# Roadmap for a Flexible Trading Strategy Backtesting System

## Introduction and Objectives

We aim to develop an **automated stock trading strategy backtesting system** that can evaluate various trading strategies (investment techniques) for their profitability and robustness. Unlike a typical stock portfolio, here *portfolio* refers to a collection of trading **strategies**, not a set of stocks. The system should allow us to input trading rules (derived from sources like YouTube videos, PDFs, etc.) in a **modular, declarative JSON format**, backtest these rules on historical data, and determine if they yield consistent profits. This will help filter out strategies that performed poorly in the past (since a strategy that fails on historical data is even less likely to succeed in the future, acknowledging that past performance doesn’t guarantee future results[[1]](https://kernc.github.io/backtesting.py/#:~:text=Backtesting,including%20a%20handful%20of%20tutorials)). Key objectives include:

* Designing a **flexible JSON schema** to define strategies in a multi-layered, declarative manner (including indicators and entry/exit rules).
* Building a **backtesting engine** that reads a strategy JSON, applies it to historical stock data, and simulates trades.
* Testing each strategy across multiple stocks (e.g., ~30 KOSPI and ~10 KOSDAQ stocks) over ~5 years of daily data to evaluate *universal profitability*.
* Producing detailed **results per stock** (trade log, equity curve, etc.) and a **comprehensive summary** of overall performance across all tested stocks.
* Computing standard **performance metrics**: profitability (total and annualized returns), stability (volatility, drawdowns), win rate, profit/loss ratio, Sharpe ratio, etc., for each strategy[[2]](https://kernc.github.io/backtesting.py/)[[3]](https://kernc.github.io/backtesting.py/#:~:text=Win%20Rate%20%5B,69468).
* Ensuring **accuracy** of results (potentially by cross-verification with a known library like *Backtesting.py*[[1]](https://kernc.github.io/backtesting.py/#:~:text=Backtesting,including%20a%20handful%20of%20tutorials)) and handling various strategy time horizons (some strategies may hold positions for days vs. months).
* Creating **intuitive, interactive charts** for visualization, so users can easily interpret how the strategy performed over time (e.g., equity curve with buy/sell points).
* Assuming *no advanced money management*: to isolate stock-picking performance, each buy signal will use 100% of available capital on that stock (no leverage, no simultaneous multi-stock positions). We will evaluate this assumption and discuss alternatives in the capital allocation section.

The end result will be a clear development roadmap and some illustrative code snippets. This will guide an AI or developer in implementing the system step-by-step in Python.

## System Architecture Overview

The backtesting system will consist of several **modular components** working together:

* **Strategy Definition (JSON file)**: Defines the trading strategy in a structured way (indicators, entry/exit rules, etc.). This JSON is *declarative* – it describes *what* the strategy rules are, without code. (See **Strategy JSON Schema** below for details.)
* **Data Provider**: Fetches historical OHLCV data for the specified stocks. We plan to use the Yahoo Finance API via the yfinance Python library for convenience, since it covers global stocks including KOSPI/KOSDAQ and can provide ~5 years of daily data. (We will ensure the chosen stocks have complete data to avoid missing values. Alternatives or backups can be discussed in **Data Acquisition**.)
* **Backtesting Engine**: The core module that takes the strategy JSON and historical data as input, and simulates the strategy’s trades bar by bar. This includes an **Indicator Calculator** (to compute technical indicator series as defined in JSON) and a **Signal Evaluator** (to check entry/exit conditions each day). The engine will simulate orders (buy/sell) and track portfolio state (position, cash, equity) over time.
* **Performance Analyzer**: After simulation, this component calculates performance metrics (returns, Sharpe, drawdown, etc.) from the trade results or equity curve.
* **Results Aggregator**: If multiple stocks are tested, this component collates results from each and produces a summary (e.g., how many stocks were profitable, average Sharpe, yearly performance breakdown, etc.).
* **Visualization Module**: Generates interactive charts to help interpret results, such as equity curves with buy/sell markers for each stock’s backtest, and maybe comparison of performance across stocks. We can use libraries like **Plotly** or **Bokeh** (note: Backtesting.py uses Bokeh to provide interactive charts[[4]](https://kernc.github.io/backtesting.py/#:~:text=)) for this purpose.

The system will be designed for a local Python environment, ensuring all components (data retrieval, calculation, plotting) run in that context. Next, we detail each component and the development steps, including code sketches.

## Data Acquisition (Historical Price Data)

**Step 1: Retrieve historical data** for all stocks of interest. We plan to use the yfinance library to fetch daily OHLCV (Open-High-Low-Close-Volume) data. This library is a convenient wrapper for Yahoo Finance data, which is free and covers many markets. For example, KOSPI stocks can be accessed with ticker symbols like "005930.KS" (Samsung Electronics on KOSPI), and KOSDAQ with a different suffix (Yahoo uses .KS and .KQ for Korean exchanges). We will gather about 5 years of daily data (e.g., 2018-01-01 to 2023-12-31).

Using yfinance addresses the need for future expansion to foreign stocks — it can fetch U.S. or other international stock data similarly. (Yahoo Finance does occasionally have missing data issues[[5]](https://github.com/ranaroussi/yfinance/issues/525#:~:text=Missing%20Data%20%C2%B7%20Issue%20,Monday%2C%2010%2F29%2F2012%20and%20Tuesday%2C%2010%2F30%2F2012), but we will mitigate this by selecting stocks with complete data or cleaning the dataset as needed. In backtesting, using a reliable data source is crucial; alternatives include **Quandl**, **Alpha Vantage**, or official exchange APIs[[6]](https://algotrading101.com/learn/backtesting-py-guide/#:~:text=Keep%20in%20mind%20that%20Backtesting,Finance%20%20or%20%2033) if higher data integrity is required.)

