PyQuantLab

BACKTESTER v3.0.2

Strategies Manual

www.pyquantlab.com
May 2025

Table of Contents

Introduction: Strategies in the Backtester App 2

Chapter 1: The `Strategy` Class Structure 5

Chapter 2: Working with Data Inside a Strategy 12

Chapter 3: Using Indicators in Your Strategy 18

Chapter 4: Order Execution 26

Chapter 5: Position Sizing and Management 33

Chapter 6: Receiving Feedback: Notifications 39

Chapter 7: Advanced Strategy Features 45

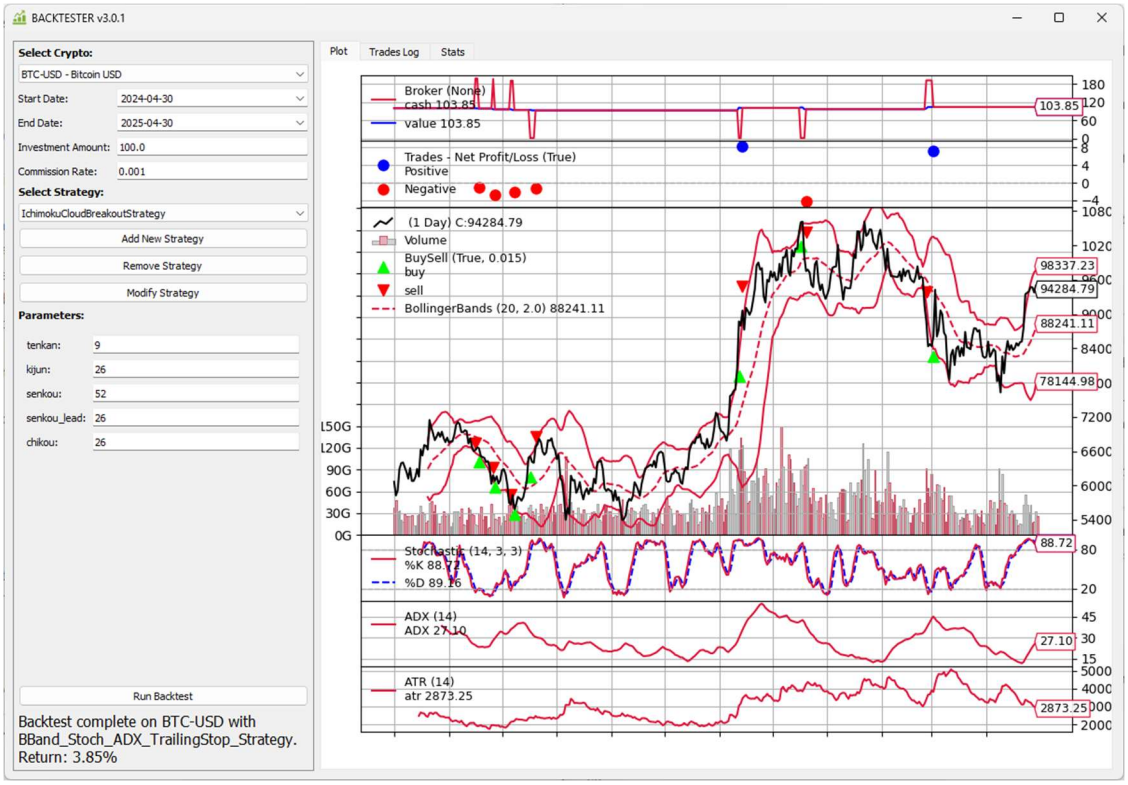
Chapter 8: Strategy Examples for Backtester 54

Appendix 68

Introduction: Strategies in the Backtester App

0.1. Welcome to Backtester

Welcome to Backtester v3.0.2! This application is designed to help you develop, test, and refine your trading strategies using historical market data before deploying them in live markets.



As you can see from the main interface (picture above), Backtester allows you to:

- Select your desired financial instrument (like BTC-USD).
- Define the historical date range for your backtest.
- Set your initial investment amount and commission rate.
- Choose, manage, and run different trading strategies.
- View and adjust strategy-specific parameters.
- Visualize the results through interactive plots and review detailed trade logs.
- See a summary of the backtest performance, including the total return.

This manual will guide you through the process of creating and managing the core logic of your trading ideas: the Strategies.

0.2. Understanding Strategies (Based on `backtrader`)

At the heart of Backtester lies the concept of a **Strategy**. A Strategy is essentially a set of rules, coded in Python, that determines when to buy, sell, or hold an asset based on market data and technical indicators.

Backtester leverages the powerful and flexible open-source `backtrader` framework. This means that the strategies you create will follow `backtrader`'s conventions and structure. If you have prior experience with `backtrader`, you'll feel right at home. If not, don't worry – this manual will cover everything you need to know to build strategies from scratch within the Backtester environment.

In essence, your strategy code will analyze the incoming price data (Open, High, Low, Close, Volume) bar by bar, calculate indicator values, check your custom conditions, and issue trading orders accordingly.

0.3. Adding, Modifying, and Removing Strategies in the App

Backtester provides a straightforward interface for managing your strategy library, visible on the left-hand panel:

- **Select Strategy:** This dropdown menu lists all the strategies currently available in the app. You choose the one you want to backtest here.
- **Parameters:** Below the strategy selection, you'll see the adjustable parameters specific to the currently selected strategy (like `tenkan`, `kijun`, `senkou`, etc. for the `IchimokuCloudBreakoutStrategy` shown).
- **Add New Strategy:** Click this button to create a new, blank strategy file where you can start coding your rules.
- **Remove Strategy:** Click this button and select a strategy to delete it from your library.
- **Modify Strategy:** Select an existing strategy and click this button to open its code, allowing you to view and edit the underlying Python logic.

This system allows you to build up a collection of different trading ideas and easily switch between them for testing.

0.4. Leveraging `backtrader`'s Functionality

When you write or modify strategies in Backtester, you are working within the `backtrader` ecosystem. This gives you access to its rich features for handling data, executing orders, managing positions, and analyzing results.

Crucially, `backtrader` includes a comprehensive set of **built-in technical indicators** (`bt.indicators.*`) and also provides seamless **wrappers for most functions in the popular TA-Lib library** (`bt.talib.*`). This means you have a vast array of technical analysis tools readily available directly within the framework, which generally simplifies development and helps avoid compatibility issues.

The following chapters will guide you through the structure of a `backtrader` strategy, how to work with data, how to use the integrated indicators (both `backtrader`'s own and its TA-Lib

wrappers), how to execute trades, and how to utilize other advanced features available within the Backtester app.

Chapter 1: The `Strategy` Class Structure

When you "Add New Strategy" or "Modify Strategy" in the Backtester app, you are working with a Python file containing a class definition. This class holds all the logic for your trading strategy. It builds upon the `backtrader` framework, providing a structured way to interact with market data, indicators, and order execution.

1.1. Inheriting from `bt.Strategy`

Every strategy you create in Backtester must be a Python class that *inherits* from `backtrader`'s base `Strategy` class. This is fundamental, as it gives your class all the built-in capabilities and expected methods of the `backtrader` engine.

At the very minimum, your strategy file will start like this:

Python

```
# Import the backtrader library
import backtrader as bt
# Import the TA-Lib library (if you plan to use it)
# import talib # Or use bt.talib wrappers

# Define your strategy class, inheriting from bt.Strategy
class MyStrategy(bt.Strategy):
    # --- Strategy parameters ---
    # params = (
    #    ('my_param', 20),
    # )

    def __init__(self):
        # --- Initialization code ---
        print("Strategy Initialized")
        # Keep a reference to the "close" line in the data[0] dataseri
es
        self.dataclose = self.datas[0].close

    def next(self):
        # --- Main logic per bar ---
        print(f"Processing bar with closing price: {self.dataclose[0]}
")
        # Example: Simple buy logic
        # if self.dataclose[0] > self.dataclose[-1]: # If close is hig
her than previous close
        #     if not self.position: # Check if not already in the mark
et
        #         self.buy()

    # --- Optional methods ---
```

```
    # def start(self):
    #     print("Strategy Started")

    # def stop(self):
    #     print("Strategy Stopped")

    # def log(self, txt, dt=None):
    #     ''' Logging function for this strategy'''
    #     dt = dt or self.datas[0].datetime.date(0)
    #     print(f'{dt.isoformat()}, {txt}')

    # def notify_order(self, order):
    #     # Handle order notifications
    #     pass

    # def notify_trade(self, trade):
    #     # Handle trade notifications
    #     pass
```

- We import the `backtrader` library, usually aliased as `bt`.
- We define a class (e.g., `MyStrategy`) that includes `(bt.Strategy)` in its definition, signifying inheritance.
- Inside the class, we define specific methods (`__init__`, `next`, etc.) that `backtrader` expects and calls during the backtest.

1.2. The Initialization Method (`__init__`)

The `__init__(self)` method is a standard Python constructor. In the context of a `backtrader` strategy, it's called **once** when the strategy is first loaded by the Backtester engine, *before* any historical data is processed.

Its primary purposes are:

1. **Accessing Data Feeds:** Getting references to the price/volume data.
2. **Defining Parameters:** Setting up adjustable variables for the strategy.
3. **Instantiating Indicators:** Creating the technical indicators you'll use.
4. **Initializing State Variables:** Setting up any other variables your strategy needs to track its internal state.

Let's break these down:

1.2.1. Accessing Data Feeds (`self.datas`, `self.data`, `self.dataX`, `self.dnames`)

When Backtester runs your strategy, it feeds it the historical data you selected (e.g., BTC-USD daily data). Inside `__init__`, you can access this data:

- `self.datas`: A list-like object containing all data feeds passed to the engine. For most single-instrument strategies in Backtester, this will contain just one item.

- `self.data` or `self.datas[0]`: A convenient alias for the *first* data feed (`self.datas[0]`). This is the most common way to access your primary instrument data.
- `self.dataX` (e.g., `self.data1`, `self.data2`): Aliases for subsequent data feeds (`self.datas[1]`, `self.datas[2]`), used if you load multiple data sources (an advanced topic).
- Data Lines: Each data feed contains several "lines" representing the Open, High, Low, Close, Volume, etc. You typically store references to the lines you need frequently.

Python

```
    def __init__(self):
        # Get a reference to the primary data feed
        self.my_data = self.datas[0] # Or simply use self.data

        # Get references to specific lines within the primary data fee
d
        self.dataclose = self.my_data.close
        self.dataopen = self.my_data.open
        self.datavolume = self.my_data.volume
```

Storing these references (like `self.dataclose`) makes accessing the data values later in the `next` method cleaner.

1.2.2. Defining Strategy Parameters (`params`)

Strategies often have parameters you might want to tweak without changing the core code (e.g., the period for a moving average). `backtrader` handles this through a special class attribute called `params`.

You define `params` as a tuple of tuples (or a dictionary). Each inner tuple contains the parameter name (string) and its default value.

Python

```
class MyStrategyWithParams(bt.Strategy):
    params = (
        ('sma_period', 20),  # Parameter for Simple Moving Average per
iod
        ('rsi_period', 14),  # Parameter for RSI period
        ('print_log', True), # A boolean parameter
    )

    def __init__(self):
        # Access parameters using self.params or self.p
        self.sma = bt.indicators.SimpleMovingAverage(
            self.datas[0], period=self.params.sma_period)
```

```
        self.rsi = bt.indicators.RelativeStrengthIndex(
            period=self.p.rsi_period) # self.p is a shortcut for self.
params

        print(f"SMA Period: {self.p.sma_period}, RSI Period: {self.p.r
si_period}")

    def next(self):
        if self.p.print_log: # Use the boolean parameter
             self.log(f"Close: {self.data.close[0]}")
```

These parameters are the ones displayed and potentially adjustable in the "Parameters" section of the Backtester app interface for the selected strategy.

1.2.3. Instantiating Indicators (See Part 3 for TA-Lib)

Technical indicators (`backtrader` built-in, TA-Lib, or custom ones) need to be created *before* they can be used. The standard place to do this is within the `__init__` method. You create an instance of the indicator and assign it to an attribute of `self`.

