12

Deploy Strategies to a Live Environment

In ***Chapter 10***, *Set up the Interactive Brokers Python API*, and ***Chapter 11***, *Manage Orders, Positions, and Portfolios with the IB API*, we set the stage to begin deploying algorithmic trading strategies into a live (or paper trading) environment. Before we get there, we need two more critical pieces of the algorithmic trading puzzle: risk and performance metrics and more sophisticated order strategies that allow us to build and rebalance asset portfolios. For risk and performance metrics, we will introduce the **Empyrical Reloaded** library, which generates statistics based on portfolio returns. `empyrical-reloaded` is the library that provides the performance and risk analytics behind **Pyfolio Reloaded**, which we learned about in ***Chapter 9***, *Assess Backtest Risk and Performance Metrics with Pyfolio*. In this chapter, we'll use `empyrical-reloaded` to calculate key performance indicators such as the Sharpe ratio, Sortino ratio, and the maximum drawdown, among others, using real-time portfolio return data. To allow us to compute real-time return data while executing trades or other code in our trading app, we'll learn how to run code asynchronously on a thread.

In addition to calculating risk and performance metrics, we will finalize our position management code by introducing methods to submit orders based on a target number of contracts, monetary value, or percentage allocation. This extended functionality unlocks portfolio-based strategies as opposed to only single-asset strategies. Finally, we will introduce three algorithmic trading strategies that you can deploy right away: a monthly factor-based strategy using the `zipline-reloaded` pipeline API that we learned about in ***Chapter 7***, *Event-Based Backtesting FactorPortfolios with Zipline Reloaded*, an options combo strategy, and an intraday multi-asset

mean-reversion strategy. These strategies take advantage of the code we've built throughout this book.

This chapter contains the following recipes:

- Calculating real-time key performance and risk indicators
- Sending orders based on portfolio targets
- Deploying a monthly factor portfolio strategy
- Deploying an options combo strategy
- Deploying an intraday multi-asset mean reversion strategy

Calculating real-time key performance and risk indicators

Real-time performance and risk metrics are important for maintaining robust trading strategies. They allow us to compare real-life performance to the performance of our backtests. They provide immediate feedback on the effectiveness of our trading algorithms and let us make adjustments in response to market volatility or unexpected events. By continuously monitoring risk metrics such as drawdowns, volatility, and value at risk, we can effectively manage exposure and mitigate potential losses. Most professional algorithmic traders spend their time analyzing and explaining deviations from the performance that they expect in their backtests to the performance that they observe during live trading. This recipe will introduce the tools we need to do the same.

Getting ready

We'll use **`empyrical-reloaded`** to compute performance and risk statistics. To install it, use **`pip`**:

```
pip install empyrical-reloaded
```

At the top of **`app.py`**, import **`Empyrical Reloaded`**:

```
import empyrical as ep
```

To calculate real-time performance and risk indicators, we need periodic portfolio returns. Unfortunately, the IB API does not provide a way to re-

trieve portfolio returns, so we'll need to build our own method. We'll do this by periodically requesting account PnL and then using it to compute returns. To allow the rest of our trading app to run independently of the PnL calculations, we'll run the method on its own thread.

Add a new instance variable to **`wrapper.py`**, which we'll use to capture account PnL. At the end of the **`__init__`** method in the **`IBWrapper`** class, add the following:

```
self.portfolio_returns = None
```

Add a method to **`client.py`** that continuously requests account PnL from the **`get_pnl`** method we built in ***Chapter 9***:

```
def get_streaming_pnl(self, request_id, interval=60,
    pnl_type="unrealized_pnl"):
        interval = max(interval, 5) - 2
        while True:
            pnl = self.get_pnl(request_id=request_id)
            yield {"date": pd.Timestamp.now(),
                "pnl": pnl[request_id].get(pnl_type)}
        time.sleep(interval)
```

The method first ensures that the interval between data retrievals is at least three seconds. This accounts for the three-second update time from IB and the two-second wait time in **`get_pnl`**. In an infinite loop, it retrieves the PnL data and then yields a dictionary containing the current timestamp and the specified type of PnL before pausing for the defined interval duration.

Add a method to stream portfolio PnL and compute the associated returns:

```
def get_streaming_returns(self, request_id, interval,
    pnl_type):
        returns = pd.Series(dtype=float)
        for snapshot in self.get_streaming_pnl(
            request_id=request_id,
            interval=interval,
            pnl_type=pnl_type
        ):
            returns.loc[snapshot["date"]] = snapshot["pnl"]
            if len(returns) > 1:
```

```
                self.portfolio_returns = (
                    returns
                    .pct_change()
                    .dropna()
                )
```

The **`get_streaming_returns`** method calculates the periodic PnL. It initializes a pandas Series to store PnL values, then iterates over PnL data snapshots obtained from **`get_streaming_pnl`**, adding each PnL value to the **`returns`** Series indexed by the snapshot's timestamp. If the Series has more than one entry, it calculates the percentage change in PnL, updating the **`portfolio_returns`** attribute with these calculated returns after dropping the **`NaN`** values.

In **`wrapper.py`**, add the following instance variable at the end of the **`__init__`** method:

```
self.portfolio_returns = None
```

In **`app.py`**, update the **`__init__`** method so it resembles the following:

```
def __init__(self, ip, port, client_id, account,
    interval=5):
        IBWrapper.__init__(self)
        IBClient.__init__(self, wrapper=self)
        self.account = account
        self.create_table()
        self.connect(ip, port, client_id)
        threading.Thread(
            target=self.run, daemon=True).start()
        time.sleep(2)
        threading.Thread(
            target=self.get_streaming_returns,
            args=(99, interval, "unrealized_pnl"),
            daemon=True
        ).start()
```

Upon instantiation of our **`IBApp`** class, we start two separate threads: one to continuously run the main event loop of the API client and another to stream unrealized PnL returns every five seconds, both beginning their execution after a two-second pause to ensure that the connection is established. This allows us to continue to access the periodic portfolio returns while executing other code.

How to do it...

We'll implement six methods (decorated as properties) to compute performance and risk metrics in our **`IBApp`** class using **`empyrical-reloaded`**. You should add these methods after the **`stream_to_sql`** method we added in ***Chapter 11***, *Manage Orders, Positions, and Portfolios with the IB API*.

1. Add a method to compute cumulative returns:

```
@property
def cumulative_returns(self):
    return ep.cum_returns(self.portfolio_returns, 1)
```

2. Add a method to compute the maximum drawdown:

```
@property
def max_drawdown(self):
    return ep.max_drawdown(self.portfolio_returns)
```

3. Add a method to compute the volatility of returns:

```
@property
def volatility(self):
    return self.portfolio_returns.std(ddof=1)
```

4. Add a method to compute the Omega ratio:

```
@property
def omega_ratio(self):
    return ep.omega_ratio(self.portfolio_returns,
        annualization=1)
```

5. Add a method to compute the Sharpe ratio:

```
@property
def sharpe_ratio(self):
    return self.portfolio_returns.mean() / self.portfolio_returns.std(ddof=1)
```

6. Add a method to compute the percentage and dollar conditional value at risk:

```
@property
def cvar(self):
    net_liquidation = self.get_account_values(
        "NetLiquidation")[0]
    cvar_ = ep.conditional_value_at_risk(
        self.portfolio_returns)
    return (
        cvar_,
```

```
            cvar_ * net_liquidation
        )
```

How it works…

We'll cover the details of each method in detail in the following sub-sections.

Cumulative returns

To get the cumulative returns of our portfolio, we add the following in the **`app.py`** file after defining our trading app:

```
app.cumulative_returns
```

The **`cumulative_returns`** method computes the cumulative returns of our portfolio returns. The first argument, **`returns`**, is a pandas Series representing the periodic returns of our portfolio. The second argument, **`1`**, specifies the starting value for the cumulative returns calculation. This function effectively calculates the compounded return at each point in time, assuming that the initial investment value is **`1`**. We can use this method to generate the following intraday equity curve by plotting the data from the resulting Series.

