ALGORITHMIC TRADING

with VectorBT

Ali AZARY

Algorithmic Trading with VectorBT

Ali AZARY

Copyright

First edition: February 2026

Published independently

Luxembourg, Luxembourg

Website: https://www.aliazary.com/

Disclaimer: This book is for educational purposes only and does not constitute financial, legal, or professional advice. Trading involves risk, including the possible loss of principal.

Table of Contents

Table of Contents

1 Downloading Market Data

The first thing we need is to obtain a clean time-indexed price series that can feed indicators and a portfolio simulator without hidden alignment bugs. In vectorbt, the cleanest route is to download OHLCV bars (Open, High, Low, Close, Volume), then decide which field is used for (1) computing signals and (2) simulating execution.

1.1 Using `vbt.YFData.download`

`vbt.YFData.download` is a convenience wrapper around Yahoo Finance that returns a vectorbt data object. You can pull one symbol or many symbols, and then extract fields by name.

```
import vectorbt as vbt

symbol = "BTC-USD"
data = vbt.YFData.download(symbol, start="2020-01-01", end="2025-01-01")
```

What you get back is not just a DataFrame. It is a vectorbt data container that knows how to return specific fields in aligned pandas structures.

```
close = data.get("Close")
```

Conceptually, `close` is a time series indexed by timestamps:

- The index is a `DatetimeIndex` (each row corresponds to a bar).
- The values are numeric prices.

Why this matters: vectorbt indicators and portfolios assume that all inputs share the same index. Using `data.get(...)` helps ensure that each field you extract is aligned the same way.

If you want multiple assets at once:

```
symbols = ["BTC-USD", "ETH-USD"]
data = vbt.YFData.download(symbols, start="2020-01-01", end="2025-01-01")
close = data.get("Close")
```

Now `close` becomes a DataFrame where:

- rows = time
- columns = tickers (`BTC-USD`, `ETH-USD`)
- each column is a separate price series, aligned on the same timestamps

This is powerful because vectorbt can broadcast computations across columns automatically. For example, the exact same moving-average logic can run for every asset with one call.

If you use `yfinance` directly, the raw return may have a MultiIndex for columns (especially when downloading multiple tickers). Your requirement `.droplevel(1, 1)` removes that extra level.

```
import yfinance as yf

df = yf.download("BTC-USD", start="2020-01-01", end="2025-01-01").droplevel(1,
 1)
close = df["Close"]
```

Use this when you want full manual control, but for vectorbt workflows `vbt.YFData.download` is usually the shortest path from download to backtest.

1.2 Understanding OHLCV fields

A price bar summarizes trading within a fixed interval (day, hour, etc.). OHLCV is the standard set of fields:

- **Open**: first traded price in the interval
- **High**: maximum traded price
- **Low**: minimum traded price
- **Close**: last traded price
- **Volume**: total traded quantity during the interval

Extracting them:

```
open_  = data.get("Open")
high   = data.get("High")
low    = data.get("Low")
close  = data.get("Close")
volume = data.get("Volume")
```

A quick integrity check: for each bar, the high should be at least as large as open and close, and the low should be at most as small as open and close.

```
assert (high >= open_).all().all()
assert (high >= close).all().all()
assert (low  <= open_).all().all()
assert (low  <= close).all().all()
```

Why do we care in a DMAC strategy?

- A **moving average** is usually computed on **Close**.
- A **signal** is generated when one MA crosses another.
- The **execution price** in a simulation might be the Close (optimistic) or the next bar's Open (more realistic).

So even if the strategy "looks like" it only uses Close, having full OHLCV available helps you later when you want to tighten realism (next-open execution, stop-loss logic, slippage approximations, etc.).

1.3 Timezones, timestamps, and resampling

1.4 Time index basics

Your series must be correctly ordered and consistently timestamped. The most common failure mode in backtesting is not the strategy logic; it is misaligned time series.

Check:

```
idx = close.index
print(idx.is_monotonic_increasing)
print(idx.tz)
```

- `is_monotonic_increasing` should be `True` (time moves forward).
- `idx.tz` tells you whether the index is timezone-aware.

For crypto (BTC-USD), trading is continuous. For stocks, you must also consider sessions and holidays. The important lesson is: if you mix assets with different trading calendars, you can create gaps and misalignment.

1.5 Resampling correctly

Sometimes you download daily bars but want weekly bars, or hourly bars but want 4-hour bars. Resampling must preserve OHLCV meaning.

Correct OHLCV aggregation rules:

- Open → first
- High → max
- Low → min
- Close → last
- Volume → sum

Example: daily to weekly.

```
ohlcv = data.get(["Open", "High", "Low", "Close", "Volume"])

weekly = ohlcv.resample("W").agg({
    "Open": "first",
    "High": "max",
    "Low": "min",
    "Close": "last",
    "Volume": "sum"
})

weekly_close = weekly["Close"]
```

Why not just do `close.resample("W").last()`?

That is fine only if everything in your simulation uses Close and you never reference intrabar ranges. The moment you introduce next-open execution or stop rules, you need proper Open/High/Low/Close resampling to avoid fabricating unrealistic bars.

1.6 Missing data, NaNs, and cleaning rules

There are two different NaN problems, and you should treat them differently.

1.7 NaNs from the data source (gaps)

These happen because:

- history is missing for some timestamps
- the API returns incomplete data
- multi-asset alignment introduces NaNs where one asset trades and another does not

Find them:

```
close.isna().sum()
```

A conservative default is: drop rows where the main price series is missing.

```
close_clean = close.dropna()
```

Avoid forward-filling by default. Forward-filling can create “flat” synthetic prices across gaps, which can reduce volatility and drawdowns artificially, and can trigger indicator logic that should not have been triggered.

1.8 NaNs from indicator warmup (expected)

Moving averages require a minimum number of bars equal to the window. Until then, the MA is undefined.

```
fast_ma = vbt.MA.run(close_clean, window=20).ma
slow_ma = vbt.MA.run(close_clean, window=50).ma

fast_ma.isna().sum(), slow_ma.isna().sum()
```

This is normal and desirable. The correct handling is: allow NaNs during warmup, and let the first valid signals appear only after both MAs exist. Do not fill warmup NaNs, because that invents indicator values before enough data exists.

1.9 Selecting the price series for signals (Close vs others)

This is where many backtests become unintentionally optimistic.

There are two separate choices:

1. Which series generates signals?
2. Which series is used as the execution price?

1.10 Typical research choice: signals on Close

```
entries = (fast_ma.shift() < slow_ma.shift()) & (fast_ma > slow_ma)
exits = (fast_ma.shift() > slow_ma.shift()) & (fast_ma < slow_ma)
```

Interpretation:

- `entries` becomes `True` on bars where the fast MA crosses above the slow MA.
- `exits` becomes `True` on bars where the fast MA crosses below the slow MA.

These are boolean series aligned with the same timestamp index as `close`.

1.11 Execution realism: “decide at close, trade next open”

If you compute signals at the Close, you generally cannot assume you traded at that same Close without strong justification. A more conservative model is: you observe the Close, compute the signal, then execute at the next bar’s Open.

That requires shifting signals forward by one bar and using Open as the price series for the portfolio simulation.

```
open_ = data.get("Open").reindex(close.index)

entries_next = entries.vbt.fshift(1)
exits_next   = exits.vbt.fshift(1)
```

Explanation:

- `fshift(1)` moves each signal one bar forward in time.

- This prevents “using the future” by trading on the same bar that generated the signal.

Now you can backtest:

```
pf = vbt.Portfolio.from_signals(
    open_,
    entries_next,
    exits_next,
    fees=0.001
)
```

Key idea: a backtest is not only about signals. It is also about the implied timeline of information and execution. The notebook’s later comparisons (parameter sweeps, walk-forward tests, benchmark comparisons) are only meaningful if this choice is consistent across all runs.

let’s see some results:

```
print("Portfolio stats (next-open execution):")
print(pf_next_open.stats())
```

Output:

```
Portfolio stats (next-open execution):
Start                         2019-12-31 00:00:00+00:00
End                           2024-12-31 00:00:00+00:00
Period                                1828 days 00:00:00
Start Value                                        100.0
End Value                                     686.048599
Total Return [%]                              586.048599
Benchmark Return [%]                         1170.053125
Max Gross Exposure [%]                             100.0
Total Fees Paid                                15.745005
Max Drawdown [%]                               58.647832
Max Drawdown Duration                 1193 days 00:00:00
Total Trades                                          19
Total Closed Trades                                   18
Total Open Trades                                      1
Open Trade PnL                                218.022836
Win Rate [%]                                   33.333333
Best Trade [%]                                380.968946
Worst Trade [%]                               -19.100004
Avg Winning Trade [%]                          83.765988
Avg Losing Trade [%]                           -8.464545
Avg Winning Trade Duration              93 days 08:00:00
Avg Losing Trade Duration               27 days 04:00:00
Profit Factor                                   1.75225
Expectancy                                    20.445876
Sharpe Ratio                                   1.076319
```

```
Calmar Ratio                              0.799542
Omega Ratio                               1.248288
Sortino Ratio                             1.687405
```

1.12 When to use other series

- Use **Open** if your model is explicitly "trade at open" (e.g., next bar open).
- Use **High/Low** only when you define strict intrabar assumptions (stops, take-profit rules). Otherwise you risk look-ahead, because a daily bar's High/Low contains information you would not have at the time you decide.

A disciplined research rule is:

- Compute indicators on Close.
- Decide whether fills occur at the same Close (optimistic) or next Open (conservative).
- Apply that consistently everywhere: single backtests, parameter sweeps, and walk-forward evaluation.

1.13 Portfolio Backtesting with vectorbt

Backtesting in vectorbt is the process of turning three ingredients into a simulated trading record:

1. a price series (what you trade and what you get filled at),
2. entry/exit signals (when you want to be in/out),
3. execution assumptions (timing, fees, slippage, position rules).

vectorbt's portfolio engine converts those into orders, positions, and trade records, then computes performance metrics and plots.

2 From Signals to Trades

2.1 `vbt.Portfolio.from_signals` overview

`vbt.Portfolio.from_signals` is the most common way to backtest "signal-based" strategies (like a moving-average crossover). You provide:

- `close` (or open, depending on your fill assumption) as the price series,
- `entries` as a boolean Series/Frame,
- `exits` as a boolean Series/Frame,
- optional realism parameters (fees, slippage),
- optional position rules (long-only, short-only, both).

Minimal example (signals on Close, filled at Close — optimistic):

```
import vectorbt as vbt

price = close  # Series with DatetimeIndex
pf = vbt.Portfolio.from_signals(
    price,
    entries,
    exits,
    init_cash=10_000
)
```

What happens internally:

- When `entries` turns True and you are flat, vectorbt creates a buy order.
- When `exits` turns True and you are long, vectorbt creates a sell order.
- Positions persist between signals (you stay long until an exit, and flat until an entry).
- The engine then derives trade records, equity curve, drawdowns, and stats.

Multi-parameter / multi-asset works the same way. If `price` is a DataFrame with columns for assets or parameter combinations, vectorbt backtests all columns in one vectorized run.

2.2 Long-only vs short vs both (typical DMAC configuration)

A dual moving-average crossover (DMAC) is commonly implemented as **long-only**:

- Entry: fast MA crosses above slow MA (go long).

- Exit: fast MA crosses below slow MA (go flat).

```
fast = vbt.MA.run(close, window=20).ma
slow = vbt.MA.run(close, window=50).ma

entries = fast.ma_crossed_above(slow)
exits   = fast.ma_crossed_below(slow)

pf_long_only = vbt.Portfolio.from_signals(
    close,
    entries,
    exits,
    direction="longonly",     # explicit
    init_cash=10_000
)
```

Short-only variant (trend-following "downtrend" participation) flips the interpretation:

```
pf_short_only = vbt.Portfolio.from_signals(
    close,
    entries,
    exits,
    direction="shortonly",
    init_cash=10_000
)
```

Long/short variant allows taking both sides. A common pattern is:

- Long entry when fast crosses above slow.
- Short entry when fast crosses below slow.

This uses *two* entry signals (one for long, one for short). The exact argument names can differ by vectorbt version; conceptually:

```
long_entries  = fast.ma_crossed_above(slow)
long_exits    = fast.ma_crossed_below(slow)

short_entries = fast.ma_crossed_below(slow)
short_exits   = fast.ma_crossed_above(slow)

pf_long_short = vbt.Portfolio.from_signals(
    close,
    entries=long_entries,
    exits=long_exits,
    short_entries=short_entries,
    short_exits=short_exits,
    init_cash=10_000
)
```

If you keep the strategy long-only (as most DMAC research does), you avoid short-specific issues (borrow costs, asymmetric risk, different fill behavior).

2.3 Trade lifecycle: entry, exit, position state

A signal strategy has a simple state machine:

- Flat → (entry signal) → Long
- Long → (exit signal) → Flat

vectorbt maintains this automatically. You can inspect what it produced:

```
# Position series (exposure over time)
pos = pf_long_only.position

# Trades / records
trades = pf_long_only.trades.records_readable
orders = pf_long_only.orders.records_readable
```

Interpretation:

- **Orders** are executions (buy/sell fills).
- **Trades** bundle an entry fill and an exit fill into a single "round trip" record (PnL, duration, return, etc.).
- **Position** shows whether you were in the market at each timestamp (useful for exposure and turnover analysis).

2.4 Order timing assumptions: bar close vs next open

This is a core realism decision.

If your signals are computed using `Close`, you should usually assume you can only trade **after** that Close is known. A safer approximation is: "signal at close, fill at next open".

Fill at the same Close (optimistic):

```
pf_close_fill = vbt.Portfolio.from_signals(
    close,
    entries,
    exits,
    init_cash=10_000
)
```

Fill at next Open (more realistic for bar-based strategies):

```
open_ = data.get("Open").reindex(close.index)
```

```
entries_next = entries.vbt.fshift(1)
exits_next   = exits.vbt.fshift(1)

pf_next_open = vbt.Portfolio.from_signals(
    open_,
    entries_next,
    exits_next,
    init_cash=10_000
)
```

Why the shift matters:

- Without shifting, you implicitly trade using information from the same bar you are "evaluating," which can create look-ahead bias depending on your interpretation.
- Shifting enforces an information timeline: close happens → signal computed → next bar opens → order fills.

2.5 Cash sharing and initial capital

In single-asset backtests, `init_cash` is the starting money. In multi-asset or multi-column tests, you must understand whether capital is shared or treated as separate "independent accounts".

Common patterns:

Independent capital per column (useful for comparing parameter sets side-by-side):

```
pf = vbt.Portfolio.from_signals(
    close, entries, exits,
    init_cash=10_000
)
```

Shared capital across assets (portfolio allocation behavior). In vectorbt, this is typically controlled by a "cash sharing" option; conceptually:

```
pf_shared = vbt.Portfolio.from_signals(
    close_df,
    entries_df,
    exits_df,
    init_cash=10_000,
    cash_sharing=True
)
```

Interpretation:

- Without cash sharing, each column behaves like its own isolated strategy account.
- With cash sharing, multiple assets compete for the same pool of cash, which is closer to a real multi-asset portfolio.

2.6 Costs and Realism

2.7 Fees and commissions

Fees reduce performance and are often the difference between “works on paper” and “doesn’t survive trading”.

Example: 0.10% fee per trade (0.001):

```
pf = vbt.Portfolio.from_signals(
    open_, entries_next, exits_next,
    init_cash=10_000,
    fees=0.001
)
```

Notes:

- Fees apply on execution (typically on both entry and exit).
- For high-turnover strategies, even small fees compound quickly.

2.8 Slippage modeling (practical approximations)

Slippage is the idea that you do not get the displayed price; you get a worse fill.

A simple approximation is a fixed slippage percentage:

```
pf = vbt.Portfolio.from_signals(
    open_, entries_next, exits_next,
    init_cash=10_000,
    fees=0.001,
    slippage=0.0005  # 0.05%
)
```

This is not “market impact modeling,” but it is often enough to stress-test sensitivity. A useful habit is to run a grid of costs:

- fees: 0.0%, 0.05%, 0.10%
- slippage: 0.0%, 0.05%, 0.10%

If results collapse quickly as you add realistic costs, the edge is fragile.

