==

Challenge: Automated Market Making

==

```
#!/usr/bin/env python3
"""
Robust, risk-aware market-maker (ASCII only).
Outputs submission.csv for the first 3000 timestamps.
"""

import sys
from collections import deque
import numpy as np
import pandas as pd


class AdaptiveMarketMaking:
    """Core engine."""

    def __init__(
        self,
        tick_size=0.1,
        lot_size=2,
        max_inventory=20,
        ema_half_life=30,       # midprice volatility EMA half-life (rows)
        flow_window=40,         # recent trade imbalance
        k_vol=1.2,              # vol -> half spread
        base_half=0.15,         # static half-spread cushion
        k_inv=0.6,              # inventory in spread
        inv_skew=1.5,           # inventory skew multiplier
        flow_skew=1.5,          # flow skew multiplier
        book_skew=1.5,          # depth-imbalance skew multiplier
        cool_ticks=2,           # widen filled side
        cool_steps=3,           # for N rows
    ):
        self.tick = tick_size
        self.lot = lot_size
        self.max_inv = max_inventory

        self.vol_alpha = 2.0 / (ema_half_life + 1.0)
        self.flow_window = flow_window

        self.k_vol = k_vol
        self.base_half = base_half
        self.k_inv = k_inv
        self.inv_skew = inv_skew
        self.flow_skew = flow_skew
        self.book_skew = book_skew

        self.cool_ticks = cool_ticks
        self.cool_steps = cool_steps

        self.reset_simulator()

    # ---------- helpers --------------------------------------------------
    def _round_tick(self, p):
        return round(p / self.tick) * self.tick

    def _get_row(self, ob_df, ts):
        if ts in ob_df.index:
            return ob_df.loc[ts]
        return ob_df[ob_df["timestamp"] == ts].iloc[0]

    # ---------- state ----------------------------------------------------
    def reset_simulator(self):
        self.inventory = 0
        self.active_bid = None
        self.active_ask = None
        self.valid_from = None

        self.ema_mid = None
        self.ema_var = None

        self.flow_q = deque(maxlen=self.flow_window)

        self.cool_side = None
        self.cool_left = 0

        self.row_counter = 0

    # ---------- volatility update ---------------------------------------
    def _update_vol(self, mid):
        if self.ema_mid is None:
            self.ema_mid = mid
            self.ema_var = 0.0
            return
        diff = mid - self.ema_mid
        self.ema_mid += self.vol_alpha * diff
        self.ema_var = (1.0 - self.vol_alpha) * (self.ema_var + self.vol_alpha * diff * diff)

    # ---------- trade processing ----------------------------------------
    def process_trades(self, ts, trades_t):
        if self.valid_from is None or ts < self.valid_from:
            return self.inventory

        filled = False
        if self.active_bid is not None:
            sells = trades_t[trades_t.side == "sell"]
            if not sells.empty and sells.price.max() <= self.active_bid:
                self.inventory += self.lot
                self.active_bid = None
                self.cool_side = "bid"
                self.cool_left = self.cool_steps
                filled = True

        if self.active_ask is not None:
            buys = trades_t[trades_t.side == "buy"]