**Implementation:** We will create a list of target ticker symbols (e.g., 30 KOSPI tickers + 10 KOSDAQ tickers chosen, possibly filtered by criteria like market cap and liquidity as in the strategy’s universe filters). Then use yfinance.download or Ticker.history to fetch data. For example:

import yfinance as yf  
  
tickers = ["000660.KS", "005930.KS", "..."] # KOSPI & KOSDAQ tickers  
data = yf.download(tickers, start="2018-01-01", end="2023-12-31")

This returns a multi-index DataFrame (ticker and date indexed) with OHLCV columns. We can also fetch each ticker in a loop if easier. We will likely store each stock’s DataFrame separately for simplicity during backtesting.

Before proceeding, the **universe filters** defined in the strategy JSON (e.g., minimum market cap or trading value) should be applied to decide which stocks from our list to actually test. If these filters require additional data (like market cap), we might retrieve those via yfinance.Ticker.info or an external source. For instance, a filter like {"filter\_type": "market\_cap", "min\_market\_cap": 1e12} means we exclude stocks with market cap below 1 trillion KRW. Since data for such filters might not be directly in price history, we can handle it by pre-defining our stock list accordingly or by checking attributes via the API. Once the universe is finalized, we proceed with the backtest on each.

## Strategy JSON Schema and Parsing

**Step 2: Define and parse the strategy JSON.** The strategy JSON file provides a flexible, declarative way to specify trading rules. It is designed to be hierarchical and modular, allowing a wide variety of strategies to be described without hardcoding logic. Key sections of the JSON schema include:

* **Metadata**: e.g. strategy\_name, strategy\_version, description, etc., which document the strategy.
* **Universe Filters**: Criteria for selecting which stocks or instruments the strategy applies to (e.g., min market cap, liquidity). For example, the JSON might have:
* "universe\_filters": [  
   {"filter\_type": "market\_cap", "min\_market\_cap": 1000000000000},  
   {"filter\_type": "trading\_value", "min\_value": 10000000000, "days": 20}  
  ]
* This would mean we only consider stocks with market cap ≥ 1e12 and with average trading value ≥ 1e10 (over last 20 days)[[7]](file://file-3AMt34TMiFstS8URbAHW4Q#:~:text=,20). Our code will interpret these and filter the initial ticker list accordingly (as noted in Data Acquisition).
* **Indicators**: A list of technical indicators to compute. Each indicator entry has an id (name for reference in rules), a type (the kind of indicator), and params (such as period length and price field). For example:
* "indicators": [  
   {"id": "SMA\_5\_Close", "type": "SMA", "params": {"period": 5, "source": "Close"}},  
   {"id": "SMA\_20\_Close", "type": "SMA", "params": {"period": 20, "source": "Close"}},  
   {"id": "MIN\_60\_Low", "type": "MIN", "params": {"period": 60, "source": "Low"}}  
  ]
* This defines a 5-day moving average of Close price, a 20-day moving average of Close, and a 60-day rolling minimum of Low price[[8][9]](file://file-3AMt34TMiFstS8URbAHW4Q#:~:text=%7B%20,%7D). The backtester will dynamically calculate these series from the price data before evaluating any rules. We will design the code to support common indicator types (SMA, EMA, MIN, MAX, RSI, etc.), and it's easy to extend by adding new types in the future. The calculated series will be stored in a dictionary (e.g., indicators\_data[id] = pd.Series of values).
* **Entry Rules**: The conditions that trigger a **buy/entry** signal. The JSON structure for rules typically contains a logical combination ("logic": "AND" or "OR") and a list of condition objects. Each **condition** may have: an operator (op), a left operand (left), and a right operand (right). Operands can be price fields (Open, Close, High, Low, Volume at the current bar) or an indicator ID referencing one of the computed indicator series. For example, an entry rule might be:
* "entry\_rules": {  
   "logic": "AND",  
   "conditions": [  
   {"op": ">", "left": "Close", "right": "MIN\_60\_Low"},  
   {"op": "crosses\_above", "left": "SMA\_5\_Close", "right": "SMA\_20\_Close"}  
   ]  
  }
* This represents an **AND** of two conditions: (1) current Close price is greater than the 60-day Low, and (2) the 5-day SMA has just crossed above the 20-day SMA (a golden cross type signal)[[10]](file://file-3AMt34TMiFstS8URbAHW4Q#:~:text=%7B%20,SMA_20_Close). Both must be true on the same day to trigger a buy. The operator "crosses\_above" implies a *crossover event* happening at this bar (we will implement this by checking previous bar values: SMA\_5 was below SMA\_20 yesterday, and is above or equal today). Other possible ops might include "<" (less than), "==" (equal), "crosses\_below" (for moving average death cross, etc.), or even pattern-based conditions. Our engine will support at least basic comparison ops and crossovers initially. If needed, the JSON could allow nested condition groups for more complex logic (multi-layer logic), but many strategies can be expressed with a single layer using AND/OR combinations as above.
* **Exit Rules**: Conditions for **sell/exit** signals, structured similarly to entry. Typically exit rules might use an "OR" logic (sell if any of the conditions trigger). For example, the strategy might exit when a fast SMA crosses below a slow SMA or when a certain profit target or stop-loss is hit. In the given example JSON, the exit rule was:
* "exit\_rules": {  
   "logic": "OR",  
   "conditions": [  
   {"op": "crosses\_below", "left": "SMA\_5\_Close", "right": "SMA\_20\_Close"}  
   ]  
  }
* Meaning sell when the 5-day SMA crosses below the 20-day SMA[[11]](file://file-3AMt34TMiFstS8URbAHW4Q#:~:text=,%7D). We can imagine extending this to include other conditions (e.g., OR with a stop-loss condition). The backtester will check exit conditions for an open position on each bar.