2.9 Spread, latency, and market impact (limitations)

Bar-based backtests typically ignore:

- bid/ask spread dynamics,
- order book depth and impact,

- latency (signal delay, exchange delay),
- partial fills.

Fixed `fees` and `slippage` are a coarse proxy. For educational and research backtests, the key is to state the assumptions clearly and test robustness under worse execution.

2.10 Why costs matter more in high-turnover systems

Turnover means you trade frequently. Each trade pays:

- entry fee + exit fee,
- entry slippage + exit slippage,
- plus any spread/impact hidden inside slippage.

If a strategy makes small average gains per trade, costs can exceed expected edge. This is why trend-following strategies often prefer fewer, larger moves and accept missing some turns.

2.11 Reading Results

2.12 Portfolio stats table (core metrics)

vectorbt provides a summary table:

```
pf.stats()
```

Output:

```
Start                         2019-12-31 00:00:00+00:00
End                           2024-12-31 00:00:00+00:00
Period                               1828 days 00:00:00
Start Value                                     10000.0
End Value                                  67347.346247
Total Return [%]                             573.473462
Benchmark Return [%]                        1170.053125
Max Gross Exposure [%]                            100.0
Total Fees Paid                             1559.229931
Max Drawdown [%]                              59.100213
Max Drawdown Duration                1195 days 00:00:00
Total Trades                                         19
Total Closed Trades                                  18
Total Open Trades                                     1
Open Trade PnL                             21379.679694
Win Rate [%]                                  33.333333
Best Trade [%]                               380.488118
Worst Trade [%]                              -19.180964
Avg Winning Trade [%]                         83.582214
```

```
Avg Losing Trade [%]                        -8.556135
Avg Winning Trade Duration          93 days 08:00:00
Avg Losing Trade Duration           27 days 04:00:00
Profit Factor                           1.734998
Expectancy                           1998.203697
Sharpe Ratio                            1.068112
Calmar Ratio                            0.784258
Omega Ratio                             1.246062
Sortino Ratio                           1.674267
dtype: object
```

Common metrics and what they mean:

- Total Return: growth over the period
- CAGR: annualized growth rate (time-normalized)
- Volatility: variability of returns
- Sharpe: return per unit risk (higher is better, but depends on assumptions)
- Max Drawdown: worst peak-to-trough decline (risk reality check)

2.13 Equity curve and drawdowns

Two essential plots:

```
pf.value().vbt.plot()
pf.drawdowns.plot()
pf.drawdown().vbt.plot(trace_kwargs=dict())
```

The first one gives us the equity curve. the second line marks peaks and valleys on the equity curve and shades the drawdown and recovery periods in red and green. The last line plots the drawdown curve:

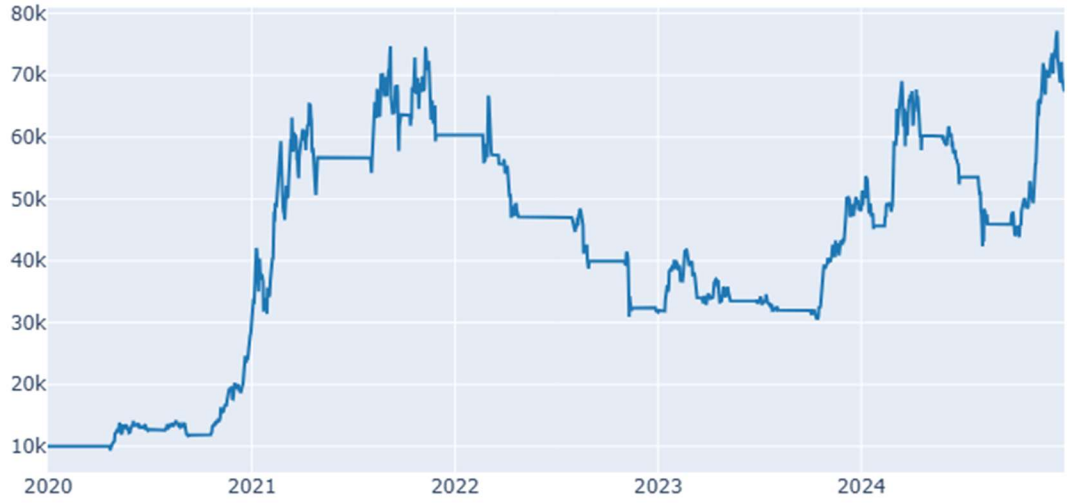

Figure 2.13.1: Equity curve

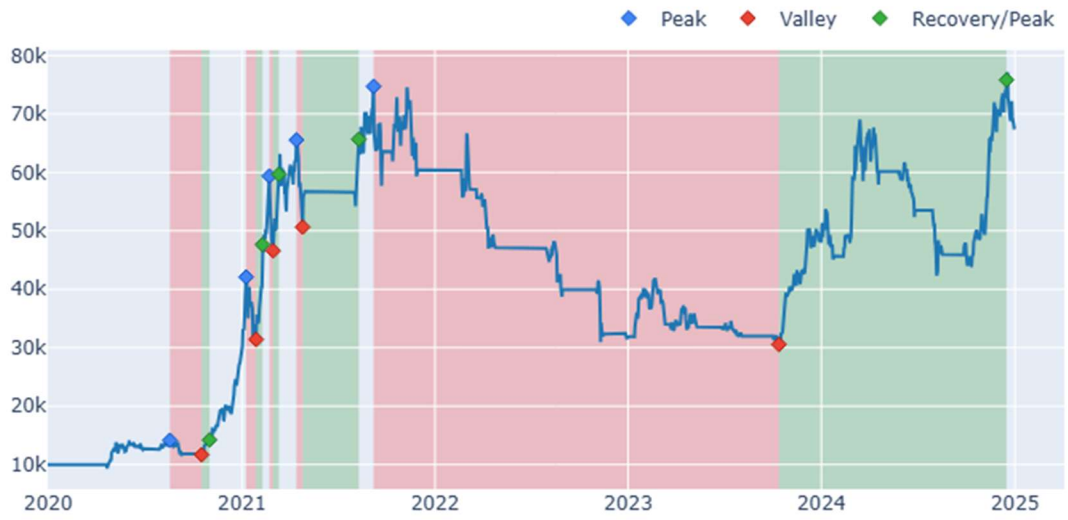

Figure 2.13.2: Equity curve with drawdowns and recoveries

Figure 2.13.3: Drawdown

What to look for:

- smoothness vs “boom/bust”
- whether growth is steady or dependent on a few lucky periods
- size and duration of drawdowns (psychological and capital constraints)

2.14 Trades/records: win rate, expectancy, profit factor

Trade-level diagnostics come from trade records:

```
pf.trades.stats()
pf.trades.records_readable
```

The first one reports trades statistics:

```
Start                         2019-12-31 00:00:00+00:00
End                           2024-12-31 00:00:00+00:00
Period                               1828 days 00:00:00
First Trade Start             2020-04-20 00:00:00+00:00
Last Trade End                2024-12-31 00:00:00+00:00
Coverage                              983 days 00:00:00
Overlap Coverage                        0 days 00:00:00
Total Records                                        19
Total Long Trades                                    19
Total Short Trades                                    0
Total Closed Trades                                  18
Total Open Trades                                     1
Open Trade PnL                                21379.679694
Win Rate [%]                                     33.333333
```

```
Max Win Streak                                  2
Max Loss Streak                                 7
Best Trade [%]                          380.488118
Worst Trade [%]                         -19.180964
Avg Winning Trade [%]                    83.582214
Avg Losing Trade [%]                     -8.556135
Avg Winning Trade Duration    93 days 08:00:00
Avg Losing Trade Duration     27 days 04:00:00
Profit Factor                            1.734998
Expectancy                            1998.203697
SQN                                      0.670922
dtype: object
```

And the second line reports the individual trades:

Exit Trade Id	Column	Size	Entry Timestamp	Avg Entry Price	Entry Fees	Exit Timestamp	Avg Exit Price	Exit Fees	PnL	Return	Direction	Status	Position Id
0	0	1.389341	2020-04-20 00:00:00+00:00	7190.466972	9.990010	2020-06-30 00:00:00+00:00	9180.988264	12.755523	2742.767149	0.274551	Long	Closed	0
1	0	1.154864	2020-07-28 00:00:00+00:00	11022.972599	12.730037	2020-09-10 00:00:00+00:00	10237.208913	11.822587	-932.003023	-0.073213	Long	Closed	1
2	0	1.031850	2020-10-14 00:00:00+00:00	11434.762375	11.798965	2021-04-28 00:00:00+00:00	55009.118400	56.761186	44893.660466	3.804881	Long	Closed	2
3	0	1.418776	2021-08-02 00:00:00+00:00	39927.215350	56.647777	2021-09-24 00:00:00+00:00	44871.853631	63.663111	6895.023473	0.121717	Long	Closed	3
4	0	1.103905	2021-10-12 00:00:00+00:00	57555.595447	63.535912	2021-11-28 00:00:00+00:00	54785.616926	60.478119	-3181.807294	-0.050079	Long	Closed	4
5	0	1.503738	2022-02-20 00:00:00+00:00	40138.160613	60.357283	2022-03-08 00:00:00+00:00	38040.872393	57.203511	-3271.333109	-0.054199	Long	Closed	5
6	0	1.352424	2022-03-20 00:00:00+00:00	42212.501953	57.089218	2022-03-21 00:00:00+00:00	41225.509746	55.754386	-1447.675967	-0.025358	Long	Closed	6
7	0	1.187804	2022-03-28 00:00:00+00:00	46845.262488	55.642989	2022-04-23 00:00:00+00:00	39718.853295	47.178212	-8567.598324	-0.153974	Long	Closed	7
8	0	1.973579	2022-07-29 00:00:00+00:00	23857.135497	47.083949	2022-08-30 00:00:00+00:00	20288.462022	40.040889	-7130.185033	-0.151436	Long	Closed	8
9	0	1.948822	2022-11-01 00:00:00+00:00	20505.145887	39.960887	2022-11-15 00:00:00+00:00	16609.175633	32.368333	-7664.883289	-0.191810	Long	Closed	9
10	0	1.917055	2022-12-26 00:00:00+00:00	16850.671125	32.303661	2023-01-04 00:00:00+00:00	16671.864976	31.960880	-407.045744	-0.012601	Long	Closed	10
11	0	1.689609	2023-01-13 00:00:00+00:00	18878.340703	31.897022	2023-03-11 00:00:00+00:00	20177.783015	34.092572	2129.560376	0.066764	Long	Closed	11
12	0	1.212750	2023-03-20 00:00:00+00:00	28055.622363	34.024455	2023-05-09 00:00:00+00:00	27681.220825	33.570400	-521.650310	-0.015332	Long	Closed	12
13	0	1.098622	2023-06-26 00:00:00+00:00	30495.763699	33.503326	2023-08-04 00:00:00+00:00	29159.795621	32.035602	-1533.263250	-0.045765	Long	Closed	13
14	0	1.142218	2023-10-02 00:00:00+00:00	27990.787228	31.971595	2024-01-25 00:00:00+00:00	40055.513006	45.752147	13702.829035	0.428594	Long	Closed	14
15	0	0.944955	2024-02-12 00:00:00+00:00	48320.534912	45.660734	2024-04-20 00:00:00+00:00	63819.176012	60.306254	14539.552487	0.318426	Long	Closed	15
16	0	0.868495	2024-05-26 00:00:00+00:00	69298.921207	60.185762	2024-06-26 00:00:00+00:00	61758.780943	53.637188	-6662.396558	-0.110697	Long	Closed	16
17	0	0.783827	2024-07-29 00:00:00+00:00	68293.184215	53.530021	2024-08-15 00:00:00+00:00	58703.895088	46.013680	-7615.884532	-0.142273	Long	Closed	17
18	0	0.726954	2024-09-26 00:00:00+00:00	63170.116148	45.921745	2024-12-31 00:00:00+00:00	92643.250000	0.000000	21379.679694	0.465568	Long	Open	18

Figure 2.14.1: Trades report snapshot

Interpretation:

- Win rate: percentage of profitable trades (can be low in trend systems)
- Expectancy: average profit per trade (what matters most)
- Profit factor: gross profit / gross loss (sensitive to outliers)

A healthy strategy can have a low win rate but high expectancy if winners are much larger than losers.

2.15 Exposure time and turnover

Exposure shows how much time you are in the market.

```
exposure_time_pct = (pf.asset_value() > 0).mean() * 100
```

Turnover (or trade frequency) can be inferred from number of trades and holding times:

```
pf.trades.count()
pf.trades.duration.mean()
```

High turnover plus modest edge is a warning sign unless you have excellent execution.

2.16 Interpreting results responsibly

Minimum discipline for conclusions:

- Compare to a benchmark (buy-and-hold for BTC).
- Test multiple periods (walk-forward/rolling windows).
- Stress costs upward (fees + slippage).
- Prefer parameter regions that are stable (not a single sharp optimum).

A backtest is a model of reality, not reality. Your job is to make the model's assumptions explicit, then see whether the edge survives when those assumptions are tightened.

3 Parameter Search (Optimization)

A parameter search answers a specific question: “If we keep the strategy rules fixed, which parameter choices behave best on the data?” For a DMAC strategy, the parameters are usually the fast and slow moving-average windows. vectorbt is designed for this because it can evaluate many parameter combinations in one vectorized run, without writing nested loops.

The correct mindset is not “find the best number.” It is “map the behavior of the strategy across a region of reasonable parameters, then look for stability.” A sharp isolated winner is often overfit.

3.1 Parameter Sweeps (Grid Search)

3.2 Defining fast/slow window ranges

A moving-average window is the number of bars used in the average. Before you search, you must define:

- the bar interval (daily, hourly, etc.)
- a reasonable parameter region for your strategy idea
- constraints (fast must be smaller than slow)

Example ranges (daily bars):

```
import numpy as np

fast_range = np.arange(5, 51, 1)      # 5..50
slow_range = np.arange(20, 301, 5)    # 20..300 step 5
```

Guidelines for choosing ranges:

- Fast windows should be short enough to react but not so short that the strategy becomes noise trading.
- Slow windows should represent the “trend” horizon.
- The ranges should reflect the timeframe: hourly windows are not comparable to daily windows.

A common beginner mistake is searching extremely wide ranges without rationale. This expands the search space and increases the chance of finding a “lucky” combination.

3.3 Using vectorbt's combinatorial runs (`run_combs`)

vectorbt indicators support "parameter broadcasting." Instead of computing one MA, you compute many at once.

For DMAC you need *pairs* of windows. `run_combs` is a convenient way to generate all combinations of two parameter sets.

```
import vectorbt as vbt
close = data.get("Close").dropna()
fast_ma, slow_ma = vbt.MA.run_combs(
    close,
    window=fast_range,
    r=2,              # choose 2 windows
)
```

Conceptually, you get arrays (or DataFrames) of moving averages corresponding to many window pairs. The exact column naming depends on version, but the key point is: vectorbt builds a grid of MA outputs aligned on the same timestamps.

In practice, many people prefer a clearer pattern: compute MA for all candidate windows once, then pair them. That can be more memory-friendly for large grids.

3.4 Building the full strategy grid efficiently

The DMAC rule:

- Entry when fast MA crosses above slow MA
- Exit when fast MA crosses below slow MA

Once you have MA outputs for many parameter pairs, signals become vectorized boolean matrices, and `Portfolio.from_signals` can backtest them all at once.

A typical pattern:

```
fast_ma = vbt.MA.run(close, window=fast_range).ma
slow_ma = vbt.MA.run(close, window=slow_range).ma
```

Now you need a grid of pairs. The efficient idea is: broadcast the 1D sets into a 2D grid, compare fast vs slow, and generate signals across all pairs. How you do this depends on your vectorbt version and how it structures columns. The goal is always the same:

- one "column" per parameter pair
- backtest all columns in one portfolio object

Once signals are prepared, the backtest call is still one line:

```
pf = vbt.Portfolio.from_signals(
    close,
    entries,
    exits,
    init_cash=10_000,
    fees=0.001,
    slippage=0.0005,
    direction="longonly"
)
```

This produces a portfolio object where each column corresponds to one parameter set. You can then compute stats for all parameter sets and compare them.

