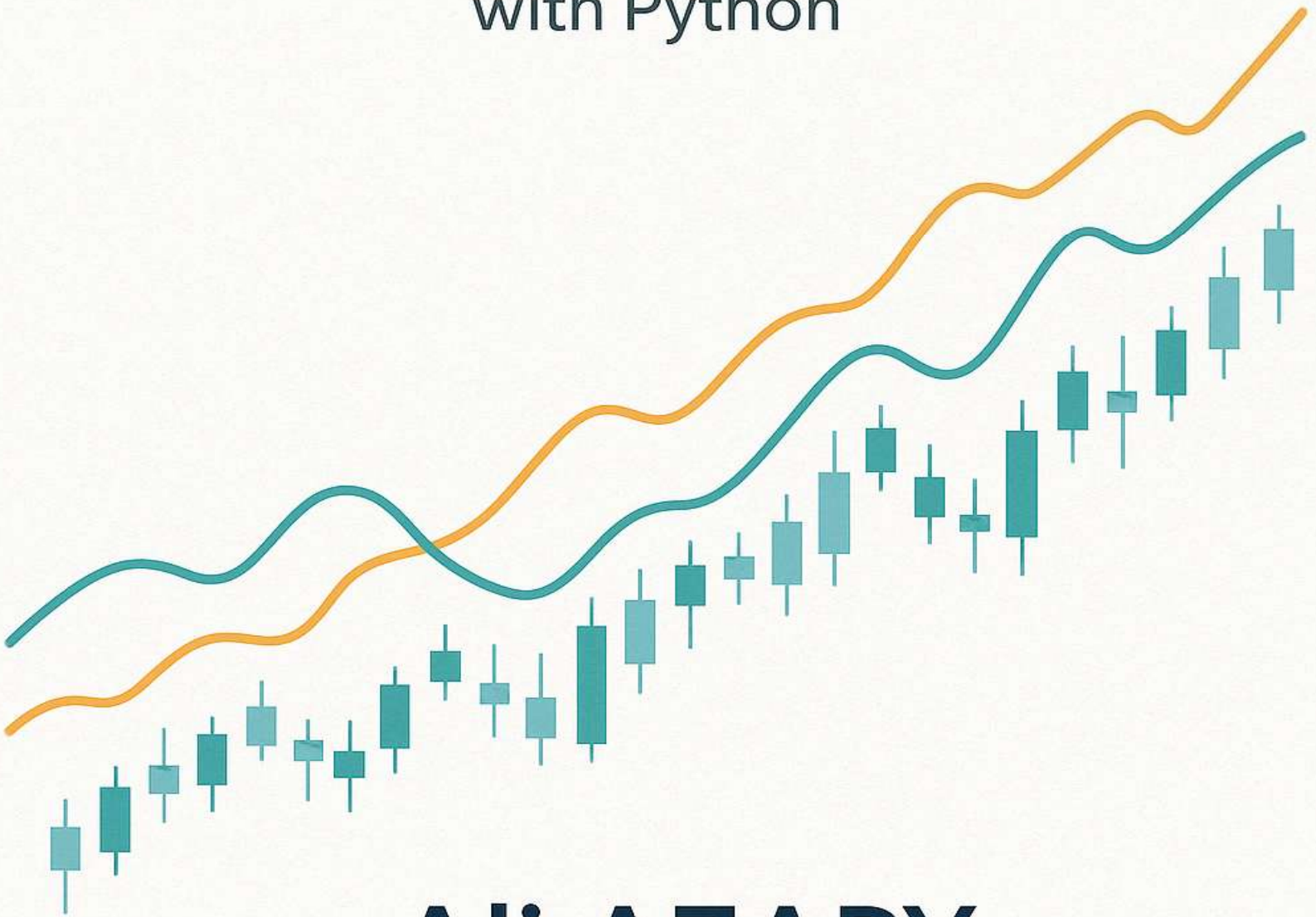
Backtrader Essentials

Building Successful Strategies with Python



Ali AZARY

BACKTRADER ESSENTIALS: BUILDING SUCCESSFUL STRATEGIES WITH PYTHON

Written by Ali AZARY

First edition, 2025

10 9 8 7 6 5 4 3 2 1

ISBN 979-8230304517

Published by Ali AZARY

www.aliazary.com

Contents

Preface

Technical analysis. For generations, traders and investors have peered at charts, drawing lines, identifying patterns, and calculating indicators, all in an attempt to decipher the market's next move. From Moving Averages and RSI to MACD and Bollinger Bands, these mathematical tools transform raw price and volume data into potentially actionable insights about trends, momentum, and volatility.

But manual analysis, while valuable for developing intuition, has its limits. Keeping track of multiple indicators across various assets and timeframes in real-time is demanding. Drawing trendlines can be subjective. More importantly, executing trades based on fleeting chart signals consistently, without emotional interference, is a significant challenge for even seasoned traders. How do you *know* if that SMA crossover strategy truly worked over the last five years, not just the last five days? How do you rigorously test the impact of adding an RSI filter?

This is where the power of automation comes in, and where `backtrader`, a flexible and powerful open-source Python framework, truly shines.

Why Automate Technical Indicators?

Automating the calculation, analysis, and even the execution of trading signals based on technical indicators offers compelling advantages:

Speed and Efficiency: Backtest decades of historical data in minutes, not months. Evaluate strategy variations rapidly.

Objectivity and Consistency: Rules are coded and executed precisely the same way, every single time, eliminating emotional decision-making (fear, greed) and subjective interpretation at the point of execution.3

Rigorous Testing: Quantify the historical performance of a trading idea. Did it *actually* work? Under what conditions? What were the risks? Backtesting provides data-driven answers.

Broader Scope: Simultaneously monitor and analyze numerous indicators across multiple financial instruments and timeframes – a task impossible to perform manually.

Discovery: Systematic testing can reveal subtle interactions between indicators or optimal parameter settings that are not intuitively obvious.

Foundation for Automation: A thoroughly backtested strategy is the essential first step towards building a potentially automated trading system.

`backtrader` allows you to take the standard technical indicators you might use on a charting platform, implement them with code, combine them into logical trading rules, and test their performance systematically. It bridges the gap between trading ideas and quantifiable results.

Who This Guide Is For

This guide is designed as a concise, practical introduction to using `backtrader`, with a specific focus on leveraging technical indicators for signal generation. It's aimed at two primary groups:

Python Users: If you have a grasp of Python basics (variables, loops, functions, classes) and are interested in applying your programming skills to the financial markets, quantitative analysis, or algorithmic trading, this guide is for you. You don't necessarily need deep financial expertise to start; `backtrader` provides the tools, and this guide provides the context to build indicator-based strategies.

Traders and Market Analysts: If you are familiar with technical analysis concepts, charting, and perhaps some discretionary trading, but want to move towards more systematic, evidence-based methods, this guide will show you how. You'll learn to translate your indicator knowledge into Python code, test your strategies rigorously, and remove the guesswork and emotional stress from rule-based trading.

Whether you're a student exploring quantitative finance, a developer curious about trading applications, or a trader seeking automation, this guide aims to get you up and running quickly with the core functionalities of `backtrader needed for indicator-driven strategies.

Setup: Your Development Environment

Before diving in, you'll need a working Python environment and a few key libraries. We assume you have Python installed (ideally version 3.6 or newer). You can check your installation by opening a terminal or command prompt and typing `python --version` or `python3 --version`. If you don't have Python, download it from python.org.

With Python ready, install the necessary libraries using `pip`, Python's package installer. Open your terminal or command prompt and run the following commands:

Install Backtrader: This is the core backtesting framework.

Bash

```
pip install backtrader
```

Install Matplotlib: Used by `backtrader` for plotting results (charts, indicators, trades).

Bash

```
pip install matplotlib
```

Install yfinance (Optional but Recommended): While not strictly required for all examples (some use built-in or local data), `yfinance` is a very convenient library for downloading historical market data from Yahoo Finance, perfect for getting started quickly with different assets.

Bash

```
pip install yfinance
```

With these libraries installed, you have the essential toolkit ready.

This guide will walk you through loading data, defining strategies, implementing built-in and custom technical indicators, combining their signals, placing simulated orders, and analyzing the performance of your backtests. Let's begin exploring the world of systematic trading with Python and `backtrader`.

Chapter 1: Backtrader Basics & Data Feeds

Welcome to the practical core of our journey! In this chapter, we'll move from concept to code. We'll set up our first `backtrader` script, focusing on the essential scaffolding: importing necessary tools, fetching historical market data, and introducing the main engine that powers `backtrader` simulations – Cerebro. By the end of this chapter, you'll have a working script that loads data and runs a (very simple) backtest, laying the groundwork for the indicator-based strategies we'll build later.

Setting the Stage: Imports

Every Python script starts by importing the libraries it needs. For our `backtrader` work, especially in these initial stages, we'll typically need the following:

- **`backtrader`:** The core library itself, usually imported with the alias `bt` for brevity.
- **`datetime`:** Python's built-in library for handling dates and times, often needed for specifying date ranges.
- **`pandas`:** A fundamental library for data manipulation in Python. While `backtrader` can work with various data formats, we'll often use Pandas DataFrames as an intermediary, especially when fetching data from external sources.
- **`yfinance`:** A popular library for downloading historical market data from Yahoo Finance. It's a convenient way to get data for testing purposes.
- **`matplotlib`:** Although we often don't import it directly into our script using `import matplotlib`, `backtrader` uses it behind the scenes for plotting. Ensuring it's installed (as covered in the Preface) is crucial for visualization.