Python

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

        # Instantiate a Simple Moving Average indicator
        self.sma50 = bt.indicators.SimpleMovingAverage(
            self.datas[0], # Pass the data feed
            period=50      # Pass indicator-specific parameters
        )

        # Instantiate another indicator on the first one (SMA of SMA)
        self.sma_of_sma = bt.indicators.SimpleMovingAverage(
            self.sma50,    # Pass the previous indicator's output line
            period=10
        )

        # TA-Lib indicators are also instantiated here (more details i
n Part 3)
        # self.rsi = bt.talib.RSI(self.data, period=14)
```

By creating them in `__init__`, the indicators are ready to calculate their values as the backtest progresses through the historical data.

1.2.4. Initializing State Variables

You often need variables to keep track of the strategy's state across different bars. For example, you might need to store a reference to a pending order or track whether a specific condition was met on the previous bar. These should also be initialized in `__init__`.

Python

```
    def __init__(self):
        self.order_pending = None # Variable to hold a reference to an
order
        self.consecutive_up_days = 0 # Variable to count something
        self.entry_price = 0.0 # Variable to store entry price
```

Initializing them ensures they have a known starting value before the `next` method is called for the first time.

1.3. The Strategy Lifecycle Methods

`backtrader` calls specific methods on your strategy object at different points during the backtest. Understanding this lifecycle is key:

1.3.1. `start()`: One-time setup

- Called **once** at the very beginning of the `run()` process, even before `prenext` or `next`.
- Useful for any setup that doesn't depend on the data or indicator minimum periods. Often left unimplemented if `__init__` handles all setup.

1.3.2. `prenext()`: Handling the "minimum period" phase

- Called for each bar *while* `backtrader` is waiting for enough data to accumulate to satisfy the longest "minimum period" required by any indicator in your strategy (e.g., a 20-period SMA needs 20 bars before it can calculate its first value).
- Useful if you need to perform actions even before your indicators are ready, but often not needed for typical strategies. If not defined, nothing happens during this phase.

1.3.3. `nextstart()`: The bridge to active trading logic

- Called **exactly once** on the bar where all indicators have met their minimum period requirements and can start producing valid output.
- Its *default* behavior is to simply call the `next()` method.
- You can override `nextstart()` if you need to perform a specific one-time action precisely when your indicators become "live".

1.3.4. `next()`: The Heartbeat - Processing Each Bar/Tick

- This is the **most important method** for most strategies.

- Called for **every bar** of data *after* the minimum period has been reached (either directly by `backtrader` if `nextstart` is not overridden, or via the default `nextstart`).
- This is where your core logic resides:
  - Check indicator values (e.g., `self.sma50[0]`).
  - Compare prices (e.g., `self.dataclose[0] > self.sma50[0]`).
  - Check if you have an open position (`self.position`).
  - Make trading decisions (`self.buy()`, `self.sell()`).

1.3.5. `stop()`: Final actions and cleanup

- Called **once** at the very end of the backtest, after the last bar has been processed by `next()`.
- Useful for:
  - Final calculations or logging.
  - Accessing results from Analyzers (tools for calculating performance metrics like Sharpe Ratio, Drawdown, etc.).
  - Cleaning up any resources if necessary.

1.4. Logging Made Easy: The `log()` Method

While you can use simple `print()` statements, defining a dedicated `log()` method within your strategy class is highly recommended for consistent and informative output. This helps populate the "Trades Log" tab in Backtester.

A common pattern looks like this:

Python

```
    def log(self, txt, dt=None):
        ''' Logging function for this strategy'''
        # Use the datetime from the current bar if no specific date is
provided
        dt = dt or self.datas[0].datetime.date(0)
        print(f'{dt.isoformat()}, {txt}')

    def next(self):
        # Example usage within next()
        self.log(f'Close Price: {self.dataclose[0]:.2f}')

        if self.sma50[0] > self.dataclose[0]:
            self.log('SMA50 is above Close')

        # You would also call self.log() inside notify_order and notif
y_trade
```

This ensures all your strategy's output includes a timestamp and follows a standard format, making it much easier to understand the sequence of events during the backtest.

Chapter 2: Working with Data Inside a Strategy

Once your strategy's `__init__` method has run, the Backtester engine starts feeding it historical data, bar by bar. Your `next()` method (and potentially `prenext`/`nextstart`) needs to access the price, volume, and time information for each bar to make decisions.

2.1. Accessing OHLCV and Other Lines (`self.data.close`, `self.data.open`, etc.)

As established in Section 1.2.1, you typically access the primary data feed via `self.data` (which is an alias for `self.datas[0]`). This data feed object contains several attributes, known as "lines", which represent the different data points available for each bar.

The standard lines available on most datasets (like the ones from Yahoo Finance or typical broker APIs) are:

- `self.data.open`: The opening price of the bar.
- `self.data.high`: The highest price reached during the bar.
- `self.data.low`: The lowest price reached during the bar.
- `self.data.close`: The closing price of the bar.
- `self.data.volume`: The volume traded during the bar.
- `self.data.openinterest`: Open interest information (often 0 or unavailable for many data sources, especially for non-futures).
- `self.data.datetime`: Information about the date and time of the bar (see Section 2.4).

You can access the *entire series* of values for any of these lines using these attributes. For example, `self.data.close` represents the sequence of all closing prices for the duration of the backtest. However, you usually work with specific values from these lines within your `next()` method using indexing.

Python

```
import backtrader as bt

class DataAccessDemo(bt.Strategy):
    def __init__(self):
        # Store references for convenience (optional, but good practic
e)
        self.open = self.data.open
        self.high = self.data.high
        self.low = self.data.low
        self.close = self.data.close
        self.volume = self.data.volume
        print("Data references stored in __init__")

    def next(self):
        # Access specific values using indexing (see section 2.3)
```

```
        current_open = self.open[0]
        current_high = self.high[0]
        current_low = self.low[0]
        current_close = self.close[0]
        current_volume = self.volume[0]

        print(f"Bar processed - O:{current_open}, H:{current_high}, L:
{current_low}, C:{current_close}, V:{current_volume}")
```

2.2. Understanding the "Lines" Concept

In `backtrader`, a time-ordered sequence of data points (like closing prices, indicator values, or even results of calculations) is abstracted into a "Lines" object.

- **Data Feeds are Lines Objects:** `self.data` itself is a lines object, containing multiple individual lines (open, high, low, close, etc.).
- **Indicators are Lines Objects:** When you create an indicator (e.g., `self.sma = bt.indicators.SimpleMovingAverage(...)`), the indicator itself (`self.sma`) is a lines object, and its output(s) (e.g., the moving average value) are also lines.
- **Operations Create Lines Objects:** Performing arithmetic or logical operations between lines objects often results in a new lines object (e.g., `diff = self.data.close - self.data.open`).

This "lines" concept is fundamental because it allows `backtrader` to automatically synchronize all calculations. When the engine moves to the next bar, it ensures that all lines objects (data, indicators, calculations) advance together, making their current values accessible via the `[0]` index.

2.3. The Indexing Approach (`[0]` for current, `[-1]` for previous)

This is the most critical concept for working with data inside the `next()` method. Since `next()` is called iteratively for each bar, you need a way to refer to the values for the *current* bar being processed and potentially previous bars. `backtrader` uses Python's indexing syntax on its lines objects for this:

- `line[0]`: Accesses the value of the `line` for the **current** bar being processed in `next()`.
- `line[-1]`: Accesses the value of the `line` for the **previous** bar.
- `line[-2]`: Accesses the value from **two bars ago**.
- And so on...

This relative indexing is powerful because your logic doesn't need to know the absolute bar number (e.g., bar #537). You only need to define relationships between the current bar and recent past bars.

Python

```
import backtrader as bt

class IndexingDemo(bt.Strategy):
    params = (('print_log', True),)

    def __init__(self):
        self.close = self.data.close
        self.sma = bt.indicators.SimpleMovingAverage(self.data, period
=20)

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

    def next(self):
        self.log(f'Current Close: {self.close[0]:.2f}, Previous Close:
{self.close[-1]:.2f}')
        self.log(f'Current SMA: {self.sma[0]:.2f}, Previous SMA: {self
.sma[-1]:.2f}')

        # --- Example Logic using indexing ---

        # Condition 1: Did the price close higher than it opened on th
e current bar?
        if self.close[0] > self.data.open[0]:
             self.log('Closed higher than Open today.')

        # Condition 2: Is the current close above the current SMA valu
e?
        if self.close[0] > self.sma[0]:
            self.log('Close is above SMA(20).')

        # Condition 3: Did the close cross above the SMA *on this bar*
?
        # Check if previous close was below previous SMA AND current c
lose is above current SMA
        if self.close[-1] < self.sma[-1] and self.close[0] > self.sma[
0]:
            self.log('*** CROSSOVER DETECTED: Close crossed above SMA
***')
            # self.buy() # Place buy order here

        # Condition 4: Has the price closed up for 3 consecutive bars?
        if self.close[0] > self.close[-1] and \
           self.close[-1] > self.close[-2] and \
```

```
            self.close[-2] > self.close[-3]:
             self.log('Price closed up for 3 consecutive bars.')
```

Important: Using positive indices like `line[1]` is generally **not** recommended within `next()`, as it implies looking into the future, which leads to unrealistic backtest results (look-ahead bias). Always use `[0]` for the current bar and negative indices for past bars.

2.4. Working with Datetime (`self.data.datetime`)

Each bar is associated with a specific date and time. The `datetime` line provides access to this information.

- `self.data.datetime`: Represents the sequence of timestamps for all bars.
- `self.data.datetime[0]`: Within `next()`, this gives the timestamp for the **current** bar. The exact value depends on the data's timeframe (e.g., for daily data, it might represent the date; for minute data, the date and time at the end of that minute). `backtrader` internally converts this to a float representation for processing, but provides methods to get standard Python objects.

Common ways to use the datetime object for the current bar `[0]`:

Python

```
    def next(self):
        # Get the raw float representation (rarely needed directly)
        # raw_dt_float = self.data.datetime[0]

        # Get a Python date object (useful for daily data)
        current_date = self.data.datetime.date(0)

        # Get a Python time object (useful for intraday data)
        current_time = self.data.datetime.time(0)

        # Get a Python datetime object
        current_datetime = self.data.datetime.datetime(0)

        # Get parts of the date/time
        current_year = current_datetime.year
        current_month = current_datetime.month
        current_day = current_datetime.day
        current_weekday = current_datetime.weekday() # Monday is 0, Su
nday is 6
        current_hour = current_datetime.hour # For intraday data

        # Example logging
        if current_weekday == 4: # If it's Friday (weekday index 4)
             self.log(f"Processing bar for Friday {current_date.isofor
mat()}")
```

```
        self.log(f"Current Datetime: {current_datetime}")
```

The `log` method example in Section 1.4 demonstrated using `self.datas[0].datetime.date(0)` to automatically timestamp log messages.

2.5. Handling Multiple Data Feeds

While many strategies focus on a single instrument, Backtester (via `backtrader`) supports adding multiple data feeds to your analysis. This could be for:

- Comparing two related assets (e.g., stock vs index for Beta calculation, pairs trading).
- Using a higher timeframe for context (e.g., checking the daily trend while trading on hourly data).
- Using economic data alongside price data.

If you were to load multiple data feeds into the Backtester engine (this setup happens outside the strategy code itself), you would access them within the strategy using the `self.datas` list or the `self.dataX` aliases:

- `self.data` or `self.datas[0]` or `self.data0`: The first data feed added. This typically drives the main clock/timing of the backtest.
- `self.data1` or `self.datas[1]`: The second data feed added.
- `self.data2` or `self.datas[2]`: The third data feed added.
- ...and so on.

Python

```
import backtrader as bt

# Assume data0 is AAPL daily, data1 is SPY daily
class MultiDataDemo(bt.Strategy):
    def __init__(self):
        # Access lines from the first data feed (AAPL)
        self.aapl_close = self.datas[0].close

        # Access lines from the second data feed (SPY)
        self.spy_close = self.datas[1].close

        print("Strategy initialized with 2 data feeds")

    def next(self):
        # Access current values from both feeds using [0]
        current_aapl_close = self.aapl_close[0]
        current_spy_close = self.spy_close[0]

        # Access previous values using [-1]
```

```
        prev_aapl_close = self.aapl_close[-1]
        prev_spy_close = self.spy_close[-1]

        print(f"Date: {self.data.datetime.date(0)}")
        print(f"  AAPL Close: {current_aapl_close:.2f} (Prev: {prev_aa
pl_close:.2f})")
        print(f"  SPY Close : {current_spy_close:.2f} (Prev: {prev_spy
_close:.2f})")

        # Example: Check if AAPL outperformed SPY on the previous day
        aapl_ret = (self.aapl_close[-1] / self.aapl_close[-2]) - 1
        spy_ret = (self.spy_close[-1] / self.spy_close[-2]) - 1

        if aapl_ret > spy_ret:
            print("  AAPL outperformed SPY yesterday.")
```

Synchronization: `backtrader` automatically handles the synchronization between multiple data feeds. The `next()` method will only be called when *all* active data feeds have a bar corresponding to the timestamp of the *primary* data feed (`self.datas[0]`). If one feed has missing data for a particular day, `next()` might not be called for that day.

Chapter 3: Using Indicators in Your Strategy

Technical indicators are the backbone of many trading strategies, helping to analyze market conditions and generate potential trading signals. Backtester, via the `backtrader` framework, provides easy access to a wide range of indicators from two main sources:

5. **`backtrader`'s built-in indicators:** A collection of common indicators directly implemented within the framework (`bt.indicators.*`).
6. **TA-Lib wrappers:** Seamless integration with most functions from the popular TA-Lib library (`bt.talib.*`).

This chapter explains how to instantiate, access, and manage these indicators within your strategies.

3.1. Instantiating Indicators in `__init__`

Regardless of whether you're using a built-in `backtrader` indicator or a TA-Lib wrapper, the process starts in your strategy's `__init__` method. Here, you create instances of the indicators you need and assign them to attributes of `self`. This makes them available throughout the lifetime of your strategy instance.

Python

```
import backtrader as bt

class IndicatorInitDemo(bt.Strategy):
    params = (
        ('ma_period', 20),
        ('rsi_period', 14),
    )

    def __init__(self):
        # Get reference to close price line
        self.dataclose = self.datas[0].close

        # Instantiate a built-in backtrader indicator
        self.sma = bt.indicators.SimpleMovingAverage(
            self.datas[0], # Can operate on the data feed directly
            period=self.p.ma_period
        )

        # Instantiate a TA-Lib indicator using the backtrader wrapper
        self.rsi = bt.talib.RSI(
            self.dataclose, # Can also operate on a specific line
            timeperiod=self.p.rsi_period # Note: TA-Lib often uses 'ti
meperiod'
        )
```

```
        self.log('Indicators have been instantiated in __init__')

    # Required methods log, next, notify_order etc. would follow
    def log(self, txt, dt=None): # Basic log method
      dt = dt or self.datas[0].datetime.date(0); print(f'{dt.isoformat
()}, {txt}')
    def next(self): pass # Placeholder
```

Creating indicators in `__init__` ensures they are set up correctly before the engine starts processing historical bars.

3.2. Accessing Indicator Values in `next()` (using `[0]`, `[-1]`, attribute access)

Once instantiated, the indicator objects (like `self.sma` or `self.rsi` above) behave like `backtrader` "lines" objects (see Chapter 2.2). Inside your `next()` method (after the initial minimum period), you access their calculated values using the standard relative indexing:

- `self.indicator_name[0]`: Gets the calculated value for the **current** bar.
- `self.indicator_name[-1]`: Gets the value from the **previous** bar.
- `self.indicator_name[-n]`: Gets the value from **n** bars ago.

For indicators that produce multiple output lines (like MACD or Bollinger Bands), you access the specific lines using attributes before applying the index:

- `self.macd.lines.macd[0]` or simply `self.macd.macd[0]`
- `self.macd.lines.signal[0]` or `self.macd.signal[0]`
- `self.bbands.lines.top[0]` or `self.bbands.top[0]` (Note: Line names can vary slightly between built-in and TA-Lib versions, check documentation if unsure).

Python