3.5 Preventing invalid combos (fast >= slow)

Invalid DMAC pairs occur when:

- fast window is equal to slow window (no crossover meaningfully)
- fast window is larger than slow window (logic is reversed)

Two standard solutions:

Mask them out by creating a validity condition:

```
valid = fast_window_values < slow_window_values
```

Or generate only valid pairs by construction:

- choose fast from a low range and slow from a high range
- enforce `fast < slow` when building combinations

If you do not prevent invalid pairs, your heatmap will contain nonsense regions that can be mistaken for real structure.

3.6 Runtime and memory considerations

Grid searches can explode in size:

- If you test 46 fast windows and 57 slow windows, that's 2,622 combinations.
- If you test 200 × 200, that's 40,000 combinations.
- Each combination can generate signals and store results.

Practical constraints:

- Prefer fewer, meaningful values (step sizes that make sense).
- Start small to validate logic.

- Reduce plotting resolution (heatmaps do not need 1-step granularity everywhere).
- Use realistic execution assumptions early (fees/slippage), because they change the landscape.

A useful practice is to measure and scale:

- run 200 combos, measure runtime,
- extrapolate before launching 50,000 combos.

3.7 Comparing Parameter Sets

3.8 Selecting evaluation metrics (Sharpe, CAGR, total return, etc.)

Different metrics answer different questions:

- Total Return: raw performance over the sample (sensitive to sample choice)
- CAGR: time-normalized growth (better for comparing different spans)
- Volatility: risk magnitude
- Sharpe: risk-adjusted return (depends on assumptions and bar frequency)
- Max Drawdown: worst loss from peak (risk reality check)

For trend strategies, Max Drawdown and robustness across periods usually matter more than peak Sharpe on one sample.

In vectorbt, you typically compute a stats table for all columns:

```
stats = pf.stats()
```

Then select the metric column you care about and reshape into a grid for visualization.

```
Stats (first 10 rows):
Start                     2019-12-31 00:00:00+00:00
End                       2024-12-31 00:00:00+00:00
Period                           1828 days 00:00:00
Start Value                                  10000.0
End Value                               50269.163485
Total Return [%]                          402.691635
Benchmark Return [%]                     1198.782453
Max Gross Exposure [%]                         100.0
Total Fees Paid                           580.667072
Max Drawdown [%]                           47.390353
Name: agg_func_mean, dtype: object
```

3.9 Heatmaps: reading and interpreting them

A heatmap turns “one value per parameter pair” into a picture.

```
sharpe = pf.sharpe_ratio()

sharpe_hm = sharpe.unstack("slow").sort_index().sort_index(axis=1)
sharpe_hm.vbt.heatmap(
    xaxis_title="Slow window",
    yaxis_title="Fast window",
    trace_kwargs=dict(colorbar_title="Sharpe")
).show()
```

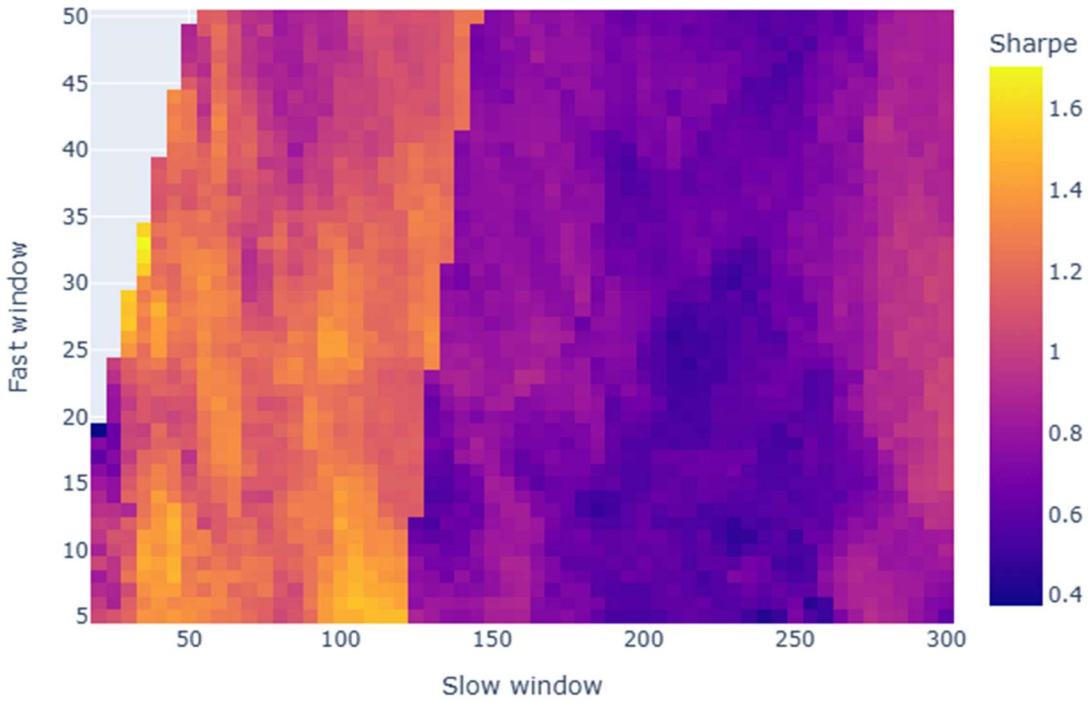

Figure 3.9.1: Sharpe ratio heatmap

Typical axes:

- x-axis: fast window
- y-axis: slow window
- color: metric value (Sharpe, CAGR, etc.)

Interpretation rules:

- Look for *regions* of good performance, not isolated cells.

- A stable strategy has a “plateau” where many nearby parameters behave similarly.
- A single bright pixel is often luck or a data-specific artifact.

3.10 Ranking parameter pairs and extracting “top-N”

A ranking step is convenient for inspection:

- “What are the top 20 Sharpe combinations?”
- “What are the top 20 CAGR combinations?”

```
sharpe = pf.sharpe_ratio()
cagr = pf.annualized_return()

print("\nTop Sharpe parameter pairs:")
print(sharpe.sort_values(ascending=False).head(TOP_N))

print("\nTop CAGR parameter pairs:")
print(cagr.sort_values(ascending=False).head(TOP_N))
```

```
Top Sharpe parameter pairs:
fast  slow
33    35      1.706279
32    35      1.653469
34    35      1.587443
31    35      1.568485
29    30      1.558514
27    30      1.546792
6     105     1.534246
28    30      1.524132
5     110     1.513248
6     110     1.507153
5     120     1.506043
      115     1.503500
7     105     1.499468
8     45      1.486008
6     100     1.483759
12    45      1.483340
6     115     1.478877
8     105     1.476291
28    40      1.475560
9     105     1.473729
Name: sharpe_ratio, dtype: float64

Top CAGR parameter pairs:
fast  slow
33    35      0.927445
32    35      0.891139
34    35      0.845492
```

```
29    30      0.820198
31    35      0.814870
27    30      0.814741
6     105     0.810093
28    30      0.800282
5     110     0.792156
6     110     0.787380
5     120     0.786609
      115     0.784270
7     105     0.782203
6     100     0.770416
28    40      0.769850
8     105     0.763958
6     115     0.763192
9     105     0.761528
5     105     0.752523
8     45      0.752171
Name: annualized_return, dtype: float64
```

But top-N should never be your final selection rule. It is a diagnostic.

Good workflow:

- rank combinations by a metric,
- inspect the surrounding neighborhood in the heatmap,
- test stability with walk-forward splits (next part of the book).

3.11 Sensitivity and stability (avoid single-point winners)

Stability means:

- nearby parameter values perform similarly
- performance persists across subperiods
- the strategy does not rely on one or two lucky trades

Ways to test stability:

- Compare heatmaps for different metrics (Sharpe vs drawdown).
- Use rolling windows (walk-forward).
- Prefer parameters in the middle of a good region rather than at an extreme edge.

3.12 Avoiding overfitting in grid searches

Overfitting happens when you select parameters that are tuned to noise in your sample.

Practical defenses:

- Use a restricted, justified search space.
- Penalize complexity (extremely short fast windows usually mean high turnover).
- Add realistic costs early (fees/slippage).
- Validate on multiple time splits (walk-forward).
- Prefer robust regions over peak values.

If a parameter set looks amazing only without costs, or only on one subperiod, treat it as a warning signal, not a discovery.

4 Backtesting Strategies

This chapter builds an interactive “strategy lab” that lets you evaluate a Dual Moving Average Crossover (DMAC) strategy against two baselines:

1. A random strategy that fires the same number of entry signals as DMAC (a fair null model).
2. A buy-and-hold strategy (the simplest benchmark).

You will be able to:

- Select any time window directly on the price chart.
- Instantly recompute strategy performance for that window.
- Explore the full MA-parameter grid via a heatmap.
- Compare DMAC vs Random distributions in a histogram, with “Hold” marked as a reference.

The result is a compact dashboard that answers two practical questions:

- Does DMAC outperform randomness (not just holding) over this period?
- Are the “good” results robust (a region on the heatmap) or fragile (a few isolated pixels)?

4.1 Portfolio builders by date range

We create three small helper functions. Each takes `from_date` and `to_date`, filters the input series, generates signals, and returns a `vbt.Portfolio`.

Key design choice: all strategies use the same `Close` slice for the date window, making comparisons apples-to-apples.

```
def dmac_pf_from_date_range(from_date, to_date):
    # Portfolio from MA crossover, filtered by time range
    range_mask = (ohlcv.index >= from_date) & (ohlcv.index <= to_date)
    range_fast_ma = fast_ma[range_mask]
    range_slow_ma = slow_ma[range_mask]

    dmac_entries = range_fast_ma.ma_crossed_above(range_slow_ma)
    dmac_exits = range_fast_ma.ma_crossed_below(range_slow_ma)

    dmac_pf = vbt.Portfolio.from_signals(
        ohlcv.loc[range_mask, 'Close'],
        dmac_entries,
        dmac_exits
```

```
    )
    return dmac_pf
```

Explanation:

- `range_mask` restricts everything to the selected period.
- `ma_crossed_above` and `ma_crossed_below` generate entry/exit signals at crossover points.
- If your MAs are parameterized (fast/slow window grids), `dmac_pf` contains a whole grid of portfolios.

4.2 Random strategy baseline (same number of entries as DMAC)

To test whether DMAC has signal "edge" beyond mere activity, we randomize entry placement while preserving the entry count.

```
def rand_pf_from_date_range(from_date, to_date):
    # Portfolio from random strategy, filtered by time range
    range_mask = (ohlcv.index >= from_date) & (ohlcv.index <= to_date)
    range_fast_ma = fast_ma[range_mask]
    range_slow_ma = slow_ma[range_mask]

    dmac_entries = range_fast_ma.ma_crossed_above(range_slow_ma)
    dmac_exits = range_fast_ma.ma_crossed_below(range_slow_ma)

    rand_entries = dmac_entries.vbt.signals.shuffle(seed=seed)
    rand_exits = rand_entries.vbt.signals.generate_random_exits(seed=seed)

    rand_pf = vbt.Portfolio.from_signals(
        ohlcv.loc[range_mask, 'Close'],
        rand_entries,
        rand_exits
    )
    return rand_pf
```

Explanation:

- We compute DMAC entries first only to match its number of entries.
- `shuffle` preserves the count of signals but relocates them in time.
- `generate_random_exits` creates exits after entries (so trades close properly).
- With a fixed `seed`, results are reproducible.

4.3 Buy-and-hold baseline

This benchmark enters at the first bar of the selected window and exits at the last bar.

```
def hold_pf_from_date_range(from_date, to_date):
    # Portfolio from holding strategy, filtered by time range
    range_mask = (ohlcv.index >= from_date) & (ohlcv.index <= to_date)

    hold_entries = pd.Series.vbt.signals.empty(
        range_mask.sum(),
        index=ohlcv[range_mask].index
    )
    hold_entries.iloc[0] = True

    hold_exits = pd.Series.vbt.signals.empty_like(hold_entries)
    hold_exits.iloc[-1] = True

    hold_pf = vbt.Portfolio.from_signals(
        ohlcv.loc[range_mask, 'Close'],
        hold_entries,
        hold_exits
    )
    return hold_pf
```

Explanation:

- `empty` creates a boolean signal series (all False).
- We set the first element to True for entry and the last to True for exit.
- This yields a single portfolio for the chosen window.

4.4 Dashboard figures

We use three figures:

1. Price chart (OHLC) to choose the time window.
2. Histogram comparing DMAC vs Random distributions for the chosen `metric`.
3. Heatmap showing DMAC metric across the MA window grid.

4.5 Price chart (the “time selector”)

```
ts_fig = ohlcv.vbt.ohlcv.plot(
    title=symbol,
    show_volume=False,
    annotations=[dict(
        align='left',
        showarrow=False,
        xref='paper',
        yref='paper',
        x=0.5,
        y=0.9,
```

```
        font=dict(size=14),
        bordercolor='black',
        borderwidth=1,
        bgcolor='white'
    )],
    width=700,
    height=250
)
```

What it does:

- Displays the OHLC series.
- Contains a text annotation we'll update to show the hold metric for the active window.
- Later, we attach a callback to react to x-axis range changes.

4.6 Histogram (DMAC vs Random)

```
histogram = vbt.plotting.Histogram(
    trace_names=['Random strategy', 'DMAC strategy'],
    title='%s distribution' % metric,
    xaxis_tickformat='%',
    annotations=[dict(
        y=0,
        xref='x',
        yref='paper',
        showarrow=True,
        arrowcolor="black",
        arrowsize=1,
        arrowwidth=1,
        arrowhead=1,
        xanchor='left',
        text='Hold',
        textangle=0,
        font=dict(size=14),
        bordercolor='black',
        borderwidth=1,
        bgcolor='white',
        ax=0,
        ay=-50,
    )],
    width=700,
    height=250
)
```

What it does:

- Plots two distributions: Random and DMAC, each across the MA parameter grid.

- Marks the buy-and-hold metric value with an arrow labeled “Hold”.

Interpretation:

- If DMAC distribution is clearly shifted right of Random, the rule adds value.
- If DMAC and Random largely overlap, DMAC might be indistinguishable from chance for that window.
- If “Hold” is to the right of most outcomes, buy-and-hold dominates most parameter choices.

4.7 Heatmap (DMAC across the parameter grid)

```
heatmap = vbt.plotting.Heatmap(
    x_labels=np.arange(min_window, max_window + 1),
    y_labels=np.arange(min_window, max_window + 1),
    trace_kwargs=dict(
        colorbar=dict(
            tickformat='%',
            ticks="outside"
        ),
        colorscale='RdBu'
    ),
    title='%s by window' % metric,
    width=650,
    height=420
)
```

What it does:

- Shows the metric surface over fast/slow windows.
- Later we will center the colormap around the hold value, so “hold-level” becomes the neutral midpoint.