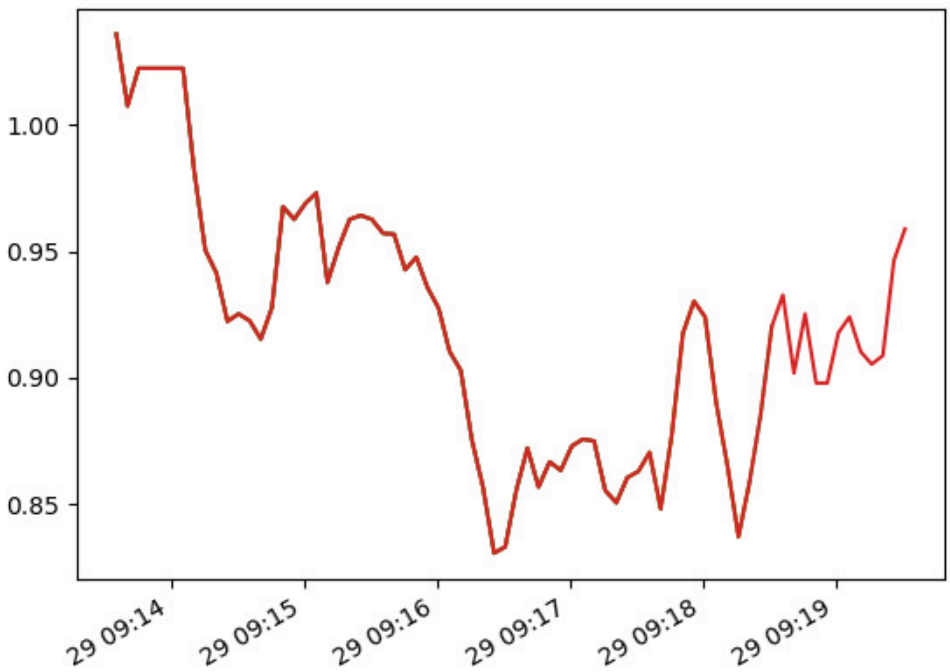

Figure 12.1: An intraday equity curve computed every five seconds based on cumulative returns

Max drawdown

To get the max drawdown of our portfolio, we add the following in the **`app.py`** file after defining our trading app:

```
app.max_drawdown
```

The **`max_drawdown`** method computes the maximum drawdown of our portfolio. The maximum drawdown is calculated by determining the largest decrease in value from a portfolio's peak to its lowest point before it reaches a new peak. It's typically expressed as a negative percentage. This metric is crucial in risk management and performance evaluation in algorithmic trading, as it provides insight into the potential downside risk of an investment strategy.

Volatility

To get the volatility of our portfolio, we add the following in the **`app.py`** file after defining our trading app:

```
app.volatility
```

The **`volatility`** method calculates the sample standard deviation of portfolio returns. This measure provides insight into the volatility or risk associated with the portfolio over the time period covered by the data. By setting **`ddof`** to **`1`**, the calculation adjusts for bias in the standard deviation estimate, making it more accurate for a sample (such as a subset of returns over a specific period) rather than the entire population of returns.

> *IMPORTANT NOTE*
>
> *We don't use the* **`empyrical-reloaded`** *method in this example because it assumes daily returns. When we compute volatility using* **`empyrical-reloaded`***, it annualizes the volatility metric, which is not appropriate for intraday values.*

Omega ratio

To get the Omega ratio of our portfolio, we add the following in the **app.py** file after defining our trading app:

```
app.omega_ratio
```

The **omega_ratio** method calculates the Omega ratio for our portfolio returns. The Omega ratio is a risk-adjusted return measure of an investment asset, portfolio, or strategy, comparing the probability of achieving a threshold return level to the probability of falling below it. It's calculated by dividing the sum of excess returns above a threshold (**0** in our case) by the absolute value of the sum of returns below that threshold, where excess returns are annualized if an **annualization** factor is provided (which it is not in our case). In trading, this ratio is particularly useful as it provides a more comprehensive view of the risk-reward tradeoff than traditional measures such as the Sharpe ratio, especially in portfolios with non-normal return distributions.

Sharpe ratio

To get the Sharpe ratio of our portfolio, we add the following in the **app.py** file after defining our trading app:

```
app.sharpe_ratio
```

The **sharpe_ratio** method computes the Sharpe ratio for our portfolio returns. Typically, the **sharpe_ratio** method is computed by first calculating the difference between the returns of the portfolio and the risk-free rate. This excess return is then annualized by multiplying it by the square root of the **annualization** parameter. The annualized excess return is then divided by the standard deviation of the portfolio's returns, which is a measure of the portfolio's risk. Since we are using intraday returns, we won't use **empyrical-reloaded** because it assumes daily returns.

Conditional value at risk

To get the conditional value at risk of our portfolio, we add the following in the **app.py** file after defining our trading app:

```
app.cvar
```

The **Conditional Value at Risk (CVaR)**, also known as the **Expected Shortfall (ES)**, is a risk assessment metric that estimates the expected loss in our portfolio under extreme market conditions, focusing on the tail end of the distribution of potential losses. The method first retrieves the `NetLiquidation` value of the account, which represents the total value of the portfolio if all positions were liquidated at current market prices. It then calculates the CVaR of the account's returns using `empyrical-reloaded`'s `conditional_value_at_risk` function, which computes CVaR as the average of the worst-performing returns (those in the tail of the distribution) below a 5% percentile. The method returns two values: the CVaR as a percentage and the CVaR in terms of the actual monetary value, calculated by multiplying the CVaR percentage by the portfolio's net liquidation value.

There's more...

There are more than 40 risk and performance metrics available in `Empyrcal Reloaded`, including rolling metrics and those that compare portfolio returns to a benchmark. Here are a few more you can explore:

- `calmar_ratio`: Ratio of annualized return over maximum drawdown, assessing returns relative to downside risks
- `downside_risk`: Calculates the risk of negative returns, aiding in understanding a strategy's downside volatility
- `sortino_ratio`: Similar to Sharpe, but this focuses only on downside risk and is useful for strategies with asymmetric risk
- `stability_of_timeseries`: Measures consistency of returns; important for assessing strategy reliability over time
- `tail_ratio`: Compares the right (positive) versus left (negative) tail of a distribution, indicating extreme outcome frequency
- `value_at_risk`: Estimates the maximum potential loss over a specified time frame and is crucial for risk assessment
- `excess_sharpe`: Calculates the difference in Sharpe ratio relative to a benchmark; useful for strategy comparison

See also

To learn more about `empyrical-reloaded`, check out the documentation here: **https://empyrical.ml4trading.io/**

Sending orders based on portfolio targets

We now have the components of our trading app built to add flexibility to the way we submit orders. By combining real-time position data and portfolio net liquidation value, we can build more sophisticated order techniques. For example, now that we can access current positions, we're able to dynamically adjust our positions to align with quantity or value targets. Similarly, with live net liquidation value, we can calculate order sizes as a percentage of the portfolio. Building orders based on portfolio percentage targets unlocks advanced portfolio and risk management capabilities. This integration results in a more responsive trading system that is capable of adapting to market changes swiftly and executing orders that are consistently in tune with our overall risk management and investment objectives. In this recipe, we'll implement methods to submit orders based on target values, quantities, and percentage allocations.

Getting ready

We will assume that you've created the **`client.py`** and **`app.py`** files in the **`trading-app`** directory. If you have not, do it now.

We'll start by adding a default market data request ID to the **`utils.py`** file. Since we cancel market data requests immediately after requesting the data, we can use the same ID.

Open **`utils.py`** and add the following code after the **`TABLE_BAR_PROPERTIES`** list:

```
DEFAULT_MARKET_DATA_ID = 55
```

How to do it...

We'll create five separate order methods and three associated helper methods. The order methods will generate the number of contracts to order so that we can achieve the desired portfolio allocation. The helper methods will do the work of calculating the quantities. Add all of the following code under the **`send_order`** method in the **`IBClient`** class in the **`client.py`** file.