            if not buys.empty and buys.price.min() >= self.active_ask:
                self.inventory -= self.lot
                self.active_ask = None
                self.cool_side = "ask"
                self.cool_left = self.cool_steps
                filled = True

        if filled:
            self.valid_from = float("inf")
        return self.inventory

    # ---------- quote builder -------------------------------------------
    def _build_quote(self, row, inv, flow_imb):
        # top two levels for book imbalance
        bid1 = row.bid_1_price
        ask1 = row.ask_1_price
        bid1_sz = row.bid_1_size
        ask1_sz = row.ask_1_size

        bid2 = row.bid_2_price
        ask2 = row.ask_2_price
        bid2_sz = row.bid_2_size
        ask2_sz = row.ask_2_size
```

```python
depth_bid = bid1_sz + bid2_sz
depth_ask = ask1_sz + ask2_sz
if depth_bid + depth_ask > 0:
    book_imb = (depth_bid - depth_ask) / (depth_bid + depth_ask)
else:
    book_imb = 0.0

mid = (bid1 + ask1) / 2.0
self._update_vol(mid)
sigma = np.sqrt(self.ema_var) if self.ema_var is not None else 0.0

half = max(self.tick,
           self.base_half,
           self.k_vol * sigma,
           self.k_inv * abs(inv) * self.tick)

# inventory + flow + book skew (ticks)
skew_ticks = (
    -self.inv_skew * inv / self.max_inv
    + self.flow_skew * flow_imb
    + self.book_skew * book_imb
)

bid = mid - half + skew_ticks * self.tick
ask = mid + half + skew_ticks * self.tick

# cool-down widens only filled side
if self.cool_left > 0:
    if self.cool_side == "bid":
        bid -= self.cool_ticks * self.tick
    else:
        ask += self.cool_ticks * self.tick

# stay at least one tick from best on that side
bid = min(bid, bid1 - self.tick)
ask = max(ask, ask1 + self.tick)

bid = self._round_tick(bid)
ask = self._round_tick(ask)
if bid >= ask:
    bid -= self.tick
    ask += self.tick
return bid, ask
```

```
# ---------- public strategy (checker calls this) --------------------
def strategy(self, ob_df, tr_df, inventory, ts):
    row = self._get_row(ob_df, ts)
    self.row_counter += 1

    # update flow imbalance
    trades_now = tr_df[tr_df["timestamp"] == ts]
    if not trades_now.empty:
        sign = trades_now.side.map({"buy": 1, "sell": -1})
        self.flow_q.append(sign.mean())
    flow_imb = sum(self.flow_q) / len(self.flow_q) if self.flow_q else 0.0

    bid, ask = self._build_quote(row, inventory, flow_imb)

    # late session flattening
    if self.row_counter > 0.95 * 3000:
        if inventory > 0:
            bid = None
        elif inventory < 0:
            ask = None

    # hard inventory guards
    if inventory >= self.max_inv:
        bid = None
    if inventory <= -self.max_inv:
        ask = None

    # cool-down countdown
    if self.cool_left > 0:
        self.cool_left -= 1
        if self.cool_left == 0:
            self.cool_side = None

    return bid, ask

# ---------- back-test helpers ---------------------------------------
def update_quote(self, ts, bid, ask):
    self.active_bid = bid
    self.active_ask = ask
    self.valid_from = ts + 1

def run(self, ob_df, tr_df):
    self.reset_simulator()
```

```
        ob_df = ob_df.head(3000).copy()
        tr_df = tr_df.head(3000).copy()

        ob_df.set_index("timestamp", inplace=True)
        tr_groups = tr_df.groupby("timestamp")

        quotes = []
        for ts in ob_df.index:
            trades_t = tr_groups.get_group(ts) if ts in tr_groups.groups else pd.DataFram
e()
            inv = self.process_trades(ts, trades_t)

            bid, ask = self.strategy(ob_df, tr_df, inv, ts)
            self.update_quote(ts, bid, ask)

            quotes.append(
                {
                    "timestamp": ts,
                    "bid_price": bid if bid is not None else "",
                    "ask_price": ask if ask is not None else "",
                }
            )
        return pd.DataFrame(quotes)


# ---------- alias for checker -------------------------------------------
class AutomatedMarketMaking(AdaptiveMarketMaking):
    pass



# ---------- I / O --------------------------------------------------------
def get_paths():
    if len(sys.argv) >= 3:
        return sys.argv[1], sys.argv[2]
    return input().strip(), input().strip()



if __name__ == "__main__":
    ob_path, tr_path = get_paths()
    ob_df = pd.read_csv(ob_path)
    tr_df = pd.read_csv(tr_path)

    amm = AutomatedMarketMaking(tick_size=0.1, lot_size=2)
    submission = amm.run(ob_df, tr_df)
    submission.to_csv("submission.csv", index=False)




==================================================
Challenge: Exotic Option Pricing using Monte Carlo Simulation
==================================================

import pandas as pd
import numpy as np
from scipy.stats import norm
from scipy.optimize import brentq
import io

# Calibration Data (ATM Options for Implied Vols)
calib_data = [
    # Stock, Strike, Maturity, Price
    ('DTC', 50, 1, 52.44),
    ('DTC', 50, 2, 54.77),
    ('DTC', 50, 5, 61.23),
    ('DTC', 75, 1, 28.97),
    ('DTC', 75, 2, 33.04),
    ('DTC', 75, 5, 43.47),
    ('DTC', 100, 1, 10.45),
    ('DTC', 100, 2, 16.13),
    ('DTC', 100, 5, 29.14),
    ('DTC', 125, 1, 2.32),
    ('DTC', 125, 2, 6.54),
    ('DTC', 125, 5, 18.82),
    ('DTC', 150, 1, 0.36),
    ('DTC', 150, 2, 2.34),
    ('DTC', 150, 5, 11.89),

    ('DFC', 50, 1, 52.45),
    ('DFC', 50, 2, 54.9),
    ('DFC', 50, 5, 61.87),
    ('DFC', 75, 1, 29.11),
    ('DFC', 75, 2, 33.34),
    ('DFC', 75, 5, 43.99),
    ('DFC', 100, 1, 10.45),
    ('DFC', 100, 2, 16.13),
    ('DFC', 100, 5, 29.14),
    ('DFC', 125, 1, 2.8),
    ('DFC', 125, 2, 7.39),
    ('DFC', 125, 5, 20.15),
    ('DFC', 150, 1, 1.26),
    ('DFC', 150, 2, 4.94),
    ('DFC', 150, 5, 17.46),

    ('DEC', 50, 1, 52.44),
    ('DEC', 50, 2, 54.8),
    ('DEC', 50, 5, 61.42),
    ('DEC', 75, 1, 29.08),
    ('DEC', 75, 2, 33.28),
    ('DEC', 75, 5, 43.88),
    ('DEC', 100, 1, 10.45),
    ('DEC', 100, 2, 16.13),
    ('DEC', 100, 5, 29.14),
    ('DEC', 125, 1, 1.96),
    ('DEC', 125, 2, 5.87),
    ('DEC', 125, 5, 17.74),
    ('DEC', 150, 1, 0.16),
    ('DEC', 150, 2, 1.49),
    ('DEC', 150, 5, 9.7),
]

spot_price = 100
risk_free_rate = 0.05
stocks = ['DTC', 'DFC', 'DEC']

# Correlation matrix
correlations = {
    ('DTC', 'DFC'): 0.75,
    ('DTC', 'DEC'): 0.5,
    ('DFC', 'DEC'): 0.25,
}

corr_matrix = np.ones((3, 3))
for i in range(3):
    for j in range(3):
        if i != j:
            pair = (stocks[i], stocks[j])
            corr_matrix[i, j] = correlations.get(pair, correlations.get((stocks[j], stocks[i]), 1.0))

# Black-Scholes
def bs_call_price(S, K, T, r, sigma):
    if T == 0 or sigma == 0:
        return max(S - K, 0)
