momentum

April 13, 2025

[2]:
```
import os
import time
import logging
from datetime import datetime, timedelta
import pandas as pd
import numpy as np
from alpaca.trading.client import TradingClient
from alpaca.trading.requests import MarketOrderRequest
from alpaca.trading.enums import OrderSide, TimeInForce, AssetClass
from alpaca.data.historical import StockHistoricalDataClient
from alpaca.data.requests import StockBarsRequest
from alpaca.data.timeframe import TimeFrame
from pathlib import Path
```

[3]:
```
from dotenv import load_dotenv
import os

# Load variables from .env into the environment
load_dotenv()
```

[3]:
```
False
```

[4]:
```
# Bot name (same as directory name)
BOT_NAME = "momentum_strat"
log_dir = Path.cwd() / BOT_NAME
# Create directory if it doesn't exist
log_dir.mkdir(parents=True, exist_ok=True)
logger = get_bot_logger(BOT_NAME, f"{Path.cwd()}/{BOT_NAME}")

class MomentumStrategy:
    """
    A trading bot implementing a momentum strategy using the Alpaca API.
    The strategy calculates momentum as the percentage change over a lookback␣
 ↪period.
    A BUY signal is generated if the momentum exceeds a positive threshold and␣
 ↪no position exists,
    while a SELL signal is generated if momentum falls below a negative␣
 ↪threshold and a position exists.
```

```
    This version automatically liquidates positions if the momentum no longer␣
↪supports them.
    """

    def __init__(self):
        """Initialize the momentum strategy bot with API credentials and␣
↪settings."""
        # API Keys from environment variables
        self.api_key = os.getenv('API_KEY')
        self.api_secret = os.getenv('SECRET_KEY')

        if not self.api_key or not self.api_secret:
            raise ValueError("API key and secret must be provided in␣
↪environment variables")

        # Initialize Alpaca clients
        self.trading_client = TradingClient(self.api_key, self.api_secret,␣
↪paper=True)
        self.data_client = StockHistoricalDataClient(self.api_key, self.
↪api_secret)

        # Trading parameters
        self.symbols = ["GOOGL", "AAPL", "AMZN", "META", "MSFT", "NVDA"]
        self.timeframe = TimeFrame.Day
        self.lookback_days = 90                # Lookback period for momentum␣
↪calculation
        self.momentum_threshold = 0.05         # 5% momentum threshold for␣
↪trading signals

        logger.info(f"Momentum strategy bot {BOT_NAME} initialized with␣
↪{len(self.symbols)} symbols")

    def get_account_info(self):
        """Retrieve and display account information."""
        account = self.trading_client.get_account()
        logger.info(f"Account ID: {account.id}")
        logger.info(f"Cash: ${account.cash}")
        logger.info(f"Portfolio value: ${account.portfolio_value}")
        logger.info(f"Buying power: ${account.buying_power}")
        return account

    def get_positions(self):
        """Get current positions and return as a dict symbol -> position."""
        positions = self.trading_client.get_all_positions()
        pos_dict = {}
```

```
        for position in positions:
            logger.info(f"Position: {position.symbol}, "
                        f"Qty: {position.qty}, "
                        f"Market value: ${position.market_value}")
            pos_dict[position.symbol] = position
        return pos_dict

    def get_historical_data(self, symbol, days=None):
        """Fetch historical stock data for a given symbol."""
        if days is None:
            days = self.lookback_days
        end = datetime.now()
        start = end - timedelta(days=days)

        request_params = StockBarsRequest(
            symbol_or_symbols=symbol,
            timeframe=self.timeframe,
            start=start,
            end=end
        )

        bars = self.data_client.get_stock_bars(request_params)
        df = bars.df
        if df.empty:
            logger.warning(f"No data found for {symbol}")
            return None

        # Reset index to make timestamp a column and sort by timestamp
        df = df.reset_index().sort_values(by=["timestamp"])
        logger.info(f"Retrieved {len(df)} bars for {symbol}")
        return df

    def calculate_momentum(self, df):
        """
        Calculate momentum as the percentage change between the first and last␣
↪closing prices.
        """
        if df is None or len(df) < 2:
            return None
        initial_price = df['close'].iloc[0]
        latest_price = df['close'].iloc[-1]
        momentum = (latest_price - initial_price) / initial_price
        logger.info(f"Calculated momentum: {momentum:.2%}")
        return momentum

    def submit_order(self, symbol, side, qty):
        """Submit a market order for a given symbol."""
```

```
        try:
            order_data = MarketOrderRequest(
                symbol=symbol,
                qty=qty,
                side=side,
                time_in_force=TimeInForce.DAY
            )
            order = self.trading_client.submit_order(order_data)
            logger.info(f"Order placed: {side} {qty} shares of {symbol}")
            logger.info(f"Order ID: {order.id}")
            return order
        except Exception as e:
            logger.error(f"Error submitting order for {symbol}: {e}")
            return None

    def run_strategy(self):
        """Run the momentum strategy, automatically liquidating positions whose momentum is invalid."""
        logger.info("Running momentum strategy...")

        # 1. (Optional) Check market hours for logic or simulation
        current_hour = datetime.now().hour
        if current_hour < 9 or current_hour >= 16:
            logger.info("Market is closed. Running in simulation mode.")

        # 2. Pull current positions & account info
        positions = self.get_positions()
        account = self.get_account_info()

        # 3. Calculate total cash allocation (90% of available cash) and per-stock allocation