4.8 State variables

Because callbacks need access to the most recent computed matrices, we keep them in module-level variables.

```
dmac_perf_matrix = None
rand_perf_matrix = None
hold_value = None
```

4.9 Updating visuals: color scale and histogram

4.10 Center the heatmap around buy-and-hold

We use `hold_value` as the midpoint (`zmid`). That makes the heatmap visually answer: "better or worse than hold?"

```
def update_heatmap_colorscale(perf_matrix):
    # Update heatmap colorscale based on performance matrix
    with heatmap.fig.batch_update():
        heatmap.fig.data[0].zmid = hold_value
        heatmap.fig.data[0].colorbar.tickvals = [
            np.nanmin(perf_matrix),
            hold_value,
            np.nanmax(perf_matrix)
        ]
        heatmap.fig.data[0].colorbar.ticktext = [
            'Min: {:.0%}'.format(np.nanmin(perf_matrix)).ljust(12),
            'Hold: {:.0%}'.format(hold_value).ljust(12),
            'Max: {:.0%}'.format(np.nanmax(perf_matrix)).ljust(12)
        ]
```

4.11 Update histogram with matrices

We flatten both matrices into arrays and update the histogram. We also move the "Hold" arrow to the current hold metric.

```
def update_histogram(dmac_perf_matrix, rand_perf_matrix, hold_value):
    # Update histogram figure
    with histogram.fig.batch_update():
        histogram.update(
            np.asarray([
                rand_perf_matrix.values.flatten(),
                dmac_perf_matrix.values.flatten()
            ]).transpose()
        )
        histogram.fig.layout.annotations[0].x = hold_value
```

4.12 The main recomputation function

This function is called whenever the time window changes on the price chart.

```
def update_figs(from_date, to_date):
    global dmac_perf_matrix, rand_perf_matrix, hold_value

    # Build portfolios
    dmac_pf = dmac_pf_from_date_range(from_date, to_date)
    rand_pf = rand_pf_from_date_range(from_date, to_date)
    hold_pf = hold_pf_from_date_range(from_date, to_date)
```

```
    # Calculate performance
    dmac_perf_matrix = dmac_pf.deep_getattr(metric)
    dmac_perf_matrix = dmac_perf_matrix.vbt.unstack_to_df(
        symmetric=True,
        index_levels='fast_ma_window',
        column_levels='slow_ma_window'
    )

    rand_perf_matrix = rand_pf.deep_getattr(metric)
    rand_perf_matrix = rand_perf_matrix.vbt.unstack_to_df(
        symmetric=True,
        index_levels='fast_ma_window',
        column_levels='slow_ma_window'
    )

    hold_value = hold_pf.deep_getattr(metric)

    # Update figures
    update_histogram(dmac_perf_matrix, rand_perf_matrix, hold_value)

    with ts_fig.batch_update():
        ts_fig.update_xaxes(range=(from_date, to_date))
        ts_fig.layout.annotations[0].text = 'Hold: %.f%%' % (hold_value * 100)

    with heatmap.fig.batch_update():
        heatmap.update(dmac_perf_matrix)
        update_heatmap_colorscale(dmac_perf_matrix.values)
```

What happens here:

- All three portfolios are rebuilt for the active time range.
- The chosen `metric` is extracted.
- DMAC and Random metrics are reshaped into a 2D grid keyed by MA windows.
- Hold remains a scalar reference value.
- All visuals update in sync.

4.13 Interactive callbacks

4.14 Recompute when the time window changes

```
def on_ts_change(layout, x_range):
    global dmac_perf_matrix, rand_perf_matrix, hold_value

    if isinstance(x_range[0], str) and isinstance(x_range[1], str):
```

```
        update_figs(x_range[0], x_range[1])

ts_fig.layout.on_change(on_ts_change, 'xaxis.range')
```

Explanation:

- Plotly provides the axis range as two values; often date strings.
- When the user zooms/drags the price chart, this callback fires and recomputes everything.

4.15 Filter distributions when the heatmap view changes

When you zoom into a region of the heatmap, we filter both matrices to that visible window region and update histogram + color scaling accordingly.

```
def on_heatmap_change(layout, x_range, y_range):
    if dmac_perf_matrix is not None:
        x_mask = (dmac_perf_matrix.columns >= x_range[0]) & (dmac_perf_matrix.
columns <= x_range[1])
        y_mask = (dmac_perf_matrix.index >= y_range[0]) & (dmac_perf_matrix.i
ndex <= y_range[1])

        if x_mask.any() and y_mask.any():
            sub_dmac_perf_matrix = dmac_perf_matrix.loc[y_mask, x_mask]
            sub_rand_perf_matrix = rand_perf_matrix.loc[y_mask, x_mask]

            update_histogram(sub_dmac_perf_matrix, sub_rand_perf_matrix, hold
_value)
            update_heatmap_colorscale(sub_dmac_perf_matrix.values)

heatmap.fig.layout.on_change(on_heatmap_change, 'xaxis.range', 'yaxis.range')
```

Why this matters:

- You can examine whether a “promising” region on the heatmap is genuinely better than random, not just in absolute terms but distributionally.

4.16 Assemble the dashboard

```
dashboard = widgets.VBox([
    ts_fig,
    histogram.fig,
    heatmap.fig
])

dashboard
```

This creates a single interactive object to display in a notebook cell.

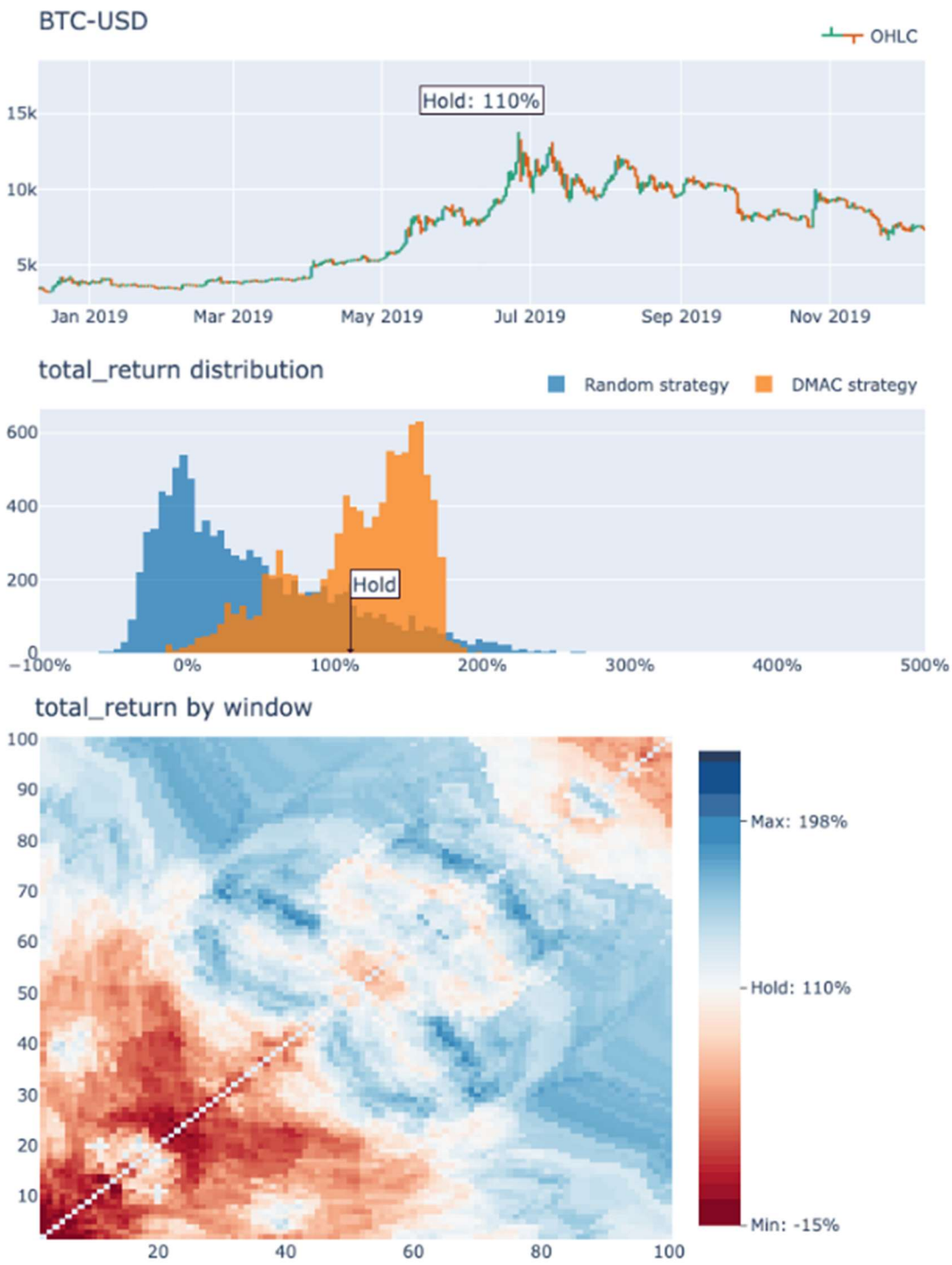

Figure 4.16.1: Interactive dashboard for strategies

4.17 Practical interpretation checklist

When you interact with the dashboard:

1. Choose a regime (time window) on the price chart.
2. Look at the histogram:
    - DMAC shifted right of Random suggests informational value.
    - DMAC overlapping Random suggests little edge.
    - Compare “Hold” to the bulk of outcomes.
3. Look at the heatmap:
    - Broad regions above hold suggest robustness.
    - Isolated hotspots suggest overfitting risk.
4. Zoom into a heatmap region:
    - The histogram updates to reflect only that region.
    - This tells you if “good-looking parameters” remain statistically distinct from random.

5 Walk-Forward Optimization

A parameter sweep can make almost any strategy look good if you search long enough. Walk-forward optimization is the discipline that prevents you from "discovering" parameters that only work because they were fit to one historical regime.

The idea is simple:

1. Split history into many sequential train/test segments.
2. For each segment:
    - optimize parameters on the training slice (in-sample),
    - apply the chosen parameters to the next slice (out-of-sample),
    - record performance out-of-sample.
3. Aggregate out-of-sample results across all segments.

This chapter implements that workflow in vectorbt for a DMAC strategy on BTC-USD, using Sharpe ratio as the metric and a rolling split configuration.

5.1 What we will build

For each rolling split:

- In-sample:
    - compute Sharpe for every fast/slow MA pair in a grid,
    - select the best pair for that split,
    - store the best Sharpe (and median Sharpe for context),
    - compare against Hold.
- Out-of-sample:
    - compute Hold Sharpe,
    - compute median Sharpe over the full grid (for reference),
    - compute "test Sharpe" by applying the in-sample best parameters to that out-of-sample slice.
- Collect everything into a single table and plot it.

5.2 Setup: data, parameters, and split configuration

We use daily close prices and create 30 rolling splits. Each split uses a 2-year in-sample window and a 180-day out-of-sample window.

```
import numpy as np
import pandas as pd
import vectorbt as vbt

# 30 rolling windows, each window is 2 years long, reserve last 180 days for
test (out-sample)
split_kwargs = dict(
    n=30,
    window_len=365 * 2,
    set_lens=(180,),
    left_to_right=False
)

# Portfolio settings
pf_kwargs = dict(
    direction="both",   # long and short
    freq="d"
)

# MA parameter search space
windows = np.arange(10, 50)

# Download data (Close)
price = vbt.YFData.download("BTC-USD").get("Close")

print("Price series:")
print(price)

# Visual sanity check
price.vbt.plot(title="BTC-USD Close").show()
```

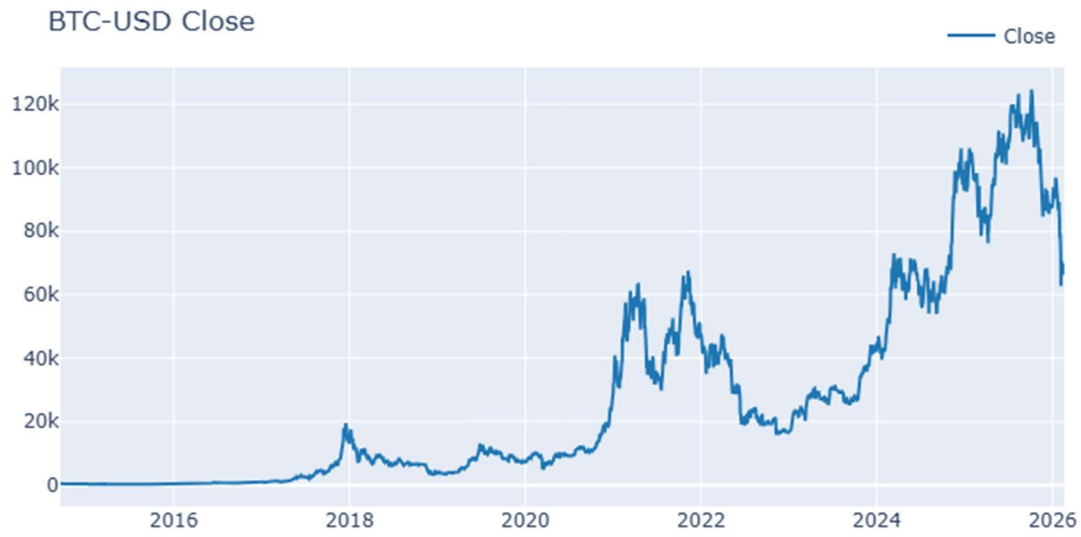

Figure 5.2.1: Bitcoin price chart

Notes:

- `direction="both"` allows long and short positions.
- `windows = np.arange(10, 50)` defines MA windows from 10 to 49.
- We optimize over all 2-combinations `(fast, slow)` in that range.

5.3 Create rolling train/test splits

`rolling_split` returns two datasets (in-sample and out-of-sample), each with a split index that vectorbt can group on.

```
(in_price, in_indexes), (out_price, out_indexes) = price.vbt.rolling_split(**
split_kwargs)

print("\nIn-sample shape / splits:", in_price.shape, len(in_indexes))
print("Out-sample shape / splits:", out_price.shape, len(out_indexes))

# Optional: plot the split ranges (useful for screenshots in the ebook)
price.vbt.rolling_split(
    **split_kwargs,
    plot=True,
    trace_names=["in-sample", "out-sample"]
).show()
```

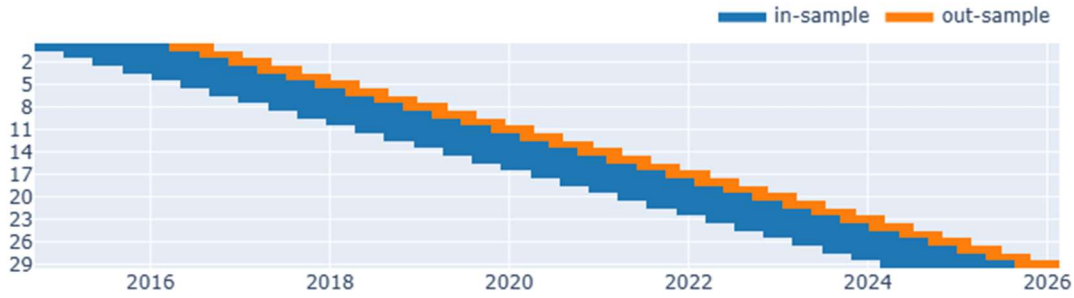

Figure 5.3.1: Walk-forward windows

5.4 Build the baseline (Hold) for each split

To interpret strategy results, you want a benchmark per split. Here we use holding (buy-and-hold) on the same in/out slices.

```
in_hold_pf = vbt.Portfolio.from_holding(in_price, **pf_kwargs)
in_hold_sharpe = in_hold_pf.sharpe_ratio()

out_hold_pf = vbt.Portfolio.from_holding(out_price, **pf_kwargs)
out_hold_sharpe = out_hold_pf.sharpe_ratio()

print("\nIn-sample hold Sharpe (per split):")
print(in_hold_sharpe)

print("\nOut-sample hold Sharpe (per split):")
print(out_hold_sharpe)
```

This gives you a per-split “market difficulty” reference: some periods are simply more favorable than others.