1. Add the method that computes the quantity and places an order for a fixed amount of money:

```
def order_value(self, contract, order_type, value, **kwargs):
    quantity = self._calculate_order_value_quantity(
        contract, value)
    order = order_type(quantity=quantity, **kwargs)
    return self.send_order(contract, order)
```

2. Add the method to place an order to adjust a position to the target number of contracts:

```
def order_target_quantity(self, contract, order_type,
    target, **kwargs):
        quantity=self._calculate_order_target_quantity(contract,            tar
        order = order_type(
            action=SELL if quantity < 0 else BUY,
            quantity=abs(quantity),
            **kwargs
        )
        return self.send_order(contract, order)
```

3. Add the helper method that loops through the positions and computes the target number of contracts to order:

```
def _calculate_order_target_quantity(self, contract,
    target):
        positions = self.get_positions()
        if contract.symbol in positions.keys():
            current_position = positions[
                contract.symbol]["position"]
            target -= current_position
        return int(target)
```

4. Add the method to order the specified asset according to the percent of the current portfolio value:

```
def order_percent(self, contract, order_type, percent,
    **kwargs):
        quantity = self._calculate_order_percent_quantity(
            contract, percent)
        order = order_type(quantity=quantity,
            **kwargs)
        return self.send_order(contract, order)
```

5. Add the associated helper function that computes the percentage of the portfolio based on the net liquidation value and the number of contracts to order:

```
def _calculate_order_percent_quantity(self, contract, percent):
    net_liquidation_value = self.get_account_values(
        key="NetLiquidation")[0]
    value = net_liquidation_value * percent
    return self._calculate_order_value_quantity(contract, value)
```

6. Add the method to adjust a position to a target dollar value. If the position doesn't already exist, the code sends a new order. If the position exists, the code sends an order for the difference between the target value and the current position value:

```
def order_target_value(self, contract, order_type,
    target, **kwargs):
    target_quantity = self._calculate_order_value_quantity(
        contract, target)
    quantity = self._calculate_order_target_quantity(contract,
        target_quantity)
    order = order_type(
        action=SELL if quantity < 0 else BUY,
        quantity=abs(quantity),
        **kwargs
    )
    return self.send_order(contract, order)
```

7. Add the associated helper method:

```
def _calculate_order_value_quantity(self, contract,
    value):
        last_price = self.get_market_data(
            request_id=DEFAULT_MARKET_DATA_ID,
                contract=contract, tick_type=4)
        multiplier = contract.multiplier if contract.multiplier
            != "" else 1
        return int(value / (last_price * multiplier))
```

8. Add the method to send an order to update a position to a percent of the portfolio value. If the position doesn't already exist, the code sends a new order. If the position exists, the code sends an order for the difference between the target percent and the current portfolio percent:

```
def order_target_percent(self, contract, order_type,
    target, **kwargs):
    quantity = self._calculate_order_target_percent_quantity(
        contract, target)
    order = order_type(
        action=SELL if quantity < 0 else BUY,
        quantity=abs(quantity),
```

```
            **kwargs
        )
        return self.send_order(contract, order)
```

9. Add the associated helper method:

```
def _calculate_order_target_percent_quantity(self,
    contract, target):
    target_quantity = self._calculate_order_percent_quantity(
        contract, target)
    return self._calculate_order_target_quantity(
        contract, target_quantity)
```

How it works…

Our order methods take two required arguments and various keyword arguments based on the order type that's passed. The two required arguments are the contract to trade and the order type to use. The keyword arguments are additional arguments to be passed to the order.

Ordering for a fixed amount of money with order_value

The **`order_value`** method places an order for a specific monetary value rather than a specified quantity of contracts. It calculates the quantity of the asset to be ordered by calling the **`_calculate_order_value_quantity`** method, which determines the quantity based on the current market price of the asset and its multiplier. This calculation divides the specified dollar value by the product of the asset's last market price and its multiplier (defaulting to **`1`** if none are specified).

The method then creates an order of the given **`order_type`** object with the calculated quantity, along with any additional parameters passed via **`**kwargs`**. Finally, it places the order by invoking **`send_order`** with the contract and the order.

Ordering to adjust a position to the target number of shares with order_target_quantity

The **`order_target_quantity`** method adjusts a position to a specified target quantity of contracts. It calculates the quantity by determining the difference between the target quantity and the current position size using

the **_calculate_order_target_quantity** method. If the contract already has an existing position, this method adjusts the position to the target by calculating the difference. If no position exists, it treats the target as the total quantity for a new order.

The method then creates an order of the specified **order_type** object, setting the action to **BUY** or **SELL** based on whether the calculated quantity is positive or negative. It uses the absolute value of this quantity. Finally, it places the order by calling **send_order** with the contract and the order.

Ordering to a given percent of current portfolio value with order_percent

The **order_percent** method uses the **_calculate_order_percent_quantity** helper method, which multiplies the portfolio's net liquidation value by the specified percentage, then converts this value into the appropriate quantity of the asset using **_calculate_order_value_quantity**.

The method then creates an order of the specified **order_type** object with this calculated quantity (and any additional parameters passed via ****kwargs**). Finally, it places the order by calling **send_order** with the contract and the newly created order.

Ordering to adjust a position to a target value with order_target_value

The **order_target_value** method adjusts a position to a specified target monetary value for a contract. It first calculates the target quantity of contracts needed to reach the target value using the **_calculate_order_value_quantity** method, which determines the quantity based on the current market price and the contract's multiplier. Then, it calculates the actual quantity to order, taking into account the current position, by using the **_calculate_order_target_quantity** method. This method computes the difference between the target quantity and the current position, treating it as a new order if no position exists or as an adjustment if a position already exists.

The method then creates an order of the specified **order_type** object, setting the action to **BUY** or **SELL** based on whether the calculated quantity is

positive or negative, and uses the absolute value of this quantity. Finally, the method places the order by calling **`send_order`** with the contract and the order.

Ordering to adjust a position to a target percent of the current portfolio value with order_target_percent

The **`order_target_percent`** method adjusts a position so that it represents a specific target percentage of the current portfolio value. This method first calculates the target quantity of the asset needed to achieve the desired portfolio percentage using the **`_calculate_order_target_percent_quantity`** method. This calculation takes the current value of the portfolio and the target percentage into account, adjusting the position size accordingly. If the position in the specified contract already exists, the method calculates the difference between the current and target percentages and adjusts the position size to match the target.

The method then creates an order of the specified **`order_type`** object, determining whether to buy or sell based on whether the calculated quantity is positive or negative, and sets the quantity to the absolute value of the calculated amount. Finally, the method places the order by calling **`send_order`** with the contract and the order.

There's more...

The following examples show how to use the new order methods.

1. In **`app.py`**, add the following imports at the top of the file:
   ```
   from order import market, limit, BUY, SELL
   ```
2. In the main section of the file under the instantiation of the trading app, define an AAPL contract:
   ```
   aapl = stock("AAPL", "SMART", "USD")
   ```
3. Submit a **`market`** order to buy $1,000 worth of AAPL:
   ```
   app.order_value(aapl, market, 1000, action=BUY)
   ```
4. Submit a **`market`** order to adjust the portfolio to be short five shares of AAPL:

```
app.order_target_quantity(aapl, market, -5)
```

5. Submit a **`limit`** order for AAPL for an equivalent of 10% of the net liquidation value of the portfolio:

```
app.order_percent(aapl, limit, 0.1, action=BUY,
    limit_price=185.0)
```

6. Submit a **`stop loss`** order to adjust the portfolio to $3,000 worth of AAPL:

```
app.order_target_value(aapl, stop, 3000,
    stop_price=180.0)
```

7. Submit a **`market`** order to adjust the portfolio so that 50% of the net liquidation value is allocated to AAPL:

```
app.order_target_percent(aapl, market, 0.5)
```

See also

The logic to compute the target values and percentages is based on the **`zipline-reloaded`** implementation for backtesting. You can review the API reference for details at **https://zipline.ml4trading.io/api-reference.html**.

Deploying a monthly factor portfolio strategy

We'll now integrate the momentum factor we built in ***Chapter 5***, *Build Alpha Factors for Stock Portfolios*, into our trading app. The app is designed to download and process premium U.S. equities data encompassing a comprehensive universe of approximately 20,000 stocks. The advantage of using the premium data is that it lets us build factor portfolios that include the entire universe of U.S.-traded equities.

The trading app is designed to be run on a periodic rebalancing schedule after market hours, typically monthly. Each time it runs, it acquires the latest price data for the entire stock universe. It then computes the momentum factor for these stocks. Based on this computation, the app identifies the top stocks exhibiting the strongest momentum and the bottom

stocks showing the weakest momentum. The trading strategy involves going long on the top stocks and short on the bottom stocks.

Our trading app can execute orders that align with a specific target percentage allocation for each stock in the portfolio. This feature simplifies the process of maintaining the desired portfolio balance, requiring only a single line of code to buy or sell shares as needed. This approach lets us rebalance the portfolio so it consistently reflects the target portfolio adjusted for the computed momentum factor.Top of Form

Bottom of Form

Getting ready

In ***Chapter 5***, *Build Alpha Factors for Stock Portfolios*, we explained how to set up an account with Nasdaq Data Link. For this recipe, we'll rely on a paid subscription to **QuoteMedia's End of Day US Stock Prices**. If you decide against the paid subscription, you can use the free data described in ***Chapter 7***, *Event-Based Backtesting Factor Portfolios with Zipline Reloaded*. QuoteMedia offers end-of-day prices, dividends, adjustments, and splits for U.S. publicly traded stocks with price history dating back to 1996. The product covers all stocks with a primary listing on NASDAQ, AMEX, NYSE, and ARCA. You can find the page to subscribe at **https://data.nasdaq.com/databases/EOD/data**. Once you are subscribed, you'll be able to use the data through the same API key we set up in ***Chapter 1***, *Acquire Free Financial Market Data with Cutting-edge Python Libraries*.