**Parsing the JSON:** We will use Python’s json library to load the JSON file into a dictionary. For example:

import json  
  
with open("Double\_Bottom\_Recovery\_v2.json") as f:  
 strategy = json.load(f)  
# Access parts:  
entry\_logic = strategy["entry\_rules"]["logic"] # e.g., "AND"  
entry\_conditions = strategy["entry\_rules"]["conditions"]

The system will then iterate through the conditions list and interpret each condition. We might set up a helper function or mapping for evaluating conditions given the current data context (more in the backtest engine section). The modular JSON approach makes it easy to add new strategies—just create a new JSON file with the appropriate structure, and the backtester can run it without changing code.

## Indicator Computation Module

Before the simulation, we need to compute all the **technical indicators** listed in the strategy JSON for the given stock’s price data. This is essentially a preparation step within the backtesting engine for each stock.

**Step 3: Compute indicator series.** For each indicator in strategy["indicators"], the engine will calculate the series using the stock’s OHLCV DataFrame. We will likely use **pandas** for rolling calculations, or possibly a library like **pandas\_ta** for more complex indicators if needed. Initially, simple indicators (moving averages, rolling highs/lows, etc.) can be done with pandas built-in rolling functions.

For example, given a DataFrame df with columns Open, High, Low, Close, Volume indexed by date:

indicators\_data = {} # dictionary to store computed indicator series  
for ind in strategy["indicators"]:  
 ind\_id = ind["id"] # e.g., "SMA\_5\_Close"  
 ind\_type = ind["type"] # e.g., "SMA"  
 params = ind["params"] # e.g., {"period": 5, "source": "Close"}  
 if ind\_type == "SMA":  
 period = params["period"]  
 src = params["source"]  
 indicators\_data[ind\_id] = df[src].rolling(window=period).mean()  
 elif ind\_type == "EMA":  
 period = params["period"]  
 src = params["source"]  
 indicators\_data[ind\_id] = df[src].ewm(span=period).mean()  
 elif ind\_type == "MIN":  
 period = params["period"]  
 src = params["source"]  
 indicators\_data[ind\_id] = df[src].rolling(window=period).min()  
 elif ind\_type == "MAX":  
 period = params["period"]  
 src = params["source"]  
 indicators\_data[ind\_id] = df[src].rolling(window=period).max()  
 # ... (additional indicator types as needed)

Each indicators\_data[ind\_id] will be a pandas Series aligned with the DataFrame’s index (dates). We will likely drop the initial NaN values (where not enough data to compute the indicator period) or be careful to start the simulation after the longest lookback period to avoid invalid signals at the very start.

This design ensures **flexibility**: adding a new indicator type (say RSI or MACD) would involve computing it here and then it can be used in conditions. If needed, we could integrate a library like ta (pandas-ta) for a rich set of indicators instead of manually coding each, but manually coding basic ones as above is straightforward and avoids external dependencies.

## Backtesting Engine: Trade Simulation Logic

Now comes the core: **simulating trades** based on the strategy rules and historical data.

**Step 4: Implement trade simulation** for one stock and one strategy. We will iterate through each day’s data (each row of the DataFrame) and evaluate entry/exit conditions. Key aspects of the simulation design:

* We assume **long-only trading** (buy low, sell high) unless strategy specifies shorting (the provided example strategy is long-only; shorting could be added later but isn't a focus now).
* Only one position at a time per stock, using **full capital** on entry. This means if we buy a stock, we hold it until an exit signal, then potentially move to another trade (or re-enter later). There are no concurrent positions on the same stock (and since we handle each stock separately for now, no concurrent multi-stock positions either in a single test).
* **Signal evaluation timing**: We will use **daily close data to generate signals**, and assume trades execute at the **next day’s open price** (to avoid lookahead bias). For example, if on Day T the entry conditions become true at close, we simulate buying at the opening of Day T+1. Similarly, exit at next day’s open after an exit condition. (If desired, the system could allow same-day execution at close, but by then the information is fully known, which can overestimate performance. Using next open is more conservative and realistic for daily strategies.)
* **Position management**: We maintain a state of whether we are currently in a trade (position held) or not, along with variables like entry\_price, shares\_held, and cash. With 100% capital usage, this simplifies to either fully invested in X shares or fully in cash. We start with an initial capital (say 1,000,000 KRW or any amount; since results can be reported in percentages, the initial capital value is somewhat arbitrary).

The simulation will proceed as follows (pseudo-code outline):

initial\_capital = 1000000.0 # example starting cash  
cash = initial\_capital  
shares = 0  
entry\_price = 0.0  
position\_open = False  
  
equity\_curve = [] # to record equity value each day for metrics  
  
for i, current\_day in enumerate(df.index):  
 # Access current day's price and possibly next day's price for trade execution  
 close\_price = df.loc[current\_day, "Close"]  
 if position\_open:  
 # Update equity value while holding (mark-to-market)  
 equity = shares \* close\_price  
 else:  
 equity = cash  
 equity\_curve.append(equity)  
  
 # At end of day, evaluate signals (for next day action)  
 entry\_ok = False  
 exit\_ok = False  
 # Evaluate entry conditions if no position  
 if not position\_open:  
 # Check entry rule logic (e.g., all conditions true if logic == "AND")  
 if evaluate\_conditions(strategy["entry\_rules"], i):  
 entry\_ok = True  
 else:  
 # If a position is open, check exit rule  
 if evaluate\_conditions(strategy["exit\_rules"], i):  
 exit\_ok = True  
  