5.5 Optimize strategy parameters in-sample (per split)

We compute fast and slow moving averages for all parameter combinations, generate crossover signals, and build portfolios for every pair.

```
fast_ma, slow_ma = vbt.MA.run_combs(in_price, windows, r=2, short_names=["fas
t", "slow"])
entries = fast_ma.ma_crossed_above(slow_ma)
exits = fast_ma.ma_crossed_below(slow_ma)

in_pf = vbt.Portfolio.from_signals(in_price, entries, exits, **pf_kwargs)
in_sharpe = in_pf.sharpe_ratio()

print("\nIn-sample Sharpe (all window pairs, per split):")
print(in_sharpe)
```

Output:

```
In-sample Sharpe (all window pairs, per split):
fast_window  slow_window  split_idx
10           11           0            0.792561
                          1            0.587436
                          2            0.657334
                          3            0.568536
                          4            0.752250
                                         ...
48           49           25           0.596906
                          26          -0.243144
                          27           0.094382
                          28          -0.329286
                          29           0.085467
Name: sharpe_ratio, Length: 23400, dtype: float64
```

5.6 Selecting the best parameters per split

Now we pick the parameter pair with the maximum Sharpe within each split.

```
in_best_index = in_sharpe[in_sharpe.groupby("split_idx").idxmax()].index

print("\nBest (fast_window, slow_window) index per split:")
print(in_best_index)

in_best_fast_windows = in_best_index.get_level_values("fast_window").to_numpy
()
in_best_slow_windows = in_best_index.get_level_values("slow_window").to_numpy
()

in_best_window_pairs = np.array(list(zip(in_best_fast_windows, in_best_slow_w
indows)))
print("\nBest window pairs per split:")
print(in_best_window_pairs)
```

A useful visualization is how chosen parameters drift across time:

```
pd.DataFrame(in_best_window_pairs, columns=["fast_window", "slow_window"]).vb
t.plot(
    title="Selected in-sample best parameters per split"
).show()
```

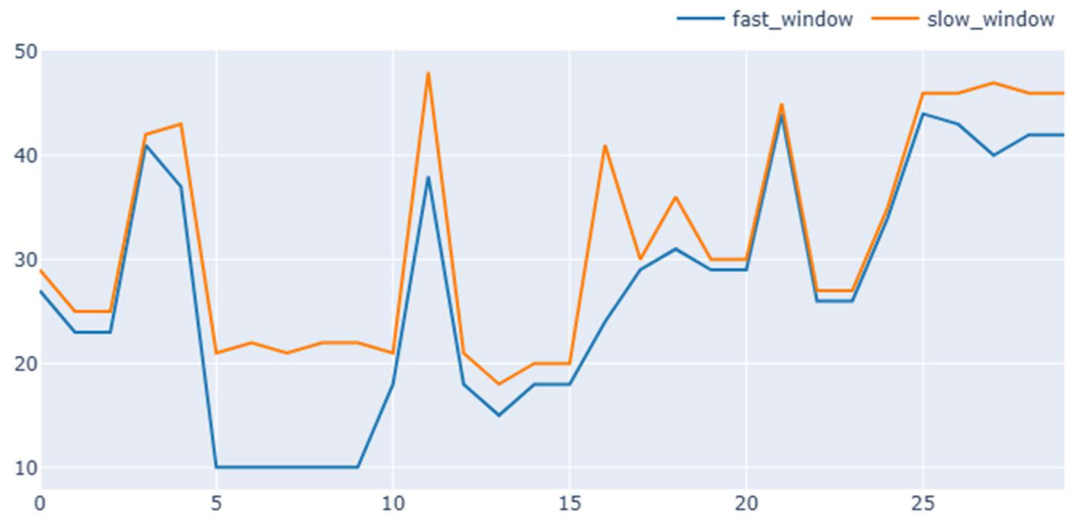

Figure 5.6.1: Selected in-sample best parameters per window

5.7 Context: best vs median in-sample performance

It helps to track both:

- "best" Sharpe (what you would pick),
- "median" Sharpe (what a typical parameter pair looks like).

```
in_best_sharpe = in_sharpe[in_best_index]
in_median_sharpe = in_sharpe.groupby("split_idx").median()
```

If the best Sharpe is only slightly above the median, the edge is weak. If best is far above median, it may be a sign of overfitting (especially if it fails out-of-sample).

5.8 Out-of-sample grid performance (reference only)

This is not the walk-forward test yet. This is just measuring how the parameter grid behaves out-of-sample so you can compare against the walk-forward-selected parameters.

```
fast_ma_out, slow_ma_out = vbt.MA.run_combs(out_price, windows, r=2, short_na
mes=["fast", "slow"])
entries_out = fast_ma_out.ma_crossed_above(slow_ma_out)
exits_out = fast_ma_out.ma_crossed_below(slow_ma_out)

out_pf_all = vbt.Portfolio.from_signals(out_price, entries_out, exits_out, **
pf_kwargs)
out_sharpe = out_pf_all.sharpe_ratio()

print("\nOut-sample Sharpe (all window pairs, per split):")
print(out_sharpe)
```

```
out_median_sharpe = out_sharpe.groupby("split_idx").median()
```

This gives you “typical out-of-sample DMAC behavior” per split.

5.9 The actual walk-forward test

This is the core WFO step:

- take the best in-sample parameters for split *i*,
- apply them to out-of-sample slice *i*,
- compute out-of-sample Sharpe.

```
fast_ma_test = vbt.MA.run(out_price, window=in_best_fast_windows, per_column=
True)
slow_ma_test = vbt.MA.run(out_price, window=in_best_slow_windows, per_column=
True)
entries_test = fast_ma_test.ma_crossed_above(slow_ma_test)
exits_test = fast_ma_test.ma_crossed_below(slow_ma_test)

out_pf_test = vbt.Portfolio.from_signals(out_price, entries_test, exits_test,
 **pf_kwargs)
out_test_sharpe = out_pf_test.sharpe_ratio()

print("\nOut-sample TEST Sharpe (best in-sample params applied out-of-sample):
")
print(out_test_sharpe)
```

`per_column=True` is critical here: each split column uses its own chosen window pair.

This is the difference between “parameter tuning” and “walk-forward validation”.

5.10 Aggregate and plot walk-forward results

Finally, we create a per-split table that contains:

- in-sample: hold, median, best
- out-of-sample: hold, median, test (walk-forward)

```
cv_results_df = pd.DataFrame({
    "in_sample_hold": in_hold_sharpe.values,
    "in_sample_median": in_median_sharpe.values,
    "in_sample_best": in_best_sharpe.values,
    "out_sample_hold": out_hold_sharpe.values,
    "out_sample_median": out_median_sharpe.values,
    "out_sample_test": out_test_sharpe.values
})
```

```
print("\nWalk-forward results table (per split):")
print(cv_results_df)
```

Plotting:

```
# Plot comparison (useful screenshot)
color_schema = vbt.settings["plotting"]["color_schema"]

fig = cv_results_df.vbt.plot(
    title="Walk-forward evaluation: in-sample vs out-sample",
    trace_kwargs=[
        dict(line_color=color_schema["blue"]),
        dict(line_color=color_schema["blue"], line_dash="dash"),
        dict(line_color=color_schema["blue"], line_dash="dot"),
        dict(line_color=color_schema["orange"]),
        dict(line_color=color_schema["orange"], line_dash="dash"),
        dict(line_color=color_schema["orange"], line_dash="dot"),
    ],
    return_fig=True
)

fig.update_layout(
    margin=dict(t=150, r=20, b=40, l=60),
    legend=dict(
        orientation="h",
        x=0,
        y=1.15,
        xanchor="left",
        yanchor="bottom"
    )
)

fig.show()
```

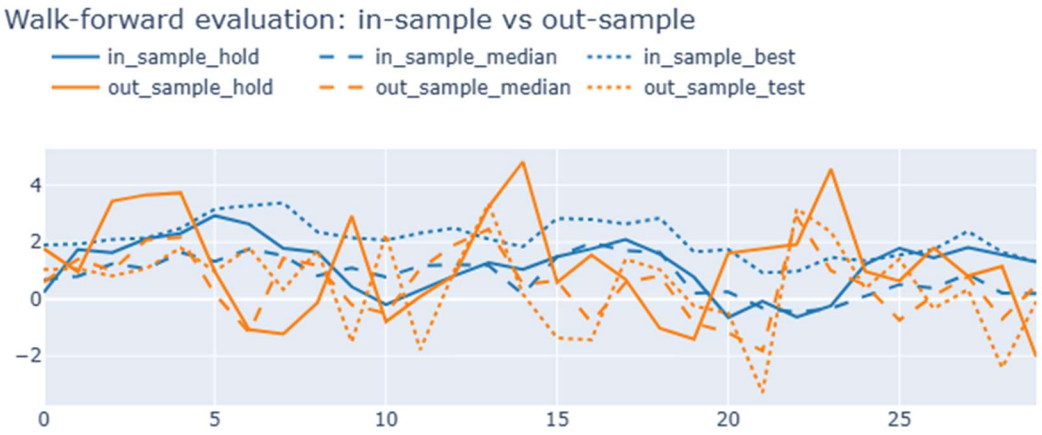

Figure 5.10.1: Walk-forward Sharpe ratios

5.11 Interpreting the result correctly

If your in-sample “best” line is consistently high but your out-of-sample “test” line is mediocre or unstable, you have classic overfitting.

What you want to see:

- Out-of-sample test Sharpe that is reasonably stable across splits.
- Out-of-sample test Sharpe that beats Hold often enough to be meaningful.
- Parameter selections that are not wildly chaotic (unless the market regime truly changes that hard).

Even then, this is still “research-grade validation,” not a live-trading guarantee. The next realism upgrades are fees, slippage, and execution timing assumptions (for example, “signal at close, trade next open”).

No. The previous chapter draft was not finished and did not match the notebook’s later sections (the dynamic “search + rebalance” part in particular). Below is the completed chapter, rebuilt directly from the notebook, with every code cell included in the text.

6 Portfolio Optimization

Portfolio optimization is the process of converting a universe of assets into a set of portfolio weights under an explicit objective and explicit constraints. In this notebook, the objective is risk-adjusted performance (Sharpe ratio), and the constraints are long-only and fully invested (weights sum to 1, no shorting).

This chapter implements two optimization workflows:

1. Random-search optimization (generate many candidate weight vectors, pick the best by Sharpe), and
2. Mean-variance optimization via PyPortfolioOpt (Efficient Frontier max-Sharpe weights).

Both workflows are tested under three allocation regimes:

- One-time allocation (buy-and-hold with drift),
- Fixed-schedule rebalancing (monthly),
- "Optimize then rebalance" every 30 days (weights re-computed periodically using only past data).

The key discipline is keeping the information timeline consistent:

- Weights must be computed using data available strictly before the rebalance decision.
- Trades that enforce weights must respect execution ordering inside the rebalance bar (sell before buy), otherwise the simulation can accidentally require temporary leverage.

6.1 Setup

Import dependencies. vectorbt is used for simulation and visualization. PyPortfolioOpt is used for mean-variance weights. Numba is used to make the rolling optimizer fast inside vectorbt's low-level engine.

```
import os
import numpy as np
import pandas as pd
import yfinance as yf
from datetime import datetime
import pytz
from numba import njit

from pypfopt.efficient_frontier import EfficientFrontier
```

```
from pypfopt import risk_models
from pypfopt import expected_returns
from pypfopt import base_optimizer

import vectorbt as vbt
from vectorbt.generic.nb import nanmean_nb
from vectorbt.portfolio.nb import order_nb, sort_call_seq_nb
from vectorbt.portfolio.enums import SizeType, Direction
```

Define the universe, the backtest window, and how many random portfolios to test. Also set vectorbt global settings for frequency and annualization (needed for metrics like Sharpe).

```
# Define params
symbols = ['FB', 'AMZN', 'NFLX', 'GOOG', 'AAPL']
start_date = datetime(2017, 1, 1, tzinfo=pytz.utc)
end_date = datetime(2020, 1, 1, tzinfo=pytz.utc)
num_tests = 2000

vbt.settings.array_wrapper['freq'] = 'days'
vbt.settings.returns['year_freq'] = '252 days'
vbt.settings.portfolio['seed'] = 42
vbt.settings.portfolio.stats['incl_unrealized'] = True
```

Download Yahoo Finance data through vectorbt's data wrapper, then concatenate OHLCV. We use Close as the traded price series throughout this notebook.

```
yfdata = vbt.YFData.download(symbols, start=start_date, end=end_date)

print(yfdata.symbols)

ohlcv = yfdata.concat()

print(ohlcv.keys())

price = ohlcv['Close']
```

Plot normalized price series (each divided by its first value) to compare relative performance.

```
# Plot normalized price series
(price / price.iloc[0]).vbt.plot().show()
```

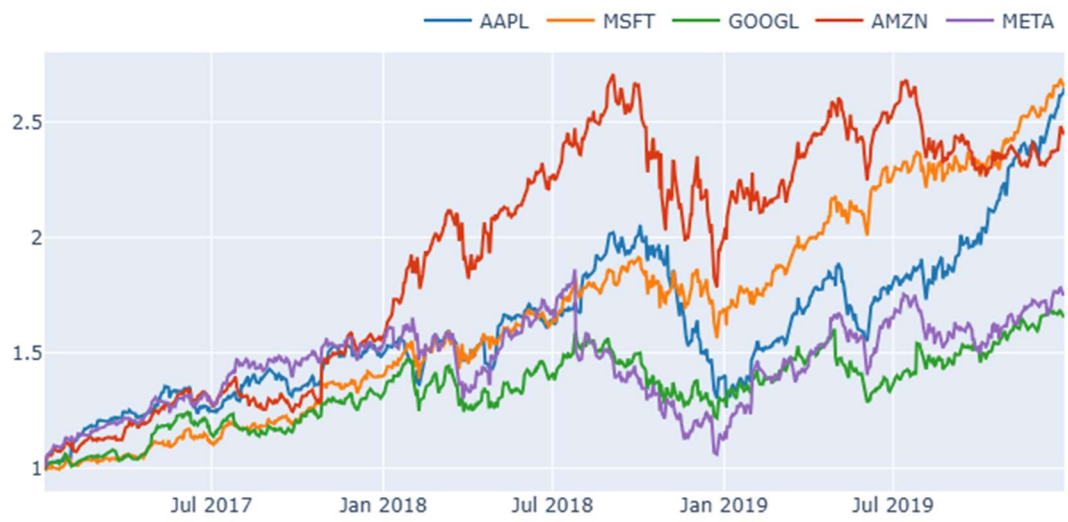

Figure 6.1.1: Normalized price series

Compute daily returns and basic summary statistics (mean, std, correlation). This is not the optimizer yet, but it frames the raw properties of the assets.

```
returns = price.pct_change()

print(returns.mean())

print(returns.std())

print(returns.corr())
```

6.2 Random-search optimization (vectorbt)

Random search generates many feasible weight vectors, simulates each candidate, and selects the best by a target metric (Sharpe ratio in this notebook). This is not "closed form" optimization; it is a Monte-Carlo approximation of the objective surface.

6.2.1 Step 1: generate random weights

Generate `num_tests` weight vectors, each long-only and normalized to sum to 1.

```
np.random.seed(42)

# Generate random weights, n times
weights = []
for i in range(num_tests):
    w = np.random.random_sample(len(symbols))
    w = w / np.sum(w)
    weights.append(w)

print(len(weights))
```

6.2.2 Step 2: restructure price data to evaluate all candidates at once

To backtest all weight vectors in one vectorized run, the notebook "tiles" the price DataFrame `num_tests` times (one group per candidate), then stacks an additional index level that stores the weights.

```
# Build column hierarchy such that one weight corresponds to one price series
_price = price.vbt.tile(num_tests, keys=pd.Index(np.arange(num_tests), name='
symbol_group'))
_price = _price.vbt.stack_index(pd.Index(np.concatenate(weights), name='weigh
ts'))

print(_price.columns)
```

6.2.3 Step 3: one-time allocation (buy-and-hold with drift)

Create an order size array that is NaN everywhere except at the first timestamp, where it contains the target weights. With `size_type='targetpercent'`, vectorbt interprets each size as a target portfolio weight.

```
# Define order size
size = np.full_like(_price, np.nan)
size[0, :] = np.concatenate(weights)  # allocate at first timestamp, do nothi
ng afterwards

print(size.shape)
```

Run the portfolio simulation. `cash_sharing=True` ensures all assets inside a candidate portfolio share the same cash pool.

```
# Run simulation
pf = vbt.Portfolio.from_orders(
    close=_price,
    size=size,
    size_type='targetpercent',
    group_by='symbol_group',
    cash_sharing=True
) # all weights sum to 1, no shorting, and 100% investment in risky assets

print(len(pf.orders))
```

Visualize the "candidate cloud": annualized return vs annualized volatility, colored by Sharpe ratio.

```
# Plot annualized return against volatility, color by sharpe ratio
annualized_return = pf.annualized_return()
annualized_return.index = pf.annualized_volatility()
annualized_return.vbt.scatterplot(
    trace_kwargs=dict(
        mode='markers',
```

```
        marker=dict(
            color=pf.sharpe_ratio(),
            colorbar=dict(
                title='sharpe_ratio'
            ),
            size=5,
            opacity=0.7
        )
    ),
    xaxis_title='annualized_volatility',
    yaxis_title='annualized_return'
).show()
```

Figure 6.2.1: Feasible portfolios

Select the best candidate by Sharpe ratio and inspect its weights and stats.

```
# Get index of the best group according to the target metric
best_symbol_group = pf.sharpe_ratio().idxmax()

print(best_symbol_group)

# Print best weights
print(weights[best_symbol_group])

# Compute default stats
print(pf.iloc[best_symbol_group].stats())
```

6.2.4 Step 4: fixed monthly rebalancing

One-time allocation is the simplest case, but real portfolios drift away from target weights as prices move. Monthly rebalancing forces weights back to target on the first index of each month.