To ingest the data to build our factor portfolio, we'll need to set up some custom files. Please visit the following URL to get the instructions:

https://pyquantnews.com/ingest-premium-market-data-with-zipline-reloaded/

After you have set up the custom extensions that we have linked, create a file named **`.env`** in the same directory as your **`app.py`** file. In this file, add the following line:

```
DATALINK_API_KEY=<YOUR-API-KEY>
```

Replace **<YOUR-API-KEY>** with your Nasdaq Data Link API key, which you acquired in ***Chapter 5***, *Build Alpha Factors for Stock Portfolios*.

How to do it…

We'll integrate the code we built in ***Chapter 5***, *Build Alpha Factors for Stock Portfolios* into our trading app.

1. In **app.py**, add the following imports to the top of the file and reorder them to adhere to PEP8 specifications:

```
import os
import sqlite3
import threading
import time
import empyrical as ep
import exchange_calendars as xcals
import numpy as np
import pandas as pd
from dotenv import load_dotenv
from zipline.data import bundles
from zipline.data.bundles.core import load
from zipline.pipeline import Pipeline
from zipline.pipeline.data import USEquityPricing
from zipline.pipeline.engine import SimplePipelineEngine
from zipline.pipeline.factors import CustomFactor, Returns
from zipline.pipeline.loaders import USEquityPricingLoader
from zipline.utils.run_algo import load_extensions
from client import IBClient
from contract import stock
from order import market
from wrapper import IBWrapper
load_dotenv()
```

2. Below our **IBApp** class, add the custom momentum factor that **zipline-reloaded** uses to compute the factor value for each asset in our universe:

```
class MomentumFactor(CustomFactor):
    inputs = [USEquityPricing.close, Returns(window_length=126)]
    window_length = 252
    def compute(self, today, assets, out, prices,
        returns):
            out[:] = (
                (prices[-21] - prices[-252]) / prices[-252]
```

```
                - (prices[-1] - prices[-21]) / prices[-21]
            ) / np.nanstd(returns, axis=0)
```

3. Now add a function that creates and returns a **`zipline-reloaded`** pipeline. The pipeline is responsible for creating a DataFrame with the data that we'll use to select which assets to trade:

```
def make_pipeline():
    momentum = MomentumFactor()
    return Pipeline(
        columns={
            "factor": momentum,
            "longs": momentum.top(top_n),
            "shorts": momentum.bottom(top_n),
            "rank": momentum.rank(ascending=False),
        },
    )
```

4. Now, we'll add the code to generate the weights required to rebalance our portfolio. You'll need to add all of the following code in the main section of the app under the **`app = IBApp("127.0.0.1", 7497, client_id=11, account="DU*******")`** line. Start with creating a session calendar based on the **New York Stock Exchange** to set the look-back time for computing the momentum. The **`top_n`** variable determines how many long and short positions we want in our portfolio:

```
top_n = 10
xnys = xcals.get_calendar("XNYS")
today = pd.Timestamp.today().strftime("%Y-%m-%d")
start_date = xnys.session_offset(today,
    count=-252).strftime("%Y-%m-%d")
```

5. Add the code to load the extension file that we created in the *Getting ready* section of this recipe. After the extension file has been loaded, we can invoke the code to ingest our data bundle using the premium data subscription. Finally, we will load the data bundle into memory so it's ready to use:

```
load_extensions(True, [], False, os.environ)
bundles.ingest("quotemedia")
bundle_data = load("quotemedia", os.environ, None)
```

   The ingestion can take around 30 minutes once run, so be patient. You should see output resembling the following once the bundling process begins:

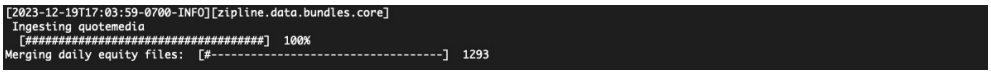

Figure 12.2: The data ingestion process output to the console

6. Add the code that creates the U.S. equity pricing loader, which aligns the price data to U.S. exchange calendars:

```
pipeline_loader = USEquityPricingLoader(
    bundle_data.equity_daily_bar_reader,
    bundle_data.adjustment_reader,
    fx_reader=None
)
```

7. Create the engine that brings the pipeline loader and the bundled data together:

```
engine = SimplePipelineEngine(
    get_loader=lambda col: pipeline_loader,
    asset_finder=bundle_data.asset_finder
)
```

8. Add the code that runs the pipeline and generates the output:

```
results = engine.run_pipeline(
    make_pipeline(),
    start_date,
    today
)
```

9. Clean up the resulting DataFrame by removing records with no factor data, adding names to the MultiIndexes, and sorting the values first by date, then by factor value:

```
results.dropna(subset="factor", inplace=True)
results.index.names = ["date", "symbol"]
results.sort_values(by=["date", "factor"],
    inplace=True)
```

The result is a MultiIndex DataFrame including the raw factor value, boolean values indicating a short or long position, and details on how the stock's factor value is ranked among the universe. The last date in the DataFrame should reflect the last trading day which will allow us to use the results to rebalance our portfolio:

		factor	longs	shorts	rank
date	**symbol**				
2022-12-14	**Equity(20942 [XBIOW])**	-109.975559	False	True	10339.0
	Equity(16489 [ROCGW])	-65.102756	False	True	10338.0
	Equity(3014 [BWACW])	-55.947030	False	True	10337.0
	Equity(18846 [TGVCW])	-40.394871	False	True	10336.0
	Equity(20099 [VIIAW])	-38.196519	False	True	10335.0
...	**...**	...	...	...	...
2023-12-15	**Equity(1113 [APCA])**	15.747811	True	False	5.0
	Equity(11471 [LBBB])	16.065618	True	False	4.0
	Equity(15639 [PUCK])	20.079347	True	False	3.0
	Equity(10485 [IVCB])	20.102028	True	False	2.0
	Equity(14991 [PHYT])	22.416517	True	False	1.0

Figure 12.3: A MultiIndex DataFrame with factor information and trading indicators

10. Use the **`query`** method on the **`results`** MultiIndex DataFrame to select the long and short positions that will make up our portfolio:

```
longs = results.xs(today, level=0).query(
    "longs == True")
shorts = results.xs(today, level=0).query(
    "shorts == True")
```

11. Create a simple equal-weight to determine how to allocate the assets in our portfolio:

```
weight = 1 / top_n / 2
```

12. Add the code to loop through each of the **`longs`** and **`shorts`** DataFrames and order a target percent of each asset based on the weight:

```
for row in pd.concat([longs, shorts]).itertuples():
    side = 1 if row.longs else -1
    symbol = row.Index.symbol
    contract = stock(symbol, "SMART", "USD")
    app.order_target_percent(contract, market,
        side * weight)
```

How it works...

The mechanics of the `zipline-reloaded` pipeline API are covered extensively in ***Chapter 5***, *Build Alpha Factors for Stock Portfolios*. The contribution of this chapter is to isolate the stocks that we want to go long and short, generate a simple weighting scheme, and send the orders for the target percentage.

The `longs` and `shorts` variables are derived from a `results` DataFrame. For a specific date, the `xs` method extracts a subset of the DataFrame at a `date` level, and the `query` method filters this subset to include only rows where `longs` or `shorts` are `True`, respectively.

The `weight` variable is calculated as the inverse of twice the number of top stocks (`top_n`) to be traded, creating an equally distributed allocation among all the stocks.

The `for` loop iterates over each row of the concatenated `longs` and `shorts` DataFrames. For each row, it determines the side of the trade: `1` for long positions and `-1` for short positions. The symbol of the stock is extracted from the row's index.

For each stock symbol, a `contract` object is created using the `stock` function, which specifies the stock symbol, the exchange, and the currency. Finally, the `order_target_percent` method is called for each stock, setting the target percentage of the portfolio to be allocated to this stock. The percentage is calculated by multiplying the integer `side` by the `weight` float, effectively scaling the allocation based on whether the position is long or short.

There's more...

You may want to include a screen while building the universe for which to compute momentum and rank. One way to do that is to filter stocks based on their daily average dollar volume, which helps us focus on more liquid and potentially less volatile assets. We can use the `AverageDollarVolume` class, which includes only those stocks that exceed a specified threshold in daily average dollar volume.