```
    def next(self):
        # Access the current calculated value of the SMA
        current_sma_value = self.sma[0]

        # Access the current and previous RSI values
        current_rsi_value = self.rsi[0]
        previous_rsi_value = self.rsi[-1]

        self.log(f'Close: {self.dataclose[0]:.2f}, SMA: {current_sma_v
alue:.2f}, RSI: {current_rsi_value:.2f}')

        # Example using indicator values in logic
        if self.dataclose[0] > current_sma_value and current_rsi_value
< 70:
            self.log("Close above SMA and RSI not overbought")
            # Potentially place a buy order here
```

3.3. `backtrader` Built-in Indicators (`bt.indicators.*`)

This module contains `backtrader`'s natively implemented indicators.

- **Common Examples:**
  - `bt.indicators.SimpleMovingAverage` (SMA)
  - `bt.indicators.ExponentialMovingAverage` (EMA)
  - `bt.indicators.WeightedMovingAverage` (WMA)
  - `bt.indicators.MACD` (Moving Average Convergence Divergence - provides `macd`, `signal`, `histo` lines)
  - `bt.indicators.RSI` (Relative Strength Index)
  - `bt.indicators.Stochastic` (Provides `percK`, `percD` lines)
  - `bt.indicators.BollingerBands` (Provides `top`, `mid`, `bot` lines)
  - `bt.indicators.ATR` (Average True Range)
  - `bt.indicators.CCI` (Commodity Channel Index)
  - `bt.indicators.Momentum`
  - `bt.indicators.ROC` (Rate of Change)
  - `bt.indicators.StandardDeviation`
  - *...and many others.*
- **Usage Example:**

Python

```
import backtrader as bt

class BuiltInIndicatorsDemo(bt.Strategy):
    params = (('macd1', 12), ('macd2', 26), ('macdsig', 9), ('stoch_k'
, 14), ('stoch_d', 3))

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

        self.macd = bt.indicators.MACD(
            self.dataclose,
            period_me1=self.p.macd1, # Note different param names some
times
            period_me2=self.p.macd2,
            period_signal=self.p.macdsig
        )

        self.stoch = bt.indicators.Stochastic(
            self.data, # Can operate on the data feed
            period=self.p.stoch_k,
            period_dfast=self.p.stoch_d # Note different param names s
```

```
ometimes
            # Default is Slow Stochastic
        )

        # Using an indicator on another indicator's output
        self.macd_sma = bt.indicators.SimpleMovingAverage(self.macd.ma
cd, period=5)


    def log(self, txt, dt=None):
        dt = dt or self.datas[0].datetime.date(0); print(f'{dt.isoform
at()}, {txt}')

    def next(self):
        # Access MACD lines
        current_macd = self.macd.macd[0]
        current_signal = self.macd.signal[0]
        current_histo = self.macd.histo[0]

        # Access Stochastic lines
        current_stoch_k = self.stoch.percK[0] # Check docs for line na
mes
        current_stoch_d = self.stoch.percD[0]

        # Access SMA of MACD line
        current_macd_sma = self.macd_sma[0]

        self.log(f'MACD: {current_macd:.2f}, Sig: {current_signal:.2f}
, Histo: {current_histo:.2f}')
        self.log(f'Stoch %K: {current_stoch_k:.2f}, %D: {current_stoch
_d:.2f}')
        self.log(f'SMA(MACD): {current_macd_sma:.2f}')
```

3.4. Using TA-Lib via `backtrader` Wrappers (`bt.talib.*`)

`backtrader` significantly extends the range of available indicators by providing wrappers that allow you to use most functions from the TA-Lib library as if they were native `backtrader` indicators.

- **Explanation of Wrappers:** These wrappers (accessed via `bt.talib.FunctionName`) are highly recommended over using `talib` directly because they:
  - Accept `backtrader` data/indicator lines as input automatically.
  - Return standard `backtrader` lines objects as output.

  - Correctly handle `NaN` values during the initial calculation period, integrating with `backtrader`'s minimum period system.
  - Work seamlessly with plotting and the engine's data synchronization.
- **Common Examples (`bt.talib.*`):** (Note: Parameter names like `timeperiod` are common in TA-Lib)
  - `bt.talib.SMA(line, timeperiod=...)`
  - `bt.talib.EMA(line, timeperiod=...)`
  - `bt.talib.RSI(line, timeperiod=...)`
  - `bt.talib.MACD(line, fastperiod=..., slowperiod=..., signalperiod=...)` - Returns `macd`, `macdsignal`, `macdhist` lines.
  - `bt.talib.BBANDS(line, timeperiod=..., nbdevup=..., nbdevdn=...)` - Returns `upperband`, `middleband`, `lowerband` lines.
  - `bt.talib.ATR(high_line, low_line, close_line, timeperiod=...)`
  - `bt.talib.STOCH(high, low, close, fastk_period=..., slowk_period=..., slowd_period=...)` - Returns `slowk`, `slowd` lines.
  - `bt.talib.ADX(high, low, close, timeperiod=...)`
  - `bt.talib.CDLDOJI(open, high, low, close)` - Returns a boolean-like line (evaluates True if pattern found).
  - *... and hundreds more covering all TA-Lib categories.*
- **Usage Example:**

Python

```
import backtrader as bt
# No need to 'import talib' when using wrappers

class TALibWrapperDemo(bt.Strategy):
    params = (
        ('bb_period', 20),
        ('atr_period', 14),
    )

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

        # Instantiate TA-Lib indicators via wrappers
        self.bbands = bt.talib.BBANDS(
            self.dataclose,
            timeperiod=self.p.bb_period,
            nbdevup=2.0,
```

```
            nbdevdn=2.0
        )

        self.atr = bt.talib.ATR(
            self.datahigh, self.datalow, self.dataclose,
            timeperiod=self.p.atr_period
        )

        self.doji = bt.talib.CDLDOJI( # Candlestick pattern
             self.dataopen, self.datahigh, self.datalow, self.dataclos
e
        )

    def log(self, txt, dt=None):
        dt = dt or self.datas[0].datetime.date(0); print(f'{dt.isoform
at()}, {txt}')

    def next(self):
        # Access Bollinger Bands lines
        upper_band = self.bbands.upperband[0]
        middle_band = self.bbands.middleband[0]
        lower_band = self.bbands.lowerband[0]

        # Access ATR value
        current_atr = self.atr[0]

        # Access Doji pattern result (evaluates True if pattern detect
ed)
        is_doji = self.doji[0]

        self.log(f'Close={self.dataclose[0]:.2f}, LowerB={lower_band:.
2f}, UpperB={upper_band:.2f}, ATR={current_atr:.2f}')

        if is_doji:
             self.log('*** Doji Pattern Detected! ***')
```

Using the `bt.talib.*` wrappers gives you access to a vast library of indicators with minimal friction within the Backtester environment.

3.5. Creating Custom Indicators (Brief Overview)

If you need a unique indicator not found in `backtrader`'s built-ins or TA-Lib, you can develop your own. This involves creating a Python class that inherits from `bt.Indicator`, defining its parameters and output lines, and implementing the calculation logic. This is an advanced feature; consult the `backtrader` documentation for details if required.

3.6. Indicator Plotting Control (`plotinfo` dictionary)

As seen in the Backtester app's plot pane, indicators are automatically visualized. You can control *how* they are plotted using the `plotinfo` parameter when instantiating an indicator.

Python

```
    def __init__(self):
        # SMA - make sure it's on the main plot (subplot=False)
        # and give it a custom name
        self.sma = bt.indicators.SimpleMovingAverage(
            self.datas[0], period=50,
            plotinfo=dict(subplot=False, plotname='SMA(50)')
        )

        # RSI - Ensure it's in a subplot with lines at 30/70
        self.rsi = bt.talib.RSI(
             self.dataclose, timeperiod=14,
             # Can pass plotinfo directly to wrappers too
             plotinfo=dict(subplot=True, plothlines=[30.0, 70.0])
        )

        # ATR - Plot in a subplot, maybe above price data
        self.atr = bt.talib.ATR(
             self.datahigh, self.datalow, self.dataclose, timeperiod=1
4
        )
        # Modify plotinfo after instantiation
        self.atr.plotinfo.subplot = True
        self.atr.plotinfo.plotabove = True
        self.atr.plotinfo.plotname = f'ATR({self.atr.p.timeperiod})'

        # Don't plot this indicator at all
        self.ema_noplot = bt.indicators.EMA(self.dataclose, period=10,
plot=False)
```

Common `plotinfo` keys:

- `plot=True/False`: Enable/disable plotting.
- `subplot=True/False`: Plot in separate panel or overlay on price data.
- `plotname='Custom Name'`: Name shown on the plot.
- `plotabove=True/False`: If subplot, plot above price instead of below.
- `plothlines=[value1, value2]`: Draw horizontal lines at specified y-values.
- `plotyticks=[value1, value2]`: Suggest specific y-axis ticks.

Customizing `plotinfo` helps tailor the charts generated by Backtester to best visualize the indicators relevant to your strategy.

Chapter 4: Order Execution

Once your strategy logic, using data and indicators, determines it's time to enter or exit a position, you need to communicate that decision to the trading engine. This is done by creating trade orders.

4.1. Interacting with the Broker (`self.broker`)

Backtester simulates a brokerage account for each backtest. This simulation handles your cash balance, tracks your open positions, calculates portfolio value, applies commissions, and processes your trade orders.

While you don't usually interact with *all* aspects of the broker directly within the main strategy logic, your primary interaction happens through placing orders using methods provided by the `Strategy` class itself (like `buy()`, `sell()`, `close()`). These methods, behind the scenes, create order objects and submit them to the simulated broker (`self.broker`) for processing. The broker object holds the current state of your simulated account (e.g., `self.broker.get_cash()`, `self.broker.get_value()`), which we'll explore more in later chapters.

4.2. Placing Basic Orders: `buy()`, `sell()`, `close()`

These are the fundamental methods called from within your `next()` method to initiate trades based on your strategy's signals:

- `self.buy()`: Creates an order to **enter a long position** (if not already long) or to **cover/reduce a short position**.
- `self.sell()`: Creates an order to **enter a short position** (if not already short) or to **sell/reduce a long position**.
- `self.close()`: Creates an order to **close the current open position** for the default data feed (`self.data`). It automatically creates a sell order if you are long, or a buy order if you are short, with the correct size to flatten the position.

Python

```
import backtrader as bt
# Assuming other imports like numpy, talib if used by conditions

class BasicOrderDemo(bt.Strategy):
    params = (('exit_bars', 5),) # Exit after holding 5 bars

    def __init__(self):
        self.dataclose = self.datas[0].close
        self.order = None # To track pending orders
        self.buyprice = None
        self.buycomm = None
        self.bar_executed = 0 # When was the trade executed
```

```
    def notify_order(self, order): # Covered in Chapter 6
        if order.status in [order.Submitted, order.Accepted]:
            return # Buy/Sell order submitted/accepted - Nothing to do

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

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

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

        # Write down: no pending order
        self.order = None

    def log(self, txt, dt=None):
        dt = dt or self.datas[0].datetime.date(0)
        # Use current time in log message format
        current_timestamp = self.datas[0].datetime.datetime(0).strftim
e("%Y-%m-%d %H:%M:%S") if hasattr(self.datas[0].datetime, 'datetime')
else dt.isoformat()
        print(f'{current_timestamp}, {txt}')


    def next(self):
        # Log the closing price of the current bar
        self.log(f'Close: {self.dataclose[0]:.2f}')


        # Check if an order is pending ... if yes, we cannot send a 2n
d one
        if self.order:
            return
```

```
        # Check if we are in the market
        if not self.position: # Not in market
            # Example entry condition: Simple Crossover (replace with
your logic)
            # Check if enough data exists for the condition
            if len(self.dataclose) > 2: # Need at least 3 bars for thi
s condition
                if self.dataclose[0] > self.dataclose[-1] and self.dat
aclose[-1] < self.dataclose[-2]:
                    self.log('BUY CREATE, %.2f' % self.dataclose[0])
                    # Keep track of the created order to avoid a 2nd o
rder
                    self.order = self.buy()

        else: # Already in the market
            # Example exit condition: Close after holding for N bars
            if len(self) >= (self.bar_executed + self.p.exit_bars):
                self.log('CLOSE CREATE, %.2f' % self.dataclose[0])
                # Keep track of the created order
                self.order = self.close()
```

- **Order Creation vs Execution:** Calling `buy()`, `sell()`, or `close()` only *creates* the order request. The actual execution (when the trade occurs and at what price) depends on the order type and the simulation timing rules.
- **Size:** You can specify the size of the trade (e.g., `self.buy(size=10)`). If omitted, `backtrader` uses a "Sizer" object (defaulting to a size of 1 unit). Sizers are covered in Chapter 5.

4.3. Understanding Order Execution Timing (Default: Next Bar Open)

A critical concept in backtesting is *when* orders are assumed to be executed relative to when the signal occurs. By default, `backtrader` simulates a realistic delay:

- If your logic inside `next()` for bar `N` generates a `buy()` or `sell()` signal (typically using data *up to* the close of bar `N`), the order is submitted to the simulated broker.
- The broker, by default, executes standard **Market orders** at the **opening price of the *next* bar (bar `N+1`)**.

This mimics the real-world scenario where you observe the market close, make a decision, place an order, and it gets filled when the market reopens or at the next available tick.

While `backtrader` offers ways to modify this timing (e.g., "Cheat-on-Close" settings, covered in advanced chapters), you should generally assume this "next bar open" execution for standard Market orders unless you explicitly configure otherwise.

4.4. Order Types (`exectype` parameter)

The `buy()` and `sell()` methods can create different types of orders by specifying the `exectype` parameter. The most common types are:

4.4.1. `Order.Market`

- `self.buy(exectype=bt.Order.Market)` or simply `self.buy()`
- `self.sell(exectype=bt.Order.Market)` or simply `self.sell()`
- This is the **default** type.
- Executes at the best available price as soon as possible.
- In backtesting simulation: Typically executes at the **open of the next bar**.
- Does not require a `price` parameter.

4.4.2. `Order.Limit`

- `self.buy(exectype=bt.Order.Limit, price=desired_buy_price)`
- `self.sell(exectype=bt.Order.Limit, price=desired_sell_price)`
- Executes only at the specified `price` or a *better* price (lower for buy, higher for sell).
- **Requires** the `price` parameter.
- In simulation: Checked against the H/L/O prices of subsequent bars. For example, a buy limit might fill if the low of the next bar is at or below the limit price. The fill price could be the limit price itself or the open if the open is already better than the limit.

4.4.3. `Order.Stop`

- `self.buy(exectype=bt.Order.Stop, price=trigger_buy_price)`
- `self.sell(exectype=bt.Order.Stop, price=trigger_sell_price)`
- Triggers a **Market order** *if* the market price touches or moves beyond the specified `price`.
- **Requires** the `price` parameter (the trigger price).
- Commonly used for:
  - **Stop-Loss:** A sell stop below the current price for a long position, or a buy stop above the current price for a short position.
  - **Entry on Breakout:** A buy stop above the current price, or a sell stop below the current price.
- In simulation: If the high/low of the next bar crosses the stop `price`, a market order is triggered and typically filled at the open of the *following* bar, or potentially at the stop price itself depending on simulation details.

4.4.4. `Order.StopLimit`

- `self.buy(exectype=bt.Order.StopLimit, price=trigger_buy_price, pricelimit=limit_buy_price)`
- `self.sell(exectype=bt.Order.StopLimit, price=trigger_sell_price, pricelimit=limit_sell_price)`

- Combines Stop and Limit features. Triggers *if* the market price touches or moves beyond the `price` (stop price).
- *If triggered*, it then places a **Limit order** at the `pricelimit`.
- **Requires** both `price` (trigger) and `pricelimit` (limit).
- Offers more price control than a regular Stop order after the trigger, but carries the risk the limit order might not fill if the price moves quickly past the `pricelimit`.

4.4.5. `Order.Close`

- `self.buy(exectype=bt.Order.Close)`
- `self.sell(exectype=bt.Order.Close)`
- This type is intended to simulate a "Market-on-Close" (MOC) order.
- It aims to execute at the closing price of the bar.
- In `backtrader`'s default simulation, this usually means the order created during bar `N` executes at the **closing price of bar `N+1`**. This behavior can be altered by "Cheat-on-Close" settings (advanced topic).
- Note: This is different from the `self.close()` *method*, which creates a standard *Market* order to exit a position on the *next bar's open*. `Order.Close` specifically targets a close price execution.

4.5. Specifying Price Parameters (`price`, `pricelimit`)

As seen above, certain order types require you to specify price levels:

- `price`: Used by `Limit`, `Stop`, and `StopLimit` orders to define the limit level or the trigger level.
- `pricelimit`: Used *only* by `StopLimit` orders to define the price for the subsequent limit order once the stop `price` is triggered.

Market orders do not use these parameters as they execute at the prevailing market price.

4.6. Order Validity (`valid` parameter, Good-'til-Canceled)

You can control how long an order remains active if it's not immediately filled using the `valid` parameter:

- `valid=None` (Default): The order is **Good 'til Canceled (GTC)**. It stays active in the simulation across multiple bars until it's either filled or you explicitly cancel it using `self.cancel()`. *Note: Real brokers might impose very long-term expiration dates even on GTC orders.*

- `valid=datetime_object`: Provide a specific `datetime.datetime` or `datetime.date` object. The order becomes **Good 'til Date (GTD)** and will expire if not filled by that time.

  Python

```
        import datetime
        # Make sure expiry_date is in the future relative to the backtest
        data
        expiry_date = datetime.date(2025, 12, 31) # Example date
        # Ensure the date is valid for the data timeframe being used
        # For example, if using daily data, specify just the date
        # If using intraday, specify datetime
        try:
            # Check if backtest end date is available, requires Cerebro s
        etup info
            # backtest_end_date = self.datas[0].p.todate # This might not
        be directly available
            # A safer approach might be to set a date far in the future i
        f GTC isn't suitable
            # Or calculate based on current bar's date
            current_date = self.datas[0].datetime.date(0)
            expiry_date_relative = current_date + datetime.timedelta(days
        =30) # Example: valid for 30 days
            self.buy(exectype=bt.Order.Limit, price=100.0, valid=expiry_d
        ate_relative)
        except IndexError: # Handle cases where date might not be availab
        le yet
            self.log("Cannot determine current date yet for GTD order.")
```

- `valid=bt.Order.DAY` or `valid=0` or `valid=datetime.timedelta()`: Creates a **Day Order**. If not filled during the session it's placed in (often interpreted as the next bar in basic backtesting), it expires automatically.

4.7. Cancelling Pending Orders (`cancel(order)`)

Sometimes you might want to cancel an order you placed earlier (e.g., a limit order that hasn't filled and market conditions have changed).

7. **Store the Order Reference:** When you call `buy()` or `sell()`, the method returns an `Order` object. You need to store this reference if you intend to cancel it later.
8. **Call `self.cancel()`:** Pass the stored order reference to the `self.cancel()` method.

Python

```
    def __init__(self):
        # ... other initializations ...
        self.my_limit_order = None # Variable to hold the order refere
nce

    def next(self):
        # --- Placing a limit order ---
```

```
        # Ensure we have data before accessing low[0]
        if len(self.data.low) > 0 and self.my_limit_order is None and
some_condition: # Only place if no order pending
            limit_price = self.data.low[0] - 0.5 # Example price
            self.log(f'Placing BUY LIMIT order at {limit_price:.2f}')
            self.my_limit_order = self.buy(exectype=bt.Order.Limit, pr
ice=limit_price, size=10)

        # --- Cancelling the order ---
        if some_other_condition and self.my_limit_order is not None:
            # Check if the order is still active before cancelling
            # Use getstatusname() for readable status
            order_status = self.my_limit_order.status
            if order_status in [bt.Order.Submitted, bt.Order.Accepted,
bt.Order.Partial]: # Active states
                self.log(f'Cancelling order {self.my_limit_order.ref}
')
                self.cancel(self.my_limit_order)
                # self.my_limit_order will be set to None in notify_o
rder
                # when cancellation is confirmed (status becomes Canc
eled)
            else:
                # Order already filled, completed, or rejected - cann
ot cancel
                status_name = self.my_limit_order.getstatusname() if
hasattr(self.my_limit_order, 'getstatusname') else order_status
                self.log(f'Cannot cancel order {self.my_limit_order.r
ef}, status: {status_name}')
                # It might be prudent to set self.my_limit_order = No
ne here too,
                # as it's no longer a pending, cancellable order.
                self.my_limit_order = None

    # Assume notify_order exists to handle setting self.my_limit_order
= None on final states
    # Assume log method exists
    # Assume some_condition and some_other_condition are defined
```

- **Cancellation is a Request:** Calling `self.cancel()` submits a cancellation request. It's possible the order could be filled just before the cancellation is processed by the simulated broker. You need to check the order status updates via the `notify_order` method (covered in Chapter 6) to confirm if cancellation was successful.

Chapter 5: Position Sizing and Management

Simply generating buy or sell signals isn't enough; you also need to decide *how much* to trade. This is known as position sizing, and it's a critical component of risk management. Backtester, through `backtrader`, offers several ways to manage the size of your orders.

5.1. Default Sizing

If you call `self.buy()` or `self.sell()` without specifying a `size` argument, `backtrader` uses a default position sizing mechanism. By default, this mechanism is usually a **Fixed Size Sizer** configured to trade **1 unit** (e.g., 1 share, 1 contract, 1 coin) per order.

Python

```
    def next(self):
        if some_buy_condition:
            # This will typically buy 1 unit if no other sizer is conf
igured
            self.buy()
```

So, unless you specify otherwise (either via the methods below or potentially through settings in the main Backtester app interface), simple `buy()` and `sell()` calls will trade a quantity of 1.

5.2. Using `backtrader` Sizers (`bt.Sizer`, `addsizer`)

`backtrader` has a concept of **Sizers**, which are reusable components designed to automatically calculate the trade size based on predefined rules. While the configuration of which Sizer to use is typically done *outside* the strategy code (likely in the Backtester app's main setup or configuration for a specific backtest run, using `cerebro.addsizer(...)`), it's important to understand how your strategy interacts with them.

Common built-in Sizers include:

- `bt.sizers.FixedSize`: Trades a fixed number of units specified by a `stake` parameter (the default sizer uses this with `stake=1`).
- `bt.sizers.PercentSizer`: Calculates the size based on a percentage of the available cash or portfolio value (configurable) in the simulated broker account. For example, `percents=10` would try to use 10% of the base for the trade.
- `bt.sizers.AllInSizer`: Uses (almost) all available cash or portfolio value for the trade, effectively going "all in".
- `bt.sizers.FixedReverser`: Closes the current position and enters an opposite position of a fixed size.

How it works with your strategy: If a Sizer (other than the default FixedSize=1) has been configured for your backtest run in the Backtester app, simply calling `self.buy()` or `self.sell()` **without** the `size` parameter will automatically invoke that Sizer's logic to determine the trade quantity. Your strategy code doesn't need to change, but the resulting

trade size will vary based on the active Sizer and the current account state (like cash balance or portfolio value).

5.3. Setting Size Directly in buy/sell

You always have the option to **override** any active Sizer and specify the exact trade quantity directly within your strategy logic by using the `size` parameter in your `buy()` or `sell()` calls.

Python

```
import backtrader as bt
# Assuming condition_A, condition_B, condition_C are defined elsewhere

class DirectSizingDemo(bt.Strategy):
    params = (
        ('fixed_trade_size', 10),
        ('risk_percent', 0.02), # e.g., risk 2% of portfolio per trade
        ('atr_period', 14),
        ('atr_stop_multiplier', 2.0) # Stop distance in multiples of A
TR
        )

    def __init__(self):
        self.dataclose = self.data.close
        self.datahigh = self.data.high
        self.datalow = self.data.low
        self.order = None

        # Instantiate ATR using bt.talib wrapper for risk sizing
        self.atr = bt.talib.ATR(self.datahigh, self.datalow, self.data
close,
                                timeperiod=self.p.atr_period) # Note:
timeperiod param

    def notify_order(self, order): # Basic order notification
        if order.status in [order.Submitted, order.Accepted]: return
        if order.status in [order.Completed]:
            exec_type = 'BUY' if order.isbuy() else 'SELL'
            self.log(f'{exec_type} EXECUTED, Size: {order.executed.siz
e}, Price: {order.executed.price:.2f}')
        elif order.status in [order.Canceled, order.Margin, order.Reje
cted]:
            self.log(f'Order Canceled/Margin/Rejected')
        self.order = None # Reset pending order flag

    def log(self, txt, dt=None):
```

```
        dt = dt or self.datas[0].datetime.date(0)
        # Use current time in log message format
        current_timestamp = self.datas[0].datetime.datetime(0).strftim
e("%Y-%m-%d %H:%M:%S") if hasattr(self.datas[0].datetime, 'datetime')
else dt.isoformat()
        print(f'{current_timestamp}, {txt}')

    def next(self):
        # Example 1: Trade a fixed number of units defined by a parame
ter
        if condition_A and not self.order and not self.position:
            self.log(f"BUYING {self.p.fixed_trade_size} units")
            self.order = self.buy(size=self.p.fixed_trade_size)

        # Example 2: Trade a specific fractional size (e.g., for crypt
o)
        elif condition_B and not self.order and not self.position:
             self.log("BUYING 0.5 units")
             self.order = self.buy(size=0.5)

        # Example 3: Simple risk-based sizing using ATR (Illustrative)
        elif condition_C and not self.order and not self.position:
            # Get the current ATR value from the instantiated indicato
r
            current_atr = self.atr[0]

            # Ensure ATR is valid (not NaN) and positive
            # Use bt.numsupport.isnan for robustness within backtrader
            if current_atr is not None and not bt.numsupport.isnan(cur
rent_atr) and current_atr > 0:
                portfolio_value = self.broker.get_value() # Use portfo
lio value
                price = self.data.close[0] # Use current close as esti
mated entry price
                cash = self.broker.get_cash()

                # Calculate stop distance based on ATR
                stop_distance_points = self.p.atr_stop_multiplier * cu
rrent_atr

                # Calculate amount to risk based on portfolio percenta
ge
                risk_amount = portfolio_value * self.p.risk_percent

                # Calculate risk per unit (share/coin/contract)
                # NOTE: This assumes 1 point move = $1 risk per unit.
```

```
                # Adjust for futures point values, forex pip values, e
tc.
                risk_per_unit = stop_distance_points

                trade_size = 0 # Default size if calculation fails
                if risk_per_unit > 1e-9: # Avoid division by zero
                    calculated_size = risk_amount / risk_per_unit

                    # Ensure size is reasonable (e.g., round down, app
ly minimums/multiples)
                    # Example rounding for crypto/fractional: round do
wn to 3 decimal places
                    trade_size = max(0.0, int(calculated_size * 1000)
/ 1000.0)

                    # Final checks
                    required_cash = trade_size * price # Rough estimat
e, ignores commission/slippage
                    if trade_size <= 0:
                         self.log("Sizing resulted in 0 size.")
                    elif required_cash > cash:
                         self.log(f"Insufficient cash for calculated s
ize. Need {required_cash:.2f}, Have {cash:.2f}")
                         trade_size = 0 # Cannot afford trade
                    else:
                         self.log(f"ATR={current_atr:.2f}, RiskAmt={ri
sk_amount:.2f}, Risk/Unit={risk_per_unit:.2f}, TradeSize={trade_size}"
)
                         self.order = self.buy(size=trade_size) # Plac
e the sized order
                else:
                    self.log("Cannot calculate size: Risk per unit is
zero or near zero.")

            else:
                 self.log(f"ATR is not valid yet or zero: {current_atr
}")