Build a monthly rebalance mask.

```
# Select the first index of each month
rb_mask = ~_price.index.to_period('m').duplicated()

print(rb_mask.sum())
```

Build the size array: assign the same target weights at each rebalance date.

```
rb_size = np.full_like(_price, np.nan)
rb_size[rb_mask, :] = np.concatenate(weights)  # allocate at mask

print(rb_size.shape)
```

Run the monthly rebalancing simulation. The notebook sets `call_seq='auto'` to enforce correct ordering (sell before buy) during rebalances.

```
# Run simulation, with rebalancing monthly
rb_pf = vbt.Portfolio.from_orders(
    close=_price,
    size=rb_size,
    size_type='targetpercent',
    group_by='symbol_group',
    cash_sharing=True,
    call_seq='auto'  # important: sell before buy
)

print(len(rb_pf.orders))
```

Select the best candidate under monthly rebalancing.

```
rb_best_symbol_group = rb_pf.sharpe_ratio().idxmax()

print(rb_best_symbol_group)

print(weights[rb_best_symbol_group])

print(rb_pf.iloc[rb_best_symbol_group].stats())
```

Output:

```
Start                         2017-01-03 05:00:00+00:00
End                           2019-12-31 05:00:00+00:00
Period                                754 days 00:00:00
Start Value                                       100.0
End Value                                    266.259217
Total Return [%]                             166.259217
Benchmark Return [%]                         123.439824
Max Gross Exposure [%]                            100.0
Total Fees Paid                                     0.0
Max Drawdown [%]                              24.600494
```

```
Max Drawdown Duration                    133 days 00:00:00
Total Trades                                            87
Total Closed Trades                                     82
Total Open Trades                                        5
Open Trade PnL                                  140.181523
Win Rate [%]                                     98.780488
Best Trade [%]                                  143.996371
Worst Trade [%]                                  -5.166532
Avg Winning Trade [%]                            46.177825
Avg Losing Trade [%]                             -5.166532
Avg Winning Trade Duration    353 days 01:11:06.666666664
Avg Losing Trade Duration                502 days 00:00:00
Profit Factor                                 1614.724666
Expectancy                                       0.318021
Sharpe Ratio                                     1.690072
Calmar Ratio                                     1.574019
Omega Ratio                                       1.36125
Sortino Ratio                                    2.498455
```

Plot the allocation through time (stacked area). Rebalance dates are marked as vertical lines.

```
def plot_allocation(rb_pf):
    # Plot weights development of the portfolio
    rb_asset_value = rb_pf.asset_value(group_by=False)
    rb_value = rb_pf.value()
    rb_idxs = np.flatnonzero((rb_pf.asset_flow() != 0).any(axis=1))
    rb_dates = rb_pf.wrapper.index[rb_idxs]
    fig = (rb_asset_value.vbt / rb_value).vbt.plot(
        trace_names=symbols,
        trace_kwargs=dict(
            stackgroup='one'
        )
    )
    for rb_date in rb_dates:
        fig.add_shape(
            dict(
                xref='x',
                yref='paper',
                x0=rb_date,
                x1=rb_date,
                y0=0,
                y1=1,
                line_color=fig.layout.template.layout.plot_bgcolor
            )
        )
    fig.show()

plot_allocation(rb_pf.iloc[rb_best_symbol_group])  # best group
```

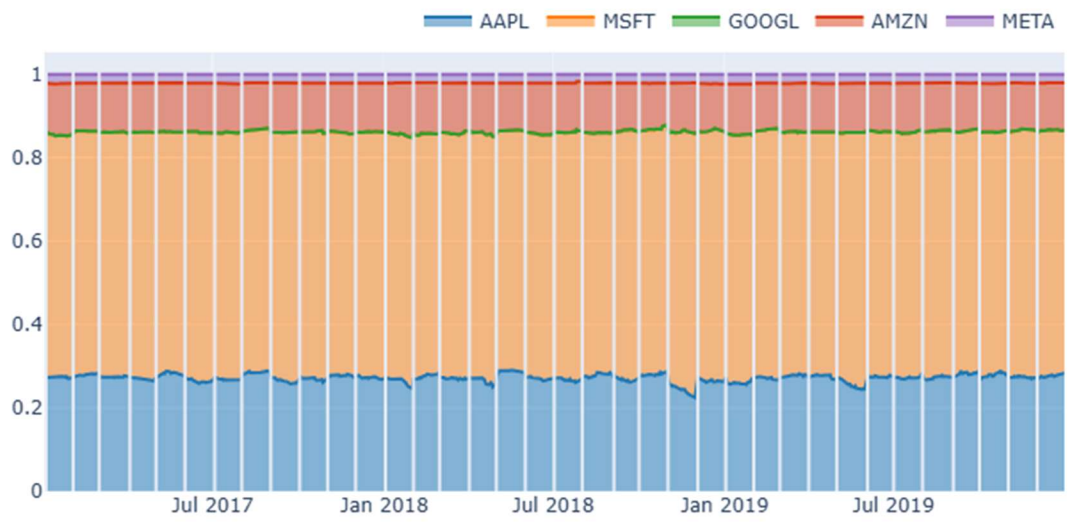

Figure 6.2.2: Portfolio allocation stack plot

6.3 Optimize then rebalance every 30 days (rolling optimization)

The previous sections tested fixed weights (choose once, or choose once and keep rebalancing back to them). This section changes the problem:

- Every 30 days, recompute the "best" weights using historical data up to that point.
- Apply those weights as target-percent allocations during that rebalance.
- Repeat.

This is closer to what many people mean by "portfolio optimization" in practice: weights are updated on a schedule.

The notebook implements this using vectorbt's low-level `from_order_func` API, where you can define:

- a pre-simulation function that defines rebalance segments,
- a pre-segment function that computes the optimal weights for the segment,
- an order function that applies target-percent orders.

6.3.1 Diagnostics container

Track the best Sharpe ratio found at each rebalance point.

```
srb_sharpe = np.full(price.shape[0], np.nan)
```

Define the segment mask: rebalance every `every_nth` bars.

```
@njit
def pre_sim_func_nb(c, every_nth):
    # Define rebalancing days
    c.segment_mask[:, :] = False
    c.segment_mask[every_nth::every_nth, :] = True
    return ()
```

Define the random-search optimizer (Numba). This computes returns from the price history slice, estimates mean and covariance, then runs random weight trials and selects the best Sharpe.

```
@njit
def find_weights_nb(c, price, num_tests):
    # Find optimal weights based on best Sharpe ratio
    returns = (price[1:] - price[:-1]) / price[:-1]
    returns = returns[1:, :]  # cannot compute np.cov with NaN
    mean = nanmean_nb(returns)
    cov = np.cov(returns, rowvar=False)  # masked arrays not supported by Num
ba (yet)
    best_sharpe_ratio = -np.inf
    weights = np.full(c.group_len, np.nan, dtype=np.float64)

    for i in range(num_tests):
        # Generate weights
        w = np.random.random_sample(c.group_len)
        w = w / np.sum(w)

        # Compute annualized mean, covariance, and Sharpe ratio
        p_return = np.sum(mean * w) * ann_factor
        p_std = np.sqrt(np.dot(w.T, np.dot(cov, w))) * np.sqrt(ann_factor)
        sharpe_ratio = p_return / p_std
        if sharpe_ratio > best_sharpe_ratio:
            best_sharpe_ratio = sharpe_ratio
            weights = w

    return best_sharpe_ratio, weights
```

Define the pre-segment function: slice the available history, compute best weights, store the diagnostic Sharpe, and enforce correct order sequencing.

Two modes exist here:

- `history_len = -1`: look back over the entire available history (from start to current rebalance point),
- `history_len = 252`: look back only 252 bars (about one trading year).

```
@njit
def pre_segment_func_nb(c, find_weights_nb, history_len, ann_factor, num_test
s, srb_sharpe):
```

```
    if history_len == -1:
        # Look back at the entire time period
        close = c.close[:c.i, c.from_col:c.to_col]
    else:
        # Look back at a fixed time period
        if c.i - history_len <= 0:
            return (np.full(c.group_len, np.nan),)  # insufficient data
        close = c.close[c.i - history_len:c.i, c.from_col:c.to_col]

    # Find optimal weights
    best_sharpe_ratio, weights = find_weights_nb(c, close, num_tests)
    srb_sharpe[c.i] = best_sharpe_ratio

    # Update valuation price and reorder orders
    size_type = SizeType.TargetPercent
    direction = Direction.LongOnly
    order_value_out = np.empty(c.group_len, dtype=np.float64)
    for k in range(c.group_len):
        col = c.from_col + k
        c.last_val_price[col] = c.close[c.i, col]
    sort_call_seq_nb(c, weights, size_type, direction, order_value_out)

    return (weights,)
```

Define the order function that applies target-percent sizing.

```
@njit
def order_func_nb(c, weights):
    col_i = c.call_seq_now[c.call_idx]
    return order_nb(
        weights[col_i],
        c.close[c.i, c.col],
        size_type=SizeType.TargetPercent
    )
```

Get the annualization factor used by vectorbt returns.

```
ann_factor = returns.vbt.returns.ann_factor
```

6.3.2 Run: optimize every 30 days using all history

Run the simulation. `group_by=True` means the assets are treated as a group (one portfolio) and weights are solved jointly.

```
# Run simulation using a custom order function
srb_pf = vbt.Portfolio.from_order_func(
    price,
    order_func_nb,
    pre_sim_func_nb=pre_sim_func_nb,
    pre_sim_args=(30,),
```

```
    pre_segment_func_nb=pre_segment_func_nb,
    pre_segment_args=(find_weights_nb, -1, ann_factor, num_tests, srb_sharpe),
    cash_sharing=True,
    group_by=True
)
```

Plot the best Sharpe ratio achieved at each rebalance point.

```
# Plot best Sharpe ratio at each rebalancing day
pd.Series(srb_sharpe, index=price.index).vbt.scatterplot(trace_kwargs=dict(mo
de='markers')).show()
```

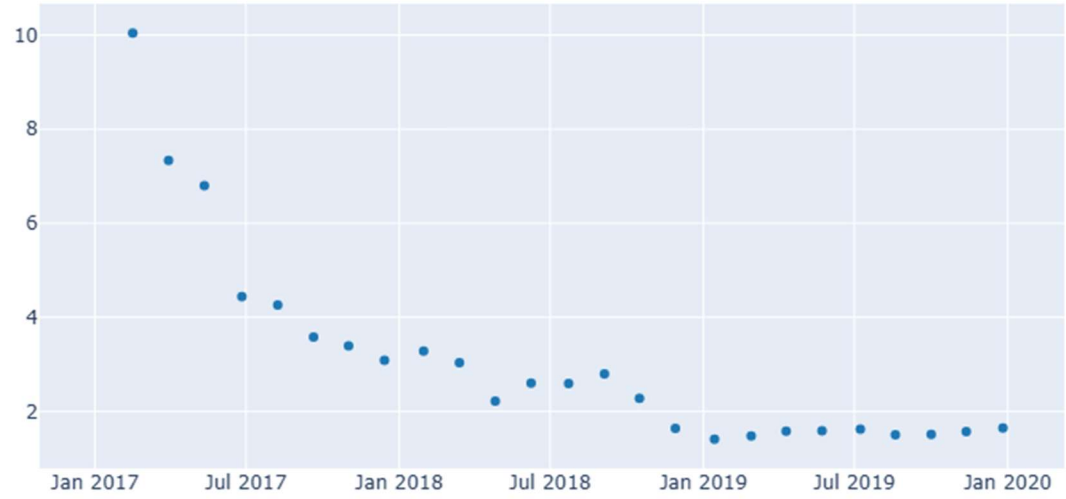

Figure 6.3.1: Sharpe ratios at rebalance points

Inspect portfolio stats and allocation.

```
print(srb_pf.stats())
```

Output:

```
Start                         2017-01-03 05:00:00+00:00
End                           2019-12-31 05:00:00+00:00
Period                                754 days 00:00:00
Start Value                                       100.0
End Value                                    193.010806
Total Return [%]                              93.010806
Benchmark Return [%]                         123.439824
Max Gross Exposure [%]                            100.0
Total Fees Paid                                     0.0
Max Drawdown [%]                             30.573129
Max Drawdown Duration                 211 days 00:00:00
Total Trades                                         64
Total Closed Trades                                  59
Total Open Trades                                     5
```

```
Open Trade PnL                                      38.482935
Win Rate [%]                                        83.050847
Best Trade [%]                                      37.060253
Worst Trade [%]                                    -14.619163
Avg Winning Trade [%]                               10.927445
Avg Losing Trade [%]                                -6.088488
Avg Winning Trade Duration    358 days 03:55:06.122448980
Avg Losing Trade Duration               405 days 00:00:00
Profit Factor                                        6.455273
Expectancy                                           0.924201
Sharpe Ratio                                         1.107023
Calmar Ratio                                         0.803956
Omega Ratio                                          1.228597
Sortino Ratio                                        1.567907
```

```
plot_allocation(srb_pf)
```

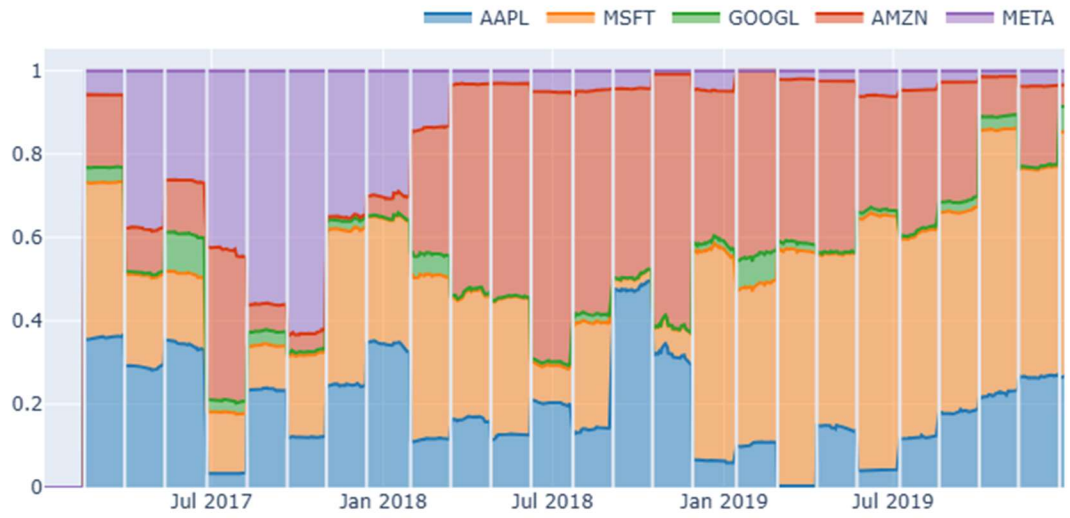

Figure 6.3.2: Portfolio allocation stack plot

6.3.3 Run: optimize every 30 days using a 252-day lookback

Same process, but weights are estimated only from the last 252 days at each rebalance.

```
# Run simulation, but now consider only the last 252 days of data
srb252_sharpe = np.full(price.shape[0], np.nan)

srb252_pf = vbt.Portfolio.from_order_func(
    price,
    order_func_nb,
    pre_sim_func_nb=pre_sim_func_nb,
    pre_sim_args=(30,),
    pre_segment_func_nb=pre_segment_func_nb,
    pre_segment_args=(find_weights_nb, 252, ann_factor, num_tests, srb252_sha
```

```
rpe),
    cash_sharing=True,
    group_by=True
)
```

Plot diagnostics, stats, and allocation.

```
pd.Series(srb252_sharpe, index=price.index).vbt.scatterplot(trace_kwargs=dict
(mode='markers')).show()
```