 # Execute orders at next day open (if signal is true and next day exists)  
 if entry\_ok and not position\_open:  
 if i < len(df.index) - 1: # ensure next day exists  
 next\_day = df.index[i+1]  
 buy\_price = df.loc[next\_day, "Open"]  
 # Calculate shares to buy with all cash:  
 shares = cash / buy\_price # (we can allow fractional shares for simplicity)  
 cash = 0  
 position\_open = True  
 entry\_price = buy\_price  
 # record trade entry (date, price, shares)  
 elif exit\_ok and position\_open:  
 if i < len(df.index) - 1:  
 next\_day = df.index[i+1]  
 sell\_price = df.loc[next\_day, "Open"]  
 # Sell all shares:  
 cash = shares \* sell\_price  
 shares = 0  
 position\_open = False  
 # record trade exit (date, price, P&L)

A few notes on this logic:  
- The function evaluate\_conditions(rules, i) will encapsulate checking the rule conditions on the *current* bar index i. If rules["logic"] is "AND", all conditions must be true; if "OR", at least one true. Each condition check will fetch the **left** and **right** operands values for index i and apply the operator. For example, for {"op": ">", "left": "Close", "right": "MIN\_60\_Low"}, it will compare df["Close"][i] to indicators\_data["MIN\_60\_Low"][i]. For crossover events like crosses\_above, it will check indices i-1 and i: e.g.,

left\_today = get\_value(condition["left"], i)  
right\_today = get\_value(condition["right"], i)  
left\_prev = get\_value(condition["left"], i-1)  
right\_prev = get\_value(condition["right"], i-1)  
condition\_true = (left\_prev < right\_prev) and (left\_today >= right\_today)

(with appropriate boundary-check for i-1). get\_value(name, i) would return either df[name][i] if name is a price column or indicators\_data[name][i] if it’s an indicator. This design cleanly separates indicator computation (done beforehand) from rule evaluation.  
- We record trades when they happen: on entry, note the date, price, and shares; on exit, note date, price, and profit or loss. These can be stored in a list of trade records for later analysis or output. Each record might contain {entry\_date, entry\_price, exit\_date, exit\_price, profit\_pct, holding\_period} etc.

By the end of the loop, we will have: a complete equity\_curve time series for that stock (which reflects account equity day by day, either flat when in cash or fluctuating with price when in a position), and a list of completed trades with outcomes.

This custom engine ensures we have **full control and transparency** over the backtesting process. It also easily accommodates the JSON-defined rules. We should double-check the logic by testing simple strategies on a small dataset. For instance, we might verify that a basic SMA crossover strategy yields the same trade points as known results or as the Backtesting.py library for the same data, to ensure our implementation is correct.

## Performance Metrics Calculation

**Step 5: Calculate performance metrics** for the strategy on a given stock’s backtest results. Once we have the equity curve and trade list, we can compute various evaluation metrics that measure profitability and risk. Important metrics to compute include:

* **Total Return (%)**: The percentage increase of the account equity over the test period. This is (final equity / initial equity - 1) \* 100. For example, ending with \$1.5 million from \$1.0 million initial means +50% total return. We can also compare this to a buy-and-hold return for the stock (or an index) as a benchmark.
* **Annualized Return (CAGR)**: The compounded annual growth rate. If is total return and the test period is years, CAGR = . This normalizes performance to per-year growth.
* **Win Rate (%)**: Number of profitable trades divided by total number of trades, \*100. If out of 10 trades 6 were winners, win rate = 60%. This indicates consistency of winning.
* **Average Profit vs Loss (Payoff Ratio)**: Also called *profit/loss ratio* or *average win to average loss*. We compute the average gain on winning trades and average loss on losing trades. The ratio = (mean win %)/(mean loss %). For example, if average win = +10% and average loss = -5%, payoff ratio = 2.0. A ratio > 1 indicates wins are larger than losses on average. We should also report the **Profit Factor**, which is the sum of all profits divided by sum of all losses (a similar concept on a cumulative basis). These measures show how much the strategy gains relative to losses[[3]](https://kernc.github.io/backtesting.py/#:~:text=Win%20Rate%20%5B,69468).
* **Sharpe Ratio**: A standard risk-adjusted return metric. It is typically (mean excess return divided by standard deviation of return). Assuming a risk-free rate ~0 for simplicity, we can take daily returns of the equity curve (percentage change each day) and compute Sharpe = (mean daily return / std dev of daily return) \* √252 (to annualize, since ~252 trading days in a year). A higher Sharpe indicates better risk-adjusted performance. We will ensure to use the entire equity curve (which includes flat periods as zero returns) for this calculation so that it accounts for volatility of the strategy’s equity.
* **Max Drawdown (%)**: The worst peak-to-valley drop in equity during the period. We can compute this by scanning the equity curve for the largest percentage decline from any historical peak. This indicates risk: e.g., -20% max drawdown means at one point the strategy had a 20% loss from its highest equity point. We can also compute **Calmar Ratio = CAGR / |Max Drawdown|** as another stability metric (higher is better, implying high return for low drawdown).
* **Volatility**: Annualized standard deviation of daily returns (which we actually get in the Sharpe calculation denominator). We might report daily volatility or annualized volatility as a measure of stability of returns. Lower volatility and drawdown means more stable performance.
* **Other metrics**: Depending on needs, we can add Sortino Ratio (like Sharpe but considers only downside volatility), **Expectancy** (average % return per trade = win\_rate \* avg\_win% + loss\_rate \* avg\_loss%), **Average trade duration** (how many days a typical trade lasts), etc. For a comprehensive view, frameworks like Backtesting.py output many of these[[12][13]](https://kernc.github.io/backtesting.py/). We will focus on the ones mentioned (profitability, stability, win rate, payoff, Sharpe) and can extend if needed.

**Implementation:** Once the trade list and equity curve are available, most metrics are straightforward to calculate with pandas/numpy. For example:

import numpy as np  
  
# Assuming equity\_curve is a pandas Series of daily equity values:  
returns = equity\_curve.pct\_change().dropna() # daily returns  
sharpe = (returns.mean() / returns.std()) \* np.sqrt(252)  
  
# Trades data:  
profits = [trade['profit\_pct'] for trade in trades] # percentage profit per trade  
wins = [p for p in profits if p > 0]  
losses = [p for p in profits if p <= 0]  
win\_rate = len(wins) / len(profits) \* 100 if profits else 0  
avg\_win = np.mean(wins) if wins else 0  
avg\_loss = np.mean(losses) if losses else 0  
payoff\_ratio = abs(avg\_win / avg\_loss) if losses else None # define carefully  
profit\_factor = sum(p for p in profits if p>0) / abs(sum(p for p in profits if p<0)) if losses else None

And similarly, track equity peaks to calculate drawdown:

equity\_array = np.array(equity\_curve)  
cum\_max = np.maximum.accumulate(equity\_array)  
drawdowns = (equity\_array - cum\_max) / cum\_max  
max\_drawdown = drawdowns.min() \* 100 # in percentage (will be negative)

We will gather all these metrics into a results structure (like a dict or DataFrame row for that stock). Also, for **periodic performance**, we can break down the equity curve by year: e.g., compute yearly returns (final Dec vs Jan of that year), or even do a year-by-year Sharpe. This will show how the strategy performed in different market conditions (bull year vs bear year, etc.). For instance, we might produce a table of annual returns for each year in the test range for that stock. Later, we can aggregate such data across stocks.

## Backtesting Across Multiple Stocks

**Step 6: Batch testing and aggregation.** With the ability to backtest one stock, we will automate testing across the list of selected stocks. This involves looping over each stock’s data, running the simulation, and collecting results. Pseudocode:

all\_results = [] # to store metrics for each stock  
for ticker, df in stock\_data\_map.items(): # iterate over each stock's DataFrame  
 # (Apply universe filters if needed to skip some)  
 # Compute indicators:  
 indicators\_data = compute\_indicators(df, strategy["indicators"])  
 # Run backtest simulation:  
 trades, equity\_curve = run\_backtest(df, indicators\_data, strategy["entry\_rules"], strategy["exit\_rules"])  
 # Calculate metrics:  
 metrics = analyze\_performance(trades, equity\_curve)  
 metrics["ticker"] = ticker  
 all\_results.append(metrics)

After this loop, all\_results will contain performance metrics for each stock. We can then produce a **summary report**. The *comprehensive result* could include:

* A table listing each stock and key metrics (Total Return, CAGR, Sharpe, Win%, Max Drawdown, etc.). This allows identifying on which stocks the strategy performed well or poorly.
* An **aggregate view**: e.g., the average and median of each metric across all stocks, the number (or percentage) of stocks that were profitable (total return > 0) vs. losing. This tells us if the strategy is *universally* effective or only works on certain stocks. If, say, 35 out of 40 stocks showed a positive return and the majority have Sharpe > 1, that indicates a robust strategy. On the other hand, if only a few stocks drove the profits and many were negative, the strategy may not be universally reliable.
* **Distribution of outcomes**: We could calculate the distribution of total returns or Sharpe ratios across the stocks. For example, maybe the worst stock had -20% over 5 years, the best +80%, and median +10%. Presenting this range is useful.
* **Period-by-period performance**: We might aggregate how the strategy did in each year across all stocks. For instance, calculate the average return of the strategy in 2018 for all stocks, in 2019, etc. This could be done by taking each stock’s equity curve and seeing what % it gained in that calendar year, then averaging. This analysis shows if the strategy struggled in specific years (perhaps market-wide issues like a crash year) or was consistently profitable every year. Another way is to simulate a *combined equity curve* as if one rotated through stocks (though interpreting that without simultaneous positions must be done carefully; see below).

**Combined vs. Separate Backtests:** We must clarify that because we are isolating stock selection, we treated each stock independently with full capital. We did not simulate an actual portfolio holding multiple stocks at once. Therefore, there isn’t a single unified equity curve for all stocks directly. However, we can construct a hypothetical **combined equity curve** by sequentially chaining trades from all stocks: e.g., start with capital, and whenever a stock trade finishes, immediately move that capital to the next trade (perhaps the next chronological signal among all stocks). This requires merging all trade signals by date and ensures only one trade at a time. This approach can show what if an investor followed this strategy across the market, always entering whichever stock gave a signal next. It would yield an overall equity curve and performance metrics. The drawback is if multiple stocks give signals around the same time, one would have to choose one (or divide capital, which we’re avoiding here). For our scope, it might be sufficient to **report the distribution of single-stock results** rather than one hypothetical combined result. In the report, we will clearly indicate that money management and simultaneous trades were not considered – each stock was an independent test. The summary metrics (like average return) are therefore more statistical than representing a real portfolio outcome.

Nonetheless, if desired, we can attempt an approximate combined performance: for example, *equal-weighted* average of all equity curves (treating as if we put 1/40th of capital in each stock’s strategy). That effectively assumes running all stocks in parallel with equal capital split. However, that reintroduces capital allocation issues. Given the user’s focus, we will likely stick to summarizing individual results and highlighting how many stocks passed certain thresholds.

## Interactive Visualization of Results

**Step 7: Visualization.** Interpreting backtest results is much easier with charts and interactive elements. We will implement interactive charts for both individual stock performance and the aggregate results:

* **Per-Stock Equity Curve**: For each stock’s backtest, we can plot the equity curve over time. On the same chart, we may overlay the stock’s price to see how the strategy navigated it, and mark buy/sell points. Interactive charts (using Plotly or Bokeh) allow zooming into specific periods, which is helpful to examine trades in detail. For example, using Plotly in Python:

import plotly.graph\_objects as go  
  
fig = go.Figure()  
# Add price line  
fig.add\_trace(go.Scatter(x=df.index, y=df['Close'], name=f"{ticker} Price", line=dict(color='gray')))  
# Add equity curve line  
fig.add\_trace(go.Scatter(x=df.index, y=equity\_curve, name=f"Equity Curve", line=dict(color='blue')))  
# Add buy/sell markers  
for trade in trades:  
 buy\_date = trade['entry\_date']; sell\_date = trade['exit\_date']  
 fig.add\_trace(go.Scatter(x=[buy\_date], y=[trade['entry\_price']], mode='markers', marker\_symbol='triangle-up',   
 marker\_color='green', marker\_size=10, name='Buy'))  
 fig.add\_trace(go.Scatter(x=[sell\_date], y=[trade['exit\_price']], mode='markers', marker\_symbol='triangle-down',   
 marker\_color='red', marker\_size=10, name='Sell'))  
fig.update\_layout(title=f"{strategy\_name} on {ticker}", yaxis\_title="Price / Equity", xaxis\_title="Date")  
fig.show()

This would generate an interactive chart in a Jupyter notebook or web context. The user can hover to see values and zoom into regions. If using Backtesting.py’s built-in bt.plot(), it produces a similar interactive chart automatically[[4]](https://kernc.github.io/backtesting.py/#:~:text=)[[14]](https://algotrading101.com/learn/backtesting-py-guide/#:~:text=,Is%20actively%20maintained) – however, customizing that or combining multiple series might be harder, so using Plotly directly provides more flexibility for custom overlays and multi-stock support.

* **Aggregate Performance Charts**: We can visualize summary statistics across stocks. For example, a bar chart of each stock’s total return or Sharpe ratio, to quickly see which stocks did well vs poorly. Or a boxplot of the distribution of returns. An interactive bar chart with hover text showing detailed metrics per stock could be useful. Another idea is a heatmap where one axis is stock and the other is year, colored by that year’s return for the strategy on that stock – this would highlight if certain stocks or years were outliers.
* **User Interaction**: We might incorporate dropdowns or sliders for interactivity (e.g., select which stock’s chart to view from a dropdown). Plotly supports adding dropdown menus to toggle data series, which could allow a single chart to display different stocks' equity curves one at a time. This prevents having to generate 40 separate static charts. Given this is a local environment, an interactive Jupyter Notebook might be the interface, which is acceptable.

Our focus is to ensure results are **intuitively interpretable**: for instance, seeing an equity curve steadily rising indicates a good strategy, while a volatile or downward curve indicates issues. The buy/sell markers let the user verify that the strategy logic makes sense (e.g., did it buy at reasonable points). Similarly, the distribution charts for all stocks will show if performance is consistent or skewed by a few outliers.

## Capital Allocation Considerations (100% per Trade Assumption)

We specifically assume that whenever the strategy signals a buy, we allocate **100% of available capital** to that one trade. This simplifies the backtest and isolates the strategy’s entry/exit logic effectiveness. We need to evaluate this decision:

* **Rationale**: By investing fully in each signal, we are essentially testing the strategy in its purest form on each stock. We’re not letting cash sit idle when a signal is present, and we’re not diluting the impact of a trade by only partially investing. This is a common approach in single-strategy backtests. For example, the Backtesting.py library’s default behavior in its SMA crossover example is to buy as many shares as possible when the signal triggers[[15]](https://kernc.github.io/backtesting.py/#:~:text=Whenever%20the%20fast%2C%2010,CFD%20and%20can%20be%20shorted)[[16]](https://kernc.github.io/backtesting.py/#:~:text=def%20next%28self%29%3A%20if%20crossover%28self,sell), i.e., essentially using all capital for the position. This maximizes the exposure to the strategy’s edge. Moreover, since we are evaluating many stocks independently, using full capital per stock makes the results comparable (each stock’s returns assume you put a full allocation into that stock when trading it).
* **No Money Management/Diversification**: We explicitly avoid simulating simultaneous trades in multiple stocks. In reality, a trader might split capital among multiple positions to diversify or control risk, but that introduces additional factors (how to allocate, correlation between stocks, etc.). Our goal is to see if the strategy logic itself is sound (i.e., does it pick good entry/exit points that yield profit on most stocks?). By removing capital allocation complexity, we can attribute performance differences to the strategy’s stock picking/timing ability alone.
* **Overlapping Signals**: One consequence of this approach is that if two stocks generate a buy signal on the same day, our independent tests would count both (each with full capital in their own test). In a real single-portfolio scenario, you couldn’t take both with 100% each. But since we treat each stock separately, it’s fine. If we were to construct a single combined equity (like sequential trading as discussed earlier), we would need a rule to choose or skip overlapping signals. For now, that isn’t necessary; we interpret the results statistically across stocks rather than as a literal multi-position portfolio.
* **Alternative Approaches**: If we wanted to incorporate a bit more realism or consider another angle, one alternative is to allocate a fixed fraction of capital per trade (say 50% or equal split among concurrently triggered signals). However, that effectively mixes money management into the test. Another alternative is the sequential simulation across stocks (rotational portfolio) described earlier, which uses 100% capital but only one trade at a time across all stocks – but then the strategy’s performance might depend on the order of signals and could underrepresent its potential (since in reality one could potentially deploy capital in multiple uncorrelated opportunities). Given our focus, the chosen approach (full capital per trade per stock, analyzed separately) is sound. It stresses each stock’s strategy performance individually and checks if the strategy holds up generally. In the **future**, once we identify a robust strategy, we could explore portfolio-level tests (with position sizing, risk management, etc.) as a next phase. At that point, frameworks that support multi-asset backtesting (like Backtrader or vectorbt) might be worth considering, or we would extend our engine to handle multiple simultaneous positions with an allocation scheme.
* **Conclusion on 100% allocation**: We agree with the assumption as the *right approach for testing purposes*. It ensures maximum clarity: a trade either succeeds or fails significantly without being masked by partial allocations. It also simplifies implementation. We just must be careful to interpret the aggregate results appropriately, as mentioned. For initial strategy vetting, this approach is appropriate and commonly used.