        # Placeholder for sell/close logic if needed
        elif self.position and some_exit_condition:
             self.order = self.close() # Uses default size (1) or Size
r unless size specified
```

Specifying `size` directly gives you complete control for that specific order, bypassing any globally configured Sizer for that trade.

5.4. Checking Your Position (`self.position`)

It's essential for your strategy to know whether it currently holds an open position in the market before deciding to enter a new one or exit an existing one. The `self.position` attribute provides this information for the primary data feed (`self.data`).

- **Boolean Context:** You can use `self.position` directly in `if` statements:
  - `if self.position:` evaluates to `True` if you currently hold *any* position (long or short).
  - `if not self.position:` evaluates to `True` if you are currently flat (no position).
- **Position Details:** If a position exists, `self.position` is an object containing details about that position.

5.4.1. `self.position.size`

This attribute tells you the **quantity** of the asset you currently hold.

- `self.position.size > 0`: You have a **long** position of that size.
- `self.position.size < 0`: You have a **short** position of that size (represented as a negative number).
- `self.position.size == 0`: You have **no** open position (you are flat).

Python

```
    def next(self):
        current_size = self.position.size # Get current size

        if current_size > 0:
            self.log(f"Currently LONG {current_size} units.")
            # Apply logic for long positions (e.g., check for sell sig
nal)
            if sell_condition_for_long:
                 self.order = self.close() # Or self.sell(size=current
_size)

        elif current_size < 0:
            self.log(f"Currently SHORT {abs(current_size)} units.")
            # Apply logic for short positions (e.g., check for buy-to-
cover signal)
            if cover_condition_for_short:
                 self.order = self.close() # Or self.buy(size=abs(curr
ent_size))

        else: # current_size == 0
            self.log("Currently FLAT (no position).")
            # Place entry order only if flat
```

```
            if entry_condition:
                self.order = self.buy(size=10) # Or use a sizer, or ca
lculate size
```

5.4.2. `self.position.price`

This attribute gives you the **average entry price** for your current open position. This price includes the weighted average if you added to the position multiple times.

Python

```
    def next(self):
        if self.position: # Check if a position exists
            entry_price = self.position.price
            current_price = self.data.close[0]
            size = self.position.size

            # Calculate unrealized PnL based on average entry price
            # Note: For shorts (size<0), price diff is (entry - curren
t)
            price_diff = current_price - entry_price
            unrealized_pnl = price_diff * size

            self.log(f"Pos Size: {size}, Entry Price: {entry_price:.2f
}, Cur Price: {current_price:.2f}, Unrealized PnL: {unrealized_pnl:.2f
}")

            # Example: Set a take-profit based on entry price
            if size > 0: # Long position
                 target_price = entry_price * 1.10 # 10% take profit t
arget
                 if current_price >= target_price:
                     self.log(f"TAKE PROFIT (Long) triggered at {curre
nt_price:.2f}")
                     self.order = self.close()
            elif size < 0: # Short position
                 target_price = entry_price * 0.90 # 10% take profit t
arget for short
                 if current_price <= target_price:
                     self.log(f"TAKE PROFIT (Short) triggered at {curr
ent_price:.2f}")
                     self.order = self.close()
```

Using `self.position.size` and `self.position.price` allows your strategy to make decisions based on its current market exposure and profitability.

Chapter 6: Receiving Feedback: Notifications

Placing an order with `self.buy()`, `self.sell()`, or `self.close()` is just sending an instruction. You need a way to know what actually happened: Did the order get accepted? Did it fill? At what price? Was it rejected? Did a complete trade result in a profit or loss?

`backtrader` provides this crucial feedback through two special methods you can implement in your strategy class: `notify_order()` and `notify_trade()`.

6.1. The `notify_order(order)` Method

This method is automatically called by the Backtester engine whenever there is an update to the status of an order you previously created. It receives an `order` object as an argument, which contains all the information about that specific order and its current state.

You need to define this method in your strategy class to process these updates.

Python

```
import backtrader as bt

class NotifyOrderExample(bt.Strategy):

    def __init__(self):
        self.dataclose = self.datas[0].close
        self.order = None # Variable to hold reference to pending orde
r

    def log(self, txt, dt=None):
        dt = dt or self.datas[0].datetime.date(0)
        # Use current time in log message format
        current_timestamp = self.datas[0].datetime.datetime(0).strftim
e("%Y-%m-%d %H:%M:%S") if hasattr(self.datas[0].datetime, 'datetime')
else dt.isoformat()
        print(f'{current_timestamp}, {txt}')


    def notify_order(self, order):
        # Called by backtrader upon order status changes

        if order.status == order.Submitted:
            # Order submitted to broker/simulation
            self.log(f'ORDER SUBMITTED Ref: {order.ref}')

        elif order.status == order.Accepted:
            # Order accepted by broker/simulation
            self.log(f'ORDER ACCEPTED Ref: {order.ref}')
```

```
        elif order.status == order.Completed:
            # Order has been fully executed
            if order.isbuy():
                exec_type = 'BUY'
            elif order.issell():
                exec_type = 'SELL'
            else:
                exec_type = 'CLOSE' # Could be close order completion

            self.log(
                f'ORDER COMPLETED: {exec_type}, Ref: {order.ref}, '
                f'Size: {order.executed.size}, Price: {order.executed.
price:.2f}, '
                f'Value: {order.executed.value:.2f}, Comm: {order.exec
uted.comm:.2f}'
            )
            # Store execution price if needed
            # self.last_execution_price = order.executed.price

            # Reset pending order reference AFTER completion
            self.order = None

        elif order.status == order.Partial:
            # Order partially executed
             self.log(f'ORDER PARTIAL: Ref: {order.ref}, Executed Size
: {order.executed.size}')

        elif order.status == order.Rejected:
             self.log(f'ORDER REJECTED: Ref: {order.ref}')
             self.order = None # Reset pending order reference

        elif order.status == order.Margin:
             self.log(f'ORDER MARGIN: Ref: {order.ref} - Not enough ca
sh/margin?')
             self.order = None # Reset pending order reference

        elif order.status == order.Cancelled or order.status == order.
Canceled: # Handle both spellings
             self.log(f'ORDER CANCELED: Ref: {order.ref}')
             self.order = None # Reset pending order reference

        elif order.status == order.Expired:
             self.log(f'ORDER EXPIRED: Ref: {order.ref}')
             self.order = None # Reset pending order reference

        # Note: Status flow might be Submitted -> Accepted -> Complete
```

```
d
        # Or Submitted -> Accepted -> Canceled etc.

    def next(self):
        # Example: Avoid placing new orders if one is already pending
        if self.order:
            self.log("Order pending, skipping bar.")
            return

        # Simple example: Buy on first bar if not in position
        if not self.position and len(self) == 1:
             self.log('Creating Market Buy Order')
             self.order = self.buy(size=1) # Store the order reference

        # Simple example: Close on bar 10 if in position
        elif self.position and len(self) == 10:
             self.log('Creating Market Close Order')
             self.order = self.close() # Store the order reference
```

6.1.1. Tracking Order Status (`order.status`, `Order.*` constants)

The core of `notify_order` involves checking the `order.status` attribute. Common statuses include:

- `Submitted`: Your order request has been received by the simulation engine.
- `Accepted`: The simulated broker has acknowledged the order, and it's now "working" in the market (e.g., a limit order waiting for price, or a market order waiting for the next bar).
- `Partial`: Only part of your order size has been filled (more relevant for advanced simulations or live trading).
- `Completed`: Your order has been fully filled. **This is a key status to check for confirming trade execution.**
- `Canceled` / `Cancelled`: Your cancellation request (`self.cancel(order)`) was successful.
- `Expired`: The order was not filled within its validity period (e.g., a Day order, GTD order).
- `Margin`: The order could not be accepted or was canceled because it would violate margin requirements (e.g., insufficient cash).
- `Rejected`: The broker simulation rejected the order for another reason (e.g., invalid size, price).

It's crucial to handle the "final" states (`Completed`, `Canceled`, `Expired`, `Margin`, `Rejected`) to know that the order is no longer active and to reset any internal tracking variables your strategy uses (like `self.order` in the example, preventing placement of new orders while one is pending).

6.1.2. Handling Rejections, Margin Calls, Expirations

`notify_order` is your strategy's only way of knowing if an order failed. If you receive a `Margin` or `Rejected` status, your intended trade did not happen. Similarly, an `Expired` status means a non-Market order didn't fill in time. Your strategy logic might need to log these events or potentially adjust its state based on such failures. For instance, after a rejection, you might clear the `self.order` flag so the strategy can attempt a new order on the next bar if conditions still hold.

6.1.3. Accessing Execution Details (`order.executed`)

When an order status is `Completed` (or `Partial`), the `order.executed` attribute becomes populated with details about the actual execution(s):

- `order.executed.price`: The price at which the order was filled. For partial fills, this might be the average price so far.
- `order.executed.size`: The quantity that was filled in the last execution event.
- `order.executed.value`: The total monetary value of the last fill (`size * price`).
- `order.executed.comm`: The commission charged for this fill.
- `order.executed.dt`: The timestamp of the execution (useful for detailed logs).

You typically access these attributes inside the `if order.status == order.Completed:` block to log the execution details, store the entry/exit price (as shown in the example), or perform other actions related to the successful trade execution.

6.2. The `notify_trade(trade)` Method

While `notify_order` deals with individual order events, `notify_trade` operates at the level of a complete **trade** – typically a round trip involving an entry and a subsequent exit for a given asset.

This method is called by the engine twice per trade:

9. When the trade is **opened** (i.e., the entry order is completed).
10. When the trade is **closed** (i.e., the exit order is completed).

It receives a `trade` object containing aggregated information about the entire trade lifecycle.

Python

```
import backtrader as bt

class NotifyTradeExample(bt.Strategy):
    # (Requires __init__, log, notify_order, next methods as well)

    def notify_trade(self, trade):
        # Called by backtrader when a trade is opened or closed
```

```
        if not trade.isclosed:
            # If isopen is True, the trade just opened.
            # If isopen is False but isclosed is False, it's an update
            # (e.g., adding to a position), but we usually care more
            # about the final closing event. We can just return here.
            return

        # If we reach here, trade.isclosed is True

        pnl_gross = trade.pnl # Profit/Loss without commission
        pnl_net = trade.pnlcomm # Profit/Loss including commission

        self.log(f'TRADE CLOSED: Ref: {trade.ref}, Symbol: {trade.data
._name}, '
                 f'Gross PnL: {pnl_gross:.2f}, Net PnL: {pnl_net:.2f},
'
                 f'Bars Held: {trade.barlen}')

        # You could add logic here to track overall performance,
        # calculate win rate, etc., across multiple trades.

    # --- Need __init__, log, notify_order, next for a runnable strate
gy ---
    def __init__(self): self.order = None
    def log(self, txt, dt=None):
        dt = dt or self.datas[0].datetime.date(0)
        current_timestamp = self.datas[0].datetime.datetime(0).strftim
e("%Y-%m-%d %H:%M:%S") if hasattr(self.datas[0].datetime, 'datetime')
else dt.isoformat()
        print(f'{current_timestamp}, {txt}')
    def notify_order(self, order):
        if order.status in [order.Completed, order.Canceled, order.Mar
gin, order.Rejected, order.Expired]: self.order=None
    def next(self):
        if self.order: return
        # Simple alternating buy/close logic for demonstration
        if not self.position:
             if len(self) % 5 == 1: # Buy every 5 bars if flat
                 self.order = self.buy()
        elif self.position:
             if len(self) % 5 == 4: # Close 3 bars after entry
                 self.order = self.close()
```

6.2.1. Tracking Trade Lifecycle (`trade.isopen`, `trade.isclosed`)

The `trade` object has boolean attributes to indicate its state when `notify_trade` is called:

- `trade.isopen`: True if the notification is for the opening of the trade. `notify_trade` is called once when the entry order completes.
- `trade.isclosed`: True if the notification is for the closing of the trade. `notify_trade` is called *again* when the exit order completes.

Most often, you are interested in the `if trade.isclosed:` block (or checking `if not trade.isclosed: return` at the start) to analyze the outcome of the completed trade.

6.2.2. Accessing Trade Profit/Loss (`trade.pnl`, `trade.pnlcomm`)

When `trade.isclosed` is `True`, you can access the profitability of the completed trade:

- `trade.pnl`: The **Gross Profit or Loss** for the trade (calculated as `(exit price - entry price) * size`, adjusted for shorts). This *does not* include commission costs.
- `trade.pnlcomm`: The **Net Profit or Loss** for the trade. This *includes* the impact of commissions paid on both the entry and exit orders. This is usually the more relevant figure for performance analysis.

Other useful attributes when a trade is closed include:

- `trade.ref`: A unique reference for the trade.
- `trade.data`: The data feed the trade was executed on (useful if using multiple data feeds). You can get the name via `trade.data._name`.
- `trade.size`: The size of the position that was closed.
- `trade.price`: The average entry price of the position.
- `trade.value`: The initial value of the position.
- `trade.commission`: Total commission for the round-trip trade.
- `trade.baropen`: The bar number (index) when the trade was opened.
- `trade.barclose`: The bar number when the trade was closed.
- `trade.barlen`: The duration of the trade in bars (`barclose - baropen`).

Implementing `notify_order` and `notify_trade` is essential for understanding how your strategy's instructions translate into actual simulated market actions and their outcomes. They are invaluable for debugging and verifying your strategy's execution flow.

Chapter 7: Advanced Strategy Features

Beyond the core mechanics of data handling, indicators, and basic orders, `backtrader` (and thus Backtester) offers several advanced capabilities to build more sophisticated strategies and analysis.

7.1. Managing Cash and Portfolio Value (`self.broker.get_cash()`, `self.broker.get_value()`)

Your strategy can dynamically access the current state of your simulated brokerage account through the `self.broker` object. Two key methods are:

- **`self.broker.get_cash()`**: Returns the amount of cash currently available in the account. This is useful for dynamic position sizing calculations (e.g., risking a percentage of available cash) or simply checking if enough funds exist before placing a trade.
- **`self.broker.get_value()`**: Returns the total current value of the portfolio, which includes the available cash *plus* the market value of any currently held positions. This represents your total equity and is often used as the base for percentage-based risk management or performance tracking.

Python

```
    def next(self):
        # Get current cash and portfolio value
        cash = self.broker.get_cash()
        portfolio_value = self.broker.get_value()

        self.log(f'Cash: {cash:.2f}, Portfolio Value: {portfolio_value
:.2f}')

        # Example: Only trade if portfolio value > initial capital (de
fined in params)
        # if hasattr(self.p, 'initial_capital') and portfolio_value <
self.p.initial_capital:
        #     # Consider logging this event
        #     # self.log("Portfolio value below initial capital, halti
ng trading.")
        #     return # Stop trading if below initial capital

        # Example: Use cash for sizing (Simplified)
        if not self.position and some_entry_condition: # Ensure some_e
ntry_condition is defined
            price = self.data.close[0]
            if price > 0: # Avoid division by zero
                # Simple sizing: invest 10% of available cash
                size_to_buy = (cash * 0.10) / price
```

```
                # Ensure calculated size is positive before logging/b
uying
                if size_to_buy > 0:
                   self.log(f'Attempting to buy {size_to_buy:.4f} uni
ts based on cash.')
                   self.buy(size=size_to_buy) # self.order assignment
might be needed
                else:
                   self.log(f'Calculated size is zero or negative.')
            else:
               self.log(f'Price is zero or negative, cannot calculate
size.')