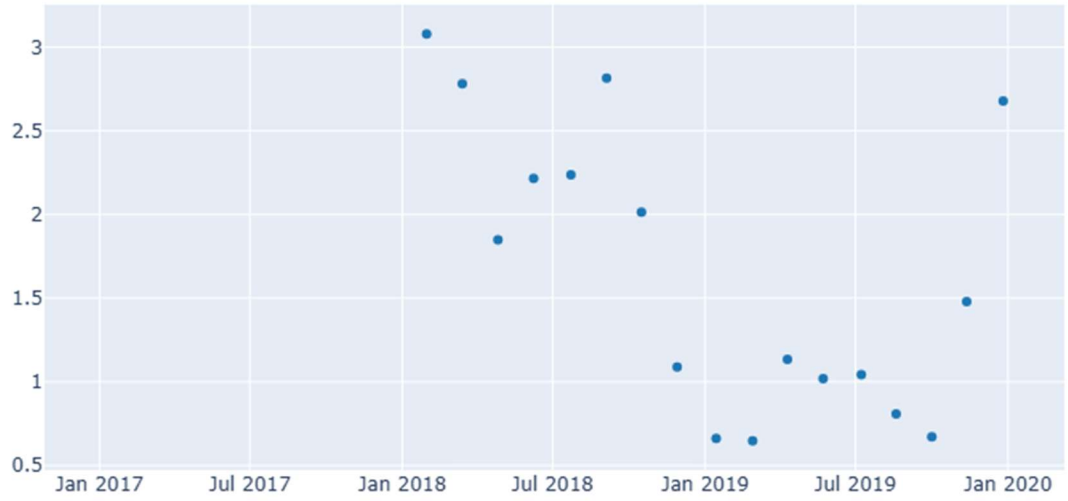

Figure 6.3.3: Sharpe ratios at rebalance points

```
print(srb252_pf.stats())
```

output:

```
Start                         2017-01-03 05:00:00+00:00
End                           2019-12-31 05:00:00+00:00
Period                               754 days 00:00:00
Start Value                                      100.0
End Value                                   152.699061
Total Return [%]                             52.699061
Benchmark Return [%]                        123.439824
Max Gross Exposure [%]                           100.0
Total Fees Paid                                    0.0
Max Drawdown [%]                             30.869013
Max Drawdown Duration                211 days 00:00:00
Total Trades                                        47
Total Closed Trades                                 42
Total Open Trades                                    5
Open Trade PnL                               22.669604
Win Rate [%]                                 73.809524
Best Trade [%]                               42.246782
Worst Trade [%]                             -27.286833
```

```
Avg Winning Trade [%]                          13.414206
Avg Losing Trade [%]                          -10.932123
Avg Winning Trade Duration    285 days 11:36:46.451612904
Avg Losing Trade Duration             240 days 00:00:00
Profit Factor                                  4.378897
Expectancy                                     0.714987
Sharpe Ratio                                   0.793423
Calmar Ratio                                   0.492307
Omega Ratio                                    1.191518
Sortino Ratio                                  1.119308
```

```
plot_allocation(srb252_pf)
```

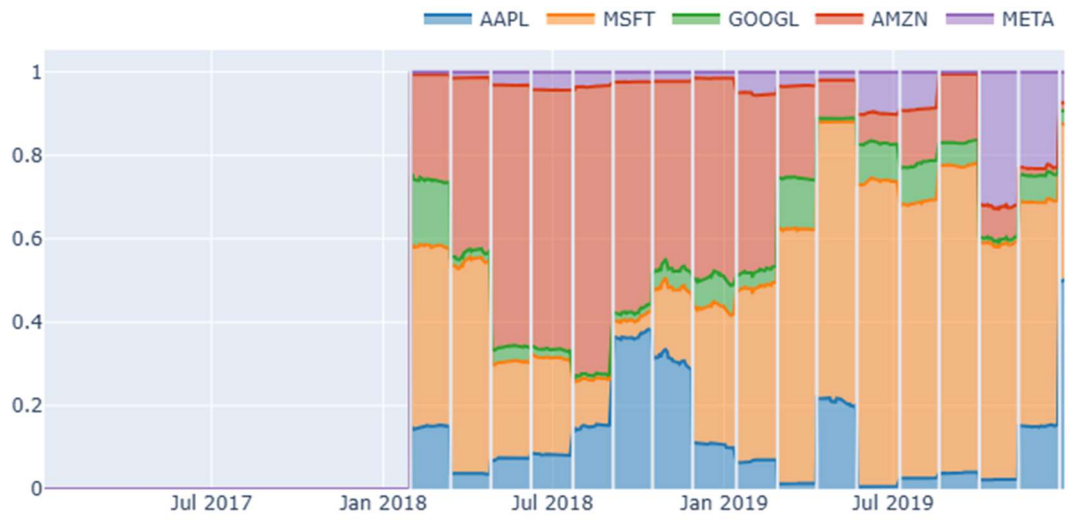

Figure 6.3.4: Portfolio allocation

6.4 PyPortfolioOpt mean-variance optimization

Random search is objective-driven and simple, but mean-variance optimization provides a standard baseline: estimate expected returns and covariance, then solve for max-Sharpe weights on the efficient frontier.

6.4.1 One-time PyPortfolioOpt weights

Compute expected returns and covariance matrix from the full sample, then solve max Sharpe and extract clean weights in the same order as `symbols`.

```
# Calculate expected returns and sample covariance amtrix
avg_returns = expected_returns.mean_historical_return(price)
cov_mat = risk_models.sample_cov(price)

# Get weights maximizing the Sharpe ratio
ef = EfficientFrontier(avg_returns, cov_mat)
weights = ef.max_sharpe()
```

```
clean_weights = ef.clean_weights()
pyopt_weights = np.array([clean_weights[symbol] for symbol in symbols])

print(pyopt_weights)
```

Allocate once at the beginning and simulate.

```
pyopt_size = np.full_like(price, np.nan)
pyopt_size[0, :] = pyopt_weights  # allocate at first timestamp, do nothing a
fterwards

print(pyopt_size.shape)

# Run simulation with weights from PyPortfolioOpt
pyopt_pf = vbt.Portfolio.from_orders(
    close=price,
    size=pyopt_size,
    size_type='targetpercent',
    group_by=True,
    cash_sharing=True
)

print(len(pyopt_pf.orders))

print(pyopt_pf.stats())
```

6.4.2 PyPortfolioOpt optimize then rebalance every 30 days

PyPortfolioOpt itself cannot run inside Numba. The notebook keeps the same vectorbt structure but executes the pre-segment step as pure Python. The optimizer function returns both the best Sharpe ratio and the weight vector.

```
def pyopt_find_weights(sc, price, num_tests):  # no @njit decorator = it's a
pure Python function
    # Calculate expected returns and sample covariance matrix
    price = pd.DataFrame(price, columns=symbols)
    avg_returns = expected_returns.mean_historical_return(price)
    cov_mat = risk_models.sample_cov(price)

    # Get weights maximizing the Sharpe ratio
    ef = EfficientFrontier(avg_returns, cov_mat)
    weights = ef.max_sharpe()
    clean_weights = ef.clean_weights()
    weights = np.array([clean_weights[symbol] for symbol in symbols])
    best_sharpe_ratio = base_optimizer.portfolio_performance(weights, avg_ret
urns, cov_mat)[2]

    return best_sharpe_ratio, weights
```

Diagnostics container.

```
pyopt_srb_sharpe = np.full(price.shape[0], np.nan)
```

Run the simulation. Note the two critical flags:

- `pre_segment_func_nb=pre_segment_func_nb.py_func` runs the segment logic as Python,
- `use_numba=False` runs the simulator as Python, which is required when the optimization step is Python.

```
# Run simulation with a custom order function
pyopt_srb_pf = vbt.Portfolio.from_order_func(
    price,
    order_func_nb,
    pre_sim_func_nb=pre_sim_func_nb,
    pre_sim_args=(30,),
    pre_segment_func_nb=pre_segment_func_nb.py_func,  # run pre_segment_func_
nb as pure Python function
    pre_segment_args=(pyopt_find_weights, -1, ann_factor, num_tests, pyopt_sr
b_sharpe),
    cash_sharing=True,
    group_by=True,
    use_numba=False  # run simulate_nb as pure Python function
)
```

Plot Sharpe at rebalance points, then inspect stats and allocation.

```
pd.Series(pyopt_srb_sharpe, index=price.index).vbt.scatterplot(trace_kwargs=d
ict(mode='markers')).show()
```

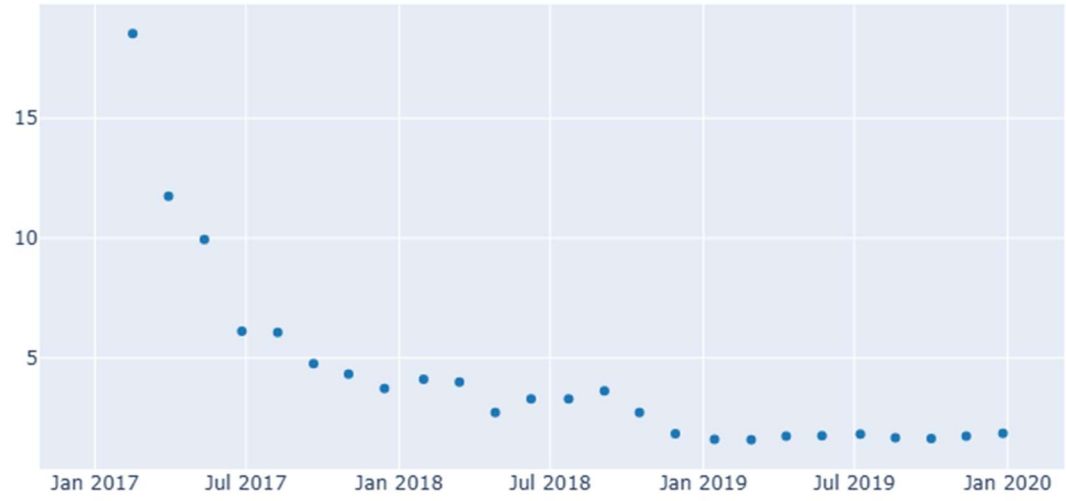

Figure 6.4.1: Sharpe ratios at rebalance points

```
print(pyopt_srb_pf.stats())
```

Output:

```
Start                          2017-01-03 05:00:00+00:00
End                            2019-12-31 05:00:00+00:00
Period                                 754 days 00:00:00
Start Value                                        100.0
End Value                                     189.311808
Total Return [%]                               89.311808
Benchmark Return [%]                          123.439824
Max Gross Exposure [%]                             100.0
Total Fees Paid                                      0.0
Max Drawdown [%]                               33.357502
Max Drawdown Duration                  299 days 00:00:00
Total Trades                                          37
Total Closed Trades                                   35
Total Open Trades                                      2
Open Trade PnL                                41.841905
Win Rate [%]                                  82.857143
Best Trade [%]                                33.171332
Worst Trade [%]                              -27.022295
Avg Winning Trade [%]                         12.113996
Avg Losing Trade [%]                         -11.950639
Avg Winning Trade Duration    199 days 15:43:26.896551724
Avg Losing Trade Duration              240 days 00:00:00
Profit Factor                                  5.372832
Expectancy                                     1.356283
Sharpe Ratio                                    1.04776
Calmar Ratio                                   0.712773
Omega Ratio                                    1.212698
Sortino Ratio                                  1.474522
```

```
plot_allocation(pyopt_srb_pf)
```

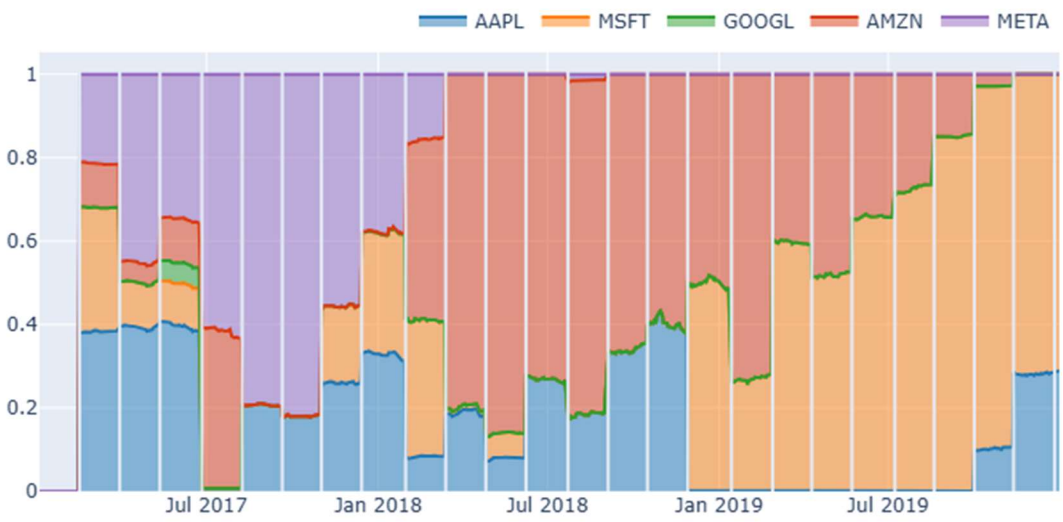

Figure 6.4.2: Portfolio allocations

6.5 Interpreting optimization results responsibly

Optimization can look “precise” but is often unstable. Practical checks that matter more than a single best Sharpe:

- Weight stability: do weights drift slowly, or flip aggressively at each rebalance?
- Turnover implied by rebalancing: frequent large weight changes amplify fees/slippage sensitivity.
- Lookback sensitivity: compare “all history” vs “252 days.” If conclusions change drastically, the solution is regime-dependent.
- Out-of-sample logic: rolling optimization is still in-sample at each step unless you explicitly forward-test with a proper walk-forward design.

In this notebook, you have three progressively more realistic models:

- fixed weights (baseline),
- fixed schedule rebalancing (controls drift),
- rolling optimization (reactive but potentially unstable).

The correct next step is to add cost assumptions (fees, slippage) and test whether any apparent edge survives under tighter execution realism.

7 Portfolio Allocation and Risk Management

This chapter is a walkthrough of two allocation policies that sit “below” your signal logic:

1. Equal-dollar allocation (baseline: “same dollars per trade”).
2. Volatility-targeted allocation (risk management: “same risk per trade”).

Both scripts use the same signal engine (EMA 7 vs EMA 21 crossover). That is deliberate: it isolates the effect of allocation and risk controls.

7.1 Common pipeline (used in both scripts)

Universe, data download, indicators, and signals are identical.

Data download (vectorbt Yahoo wrapper) and Close extraction:

```
prices = vbt.YFData.download(assets,
start="2025-01-01",
end=None).get("Close")
```

Important syntax:

- `vbt.YFData.download(assets, start=..., end=...)` downloads data for multiple tickers and returns a vectorbt data container.
- `.get("Close")` extracts the Close field as a pandas DataFrame aligned across all assets.

Indicators (EMA via `vbt.MA.run(..., ewm=True)`) and extraction:

```
ema7 = vbt.MA.run(prices, window=7, ewm=True).ma
ema21 = vbt.MA.run(prices, window=21, ewm=True).ma
```

Important syntax:

- `vbt.MA.run(prices, window=7, ewm=True)` computes a moving average on every column of `prices`.
- `ewm=True` makes it an exponential moving average (EMA) rather than a simple moving average.
- `.ma` accesses the resulting MA series (vectorbt indicator objects store outputs as attributes).

Signal definition (long/flat):

```
entries = ema7 > ema21
exits = ema7 < ema21
```

Meaning:

- Entry signal becomes True when the fast EMA is above the slow EMA (trend “up”).
- Exit signal becomes True when fast drops below slow (trend “down”).
- vectorbt interprets these as boolean arrays aligned to `prices.index` and each asset column.

7.2 Allocation Policy A: Equal-dollar sizing

Baseline idea: whenever an entry occurs, allocate the same dollar amount per asset.

From the script:

```
init_cash = 100000
target_allocation = init_cash / len(assets)  # $20,000
```

Trade sizing happens through `vbt.Portfolio.from_signals` by passing `size` and `size_type`.