## Verification and Alternatives

To guarantee the **accuracy of the backtesting results**, we can incorporate a verification step using a known library. As suggested, **Backtesting.py** is a good choice for cross-checking because it’s lightweight and user-friendly[[17]](https://kernc.github.io/backtesting.py/#:~:text=in%20the%20future,including%20a%20handful%20of%20tutorials)[[18]](https://algotrading101.com/learn/backtesting-py-guide/#:~:text=,Is%20actively%20maintained). We can take one of our strategies (or a simplified version of it) and implement it with Backtesting.py’s Strategy class, then compare key results: number of trades, final equity, etc., to our engine’s output. If they match, we gain confidence that our logic is correct. Backtesting.py also provides interactive charts and many metrics out-of-the-box, which can serve as a reference[[2]](https://kernc.github.io/backtesting.py/). One limitation noted is that Backtesting.py doesn’t support multi-asset strategies concurrently[[19]](https://algotrading101.com/learn/backtesting-py-guide/#:~:text=Why%20shouldn%E2%80%99t%20I%20use%20Backtesting), so we would verify one stock at a time (which is exactly how we’re running our tests anyway).

**Example verification:** For the SMA crossover strategy, we could do:

from backtesting import Backtest, Strategy  
from backtesting.lib import crossover  
  
class SmaCross(Strategy):  
 def init(self):  
 price = self.data.Close  
 self.sma5 = self.I(SMA, price, 5)  
 self.sma20 = self.I(SMA, price, 20)  
 self.min60 = self.I(lambda x: x.rolling(60).min(), self.data.Low) # 60-day low  
  
 def next(self):  
 if (self.data.Close[-1] > self.min60[-1]) and crossover(self.sma5, self.sma20):  
 self.buy()  
 elif crossover(self.sma20, self.sma5):  
 self.position.close()  
  
bt = Backtest(stock\_dataframe, SmaCross, cash=initial\_capital, commission=0.0)  
result = bt.run()

This uses the same logic as our JSON example: buy if Close > 60-day Low and SMA5 crosses above SMA20; sell when SMA5 crosses below SMA20. We would compare result (which contains metrics) and the trade log with our own output for that stock. This kind of testing can be done for a couple of sample stocks.

If any discrepancies arise, we’d debug our engine (for example, ensure we handle the crossover exactly as Backtesting.py’s crossover does, and confirm if they buy at next bar or same bar – by default Backtesting.py executes signals on the same bar’s close for entry if I'm not mistaken, but since we used crossover which detects on the current bar, it likely buys that bar close. Our approach was next day open; this is a known difference we should account for. We might need to adjust our test or code accordingly to align methodologies when comparing).

**Alternative libraries**: Besides Backtesting.py, other frameworks include **Backtrader** and **vectorbt**. Backtrader is a powerful backtesting engine that supports multi-asset and more complex strategies, but it has a steeper learning curve and would require writing strategy logic in Python (not via JSON) – which is less flexible for non-programmers. Still, it’s proven and could handle portfolio tests in future. **vectorbt** is a vectorized backtesting library that can handle many assets and very large datasets efficiently using NumPy/pandas. It could theoretically take our strategy rules and apply them to a matrix of data across many stocks simultaneously. This is great for performance (especially if we scale to hundreds of stocks or higher frequency data). However, vectorbt might be overkill for 40 stocks × 5 years and might require rewriting our logic into its framework. For now, a custom solution is sufficient, and we can consider vectorbt if needed later (for example, to easily compute signals across all stocks in a few lines, or to optimize parameters).

Given that we want a **transparent, easily customizable** system, the plan is to proceed with our own engine and use Backtesting.py only as a validation tool. This ensures we’re not treating Backtesting.py as a black box and we maintain control over how strategies are defined (via JSON) and executed.

## Implementation Plan Summary

To conclude, here is a step-by-step **roadmap** for development, summarizing the above sections in order:

1. **Prepare Data**: Determine the list of stocks to test (e.g., KOSPI 30 and KOSDAQ 10 stocks that meet the strategy’s universe filters). Use yfinance to download 5 years of daily OHLCV data for each stock. Handle any data cleansing (drop missing days or ensure continuous date index).
2. **Load Strategy JSON**: Define the JSON schema for strategy rules. Parse the JSON file into a Python dict. This provides the parameters for the backtest (indicators to compute, entry/exit conditions, etc.).
3. **Compute Indicators**: For each stock’s price DataFrame, calculate all indicator series required by the strategy (SMA, EMA, rolling min/max, etc.), storing them in a dictionary for easy access during simulation. Ensure alignment with price data index and manage initial NaNs (skip signals until indicators are available).
4. **Simulate Trades**: Implement the backtesting loop for one stock: iterate through each day’s data, use flag position\_open to track if currently in a trade. On each day, evaluate entry conditions (if no position) or exit conditions (if holding a position) by checking the JSON rules. When an entry signal triggers, simulate a buy on the next day’s open (record entry price, shares, etc.). When an exit triggers, sell on next open (record exit info). Track daily equity. This will yield a trade log and equity curve for that stock.
5. **Calculate Metrics**: After the loop, compute performance metrics for that stock’s results: total return, CAGR, win rate, average win/loss, profit factor, Sharpe ratio, max drawdown, etc. Store these in a results structure (dictionary or DataFrame row). Also compute any per-year returns for that stock if needed.
6. **Repeat for All Stocks**: Loop over all selected stocks, performing steps 3–5 for each. Collect all individual results. Apply any global analytics: e.g., determine how many stocks were profitable, average Sharpe ratio across stocks, etc. This gives a comprehensive view of strategy performance.
7. **Visualization**: Generate output charts. For each stock, plot its equity curve vs price with buy/sell markers (interactive). Also create summary charts (bar charts or boxplots of metrics across stocks, etc.). Make these visualizations available (for example, in a Jupyter Notebook or as HTML files).
8. **Review and Verification**: Cross-verify a subset of results with the Backtesting.py library or another method to ensure our engine’s correctness. Refine any discrepancies. Ensure that the backtest outcomes make sense (for example, check a particular trade to see if it matches the intended rule conditions at that time).
9. **Documentation and Next Steps**: Document how to add new strategies via JSON, and how to interpret the results. Note possible improvements: e.g., include transaction costs in simulation, add support for stop-loss/profit-target rules in JSON, implement multi-strategy or multi-stock portfolio simulation, or integrate a better data source for international stocks if needed.