To do it, import the class:

```
from zipline.pipeline.factors import AverageDollarVolume
```

Then update the **`make_pipeline`** function to resemble the following:

```
def make_pipeline():
    momentum = MomentumFactor()
    dollar_volume = AverageDollarVolume(window_length=21)
    return Pipeline(
        columns={
            "factor": momentum,
            "longs": momentum.top(top_n),
            "shorts": momentum.bottom(top_n),
            "rank": momentum.rank(ascending=False),
        },
        screen=dollar_volume.top(100)
    )
```

The screen will apply the filter after it computes the momentum value and ranks the stocks. That means that even though we asked for the **`top_n`** number of assets to include in the pipeline, we may end up with fewer than **`top_n`** if they are screened out after ranking.

See also

Learn how to use the Pipeline API in a **`zipline-reloaded`** backtest at **https://www.pyquantnews.com/the-pyquant-newsletter/backtest-a-custom-momentum-strategy-with-zipline**.

Deploying an options combo strategy

Options can provide algorithmic traders with the ability to express market views that go beyond the linear profit and loss potential of stock trading. Unlike stocks, whose profit or loss is directly tied to the movement of the price on a dollar-for-dollar basis, options are non-linear instruments whose value is affected by multiple dimensions such as the price of the underlying asset, time to expiration, volatility, and interest rates. This multidimensionality allows for strategies that can profit from a range of outcomes.

A straddle strategy includes buying a call and a put option with the same strike prices and expiration dates. This lets traders speculate on the asset's volatility instead of predicting the market's directional movement. This approach makes money for the trader when the underlying asset experiences substantial movement in any direction.

Premium collection strategies such as a short iron condor involve selling both a call and a put spread with the goal of earning the premium from the options sold. This is predicated on the belief that the underlying asset's price will remain within a specific range, leading to the options expiring worthless and the trader keeping the premium collected.

In this recipe, you'll learn how to model options trades in **Trader Workstation** (**TWS**) and submit the orders using Python through the IB API.

Getting ready

We need to add two functions to our **`contract.py`** file. Open it and add the following code at the end of the file:

```
def combo_leg(contract_details, ratio, action):
    leg = ComboLeg()
    leg.conId = contract_details.contract.conId
    leg.ratio = ratio
    leg.action = action
    leg.exchange = contract_details.contract.exchange
    return leg
def spread(legs):
    contract = Contract()
    contract.symbol = "USD"
    contract.secType = "BAG"
    contract.currency = "USD"
    contract.exchange = "SMART"
    contract.comboLegs = legs
    return contract
```

These two functions allow us to create the legs of a combo order and use those legs to construct a single tradeable instrument.

Note that in the **`combo_leg`** method, we need to pass in a contract's **`conId`** string. The **`conId`** string is a unique identifier maintained by IB. To get the

conId string, we have to send a request to IB to get the details of that contract maintained by IB. To do that, we use an IB method called **reqContractDetails**. In the **client.py** file, add the following method directly under the **__init__** method in the **IBClient** class:

```
def resolve_contract(self, contract,
    request_id=DEFAULT_CONTRACT_ID):
        self.reqContractDetails(reqId=request_id,
             contract=contract)
        time.sleep(2)
        self.contractDetailsEnd(reqId=request_id)
        return self.resolved_contract
```

This method takes a contract object, requests the details from IB, and returns its details.

Let's see how it works.

How to do it...

We'll construct an options position called a **strangle**, which will let us bet on an increase in the volatility of the underlying. The following code should be added under the creation of the app under the **if __name__ == "__main__":** line in **app.py**.

1. Add the code to create a long call option contract on TSLA expiring in March 2024 with a strike price of $260:
   ```
   long_call_contract = option("TSLA", "SMART", "202403",
       260, "CALL")
   long_call = app.resolve_contract(long_call_contract)
   ```
2. Add the code to create a long put option contract on TSLA expiring on the same date with the same strike price:
   ```
   long_put_contract = option("TSLA", "SMART", "202404",
       260, "PUT")
   long_put = app.resolve_contract(long_put_contract)
   ```
3. Convert each resolved contract into a leg of our spread contract:
   ```
   leg_1 = combo_leg(long_call, 1, BUY)
   leg_2 = combo_leg(long_put, 1, BUY)
   ```
4. Pass a list of the legs to the spread function to put the legs together into a single tradeable instrument:

```
long_strangle = spread([leg_1, leg_2])
```

5. Create a market order object and send the order to the IB API:

```
order = market(BUY, 1)
app.send_order(long_strangle, order)
```

How it works…

The **combo_leg** function takes three parameters: **contract_details**, **ratio**, and **action**. It creates an instance of **ComboLeg** and sets **conId**, **ratio**, **action**, and **exchange**. The fully defined **ComboLeg** object is then returned. The **spread** function takes a list of **ComboLeg** objects created by the **combo_leg** function. The function creates an instance of **Contract**, sets its **symbol** to **USD**, **type** to **BAG** (which stands for a **bag** or combination of contracts), **currency** to **USD**, and **exchange** to **SMART**. It then assigns the **legs** parameter to the **comboLegs** attribute of the contract. The complete **Contract** object is then returned.

In this example, we constructed a long straddle. A long options straddle is a market-neutral strategy whereby we purchase both a call option and a put option on the same underlying asset with the same strike price and expiration date. Straddles allow us to bet on volatility rather than the direction of the price movement. The position profits if the underlying asset moves up or down, allowing the trader to exercise one of the options for a profit that exceeds the total cost of both premiums.

We can model options trades in TWS. To get started, click the plus sign to the right of the tabs at the bottom of TWS. Click **Browse** on the right side of the popup to browse predefined layouts. Click **Options** on the left side, then select **Add Layout** on the **Options Trading** card in the top left. Once this has been added, load your favorite underlying. In the **Options Chains** window, toggle **Strategy Builder**.

For example, using the Strategy Builder, we can construct our straddle.

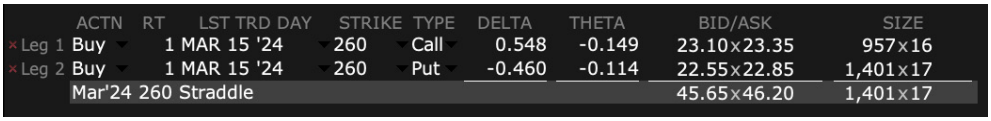

Figure 12.4: A long straddle modeled using the Strategy Builder in TWS

By clicking the **Profile** button just under the **Strategy Builder** window, we can get more details about the strategy including the payoff charts as shown in the following screenshot:

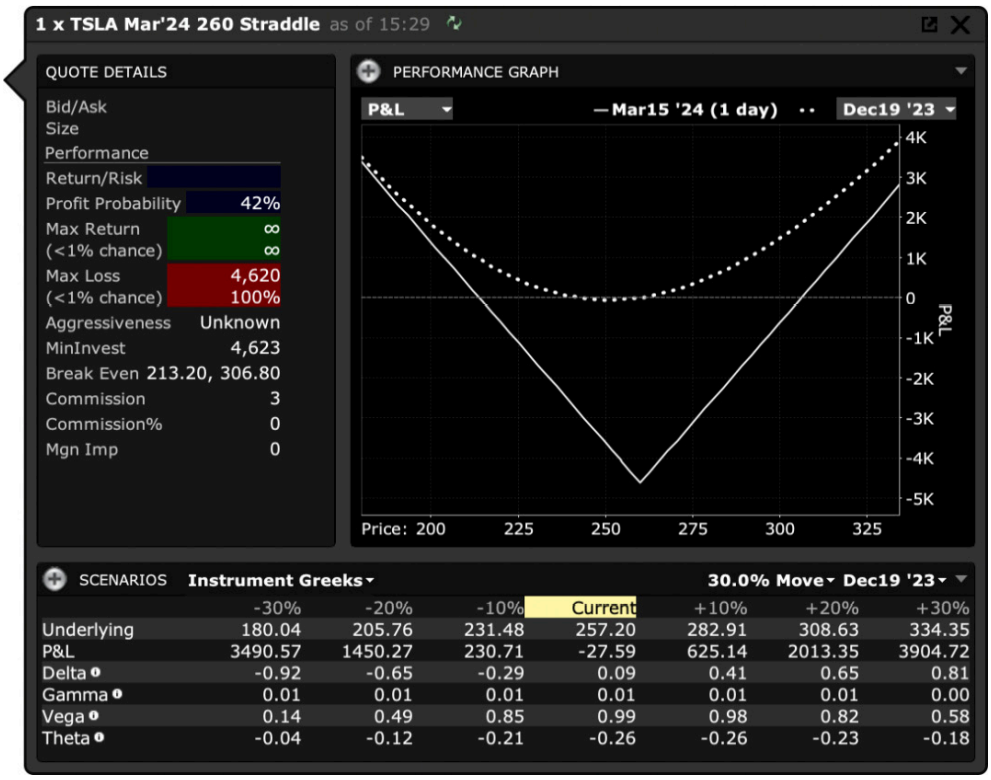

Figure 12.5: A long straddle profile in TWS

After running our app, the legs of the trade will be grouped together in the orders pane as follows:

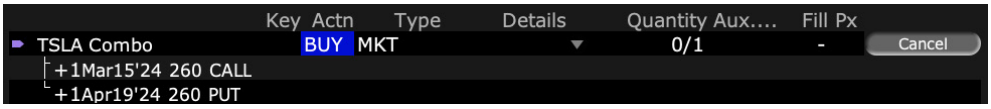

Figure 12.6: Long straddle legs grouped together in the orders pane in TWS

There's more...