Let's start our script with the standard imports:

```
# -*- coding: utf-8 -*-
# chapter1_basics.py
!pip install backtrader yfinance pandas matplotlib
# Import necessary libraries
from __future__ import (absolute_import, division, print_function,
                        unicode_literals)

import backtrader as bt
import datetime
import pandas as pd
import yfinance as yf # Import yfinance

print("Libraries Imported Successfully!")
```

The first line (`# -*- coding: utf-8 -*-`) specifies the file encoding, which is good practice. The `__future__` imports ensure compatibility between Python 2 and 3, which is standard practice in many `backtrader` examples, although less critical if you are exclusively using Python 3. The main imports bring in `backtrader` as `bt`, `datetime`, `pandas` as `pd`, and `yfinance` as `yf`.

Acquiring Historical Market Data

A backtest needs historical data to simulate trading. This data typically includes the Open, High, Low, and Close prices (OHLC), along with the trading Volume for each period (e.g., daily, hourly). Sometimes, Open Interest is also included, especially for futures contracts.

For this guide, we'll primarily use `yfinance` to download data. It's simple and provides easy access to a vast range of assets. Let's download daily data for Apple (ticker: AAPL) for a specific period.

```
# Define the ticker symbol and date range
ticker = 'AAPL'
start_date = '2020-01-01'
end_date = '2023-12-31' # Use a date in the past

print(f"Downloading {ticker} data from {start_date} to {end_date}...")

# Download data using yfinance
try:
    # Use yf.download for simplicity
    dataframe = yf.download(ticker, start=start_date, end=end_date)
    dataframe.columns = dataframe.columns.droplevel(1)
    print(f"Data downloaded successfully. Shape: {dataframe.shape}")

    # Check the first few rows and column names
    print("\nDataFrame Head:")
    print(dataframe.head())

    print("\nDataFrame Info:")
    dataframe.info()

except Exception as e:
    print(f"Error downloading data: {e}")
    # Exit or handle error appropriately
    exit()

# Ensure the DataFrame index is a DatetimeIndex (yf.download usually does
this)
if not isinstance(dataframe.index, pd.DatetimeIndex):
    print("Converting index to DatetimeIndex...")
    dataframe.index = pd.to_datetime(dataframe.index)

print("\nData is ready in Pandas DataFrame format.")
```

```
[*********************100%***********************]  1 of 1 completed
Downloading AAPL data from 2020-01-01 to 2023-12-31...
Data downloaded successfully. Shape: (1006, 5)

DataFrame Head:
Price           Close       High        Low       Open     Volume
Date
2020-01-02  72.716064  72.776591  71.466805  71.721011  135480400
2020-01-03  72.009140  72.771768  71.783985  71.941351  146322800
2020-01-06  72.582909  72.621646  70.876075  71.127866  118387200
```

```
2020-01-07  72.241524  72.849201  72.021208  72.592571  108872000
2020-01-08  73.403648  73.706279  71.943759  71.943759  132079200

DataFrame Info:
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 1006 entries, 2020-01-02 to 2023-12-29
Data columns (total 5 columns):
 #   Column  Non-Null Count  Dtype
---  ------  --------------  -----
 0   Close   1006 non-null   float64
 1   High    1006 non-null   float64
 2   Low     1006 non-null   float64
 3   Open    1006 non-null   float64
 4   Volume  1006 non-null   int64
dtypes: float64(4), int64(1)
memory usage: 47.2 KB

Data is ready in Pandas DataFrame format.
```

This code snippet defines the stock ticker and the date range. It then calls `yf.download()`, which returns the historical data as a Pandas DataFrame. We print the "head" (first few rows) and the "info" (column names and data types) to inspect the result.

You should see columns like `Open`, `High`, `Low`, `Close`, `Adj Close` (Adjusted Close, accounting for dividends and stock splits), and `Volume`. The index of the DataFrame should be the Date. Having a `DatetimeIndex` is crucial for time-series analysis and for `backtrader`.

Feeding Data into Backtrader

`backtrader` doesn't work directly with Pandas DataFrames. It uses its own optimized `Data Feed` objects. Fortunately, it provides convenient ways to convert common formats, like Pandas DataFrames, into these `Data Feed` objects.

The primary tool for this is `bt.feeds.PandasData`. We need to tell it which DataFrame to use and, optionally, how the columns in our DataFrame map to the standard OHLCV names that `backtrader` expects (`open`, `high`, `low`, `close`, `volume`, `openinterest`).

By default, `bt.feeds.PandasData` looks for columns with these exact lowercase names, or variations like 'Open', 'High', 'Low', 'Close', 'Volume'. The column names from `yfinance` (`Open`, `High`, `Low`, `Close`, `Volume`) usually match well enough for the defaults to work for the basic OHLCV fields.

Let's create a `backtrader` data feed from our downloaded DataFrame:

```
# Create a Backtrader Data Feed from the Pandas DataFrame
# Ensure the DataFrame has the expected column names or map them explicitly
# Default expected names: open, high, low, close, volume, openinterest
# yfinance names (Open, High, Low, Close, Adj Close, Volume) are usually
compatible
```

```
data = bt.feeds.PandasData(
    dataname=dataframe,
    fromdate=datetime.datetime.strptime(start_date, '%Y-%m-%d'), # Optional:
Set start date filter
    todate=datetime.datetime.strptime(end_date, '%Y-%m-%d')      # Optional:
Set end date filter
)

print(f"\nBacktrader Data Feed created: {data}")
```

```
Backtrader Data Feed created: <backtrader.feeds.pandafeed.PandasData object a
t 0x7d44de125610>
```

Here, `dataname=dataframe` tells `PandasData` to use our DataFrame. We also explicitly pass `fromdate` and `todate` using `datetime` objects converted from our start/end date strings. While `PandasData` *can* infer the range, explicitly setting it ensures `backtrader` only operates within the desired window, matching the downloaded data range.

Note on Adjusted Close: For serious backtesting, using the `Adj Close` price (which accounts for dividends and splits) is often preferred over the nominal `Close` price. `PandasData` can be configured to use different columns. For example: `bt.feeds.PandasData(dataname=dataframe, close='Adj Close', ...)`. For simplicity in this initial chapter, we'll stick with the default 'Close'.

The Cerebro Engine: Setting the Stage

Now that we have our data prepared in a format `backtrader` understands, we need to introduce the main controller: the **Cerebro** engine. Think of Cerebro (Spanish for "brain") as the orchestrator of your backtest. It brings together the data feeds, the trading strategy, the broker simulation (cash, commissions), and any analyzers you might want to use.

Let's create a Cerebro instance and configure some basic settings:

```
# Create a Cerebro entity
cerebro = bt.Cerebro()
print("\nCerebro engine initialized.")

# Add the Data Feed to Cerebro
cerebro.adddata(data)
print(f"Data feed added to Cerebro.")

# Set our desired cash start
initial_cash = 10000.0
cerebro.broker.setcash(initial_cash)
print(f"Initial cash set to: ${initial_cash:,.2f}")

# Set the commission scheme
# Example: 0.1% commission per trade (0.001)
commission_perc = 0.001 # 0.1%
cerebro.broker.setcommission(commission=commission_perc)
print(f"Commission set to: {commission_perc*100:.3f}% per trade")
```

```
# --- Strategy will be added here later ---
# cerebro.addstrategy(YourStrategyClass)

Cerebro engine initialized.
Data feed added to Cerebro.
Initial cash set to: $10,000.00
Commission set to: 0.100% per trade
```

In this block:

1. `cerebro = bt.Cerebro()`: We create the main engine instance.
2. `cerebro.adddata(data)`: We attach our prepared data feed. You can add multiple data feeds if your strategy trades multiple assets or uses different timeframes.
3. `cerebro.broker.setcash(...)`: We tell the simulated broker how much starting capital our strategy has.
4. `cerebro.broker.setcommission(...)`: We define the transaction costs. Ignoring commissions can drastically overestimate performance, so it's crucial to include a realistic estimate. Here, we set a 0.1% commission per trade.

Cerebro is now aware of our market data and the initial trading conditions (cash, commission). The next step is to give it a strategy to execute.

Adding a Minimal Strategy and Running the Test

A backtest isn't complete without a trading strategy. `backtrader` strategies are defined as Python classes inheriting from `bt.Strategy`. For now, we'll create the simplest possible strategy – one that doesn't actually trade but prints a message during initialization and potentially logs some data in its `next` method. The `next` method is the heart of a strategy, called by Cerebro for each bar of data (after an initial warm-up period for indicators, which we'll cover later).

```
# Define a simple Strategy
class MyFirstStrategy(bt.Strategy):
    params = (
        ('exitbars', 5), # Example parameter
    )

    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}') # Print date and message

    def __init__(self):
        # Keep a reference to the "close" line in the data[0] dataseries
        self.dataclose = self.datas[0].close
        self.log('MyFirstStrategy Initialized')
        # To keep track of pending orders
        self.order = None

    def notify_order(self, order):
```

```
        # Basic notification logic (we'll expand this later)
        if order.status in [order.Submitted, order.Accepted]:
            # Buy/Sell order submitted/accepted to/by broker - Nothing to do
            self.log(f'ORDER {order.getstatusname()}')
            return

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

        elif order.status in [order.Canceled, order.Margin, order.Rejected]:
            self.log(f'Order {order.getstatusname()}')

        self.order = None # Reset order status

    def next(self):
        # Simply log the closing price of the series from the reference
        # self.log(f'Close Price: {self.dataclose[0]:.2f}')

        # Basic logic example: Buy on the first bar, sell after N bars
        if len(self) == 1: # Check if it's the first bar
             if not self.position: # Check if not in market
                 self.log('BUY CREATE, %.2f' % self.dataclose[0])
                 self.order = self.buy() # Place a market buy order

        # Sell after holding for 'exitbars' days
        elif len(self) >= self.params.exitbars + 1 :
             if self.position: # Check if in market
                 self.log('SELL CREATE, %.2f' % self.dataclose[0])
                 self.order = self.sell() # Place a market sell order

# Add the strategy to Cerebro
cerebro.addstrategy(MyFirstStrategy)
print("\nStrategy added to Cerebro.")

# Run the backtest
print("\nRunning backtest...")
print(f'Starting Portfolio Value: {cerebro.broker.getvalue():,.2f}')

# Run over everything
results = cerebro.run() # Execute the backtest
first_strategy_instance = results[0] # Get the first strategy instance

print(f'Final Portfolio Value: {cerebro.broker.getvalue():,.2f}')
print("Backtest complete.")
```

```
Strategy added to Cerebro.

Running backtest...
```

```
Starting Portfolio Value: 10,000.00
2023-12-29 - MyFirstStrategy Initialized
2020-01-02 - BUY CREATE, 72.72
2020-01-03 - ORDER Submitted
2020-01-03 - ORDER Accepted
2020-01-03 - BUY EXECUTED, Price: 71.94, Cost: 71.94, Comm 0.07
2020-01-09 - SELL CREATE, 74.96
2020-01-10 - ORDER Submitted
2020-01-10 - ORDER Accepted
2020-01-10 - SELL EXECUTED, Price: 75.20, Cost: 71.94, Comm 0.08
Final Portfolio Value: 10,003.11
Backtest complete.
```

Here, `MyFirstStrategy` includes:

- A `log` method for consistent output formatting.
- `__init__`: Stores the closing price line and logs initialization.
- `notify_order`: A basic method to log order status changes (essential for debugging later). We'll enhance this in future chapters.
- `next`: Contains the core logic. Here, it just logs the closing price on each bar *and* includes a very simple buy-and-hold-for-N-bars example to demonstrate basic order placement (`self.buy()`, `self.sell()`) and position checking (`self.position`). This makes the output slightly more interesting than doing nothing.
- `cerebro.addstrategy(MyFirstStrategy)`: We register our strategy class with Cerebro.
- `cerebro.run()`: This is the command that starts the simulation. Cerebro loops through the data, calling the strategy's `next` method for each bar.
- We print the portfolio value before and after the run to see the impact of the (minimal) trading activity and commissions.

Visualizing Results: Plotting

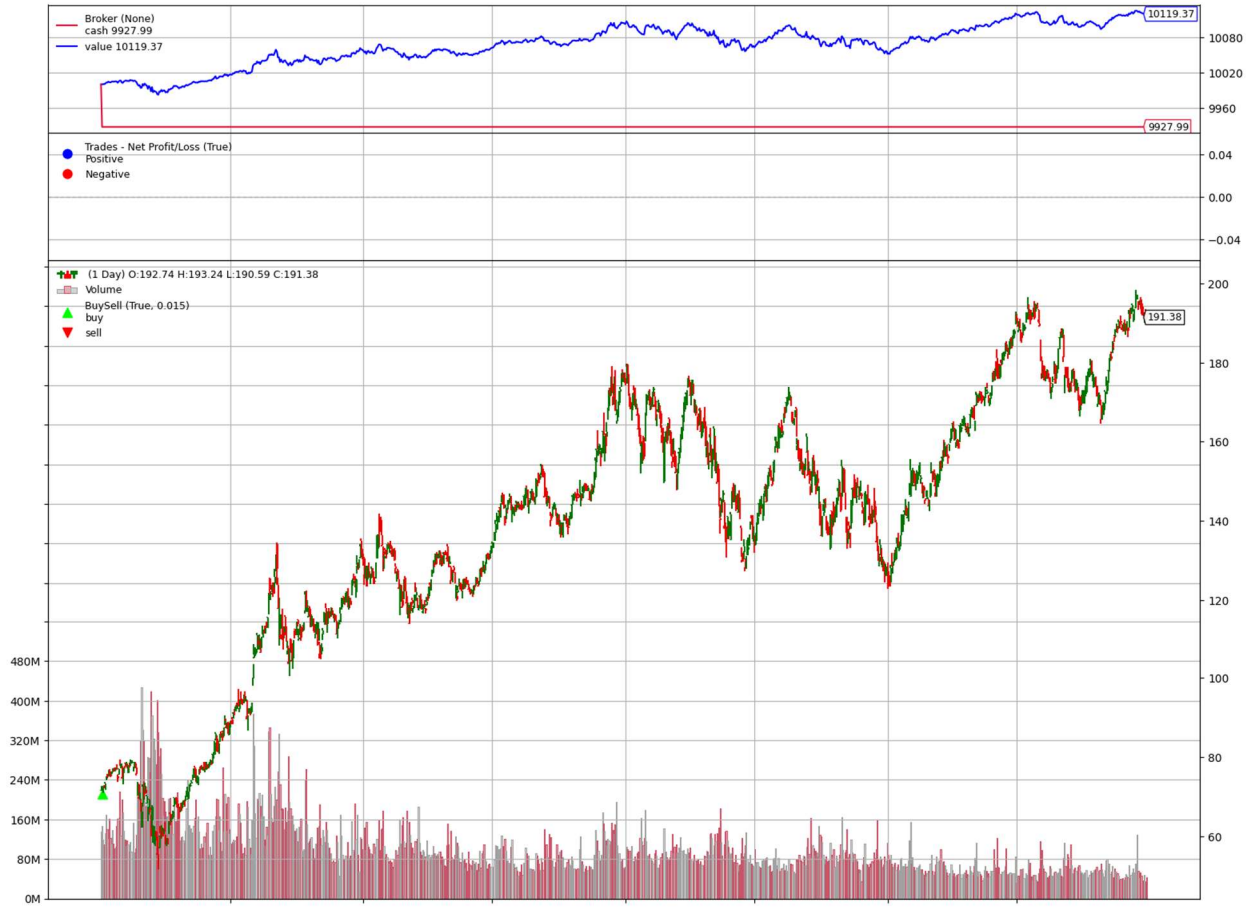
Numbers are essential, but a visual representation can often provide much deeper insights into strategy behavior. `backtrader` integrates with `matplotlib` to generate informative charts directly from the Cerebro engine after a run.

To plot the results, simply call `cerebro.plot()`:

```
# Plot the results
import matplotlib
%matplotlib inline


print("\nGenerating plot...")
try:
    # style='candlestick' is visually appealing for OHLC data
    cerebro.plot(style='candlestick', barup='green', bardown='red',
volume=True, iplot=False, show=False)
```

```
    print("Plot displayed.") # In interactive environments, plot shows
automatically
                             # In scripts, might save to file or display
window
except Exception as e:
    print(f"Could not plot results: {e}. Ensure matplotlib is installed and
working.")
```

```
Generating plot...
```




```
Plot displayed.
```

This command generates a chart typically showing:

- The main price data (as an OHLC chart or candlestick chart).
- Volume bars below the price.
- Any indicators added by the strategy (we'll see this later).
- Buy (upward triangle) and Sell (downward triangle) markers on the price chart where trades occurred.
- Portfolio value/equity curve (in later examples with analyzers).

The `style='candlestick'` makes the price chart more traditional. `barup` and `bardown` customize the candle colors.

Putting It All Together: Full Script

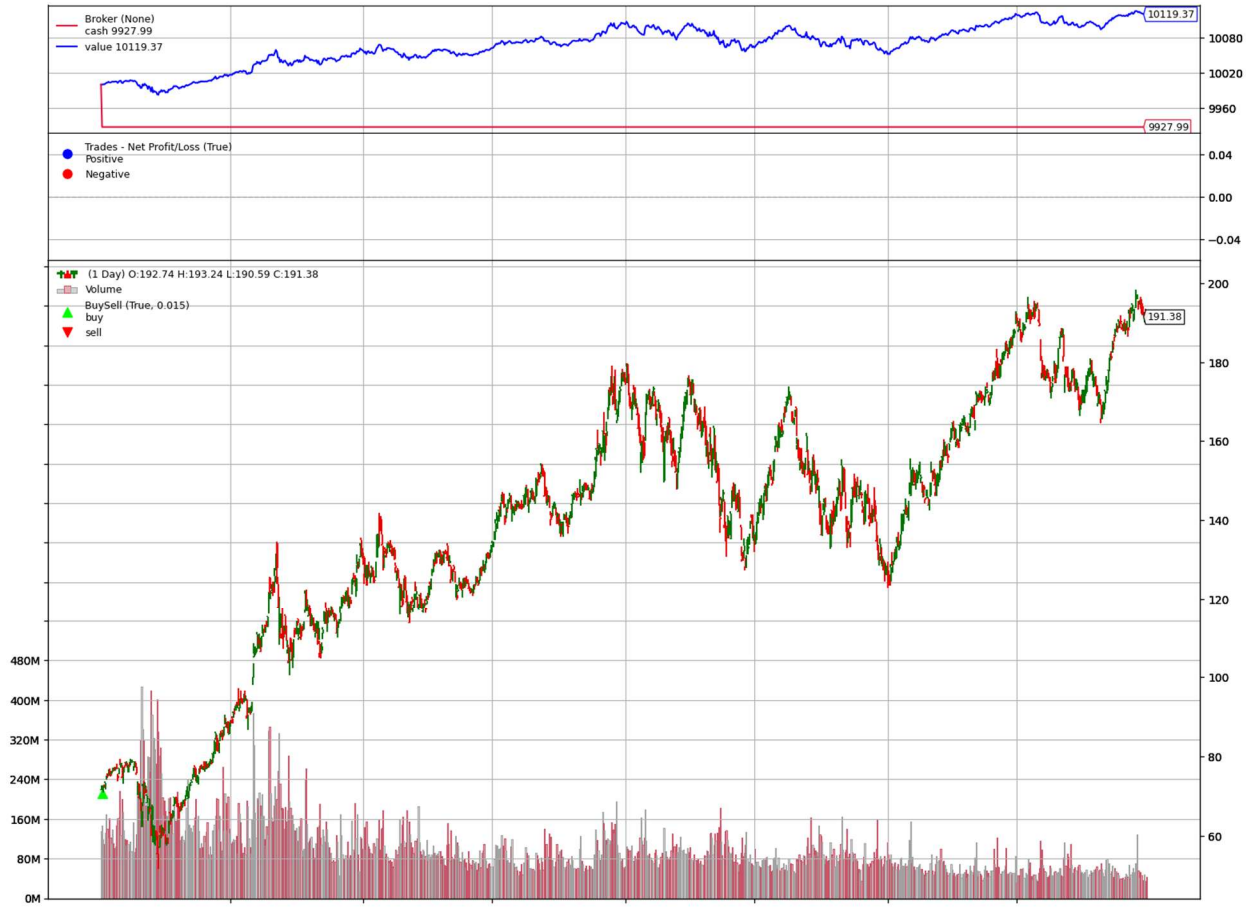
Here is the complete script combining all the steps from this chapter:

```
# -*- coding: utf-8 -*-
# chapter1_basics_full.py

from __future__ import (absolute_import, division, print_function,
                        unicode_literals)

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

# 1. Download Data
ticker = 'AAPL'
start_date = '2020-01-01'
end_date = '2023-12-31'
print(f"Downloading {ticker} data from {start_date} to {end_date}...")
try:
    dataframe = yf.download(ticker, start=start_date, end=end_date)
    dataframe.columns = dataframe.columns.droplevel(1)
    print(f"Data downloaded successfully. Shape: {dataframe.shape}")
    if dataframe.empty:
        print("No data downloaded, please check ticker and dates.")
        exit()
    if not isinstance(dataframe.index, pd.DatetimeIndex):
        dataframe.index = pd.to_datetime(dataframe.index)
except Exception as e:
    print(f"Error downloading data: {e}")
    exit()

# 2. Create Backtrader Data Feed
data = bt.feeds.PandasData(
    dataname=dataframe,
    fromdate=datetime.datetime.strptime(start_date, '%Y-%m-%d'),
    todate=datetime.datetime.strptime(end_date, '%Y-%m-%d')
)

# 3. Define Strategy
class MyFirstStrategy(bt.Strategy):
    params = (('exitbars', 5),)

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

    def __init__(self):
        self.dataclose = self.datas[0].close
        self.order = None
        self.buyprice = None
        self.buycomm = None
        self.log('MyFirstStrategy Initialized')
```

```
    def notify_order(self, order):
        if order.status in [order.Submitted, order.Accepted]:
            # self.log(f'ORDER {order.getstatusname()}') # Can be verbose
            return
        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:.2f}')
            # self.bar_executed = len(self)
        elif order.status in [order.Canceled, order.Margin, order.Rejected]:
            self.log(f'Order {order.getstatusname()}')
        self.order = None

    def notify_trade(self, trade):
         if trade.isclosed:
             self.log(f'TRADE PROFIT, GROSS {trade.pnl:.2f}, NET
{trade.pnlcomm:.2f}')

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

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

        # Check if we are in the market
        if not self.position:
             # Not yet ... we MIGHT BUY if ...
             if len(self) == 1: # Buy on first bar only for this simple
example
                 self.log('BUY CREATE, %.2f' % self.dataclose[0])
                 self.order = self.buy()
        else:
             # Already in the market ... we might sell
             if len(self) >= (self.bar_executed + self.params.exitbars): #
Sell after holding N bars
                  self.log('SELL CREATE, %.2f' % self.dataclose[0])
                  self.order = self.sell()

    # Need bar_executed for the simple exit logic above
    def __init__(self): # Redefine __init__ to include bar_executed
        self.dataclose = self.datas[0].close
        self.order = None
        self.buyprice = None
        self.buycomm = None
        self.bar_executed = 0 # Initialize bar_executed
        self.log('MyFirstStrategy Initialized')

    def notify_order(self, order): # Need to update bar_executed on
completion
        # ... (previous notify_order code) ...
```

```
        if order.status in [order.Completed]:
            # ... (buy/sell logging) ...
            if order.isbuy():
                self.bar_executed = len(self) # Record bar number when buy is
executed
            # elif order.issell(): # Not needed for this simple exit logic
        # ... (rest of notify_order code) ...


# 4. Set up Cerebro
cerebro = bt.Cerebro()
cerebro.addstrategy(MyFirstStrategy)
cerebro.adddata(data)
cerebro.broker.setcash(10000.0)
cerebro.broker.setcommission(commission=0.001)

# 5. Run and Plot
import matplotlib
%matplotlib inline
print(f'\nStarting Portfolio Value: {cerebro.broker.getvalue():,.2f}')
results = cerebro.run()
print(f'Final Portfolio Value: {cerebro.broker.getvalue():,.2f}')

print("\nGenerating plot...")
try:
    cerebro.plot(style='candlestick', barup='green', bardown='red',
volume=True, iplot=False)
    print("Plot displayed/saved.")
except Exception as e:
    print(f"Could not plot results: {e}")
```

```
[**********************100%**********************]  1 of 1 completed
Downloading AAPL data from 2020-01-01 to 2023-12-31...
Data downloaded successfully. Shape: (1006, 5)

Starting Portfolio Value: 10,000.00
2023-12-29 - MyFirstStrategy Initialized
2020-01-02 - BUY CREATE, 72.72
Final Portfolio Value: 10,119.37

Generating plot...
```




```
Plot displayed/saved.
```

Note: Added `notify_trade` and refined the simple buy/sell logic in the full script example to make it slightly more complete and ensure the exit logic works correctly by tracking `bar_executed`.

Chapter Summary

Congratulations! You've successfully set up and executed your first `backtrader` backtest. We covered the essential steps:

- Importing necessary libraries.
- Downloading historical data using `yfinance`.
- Converting the data into a `backtrader` feed using `bt.feeds.PandasData`.
- Initializing the `Cerebro` engine and configuring the broker (cash, commission).
- Defining and adding a minimal `bt.Strategy`.
- Running the backtest with `cerebro.run()`.
- Visualizing the results with `cerebro.plot()`.

This foundational workflow is the basis upon which we will build more complex, indicator-driven strategies in the coming chapters. You now have a template for loading data and running a backtest. Next, we'll dive into the core focus: implementing and utilizing technical indicators within your strategies.

Chapter 2: Built-In Indicators

Indicators as Strategy Building Blocks

In Chapter 1, we successfully set up our `backtrader` environment, loaded historical market data, and ran a minimal backtest using the Cerebro engine. We have the scaffolding in place, but a strategy needs logic – rules that decide when to buy or sell. In the world of technical analysis, these rules are most often derived from **Technical Indicators**.

Technical indicators are mathematical calculations based on an asset's price, volume, or other data points. They aim to distill complex market action into more easily interpretable signals, helping traders identify trends, measure momentum, gauge volatility, or spot potential reversals.

`backtrader` treats indicators as fundamental components. As we touched upon briefly, it has a sophisticated system for handling them:

- **First-Class Citizens:** Indicators are core objects within the framework, not just afterthoughts.
- **Lines Concept:** Indicators (like data feeds) operate on "lines". They take one or more input lines (e.g., the `close` line from a data feed) and produce one or more output lines (e.g., the SMA value line, the RSI value line).
- **Automatic `minperiod`:** When you use an indicator requiring N periods of data (like a 20-period SMA), `backtrader` automatically calculates this minimum period (`minperiod`). It ensures that your strategy's `next()` method only starts receiving calls *after* all indicators have enough data to produce their first valid output, preventing errors based on insufficient initial data.
- **Composability:** Indicators can be applied not just to raw data but also to the output lines of *other* indicators, allowing for complex, layered analysis.
- **Efficiency:** Indicator calculations are often optimized within the `backtrader` framework.

Most of the commonly used technical indicators are readily available within the `backtrader.indicators` submodule, which is often imported or accessed via the shorter alias `bt.ind`.

This chapter marks our deep dive into using these built-in indicators. We'll start with two of the most fundamental and widely used: the **Simple Moving Average (SMA)** and the **Relative Strength Index (RSI)**. We will learn how to implement them in a `backtrader` strategy, access their values, and understand how they can form the basis of trading signals.

The Simple Moving Average (SMA) - Concept

The Simple Moving Average is arguably the most basic and popular trend-following indicator. As the name suggests, it calculates the average price of an asset over a specified number of past periods.

What is it? An SMA is calculated by summing the closing prices for the last 'N' periods and then dividing that sum by 'N'. For example, a 20-period SMA takes the sum of the closing prices of the last

20 bars (days, hours, etc.) and divides by 20. As a new bar forms, the oldest price in the calculation is dropped, and the newest price is added, causing the average to "move" over time.

Purpose: The primary purpose of an SMA is to **smooth out price action** and help identify the underlying trend direction. By averaging prices, it filters out the day-to-day noise and volatility, providing a clearer picture of the market's general trajectory.

- **Trend Identification:** If the price is consistently trading above a rising SMA, it generally suggests an uptrend. Conversely, if the price is below a falling SMA, it suggests a downtrend. The slope of the SMA itself can also indicate trend strength.
- **Support and Resistance:** Longer-term SMAs (like the 50-day or 200-day SMA) are often watched by market participants and can act as dynamic levels of support (in an uptrend) or resistance (in a downtrend).
- **Signal Generation:** Crossovers between the price and the SMA, or crossovers between two SMAs of different lengths (a "fast" short-period SMA and a "slow" long-period SMA), are commonly used to generate buy or sell signals.

Visual Representation: On a price chart, the SMA appears as a single line that follows the general path of the price but is smoother and lags slightly behind it.

The `period` Parameter: The key parameter for an SMA is its **period (N)**. This determines how many past bars are included in the average.

- **Shorter Period (e.g., 10, 20):** The SMA will react more quickly to recent price changes, resulting in a line that follows the price more closely but is also more susceptible to noise ("whipsaws").
- **Longer Period (e.g., 50, 100, 200):** The SMA will be smoother and less reactive to short-term fluctuations, providing a clearer view of the longer-term trend but generating signals with more lag.

Choosing the right period(s) often depends on the trading style (short-term vs. long-term) and the characteristics of the asset being traded.

Implementing SMA in Backtrader

`backtrader` makes using an SMA straightforward through its built-in `SimpleMovingAverage` indicator class.