    # Assume log method and some_entry_condition exist
    def log(self, txt, dt=None):
        dt = dt or self.datas[0].datetime.date(0)
        current_timestamp = self.datas[0].datetime.datetime(0).strftim
e("%Y-%m-%d %H:%M:%S") if hasattr(self.datas[0].datetime, 'datetime')
else dt.isoformat()
        print(f'{current_timestamp}, {txt}')
    # Define placeholder for condition if needed for testing
    # def __init__(self): global some_entry_condition; some_entry_cond
ition = True
```

7.2. Using Strategy Parameters (`params`) for Optimization (`OptStrategy`)

As covered in Chapter 1 (Section 1.2.2), defining strategy parameters using the `params` class attribute is crucial for flexibility. A major benefit is enabling **strategy optimization**.

Optimization involves running your strategy automatically hundreds or thousands of times, systematically varying the values of your defined `params` (e.g., trying SMA periods from 10 to 100). The Backtester engine (likely through its UI) would manage this process, running each parameter combination and recording performance metrics (like final portfolio value, Sharpe ratio, drawdown) to help you identify which parameter settings worked best historically.

- **Your Strategy's Role:** Your strategy code generally doesn't need major changes for basic optimization, *as long as* the values you want to test are defined in the `params` tuple/dict and used within your logic (e.g., `self.p.sma_period`).
- **Optimization Setup:** Defining the parameter ranges (e.g., `sma_period` from 10 to 100, step 5) and launching the optimization run happens *outside* the strategy code, typically configured via the main Backtester application interface.
- **`OptStrategy`:** For advanced optimization scenarios where you might want the strategy itself to behave differently *during* an optimization run (e.g., collecting specific data across runs), `backtrader` provides `bt.OptStrategy`. You can inherit

from this instead of `bt.Strategy`. However, for standard parameter tuning, using `bt.Strategy` with well-defined `params` is usually sufficient.

7.3. Time-Based Operations and Timers (`add_timer`)

Sometimes, you need actions triggered by the passage of time, not just by price bar events. Examples include: monthly rebalancing, exiting a trade N minutes after entry, or checking external factors periodically. `backtrader` allows this using timers.

You typically set up a timer using `self.add_timer()` within `__init__` or `next`. When the timer condition is met, `backtrader` calls a specific method in your strategy: `notify_timer()`.

Python

```
import backtrader as bt
import datetime

class TimerStrategyDemo(bt.Strategy):

    def __init__(self):
        self.log("Initializing Timer Demo")
        # Example 1: Trigger approximately at the end of each month
        self.add_timer(
            when=bt.timer.SESSION_END, # Trigger at session end relati
ve to data timeframe
            monthdays=[25], # Try around the 25th calendar day
            monthcarry=True, # If 25th is holiday/weekend, trigger on
next session_end
            tzdata=self.data0.p.tz # Use timezone from data feed if av
ailable
        )

        # Example 2: Trigger 10 sessions (days for daily data) after s
trategy starts
        # self.add_timer(when=bt.timer.SESSION_START, offset=datetime.
timedelta(days=10))

        # Example 3: Trigger at a specific time (for intraday data)
        # Ensure tzdata is provided if using specific time
        # try:
        #     tz = self.data0.p.tz
        #     target_time = datetime.time(15, 30) # 3:30 PM
        #     self.add_timer(when=target_time, tzdata=tz)
        # except AttributeError:
        #     self.log("Timezone info not available on data feed for t
imed timer.")
```

```
    def notify_timer(self, timer, when, *args, **kwargs):
        # This method is called when a timer created by add_timer trig
gers
        self.log(f'TIMER TRIGGERED: Timer Ref: {timer}, Datetime: {whe
n.isoformat()}')

        # Example: Log portfolio value on timer trigger
        portfolio_value = self.broker.get_value()
        self.log(f'Periodic Check - Portfolio Value: {portfolio_value:
.2f}')

        # You could perform rebalancing or other time-based logic here

    def log(self, txt, dt=None):
        # Use datetime for timer precision if available, else date
        dt = dt or self.datas[0].datetime.datetime(0) if hasattr(self.
datas[0].datetime, 'datetime') else self.datas[0].datetime.date(0)
        print(f'{dt.isoformat()}, {txt}')

    def next(self):
        # Regular bar processing continues as normal
        pass
```

- Key `add_timer` parameters: `when` (trigger condition - can be `bt.timer.SESSION_START`, `SESSION_END`, specific `datetime.time`, `datetime.date`, or other conditions), `offset` (timedelta delay), `repeat` (timedelta interval for recurring timers), `tzdata` (timezone for time-based triggers, often obtained from the data feed like `self.data.p.tz`).
- The `notify_timer` method receives the `timer` object itself and the exact `when` datetime the timer triggered.

7.4. Accessing Analyzer Results within `stop()`

Analyzers are powerful `backtrader` tools used to calculate overall performance metrics and statistics for a backtest run. They are added to the `Cerebro` engine, typically outside the strategy code (likely via the Backtester app UI).

Common Analyzers include:

- `bt.analyzers.SharpeRatio`: Calculates the Sharpe Ratio (and other related metrics).
- `bt.analyzers.DrawDown`: Calculates maximum drawdown statistics.
- `bt.analyzers.TradeAnalyzer`: Provides detailed statistics about individual trades (win rate, profit factor, lengths, etc.).

- `bt.analyzers.SQN`: Calculates Van Tharp's System Quality Number.

While the main results are usually displayed by the Backtester app after a run, your strategy can access the results calculated by these analyzers within its `stop()` method (which runs once at the very end of the backtest).

Python

```
# --- Assumed setup outside the strategy ---
# cerebro = bt.Cerebro()
# cerebro.addstrategy(AnalyzerAccessDemo)
# ... add data ...
# cerebro.addanalyzer(bt.analyzers.SharpeRatio, _name='mysharpe')
# cerebro.addanalyzer(bt.analyzers.TradeAnalyzer, _name='mytrades')
# results = cerebro.run()

# --- Inside your strategy class ---
import backtrader as bt

class AnalyzerAccessDemo(bt.Strategy):
    # ... __init__, next, log, notify_order etc ...

    def stop(self):
        print('--- Strategy Stopped ---') # Use print in stop, as log
might rely on data datetime

        # Access analyzers by the name given during addanalyzer ('_nam
e')
        # Use self.analyzers.getbyname to safely handle missing analyz
ers
        sharpe_analyzer = self.analyzers.getbyname('mysharpe')
        trade_analyzer = self.analyzers.getbyname('mytrades')

        if sharpe_analyzer:
            # get_analysis() returns a dictionary-like object
            analysis_sharpe = sharpe_analyzer.get_analysis()
            sharpe_ratio = analysis_sharpe.get('sharperatio', 'N/A') #
Use .get for safety
            print(f'Analyzer Result - Sharpe Ratio: {sharpe_ratio}')
        else:
             print('SharpeRatio Analyzer not found.')

        if trade_analyzer:
            analysis_trades = trade_analyzer.get_analysis()
            # Print some results from TradeAnalyzer (it returns a comp
lex dictionary)
            if analysis_trades and analysis_trades.total and analysis_
```

```
trades.total.closed > 0:
                win_total = analysis_trades.won.total
                loss_total = analysis_trades.lost.total
                total_closed = analysis_trades.total.closed
                win_rate = (win_total / total_closed) * 100 if total_
closed else 0
                pnl_net = analysis_trades.pnl.net.total
                avg_win = analysis_trades.won.pnl.average if win_tota
l > 0 else 0
                avg_loss = analysis_trades.lost.pnl.average if loss_t
otal > 0 else 0
                print(f'Analyzer Result - Trades Closed: {total_close
d}')
                print(f'Analyzer Result - Win Rate: {win_rate:.2f}%')
                print(f'Analyzer Result - Net PnL: {pnl_net:.2f}')
                print(f'Analyzer Result - Avg Win: {avg_win:.2f}, Avg
Loss: {avg_loss:.2f}')
            else:
                print('Analyzer Result - Trade Analysis: No closed tr
ades found.')
        else:
             print('TradeAnalyzer not found.')

    # Minimal methods needed for the example to be conceptually comple
te
    def __init__(self): pass
    def next(self): pass
    def log(self, txt, dt=None): # Basic log for conceptual clarity
        dt = dt or self.datas[0].datetime.date(0)
        current_timestamp = self.datas[0].datetime.datetime(0).strftim
e("%Y-%m-%d %H:%M:%S") if hasattr(self.datas[0].datetime, 'datetime')
else dt.isoformat()
        print(f'{current_timestamp}, {txt}')
```

Accessing analyzers in `stop()` allows for custom logging or final calculations based on the overall backtest performance metrics generated by `backtrader`.

7.5. Handling Multiple Timeframes (Resampling)

Strategies can benefit from analyzing data on multiple timeframes simultaneously (e.g., using a daily trend filter for an hourly entry signal). `backtrader` handles this via **resampling**.

The setup typically involves:

11. Adding the primary (lower timeframe) data feed to `Cerebro`.

12. Telling `Cerebro` to create a *new*, resampled data feed at the higher timeframe using `cerebro.resampledata(datafeed, timeframe=bt.TimeFrame.Days, compression=1)`. This new feed is added to `self.datas`.

Inside the strategy:

- `self.data0` (or `self.data`): Refers to the original, lower timeframe data (e.g., hourly).
- `self.data1`: Refers to the resampled, higher timeframe data (e.g., daily).
- `next()` is called according to the lower timeframe (`self.data0`).
- When accessing the higher timeframe data (`self.data1.close[0]`), it provides the value from the *most recently completed* higher timeframe bar.

Python

```
import backtrader as bt

class MultiTimeFrameDemoStrategy(bt.Strategy):
    params = (('ema_period_daily', 50),)

    def __init__(self):
        # self.data0 is the primary (e.g., hourly) data feed
        self.primary_close = self.data0.close
        self.log(f'Primary data feed timeframe: {self.data0.p.name} -
{self.data0.p.timeframe._name}')


        # Check if the second data feed (resampled daily) exists
        if len(self.datas) > 1:
            self.daily_close = self.data1.close
            self.log(f'Secondary data feed timeframe: {self.data1.p.na
me} - {self.data1.p.timeframe._name}')

            # Calculate indicator on the daily data (self.data1)
            self.daily_ema = bt.indicators.ExponentialMovingAverage(
                self.data1.close, # Use the resampled data feed
                period=self.p.ema_period_daily
            )
        else:
            self.log("Daily resampled data feed (data1) not found!")
            self.daily_close = None
            self.daily_ema = None # Cannot calculate without daily dat
a

    def log(self, txt, dt=None):
        # Use primary (e.g., hourly) data's datetime for logging times
tamp
```

```
        dt = dt or self.datas[0].datetime.datetime(0) if hasattr(self.
datas[0].datetime, 'datetime') else self.datas[0].datetime.date(0)
        print(f'{dt.isoformat()}, {txt}')

    def next(self):
        # This runs on every primary (e.g., hourly) bar

        # Get current primary timeframe value
        current_primary_close = self.primary_close[0]

        # Proceed only if daily data and indicators are available
        if self.daily_close is None or self.daily_ema is None:
            self.log("Waiting for daily data/indicator setup...")
            return

        # Get the latest available daily close and daily EMA values
        # Note: These values only update when a daily bar completes
        current_daily_close = self.daily_close[0]
        current_daily_ema = self.daily_ema[0]

        # Check if daily EMA is ready (not NaN)
        if bt.numsupport.isnan(current_daily_ema):
            self.log("Daily EMA is still NaN.")
            return

        # Example Logic: Buy hourly dips only if daily close is above
daily EMA
        is_daily_uptrend = current_daily_close > current_daily_ema

        # Example hourly entry condition (replace with actual logic)
        hourly_buy_signal = self.primary_close[0] < self.primary_close
[-1] and self.primary_close[-1] < self.primary_close[-2] # Example: pr
ice fell 2 bars

        if not self.position:
            if is_daily_uptrend and hourly_buy_signal:
                 self.log(f'Daily Trend UP (Close {current_daily_close
:.2f} > EMA {current_daily_ema:.2f}). Hourly Buy Signal at {current_pr
imary_close:.2f}')
                 # self.buy() # Consider order placement
            elif not is_daily_uptrend:
                 self.log(f'Daily Trend DOWN - Hourly signals ignored.
Daily Close {current_daily_close:.2f} <= EMA {current_daily_ema:.2f}')
        # else: handle position exit logic
```

7.6. Cheat-on-Close / Cheat-on-Open Execution

These are settings applied to the `Cerebro` engine, typically configured in the Backtester app's settings, not within the strategy code itself. They alter the default order execution timing:

- **`cheat_on_open=True`**: Allows an order placed during bar `N`'s `next()` call to potentially execute at the *open* of that same bar `N`. This assumes you could know the open price and act instantaneously. **Use with extreme caution**, as it can easily introduce look-ahead bias if your strategy logic implicitly uses information from bar `N` that wouldn't be known at the open.
- **`cheat_on_close=True`**: Allows an order placed during bar `N`'s `next()` call to potentially execute at the *close* of that same bar `N`. This is useful for genuinely simulating Market-on-Close (MOC) orders or strategies designed to trade exactly at the closing price.

While not set *in* the strategy, understanding whether these settings are active in your Backtester run is important, as they change the fundamental assumption about when trades occur relative to signals. If your strategy relies on MOC execution, ensure cheat-on-close is enabled in the Backtester settings.

7.7. Running Multiple Strategies Concurrently

`backtrader` allows running multiple, independent strategies simultaneously within a single backtest run using the same data and broker simulation. This is configured by adding multiple strategy classes to `Cerebro` (e.g., `cerebro.addstrategy(StrategyA); cerebro.addstrategy(StrategyB)`), likely managed via the Backtester UI allowing selection/addition of multiple strategies for a single run.

- **Independence:** Each strategy instance maintains its own parameters and internal state. `StrategyA`'s `next()` call doesn't directly affect `StrategyB`'s `next()` call.
- **Shared Broker:** All strategies share the same simulated broker account. Orders from all strategies affect the single cash balance and combined positions. This allows testing portfolio effects or interactions through the shared capital pool.
- **Use Cases:** Comparing strategies side-by-side, running different strategies on different data feeds within the same backtest (if multiple feeds are loaded), or simulating a portfolio composed of multiple independent signal generators.

Your individual strategy code doesn't usually need modification to run concurrently, but its performance might be affected by the actions of other strategies depleting or increasing the shared cash. Ensure your cash management and sizing logic (if dynamic) accounts for this shared environment if running multiple strategies.

Chapter 8: Strategy Examples for Backtester

Here are complete strategy examples you can use as templates or starting points within the Backtester application.

8.1. SMA Crossover using `bt.indicators.SimpleMovingAverage`

This classic strategy uses two built-in `backtrader` Simple Moving Averages.

Python

```
import backtrader as bt

class SmaCrossStrategy(bt.Strategy):
    params = (
        ('pfast', 10),  # Period for the fast SMA
        ('pslow', 30),  # Period for the slow SMA
        ('printlog', True), # Enable/disable logging
    )

    def __init__(self):
        self.dataclose = self.datas[0].close
        self.order = None # To track pending orders

        # Instantiate the SMAs using backtrader's built-in indicator
        self.sma_fast = bt.indicators.SimpleMovingAverage(
            self.datas[0], period=self.p.pfast)
        self.sma_slow = bt.indicators.SimpleMovingAverage(
            self.datas[0], period=self.p.pslow)

        # Use CrossOver indicator for signal detection
        self.crossover = bt.indicators.CrossOver(self.sma_fast, self.s
ma_slow)

    def log(self, txt, dt=None, doprint=False):
        ''' Logging function for this strategy'''
        if self.params.printlog or doprint:
            dt = dt or self.datas[0].datetime.date(0)
            current_timestamp = self.datas[0].datetime.datetime(0).str
ftime("%Y-%m-%d %H:%M:%S") if hasattr(self.datas[0].datetime, 'datetim
e') else dt.isoformat()
            print(f'{current_timestamp}, {txt}')


    def notify_order(self, order):
        if order.status in [order.Submitted, order.Accepted]:
            return # Do nothing for pending orders
```

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

        elif order.status in [order.Canceled, order.Margin, order.Reje
cted]:
            self.log(f'Order Canceled/Margin/Rejected Ref: {order.ref}
')

        # Reset order tracking after final state
        self.order = None

    def next(self):
        # Log closing price
        self.log(f'Close: {self.dataclose[0]:.2f}, FastSMA: {self.sma_
fast[0]:.2f}, SlowSMA: {self.sma_slow[0]:.2f}')

        # Check if an order is pending
        if self.order:
            return

        # Check if we are in the market
        if not self.position:
            # Buy signal: Fast SMA crosses above Slow SMA (crossover >
0)
            if self.crossover[0] > 0:
                self.log(f'BUY CREATE, Signal Price={self.dataclose[0]
:.2f}')
                self.order = self.buy()
        else: # We are in the market
            # Sell signal: Fast SMA crosses below Slow SMA (crossover
< 0)
            if self.crossover[0] < 0:
                self.log(f'SELL CREATE (CLOSE), Signal Price={self.dat
aclose[0]:.2f}')
                self.order = self.close() # Close the existing positio
n
```

8.2. RSI Threshold using `bt.talib.RSI` (Wrapper)

This strategy uses the `backtrader` wrapper for TA-Lib's RSI.

Python

```
class RsiThresholdStrategy(bt.Strategy):
    params = (
        ('rsi_period', 14),
        ('rsi_lower', 30), # Lower threshold for buy
        ('rsi_upper', 70), # Upper threshold for sell
    )

    def __init__(self):
        self.dataclose = self.datas[0].close
        self.order = None
        self.required_lookback = self.p.rsi_period + 1 # Min data for
RSI calc
        self.rsi = bt.talib.RSI(self.dataclose, timeperiod=self.p.rsi_
period)

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

    def notify_order(self, order):
        # Basic notification logic (same as previous example)
        if order.status in [order.Submitted, order.Accepted]: return
        if order.status in [order.Completed]:
            exec_type = 'BUY' if order.isbuy() else 'SELL'
            self.log(f'{exec_type} EXECUTED, Price: {order.executed.pr
ice:.2f}')
        elif order.status in [order.Canceled, order.Margin, order.Reje
cted]:
            self.log('Order Canceled/Margin/Rejected')
        self.order = None

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

        # Access RSI values directly from the indicator line
        # Backtrader handles the minimum period. self.rsi[0] is the *c
urrent* value.
        current_rsi = self.rsi[0]
        # self.rsi[-1] is the *previous* bar's RSI value
        previous_rsi = self.rsi[-1]
```

```
        # Optional: Check if RSI values are ready (might be NaN initia
lly)
        # This check might not be strictly necessary if the logic hand
les NaN implicitly,
        # but it can prevent logs/trades on invalid data.
        try:
            # Check both current and previous are valid numbers
            if np.isnan(current_rsi) or np.isnan(previous_rsi):
                # self.log(f"RSI not ready. Current: {current_rsi}, Pr
evious: {previous_rsi}") # Optional log
                return
        except TypeError:
            # Handles cases where the values might not even be floats
yet (very early bars)
            # self.log("RSI not ready (TypeError check).") # Optional
log
            return

        # Log current close and RSI
        self.log(f'Close={self.dataclose[0]:.2f}, RSI={current_rsi:.2f
}') # Removed Prev RSI log for brevity

        # Strategy Logic (using the indicator's values)
        if not self.position: # If not in the market
            # Buy Condition: RSI crosses below lower threshold
            if current_rsi < self.p.rsi_lower and previous_rsi >= self
.p.rsi_lower:
                self.log(f'BUY CREATE (RSI < {self.p.rsi_lower}), RSI=
{current_rsi:.2f}')
                self.order = self.buy()
        else: # If in the market
             # Sell Condition: RSI crosses above upper threshold
             if current_rsi > self.p.rsi_upper and previous_rsi <= sel
f.p.rsi_upper:
                self.log(f'SELL CREATE (RSI > {self.p.rsi_upper}), RSI
={current_rsi:.2f}')
                self.order = self.close() # Close position
```

8.3. Bollinger Band Strategy using `bt.talib.BBANDS` (Wrapper)

Uses the `backtrader` wrapper for TA-Lib's Bollinger Bands.

Python

```
class BollingerBandStrategy(bt.Strategy):
    params = (
        ('bb_period', 20),
```

```
        ('bb_dev', 2.0), # Number of standard deviations
    )

    def __init__(self):
        self.dataclose = self.datas[0].close
        self.order = None

        # --- Define Bollinger Bands using Backtrader's indicator ---
        self.bbands = bt.indicators.BollingerBands(
            self.datas[0], # You can pass the data feed directly
            period=self.p.bb_period,
            devfactor=self.p.bb_dev
            # movav=bt.indicators.SimpleMovingAverage # Default is SMA
, explicitly set if needed
        )

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

    def notify_order(self, order):
        # Basic notification logic
        if order.status in [order.Submitted, order.Accepted]: return
        if order.status in [order.Completed]:
            exec_type = 'BUY' if order.isbuy() else 'SELL'
            self.log(f'{exec_type} EXECUTED, Price: {order.executed.pr
ice:.2f}')
        elif order.status in [order.Canceled, order.Margin, order.Reje
cted]:
            self.log('Order Canceled/Margin/Rejected')
        self.order = None

    def next(self):
        if self.order: return # Pending order check

        # Access band values directly from the indicator lines
        # [0] gets the current bar's value
        upper_band = self.bbands.lines.top[0]
        middle_band = self.bbands.lines.mid[0]
        lower_band = self.bbands.lines.bot[0]

        # Check if bands are calculated (use middle band as proxy)
        # Backtrader handles the minimum period internally
        if np.isnan(middle_band) or np.isnan(upper_band) or np.isnan(l
ower_band):
             # self.log("BBands not ready (NaN)...") # Optional log
```

```
            return

        # Log current close and band values
        self.log(f'Close={self.dataclose[0]:.2f}, LowerB={lower_band:.
2f}, UpperB={upper_band:.2f}')

        # Strategy Logic using indicator lines
        if not self.position:
            # Buy Condition: Price touches or crosses below lower band
            if self.dataclose[0] <= lower_band:
                self.log(f'BUY CREATE (Close <= Lower Band), Close={se
lf.dataclose[0]:.2f}, LowerB={lower_band:.2f}')
                self.order = self.buy()
        else: # In the market
            # Sell Condition: Price touches or crosses above upper ban
d (simple exit)
            # Alternative: Exit if price crosses back above middle ban
d (mean reversion target)
            # if self.dataclose[0] >= middle_band:
            #    self.log(f'SELL CREATE (Close >= Middle Band), Close=
{self.dataclose[0]:.2f}, MiddleB={middle_band:.2f}')
            #    self.order = self.close()

            if self.dataclose[0] >= upper_band:
                self.log(f'SELL CREATE (Close >= Upper Band), Close={s
elf.dataclose[0]:.2f}, UpperB={upper_band:.2f}')
                self.order = self.close() # Close position
```

8.4. MACD Strategy using `bt.talib.MACD` (Wrapper)

Uses the `backtrader` wrapper for TA-Lib's MACD.

Python

```
import backtrader as bt

class MacdWrapperStrategy(bt.Strategy):
    params = dict(
        macd_fast=12,
        macd_slow=26,
        macd_signal=9,
        printlog=True,
    )

    def __init__(self):
        self.dataclose = self.data.close
        self.order = None
```

```
        # TA-Lib MACD: outputs are .macd, .macdsignal, .macdhist
        self.macd = bt.talib.MACD(
            self.dataclose,
            fastperiod=self.p.macd_fast,
            slowperiod=self.p.macd_slow,
            signalperiod=self.p.macd_signal,
        )

        # Expose the three lines
        self.macd_line   = self.macd.macd
        self.signal_line = self.macd.macdsignal
        self.hist_line   = self.macd.macdhist

        # Detect crossovers between MACD and signal
        self.mcross = bt.indicators.CrossOver(self.macd_line, self.sig
nal_line)

    def log(self, txt, dt=None, doprint=False):
        if self.p.printlog or doprint:
            dt = dt or self.datas[0].datetime.datetime(0)
            timestamp = dt.strftime("%Y-%m-%d %H:%M:%S")
            print(f'{timestamp}, {txt}')

    def notify_order(self, order):
        # ignore submissions/acceptances
        if order.status in [order.Submitted, order.Accepted]:
            return

        # execution
        if order.status == order.Completed:
            etype = 'BUY' if order.isbuy() else 'SELL'
            self.log(f'{etype} EXECUTED, '
                     f'Price: {order.executed.price:.2f}, '
                     f'Size: {order.executed.size}')
        elif order.status in [order.Canceled, order.Margin, order.Reje
cted]:
            self.log(f'Order Canceled/Margin/Rejected Ref: {order.ref}
')

        # reset order flag
        self.order = None

    def next(self):
        # current values
        mc = self.macd_line[0]
```

```
        sg = self.signal_line[0]
        hi = self.hist_line[0]
        price = self.dataclose[0]

        self.log(f'Close={price:.2f}, MACD={mc:.2f}, '
                 f'Signal={sg:.2f}, Hist={hi:.2f}')

        # if an order is pending, skip
        if self.order:
            return

        # no position look to buy
        if not self.position:
            if self.mcross[0] > 0:
                self.log('BUY CREATE (MACD crossed above Signal)')
                self.order = self.buy()

        # in the market look to sell/close
        else:
            if self.mcross[0] < 0:
                self.log('SELL CREATE (MACD crossed below Signal)')
                self.order = self.close()
```

8.5. ATR Based Stop-Loss Implementation using `bt.talib.ATR` (Wrapper)

Combines a simple entry (SMA cross) with a stop-loss based on the Average True Range (ATR) using the `bt.talib` wrapper.

Python

```
class AtrStopLossWrapperStrategy(bt.Strategy):
    params = (
        ('sma_period', 20),   # For entry signal
        ('atr_period', 14),   # For stop loss calculation
        ('atr_multiplier', 2.0), # Multiplier for ATR stop distance
        ('printlog', True),
    )

    def __init__(self):
        self.dataclose = self.datas[0].close
        self.datahigh = self.datas[0].high
        self.datalow = self.datas[0].low
        self.order = None
        self.stop_price = None # Variable to store the calculated stop
price

        # Entry signal indicator
```

```
        self.sma = bt.indicators.SimpleMovingAverage(self.data, period
=self.p.sma_period)

        # ATR indicator using the wrapper
        self.atr = bt.talib.ATR(self.datahigh, self.datalow, self.data
close,
                                timeperiod=self.p.atr_period)

    def log(self, txt, dt=None, doprint=False):
         if self.params.printlog or doprint:
            dt = dt or self.datas[0].datetime.date(0)
            current_timestamp = self.datas[0].datetime.datetime(0).str
ftime("%Y-%m-%d %H:%M:%S") if hasattr(self.datas[0].datetime, 'datetim
e') else dt.isoformat()
            print(f'{current_timestamp}, {txt}')

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

        if order.status in [order.Completed]:
            exec_type = 'BUY' if order.isbuy() else 'SELL'
            self.log(f'{exec_type} EXECUTED, Price: {order.executed.pr
ice:.2f}, Size: {order.executed.size}')

            if order.isbuy(): # If buy order completed, calculate and
set stop price
                current_atr = self.atr[0] # Get ATR value on execution
bar
                if current_atr is not None and not np.isnan(current_at
r) and current_atr > 0:
                    self.stop_price = order.executed.price - current_a
tr * self.p.atr_multiplier
                    self.log(f'Entry ATR={current_atr:.2f}, Stop Price
set to {self.stop_price:.2f}')
                else:
                    self.log(f'Could not calculate ATR for stop loss o
n execution bar (ATR={current_atr}), stop not set.')
                    self.stop_price = None # Cannot set stop
            elif order.issell(): # If sell order completed (closing tr
ade), clear stop price
                 if self.stop_price is not None: # Log only if a stop
was active
                     self.log(f'Position closed, clearing stop price
{self.stop_price:.2f}')
                 self.stop_price = None
```

```
        elif order.status in [order.Canceled, order.Margin, order.Reje
cted]:
            self.log(f'Order Canceled/Margin/Rejected Ref: {order.ref}
')
            # If entry order failed, ensure stop price is not set
            if self.stop_price is not None and order.ref == self.order
.ref: # Check if it was the tracked order
                self.stop_price = None

        self.order = None # Reset pending order flag

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

        # Check stop loss first if in a long position
        if self.position.size > 0 and self.stop_price is not None:
            # Use low of current bar for stop check for more realistic
fill
            if self.datalow[0] <= self.stop_price:
                self.log(f'STOP LOSS HIT: Low={self.datalow[0]:.2f} <=
Stop={self.stop_price:.2f}')
                self.order = self.close() # Create close order
                return # Exit next() after placing close order

        # Entry Logic (if not in position and no stop loss triggered)
        if not self.position:
            # Simple SMA entry: Buy if close crosses above SMA
            if self.dataclose[-1] < self.sma[-1] and self.dataclose[0]
> self.sma[0]:
                self.log(f'BUY CREATE (SMA Cross), Close={self.dataclo
se[0]:.2f}')
                self.order = self.buy()
                # Stop price will be calculated in notify_order after
execution
```

8.6. Candlestick Pattern Entry using `bt.talib.CDLDOJI` (Wrapper)

Enters on a Doji pattern detection using the `bt.talib` wrapper.

Python

```
import backtrader as bt

class CandlestickWrapperStrategy(bt.Strategy):
    params = (
        ('hold_period', 5), # How many bars to hold after entry
        ('printlog', True),
```

```
    )

    def __init__(self):
        self.o = self.data.open
        self.h = self.data.high
        self.l = self.data.low
        self.c = self.data.close
        self.order = None
        self.entry_bar = None # To track when we entered

        # Instantiate the pattern detection using the wrapper
        self.doji = bt.talib.CDLDOJI(self.o, self.h, self.l, self.c)
        # Try another pattern: e.g., Hammer
        # self.hammer = bt.talib.CDLHAMMER(self.o, self.h, self.l, sel
f.c)

    def log(self, txt, dt=None, doprint=False):
         if self.params.printlog or doprint:
            dt = dt or self.datas[0].datetime.date(0)
            current_timestamp = self.datas[0].datetime.datetime(0).str
ftime("%Y-%m-%d %H:%M:%S") if hasattr(self.datas[0].datetime, 'datetim
e') else dt.isoformat()
            print(f'{current_timestamp}, {txt}')

    def notify_order(self, order):
        # Basic notification logic
        if order.status in [order.Submitted, order.Accepted]: return
        if order.status in [order.Completed]:
            exec_type = 'BUY' if order.isbuy() else 'SELL'
            self.log(f'{exec_type} EXECUTED, Price: {order.executed.pr
ice:.2f}, Size: {order.executed.size}')
            if order.isbuy():
                 self.entry_bar = len(self) # Record entry bar index
            else: # Sell executed (closed position)
                 self.entry_bar = None
        elif order.status in [order.Canceled, order.Margin, order.Reje
cted]:
            self.log(f'Order Canceled/Margin/Rejected Ref: {order.ref}
')
        self.order = None

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

        # Check pattern result from the wrapper line
        # Note: TA-Lib patterns return 0 (no), 100 (bullish), -100 (be
```

```
arish)
        # The wrapper line evaluates non-zero as True in Python boolea
n context
        is_doji_detected = self.doji[0] != 0 # Check if non-zero
        # is_hammer_detected = self.hammer[0] == 100 # Check specific
bullish value

        # Log pattern detection
        if is_doji_detected:
            self.log(f"Doji Pattern Detected on bar {len(self)}")

        # Strategy Logic
        if not self.position:
            # Buy Condition: Doji pattern detected on the current bar
            if is_doji_detected:
                 self.log(f'BUY CREATE (Doji Detected), Close={self.c[
0]:.2f}')
                 self.order = self.buy()
        else: # In position, check exit condition
             if self.entry_bar is not None and len(self) >= (self.entr
y_bar + self.p.hold_period):
                 self.log(f'SELL CREATE (Holding Period Met), Close={s
elf.c[0]:.2f}')
                 self.order = self.close()
```

8.7. Multi-Indicator Strategy (e.g., ADX + RSI using `bt.talib` Wrappers)

Combines ADX (trend strength) and RSI (momentum/overbought/oversold) using `bt.talib` wrappers.

Python

```
class AdxRsiWrapperStrategy(bt.Strategy):
    params = (
        ('adx_period', 14),
        ('adx_threshold', 25), # Minimum ADX value to consider a trend
        ('rsi_period', 14),
        ('rsi_overbought', 70), # RSI level to avoid buying / potentia
lly sell
        ('rsi_oversold', 30), # RSI level for potential buy signal (if
trend allows)
        ('printlog', True),
    )

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

```
        self.datalow = self.datas[0].low
        self.order = None

        # Instantiate indicators using wrappers
        self.adx = bt.talib.ADX(self.datahigh, self.datalow, self.data
close,
                                timeperiod=self.p.adx_period)
        self.rsi = bt.talib.RSI(self.dataclose, timeperiod=self.p.rsi_
period)

    def log(self, txt, dt=None, doprint=False):
         if self.params.printlog or doprint:
            dt = dt or self.datas[0].datetime.date(0)
            current_timestamp = self.datas[0].datetime.datetime(0).str
ftime("%Y-%m-%d %H:%M:%S") if hasattr(self.datas[0].datetime, 'datetim
e') else dt.isoformat()
            print(f'{current_timestamp}, {txt}')

    def notify_order(self, order):
        # Basic notification logic
        if order.status in [order.Submitted, order.Accepted]: return
        if order.status in [order.Completed]:
            exec_type = 'BUY' if order.isbuy() else 'SELL'
            self.log(f'{exec_type} EXECUTED, Price: {order.executed.pr
ice:.2f}, Size: {order.executed.size}')
        elif order.status in [order.Canceled, order.Margin, order.Reje
cted]:
            self.log(f'Order Canceled/Margin/Rejected Ref: {order.ref}
')
        self.order = None

    def next(self):
        # Get current indicator values
        current_adx = self.adx[0]
        current_rsi = self.rsi[0]

        # Check if indicators are valid (backtrader handles min period
for wrappers)
        # We might still get NaN if underlying data has NaN, but less
common
        if np.isnan(current_adx) or np.isnan(current_rsi):
            self.log("ADX or RSI is NaN, waiting...")
            return

        self.log(f'Close={self.dataclose[0]:.2f}, ADX={current_adx:.2f
}, RSI={current_rsi:.2f}')
```

```
        if self.order: return # Order pending

        # --- Strategy Logic ---
        is_trending = current_adx > self.p.adx_threshold

        if not self.position:
            # Buy Condition: Trend is active (ADX > threshold) AND RSI
is below overbought level
            if is_trending and current_rsi < self.p.rsi_overbought:
                # Optional: Add another condition like RSI crossing u
p from oversold
                # if self.rsi[-1] < self.p.rsi_oversold and current_r
si >= self.p.rsi_oversold:
                self.log(f'BUY CREATE (ADX Trending & RSI OK), ADX={c
urrent_adx:.2f}, RSI={current_rsi:.2f}')
                self.order = self.buy()
        else: # In position
             # Sell Condition: Trend weakens (ADX falls below threshol
d) OR RSI becomes overbought
             if not is_trending or current_rsi > self.p.rsi_overbought
:
                reason = "Trend Weakened" if not is_trending else "RS
I Overbought"
                self.log(f'SELL CREATE ({reason}), ADX={current_adx:.
2f}, RSI={current_rsi:.2f}')
                self.order = self.close()
```

These examples demonstrate how to combine different elements discussed in the manual using `backtrader`'s built-in indicators and the convenient `bt.talib` wrappers. Remember to adapt, test, and refine them for your specific trading goals in Backtester.

Appendix

A.1. `Strategy` Class Method Quick Reference

These are the primary methods you define or override within your custom strategy class (inheriting from `bt.Strategy`):

- **`__init__(self)`:**
  - **Called:** Once, when the strategy object is created before backtesting begins.
  - **Purpose:** Initialize indicators, define parameters (`params`), store data line references, set up initial state variables.
- **`start(self)`:**
  - **Called:** Once, at the very beginning of the backtest run, before any data processing.
  - **Purpose:** Perform initial setup actions that don't rely on data minimum periods. Often left unimplemented if `__init__` suffices.
- **`prenext(self)`:**
  - **Called:** For each bar *before* all indicators have met their minimum calculation periods.
  - **Purpose:** Logic that needs to run even before indicators produce valid output. Often left unimplemented.
- **`nextstart(self)`:**
  - **Called:** Exactly once, on the first bar where all indicators have met their minimum periods.
  - **Purpose:** Execute one-time logic precisely when indicators become valid. Default behavior is to call `next()`.
- **`next(self)`:**
  - **Called:** For every bar *after* the minimum period has been met.
  - **Purpose:** The main engine of the strategy. Contains the core logic for checking conditions, analyzing indicator values, and generating trade orders (`buy`/`sell`/`close`).
- **`stop(self)`:**
  - **Called:** Once, after the last bar has been processed by `next()`.
  - **Purpose:** Perform final calculations, cleanup, access Analyzer results (`self.analyzers`).
- **`log(self, txt, dt=None, ...)`:**
  - **Called:** By *you* from within other strategy methods (e.g., `next`, `notify_order`).
  - **Purpose:** Standardized way to print output and log events during the backtest. Implementation is user-defined.

- **`notify_order(self, order)`:**
  - **Called:** By `backtrader` whenever an order's status changes.
  - **Purpose:** Track order lifecycle (Submitted, Accepted, Completed, Rejected, etc.), access execution details (`order.executed`), manage pending order state.
- **`notify_trade(self, trade)`:**
  - **Called:** By `backtrader` when a trade is opened and again when it is closed.
  - **Purpose:** Track completed trades, access Profit/Loss (`trade.pnl`, `trade.pnlcomm`), log trade statistics.
- **`notify_timer(self, timer, when, *args, **kwargs)`:**
  - **Called:** By `backtrader` when a timer created with `add_timer` triggers.
  - **Purpose:** Execute time-based logic (e.g., periodic rebalancing).

Key methods called *from* your strategy:

- **`self.buy(...)`:** Creates a buy order.
- **`self.sell(...)`:** Creates a sell order.
- **`self.close(...)`:** Creates an order to close the current position on the primary data feed.
- **`self.cancel(order)`:** Requests cancellation of a specific pending order.
- **`self.add_timer(...)`:** Creates a timer to trigger `notify_timer` based on time conditions.
- **`self.broker.get_cash()`:** Gets the current cash balance.
- **`self.broker.get_value()`:** Gets the current total portfolio value.

A.2. `Order` Status Reference

The `order.status` attribute within `notify_order(order)` indicates the current state of an order. Common statuses (accessible via `bt.Order.StatusName` or `order.StatusName`):

- **`Created`**: Order instance created internally (rarely seen by user).
- **`Submitted`**: Order submitted to the broker simulation.
- **`Accepted`**: Order accepted by the broker simulation and is now working (e.g., waiting for fill).
- **`Partial`**: Order has been partially filled. `order.executed` holds details of the last partial fill.
- **`Completed`**: Order has been fully filled. `order.executed` holds details of the final fill.
- **`Canceled`** (or **`Cancelled`**): Order was successfully canceled by a `self.cancel()` request.

- **`Expired`**: Order expired based on its `valid` parameter (e.g., Day order, GTD) before being filled.
- **`Margin`**: Order rejected or canceled due to insufficient funds or margin requirements.
- **`Rejected`**: Order rejected by the broker simulation for other reasons (e.g., invalid parameters, size).

A.3. `Trade` Attributes Reference

The `trade` object passed to `notify_trade(trade)` contains information about a position lifecycle (entry to exit). Key attributes, especially relevant when `trade.isclosed` is `True`:

- **`ref`**: Unique reference ID for the trade.
- **`data`**: The data feed associated with the trade (useful for multi-data strategies). Use `trade.data._name` for the data name string.
- **`size`**: The size of the position that was opened/closed.
- **`price`**: The average entry price of the trade.
- **`value`**: The initial absolute monetary value of the position at entry (`price * size`).
- **`commission`**: Total commission paid for the round-trip trade (entry + exit).
- **`pnl`**: Gross Profit or Loss for the trade (excludes commission).
- **`pnlcomm`**: Net Profit or Loss for the trade (includes commission). This is usually the most important P&L figure.
- **`baropen`**: Index of the bar on which the trade was opened.
- **`barclose`**: Index of the bar on which the trade was closed.
- **`barlen`**: Duration of the trade in bars (`barclose - baropen`).
- **`dtopen`**: Datetime the trade was opened.
- **`dtclose`**: Datetime the trade was closed.
- **`isopen`**: Boolean, `True` if the notification is for the trade opening.
- **`isclosed`**: Boolean, `True` if the notification is for the trade closing.
- **`justopened`**: Boolean, `True` only on the very first notification for the trade (when it opens).