Following this roadmap, we will create a Python implementation that is clear, modular, and easy to maintain. The AI (coding assistant) can tackle each step in sequence: data loading, JSON parsing, indicator computation, signal loop, metrics, and so on. By adhering to this structured plan, we ensure the development is straightforward and less error-prone. The final system will enable extensive backtesting of various strategies and help determine which strategies truly have stood the test of time on diverse stocks, guiding us in deploying an effective automated trading system.

**Sources:**

* Backtesting.py official documentation – highlights the importance of robustly testing a strategy on historical data and provides interactive visualization and metrics[[1]](https://kernc.github.io/backtesting.py/#:~:text=Backtesting,including%20a%20handful%20of%20tutorials)[[4]](https://kernc.github.io/backtesting.py/#:~:text=).
* Example strategy JSON (“Double Bottom Recovery v2”) – demonstrates a declarative format with indicators and conditions (e.g., SMA crossover) that our system will support[[10]](file://file-3AMt34TMiFstS8URbAHW4Q#:~:text=%7B%20,SMA_20_Close).
* AlgoTrading101’s intro to Backtesting.py – notes that Backtesting.py is easy to use, offers interactive charts, but does not support multi-asset portfolios and complex strategies without workarounds[[18]](https://algotrading101.com/learn/backtesting-py-guide/#:~:text=,Is%20actively%20maintained)[[19]](https://algotrading101.com/learn/backtesting-py-guide/#:~:text=Why%20shouldn%E2%80%99t%20I%20use%20Backtesting). This justifies our custom approach for flexibility.
* Yahoo Finance data source info – suggests using Yahoo (via yfinance) or Quandl for obtaining historical data in backtests[[6]](https://algotrading101.com/learn/backtesting-py-guide/#:~:text=Keep%20in%20mind%20that%20Backtesting,Finance%20%20or%20%2033), aligning with our data acquisition plan.
* Backtesting.py SMA crossover example – shows standard performance metrics (Sharpe, win rate, drawdown, profit factor, etc.) that we will also compute to evaluate strategies[[2]](https://kernc.github.io/backtesting.py/)[[3]](https://kernc.github.io/backtesting.py/#:~:text=Win%20Rate%20%5B,69468).

[[1]](https://kernc.github.io/backtesting.py/" \l ":~:text=Backtesting,including%20a%20handful%20of%20tutorials) [[2]](https://kernc.github.io/backtesting.py/) [[3]](https://kernc.github.io/backtesting.py/#:~:text=Win%20Rate%20%5B,69468) [[4]](https://kernc.github.io/backtesting.py/#:~:text=) [[12]](https://kernc.github.io/backtesting.py/) [[13]](https://kernc.github.io/backtesting.py/) [[15]](https://kernc.github.io/backtesting.py/#:~:text=Whenever%20the%20fast%2C%2010,CFD%20and%20can%20be%20shorted) [[16]](https://kernc.github.io/backtesting.py/#:~:text=def%20next%28self%29%3A%20if%20crossover%28self,sell) [[17]](https://kernc.github.io/backtesting.py/#:~:text=in%20the%20future,including%20a%20handful%20of%20tutorials) Backtesting.py - Backtest trading strategies in Python

<https://kernc.github.io/backtesting.py/>

[[5]](https://github.com/ranaroussi/yfinance/issues/525#:~:text=Missing%20Data%20%C2%B7%20Issue%20,Monday%2C%2010%2F29%2F2012%20and%20Tuesday%2C%2010%2F30%2F2012) Missing Data · Issue #525 · ranaroussi/yfinance - GitHub

<https://github.com/ranaroussi/yfinance/issues/525>

[[6]](https://algotrading101.com/learn/backtesting-py-guide/#:~:text=Keep%20in%20mind%20that%20Backtesting,Finance%20%20or%20%2033) [[14]](https://algotrading101.com/learn/backtesting-py-guide/#:~:text=,Is%20actively%20maintained) [[18]](https://algotrading101.com/learn/backtesting-py-guide/#:~:text=,Is%20actively%20maintained) [[19]](https://algotrading101.com/learn/backtesting-py-guide/#:~:text=Why%20shouldn%E2%80%99t%20I%20use%20Backtesting) Backtesting.py - An Introductory Guide to Backtesting with Python - AlgoTrading101 Blog

<https://algotrading101.com/learn/backtesting-py-guide/>

[[7]](file://file-3AMt34TMiFstS8URbAHW4Q#:~:text=,20) [[8]](file://file-3AMt34TMiFstS8URbAHW4Q#:~:text=%7B%20,%7D) [[9]](file://file-3AMt34TMiFstS8URbAHW4Q#:~:text=%7B%20,%7D) [[10]](file://file-3AMt34TMiFstS8URbAHW4Q#:~:text=%7B%20,SMA_20_Close) [[11]](file://file-3AMt34TMiFstS8URbAHW4Q#:~:text=,%7D) Double\_Bottom\_Recovery\_v2.json

<file://file-3AMt34TMiFstS8URbAHW4Q>