Let's model another, more complex strategy called an iron condor. We'll describe a short iron condor. A short iron condor is executed by selling a put option at a lower out-of-the-money strike, purchasing another put with an even lower strike, selling a call option at a higher out-of-the-money strike, and then buying another call with an even higher strike. Typically, all options have the same expiration date. The position is constructed in such a way that we collect the premium from the options sold. The bet is that the underlying asset's price will remain between the inner

strikes of the options sold and the options expire worthless. This strategy allows traders to bet on low volatility in the underlying asset's price, aiming to profit from a range-bound market within the span of the options' expiration period.

This strategy is beneficial in a low-volatility environment, where large movements are not expected. It's a limited-risk strategy, as the long options of the spreads act as insurance against substantial adverse moves in the asset's price. The profitability is capped to the premiums received, while losses are limited to the difference between the strikes less the net premium collected.

We can model options trades in TWS. For example, using the Strategy Builder, we can construct our straddle as shown here:

	ACTN	RT	LST TRD DAY	STRIKE	TYPE	DELTA	THETA	BID/ASK	SIZE	
Leg 1	Sell		1 MAR 15 '24	295	Call	0.326	-0.130	10.80x10.95	198x218	Closed
Leg 2	Buy		1 MAR 15 '24	305	Call	0.274	-0.119	8.60x8.75	203x206	Closed
Leg 3	Sell		1 MAR 15 '24	240	Put	-0.322	-0.108	13.75x13.85	2x89	Closed
Leg 4	Buy		1 MAR 15 '24	233....	Put	-0.280	-0.104	11.40x11.55	63x171	Closed
	Total							-4.80x-4.25	203x206	

Figure 12.7: A short iron condor modeled using the Strategy Builder in TWS

By clicking the **Profile** button just under the **Strategy Builder** window, we can get more details about the strategy, including the payoff charts.

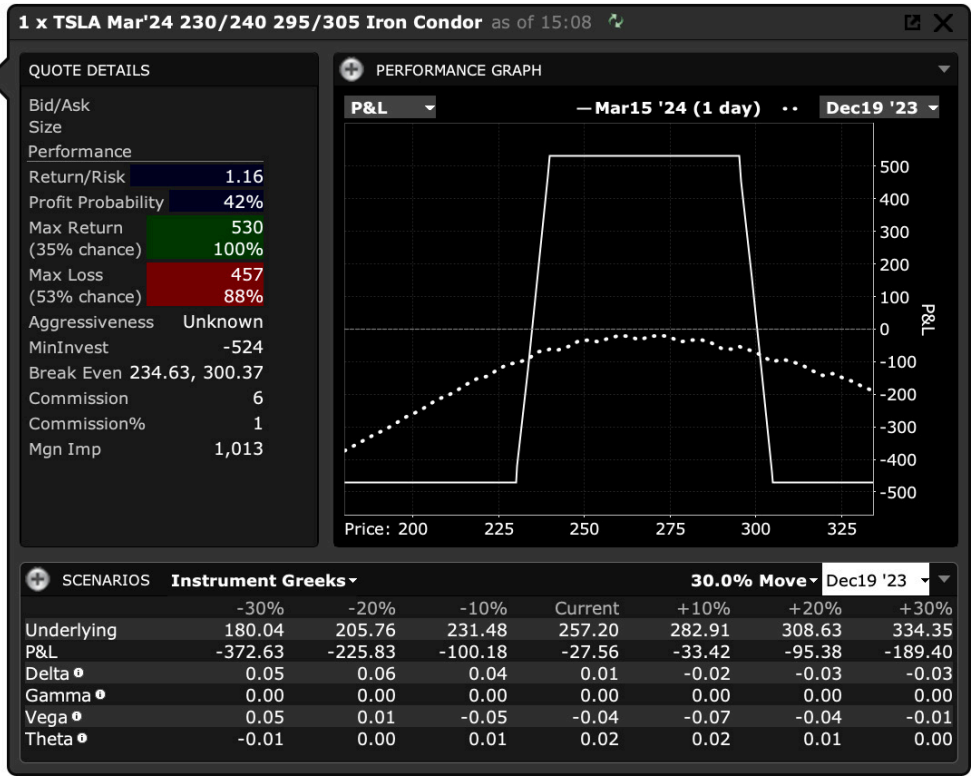

Figure 12.8: A short icon condor profile in TWS

After running our app, the legs of the trade will be grouped together in the **Orders** pane.

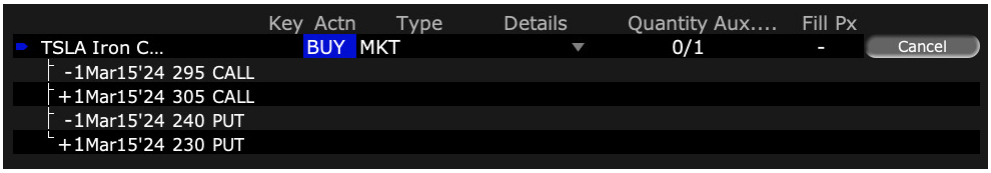

Figure 12.9: Short iron condor legs grouped together in the orders pane in TWS

See also

To learn more about options trading, you can check out the Options Industry Council's website at **https://www.optionseducation.org/**. A great desk reference is one of the most comprehensive guides to options trading on the market: *Options as a Strategic Investment*, which can be found on Amazon: **https://amzn.to/3GQRspX**

For more information on options spread orders using the IB API, see the documentation: **https://interactivebrokers.github.io/tws-api/spread_contracts.html#bag_opt**

Deploying an intraday multi-asset mean reversion strategy

In this recipe, we introduce a relative value strategy that uses the crack spread to identify opportunities to buy relatively cheap refiner stocks. The crack spread is a trading strategy in the energy sector that captures the price differential between crude oil and its refined products, most commonly gasoline and heating oil. Our strategy will use a 1:2:3 combination of one contract of **Heating Oil** (**HO**) futures, two contracts of **NYMEX RBOB Gasoline Index** (**RB**) futures, and short three **Light Sweet Crude Oil** (**CL**) futures.

When trading the crack spread against a refiner stock, we are comparing the profitability of refining operations to the market's valuation of a specific refining company. Refiner stocks should theoretically benefit from a

widening crack spread, as their margins improve when the price difference between crude and its products increases. The economic rationale behind our strategy is straightforward: refiners purchase crude oil and sell its refined products. When the spread between the two widens, the refiner's profit margin expands. Consequently, refiner stocks should move in tandem with crack spreads. Other factors such as operational efficiency, regulatory changes, and broader market sentiment can cause the relationship to break down, which we can take advantage of.

In this recipe, we'll build a strategy that identifies times when the crack spread widens substantially but a refiner's stock hasn't reacted proportionally. We'll use the opportunity to go long **Philips 66** (**PSX**), expecting the stock to catch up to the increased profitability suggested by the crack spread.

Getting ready

We assume that you've created the **`app.py`** files in the **`trading-app`** directory. If you have not, do it now.

How to do it...

We'll add the code required to connect to TWS, define the contracts for our strategy, check for trading signals every 60 seconds, and execute trades accordingly. Start by opening **`app.py`** and replacing any code under the **`if __name__ == "__main__":`** line with the following.