        total_allocated_cash = float(account.cash) * 0.9
        allocation_per_stock = total_allocated_cash / len(self.symbols)

        # 4. Loop over each symbol
        for symbol in self.symbols:
            logger.info(f"Analyzing {symbol}...")

            # Fetch historical data & calculate momentum
            df = self.get_historical_data(symbol)
            if df is None:
                continue
            momentum = self.calculate_momentum(df)
            if momentum is None:
                logger.warning(f"Insufficient data to calculate momentum for {symbol}")
                continue
```

```
            # Compute how many shares we can buy/sell based on allocated cash
            current_price = df['close'].iloc[-1]
            qty_for_new_position = int(allocation_per_stock // current_price)
            if qty_for_new_position <= 0:
                logger.warning(
                    f"Not enough allocated cash (${allocation_per_stock:.2f}) "
                    f"to trade {symbol} at ${current_price:.2f}"
                )
                continue

            # Check if a position already exists for this symbol
            existing_position = positions.get(symbol, None)

            # ----- MOMENTUM-BASED LIQUIDATION -----
            # If you have a LONG but momentum dips below +threshold => Sell to␣
↪close
            # If you have a SHORT but momentum climbs above -threshold => Buy␣
↪to cover
            if existing_position:
                pos_qty = float(existing_position.qty)

                # If currently LONG and momentum <= threshold, exit the position
                if pos_qty > 0 and (momentum <= self.momentum_threshold):
                    logger.info(
                        f"Momentum for {symbol} is {momentum:.2%}, below the␣
↪long threshold "
                        f"of {self.momentum_threshold:.2%}. Liquidating long␣
↪position of {pos_qty} shares."
                    )
                    self.submit_order(symbol, OrderSide.SELL, pos_qty)

                # If currently SHORT and momentum >= -threshold, exit the␣
↪position
                elif pos_qty < 0 and (momentum >= -self.momentum_threshold):
                    logger.info(
                        f"Momentum for {symbol} is {momentum:.2%}, above the␣
↪short threshold "
                        f"of {-self.momentum_threshold:.2%}. Covering short␣
↪position of {abs(pos_qty)} shares."
                    )
                    self.submit_order(symbol, OrderSide.BUY, abs(pos_qty))

                # Refresh positions dict in case we just closed something
                positions = self.get_positions()
                existing_position = positions.get(symbol, None)
```

```
            # ----- ENTRY LOGIC -----
            # 1) If momentum > threshold => Go long (if not already long)
            if momentum > self.momentum_threshold:
                if symbol in positions:
                    pos_qty = float(positions[symbol].qty)
                    if pos_qty > 0:
                        logger.info(f"Already long on {symbol}. No additional␣
↪buy order placed.")
                    # If you see pos_qty < 0, it should have been covered␣
↪above, so do nothing.
                else:
                    # Not in a position => place a BUY
                    logger.info(
                        f"BUY signal for {symbol}: momentum {momentum:.2%}␣
↪exceeds threshold "
                        f"{self.momentum_threshold:.2%}. Placing market buy of␣
↪{qty_for_new_position} shares."
                    )
                    self.submit_order(symbol, OrderSide.BUY,␣
↪qty_for_new_position)

            # 2) If momentum < -threshold => Go short (if not already short)
            elif momentum < -self.momentum_threshold:
                if symbol in positions:
                    pos_qty = float(positions[symbol].qty)
                    if pos_qty < 0:
                        logger.info(f"Already short on {symbol}. No additional␣
↪short order placed.")
                    # If pos_qty > 0, it would have been sold above, so do␣
↪nothing.
                else:
                    # Not in a position => place a SELL (short)
                    logger.info(
                        f"SHORT signal for {symbol}: momentum {momentum:.2%} is␣
↪below the negative "
                        f"threshold {-self.momentum_threshold:.2%}. Placing␣
↪market short of {qty_for_new_position} shares."
                    )
                    self.submit_order(symbol, OrderSide.SELL,␣
↪qty_for_new_position)

            else:
                # Momentum within ±threshold => do nothing
                logger.info(f"No trading action for {symbol}: momentum␣
↪{momentum:.2%} within thresholds.")
```

```
        logger.info("Momentum strategy execution completed")


def main():
    """Main function to run the momentum strategy bot."""
    logger.info(f"Starting {BOT_NAME}")
    try:
        bot = MomentumStrategy()
        bot.run_strategy()
        logger.info(f"{BOT_NAME} completed successfully")
    except Exception as e:
        logger.error(f"Error running {BOT_NAME}: {e}", exc_info=True)
        raise

if __name__ == "__main__":
    main()
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In[4], line 6
      4 # Create directory if it doesn't exist
      5 log_dir.mkdir(parents=True, exist_ok=True)
----> 6 logger = get_bot_logger(BOT_NAME, f"{Path.cwd()}/{BOT_NAME}")
      8 class MomentumStrategy:
      9     """
     10     A trading bot implementing a momentum strategy using the Alpaca API
     11     The strategy calculates momentum as the percentage change over a␣
 ↪lookback period.
   (…)
     15     This version automatically liquidates positions if the momentum no␣
 ↪longer supports them.
     16     """

NameError: name 'get_bot_logger' is not defined
```

[ ]:

[ ]: