val * 100


print(f"Return: {ret:.2f}% | Final Value: {final_val:.2f}")



sh = strat.analyzers.sharpe.get_analysis().get('sharperatio', None)
dd = strat.analyzers.dd.get_analysis()
rets = strat.analyzers.rets.get_analysis()
tr = strat.analyzers.trades.get_analysis()

print(f"Sharpe: {sh}")
print(f"Max DD: {dd.get('max', {}).get('drawdown', 0):.2f}%  Len:
{dd.get('max', {}).get('len', 0)}")
print(f"CAGR:   {rets.get('rnorm100', 0):.2f}%  Total: {rets.get('rtot',
0)*100:.2f}%")
print(f"Trades: {tr.get('total', {}).get('total', 0)}  Won: {tr.get('won',
{}).get('total', 0)}  Lost: {tr.get('lost', {}).get('total', 0)}")

plt.rcParams['figure.figsize'] = [10, 6]
plt.rcParams['font.size'] = 10
cerebro.plot(iplot=False)
```

- The breakout uses **yesterday's** rails ( `highest_high[-1]` , `lowest_low[-1]` ) to avoid same-bar bias.
- If your `RSI(period=14)` call ever binds to the wrong line in some Backtrader versions, use `bt.ind.RSI(self.dataclose, period=...)` explicitly.
- `trail_percent=0.03` = **3%** trailing stop. If trades get cut too early on volatile instruments, widen it (e.g., 5–8%).

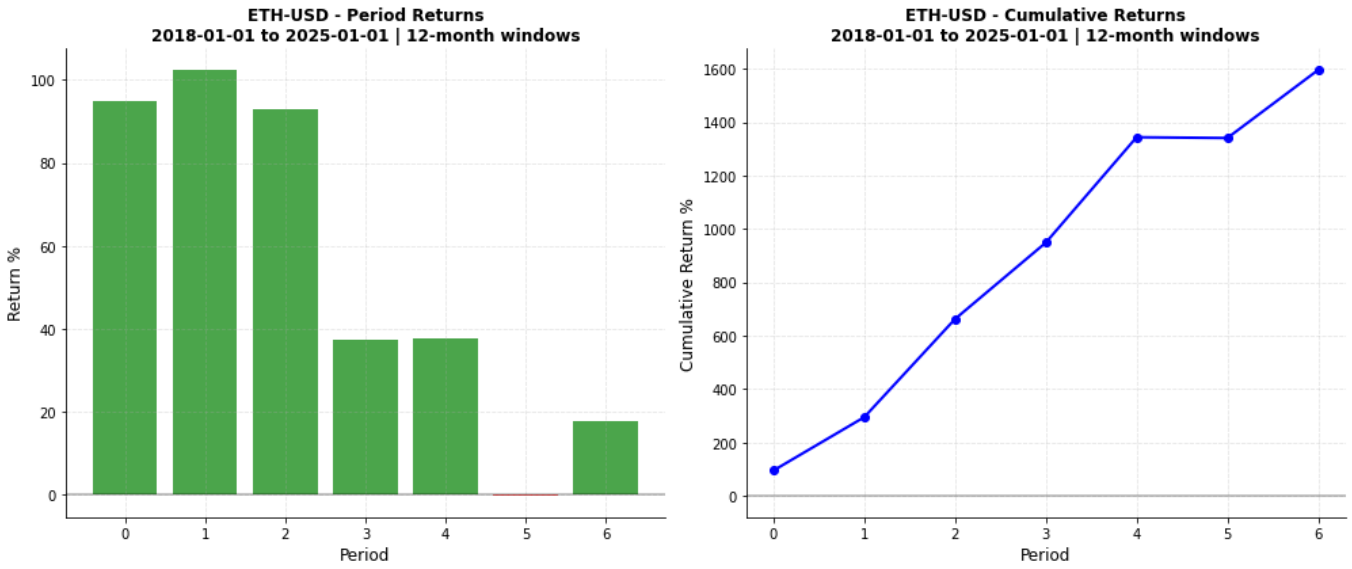



# 11.5 Rolling Backtest

Let's try a rolling backtest for a more comprehenisve analysis.

```
if __name__ == '__main__':
    df = run_rolling_backtest(
        ticker="ETH-USD",
        start="2018-01-01",
        end="2025-01-01",
        window_months=12,
        strategy_params=dict(
            obv_ma_period=7,
            rsi_period=14,
            volume_ma_period=7,
            adx_period=7,
            adx_threshold=20,
            breakout_lookback=7,
            trail_percent=0.03
        ),
        cash=100_000,
        commission=0.001,
        stake_pct=95,
        min_bars=90
    )
```

```
    print("\nPer-window results")
    print(df)
```




- OBV smoothing: lengthen `obv_ma_period` to reduce noise; shorten to react faster.
- Breakout lookback: 5–20 bars; longer windows require stronger moves but reduce false breaks.
- ADX threshold: 20–25 for conservative trend gating; lower to catch earlier phases.
- Trailing stop percent: widen on high-beta assets or swap to an ATR-based `trailamount` to normalize across regimes.
- Add a higher-timeframe trend filter (e.g., daily vs 4-hour) to avoid counter-regime trades.
- Require a minimum range expansion on entry bar (e.g., true range > k × ATR) to avoid weak breakouts.
- Replace RSI guardrails with an oscillator slope filter to prioritize acceleration over absolute level.

# Chapter 12 — OBV Momentum Strategy

Use **On-Balance Volume (OBV)** to detect accumulation/distribution turns and trade the **OBV vs. OBV-SMA crossover** only when participation is healthy (volume above average) and momentum isn't already stretched (RSI filter). Risk is handled by a broker-managed **trailing percent stop**.

This strategy is part of the **Mega Backtrader Strategy Pack** — a collection of over **250 ready-to-use strategies** for your algorithmic trading.

The **Mega Backtrader Strategy Pack** gives you **250+ fully-tested Python strategies** — adaptive filters, mean reversion, momentum, volatility breakouts, machine learning models & more — all ready to deploy in Backtrader.

**Why Traders Love This Pack**

- Instant Impact — Deploy strategies and start testing today
- Zero Guesswork — Every strategy is documented, tested, and ready-to-run
- Future-Proof — Lifetime updates included
- Proven Variety — Adaptive, mean-reversion, momentum, volatility, and ML-driven models

What's Inside

✅ **250+ Strategies across multiple categories:**

- Adaptive Filters — Kalman, Hilbert, TRIX, Guppy MMA
- Mean Reversion — Bollinger, Keltner, pair trading, RSI/Stoch
- Momentum & Trend — MACD, ADX, Donchian, MA ribbons
- Volatility — ATR breakouts, Bollinger squeeze, VIX correlation
- Specialized ML & Advanced — HMM, decision trees, isolation forest, Hurst exponent

You Get

📂 Python `.py` scripts for direct Backtrader use

📄 PDF manuals detailing logic, parameters, and best practices

🎯 Get Instant Access — Download Today, Trade Tonight

👉 **Get it here**

# 12.1 Concepts and formulas

On-Balance Volume (cumulative)

$$\mathrm{OBV}_t = \mathrm{OBV}_{t-1} + \begin{cases} +V_t & \text{if } C_t > C_{t-1} \\ -V_t & \text{if } C_t < C_{t-1} \\ 0 & \text{otherwise} \end{cases}$$

Momentum/participation filters

- **OBV crossover**: go long when OBV crosses **up** its SMA; go short when it crosses **down**.
- **RSI sanity**: avoid longs if RSI is already >70; avoid shorts if RSI <30.
- **Volume participation**: require current volume > SMA(volume).

Exit

- **Trailing stop**: broker-managed `StopTrail` with `trailpercent`.

# 12.2 Step-by-step build with explanations

## 12.2.1 Custom OBV (cumulative)

```
import backtrader as bt

class CustomOBV(bt.Indicator):
    lines = ('obv',)
    plotinfo = dict(subplot=True)

    def next(self):
        # On the first calculable bar, seed OBV with +/- today's volume
        if len(self) == 1:
            if self.data.close[0] > self.data.close[-1]:
                self.lines.obv[0] = self.data.volume[0]
            elif self.data.close[0] < self.data.close[-1]:
                self.lines.obv[0] = -self.data.volume[0]
            else:
                self.lines.obv[0] = 0
        else:
            # Thereafter, accumulate by adding/subtracting today's volume
            prev = self.lines.obv[-1]
            if self.data.close[0] > self.data.close[-1]:
                self.lines.obv[0] = prev + self.data.volume[0]
            elif self.data.close[0] < self.data.close[-1]:
                self.lines.obv[0] = prev - self.data.volume[0]
            else:
                self.lines.obv[0] = prev
```

Why: OBV is a **direction-signed volume cumulation** — it rises when up-days dominate on volume and falls when down-days dominate.

## 12.2.2 Wire indicators and crossover trigger

```
class _OBV_Momentum_Part1(bt.Strategy):
    params = dict(obv_ma_period=30, rsi_period=14, volume_ma_period=7)

    def __init__(self):
        d = self.datas[0]
        self.obv = CustomOBV(d)
        self.obv_ma = bt.ind.SMA(self.obv.lines.obv,
period=self.p.obv_ma_period)
        self.obv_cross = bt.ind.CrossOver(self.obv.lines.obv, self.obv_ma)
        self.rsi = bt.ind.RSI(d.close, period=self.p.rsi_period)
        self.vol_sma = bt.ind.SMA(d.volume, period=self.p.volume_ma_period)
        self.order = None
```

Why: `CrossOver(OBV, OBV_SMA)` provides clean, event-based signals; RSI and volume SMA are simple quality gates.

## 12.2.3 Entry rules with filters

```
class _OBV_Momentum_Part2(_OBV_Momentum_Part1):
    def next(self):
        if self.order:
            return
        if not self.position:
            # Long when OBV crosses UP its SMA, RSI not overbought, and volume
> avg
            if (self.obv_cross[0] > 0 and
                self.rsi[0] < 70 and
                self.data.volume[0] > self.vol_sma[0]):
                self.order = self.buy()
            # Short when OBV crosses DOWN its SMA, RSI not oversold, and
volume > avg
            elif (self.obv_cross[0] < 0 and
                  self.rsi[0] > 30 and
                  self.data.volume[0] > self.vol_sma[0]):
                self.order = self.sell()
```

Why: Require **signal + participation + non-extreme RSI** to avoid chasing exhausted moves on thin volume.

## 12.2.4 Broker-managed trailing stop

```
class _OBV_Momentum_Part3(_OBV_Momentum_Part2):
    params = dict(trail_percent=0.02)  # 2% trailing distance

    def notify_order(self, order):
        if order.status in [order.Submitted, order.Accepted]:
            return
        if order.status in [order.Completed]:
            if order.isbuy():
                # Attach trailing SELL to protect the long
                self.sell(exectype=bt.Order.StopTrail,
trailpercent=self.p.trail_percent)
            elif order.issell():
                # Attach trailing BUY to protect the short
                self.buy(exectype=bt.Order.StopTrail,
trailpercent=self.p.trail_percent)
        self.order = None
```

Why: `StopTrail` auto-adjusts with favorable price and fires when price pulls back by the specified percent. `0.02` = 2% (tight for volatile assets; widen if you see premature exits).

# 12.3 Complete Backtrader strategy (as provided)

The class below is the exact implementation you supplied for this chapter, kept unchanged for drop-in use.

```
import backtrader as bt


# Custom On-Balance Volume Indicator
class CustomOBV(bt.Indicator):
    lines = ('obv',)
    plotinfo = dict(subplot=True)

    def next(self):
        if len(self) == 1:
            if self.data.close[0] > self.data.close[-1]:
                self.lines.obv[0] = self.data.volume[0]
            elif self.data.close[0] < self.data.close[-1]:
                self.lines.obv[0] = -self.data.volume[0]
            else:
                self.lines.obv[0] = 0
        else:
            prev_obv = self.lines.obv[-1]
            if self.data.close[0] > self.data.close[-1]:
                self.lines.obv[0] = prev_obv + self.data.volume[0]
            elif self.data.close[0] < self.data.close[-1]:
                self.lines.obv[0] = prev_obv - self.data.volume[0]
            else:
                self.lines.obv[0] = prev_obv


class OBVmomentumStrategy(bt.Strategy):
    params = (
        ('obv_ma_period', 30),
        ('trail_percent', 0.02),
        ('rsi_period', 14),
        ('volume_ma_period', 7),
    )

    def __init__(self):
        self.order = None
```

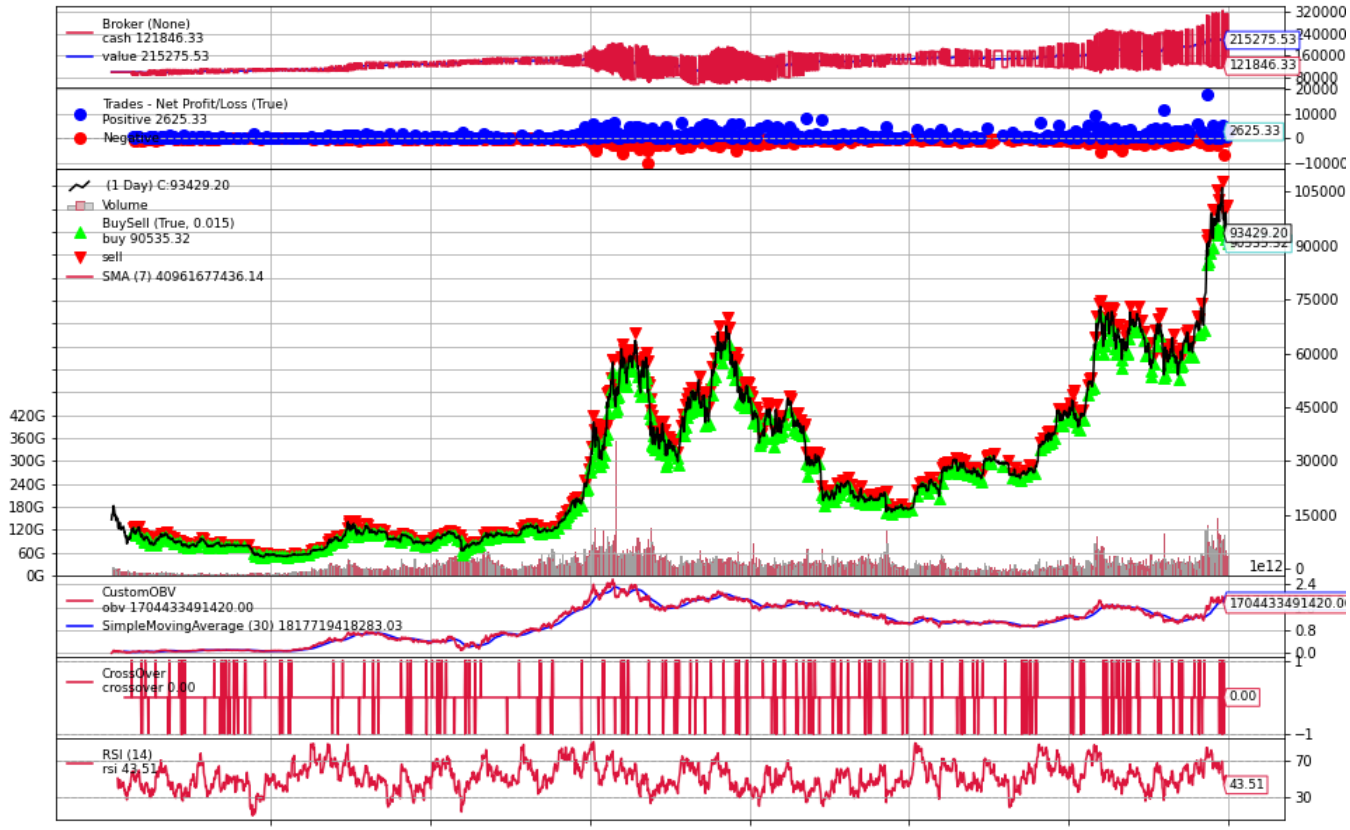
```
        # Original indicators
        self.obv = CustomOBV(self.datas[0])
        self.obv_ma = bt.indicators.SimpleMovingAverage(
            self.obv.lines.obv, period=self.params.obv_ma_period
        )
        self.obv_cross = bt.indicators.CrossOver(self.obv.lines.obv,
self.obv_ma)

        # Two simple filters for better signal quality
        self.rsi = bt.indicators.RSI(period=self.params.rsi_period)
        self.volume_ma = bt.indicators.SMA(self.data.volume,
period=self.params.volume_ma_period)

    def notify_order(self, order):
        if order.status in [order.Submitted, order.Accepted]:
            return
        if order.status in [order.Completed]:
            if order.isbuy():
                self.sell(exectype=bt.Order.StopTrail,
trailpercent=self.params.trail_percent)
            elif order.issell():
                self.buy(exectype=bt.Order.StopTrail,
trailpercent=self.params.trail_percent)
        self.order = None

    def next(self):
        if self.order:
            return

        if not self.position:
            # Long signal: OBV crosses up + RSI not overbought + volume above
average
            if (self.obv_cross[0] > 0.0 and
                self.rsi[0] < 70 and
                self.data.volume[0] > self.volume_ma[0]):
                self.order = self.buy()

            # Short signal: OBV crosses down + RSI not oversold + volume above
average
            elif (self.obv_cross[0] < 0.0 and
                  self.rsi[0] > 30 and
                  self.data.volume[0] > self.volume_ma[0]):
                self.order = self.sell()
```

# 12.4 Bakctesting

```
import backtrader as bt
import yfinance as yf
import pandas as pd
import numpy as np
import dateutil.relativedelta as rd
import matplotlib.pyplot as plt
%matplotlib inline
import seaborn as sns
import importlib

def load_strategy(class_name):
    module = importlib.import_module(class_name)
    strategy = getattr(module, class_name)
    return strategy

ticker = "BTC-USD"
start = "2018-01-01"
end = "2025-01-01"

start_dt = pd.to_datetime(start)
end_dt = pd.to_datetime(end)


strategy = load_strategy("OBVmomentumStrategy")

data = yf.download(ticker, start=start, end=end, progress=False)


data = data.droplevel(1, 1) if isinstance(data.columns, pd.MultiIndex) else
data

feed = bt.feeds.PandasData(dataname=data)
cerebro = bt.Cerebro()
cerebro.addstrategy(strategy)
cerebro.adddata(feed)
cerebro.broker.setcash(100000)
cerebro.broker.setcommission(commission=0.001)
# cerebro.addsizer(bt.sizers.PercentSizer, percents=95)

cerebro.addanalyzer(bt.analyzers.SharpeRatio, _name='sharpe',
timeframe=bt.TimeFrame.Days, riskfreerate=0.0)
cerebro.addanalyzer(bt.analyzers.DrawDown,    _name='dd')
cerebro.addanalyzer(bt.analyzers.Returns,     _name='rets')
cerebro.addanalyzer(bt.analyzers.TradeAnalyzer, _name='trades')

start_val = cerebro.broker.getvalue()
```

```
results = cerebro.run()
strat = results[0]
final_val = cerebro.broker.getvalue()
ret = (final_val - start_val) / start_val * 100


print(f"Return: {ret:.2f}% | Final Value: {final_val:.2f}")



sh = strat.analyzers.sharpe.get_analysis().get('sharperatio', None)
dd = strat.analyzers.dd.get_analysis()
rets = strat.analyzers.rets.get_analysis()
tr = strat.analyzers.trades.get_analysis()

print(f"Sharpe: {sh}")
print(f"Max DD: {dd.get('max', {}).get('drawdown', 0):.2f}%  Len:
{dd.get('max', {}).get('len', 0)}")
print(f"CAGR:   {rets.get('rnorm100', 0):.2f}%  Total: {rets.get('rtot',
0)*100:.2f}%")
print(f"Trades: {tr.get('total', {}).get('total', 0)}  Won: {tr.get('won',
{}).get('total', 0)}  Lost: {tr.get('lost', {}).get('total', 0)}")

plt.rcParams['figure.figsize'] = [10, 6]
plt.rcParams['font.size'] = 10
cerebro.plot(iplot=False)
```

- Warm-up: give the OBV/RSI/volume SMA at least `max(obv_ma_period, rsi_period, volume_ma_period)` bars before expecting valid signals.
- Long-only venues: remove the short branch.
- If you want stricter signals, require **RSI slope** confirmation ( `rsi[0] > rsi[-1]` for longs; reverse for shorts) or add a simple price trend filter ( `close > SMA(close, n)` ).




# 12.5 Rolling Backtest

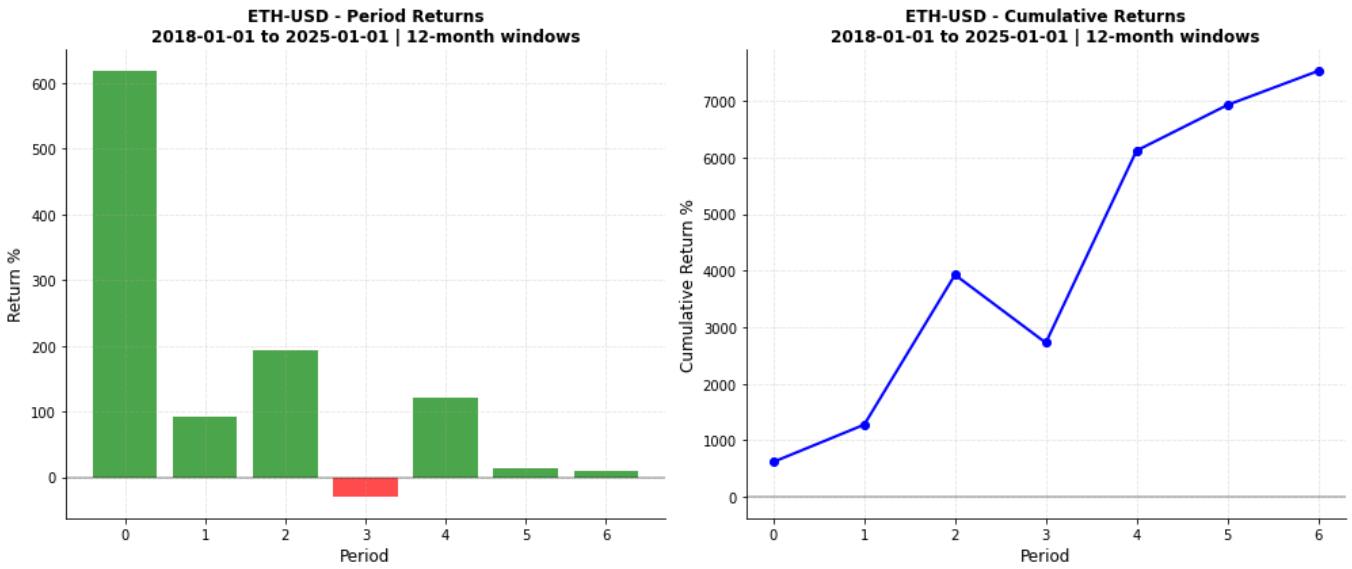
Here's a rolling backtest example:

```
strategy = load_strategy("OBVmomentumStrategy")

    ticker = "ETH-USD"
    start = "2018-01-01"
    end = "2025-01-01"
    window_months = 12

    df = run_rolling_backtest(ticker=ticker, start=start, end=end,
window_months=window_months)
```

and the results:

```
=== ROLLING BACKTEST RESULTS ===
        start         end  return_pct    final_value
0  2018-01-01  2019-01-01  618.140647  718140.646858
1  2019-01-01  2020-01-01   91.665049  191665.049024
2  2020-01-01  2021-01-01  192.492401  292492.400598
3  2021-01-01  2022-01-01  -29.841276   70158.724025
```

```
4  2022-01-01  2023-01-01  120.281440  220281.440071
5  2023-01-01  2024-01-01   12.970251  112970.251078
6  2024-01-01  2025-01-01    8.558271  108558.271083

=== ROLLING BACKTEST STATISTICS ===
Mean Return %: 144.90
Median Return %: 91.67
Std Dev %: 205.70
Min Return %: -29.84
Max Return %: 618.14
Sharpe Ratio: 0.70
```




The OBV Momentum crossover system produced **exceptionally high returns in certain years** (notably +618% in 2018 and +192% in 2020) but also faced **significant volatility** and at least one deep drawdown year (−29.8% in 2021).

The **mean annual return of ~145%** is impressive, but the **large standard deviation (~206%)** and a Sharpe ratio of **0.70** reveal that these gains came with substantial risk swings.

In short: **explosive upside potential in trending, high-volume markets** but prone to underperformance or losses when volume flow signals are noisy or trend reversals occur. The strategy thrives in strong momentum regimes and should be paired with risk overlays to tame volatility.

# Chapter 13 — Ornstein–Uhlenbeck (OU) Mean Reversion

This chapter models log-price as a continuous-time **Ornstein–Uhlenbeck** (OU) process and trades **z-score deviations** from the rolling equilibrium mean. We estimate OU parameters from recent data, compute the current deviation $z$, and take contrarian positions only when a simple trend filter agrees with the direction of expected reversion.



# 13.1 OU process, discretization, and what we estimate

Continuous-time OU

$$dX_t = \theta(\mu - X_t)\,dt + \sigma\,dW_t$$

- $X_t$: log-price (stationary state we model)
- $\mu$: long-run equilibrium level

- $\theta > 0$: mean-reversion speed
- $\sigma > 0$: diffusion (volatility of the process)

Discretization (with $\Delta t = 1$ bar for simplicity)

$$X_t - X_{t-1} = \alpha + \beta X_{t-1} + \varepsilon_t$$

Identify with OU:

$$\beta = -\theta, \qquad \alpha = \theta\mu, \qquad \varepsilon_t \sim \mathcal{N}(0, \sigma_\varepsilon^2)$$

So if we regress $\Delta X_t$ on $X_{t-1}$, we can recover

$$\hat{\theta} = -\hat{\beta}, \qquad \hat{\mu} = \frac{\hat{\alpha}}{\hat{\theta}} \ (\text{if } \hat{\theta} > 0)$$

The OU's **equilibrium standard deviation** is

$$\text{eq\_std} = \frac{\sigma}{\sqrt{2\theta}}$$

and our **z-score** at the current bar is

$$z_t = \frac{X_t - \hat{\mu}}{\widehat{\text{eq\_std}}}$$

Trade idea

- If price is **below** equilibrium in an **uptrend**, expect reversion **up** → go long when $z < -z_{\text{entry}}$.
- If price is **above** equilibrium in a **downtrend**, expect reversion **down** → go short when $z > z_{\text{entry}}$.
- Exit near the mean: for longs when $z > -z_{\text{exit}}$; for shorts when $z < z_{\text{exit}}$.

# 13.2 Step-by-step build with explanations

## 13.2.1 Strategy skeleton and inputs

```
import backtrader as bt
import numpy as np
from scipy import stats

class _OU_Part1(bt.Strategy):
    params = dict(
        lookback=30,        # window for OU estimation
        sma_period=30,      # simple trend filter on price
        entry_threshold=1., # |z| level to enter
```

```
        exit_threshold=0.,  # z level to exit (0 = cross mean)
        printlog=False
    )

    def __init__(self):
        d = self.datas[0]
        self.close = d.close
        # Trend filter: simple, transparent, and aligns direction with
expected reversion
        self.sma = bt.ind.SMA(self.close, period=self.p.sma_period)

        self.order = None
        self.position_type = None   # 'long' or 'short'
        self.ou_params = []         # diagnostics if you wish to inspect later
        self.z_scores = []
```

Why

- `lookback` controls how local your OU fit is.
- The SMA trend filter avoids fighting large drifts: long only if price above SMA; short only if below.

## 13.2.2 Estimating OU parameters by OLS on ΔX vs X_{t−1}

```
class _OU_Part2(_OU_Part1):
    def estimate_ou_parameters(self, log_prices):
        # Require enough data points to fit a regression
        if len(log_prices) < 10:
            return None, None, None, None

        x_lag = log_prices[:-1]     # X_{t-1}
        dx    = np.diff(log_prices) # ΔX_t

        # OLS: ΔX_t = intercept + slope * X_{t-1} + ε_t
        slope, intercept, *_ = stats.linregress(x_lag, dx)

        theta = -slope
        if theta > 1e-6:
            mu = intercept / theta
        else:
            mu = np.mean(log_prices)  # fallback if theta ~ 0 (no reversion
detected)

        resid = dx - (intercept + slope * x_lag)
        sigma = np.std(resid)
```

```
        eq_std = sigma / np.sqrt(2*theta) if theta > 1e-6 else sigma
        return mu, theta, sigma, eq_std
```

Why

- We're directly mapping regression coefficients back to OU.
- `theta` very close to 0 implies poor mean-reversion structure; we fall back to a simple mean and use `sigma` as scale.

## 13.2.3 Compute current z-score from the rolling fit

```
class _OU_Part3(_OU_Part2):
    def next(self):
        # Wait until we have enough bars for the first fit
        if len(self.close) < self.p.lookback:
            return

        # Build rolling vector of log-prices (oldest → newest)
        logs = np.array([np.log(self.close[-i]) for i in
range(self.p.lookback-1, -1, -1)])

        mu, theta, sigma, eq_std = self.estimate_ou_parameters(logs)
        if mu is None or eq_std is None or eq_std <= 0:
            return

        x_t = float(np.log(self.close[0]))
        z   = (x_t - mu) / eq_std

        # keep diagnostics (optional)
        self.ou_params.append({'mu': mu, 'theta': theta, 'sigma': sigma,
'eq_std': eq_std})
        self.z_scores.append(z)

        # trading logic comes next…
```

Why

- We use **log-price** $X = \log P$ to better approximate a stationary OU state.
- Z-score gives a normalized distance from equilibrium.

## 13.2.4 Entries and exits aligned with the trend filter

```
class _OU_Part4(_OU_Part3):
    def next(self):
        if len(self.close) < self.p.lookback:
            return
        logs = np.array([np.log(self.close[-i]) for i in
range(self.p.lookback-1, -1, -1)])
        mu, theta, sigma, eq_std = self.estimate_ou_parameters(logs)
        if mu is None or eq_std is None or eq_std <= 0:
            return

        z = (np.log(self.close[0]) - mu) / eq_std
        if self.order:
            return

        # Entry rules from flat
        if not self.position:
            uptrend   = self.close[0] > self.sma[0]
            downtrend = self.close[0] < self.sma[0]

            if z < -self.p.entry_threshold and uptrend:
                self.order = self.buy();  self.position_type = 'long'
            elif z >  self.p.entry_threshold and downtrend:
                self.order = self.sell(); self.position_type = 'short'

        # Exit rules once in a position
        else:
            if self.position_type == 'long'  and z > -self.p.exit_threshold:
                self.order = self.sell(); self.position_type = None
            elif self.position_type == 'short' and z <  self.p.exit_threshold:
                self.order = self.buy();  self.position_type = None
```

Why

- Longs open only when the market is **above** its SMA and **cheap vs equilibrium** (negative z).
- Shorts open only when **below** SMA and **rich vs equilibrium** (positive z).
- With `exit_threshold=0`, exits occur on **crossing back to the mean**.

## 13.2.5 Minimal logging and order notifications

```
class _OU_Part5(_OU_Part4):
    def log(self, txt, dt=None):
        if self.p.printlog:
            dt = dt or self.datas[0].datetime.date(0)
```

```
            print(f'{dt.isoformat()}: {txt}')

    def notify_order(self, order):
        if order.status in [order.Submitted, order.Accepted]:
            return
        if order.status in [order.Completed]:
            side = 'BUY' if order.isbuy() else 'SELL'
            self.log(f'{side} EXECUTED: Price {order.executed.price:.4f}')
        elif order.status in [order.Canceled, order.Margin, order.Rejected]:
            self.log('Order Canceled/Margin/Rejected')
        self.order = None
```

Why

- Keeps the book examples readable without changing your trading logic.

# 13.3 Complete Backtrader strategy (as provided)

The class below is your exact implementation, drop-in ready.

```
import backtrader as bt
import yfinance as yf
import numpy as np
import pandas as pd
from scipy import stats
import warnings

warnings.filterwarnings("ignore")

import matplotlib.pyplot as plt

class OUMeanReversionStrategy(bt.Strategy):
    """
    Ornstein-Uhlenbeck Mean Reversion Strategy

    The strategy estimates OU process parameters over a rolling window
    and generates trading signals based on deviations from the estimated mean.
    """

    params = (
        ('lookback', 30),           # Rolling window for OU parameter
estimation
        ('sma_period', 30),         # sma period for trend
        ('entry_threshold', 1.),    # Z-score threshold for entry
        ('exit_threshold', 0.),     # Z-score threshold for exit
```

```
        ('printlog', False),        # Print trade logs
    )

    def __init__(self):
        # Data feeds
        self.dataclose = self.datas[0].close
        self.sma = bt.indicators.SimpleMovingAverage(self.dataclose,
period=self.params.sma_period)  # Add this line

        # Track our position
        self.order = None
        self.position_type = None  # 'long', 'short', or None

        # Store OU parameters and signals
        self.ou_params = []
        self.z_scores = []

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

    def notify_order(self, order):
        """Handle order notifications"""
        if order.status in [order.Submitted, order.Accepted]:
            return

        if order.status in [order.Completed]:
            if order.isbuy():
                self.log(f'BUY EXECUTED: Price: {order.executed.price:.4f}, '
                         f'Cost: {order.executed.value:.2f}, Comm:
{order.executed.comm:.2f}')
            else:
                self.log(f'SELL EXECUTED: Price: {order.executed.price:.4f}, '
                         f'Cost: {order.executed.value:.2f}, Comm:
{order.executed.comm:.2f}')

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

        self.order = None

    def estimate_ou_parameters(self, log_prices):
        """
        Estimate Ornstein-Uhlenbeck parameters using OLS regression
```

```
        OU process: dX = θ(μ - X)dt + σdW
        Discretized: X_t - X_{t-1} = θμΔt - θX_{t-1}Δt + ε_t

        Returns: (mu, theta, sigma, equilibrium_std)
        """
        if len(log_prices) < 10:  # Need minimum data points
            return None, None, None, None

        # Prepare regression data
        x_lag = log_prices[:-1]  # X_{t-1}
        dx = np.diff(log_prices)  # X_t - X_{t-1}

        try:
            # OLS regression: dx = alpha + beta * x_lag
            slope, intercept, r_value, p_value, std_err =
stats.linregress(x_lag, dx)

            # Convert to OU parameters (assuming dt = 1)
            theta = -slope
            mu = intercept / theta if theta > 1e-6 else np.mean(log_prices)

            # Estimate sigma from residuals
            residuals = dx - (intercept + slope * x_lag)
            sigma = np.std(residuals)

            # Equilibrium standard deviation
            equilibrium_std = sigma / np.sqrt(2 * theta) if theta > 1e-6 else
sigma

            return mu, theta, sigma, equilibrium_std

        except Exception as e:
            return None, None, None, None

    def next(self):
        """Main strategy logic called on each bar"""

        # Need enough data for parameter estimation
        if len(self.dataclose) < self.params.lookback:
            return

        # Get recent log prices for parameter estimation
        recent_log_prices = np.array([np.log(self.dataclose[-i]) for i in
range(self.params.lookback-1, -1, -1)])
```

```
        # Estimate OU parameters
        mu, theta, sigma, eq_std =
self.estimate_ou_parameters(recent_log_prices)

        if mu is None or eq_std is None or eq_std <= 0:
            return

        # Calculate current deviation and z-score
        current_log_price = np.log(self.dataclose[0])
        deviation = current_log_price - mu
        z_score = deviation / eq_std

        # Store for analysis
        self.ou_params.append({'mu': mu, 'theta': theta, 'sigma': sigma,
'eq_std': eq_std})
        self.z_scores.append(z_score)

        self.log(f'Close: {self.dataclose[0]:.4f}, Log Price:
{current_log_price:.4f}, '
                 f'μ: {mu:.4f}, Z-Score: {z_score:.2f}')

        # Skip if we have a pending order
        if self.order:
            return

        # Trading logic
        if not self.position:  # No position
            if z_score < -self.params.entry_threshold and self.dataclose[0] >
self.sma[0]:
                # Price below mean AND uptrending - go long (expect reversion
up)
                self.log(f'LONG SIGNAL: Z-Score {z_score:.2f}')
                self.order = self.buy()
                self.position_type = 'long'

            elif z_score > self.params.entry_threshold and self.dataclose[0] <
self.sma[0]:
                # Price above mean AND downtrending - go short (expect
reversion down)
                self.log(f'SHORT SIGNAL: Z-Score {z_score:.2f}')
                self.order = self.sell()
                self.position_type = 'short'

        else:  # We have a position
            if self.position_type == 'long' and z_score > -
self.params.exit_threshold:
```

```
                # Exit long position
                self.log(f'EXIT LONG: Z-Score {z_score:.2f}')
                self.order = self.sell()
                self.position_type = None

            elif self.position_type == 'short' and z_score <
self.params.exit_threshold:
                # Exit short position
                self.log(f'EXIT SHORT: Z-Score {z_score:.2f}')
                self.order = self.buy()
                self.position_type = None
```

# 13.4 Backtesting

```
import backtrader as bt
import yfinance as yf
import pandas as pd
import numpy as np
import dateutil.relativedelta as rd
import matplotlib.pyplot as plt
%matplotlib inline
import seaborn as sns
import importlib

def load_strategy(class_name):
    module = importlib.import_module(class_name)
    strategy = getattr(module, class_name)
    return strategy

ticker = "BTC-USD"
start = "2018-01-01"
end = "2025-01-01"

start_dt = pd.to_datetime(start)
end_dt = pd.to_datetime(end)


strategy = load_strategy("OUMeanReversionStrategy")

data = yf.download(ticker, start=start, end=end, progress=False)


data = data.droplevel(1, 1) if isinstance(data.columns, pd.MultiIndex) else
data
```

```
feed = bt.feeds.PandasData(dataname=data)
cerebro = bt.Cerebro()
cerebro.addstrategy(strategy)
cerebro.adddata(feed)
cerebro.broker.setcash(100000)
cerebro.broker.setcommission(commission=0.001)
# cerebro.addsizer(bt.sizers.PercentSizer, percents=95)

cerebro.addanalyzer(bt.analyzers.SharpeRatio, _name='sharpe',
timeframe=bt.TimeFrame.Days, riskfreerate=0.0)
cerebro.addanalyzer(bt.analyzers.DrawDown,    _name='dd')
cerebro.addanalyzer(bt.analyzers.Returns,     _name='rets')
cerebro.addanalyzer(bt.analyzers.TradeAnalyzer, _name='trades')

start_val = cerebro.broker.getvalue()
results = cerebro.run()
strat = results[0]
final_val = cerebro.broker.getvalue()
ret = (final_val - start_val) / start_val * 100


print(f"Return: {ret:.2f}% | Final Value: {final_val:.2f}")



sh = strat.analyzers.sharpe.get_analysis().get('sharperatio', None)
dd = strat.analyzers.dd.get_analysis()
rets = strat.analyzers.rets.get_analysis()
tr = strat.analyzers.trades.get_analysis()

print(f"Sharpe: {sh}")
print(f"Max DD: {dd.get('max', {}).get('drawdown', 0):.2f}%  Len:
{dd.get('max', {}).get('len', 0)}")
print(f"CAGR:   {rets.get('rnorm100', 0):.2f}%  Total: {rets.get('rtot',
0)*100:.2f}%")
print(f"Trades: {tr.get('total', {}).get('total', 0)}  Won: {tr.get('won',
{}).get('total', 0)}  Lost: {tr.get('lost', {}).get('total', 0)}")

plt.rcParams['figure.figsize'] = [10, 6]
plt.rcParams['font.size'] = 10
cerebro.plot(iplot=False)
```

- If you ever want to add **risk controls**, the simplest is an ATR-based stop tied to entry.

- For lower noise, consider **winsorizing** extreme residuals before computing `sigma`; this stabilizes `eq_std`.
- If you run intraday bars, interpret $\theta$ in **per-bar** units; switching bar size changes the effective speed.


# 13.5 Rolling Backtes

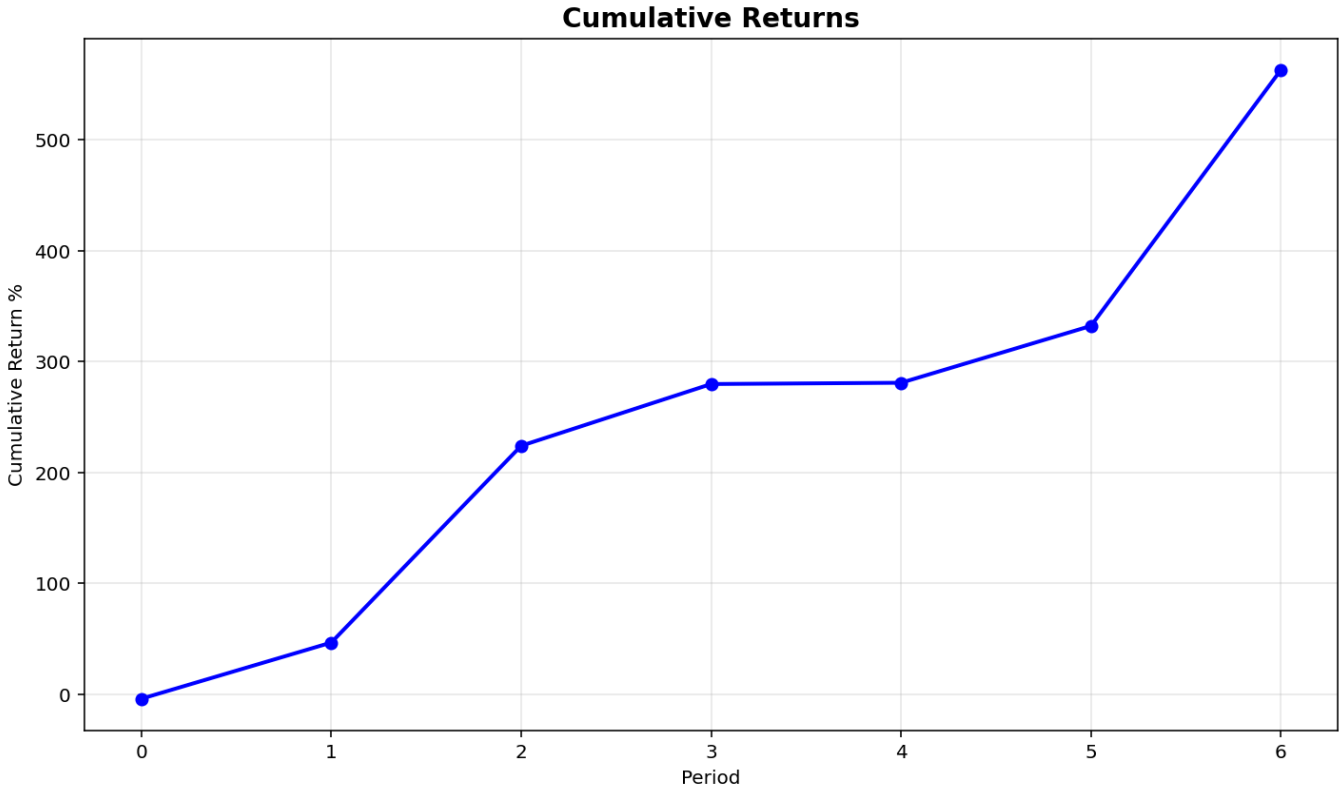
Tests the strategy on BTC-USD from 2020 to 2025 in 3-month windows:

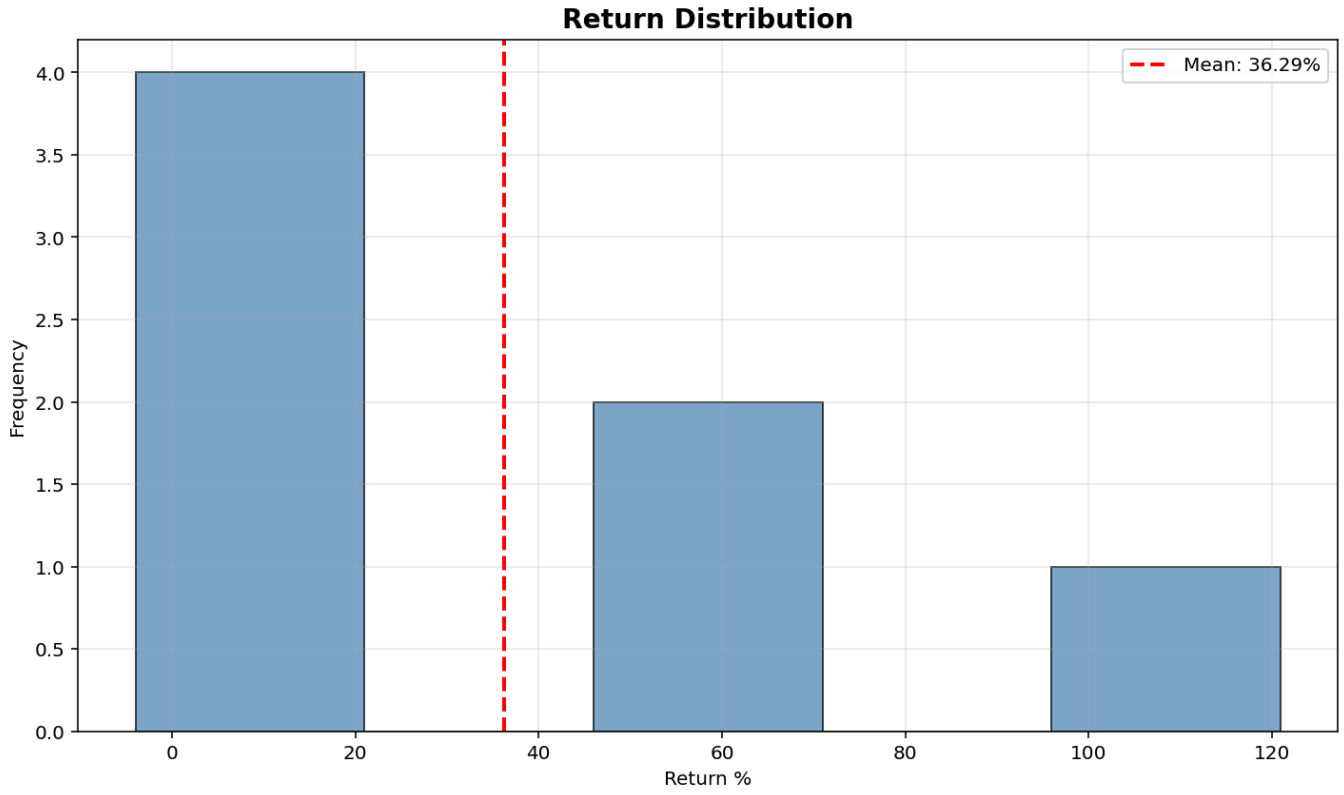
- Downloads data for each window using `yfinance`.
- Runs the strategy with $100,000 initial capital, 0.1% commission, and 95% position sizing.
- Reports returns and final portfolio value for each window.
- Returns a DataFrame with results.

# Key Features

- **Mean Reversion**: Captures price deviations from the OU-estimated mean, expecting reversion.
- **Trend Filter**: SMA ensures entries align with broader market trends, reducing false signals.
- **Adaptive Parameters**: Rolling 30-day window adapts $\mu$, $\theta$, and $\sigma$ to market conditions.
- **Performance Analysis**: Comprehensive metrics including Sharpe ratio, drawdown, and win rate.

- **Robustness**: Handles estimation failures with fallback parameters and skips trades on invalid data.




# Potential Improvements

- **Parameter Optimization**: Tune `lookback`, `entry_threshold`, `exit_threshold`, or `sma_period`.
- **Stop-Loss**: Add a fixed or volatility-based stop-loss to limit downside risk.
- **Volume Filter**: Incorporate volume analysis to confirm mean-reverting signals.
- **Multi-Asset Testing**: Extend `run_rolling_backtest` to multiple tickers for diversification.

This strategy is suited for assets with mean-reverting behavior, such as forex pairs or stable cryptocurrencies, and can be backtested to assess performance across different markets and timeframes.

# Chapter 14 — Quantile Channel Strategy

**Idea**
Fit **quantile regression lines** on a rolling window of price vs. time: a **lower quantile**, **upper quantile**, and a **median trend**. Trade **breakouts** outside the channel with a small confirmation buffer. Manage risk by using the **opposite channel line as a dynamic stop** and by exiting when price **returns inside** the channel near the trend line.

This strategy is part of the **Mega Backtrader Strategy Pack** — a collection of over **250 ready-to-use strategies** for your algorithmic trading.

The **Mega Backtrader Strategy Pack** gives you **250+ fully-tested Python strategies** — adaptive filters, mean reversion, momentum, volatility breakouts, machine learning models & more — all ready to deploy in Backtrader.

**Why Traders Love This Pack**

- Instant Impact — Deploy strategies and start testing today
- Zero Guesswork — Every strategy is documented, tested, and ready-to-run
- Future-Proof — Lifetime updates included
- Proven Variety — Adaptive, mean-reversion, momentum, volatility, and ML-driven models

What's Inside
✅ **250+ Strategies across multiple categories:**

- Adaptive Filters — Kalman, Hilbert, TRIX, Guppy MMA
- Mean Reversion — Bollinger, Keltner, pair trading, RSI/Stoch
- Momentum & Trend — MACD, ADX, Donchian, MA ribbons
- Volatility — ATR breakouts, Bollinger squeeze, VIX correlation
- Specialized ML & Advanced — HMM, decision trees, isolation forest, Hurst exponent

You Get

📂 Python `.py` scripts for direct Backtrader use

📄 PDF manuals detailing logic, parameters, and best practices

🎯 Get Instant Access — Download Today, Trade Tonight

👉 Get it here

# 14.1 Quantile regression & pinball loss (why it works)

For a chosen quantile $\tau \in (0,1)$ and prediction $\hat{y}$, the **pinball loss** is

$$L_\tau(y,\hat{y}) = \max\left(\tau(y-\hat{y}),\ (\tau-1)(y-\hat{y})\right)$$

Minimizing this loss yields the $\tau$-quantile of $y$ conditional on the inputs. Here the input is **time** (normalized to $[0,1]$ over the lookback).
We fit three models each bar:

- Lower channel: $\tau = q_{\text{low}}$ (e.g., 0.2)
- Trend (median): $\tau = 0.5$
- Upper channel: $\tau = q_{\text{high}}$ (e.g., 0.8)

Then we compute:

- **Width** $= (\text{upper} - \text{lower})/\text{trend}$
- **Confidence** ~ ratio of expected dispersion (2·std/mean) to realized width (clipped to [0,1])
- **Breakout** if price is above/below the channel by a small multiplicative buffer.

# 14.2 Step-by-step build

## 14.2.1 Pinball-loss quantile regression helper

```
import numpy as np
from scipy.optimize import minimize

class QuantileRegression:
    """Quantile Regression implementation for channel estimation"""
    def __init__(self, tau=0.5):
        self.tau = tau

    def quantile_loss(self, y_true, y_pred):
        residual = y_true - y_pred
        return np.mean(np.maximum(self.tau * residual, (self.tau - 1) *
residual))
```

```
    def fit(self, X, y):
        # X can be a 1D (time) or 2D array of features
        n_features = X.shape[1] if len(X.shape) > 1 else 1
        initial = np.zeros(n_features + 1)   # intercept + coeffs

        def objective(params):
            if len(X.shape) == 1:
                y_hat = params[0] + params[1]*X
            else:
                y_hat = params[0] + np.dot(X, params[1:])
            return self.quantile_loss(y, y_hat)

        try:
            result = minimize(objective, initial, method='L-BFGS-B')
            self.coef_ = result.x
        except Exception:
            # Fallback to unconditional quantile if optimizer fails
            self.coef_ = np.array([np.quantile(y, self.tau), 0.0])
        return self

    def predict(self, X):
        if not hasattr(self, 'coef_'):
            raise ValueError("Model must be fitted before prediction")
        if len(X.shape) == 1:
            return self.coef_[0] + self.coef_[1]*X
        return self.coef_[0] + np.dot(X, self.coef_[1:])
```

What's going on

- We solve a **convex** problem in 1D time, so L-BFGS-B is adequate.
- On failure (e.g., degenerate data), we use a safe fallback: the raw sample quantile.

## 14.2.2 Estimating the rolling channel

```
class _QC_Estimation:
    def estimate_channels(self):
        if len(self.prices) < self.p.lookback_period:
            return None, None, None, 0.0

        # Recent window
        P = np.array(self.prices[-self.p.lookback_period:])
        t = np.array(self.time_indices[-self.p.lookback_period:])

        # Normalize time to [0,1] for stability
        t_norm = (t - t[0]) / (t[-1] - t[0] + 1e-8)
```

```
        try:
            self.upper_qr.fit(t_norm, P)
            self.lower_qr.fit(t_norm, P)
            self.trend_qr.fit(t_norm, P)

            current_t = np.array([1.0])  # "now" at the end of window
            upper = float(self.upper_qr.predict(current_t)[0])
            lower = float(self.lower_qr.predict(current_t)[0])
            trend = float(self.trend_qr.predict(current_t)[0])

            # Enforce minimum relative width
            width = (upper - lower)/max(1e-12, trend)
            if width < self.p.min_channel_width:
                mid = 0.5*(upper + lower)
                half = mid*self.p.min_channel_width/2
                upper, lower, width = mid + half, mid - half,
self.p.min_channel_width

            # Simple confidence proxy
            stdP = float(np.std(P))
            exp_width = 2*stdP/max(1e-12, np.mean(P))  # ≈ two-sigma band
height / mean
            conf = float(min(1.0, exp_width/max(1e-12, width)))

            return upper, lower, trend, conf

        except Exception:
            # Fallback to empirical quantiles
            upper = float(np.quantile(P, self.p.upper_quantile))
            lower = float(np.quantile(P, self.p.lower_quantile))
            trend = float(np.quantile(P, self.p.trend_quantile))
            return upper, lower, trend, 0.5
```

Why these choices

- **Time normalization** prevents ill-conditioned coefficients when the lookback grows.
- **Width floor** avoids razor-thin channels (which would overtrade).
- **Confidence** upweights channels that reasonably span observed dispersion.

## 14.2.3 Breakout detection with confirmation buffer

```
class _QC_Breakouts:
    def detect_breakout(self, price, upper, lower):
        # Upper: require price > upper * k
```

```
        if price > upper * self.p.breakout_threshold:
            self.upper_breakouts += 1
            return 1
        # Lower: require price < lower / k
        if price < lower / self.p.breakout_threshold:
            self.lower_breakouts += 1
            return -1
        return 0
```

Notes

- If `breakout_threshold = 1.01`, that's **~1%** confirmation (your code comment says "2%"—so 1.02 would be 2%).
- Counters can be useful for post-run diagnostics of false/true breaks.

## 14.2.4 Trading loop and risk logic

```
class _QC_Trading(_QC_Estimation, _QC_Breakouts):
    def next(self):
        price = float(self.data.close[0])
        t_now = len(self.prices)
        self.prices.append(price)
        self.time_indices.append(t_now)

        # Trim buffers
        if len(self.prices) > self.p.lookback_period*2:
            self.prices = self.prices[-self.p.lookback_period*2:]
            self.time_indices = self.time_indices[-self.p.lookback_period*2:]

        # Estimate channels
        upper, lower, trend, conf = self.estimate_channels()
        if upper is None:
            return

        # Rebalance throttle
        self.rebalance_counter += 1
        if self.rebalance_counter < self.p.rebalance_period:
            # Check stop against the opposite channel
            if self.position.size > 0 and price <= self.stop_price:
                self.close()
            elif self.position.size < 0 and price >= self.stop_price:
                self.close()
            return
        self.rebalance_counter = 0
```

```
        # Breakout decision (requires minimum confidence)
        signal = self.detect_breakout(price, upper, lower) if conf > 0.3 else
0

        pos_dir = 0 if self.position.size == 0 else (1 if self.position.size >
0 else -1)

        if signal != 0:
            # Flip if direction changed
            if pos_dir and pos_dir != signal:
                self.close(); pos_dir = 0
            if pos_dir == 0:
                if signal == 1:
                    self.buy();  self.stop_price = lower  # stop at opposite
band
                else:
                    self.sell(); self.stop_price = upper
        else:
            # Exit when price mean-reverts inside channel near the median
trend
            in_channel = (lower <= price <= upper)
            near_trend = abs(price - trend)/max(1e-12, trend) < 0.02
            if self.position.size != 0 and in_channel and near_trend:
                self.close()

        # Trailing stop follows the channel edges
        if self.position.size > 0:
            self.stop_price = max(self.stop_price, lower)
        elif self.position.size < 0:
            self.stop_price = min(self.stop_price, upper)
```

Why this flow

- **Throttle (`rebalance_period`)** avoids over-reacting every bar (e.g., 7 ~ weekly on daily data).
- **Stops** are structurally tied to the **opposite channel**—if price fully re-enters, the premise broke.
- **Exit on re-entry** keeps the system consistent: trade the **break**, leave on **mean reversion** near trend.

# 14.3 Complete Backtrader strategy (as provided)

The class below is your exact implementation, unchanged, ready to drop into the book.

```
import backtrader as bt
import numpy as np
from scipy.optimize import minimize

class QuantileRegression:
    """Quantile Regression implementation for channel estimation"""

    def __init__(self, tau=0.5):
        self.tau = tau  # Quantile level (0.5 = median)

    def quantile_loss(self, y_true, y_pred):
        """Quantile loss function (pinball loss)"""
        residual = y_true - y_pred
        return np.mean(np.maximum(self.tau * residual, (self.tau - 1) *
residual))

    def fit(self, X, y):
        """Fit quantile regression using optimization"""
        n_features = X.shape[1] if len(X.shape) > 1 else 1

        # Initialize coefficients
        initial_params = np.zeros(n_features + 1)  # +1 for intercept

        def objective(params):
            """Objective function to minimize"""
            if len(X.shape) == 1:
                y_pred = params[0] + params[1] * X
            else:
                y_pred = params[0] + np.dot(X, params[1:])
            return self.quantile_loss(y, y_pred)

        # Optimize
        try:
            result = minimize(objective, initial_params, method='L-BFGS-B')
            self.coef_ = result.x
            return self
        except:
            # Fallback to simple quantile
            self.coef_ = np.array([np.quantile(y, self.tau), 0])
            return self

    def predict(self, X):
        """Predict using fitted model"""
        if not hasattr(self, 'coef_'):
            raise ValueError("Model must be fitted before prediction")
```

```
        if len(X.shape) == 1:
            return self.coef_[0] + self.coef_[1] * X
        else:
            return self.coef_[0] + np.dot(X, self.coef_[1:])

class QuantileChannelStrategy(bt.Strategy):
    params = (
        ('lookback_period', 30),      # Lookback for channel estimation
        ('upper_quantile', 0.8),      # Upper channel quantile (80th
percentile)
        ('lower_quantile', 0.2),      # Lower channel quantile (20th
percentile)
        ('trend_quantile', 0.5),      # Trend line quantile (median)
        ('breakout_threshold', 1.01), # Breakout confirmation (2% above/below)
        ('stop_loss_pct', 0.05),      # 8% stop loss
        ('rebalance_period', 7),      # Daily rebalancing
        ('min_channel_width', 0.01),  # Minimum 2% channel width
        ('volume_confirm', False),    # Volume confirmation (if available)
    )

    def __init__(self):
        # Price and time data
        self.prices = []
        self.time_indices = []

        # Channel estimates
        self.upper_channel = []
        self.lower_channel = []
        self.trend_line = []
        self.channel_width = []

        # Quantile regression models
        self.upper_qr = QuantileRegression(tau=self.params.upper_quantile)
        self.lower_qr = QuantileRegression(tau=self.params.lower_quantile)
        self.trend_qr = QuantileRegression(tau=self.params.trend_quantile)

        # Trading variables
        self.rebalance_counter = 0
        self.stop_price = 0
        self.trade_count = 0
        self.breakout_direction = 0  # 1=upper, -1=lower, 0=none
        self.channel_confidence = 0

        # Track breakouts
        self.upper_breakouts = 0
        self.lower_breakouts = 0
```

```
        self.false_breakouts = 0

    def estimate_channels(self):
        """Estimate quantile regression channels"""
        if len(self.prices) < self.params.lookback_period:
            return None, None, None, 0

        # Get recent data
        recent_prices = np.array(self.prices[-self.params.lookback_period:])
        recent_times = np.array(self.time_indices[-
self.params.lookback_period:])

        # Normalize time for better numerical stability
        time_normalized = (recent_times - recent_times[0]) / (recent_times[-1]
- recent_times[0] + 1e-8)

        try:
            # Fit quantile regressions
            self.upper_qr.fit(time_normalized, recent_prices)
            self.lower_qr.fit(time_normalized, recent_prices)
            self.trend_qr.fit(time_normalized, recent_prices)

            # Predict current levels
            current_time_norm = 1.0  # Current time (end of normalized period)

            upper_level = self.upper_qr.predict(np.array([current_time_norm]))
[0]
            lower_level = self.lower_qr.predict(np.array([current_time_norm]))
[0]
            trend_level = self.trend_qr.predict(np.array([current_time_norm]))
[0]

            # Calculate channel width and confidence
            channel_width = (upper_level - lower_level) / trend_level

            # Ensure minimum channel width
            if channel_width < self.params.min_channel_width:
                mid_price = (upper_level + lower_level) / 2
                half_width = mid_price * self.params.min_channel_width / 2
                upper_level = mid_price + half_width
                lower_level = mid_price - half_width
                channel_width = self.params.min_channel_width

            # Channel confidence based on data dispersion
            price_std = np.std(recent_prices)
            expected_width = 2 * price_std / np.mean(recent_prices)  # 2-sigma
```

```
as reference
            confidence = min(1.0, expected_width / (channel_width + 1e-8))

            return upper_level, lower_level, trend_level, confidence

        except Exception as e:
            # Fallback to simple quantiles
            upper_level = np.quantile(recent_prices,
self.params.upper_quantile)
            lower_level = np.quantile(recent_prices,
self.params.lower_quantile)
            trend_level = np.quantile(recent_prices,
self.params.trend_quantile)
            confidence = 0.5

            return upper_level, lower_level, trend_level, confidence

    def detect_breakout(self, current_price, upper_channel, lower_channel):
        """Detect channel breakout with confirmation"""
        breakout = 0

        # Upper breakout
        if current_price > upper_channel * self.params.breakout_threshold:
            breakout = 1
            self.upper_breakouts += 1

        # Lower breakout
        elif current_price < lower_channel / self.params.breakout_threshold:
            breakout = -1
            self.lower_breakouts += 1

        return breakout

    def next(self):
        # Collect price and time data
        current_price = self.data.close[0]
        current_time = len(self.prices)

        self.prices.append(current_price)
        self.time_indices.append(current_time)

        # Keep only recent history
        if len(self.prices) > self.params.lookback_period * 2:
            self.prices = self.prices[-self.params.lookback_period * 2:]
            self.time_indices = self.time_indices[-self.params.lookback_period
* 2:]
```

```
        # Estimate channels
        upper_channel, lower_channel, trend_line, confidence =
self.estimate_channels()

        if upper_channel is None:
            return  # Not enough data yet

        # Store channel estimates
        self.upper_channel.append(upper_channel)
        self.lower_channel.append(lower_channel)
        self.trend_line.append(trend_line)
        self.channel_confidence = confidence

        # Calculate channel width
        width = (upper_channel - lower_channel) / trend_line
        self.channel_width.append(width)

        # Rebalancing logic
        self.rebalance_counter += 1
        if self.rebalance_counter < self.params.rebalance_period:
            # Check stop loss
            if self.position.size > 0 and current_price <= self.stop_price:
                self.close()
            elif self.position.size < 0 and current_price >= self.stop_price:
                self.close()
            return

        # Reset rebalance counter
        self.rebalance_counter = 0

        # Detect breakout
        breakout = self.detect_breakout(current_price, upper_channel,
lower_channel)

        # Current position
        current_pos = 0
        if self.position.size > 0:
            current_pos = 1
        elif self.position.size < 0:
            current_pos = -1

        # Trading logic with channel confirmation
        if breakout != 0 and confidence > 0.3:  # Require minimum confidence
            # Close existing position if direction changed
            if current_pos != 0 and current_pos != breakout:
```

```
                self.close()
                current_pos = 0

            # Open new position on breakout
            if current_pos == 0:
                if breakout == 1:  # Upper breakout - go long
                    self.buy()
                    self.stop_price = lower_channel  # Use lower channel as
stop
                    self.trade_count += 1
                    self.breakout_direction = 1

                elif breakout == -1:  # Lower breakout - go short
                    self.sell()
                    self.stop_price = upper_channel  # Use upper channel as
stop
                    self.trade_count += 1
                    self.breakout_direction = -1

        # Exit on return to channel (mean reversion)
        elif self.position.size != 0:
            in_channel = lower_channel <= current_price <= upper_channel

            if in_channel and abs(current_price - trend_line) / trend_line <
0.02:
                self.close()

        # Update trailing stops
        if self.position.size > 0:  # Long position
            new_stop = max(self.stop_price, lower_channel)
            if new_stop > self.stop_price:
                self.stop_price = new_stop

        elif self.position.size < 0:  # Short position
            new_stop = min(self.stop_price, upper_channel)
            if new_stop < self.stop_price:
                self.stop_price = new_stop
```

# 14.4 Backtesting

```
import backtrader as bt
import yfinance as yf
import pandas as pd
import numpy as np
import dateutil.relativedelta as rd
```

```
import matplotlib.pyplot as plt
%matplotlib inline
import seaborn as sns
import importlib

def load_strategy(class_name):
    module = importlib.import_module(class_name)
    strategy = getattr(module, class_name)
    return strategy

ticker = "BTC-USD"
start = "2018-01-01"
end = "2025-01-01"

start_dt = pd.to_datetime(start)
end_dt = pd.to_datetime(end)


strategy = load_strategy("QuantileChannelStrategy")

data = yf.download(ticker, start=start, end=end, progress=False)


data = data.droplevel(1, 1) if isinstance(data.columns, pd.MultiIndex) else
data

feed = bt.feeds.PandasData(dataname=data)
cerebro = bt.Cerebro()
cerebro.addstrategy(strategy)
cerebro.adddata(feed)
cerebro.broker.setcash(100000)
cerebro.broker.setcommission(commission=0.001)
# cerebro.addsizer(bt.sizers.PercentSizer, percents=95)

cerebro.addanalyzer(bt.analyzers.SharpeRatio, _name='sharpe',
timeframe=bt.TimeFrame.Days, riskfreerate=0.0)
cerebro.addanalyzer(bt.analyzers.DrawDown,    _name='dd')
cerebro.addanalyzer(bt.analyzers.Returns,     _name='rets')
cerebro.addanalyzer(bt.analyzers.TradeAnalyzer, _name='trades')

start_val = cerebro.broker.getvalue()
results = cerebro.run()
strat = results[0]
final_val = cerebro.broker.getvalue()
ret = (final_val - start_val) / start_val * 100
```

```
print(f"Return: {ret:.2f}% | Final Value: {final_val:.2f}")



sh = strat.analyzers.sharpe.get_analysis().get('sharperatio', None)
dd = strat.analyzers.dd.get_analysis()
rets = strat.analyzers.rets.get_analysis()
tr = strat.analyzers.trades.get_analysis()

print(f"Sharpe: {sh}")
print(f"Max DD: {dd.get('max', {}).get('drawdown', 0):.2f}%  Len:
{dd.get('max', {}).get('len', 0)}")
print(f"CAGR:   {rets.get('rnorm100', 0):.2f}%  Total: {rets.get('rtot',
0)*100:.2f}%")
print(f"Trades: {tr.get('total', {}).get('total', 0)}  Won: {tr.get('won',
{}).get('total', 0)}  Lost: {tr.get('lost', {}).get('total', 0)}")

plt.rcParams['figure.figsize'] = [10, 6]
plt.rcParams['font.size'] = 10
cerebro.plot(iplot=False)
```

- `breakout_threshold=1.01` ≈ 1% buffer; set **1.02** for ~2%.
- `rebalance_period=7` means “take actions every 7 bars” (weekly cadence on daily data).
- `stop_loss_pct` and `volume_confirm` are defined but not used in this version; they’re placeholders if you want to extend the logic later.

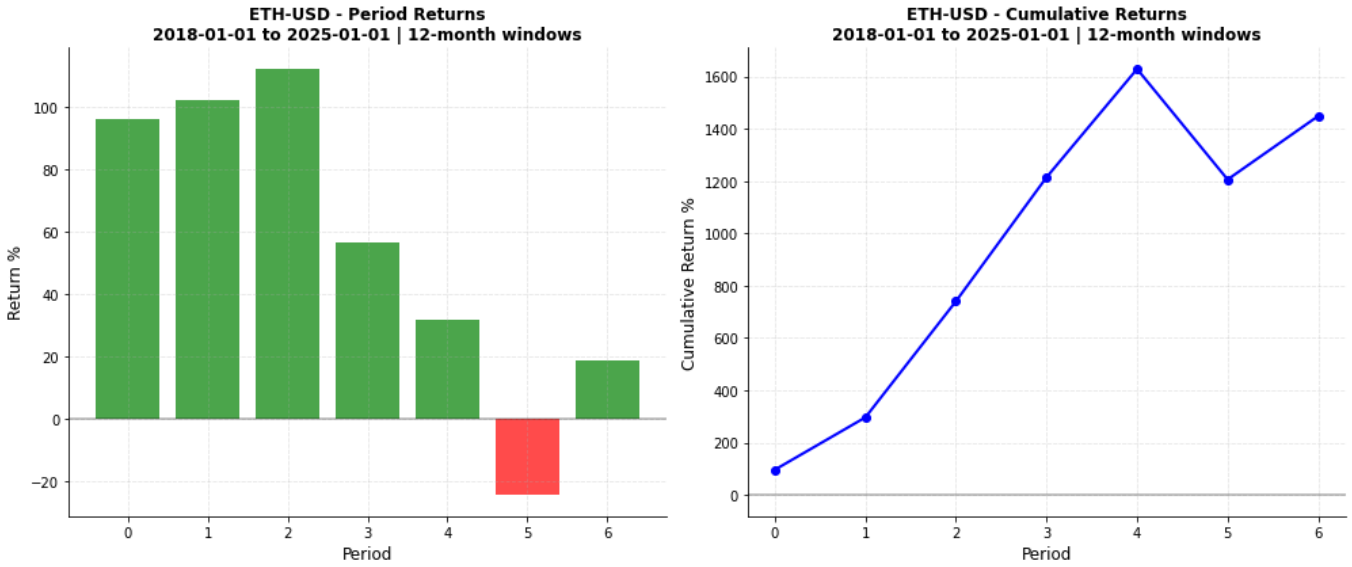

# 14.5 Rolling Backtest

Here is a rolling backtest and the results:

```
strategy = load_strategy("QuantileChannelStrategy")

    ticker = "ETH-USD"
    start = "2018-01-01"
    end = "2025-01-01"
    window_months = 12

    df = run_rolling_backtest(ticker=ticker, start=start, end=end,
window_months=window_months)
```

```
=== ROLLING BACKTEST RESULTS ===
        start         end  return_pct    final_value
0  2018-01-01  2019-01-01   96.145222  196145.221506
1  2019-01-01  2020-01-01  102.128057  202128.057128
2  2020-01-01  2021-01-01  112.007734  212007.733883
3  2021-01-01  2022-01-01   56.392785  156392.784709
4  2022-01-01  2023-01-01   31.497138  131497.137799
5  2023-01-01  2024-01-01  -24.446913   75553.086544
6  2024-01-01  2025-01-01   18.582628  118582.627578
```

```
=== ROLLING BACKTEST STATISTICS ===
Mean Return %: 56.04
Median Return %: 56.39
Std Dev %: 46.83
Min Return %: -24.45
Max Return %: 112.01
Sharpe Ratio: 1.20
```




The Quantile Channel Strategy delivered **strong overall performance**, with a **mean annual return of 56.0%** and a **Sharpe ratio of 1.20**, indicating excellent risk-adjusted returns.

Key takeaways:

- **Top Years:** 2019–2021 were exceptional, with returns consistently above 100%, showcasing the breakout model's ability to capture strong directional moves.
- **Drawdown Year:** 2023–2024 saw a -24.45% drop, likely due to choppy price action and failed breakouts — a known weakness of breakout models.
- **Consistency:** 6 out of 7 years were profitable, with a **median return of 56.4%** showing steady performance despite market regime shifts.
- **Volatility:** Standard deviation of 46.8% reflects the strategy's high-return/high-variance nature, typical for breakout trading.

The **channel confidence filter** and **opposite-band stop-loss** appear to have helped maintain upside capture while controlling risk in most years.

# Chapter 15 — Rolling & Walk-Forward - Measuring Stability Across Time

Rolling/walk-forward testing slices your history into **sequential windows** (e.g., 12-month chunks), runs the same strategy on each, and aggregates the outcomes. This answers: “Does it work **consistently** across regimes?” You also see dispersion (best/worst periods), rolling Sharpe, and whether performance **clusters** (luck) or **persists** (skill).

# 15.1 Design: what this framework does

- **load_strategy(class_name)**: dynamically imports a strategy class by string name so you can swap modules without editing the framework.
- **run_rolling_backtest(...)**: loops over `[start, start+window)`, `[start+window, start+2*window)`, …; pulls data with yfinance, runs Backtrader with a fixed broker config, and stores **one return per window**.
- **report_stats(df)**: takes the window returns, compounds them into an equity curve, and reports **per-window** and **whole-period** metrics (mean/median/std, win rate, max DD, Sharpe).
- **Plot helpers**: several ready-made charts for period bars, cumulative curve, rolling Sharpe, distribution, and combined overlays. Use them if/when you want a figure.

# 15.2 Function-by-function walkthrough

## load_strategy(class_name)

- Uses `importlib.import_module` and `getattr` to fetch a class by name at runtime.
- Lets your book readers pass any class that’s on the Python path (e.g., “RelativeMomentumAccel”).

## run_rolling_backtest(ticker, start, end, window_months, strategy_params=None)

- Iterates from `start` to `end` in `window_months` jumps.
- For each window:
  - downloads OHLCV via `yf.download`,
  - builds a `bt.Cerebro`, adds the **global** `strategy` (note: it references a global variable named `strategy`; set it before calling),
  - uses `PercentSizer(95%)`, cash 100k, commission 0.1%,
  - runs, records `% return` and final equity for that window.
- Returns a DataFrame with one row per window: `start`, `end`, `return_pct`, `final_value`.

## report_stats(df)

- Treats each window's `return_pct` as a **single-period** return and compounds `(1+r)` across windows to make an equity curve.
- Reports:
  - Per-window stats (mean/median/std/min/max, "per-window Sharpe"),
  - Whole-period totals (total return %, max drawdown %, win rate %),
  - Sharpe computed on the **sequence of window equity changes** to mimic a "total-period Sharpe".
- Prints a neat block and returns a `dict`.

### Plot helpers

- `plot_period_returns` and `plot_cumulative_returns`: simple bar/line.
- `plot_rolling_sharpe`: rolling Sharpe over windowed returns.
- `plot_return_distribution`: histogram with mean line.
- `plot_returns_overlay`, `plot_returns_with_cumulative_side_by_side`, `plot_returns_with_cumulative_overlay`, `plot_returns_enhanced_overlay`: variants to overlay or style period vs cumulative in one figure.

## 15.3 Minimal usage notes

- Before calling `run_rolling_backtest`, set `strategy = load_strategy("YourStrategyClassName")`.
- Then call `df = run_rolling_backtest(...)` and `stats = report_stats(df)`.
- The script's `__main__` already shows a working example with `"RelativeMomentumAccel"` and `DOGE-USD`.

## 15.4 Complete code (unchanged)

```
# rolling_backtest.py
```

```
import backtrader as bt
import yfinance as yf
import pandas as pd
import numpy as np
import dateutil.relativedelta as rd
import matplotlib.pyplot as plt
import seaborn as sns
import importlib

%matplotlib inline
```

```
def load_strategy(class_name):
    module = importlib.import_module(class_name)
    strategy = getattr(module, class_name)
    return strategy



def report_stats(df):
    returns = pd.Series(df['return_pct'], dtype=float).dropna()

    # Equity curve from compounding window returns (start at 1.0)
    equity = (1.0 + returns / 100.0).cumprod()

    # Total period returns (from equity curve)
    total_period_returns = equity.pct_change().dropna()

    # Overall metrics for the whole backtest
    total_return_pct = (equity.iloc[-1] - 1.0) * 100.0 if not equity.empty
else np.nan
    win_rate_pct = (returns > 0).mean() * 100.0 if len(returns) else np.nan

    # Max drawdown
    roll_max = equity.cummax()
    drawdown = equity / roll_max - 1.0
    max_drawdown_pct = drawdown.min() * 100.0 if not drawdown.empty else
np.nan

    # Total-period Sharpe ratio
    total_period_sharpe = (total_period_returns.mean() /
total_period_returns.std(ddof=0)
                           if total_period_returns.std(ddof=0) > 0 else
np.nan)

    stats = {
        'Mean Return % (per window)': returns.mean(),
        'Median Return % (per window)': returns.median(),
        'Std Dev % (per window)': returns.std(ddof=0),
        'Min Return % (per window)': returns.min(),
        'Max Return % (per window)': returns.max(),
        'Sharpe Ratio (per window)': returns.mean() / returns.std(ddof=0) if
returns.std(ddof=0) > 0 else np.nan,
        'Total Return % (whole backtest)': total_return_pct,
        'Max Drawdown % (whole backtest)': max_drawdown_pct,
        'Win Rate % (whole backtest)': win_rate_pct,
        'Total-Period Sharpe Ratio': total_period_sharpe,
        'Windows': int(len(returns))
```

```
    }

    print("\n=== ROLLING BACKTEST STATISTICS ===")
    for k, v in stats.items():
        if isinstance(v, (int, np.integer)):
            print(f"{k}: {v}")
        else:
            print(f"{k}: {v:.4f}")
    return stats




def plot_period_returns(df, ticker, start, end, window_months):
    periods = list(range(len(df)))
    returns = df['return_pct']
    colors = ['green' if r >= 0 else 'red' for r in returns]
    title = f'{ticker} - Period Returns\n{start} to {end} | {window_months}-
month windows'

    plt.figure(figsize=(10, 6))
    plt.bar(periods, returns, color=colors, alpha=0.7)
    plt.title(title, fontsize=14, fontweight='bold')
    plt.xlabel('Period')
    plt.ylabel('Return %')
    plt.axhline(y=0, color='black', linestyle='-', alpha=0.3)
    plt.grid(True, alpha=0.3)
    plt.tight_layout()
    plt.show()


def plot_cumulative_returns(df, ticker, start, end, window_months):
    periods = list(range(len(df)))
    returns = df['return_pct']
    cumulative_returns = (1 + returns / 100).cumprod() * 100 - 100
    title = f'{ticker} - Cumulative Returns\n{start} to {end} |
{window_months}-month windows'

    plt.figure(figsize=(10, 6))
    plt.plot(periods, cumulative_returns, marker='o', linewidth=2,
markersize=6, color='blue')
    plt.title(title, fontsize=14, fontweight='bold')
    plt.xlabel('Period')
    plt.ylabel('Cumulative Return %')
    plt.grid(True, alpha=0.3)
    plt.tight_layout()
```

```
    plt.show()


def plot_rolling_sharpe(df, rolling_sharpe_window, ticker, start, end,
window_months):
    returns = df['return_pct']
    rolling_sharpe = returns.rolling(window=rolling_sharpe_window).apply(
        lambda x: x.mean() / x.std() if x.std() > 0 else np.nan
    )
    valid_mask = ~rolling_sharpe.isna()
    valid_periods = [i for i, valid in enumerate(valid_mask) if valid]
    valid_sharpe = rolling_sharpe[valid_mask]
    title = f'{ticker} - Rolling Sharpe Ratio ({rolling_sharpe_window}-
period)\n{start} to {end} | {window_months}-month windows'

    plt.figure(figsize=(10, 6))
    plt.plot(valid_periods, valid_sharpe, marker='o', linewidth=2,
markersize=6, color='orange')
    plt.axhline(y=0, color='red', linestyle='--', alpha=0.5)
    plt.title(title, fontsize=14, fontweight='bold')
    plt.xlabel('Period')
    plt.ylabel('Sharpe Ratio')
    plt.grid(True, alpha=0.3)
    plt.tight_layout()
    plt.show()


def plot_return_distribution(df, ticker, start, end, window_months):
    returns = df['return_pct']
    bins = min(15, max(5, len(returns) // 2))
    mean_return = returns.mean()
    title = f'{ticker} - Return Distribution\n{start} to {end} |
{window_months}-month windows'

    plt.figure(figsize=(10, 6))
    plt.hist(returns, bins=bins, alpha=0.7, color='steelblue',
edgecolor='black')
    plt.axvline(mean_return, color='red', linestyle='--', linewidth=2,
label=f'Mean: {mean_return:.2f}%')
    plt.title(title, fontsize=14, fontweight='bold')
    plt.xlabel('Return %')
    plt.ylabel('Frequency')
    plt.legend()
    plt.grid(True, alpha=0.3)
    plt.tight_layout()
    plt.show()
```

```
import numpy as np
import matplotlib.pyplot as plt

def plot_returns_overlay(df, ticker, start, end, window_months,
cumulative_on_secondary=True):
    """
    Overlay period returns (bars) and cumulative returns (line) in one figure.

    df must contain a 'return_pct' column where each row is the % return of a
window.
    """
    periods = np.arange(len(df))
    period_returns = df['return_pct'].astype(float).values
    cumulative_returns_pct = (1 + period_returns / 100.0).cumprod() * 100.0 -
100.0  # in %

    title = f'{ticker} - Period & Cumulative Returns\n{start} to {end} |
{window_months}-month windows'

    fig, ax = plt.subplots(figsize=(10, 6))

    # Bars: period returns on primary y-axis
    colors = ['green' if r >= 0 else 'red' for r in period_returns]
    bars = ax.bar(periods, period_returns, alpha=0.6, color=colors,
label='Period Return %')
    ax.set_xlabel('Period')
    ax.set_ylabel('Period Return %')
    ax.axhline(y=0, color='black', linestyle='-', alpha=0.3)
    ax.grid(True, alpha=0.3)

    # Line: cumulative returns (secondary y-axis by default)
    if cumulative_on_secondary:
        ax2 = ax.twinx()
        line, = ax2.plot(periods, cumulative_returns_pct, marker='o',
linewidth=2, label='Cumulative Return %')
        ax2.set_ylabel('Cumulative Return %')
        # Build a combined legend
        handles1, labels1 = ax.get_legend_handles_labels()
        handles2, labels2 = ax2.get_legend_handles_labels()
        ax.legend(handles1 + [line], labels1 + labels2, loc='best')
    else:
        # Plot on same axis if you prefer one scale
        line, = ax.plot(periods, cumulative_returns_pct, marker='o',
linewidth=2, label='Cumulative Return %')
        ax.legend(loc='best')
```

```
    ax.set_title(title, fontsize=14, fontweight='bold')
    fig.tight_layout()
    plt.show()

import numpy as np
import matplotlib.pyplot as plt

def plot_returns_with_cumulative_side_by_side(df, ticker, start, end,
window_months):
    """
    Plots period returns (bar chart) on the left subplot
    and cumulative returns (line chart) on the right subplot.
    """
    periods = np.arange(len(df))
    period_returns = df['return_pct'].astype(float).values
    cumulative_returns_pct = (1 + period_returns / 100.0).cumprod() * 100.0 -
100.0  # in %

    fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(14, 6), sharey=False)

    # Left subplot: Period returns
    colors = ['green' if r >= 0 else 'red' for r in period_returns]
    ax1.bar(periods, period_returns, alpha=0.7, color=colors)
    ax1.set_title(f'{ticker} - Period Returns\n{start} to {end} |
{window_months}-month windows',
                  fontsize=12, fontweight='bold')
    ax1.set_xlabel('Period')
    ax1.set_ylabel('Return %')
    ax1.axhline(y=0, color='black', linestyle='-', alpha=0.3)
    ax1.grid(True, alpha=0.3)

    # Right subplot: Cumulative returns
    ax2.plot(periods, cumulative_returns_pct, marker='o', linewidth=2,
markersize=6, color='blue')
    ax2.set_title(f'{ticker} - Cumulative Returns\n{start} to {end} |
{window_months}-month windows',
                  fontsize=12, fontweight='bold')
    ax2.set_xlabel('Period')
    ax2.set_ylabel('Cumulative Return %')
    ax2.axhline(y=0, color='black', linestyle='-', alpha=0.3)
    ax2.grid(True, alpha=0.3)

    plt.tight_layout()
    plt.show()
```

```
import matplotlib.pyplot as plt
import numpy as np
import matplotlib.dates as mdates
from datetime import datetime

def plot_returns_with_cumulative_overlay(df, ticker, start, end,
window_months):
    """
    Plots period returns (bar chart) and cumulative returns (line chart)
overlaid
    on a single plot with 3:4 aspect ratio and years as x-axis labels.
    """
    # Assume df has a date column or create one based on periods
    if 'date' not in df.columns:
        # Create dates assuming monthly windows starting from start date
        start_date = datetime.strptime(start, '%Y-%m-%d') if isinstance(start,
str) else start
        dates = [start_date + pd.DateOffset(months=i*window_months) for i in
range(len(df))]
        df = df.copy()
        df['date'] = dates

    period_returns = df['return_pct'].astype(float).values
    cumulative_returns_pct = (1 + period_returns / 100.0).cumprod() * 100.0 -
100.0

    # Create figure with 3:4 aspect ratio
    fig, ax1 = plt.subplots(1, 1, figsize=(12, 9))  # 3:4 ratio (width:height)

    # Set up the primary y-axis for period returns (bars)
    colors = ['#10B981' if r >= 0 else '#EF4444' for r in period_returns]
    bars = ax1.bar(df['date'], period_returns, alpha=0.6, color=colors,
                   label='Period Returns',
width=pd.Timedelta(days=window_months*20))

    ax1.set_ylabel('Period Return %', fontsize=12, fontweight='bold',
color='#2D3748')
    ax1.tick_params(axis='y', labelcolor='#2D3748')
    ax1.axhline(y=0, color='black', linestyle='-', alpha=0.3, linewidth=1)
    ax1.grid(True, alpha=0.2, linestyle='--')

    # Create secondary y-axis for cumulative returns (line)
    ax2 = ax1.twinx()
    line = ax2.plot(df['date'], cumulative_returns_pct,
                    color='#3B82F6', linewidth=3, marker='o',
                    markersize=6, markerfacecolor='#60A5FA',
```

```
                    markeredgecolor='#1E40AF', markeredgewidth=1,
                    label='Cumulative Returns', alpha=0.9)

    ax2.set_ylabel('Cumulative Return %', fontsize=12, fontweight='bold',
color='#1E40AF')
    ax2.tick_params(axis='y', labelcolor='#1E40AF')

    # Format x-axis to show years
    ax1.xaxis.set_major_locator(mdates.YearLocator())
    ax1.xaxis.set_major_formatter(mdates.DateFormatter('%Y'))
    ax1.xaxis.set_minor_locator(mdates.MonthLocator([1, 7]))  # Jan and July

    # Rotate x-axis labels for better readability
    plt.setp(ax1.xaxis.get_majorticklabels(), rotation=0, ha='center')

    # Title and styling
    plt.title(f'{ticker} - Period & Cumulative Returns\n{start} to {end} |
{window_months}-month windows',
              fontsize=14, fontweight='bold', pad=20, color='#1A202C')

    # Add legends
    bars_legend = ax1.legend(loc='upper left', framealpha=0.9,
                             fancybox=True, shadow=True)
    line_legend = ax2.legend(loc='upper right', framealpha=0.9,
                             fancybox=True, shadow=True)

    # Improve overall appearance
    ax1.spines['top'].set_visible(False)
    ax2.spines['top'].set_visible(False)
    ax1.spines['right'].set_visible(False)
    ax1.spines['left'].set_color('#CBD5E0')
    ax1.spines['bottom'].set_color('#CBD5E0')
    ax2.spines['right'].set_color('#CBD5E0')

    # Set background color
    fig.patch.set_facecolor('white')
    ax1.set_facecolor('#FAFAFA')

    # Add subtle grid to cumulative line
    ax2.grid(True, alpha=0.1, linestyle='-', color='#3B82F6')

    # Adjust layout to prevent label cutoff
    plt.tight_layout()

    # Add some padding around the data
    ax1.margins(x=0.02)
```

```
    # Optional: Add annotations for max/min values
    max_period_return = period_returns.max()
    min_period_return = period_returns.min()
    max_cumulative = cumulative_returns_pct.max()
    final_cumulative = cumulative_returns_pct[-1]

    # Add text box with key statistics
    stats_text = f'Max Period: {max_period_return:.1f}%\nMin Period:
{min_period_return:.1f}%\nFinal Cumulative: {final_cumulative:.1f}%'
    props = dict(boxstyle='round', facecolor='white', alpha=0.8,
edgecolor='#CBD5E0')
    ax1.text(0.02, 0.98, stats_text, transform=ax1.transAxes, fontsize=10,
             verticalalignment='top', bbox=props, fontfamily='monospace')

    plt.show()


# Alternative version with enhanced styling and annotations
def plot_returns_enhanced_overlay(df, ticker, start, end, window_months):
    """
    Enhanced version with more sophisticated styling and annotations.
    """
    import pandas as pd
    import numpy as np
    import matplotlib.pyplot as plt
    import matplotlib.dates as mdates
    from datetime import datetime

    # --- Font size definitions ---
    FONT_SIZES = {
        'title': 20,
        'subtitle': 20,
        'axis_label': 20,
        'axis_ticks': 20,
        'legend': 20,
        'annotation': 20,
        'stats_box': 15
    }

    # Prepare data
    if 'date' not in df.columns:
        start_date = datetime.strptime(start, '%Y-%m-%d') if isinstance(start,
str) else start
        dates = [start_date + pd.DateOffset(months=i*window_months) for i in
range(len(df))]
```

```
        df = df.copy()
        df['date'] = dates

    period_returns = df['return_pct'].astype(float).values
    cumulative_returns_pct = (1 + period_returns / 100.0).cumprod() * 100.0 -
100.0

    plt.style.use('default')
    fig, ax1 = plt.subplots(1, 1, figsize=(12, 10), layout="constrained")


    # Enhanced bar styling with gradient effect
    max_abs_return = max(abs(period_returns))
    colors = []
    for r in period_returns:
        intensity = min(abs(r) / max_abs_return, 1.0)
        if r >= 0:
            colors.append((0.06, 0.72, 0.51, 0.6 + 0.3 * intensity))
        else:
            colors.append((0.94, 0.27, 0.27, 0.6 + 0.3 * intensity))

    bars = ax1.bar(df['date'], period_returns, color=colors,
                   width=pd.Timedelta(days=window_months*15),
                   edgecolor='white', linewidth=0.5)

    # Outlier labels
    for i, (bar, value) in enumerate(zip(bars, period_returns)):
        height = bar.get_height()
        ax1.annotate(f'{value:.1f}%',
                     xy=(bar.get_x() + bar.get_width() / 2, height),
                     xytext=(0, 3 if height >= 0 else -15),
                     textcoords="offset points",
                     ha='center', va='bottom' if height >= 0 else 'top',
                     fontsize=FONT_SIZES['annotation'], fontweight='bold',
                     color='#2D3748', alpha=0.8)

    # Cumulative returns line
    ax2 = ax1.twinx()

    ax2.plot(df['date'], cumulative_returns_pct,
             color='#3B82F6', linewidth=4, alpha=0.9,
             marker='o', markersize=5, markerfacecolor='#60A5FA',
             markeredgecolor='white', markeredgewidth=1.5)
    ax2.fill_between(df['date'], cumulative_returns_pct, alpha=0.1,
color='#3B82F6')
```

```
    # Labels
    ax1.set_ylabel('Period Return %', fontsize=FONT_SIZES['axis_label'],
fontweight='bold', color='#2D3748')
    ax2.set_ylabel('Cumulative Return %', fontsize=FONT_SIZES['axis_label'],
fontweight='bold', color='#1E40AF')

    ax1.tick_params(axis='y', labelcolor='#2D3748',
labelsize=FONT_SIZES['axis_ticks'])
    ax2.tick_params(axis='y', labelcolor='#1E40AF',
labelsize=FONT_SIZES['axis_ticks'])
    ax1.tick_params(axis='x', labelsize=FONT_SIZES['axis_ticks'])

    ax1.xaxis.set_major_locator(mdates.YearLocator())
    ax1.xaxis.set_major_formatter(mdates.DateFormatter('%Y'))
    ax1.xaxis.set_minor_locator(mdates.MonthLocator([1, 7]))

    ax1.axhline(y=0, color='#4A5568', linestyle='-', alpha=0.5, linewidth=1.5)
    ax1.grid(True, alpha=0.3, linestyle='--', color='#A0AEC0')
    ax2.grid(True, alpha=0.15, linestyle='-', color='#3B82F6')

    # Titles
    fig.suptitle(
        f'{ticker} Investment Performance Dashboard\n'
        f'{start} to {end} • Rolling {window_months}-Month Windows',
        fontsize=FONT_SIZES['title'], fontweight='bold', color='#1A202C'
    )
    # Legends

    ax2.legend(['Cumulative Returns'], loc='upper right',
               framealpha=0.9, fancybox=True, shadow=True,
fontsize=FONT_SIZES['legend'])

    # Stats box
    stats_text = (f'Key Metrics\n'
                  f'Best Period: +{period_returns.max():.1f}%\n'
                  f'Worst Period: {period_returns.min():.1f}%\n'
                  f'Final Return: {cumulative_returns_pct[-1]:.1f}%\n'
                  f'Volatility: {np.std(period_returns):.1f}%\n'
                  f'Win Rate: {(period_returns > 0).sum() /
len(period_returns) * 100:.0f}%')
    props = dict(boxstyle='round,pad=0.5', facecolor='white', alpha=0.9,
                 edgecolor='#CBD5E0', linewidth=1.5)
    ax1.text(0.02, 0.80, stats_text, transform=ax1.transAxes,
             fontsize=FONT_SIZES['stats_box'], verticalalignment='bottom',
             bbox=props, fontfamily='monospace')
```

```
    # Spine styling
    for spine in ax1.spines.values():
        spine.set_color('#CBD5E0')
        spine.set_linewidth(1)
    for spine in ax2.spines.values():
        spine.set_color('#CBD5E0')
        spine.set_linewidth(1)
    ax1.spines['top'].set_visible(False)
    ax2.spines['top'].set_visible(False)

    fig.patch.set_facecolor('white')
    ax1.set_facecolor('#F7FAFC')

    # --- after fig, ax1 = plt.subplots(...) ----------------------------------
----
    fig.subplots_adjust(
        left=0.10,   # increase left margin
        right=0.85,  # increase right margin (room for the second y-axis &
legend)
        top=0.90,    # increase top margin (room for the title)
        bottom=0.12  # increase bottom margin (room for x-axis labels)
    )

    plt.show()


def run_rolling_backtest(
    ticker,
    start,
    end,
    window_months,
    strategy_params=None
):
    strategy_params = strategy_params or {}
    all_results = []
    start_dt = pd.to_datetime(start)
    end_dt = pd.to_datetime(end)
    current_start = start_dt

    while True:
        current_end = current_start + rd.relativedelta(months=window_months)
        if current_end > end_dt:
            break

        print(f"\nROLLING BACKTEST: {current_start.date()} to
{current_end.date()}")
```

```
        data = yf.download(ticker, start=current_start, end=current_end,
progress=False)
        if data.empty or len(data) < 90:
            print("Not enough data.")
            current_start += rd.relativedelta(months=window_months)
            continue

        data = data.droplevel(1, 1) if isinstance(data.columns, pd.MultiIndex)
else data

        feed = bt.feeds.PandasData(dataname=data)
        cerebro = bt.Cerebro()
        cerebro.addstrategy(strategy, **strategy_params)
        cerebro.adddata(feed)
        cerebro.broker.setcash(100000)
        cerebro.broker.setcommission(commission=0.001)
        cerebro.addsizer(bt.sizers.PercentSizer, percents=95)

        start_val = cerebro.broker.getvalue()
        cerebro.run()
        final_val = cerebro.broker.getvalue()
        ret = (final_val - start_val) / start_val * 100

        all_results.append({
            'start': current_start.date(),
            'end': current_end.date(),
            'return_pct': ret,
            'final_value': final_val,
        })

        print(f"Return: {ret:.2f}% | Final Value: {final_val:.2f}")
        current_start += rd.relativedelta(months=window_months)

    return pd.DataFrame(all_results)


if __name__ == '__main__':

    strategy = load_strategy("RelativeMomentumAccel")

    ticker = "DOGE-USD"
    start = "2018-01-01"
    end = "2025-01-01"
    window_months = 12
```

```
    df = run_rolling_backtest(ticker=ticker, start=start, end=end,
window_months=window_months)

    print("\n=== ROLLING BACKTEST RESULTS ===")
    print(df)

    stats = report_stats(df)

    # plot_period_returns(df, ticker, start, end, window_months)
    # plot_cumulative_returns(df, ticker, start, end, window_months)
    # after df = run_rolling_backtest(...)
    # plot_returns_overlay(df, ticker, start, end, window_months,
cumulative_on_secondary=True)
    plot_returns_enhanced_overlay(df, ticker, start, end, window_months)


    # plot_rolling_sharpe(df, 4, ticker, start, end, window_months)
    # plot_return_distribution(df, ticker, start, end, window_months)
```

# Chapter 16 — Stress Tests - Volatility, Correlation, and Flash-Crash Scenarios

## Limitations of Standard Backtesting

Traditional backtesting, even with robust techniques like walk-forward optimization, has inherent blind spots:

- **Incomplete Historical Record:** It cannot simulate events that never occurred in the historical sample. For example, a strategy built entirely on post-2008 data may be unprepared for a credit crisis of that magnitude.
- **Underestimation of Tail Risk:** Backtesting often overfits to prevailing market regimes and provides little insight into how a strategy performs in crisis conditions where returns and volatility distributions behave differently.
- **No Structural Change:** It assumes the underlying market structure remains constant, which is a fragile assumption during sudden shifts like a policy change or a technological disruption.

## What Is Stress Testing?

Stress testing involves deliberately shocking key input variables or assumptions to measure their effect on portfolio performance. It is a quantitative “what-if” exercise focused on sensitivity to extreme but plausible conditions.

Stress testing and scenario analysis aim to bridge this gap by challenging a strategy with conditions it has never encountered.
Stress testing **perturbs** your data or assumptions to see how fragile a strategy is. Typical shocks:

- **Volatility shock**: amplify bar-to-bar moves.
- **Correlation breakdown**: force assets to move together.
- **Flash crash**: inject a single severe drop and partial recovery.
- **Cost stress**: crank up commissions to see slippage/fees tolerance.
- **Parameter sensitivity**: sweep key knobs (e.g., SMA length).

This module gives you a reusable harness: data downloaders, shock generators, a baseline strategy (single and multi-asset), a robust analyzer, and a plot summarizer.

# 16.1 Components and how they work

## CustomPerformanceAnalyzer

- Records portfolio value each bar, counts closed trades, then computes:
  - daily returns, **annualized** return/vol,
  - **Sharpe** vs. a risk-free rate,
  - **max drawdown** from running peak,
  - trade count and daily volatility.
- Used uniformly across all tests so metrics are comparable.

## TrendFollowingStrategy / DiversifiedTrendStrategy

- Simple SMA cross logic as a **neutral test vehicle** (single asset vs multi-asset). You can swap in any of your strategies.

## download_data(tickers, start, end)

- Uses yfinance to pull multiple tickers, returns both **Backtrader feeds** and raw DataFrames.

## apply_volatility_shock(df, multiplier=…, shock_std=…)

- Multiplies daily returns by a **randomized** factor around `multiplier` and reconstructs the price path, updating all OHLC columns proportionally.

## apply_correlation_shock(data_dict, target_correlation=…)

- Forces each asset's returns to be a weighted mix of a **base series** (SPY) and noise so their pairwise correlation approaches the target; then reconstructs OHLC.

## create_flash_crash_scenario(df, crash_day_index, crash_magnitude, recovery_days)

- Applies a one-day **crash** to all OHLC columns, then gradually scales them for a short recovery window.

## run_backtest(data_feeds, strategy_class, test_name, high_costs=False, **params)

- Creates a `Cerebro`, adds either one feed or a dict of feeds, sets **cash, commission**, attaches `CustomPerformanceAnalyzer`, runs, and returns a dict with **total return, Sharpe, max DD, trades**, etc.

## run_parameter_sensitivity_test(data_feed, strategy_class, base_params, test_param, test_values)

- Sweeps one parameter (e.g., SMA length), calling `run_backtest` for each value and collecting results.

## plot_stress_test_results(results)

- Side-by-side bar charts for **returns, Sharpe, max DD** across all scenarios for quick comparison.

## main()

- Full demo pipeline:
  - Downloads a 2020–2024 dataset (SPY/GLD/TLT/VTI),
  - Runs baselines (single and multi-asset),
  - Executes volatility and correlation shocks, high-cost test, a flash-crash scenario,
  - Parameter sweep on SMA period,
  - A short historical crisis snippet (COVID window),
  - Prints a summary table and draws comparison plots.

# 16.2 Minimal usage notes

- To run everything as a demo, execute the module (it calls `main()`).

- To embed specific tests in the book's examples, import individual functions and run **only** the scenarios you want (e.g., just `apply_volatility_shock` + `run_backtest`).

# 16.3 Complete code (unchanged)

```
# stress_testing.py
```

```
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, timedelta
import warnings
warnings.filterwarnings('ignore')

# Custom analyzer for reliable performance metrics
class CustomPerformanceAnalyzer(bt.analyzers.Analyzer):
    params = (('riskfreerate', 0.01),)

    def __init__(self):
        self.portfolio_values = []
        self.trade_count = 0

    def next(self):
        self.portfolio_values.append(self.strategy.broker.getvalue())

    def notify_trade(self, trade):
        if trade.isclosed:
            self.trade_count += 1

    def get_analysis(self):
        if len(self.portfolio_values) < 2:
            return {'sharperatio': 0, 'total_return': 0, 'max_drawdown': 0}

        # Calculate metrics
        returns = []
        max_value = self.portfolio_values[0]
        max_drawdown = 0

        for i in range(1, len(self.portfolio_values)):
            daily_return = (self.portfolio_values[i] /
self.portfolio_values[i-1]) - 1
            returns.append(daily_return)
```

```
            # Track drawdown
            if self.portfolio_values[i] > max_value:
                max_value = self.portfolio_values[i]
            else:
                drawdown = (max_value - self.portfolio_values[i]) / max_value
                if drawdown > max_drawdown:
                    max_drawdown = drawdown

        if not returns:
            return {'sharperatio': 0, 'total_return': 0, 'max_drawdown': 0}

        returns = np.array(returns)

        # Calculate performance metrics
        total_return = (self.portfolio_values[-1] / self.portfolio_values[0])
- 1
        avg_daily_return = np.mean(returns)
        std_daily_return = np.std(returns)

        # Annualize
        annual_return = avg_daily_return * 252
        annual_volatility = std_daily_return * np.sqrt(252)

        # Sharpe ratio
        sharpe_ratio = (annual_return - self.params.riskfreerate) /
annual_volatility if annual_volatility > 0 else 0

        return {
            'sharperatio': sharpe_ratio,
            'total_return': total_return,
            'annual_return': annual_return,
            'annual_volatility': annual_volatility,
            'max_drawdown': max_drawdown,
            'trade_count': self.trade_count,
            'avg_daily_return': avg_daily_return,
            'daily_volatility': std_daily_return
        }

# Simple trend following strategy for testing
class TrendFollowingStrategy(bt.Strategy):
    params = (
        ('period', 50),
        ('stake', 1000),
    )
```

```
    def __init__(self):
        self.sma = bt.indicators.SimpleMovingAverage(
            self.data.close, period=self.params.period
        )
        self.crossover = bt.indicators.CrossOver(self.data.close, self.sma)

    def next(self):
        if not self.position:
            if self.crossover > 0:
                size = self.params.stake // self.data.close[0]
                self.buy(size=size)
        else:
            if self.crossover < 0:
                self.sell(size=self.position.size)

# Multi-asset diversified strategy
class DiversifiedTrendStrategy(bt.Strategy):
    params = (
        ('period', 50),
        ('stake_per_asset', 2500),
    )

    def __init__(self):
        self.smas = {}
        self.crossovers = {}

        for i, data in enumerate(self.datas):
            self.smas[data] = bt.indicators.SimpleMovingAverage(
                data.close, period=self.params.period
            )
            self.crossovers[data] = bt.indicators.CrossOver(
                data.close, self.smas[data]
            )

    def next(self):
        for data in self.datas:
            pos = self.getposition(data)

            if not pos:
                if self.crossovers[data] > 0:
                    size = self.params.stake_per_asset // data.close[0]
                    self.buy(data=data, size=size)
            else:
                if self.crossovers[data] < 0:
                    self.sell(data=data, size=pos.size)
```

```
def download_data(tickers, start_date, end_date):
    """Download data for multiple tickers"""
    data_feeds = {}
    raw_data = {}

    for ticker in tickers:
        try:
            df = yf.download(ticker, start=start_date, end=end_date,
auto_adjust=False).droplevel(1, 1)
            if not df.empty:
                df.index.name = 'datetime'
                data_feeds[ticker] = bt.feeds.PandasData(dataname=df)
                raw_data[ticker] = df
                print(f"Downloaded data for {ticker}: {len(df)} rows")
        except Exception as e:
            print(f"Error downloading {ticker}: {e}")

    return data_feeds, raw_data

def apply_volatility_shock(df, multiplier=1.5, shock_std=0.2):
    """Apply volatility shock to price data"""
    df_shocked = df.copy()

    # Calculate returns
    returns = df['Close'].pct_change().dropna()

    # Create volatility multiplier (random around the base multiplier)
    np.random.seed(42)  # For reproducibility
    vol_multipliers = np.random.normal(loc=multiplier, scale=shock_std,
size=len(returns))
    vol_multipliers = np.clip(vol_multipliers, 0.5, 3.0)  # Reasonable bounds

    # Apply shock to returns
    shocked_returns = returns * vol_multipliers

    # Rebuild price series
    shocked_prices = [df['Close'].iloc[0]]
    for ret in shocked_returns:
        shocked_prices.append(shocked_prices[-1] * (1 + ret))

    # Update all OHLC data proportionally
    price_ratio = np.array(shocked_prices[1:]) / df['Close'].iloc[1:].values

    df_shocked.loc[df_shocked.index[1:], 'Open'] *= price_ratio
    df_shocked.loc[df_shocked.index[1:], 'High'] *= price_ratio
    df_shocked.loc[df_shocked.index[1:], 'Low'] *= price_ratio
```

```
    df_shocked.loc[df_shocked.index[1:], 'Close'] = shocked_prices[1:]
    df_shocked.loc[df_shocked.index[1:], 'Adj Close'] = shocked_prices[1:]

    return df_shocked

def apply_correlation_shock(data_dict, target_correlation=0.9):
    """Make all assets highly correlated (correlation breakdown scenario)"""
    shocked_data = {}

    # Use SPY as the base asset
    base_data = data_dict['SPY'].copy()
    base_returns = base_data['Close'].pct_change().dropna()

    for ticker, df in data_dict.items():
        df_shocked = df.copy()

        if ticker == 'SPY':
            shocked_data[ticker] = df_shocked
            continue

        # Get original returns
        original_returns = df['Close'].pct_change().dropna()

        # Create new returns that are correlated with SPY
        np.random.seed(42)
        noise = np.random.normal(0, 0.1, len(base_returns))

        # Weighted combination: target_correlation * SPY_returns + (1-
target_correlation) * noise
        correlated_returns = target_correlation * base_returns + (1 -
target_correlation) * noise

        # Rebuild price series
        shocked_prices = [df['Close'].iloc[0]]
        for ret in correlated_returns:
            shocked_prices.append(shocked_prices[-1] * (1 + ret))

        # Update OHLC data
        price_ratio = np.array(shocked_prices[1:]) /
df['Close'].iloc[1:].values

        df_shocked.loc[df_shocked.index[1:], 'Open'] *= price_ratio
        df_shocked.loc[df_shocked.index[1:], 'High'] *= price_ratio
        df_shocked.loc[df_shocked.index[1:], 'Low'] *= price_ratio
        df_shocked.loc[df_shocked.index[1:], 'Close'] = shocked_prices[1:]
        df_shocked.loc[df_shocked.index[1:], 'Adj Close'] = shocked_prices[1:]
```

```
        shocked_data[ticker] = df_shocked

    return shocked_data

def create_flash_crash_scenario(df, crash_day_index=100, crash_magnitude=-0.3,
recovery_days=5):
    """Create a flash crash scenario"""
    df_crashed = df.copy()

    if crash_day_index >= len(df):
        crash_day_index = len(df) // 2

    # Apply crash
    crash_multiplier = 1 + crash_magnitude
    df_crashed.iloc[crash_day_index:crash_day_index+1,
df_crashed.columns.get_loc('Open')] *= crash_multiplier
    df_crashed.iloc[crash_day_index:crash_day_index+1,
df_crashed.columns.get_loc('High')] *= crash_multiplier
    df_crashed.iloc[crash_day_index:crash_day_index+1,
df_crashed.columns.get_loc('Low')] *= crash_multiplier
    df_crashed.iloc[crash_day_index:crash_day_index+1,
df_crashed.columns.get_loc('Close')] *= crash_multiplier
    df_crashed.iloc[crash_day_index:crash_day_index+1,
df_crashed.columns.get_loc('Adj Close')] *= crash_multiplier

    # Gradual recovery over next few days
    for i in range(1, recovery_days + 1):
        if crash_day_index + i < len(df_crashed):
            recovery_factor = 1 + (abs(crash_magnitude) * (1 -
i/recovery_days) * 0.2)
            df_crashed.iloc[crash_day_index + i:crash_day_index + i + 1,
df_crashed.columns.get_loc('Open')] *= recovery_factor
            df_crashed.iloc[crash_day_index + i:crash_day_index + i + 1,
df_crashed.columns.get_loc('High')] *= recovery_factor
            df_crashed.iloc[crash_day_index + i:crash_day_index + i + 1,
df_crashed.columns.get_loc('Low')] *= recovery_factor
            df_crashed.iloc[crash_day_index + i:crash_day_index + i + 1,
df_crashed.columns.get_loc('Close')] *= recovery_factor
            df_crashed.iloc[crash_day_index + i:crash_day_index + i + 1,
df_crashed.columns.get_loc('Adj Close')] *= recovery_factor

    return df_crashed

def run_backtest(data_feeds, strategy_class, test_name, high_costs=False,
**strategy_params):
```

```
    """Run a backtest with given parameters"""
    cerebro = bt.Cerebro()

    # Add data feeds
    if isinstance(data_feeds, dict):
        for name, feed in data_feeds.items():
            cerebro.adddata(feed, name=name)
    else:
        cerebro.adddata(data_feeds)

    # Add strategy
    cerebro.addstrategy(strategy_class, **strategy_params)

    # Set broker parameters
    cerebro.broker.setcash(10000)

    if high_costs:
        # High cost environment
        cerebro.broker.setcommission(commission=0.01)  # 1% commission
        # Note: backtrader doesn't have built-in slippage, but you can
implement it
    else:
        cerebro.broker.setcommission(commission=0.001)  # 0.1% commission

    # Add analyzer
    cerebro.addanalyzer(CustomPerformanceAnalyzer, _name='performance')

    # Run backtest
    print(f"\nRunning {test_name}...")
    print(f"Starting Portfolio Value: ${cerebro.broker.getvalue():.2f}")

    results = cerebro.run()
    final_value = cerebro.broker.getvalue()

    # Extract results
    performance = results[0].analyzers.performance.get_analysis()

    print(f"Final Portfolio Value: ${final_value:.2f}")
    print(f"Total Return: {performance['total_return']*100:.2f}%")
    print(f"Sharpe Ratio: {performance['sharperatio']:.3f}")
    print(f"Max Drawdown: {performance['max_drawdown']*100:.2f}%")
    print(f"Trades: {performance['trade_count']}")

    return {
        'test_name': test_name,
        'final_value': final_value,
```

```
        'total_return': performance['total_return'] * 100,
        'sharpe_ratio': performance['sharperatio'],
        'max_drawdown': performance['max_drawdown'] * 100,
        'annual_return': performance['annual_return'] * 100,
        'annual_volatility': performance['annual_volatility'] * 100,
        'trade_count': performance['trade_count']
    }

def run_parameter_sensitivity_test(data_feed, strategy_class, base_params,
test_param, test_values):
    """Test sensitivity to parameter changes"""
    results = []

    for value in test_values:
        params = base_params.copy()
        params[test_param] = value

        test_name = f"{test_param}={value}"
        result = run_backtest(data_feed, strategy_class, test_name, **params)
        results.append(result)

    return results

def plot_stress_test_results(results):
    """Plot stress test comparison"""
    test_names = [r['test_name'] for r in results]
    returns = [r['total_return'] for r in results]
    sharpe_ratios = [r['sharpe_ratio'] for r in results]
    max_drawdowns = [r['max_drawdown'] for r in results]

    fig, (ax1, ax2, ax3) = plt.subplots(1, 3, figsize=(18, 6))

    # Returns
    colors = ['green' if r > 0 else 'red' for r in returns]
    ax1.bar(range(len(test_names)), returns, color=colors)
    ax1.set_title('Total Returns (%)')
    ax1.set_ylabel('Return (%)')
    ax1.set_xticks(range(len(test_names)))
    ax1.set_xticklabels(test_names, rotation=45, ha='right')
    ax1.grid(True, alpha=0.3)

    # Sharpe Ratios
    colors = ['green' if s > 0 else 'red' for s in sharpe_ratios]
    ax2.bar(range(len(test_names)), sharpe_ratios, color=colors)
    ax2.set_title('Sharpe Ratios')
    ax2.set_ylabel('Sharpe Ratio')
```

```
    ax2.set_xticks(range(len(test_names)))
    ax2.set_xticklabels(test_names, rotation=45, ha='right')
    ax2.grid(True, alpha=0.3)
    ax2.axhline(y=0, color='black', linestyle='--', alpha=0.5)

    # Max Drawdowns
    ax3.bar(range(len(test_names)), max_drawdowns, color='orange')
    ax3.set_title('Maximum Drawdowns (%)')
    ax3.set_ylabel('Drawdown (%)')
    ax3.set_xticks(range(len(test_names)))
    ax3.set_xticklabels(test_names, rotation=45, ha='right')
    ax3.grid(True, alpha=0.3)

    plt.tight_layout()
    plt.show()

def main():
    """Main stress testing workflow"""
    print("=" * 70)
    print("STRESS TESTING AND SCENARIO ANALYSIS DEMO")
    print("=" * 70)

    # Download data
    tickers = ['SPY', 'GLD', 'TLT', 'VTI']
    start_date = '2020-01-01'
    end_date = '2024-01-01'

    print("Downloading market data...")
    data_feeds, raw_data = download_data(tickers, start_date, end_date)

    if len(data_feeds) < 2:
        print("Not enough data. Exiting.")
        return

    # Results storage
    all_results = []

    # 1. BASELINE BACKTEST
    print("\n" + "="*50)
    print("1. BASELINE PERFORMANCE")
    print("="*50)

    # Single asset baseline
    baseline_single = run_backtest(
        data_feeds['SPY'], TrendFollowingStrategy,
        "Baseline (SPY Only)", period=50, stake=1000
```

```
    )
    all_results.append(baseline_single)

    # Multi-asset baseline
    baseline_multi = run_backtest(
        data_feeds, DiversifiedTrendStrategy,
        "Baseline (Multi-Asset)", period=50, stake_per_asset=2500
    )
    all_results.append(baseline_multi)

    # 2. VOLATILITY SHOCK TEST
    print("\n" + "="*50)
    print("2. VOLATILITY SHOCK TESTS")
    print("="*50)

    # Create volatility shocked data
    shocked_spy = apply_volatility_shock(raw_data['SPY'], multiplier=2.0)
    shocked_data_feed = bt.feeds.PandasData(dataname=shocked_spy)

    vol_shock_result = run_backtest(
        shocked_data_feed, TrendFollowingStrategy,
        "Volatility Shock (2x)", period=50, stake=1000
    )
    all_results.append(vol_shock_result)

    # 3. CORRELATION BREAKDOWN TEST
    print("\n" + "="*50)
    print("3. CORRELATION BREAKDOWN TEST")
    print("="*50)

    # Create correlation shocked data
    corr_shocked_data = apply_correlation_shock(raw_data,
target_correlation=0.95)
    corr_shocked_feeds = {}
    for ticker, df in corr_shocked_data.items():
        corr_shocked_feeds[ticker] = bt.feeds.PandasData(dataname=df)

    corr_shock_result = run_backtest(
        corr_shocked_feeds, DiversifiedTrendStrategy,
        "Correlation Shock (0.95)", period=50, stake_per_asset=2500
    )
    all_results.append(corr_shock_result)

    # 4. HIGH COST ENVIRONMENT TEST
    print("\n" + "="*50)
    print("4. HIGH COST ENVIRONMENT TEST")
```

```
print("="*50)

high_cost_result = run_backtest(
    data_feeds['SPY'], TrendFollowingStrategy,
    "High Costs (1% commission)", high_costs=True, period=50, stake=1000
)
all_results.append(high_cost_result)

# 5. FLASH CRASH SCENARIO
print("\n" + "="*50)
print("5. FLASH CRASH SCENARIO")
print("="*50)

# Create flash crash scenario
crash_spy = create_flash_crash_scenario(
    raw_data['SPY'], crash_day_index=200,
    crash_magnitude=-0.35, recovery_days=10
)
crash_data_feed = bt.feeds.PandasData(dataname=crash_spy)

crash_result = run_backtest(
    crash_data_feed, TrendFollowingStrategy,
    "Flash Crash (-35%)", period=50, stake=1000
)
all_results.append(crash_result)

# 6. PARAMETER SENSITIVITY TEST
print("\n" + "="*50)
print("6. PARAMETER SENSITIVITY TEST")
print("="*50)

# Test different SMA periods
period_test_results = run_parameter_sensitivity_test(
    data_feeds['SPY'], TrendFollowingStrategy,
    {'stake': 1000}, 'period', [20, 35, 50, 75, 100]
)
all_results.extend(period_test_results)

# 7. HISTORICAL CRISIS SCENARIOS
print("\n" + "="*50)
print("7. HISTORICAL CRISIS SCENARIOS")
print("="*50)

# COVID Crash period
try:
    covid_data = yf.download('SPY', start='2020-02-01', end='2020-05-01',
```

```
auto_adjust=False).droplevel(1, 1)
        covid_data.index.name = 'datetime'
        covid_feed = bt.feeds.PandasData(dataname=covid_data)

        covid_result = run_backtest(
            covid_feed, TrendFollowingStrategy,
            "COVID Crisis (Feb-May 2020)", period=20, stake=1000  # Shorter
period for crisis
        )
        all_results.append(covid_result)
    except:
        print("Could not download COVID crisis data")

    # 8. RESULTS ANALYSIS
    print("\n" + "="*70)
    print("STRESS TEST RESULTS SUMMARY")
    print("="*70)

    # Create summary table
    print(f"{'Test Name':<30} {'Return (%)':<12} {'Sharpe':<8} {'Max DD (%)':
<12} {'Trades':<8}")
    print("-" * 70)

    for result in all_results:
        print(f"{result['test_name']:<30} {result['total_return']:<12.2f} "
              f"{result['sharpe_ratio']:<8.3f} {result['max_drawdown']:<12.2f}
"
              f"{result['trade_count']:<8}")

    # Plot results
    print("\nGenerating stress test comparison charts...")
    plot_stress_test_results(all_results)

    # 9. ROBUSTNESS ANALYSIS
    print("\n" + "="*70)
    print("ROBUSTNESS ANALYSIS")
    print("="*70)

    baseline_return = baseline_single['total_return']
    baseline_sharpe = baseline_single['sharpe_ratio']
    baseline_dd = baseline_single['max_drawdown']

    print(f"Baseline Performance: {baseline_return:.2f}% return,
{baseline_sharpe:.3f} Sharpe, {baseline_dd:.2f}% max DD")
    print()
```

```
    # Analyze stress test impacts
    stress_tests = [r for r in all_results if 'Shock' in r['test_name'] or
'Flash Crash' in r['test_name'] or 'High Costs' in r['test_name']]

    for test in stress_tests:
        return_impact = test['total_return'] - baseline_return
        sharpe_impact = test['sharpe_ratio'] - baseline_sharpe
        dd_impact = test['max_drawdown'] - baseline_dd

        print(f"{test['test_name']}:")
        print(f"  Return Impact: {return_impact:+.2f}%
({return_impact/baseline_return*100:+.1f}%)")
        print(f"  Sharpe Impact: {sharpe_impact:+.3f}
({sharpe_impact/abs(baseline_sharpe)*100 if baseline_sharpe != 0 else
0:+.1f}%)")
        print(f"  Max DD Impact: {dd_impact:+.2f}%
({dd_impact/baseline_dd*100:+.1f}%)")
        print()

    print("KEY INSIGHTS:")
    print("• Multi-asset diversification provides some protection against
individual asset shocks")
    print("• Volatility shocks can significantly impact trend-following
strategies")
    print("• High transaction costs erode performance substantially")
    print("• Parameter sensitivity reveals optimization robustness")
    print("• Flash crashes test stop-loss and risk management effectiveness")

if __name__ == "__main__":
    main()
```

The `main` function in the code outlines a complete workflow for evaluating strategy robustness.

## Step 1: Baseline Backtest

First, you need to establish a baseline for your strategy's performance under normal market conditions. The code runs two baseline tests:

- A **single-asset** test on `SPY`.
- A **multi-asset** test on a diversified portfolio of `SPY`, `GLD`, `TLT`, and `VTI`.

This baseline gives you the metrics (e.g., `Sharpe=0.55`, `Max DD=15%`) you are trying to protect during the stress tests.

## Step 2: Micro-Level Stress Tests

These tests deliberately challenge a single variable to see how sensitive your strategy is to specific shocks.

- **Volatility Shock**: The code uses `apply_volatility_shock` to double the market's historical volatility. A trend-following strategy, which often relies on smooth trends, would likely suffer significant degradation in this test due to increased "whipsaws."
- **Correlation Breakdown**: The `apply_correlation_shock` function forces the diversified assets ( `SPY` , `GLD` , `TLT` ) to become highly correlated. A diversified strategy that performs well in the baseline should see its performance metrics collapse in this scenario, as the very benefit of diversification has been removed.
- **High Cost Environment**: By setting a high commission using `cerebro.broker.setcommission` , the code tests how a strategy's profits are affected by transaction costs. This is crucial for strategies with high turnover.
- **Parameter Sensitivity**: The `run_parameter_sensitivity_test` function is a form of stress test that helps you determine if your strategy is overfit. It runs the backtest with a range of different parameters to see if performance falls off a cliff when moved slightly away from the optimal value.

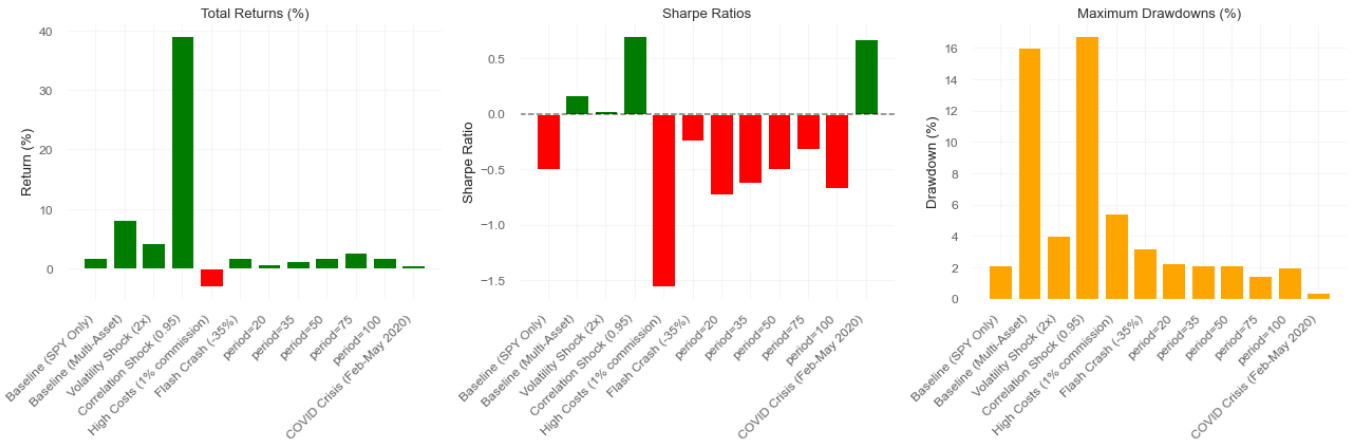
## Step 3: Macro-Level Scenarios

These tests use historical or hypothetical events to challenge the entire strategy at once.

- **Historical Crisis Scenario**: The code backtests the strategy on a real-world market crisis—the **COVID-19 crash** period from February to May 2020. This is an excellent way to see how your strategy would have actually performed in a severe, fast-moving bear market.
- **Hypothetical Flash Crash**: The `create_flash_crash_scenario` function is a great example of a forward-looking test. It simulates a sudden, severe market drop that may not be in the historical data, providing a direct test of a strategy’s stop-loss and risk management effectiveness.

```
======================================================================
STRESS TEST RESULTS SUMMARY
======================================================================
Test Name                      Return (%)   Sharpe   Max DD (%)   Trades
----------------------------------------------------------------------
Baseline (SPY Only)            1.88         -0.500   2.12         32
Baseline (Multi-Asset)         8.11         0.172    16.01        136
Volatility Shock (2x)          4.22         0.028    4.04         32
Correlation Shock (0.95)       39.12        0.705    16.75        122
High Costs (1% commission)     -2.94        -1.554   5.42         32
Flash Crash (-35%)             1.81         -0.242   3.23         33
```

| | | | | |
|---|---|---|---|---|
| period=20 | 0.83 | −0.726 | 2.29 | 47 |
| period=35 | 1.31 | −0.627 | 2.16 | 40 |
| period=50 | 1.88 | −0.500 | 2.12 | 32 |
| period=75 | 2.67 | −0.316 | 1.45 | 21 |
| period=100 | 1.73 | −0.673 | 1.96 | 16 |
| COVID Crisis (Feb–May 2020) | 0.48 | 0.674 | 0.41 | 0 |




Stress testing and scenario analysis are indispensable tools for professional quantitative risk management. While historical backtesting provides a baseline, it is insufficient for capturing the full spectrum of market risk. By systematically challenging your strategy with extreme but plausible conditions, you uncover hidden weaknesses, inform the design of hedges, and build more resilient systems.

No strategy is immune to tail events, but those prepared for them are far more likely to endure —and outperform—in the long run.

If you found the strategies in tis book useful, don't forget to checkout the **Mega Backtrader Strategy Pack** — a collection of over **250 ready-to-use strategies** for your algorithmic trading.

The **Mega Backtrader Strategy Pack** gives you **250+ fully-tested Python strategies** — adaptive filters, mean reversion, momentum, volatility breakouts, machine learning models & more — all ready to deploy in Backtrader.

**Why Traders Love This Pack**

- Instant Impact — Deploy strategies and start testing today
- Zero Guesswork — Every strategy is documented, tested, and ready-to-run
- Future-Proof — Lifetime updates included
- Proven Variety — Adaptive, mean-reversion, momentum, volatility, and ML-driven models

What’s Inside

✅ **250+ Strategies across multiple categories:**

- Adaptive Filters — Kalman, Hilbert, TRIX, Guppy MMA
- Mean Reversion — Bollinger, Keltner, pair trading, RSI/Stoch
- Momentum & Trend — MACD, ADX, Donchian, MA ribbons
- Volatility — ATR breakouts, Bollinger squeeze, VIX correlation
- Specialized ML & Advanced — HMM, decision trees, isolation forest, Hurst exponent

You Get

📂 Python `.py` scripts for direct Backtrader use

📄 PDF manuals detailing logic, parameters, and best practices

🎯 Get Instant Access — Download Today, Trade Tonight

👉 **Get it here**