1. Create the connection to our trading app, replacing **`DU*******`** with your account number:

```
app = IBApp("127.0.0.1", 7497, client_id=11,
    account="DU*******")
```

2. Define the refiner stock that we'll trade (in this case, PSX) and the futures contracts that make up the crack spread:

```
psx = stock("PSX", "SMART", "USD")
ho = future("HO", "NYMEX", "202403")
rb = future("RB", "NYMEX", "202403")
cl = future("CL", "NYMEX", "202403")
```

3. Set up a window of time to compute a rolling z-score that we'll use as our trading signal along with the number of standard deviations that

trigger a trade:

```
window = 60
thresh = 2
```

4. We'll now set up an infinite loop that will continuously run throughout the trading day. The rest of the code should be under a **`while`** statement:

```
while True:
```

5. Download the historical data for the four contracts that we just defined. In our example, we'll get one-minute bars over the last week. This will give us flexibility in case we want to extend our lookback window:

```
data = app.get_historical_data_for_many(
    request_id=99,
    contracts=[psx, ho, rb, cl],
    duration="1 W",
    bar_size="1 min",
).dropna()
```

6. Compute the 1:2:3 crack spread and add the data to a new column in our DataFrame that contains the price data:

```
data["crack_spread"] = data.HO + 2 * data.RB - 3 * data.CL
```

7. To normalize the value of the crack spread and the refiner, compute the rolling 60-period rank of the last value as a percentage of the preceding values:

```
data["crack_spread_rank"] = data.crack_spread.rolling(
    window).rank(pct=True)
data["refiner_rank"] = data.PSX.rolling(window).rank(pct=True)
```

8. Compute the difference between the crack spread's 60-period rank and the refiner's 60-period rank:

```
data["rank_spread"] = data.refiner_rank - data.crack_spread_rank
```

9. Compute the rolling 60-period z-score of the difference between the crack spread's rank and the refiner's rank and take the last value as the trading signal:

```
roll = data.rank_spread.rolling(window)
zscore = ((
    data.rank_spread - roll.mean()) / roll.std())
signal = zscore[-1]
```

10. Enter a market order to buy 10 shares of PSX if the current signal is less than the threshold and we do not currently hold a position in PSX. Set the target percent of PSX to 0% if our signal exceeds **`0`** and we do not currently hold a position in PSX:

```
holding = psx.symbol in app.positions.keys()
if signal <= -thresh and not holding:
    order = market(BUY, 10)
    app.send_order(psx, order)
elif signal >= 0 and holding:
    app.order_target_percent(psx, market, 0)
```

11. Enter a market order to sell 10 shares of PSX if the current signal is greater than the threshold and we do not currently hold a position in PSX. Set the target percent of PSX to 0% if our signal goes below **`0`** and we do not currently hold a position in PSX:

```
if signal >= thresh and not holding:
    order = market(SELL, 10)
    app.send_order(psx, order)
elif signal <= 0 and holding:
    app.order_target_percent(psx, market, 0)
```

12. Finally, wait for 60 seconds before running the process again:

```
time.sleep(60)
```

The final result should resemble the following in **`app.py`**:

```
if __name__ == "__main__":
    app = IBApp("127.0.0.1", 7497, client_id=11,
        account="DU7129120")
    psx = stock("PSX", "SMART", "USD")
    ho = future("HO", "NYMEX", "202403")
    rb = future("RB", "NYMEX", "202403")
    cl = future("CL", "NYMEX", "202403")
    window = 60
    thresh = 2
    while True:
        data = app.get_historical_data_for_many(
            request_id=99,
            contracts=[psx, ho, rb, cl],
            duration="1 W",
            bar_size="1 min",        ).dropna()
        data["crack_spread"] = data.HO + 2 * data.RB - 3 * data.
        CL
        data["crack_spread_rank"] = data.crack_spread.rolling(
            window).rank(pct=True)
```

```
            data["refiner_rank"] = data.PSX.rolling(
                window).rank(pct=True)
            data["rank_spread"] = data.refiner_rank - data.crack_
            spread_rank
            roll = data.rank_spread.rolling(window)
            zscore = ((data.rank_spread - roll.mean()) / roll.std())
            signal = zscore[-1]
            holding = psx.symbol in app.positions.keys()
            if signal <= -thresh and not holding:
                order = market(BUY, 10)
                app.send_order(psx, order)
            elif signal >= 0 and holding:
                app.order_target_percent(psx, market, 0)
            if signal >= thresh and not holding:
                order = market(SELL, 10)
                app.send_order(psx, order)
            elif signal <= 0 and holding:
                app.order_target_percent(psx, market, 0)
            time.sleep(60)
        app.disconnect()
```

How it works…

We covered connecting to TWS, defining contract objects, downloading historical data, manipulating data in pandas DataFrames, and submitting orders in previous chapters. Here, we'll focus on how the trading strategy works conceptually.

To run this strategy, we would need to start it at the beginning of the trading day. The algorithm operates in a continuous loop, fetching historical data for the contracts with a one-week duration and a one-minute bar size. For each loop, we calculate the crack spread, which is then ranked over a rolling window along with the rank of PSX stock's performance. The difference between these ranks is used to generate a trading signal. This signal is a z-score, calculated over a rolling window, indicating how many standard deviations the current rank spread is from its rolling mean.

We use the **`get_historical_data_for_many`** method that we created in ***Chapter 10***, *Set up the Interactive Brokers Python API*. This method returns a DataFrame with the open, high, low, and closing prices for the PSX equity contract and HO, RB, and CL futures contracts.

> *IMPORTANT NOTE*
>
> *As of the time of writing this, the March 2024 contract was the most active contract. You may need to adjust the expiration of the contracts used during your own implementation of this strategy.*

The result of acquiring the data is the following pandas DataFrame.

symbol	CL	HO	PSX	RB
time				
2023-12-13 09:30:00-05:00	69.36	2.4674	124.78	2.0143
2023-12-13 09:31:00-05:00	69.32	2.4649	124.75	2.0133
2023-12-13 09:32:00-05:00	69.39	2.4677	124.77	2.0155
2023-12-13 09:33:00-05:00	69.47	2.4703	124.94	2.0170
2023-12-13 09:34:00-05:00	69.47	2.4700	124.79	2.0166
...	...	...	...	...
2023-12-13 15:55:00-05:00	70.29	2.4908	127.03	2.0638
2023-12-13 15:56:00-05:00	70.29	2.4910	127.09	2.0642
2023-12-13 15:57:00-05:00	70.31	2.4920	127.09	2.0641
2023-12-13 15:58:00-05:00	70.33	2.4919	127.10	2.0646
2023-12-13 15:59:00-05:00	70.36	2.4930	127.16	2.0654

Figure 12.10: A pandas DataFrame with historical market data

Now that we have the price data, we're able to compute the crack spread using the same data manipulation techniques that we learned in ***Chapter 2**, Analyze and Transform Financial Market Data with pandas*. The result of computing the crack spread is a time series that resembles the following chart.

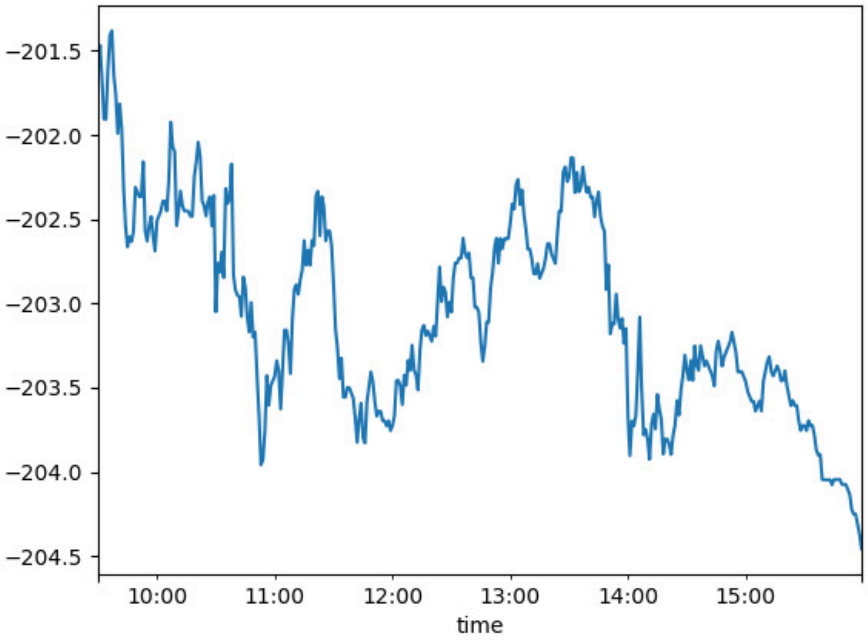

Figure 12.11: A 1:2:3 crack spread plotted over a single trading day

The prices of equities do not go negative, so we need a way to compare the relative value of the crack spread to the refiner that we're trading against. One way to do this is to determine the percent rank of the current price relative to the preceding prices. We use that technique for both the crack spread and the refiner stock. The result of these rolling percentile ranks is two time series that resemble the following chart.