Core backtest call (equal-dollar):

```
pf = vbt.Portfolio.from_signals(
    close=prices,
    entries=entries,
    exits=exits,
    init_cash=init_cash,
    fees=0.00,
    slippage=0.001,
    sl_stop=0.05,
    sl_trail=True,
    cash_sharing=True,
    size=target_allocation,
    size_type="Value",
)
```

Important arguments and syntax:

- `close=prices`: the traded price series (here, Close).
- `entries`, `exits`: boolean DataFrame with same shape as `close`.
- `init_cash=100000`: initial portfolio cash.
- `cash_sharing=True`: all assets draw from one cash pool. If multiple entries happen at once, they compete for cash instead of each asset pretending it has its own “account.”

- `size=target_allocation`: here this is a scalar (20,000), so every entry tries to buy $20k of that asset.
- `size_type="Value"`: interprets `size` in currency units (dollars). If you used `"Percent"` or `"TargetPercent"`, `size` would be interpreted as a fraction of portfolio value instead.

Risk controls in this baseline (still important):

- `sl_stop=0.05`: 5% stop (vectorbt stop model).
- `sl_trail=True`: makes it trailing (stop level follows favorable price movement).
- `slippage=0.001`: 0.10% adverse fill adjustment per trade (simple friction proxy).

What equal-dollar does (conceptually):

- Equal dollars ≠ equal risk.
- A volatile stock will contribute more to portfolio drawdowns than a stable stock, even if both get $20k.

7.3 Allocation Policy B: Volatility-targeted sizing (risk budgeting)

This is the risk management upgrade: size positions so each contributes roughly the same expected daily dollar volatility.

Step 1: estimate volatility from returns (21-day rolling standard deviation).

```
volatility = prices.vbt.pct_change().rolling(21).std()
```

Important syntax:

- `prices.vbt.pct_change()` computes percent returns per asset (like `prices.pct_change()`, but using the vectorbt accessor).
- `.rolling(21).std()` computes rolling 21-bar standard deviation per asset.

Step 2: set a risk budget in dollars per position per day.

```
target_daily_risk_dollars = 200
```

Step 3: convert volatility into a position value.

If daily volatility is 1% (0.01), then a $20,000 position has about $200 daily “one-sigma” move:
$20,000 × 0.01 = $200.

So invert volatility:

```
dynamic_size = target_daily_risk_dollars / volatility
```

Interpretation:

- Higher vol → smaller `dynamic_size`
- Lower vol → larger `dynamic_size`

Step 4: clean sizing series (avoid NaNs/inf) and cap it.

```
dynamic_size = dynamic_size.replace([np.inf, -np.inf], np.nan).fillna(0)
dynamic_size = dynamic_size.clip(upper=30000)
```

Why this is necessary:

- If volatility is 0 or missing, division yields inf/NaN.
- Capping prevents extreme position sizes when measured vol is temporarily very low.

Step 5: run the same `from_signals` backtest, but pass a DataFrame size (time-varying per asset).

```
pf = vbt.Portfolio.from_signals(
    close=prices,
    entries=entries,
    exits=exits,
    init_cash=init_cash,
    fees=0.00,
    slippage=0.001,
    sl_stop=0.05,
    sl_trail=True,
    cash_sharing=True,
    size=dynamic_size,
    size_type="Value",
    freq="1D"
)
```

Important differences vs equal-dollar:

- `size=dynamic_size` is now a DataFrame aligned to time and assets, so the desired position value can change every bar (even if the signal does not change).
- `freq="1D"` sets the portfolio frequency explicitly (used in annualization/metrics).

What volatility targeting changes (conceptually):

- The signal decides direction (in/out).
- Volatility targeting decides exposure magnitude (how big).
- In volatility spikes, positions shrink automatically without needing a separate “risk-off” signal.

7.4 Reading results: stats, allocation breakdown, and benchmarks

Both scripts output:

A) Summary stats:

```
print(pf.stats())
```

For equal-weight portfolio:

```
Start                         2024-12-31 05:00:00+00:00
End                           2026-02-19 05:00:00+00:00
Period                                              284
Start Value                                    100000.0
End Value                                 115516.252427
Total Return [%]                              15.516252
Benchmark Return [%]                          12.862926
Max Gross Exposure [%]                            100.0
Total Fees Paid                                     0.0
Max Drawdown [%]                               5.324452
Max Drawdown Duration                              82.0
Total Trades                                         42
Total Closed Trades                                  42
Total Open Trades                                     0
Open Trade PnL                                      0.0
Win Rate [%]                                  38.095238
Best Trade [%]                                39.163613
Worst Trade [%]                               -9.826617
Avg Winning Trade [%]                         10.576411
Avg Losing Trade [%]                          -3.348094
Avg Winning Trade Duration                      33.0625
Avg Losing Trade Duration                      8.230769
Profit Factor                                  1.896319
Expectancy                                   369.434582
```

For volatility-based allocation:

```
Start                         2024-12-31 05:00:00+00:00
End                           2026-02-19 05:00:00+00:00
Period                                284 days 00:00:00
Start Value                                    100000.0
End Value                                 107296.407734
Total Return [%]                               7.296408
Benchmark Return [%]                          12.862926
Max Gross Exposure [%]                        73.681705
Total Fees Paid                                     0.0
Max Drawdown [%]                               3.229625
Max Drawdown Duration                 101 days 00:00:00
Total Trades                                         41
Total Closed Trades                                  41
Total Open Trades                                     0
```

```
Open Trade PnL                                      0.0
Win Rate [%]                                  36.585366
Best Trade [%]                                39.163613
Worst Trade [%]                               -9.826617
Avg Winning Trade [%]                         11.233468
Avg Losing Trade [%]                          -3.266384
Avg Winning Trade Duration              34 days 09:36:00
Avg Losing Trade Duration     8 days 13:50:46.153846153
Profit Factor                                 1.818451
Expectancy                                  177.961164
Sharpe Ratio                                  1.241534
Calmar Ratio                                  2.933267
Omega Ratio                                   1.231919
Sortino Ratio                                 1.875414
```

B) Equity curve plot (vectorbt):

```
pf.plot(...).show()
```

Equal-weight potfolio:

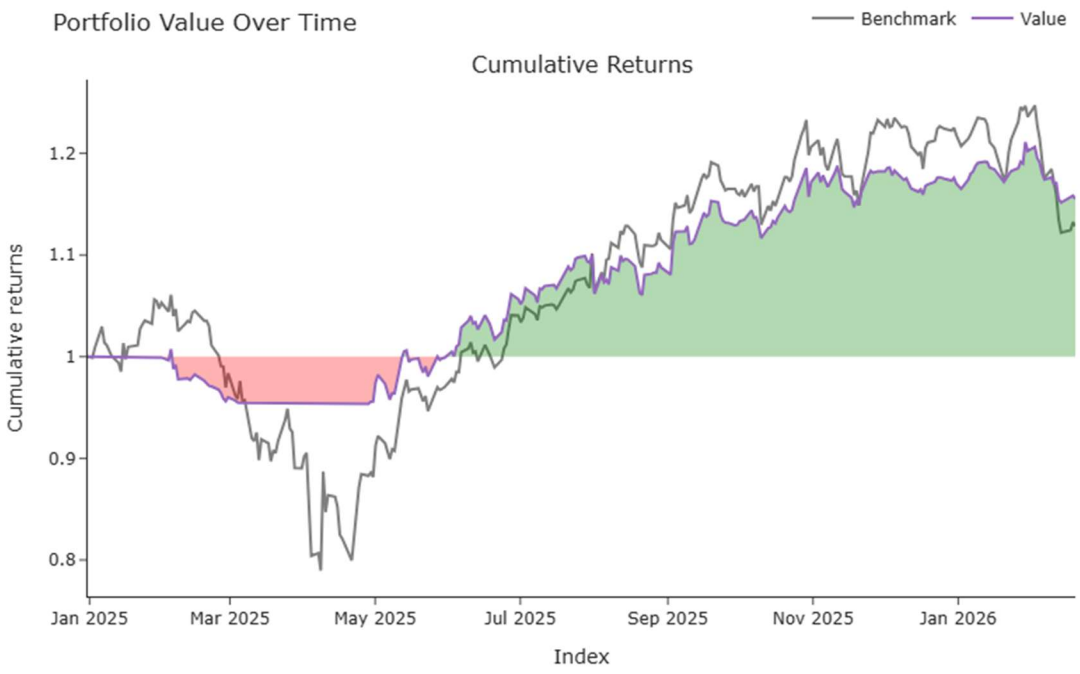

Figure 7.4.1: Equity curve for equal-Dollar portfolio

Volatility-based allocation:

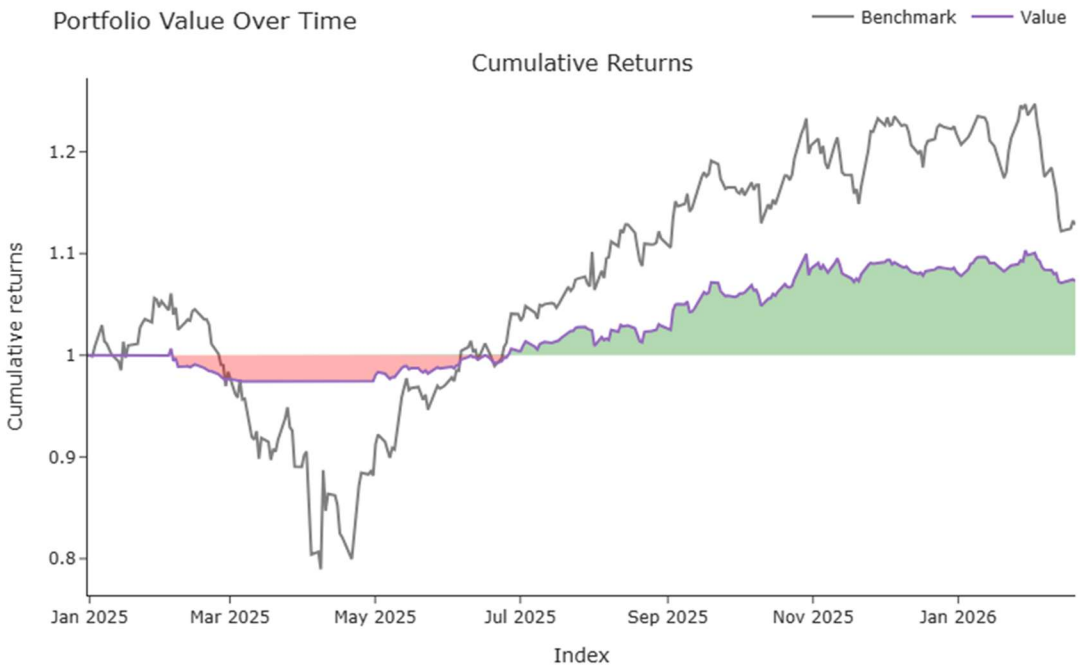

Figure 7.4.2: Equity curve for dynamic-Dollar portfolio

C) Allocation breakdown stackplot (Matplotlib):

```
asset_values = pf.asset_value(group_by=False)
cash_values = pf.cash()

plt.stackplot(
    asset_values.index,
    cash_values,
    *[asset_values[col] for col in asset_values.columns],
    labels=["Cash"] + list(asset_values.columns),
    alpha=0.8
)
```

Equal allocation:

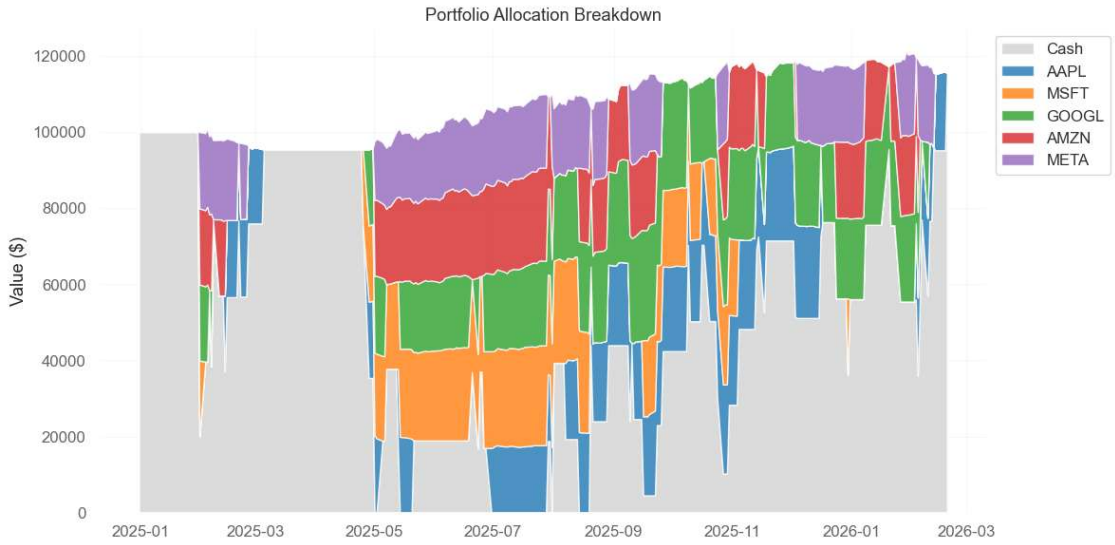

Figure 7.4.3: Allocation for equal-Dollar portfolio

Dynamic allocation:

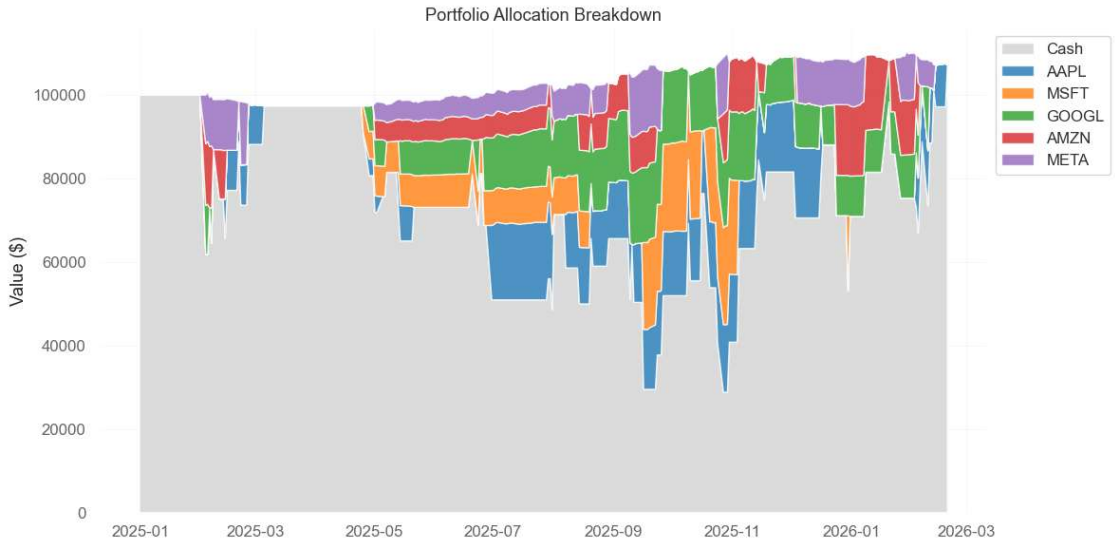

Figure 7.4.4: Allocation for dynamic-Dollar portfolio

Important function syntax:

- `pf.asset_value(group_by=False)` returns a DataFrame of per-asset position value over time (no grouping).
- `pf.cash()` returns the remaining cash series.
- Stacking cash + asset values visualizes concentration and diversification through time.

D) Buy & hold benchmarks vs strategy equity:

```
strategy_equity = pf.value()
asset_benchmarks = prices.vbt.rebase(init_cash)
```

- `pf.value()` is the portfolio equity curve.
- `prices.vbt.rebase(init_cash)` rescales each asset price series so each starts at `init_cash`. This simulates "what if I invested the whole $100k in this single asset" for easy comparison.

7.5 What to compare between the two approaches

Equal-dollar allocation vs volatility targeting typically changes:

1. Concentration: volatility targeting often reduces dominance by the most volatile asset (but not always; signals still matter).
2. Drawdowns: risk budgeting usually smooths equity swings, but stops and correlations still dominate tail events.
3. Turnover: dynamic sizing can increase "micro-adjustments" in exposure (even if entry/exit signals are unchanged).
4. Sensitivity to slippage: more trading impact means robustness to `slippage` matters more.