The Class: You can access it via `backtrader.indicators.SimpleMovingAverage` or its convenient alias `bt.ind.SMA`.

Instantiation: You typically create an instance of the SMA indicator within your strategy's `__init__` method. This is where you define which data the indicator should operate on and specify its parameters, most importantly the `period`.

```
!pip install backtrader
import backtrader as bt

class MySmaStrategy(bt.Strategy):
    params = (
        ('sma_period', 20), # Default SMA period parameter for the strategy
    )

    def __init__(self):
        self.log('Initializing Strategy...')
        # Keep a reference to the closing price line
        self.dataclose = self.datas[0].close

        # Instantiate the Simple Moving Average indicator
        self.sma = bt.indicators.SimpleMovingAverage(
            self.datas[0], # The data feed to operate on (default: datas[0])
            # self.dataclose, # Can also explicitly pass the target line
            period=self.params.sma_period # The lookback period
        )
        # You could also use the alias:
        # self.sma = bt.ind.SMA(self.datas[0], period=self.params.sma_period)

        self.log(f'SMA indicator created with period
{self.params.sma_period}')
        self.log('Strategy Initialized.')

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

    def next(self):
        # We will access self.sma here later
        pass
```

```
Collecting backtrader
  Downloading backtrader-1.9.78.123-py2.py3-none-any.whl.metadata (6.8 kB)
Downloading backtrader-1.9.78.123-py2.py3-none-any.whl (419 kB)
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 0.0/419
.5 kB ? eta -:--:--
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 419.5/4
19.5 kB 18.0 MB/s eta 0:00:00
Installing collected packages: backtrader
Successfully installed backtrader-1.9.78.123
```

Explanation:

1. **`params = (('sma_period', 20),)`:** We define a strategy parameter `sma_period` with a default value of 20. This allows us to easily change the SMA period when running the backtest without modifying the core strategy code.
2. **`self.dataclose = self.datas[0].close`:** We store a reference to the 'close' price line of the primary data feed (`self.datas[0]`) for convenience, although it's not strictly necessary for instantiating the SMA itself.
3. **`self.sma = bt.indicators.SimpleMovingAverage(...)`:** This is the core instantiation.

- The first argument, `self.datas[0]`, tells the indicator which data feed to use. If omitted, `backtrader` assumes `self.datas[0]`. If you explicitly provide a specific line like `self.dataclose` or `self.data.close`, the indicator will use that specific line from the default data feed.
- `period=self.params.sma_period`: We pass the desired lookback period, using the value defined in our strategy `params`.

4. **`self.sma = ...`:** We store the created indicator object as an attribute of our strategy instance (`self.sma`). This allows us to access its calculated values later in the `next` method.

Now, whenever the `backtrader` engine runs, it will automatically calculate the 20-period SMA for each bar of our data feed.

Accessing SMA Values

Once the SMA indicator is instantiated in `__init__`, how do we get its calculated value for the current or previous bars within our strategy's `next()` method?

This is where the "lines" concept is key. The `self.sma` object we created isn't just a single number; it represents a **line** (or array) of SMA values calculated across the entire history of the data feed, up to the current bar being processed by `next()`.

We access these values using standard Python indexing, similar to accessing elements in a list or array:

- **`self.sma[0]`**: Gets the calculated SMA value for the **current** bar that the `next()` method is currently evaluating.
- **`self.sma[-1]`**: Gets the SMA value for the **previous** bar.
- **`self.sma[-2]`**: Gets the SMA value from **two bars ago**.
- And so on... `self.sma[-n]` gets the value from `n` bars ago.

Example within `next()`:

```
# Inside the next() method of MySmaStrategy

# Get the current SMA value
current_sma_value = self.sma[0]

# Get the previous bar's SMA value
previous_sma_value = self.sma[-1]

# Get the current closing price
current_close = self.dataclose[0]

# Log the values (maybe periodically)
if len(self) % 10 == 0: # Log every 10 bars
    self.log(f'Bar Index: {len(self)}, Close: {current_close:.2f}, Current
SMA: {current_sma_value:.2f}, Previous SMA: {previous_sma_value:.2f}')
```

Minimum Period (`minperiod`) Handling: Remember, an N-period SMA requires N data points to calculate its first value. For a 20-period SMA, `self.sma[0]` will only contain a valid number starting from the 20th bar onwards (index 19, as indexing starts from 0). Before that, the value would conceptually be 'Not a Number' (NaN).

You *don't* need to manually check if enough bars have passed. `backtrader` handles this automatically. The `next()` method of your strategy will **not** be called by the Cerebro engine until *all* indicators instantiated in `__init__` have processed enough data to produce their first valid output. So, by the time your `next()` method runs for the first time, you can safely access `self.sma[0]` (and similarly for other indicators), knowing it corresponds to a valid calculation based on the indicator's required `minperiod`.

This automatic handling simplifies strategy code significantly, as you don't need boilerplate checks for data availability for each indicator.

SMA Use Case: Crossover Logic Examples

Accessing the current and previous values of the price and the SMA allows us to implement common trading signals based on crossovers.

1. Price/SMA Crossover: A simple signal is generated when the price crosses above or below the SMA.

```
# Inside next()

# Check for bullish crossover (price crosses above SMA)
if self.dataclose[0] > self.sma[0] and self.dataclose[-1] <= self.sma[-1]:
    self.log(f'BULLISH SIGNAL: Close {self.dataclose[0]:.2f} crossed ABOVE
SMA {self.sma[0]:.2f}')
    # --- Potential Buy Logic Here ---
    # self.buy()

# Check for bearish crossover (price crosses below SMA)
elif self.dataclose[0] < self.sma[0] and self.dataclose[-1] >= self.sma[-1]:
    self.log(f'BEARISH SIGNAL: Close {self.dataclose[0]:.2f} crossed BELOW
SMA {self.sma[0]:.2f}')
    # --- Potential Sell/Close Logic Here ---
    # self.sell() / self.close()
```

This logic checks two conditions:

- The *current* relationship (e.g., `close > sma`).
- The *previous* relationship (e.g., `close <= sma`). Only when the relationship flips *on the current bar* does the condition evaluate to `True`, indicating a crossover just occurred.

2. Two SMA Crossover ("Golden Cross" / "Death Cross" Concept): Another popular technique uses two SMAs: a shorter-period "fast" SMA and a longer-period "slow" SMA. A buy signal might be generated when the fast SMA crosses above the slow SMA (often called a "Golden Cross"), suggesting

increasing upward momentum. A sell signal might occur when the fast SMA crosses below the slow SMA (a "Death Cross"), suggesting waning momentum or a potential downtrend.

```
# Requires defining self.sma_fast and self.sma_slow in __init__
# Example:
# self.sma_fast = bt.ind.SMA(period=10)
# self.sma_slow = bt.ind.SMA(period=30)

# Inside next()

# Check for bullish crossover (Fast SMA crosses above Slow SMA)
if self.sma_fast[0] > self.sma_slow[0] and self.sma_fast[-1] <=
self.sma_slow[-1]:
    self.log(f'BULLISH CROSS: Fast SMA ({self.sma_fast[0]:.2f}) crossed ABOVE
Slow SMA ({self.sma_slow[0]:.2f})')
    # --- Potential Buy Logic Here ---

# Check for bearish crossover (Fast SMA crosses below Slow SMA)
elif self.sma_fast[0] < self.sma_slow[0] and self.sma_fast[-1] >=
self.sma_slow[-1]:
    self.log(f'BEARISH CROSS: Fast SMA ({self.sma_fast[0]:.2f}) crossed BELOW
Slow SMA ({self.sma_slow[0]:.2f})')
    # --- Potential Sell/Close Logic Here ---
```

Visual Confirmation: When you run `cerebro.plot()`, these SMA lines will be drawn directly on the main price chart. This is invaluable for visually verifying your strategy's logic. You can see exactly where the price crosses the SMA or where the two SMAs intersect, confirming if your code is correctly identifying these events. The plot provides immediate feedback on how the indicator behaves relative to price action.

The Relative Strength Index (RSI) - Concept

While the SMA helps identify trends, the **Relative Strength Index (RSI)** is a **momentum oscillator**. Developed by the legendary J. Welles Wilder Jr. (who also created ADX and Parabolic SAR), RSI measures the speed and change of price movements, helping to identify potential overbought or oversold conditions in the market.

What is it? RSI compares the magnitude of recent gains to recent losses over a specified time period. It calculates a ratio of the average gains on days the price closed up versus the average losses on days the price closed down, typically over the last 14 periods. This ratio is then normalized to oscillate between a fixed range of **0 and 100**.

Purpose and Interpretation: RSI helps traders gauge whether an asset's price has moved too far, too fast, potentially indicating an upcoming reversal or pause.

- **Overbought/Oversold Thresholds:** This is the most common use of RSI.
  - **Overbought:** Readings above a certain level (traditionally **70**, sometimes **80**) suggest that the asset has experienced strong buying pressure recently and might be

overvalued or due for a corrective pullback. It's considered a potential area to look for sell signals or take profits on long positions.
    - **Oversold:** Readings below a certain level (traditionally **30**, sometimes **20**) suggest the asset has faced strong selling pressure and might be undervalued or due for a relief rally (bounce). It's considered a potential area to look for buy signals or cover short positions.
- **Centerline Crossovers:** The **50** level is often considered a centerline. Some traders view sustained RSI readings above 50 as indicative of bullish momentum and readings below 50 as indicative of bearish momentum. Crossovers of the 50 line can sometimes be used as buy/sell signals, particularly in trending markets.
- **Divergence:** This is a more advanced concept.
    - *Bearish Divergence:* Occurs when the price makes a new high, but the RSI fails to make a corresponding new high. This suggests that the upward momentum is weakening and could foreshadow a reversal lower.
    - *Bullish Divergence:* Occurs when the price makes a new low, but the RSI fails to make a corresponding new low (making a higher low instead). This suggests selling momentum is weakening and might precede a reversal higher.

Important Note: RSI, especially the overbought/oversold levels, tends to work best in **ranging or sideways markets**. In strong trending markets, RSI can remain in overbought territory (in a strong uptrend) or oversold territory (in a strong downtrend) for extended periods without an immediate reversal. Therefore, using RSI signals in isolation can be risky; confirmation from other indicators or price action analysis is often recommended.

Implementing RSI in Backtrader

Using the RSI in `backtrader` is just as simple as using the SMA, thanks to the `RelativeStrengthIndex` indicator class.

The Class: You can access it via `backtrader.indicators.RelativeStrengthIndex` or its alias `bt.ind.RSI`.

Instantiation: Similar to the SMA, you create an RSI instance within your strategy's `__init__` method, specifying the `period`.

```
import backtrader as bt

class MyRsiStrategy(bt.Strategy):
    params = (
        ('rsi_period', 14), # Default RSI period (Wilder's recommendation)
    )

    def __init__(self):
        self.log('Initializing Strategy...')
        self.dataclose = self.datas[0].close # Reference to close price

        # Instantiate the Relative Strength Index indicator
        self.rsi = bt.indicators.RelativeStrengthIndex(
```

```
            # self.datas[0], # Operates on datas[0] by default
            period=self.params.rsi_period
        )
        # Alias: self.rsi = bt.ind.RSI(period=self.params.rsi_period)

        self.log(f'RSI indicator created with period
{self.params.rsi_period}')
        self.log('Strategy Initialized.')

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

    def next(self):
        # We will access self.rsi here later
        pass
```

Explanation:

1. **`params = (('rsi_period', 14),)`:** We set the default RSI lookback period to 14, the value commonly recommended by Wilder.
2. **`self.rsi = bt.indicators.RelativeStrengthIndex(...)`:** We instantiate the indicator.
    - It automatically applies to the `close` price of `self.datas[0]` if no specific data line is provided.
    - `period=self.params.rsi_period`: Sets the lookback period using our strategy parameter.
3. **`self.rsi = ...`:** The RSI indicator object, representing the line of calculated RSI values, is stored in the `self.rsi` attribute.

`backtrader` will now compute the 14-period RSI value for each bar.

Accessing RSI Values: Accessing the calculated RSI value in `next()` follows the exact same pattern as the SMA:

- `self.rsi[0]`: The RSI value for the **current** bar.
- `self.rsi[-1]`: The RSI value for the **previous** bar.
- `self.rsi[-n]`: The RSI value **n** bars ago.

Again, `backtrader` ensures `next()` is only called once the RSI has enough data (14 periods in this case) to produce its first valid output.

RSI Use Case: Overbought/Oversold Signal Logic

The most direct way to use RSI is to check if it has crossed into the overbought or oversold zones.

```
# Inside the next() method of MyRsiStrategy

# Define thresholds (can be parameters or constants)
rsi_oversold_level = 30.0
```

```
rsi_overbought_level = 70.0

# Get the current RSI value
current_rsi = self.rsi[0]

# --- Basic Threshold Check ---
# if current_rsi < rsi_oversold_level:
#     self.log(f'RSI ({current_rsi:.2f}) is OVERSOLD (<
{rsi_oversold_level})')
#     # Potential buy logic... but careful, it could stay oversold!
# elif current_rsi > rsi_overbought_level:
#     self.log(f'RSI ({current_rsi:.2f}) is OVERBOUGHT (>
{rsi_overbought_level})')
#     # Potential sell logic... but careful, it could stay overbought!

# --- More Specific: Check for Zone Entry ---
# This identifies the specific bar the RSI crosses the threshold

# Check if RSI just crossed BELOW the oversold level
if self.rsi[0] < rsi_oversold_level and self.rsi[-1] >= rsi_oversold_level:
    self.log(f'SIGNAL: RSI ({current_rsi:.2f}) crossed INTO OVERSOLD Zone (<
{rsi_oversold_level})')
    # --- Potential Buy Signal Logic ---
    # E.g., self.buy() if not self.position else None

# Check if RSI just crossed ABOVE the overbought level
elif self.rsi[0] > rsi_overbought_level and self.rsi[-1] <=
rsi_overbought_level:
    self.log(f'SIGNAL: RSI ({current_rsi:.2f}) crossed INTO OVERBOUGHT Zone
(> {rsi_overbought_level})')
    # --- Potential Sell Signal / Close Long Logic ---
    # E.g., self.sell() / self.close() if self.position else None

# Optional: Check for exit from zones (e.g., RSI crossing back above 30 or
below 70)
# elif self.rsi[0] > rsi_oversold_level and self.rsi[-1] <=
rsi_oversold_level:
#    self.log(f'INFO: RSI ({current_rsi:.2f}) crossed back ABOVE OVERSOLD
Zone')
# elif self.rsi[0] < rsi_overbought_level and self.rsi[-1] >=
rsi_overbought_level:
#    self.log(f'INFO: RSI ({current_rsi:.2f}) crossed back BELOW OVERBOUGHT
Zone')
```

The "Zone Entry" logic is often more useful for generating discrete signals, as it pinpoints the exact bar the condition was met, rather than triggering repeatedly while RSI stays within the zone.

Visualisation: When `cerebro.plot()` is called for a strategy using RSI, `backtrader` is smart enough to typically display the RSI indicator in its own **separate panel** below the main price chart. This is because RSI values (0-100) are on a completely different scale than price. This dedicated panel makes it very easy to:

- See the RSI line oscillating.

- Visually identify when it crosses the overbought (e.g., 70) or oversold (e.g., 30) levels. You might want to mentally draw or imagine horizontal lines at these levels on the plot.
- Compare RSI movements (peaks, troughs, divergences) with the price action in the panel above.

This automatic separation of indicators with different scales is a very convenient feature of `backtrader`'s plotting system.

Chaining Indicators: Smoothing the RSI

One of the powerful features of `backtrader`'s indicator system is **composability**, or **chaining**. This means you can use the output line of one indicator as the input data for another indicator.

Why do this? Indicator lines, especially oscillators like RSI, can sometimes be "noisy," meaning they fluctuate rapidly and might generate frequent signals, some of which could be false (whipsaws). Applying a smoothing mechanism, like a Simple Moving Average, directly to the indicator's output line can help filter out some of this noise.

Example: SMA of RSI A common technique is to calculate an SMA of the RSI line itself. This creates a second, smoother line based on the average RSI value over a specified period.

Potential uses for an SMA of RSI include:

- Generating signals when the raw RSI line crosses its own SMA (e.g., RSI crossing above RSI-SMA as a buy signal).
- Using the smoothed RSI-SMA line for threshold crosses instead of the raw RSI, potentially reducing whipsaws.
- Analyzing the trend of the RSI itself via the slope of the RSI-SMA.

Implementation: Implementing this in `backtrader` is intuitive. You simply pass the *indicator object* (which represents its output line) as the data input when instantiating the second indicator.

Let's add an SMA of the RSI to our strategy's `__init__`:

```
# Inside the __init__ method of a strategy

# First, define the RSI indicator
self.rsi = bt.indicators.RelativeStrengthIndex(
    period=self.params.rsi_period # e.g., 14
)

# Now, define an SMA, using self.rsi as the input data
self.rsi_sma = bt.indicators.SimpleMovingAverage(
    self.rsi, # Pass the RSI indicator object/line here!
    period=self.params.rsi_sma_period # Define a period for the SMA, e.g., 10
)
# Alias: self.rsi_sma = bt.ind.SMA(self.rsi,
period=self.params.rsi_sma_period)
```

```
self.log(f'SMA of RSI created with period {self.params.rsi_sma_period}')
```

Explanation: Notice that instead of `self.datas[0]` or `self.data.close`, we pass `self.rsi` as the first argument to `bt.ind.SimpleMovingAverage`. `backtrader` understands that `self.rsi` represents a calculated line and applies the SMA to those values.

Accessing the Chained Indicator: You access the value of the chained indicator in `next()` just like any other indicator:

```
# Inside next()
current_rsi_value = self.rsi[0]
current_rsi_sma_value = self.rsi_sma[0] # Access the SMA of RSI

# Example Logic: RSI crossing its SMA
if current_rsi_value > current_rsi_sma_value and self.rsi[-1] <=
self.rsi_sma[-1]:
    self.log(f'RSI ({current_rsi_value:.2f}) crossed ABOVE its SMA
({current_rsi_sma_value:.2f})')
    # Potential buy signal...
elif current_rsi_value < current_rsi_sma_value and self.rsi[-1] >=
self.rsi_sma[-1]:
    self.log(f'RSI ({current_rsi_value:.2f}) crossed BELOW its SMA
({current_rsi_sma_value:.2f})')
    # Potential sell signal...
```

This chaining capability allows for the creation of more sophisticated analytical tools by building upon the basic indicators provided by `backtrader`.

Hands-On: The `MyStrategyWithIndicators` Code

Now, let's put the concepts of SMA, RSI, and chaining together by examining the `MyStrategyWithIndicators` class, which incorporates these elements. This provides a practical example of how indicators are defined and accessed within a strategy structure.

Here is the code for the strategy class:

```
# Full Strategy Class Code
import backtrader as bt

class MyStrategyWithIndicators(bt.Strategy):
    """
    Demonstrates basic instantiation and access for SMA and RSI,
    including an example of chaining (SMA on RSI).
    """
    # Define parameters for the strategy
    params = (
        ('sma_period', 20),         # Period for the Simple Moving Average
        ('rsi_period', 14),         # Period for the Relative Strength Index
        ('rsi_sma_period', 10),     # Period for the SMA applied to the RSI
        ('log_interval', 20),       # How often to log values in next()
```

```
    )

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

    def __init__(self):
        ''' Strategy constructor - called once before backtesting starts '''
        self.log(f'--- Strategy {self.__class__.__name__} Initializing ---')

        # Keep a reference to the primary data feed's close line
        self.dataclose = self.datas[0].close
        self.log(f'Data feed reference stored.')

        # 1. Instantiate Simple Moving Average (SMA)
        self.sma = bt.indicators.SimpleMovingAverage(
            self.datas[0], # Apply to the default data feed
            period=self.params.sma_period
        )
        self.log(f'SMA indicator created with period
{self.params.sma_period}')

        # 2. Instantiate Relative Strength Index (RSI)
        self.rsi = bt.indicators.RelativeStrengthIndex(
            period=self.params.rsi_period
            # Automatically applies to datas[0].close
        )
        self.log(f'RSI indicator created with period
{self.params.rsi_period}')

        # 3. Instantiate Chained Indicator: SMA of RSI
        # We will keep this commented out initially for clarity,
        # but show how it would be done.
        self.rsi_sma = None # Initialize to None
        # UNCOMMENT THE FOLLOWING LINES TO ACTIVATE RSI_SMA:
        # self.rsi_sma = bt.indicators.SimpleMovingAverage(
        #     self.rsi, # Apply SMA to the self.rsi line object
        #     period=self.params.rsi_sma_period
        # )
        # self.log(f'SMA of RSI created with period
{self.params.rsi_sma_period}')
        # if self.rsi_sma is None:
        #     self.log(f'SMA of RSI is currently INACTIVE (commented out).')


        self.log(f'--- Strategy {self.__class__.__name__} Initialized ---')

    def next(self):
        ''' Called on each bar after the minimum period requirement is met
'''

        # Log indicator values periodically based on log_interval parameter
        if len(self) % self.params.log_interval == 0:
            current_close = self.dataclose[0]
            current_sma = self.sma[0]
```

```
            current_rsi = self.rsi[0]

            log_msg = (f'Bar: {len(self)}, Close: {current_close:.2f}, '
                       f'SMA({self.params.sma_period}): {current_sma:.2f}, '
                       f'RSI({self.params.rsi_period}): {current_rsi:.2f}')

            # Include RSI_SMA in log if it's active
            if self.rsi_sma:
                 current_rsi_sma = self.rsi_sma[0]
                 log_msg += f', RSI_SMA({self.params.rsi_sma_period}):
{current_rsi_sma:.2f}'

            self.log(log_msg)

        # --- Placeholder for actual trading logic ---
        # This strategy doesn't place trades yet, it just demonstrates
        # indicator setup and access. Future chapters will add buy/sell logic
here.
        # Example checks we could add:
        # if self.rsi[0] < 30 and self.rsi[-1] >= 30:
        #     self.log("RSI crossed below 30 - Potential Buy Signal Area")
        #
        # if self.dataclose[0] > self.sma[0] and self.dataclose[-1] <=
self.sma[-1]:
        #     self.log("Price crossed above SMA - Potential Buy Signal")

        pass # End of next() method
```

Code Walkthrough & Cerebro Integration

Let's break down the `MyStrategyWithIndicators` class and see how to run it.

1. `params`:

- We define default periods for SMA (20), RSI (14), and the SMA-of-RSI (10).
- `log_interval` is added to control how frequently we print values in `next()`, avoiding excessive output.

2. `log(self, txt, dt=None)`:

- A helper method for standardized logging, printing the date followed by the message.

3. `__init__(self)`:

- This method runs only *once* when the strategy is first initialized by Cerebro.
- `self.dataclose = self.datas[0].close`: Stores a reference to the close price line.
- `self.sma = bt.ind.SimpleMovingAverage(...)`: Creates the SMA instance, applying it to the primary data feed (`self.datas[0]`) using the `sma_period` parameter.
- `self.rsi = bt.ind.RelativeStrengthIndex(...)`: Creates the RSI instance using the `rsi_period`. It implicitly uses `self.datas[0].close`.

- `self.rsi_sma = ...`: The code to create the SMA of the RSI is included but commented out by default. We initialize `self.rsi_sma` to `None` first. To activate it, you would uncomment the instantiation lines.
- Logging messages confirm which indicators are created and with which periods.

4. `next(self)`:

- This method runs for *each bar* of data, but only after enough bars have passed to satisfy the `minperiod` of all active indicators (in this case, dictated by the 20-period SMA initially).
- `len(self)` gives the current bar number being processed (starting from 1 after the `minperiod`).
- The `if len(self) % self.params.log_interval == 0:` block demonstrates accessing the current values (`[0]`) of `close`, `sma`, and `rsi`.
- It includes logic to *also* log the `rsi_sma[0]` value *if* the `self.rsi_sma` indicator was activated in `__init__`.
- Crucially, this `next()` method contains **no trading logic** (`self.buy()`, `self.sell()`) yet. Its purpose here is purely to demonstrate indicator setup and value access. We'll add trading logic based on indicator signals in later chapters.

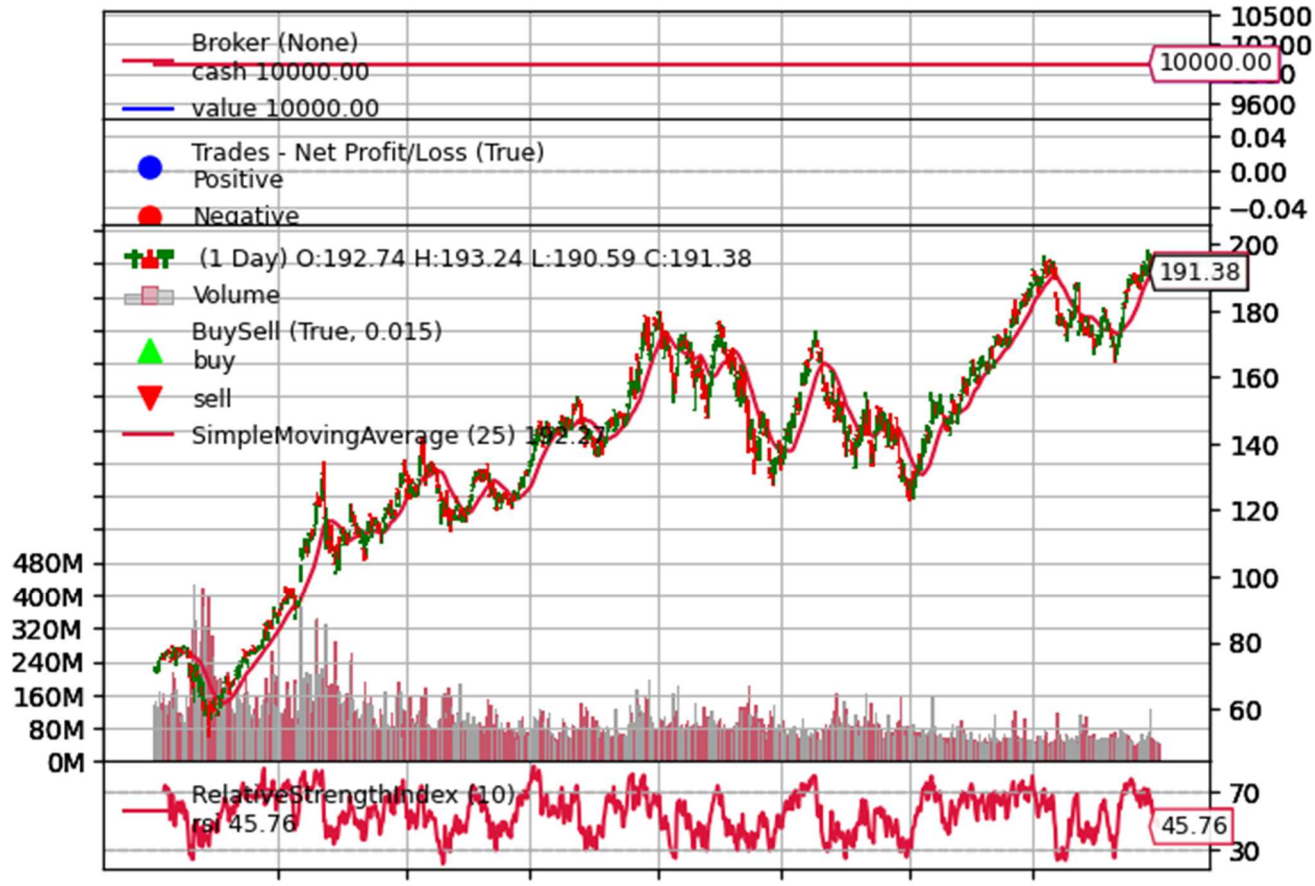
Cerebro Integration: To run this strategy, we integrate it into the Cerebro workflow established in Chapter 1.

```
# --- Assume previous code for imports and data loading ---
# (Using yfinance download and PandasData feed as in Chapter 1)
!pip install yfinance
import yfinance as yf
ticker = 'AAPL'
start_date = '2020-01-01'
end_date = '2023-12-31'
dataframe = yf.download(ticker, start=start_date, end=end_date)
dataframe.columns = dataframe.columns.droplevel(1)
data = bt.feeds.PandasData(dataname=dataframe)

# --- Cerebro Setup ---
cerebro = bt.Cerebro()

# Add the strategy - we can override parameters here
cerebro.addstrategy(
    MyStrategyWithIndicators,
    sma_period=25,          # Override default SMA period
    rsi_period=10           # Override default RSI period
    # rsi_sma_period = 7    # Override if RSI_SMA is active
)
print("Strategy MyStrategyWithIndicators added to Cerebro.")

# Add the data feed
cerebro.adddata(data)
print("Data feed added.")

# Configure broker
```

```
cerebro.broker.setcash(10000.0)
cerebro.broker.setcommission(commission=0.001)
print("Broker configured (Cash: $10,000.00, Commission: 0.1%).")

# Run the backtest
print(f'\nStarting Portfolio Value: {cerebro.broker.getvalue():,.2f}')
results = cerebro.run()
print(f'Final Portfolio Value: {cerebro.broker.getvalue():,.2f}')

# --- Plotting ---
# Ensure you run '%matplotlib inline' in a separate cell before this in
Colab/Jupyter
import matplotlib
%matplotlib inline
print("\nGenerating plot...")
try:
    cerebro.plot(style='candlestick', barup='green', bardown='red',
volume=True, iplot=False)
    print("Plot displayed/saved.")
except Exception as e:
    print(f"Could not plot results: {e}")
```

```
[**********************100%***********************]  1 of 1 completed
Strategy MyStrategyWithIndicators added to Cerebro.
Data feed added.
Broker configured (Cash: $10,000.00, Commission: 0.1%).

Starting Portfolio Value: 10,000.00
2023-12-29 - --- Strategy MyStrategyWithIndicators Initializing ---
2023-12-29 - Data feed reference stored.
2023-12-29 - SMA indicator created with period 25
2023-12-29 - RSI indicator created with period 10
2023-12-29 - --- Strategy MyStrategyWithIndicators Initialized ---
2020-02-28 - Bar: 40, Close: 66.34, SMA(25): 75.81, RSI(10): 23.14
2020-03-27 - Bar: 60, Close: 60.12, SMA(25): 64.95, RSI(10): 44.66
2020-04-27 - Bar: 80, Close: 68.72, SMA(25): 63.99, RSI(10): 59.38
2020-05-26 - Bar: 100, Close: 77.07, SMA(25): 72.82, RSI(10): 65.09
2020-06-23 - Bar: 120, Close: 89.19, SMA(25): 81.19, RSI(10): 77.42
2020-07-22 - Bar: 140, Close: 94.68, SMA(25): 90.61, RSI(10): 65.64
2020-08-19 - Bar: 160, Close: 112.83, SMA(25): 102.12, RSI(10): 72.81
2020-09-17 - Bar: 180, Close: 107.59, SMA(25): 116.66, RSI(10): 38.07
2020-10-15 - Bar: 200, Close: 117.70, SMA(25): 111.57, RSI(10): 58.46
2020-11-12 - Bar: 220, Close: 116.44, SMA(25): 113.71, RSI(10): 58.29
2020-12-11 - Bar: 240, Close: 119.57, SMA(25): 116.82, RSI(10): 57.10
2021-01-12 - Bar: 260, Close: 125.81, SMA(25): 125.70, RSI(10): 48.90
2021-02-10 - Bar: 280, Close: 132.44, SMA(25): 131.03, RSI(10): 51.26
2021-03-11 - Bar: 300, Close: 119.31, SMA(25): 124.89, RSI(10): 44.25
2021-04-09 - Bar: 320, Close: 130.11, SMA(25): 120.30, RSI(10): 74.31
2021-05-07 - Bar: 340, Close: 127.59, SMA(25): 128.88, RSI(10): 48.26
2021-06-07 - Bar: 360, Close: 123.37, SMA(25): 123.84, RSI(10): 49.10
2021-07-06 - Bar: 380, Close: 139.17, SMA(25): 128.50, RSI(10): 84.80
2021-08-03 - Bar: 400, Close: 144.40, SMA(25): 141.66, RSI(10): 60.09
2021-08-31 - Bar: 420, Close: 149.00, SMA(25): 145.04, RSI(10): 62.31
2021-09-29 - Bar: 440, Close: 140.17, SMA(25): 146.12, RSI(10): 36.16
```

```
2021-10-27 - Bar: 460, Close: 146.08, SMA(25): 142.10, RSI(10): 62.33
2021-11-24 - Bar: 480, Close: 159.16, SMA(25): 149.46, RSI(10): 82.10
2021-12-23 - Bar: 500, Close: 173.25, SMA(25): 165.45, RSI(10): 62.27
2022-01-24 - Bar: 520, Close: 158.84, SMA(25): 170.38, RSI(10): 27.12
2022-02-22 - Bar: 540, Close: 161.70, SMA(25): 166.09, RSI(10): 36.58
2022-03-22 - Bar: 560, Close: 166.13, SMA(25): 160.39, RSI(10): 61.71
2022-04-20 - Bar: 580, Close: 164.56, SMA(25): 167.49, RSI(10): 45.80
2022-05-18 - Bar: 600, Close: 138.78, SMA(25): 155.05, RSI(10): 33.54
2022-06-16 - Bar: 620, Close: 128.17, SMA(25): 140.51, RSI(10): 31.08
2022-07-18 - Bar: 640, Close: 144.94, SMA(25): 138.12, RSI(10): 57.67
2022-08-15 - Bar: 660, Close: 170.92, SMA(25): 156.30, RSI(10): 79.96
2022-09-13 - Bar: 680, Close: 151.82, SMA(25): 162.16, RSI(10): 38.81
2022-10-11 - Bar: 700, Close: 137.16, SMA(25): 147.82, RSI(10): 32.48
2022-11-08 - Bar: 720, Close: 137.90, SMA(25): 142.46, RSI(10): 40.45
2022-12-07 - Bar: 740, Close: 139.32, SMA(25): 143.85, RSI(10): 39.03
2023-01-06 - Bar: 760, Close: 128.13, SMA(25): 134.50, RSI(10): 43.72
2023-02-06 - Bar: 780, Close: 149.99, SMA(25): 136.33, RSI(10): 70.97
2023-03-07 - Bar: 800, Close: 150.09, SMA(25): 148.84, RSI(10): 56.58
2023-04-04 - Bar: 820, Close: 163.98, SMA(25): 154.28, RSI(10): 72.76
2023-05-03 - Bar: 840, Close: 165.78, SMA(25): 163.53, RSI(10): 57.60
2023-06-01 - Bar: 860, Close: 178.54, SMA(25): 171.05, RSI(10): 75.24
2023-06-30 - Bar: 880, Close: 192.30, SMA(25): 181.34, RSI(10): 80.67
2023-07-31 - Bar: 900, Close: 194.76, SMA(25): 190.12, RSI(10): 67.48
2023-08-28 - Bar: 920, Close: 178.88, SMA(25): 181.69, RSI(10): 49.80
2023-09-26 - Bar: 940, Close: 170.71, SMA(25): 177.93, RSI(10): 36.28
2023-10-24 - Bar: 960, Close: 172.18, SMA(25): 173.93, RSI(10): 39.28
2023-11-21 - Bar: 980, Close: 189.50, SMA(25): 178.00, RSI(10): 74.75
2023-12-20 - Bar: 1000, Close: 193.67, SMA(25): 191.49, RSI(10): 55.22
Final Portfolio Value: 10,000.00

Generating plot...
```




```
Plot displayed/saved.
```

Note how we can easily change the `sma_period` and `rsi_period` when calling `cerebro.addstrategy()` without altering the `MyStrategyWithIndicators` class definition itself. This makes testing different parameter values very efficient.

Interpreting the Output and Plot

When you run the script integrating `MyStrategyWithIndicators`, you'll observe a few things:

1. Console Output:

- You will see the log messages from the strategy's `__init__` method printed once at the beginning, confirming the indicator creation and periods used (reflecting any overrides from `addstrategy`).
- As the backtest runs, you'll see the periodic log messages from the `next()` method, showing the calculated Close, SMA, and RSI values for specific bars (every 20 bars by default in our example). This helps verify that the indicators are indeed being calculated.
- Finally, the starting and final portfolio values will be printed. Since this strategy doesn't trade, the final value should be very close to the starting value, perhaps slightly lower due to potential (though unlikely here) minor broker/cash handling effects over time, but certainly not reflecting any trading profit or loss.

2. The Plot:

- This is where the indicators truly come to life visually. Assuming the plotting works correctly (using `%matplotlib inline` in notebooks), you should see a multi-panel chart:
    - **Top Panel:** The main price chart (e.g., candlesticks for AAPL). Crucially, the **SMA line** (e.g., the 25-period SMA if we used the override) will be overlaid directly onto the price data. You can clearly see how it smooths the price action and how the price moves above and below it.
    - **Second Panel:** Below the price chart, a separate panel will display the **RSI indicator** (e.g., the 10-period RSI). You'll see its line oscillating primarily between 0 and 100. You can visually track when it moves into potential overbought (>70) or oversold (<30) regions relative to the price action above.
    - **Third Panel (Optional):** If you had uncommented the `self.rsi_sma` lines in the strategy, `backtrader` would likely plot this smoothed RSI line. It might appear in the *same* panel as the raw RSI (since they share a similar scale), allowing you to see the raw RSI crossing its own moving average. Alternatively, depending on internal logic, it might get its own panel.
    - **Volume Panel:** Typically, a panel showing trading volume bars is also included at the bottom.

This plot is essential. It confirms that `backtrader` has correctly calculated and displayed the indicators defined in your strategy. It allows you to visually correlate the indicator movements (SMA slope, RSI levels, crossovers) with the price action, building intuition for how these tools work and how they might be used to generate the trading signals we'll implement in later chapters.

Chapter Summary

In this chapter, we took our first deep dive into `backtrader`'s powerful indicator system. We explored two foundational indicators:

- **Simple Moving Average (SMA):** A trend-following tool used for smoothing price and identifying trend direction via its slope and crossovers.
- **Relative Strength Index (RSI):** A momentum oscillator used to identify potential overbought and oversold conditions.

We learned how to instantiate these indicators within a strategy's `__init__` method using `bt.indicators` (or `bt.ind`), how to configure their `period` parameter, and how to access their calculated values (`[0]`, `[-1]`, etc.) within the `next()` method. We also saw the concept of chaining indicators, demonstrated by applying an SMA to the RSI output.

Finally, the hands-on walkthrough of `MyStrategyWithIndicators` showed these concepts integrated into a full strategy class and Cerebro workflow, culminating in a plot that visually confirms the indicators' behavior.

With this understanding of how to implement and access basic indicators, we are now ready to explore more advanced indicators and begin combining their signals to build actual trading logic in the upcoming chapters.