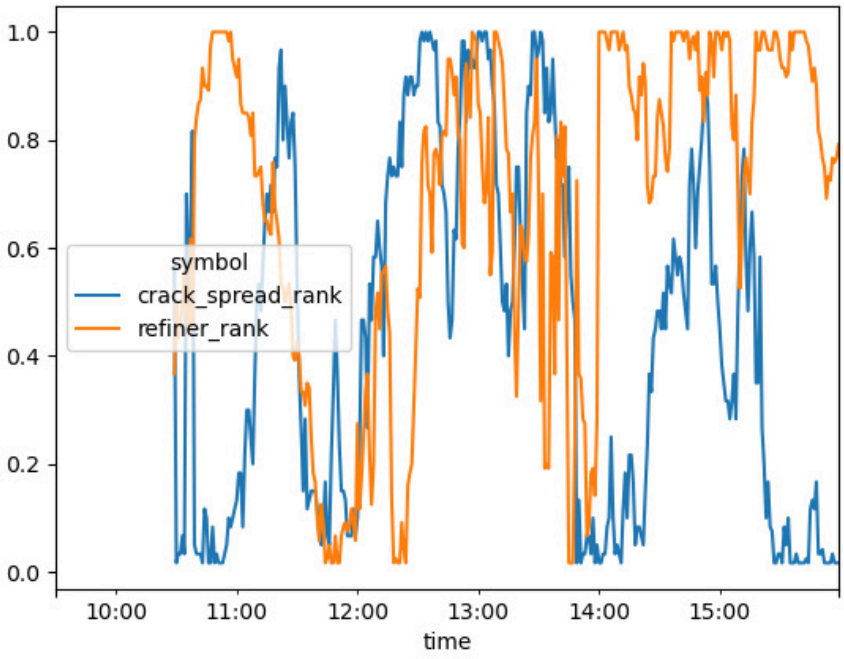

Figure 12.12: The evolution of the price rank of PSX and the crack spread relative to their 60-period windows

Now that the refiner and crack spread prices have been normalized, we can compare them by taking the difference between the two. The result of this difference is a time series that resembles the following chart.

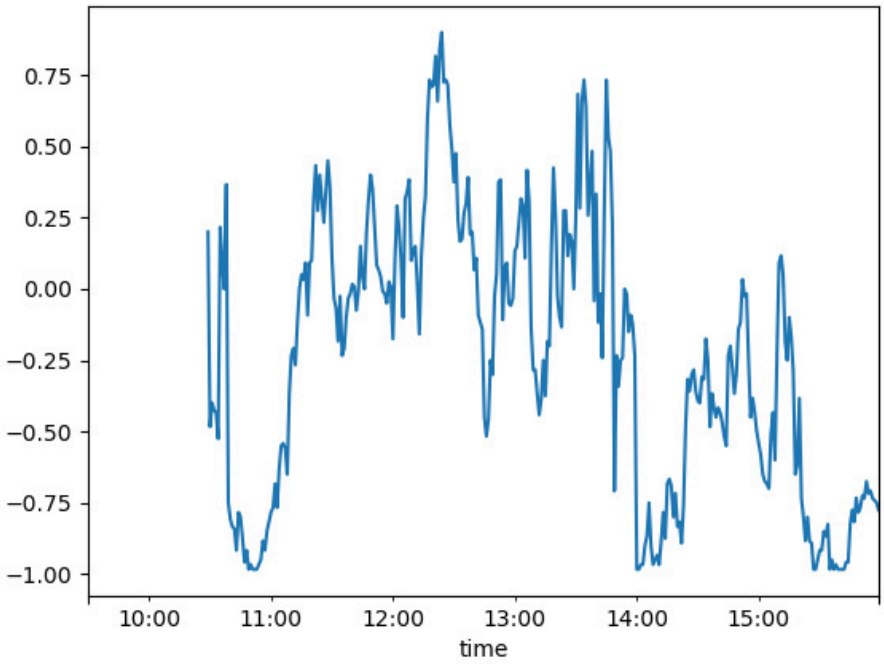

Figure 12.13: The difference between the crack spread's rank and the refiner's rank

We can see some mean reverting behavior, which we can more concisely capture through a z-score. Computing the z-score creates a time series that resembles the following chart.

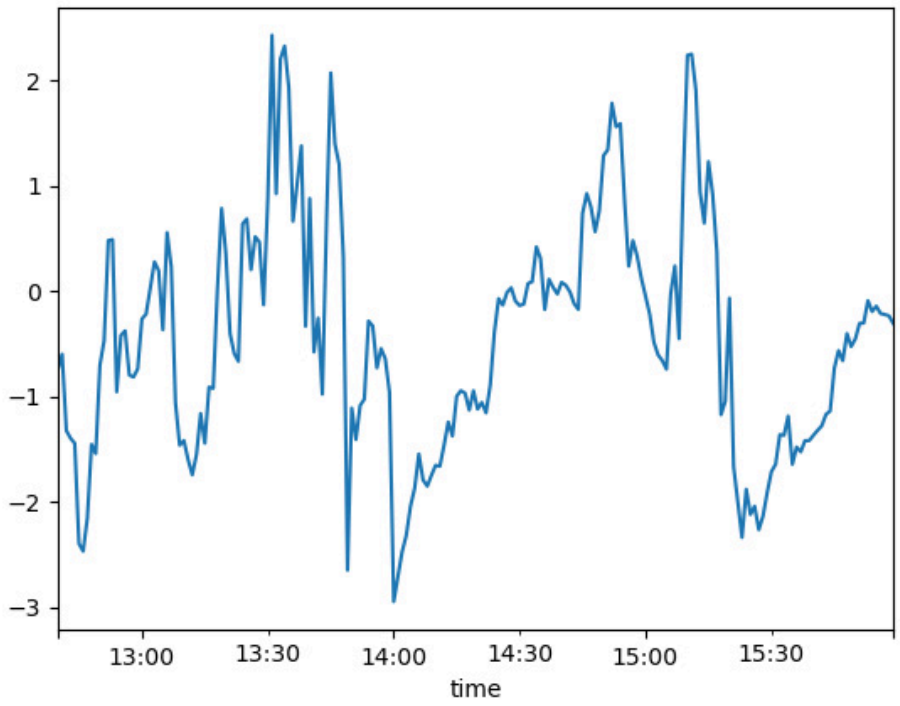

Figure 12.14: A rolling z-score of the differences between the crack spread's rank and the refiner's rank

Trading decisions are made based on this z-score. If the signal is below a negative threshold and PSX is not already held, we enter a buy order. If the signal is non-negative and PSX is held, the position is liquidated to **`0`**. Conversely, if the signal exceeds a positive threshold and PSX is not held, we enter a **`sell`** order. If the signal is negative and PSX is held, the position is again liquidated. The script pauses for 60 seconds at the end of each loop iteration. This process repeats until we stop execution.

There's more...

In this strategy, we use the crack spread as a relative value indicator. We assume that the relative value of the refiner and the crack spread will oscillate around a mean, which we can exploit. We can also create a spread order and trade the crack spread itself. The crack spread is a type of intercommodity spread by IB.

Intercommodity spreads for futures contracts are constructed using combo legs and combo orders, which allows us to simultaneously trade multiple futures contracts as a single entity. This approach is particularly relevant for strategies such as crack spreads, whereby the profit is derived from the price differences between related commodities such as crude oil and its refined products. When building these spreads, each leg

of the combo order represents a different futures contract, and it's crucial to specify the **local symbol** accurately for each contract. The local symbol is a unique identifier for futures contracts that includes information about the underlying asset, expiration date, and other contract specifics.

IB currently does not support intercommodity spreads for crack spread strategies, which prevents us from directly implementing this particular strategy on their platform. IB does support intercommodity spreads between CL and RB, as well as between RB and HO. We could technically construct a crack spread using a combination of these two intercommodity spreads but the complexity involved makes it easier to just trade the refiner stock.

See also

To learn more about the intuition behind our trade, you can read about the crack spread at **https://www.investopedia.com/terms/c/crackspread.asp**.

You can read more about building combo legs and futures spread contracts using the IB API at **https://interactivebrokers.github.io/tws-api/spread_contracts.html#bag_fut**.