    d1 = (np.log(S / K) + (r + 0.5 * sigma**2)*T) / (sigma * np.sqrt(T))
    d2 = d1 - sigma * np.sqrt(T)
    return S * norm.cdf(d1) - K * np.exp(-r*T) * norm.cdf(d2)

def implied_vol_call(mkt_price, S, K, T, r):
    def objective(sigma):
        return bs_call_price(S, K, T, r, sigma) - mkt_price
    try:
        return brentq(objective, 1e-6, 5)
    except:
        return 0.2  # fallback

# Compute Implied Vols
implied_vols = {}
for stock, K, T, price in calib_data:
    vol = implied_vol_call(price, spot_price, K, T, risk_free_rate)
    implied_vols[(stock, T)] = vol

# GBM Simulator
def simulate_correlated_gbm(S0_vec, r, vols, corr_mat, T, n_steps, n_paths, seed=42):
    np.random.seed(seed)
    dt = T / n_steps
    n_assets = len(S0_vec)
    L = np.linalg.cholesky(corr_mat)
    paths = np.zeros((n_paths, n_steps + 1, n_assets))
    paths[:, 0, :] = S0_vec

    for t in range(1, n_steps + 1):
        Z = np.random.normal(size=(n_paths, n_assets))
        dW = Z @ L.T * np.sqrt(dt)
        for i in range(n_assets):
            paths[:, t, i] = paths[:, t - 1, i] * np.exp((r - 0.5 * vols[i] ** 2) * dt +
vols[i] * dW[:, i])
    return paths

# Basket Knockout Pricer
def price_knockout_basket_option(option_type, strike, maturity, knockout_barrier,
                                 spot_vec, vols, corr_mat, r, n_steps=252, n_paths=1000):
    paths = simulate_correlated_gbm(spot_vec, r, vols, corr_mat, maturity, n_steps, n_pat
hs)
    basket_paths = paths.mean(axis=2)
    knocked_out = (basket_paths[:, :-1] > knockout_barrier).any(axis=1)
    final_price = basket_paths[:, -1]

    if option_type.lower() == 'call':
        payoffs = np.maximum(final_price - strike, 0)
    elif option_type.lower() == 'put':
        payoffs = np.maximum(strike - final_price, 0)
    else:
        raise ValueError("Option type must be Call or Put")

    payoffs[knocked_out] = 0
    discounted_payoff = np.exp(-r * maturity) * payoffs
    price = discounted_payoff.mean()
    return max(price, 0)

# Helper
def parse_maturity(maturity_str):
    if maturity_str.endswith('y'):
        return float(maturity_str[:-1])
    return float(maturity_str)

# Full Input Data (Id 1 to 36)
input_data = """Id,Asset,KnockOut,Maturity,Strike,Type
1,Basket,150,2y,50,Call
2,Basket,175,2y,50,Call
3,Basket,200,2y,50,Call
4,Basket,150,5y,50,Call
5,Basket,175,5y,50,Call
6,Basket,200,5y,50,Call
7,Basket,150,2y,100,Call
8,Basket,175,2y,100,Call
9,Basket,200,2y,100,Call
10,Basket,150,5y,100,Call
11,Basket,175,5y,100,Call
12,Basket,200,5y,100,Call
13,Basket,150,2y,125,Call
14,Basket,175,2y,125,Call
15,Basket,200,2y,125,Call
16,Basket,150,5y,125,Call
17,Basket,175,5y,125,Call
18,Basket,200,5y,125,Call
19,Basket,150,2y,75,Put
20,Basket,175,2y,75,Put
21,Basket,200,2y,75,Put
22,Basket,150,5y,75,Put
```

```
23,Basket,175,5y,75,Put
24,Basket,200,5y,75,Put
25,Basket,150,2y,100,Put
26,Basket,175,2y,100,Put
27,Basket,200,2y,100,Put
28,Basket,150,5y,100,Put
29,Basket,175,5y,100,Put
30,Basket,200,5y,100,Put
31,Basket,150,2y,125,Put
32,Basket,175,2y,125,Put
33,Basket,200,2y,125,Put
34,Basket,150,5y,125,Put
35,Basket,175,5y,125,Put
36,Basket,200,5y,125,Put"""

def price_basket_options_from_input(data_csv):
    df = pd.read_csv(io.StringIO(data_csv))
    spot_vec = np.array([spot_price] * 3)
    price_dict = {}

    for idx, row in df.iterrows():
        Id = int(row['Id'])
        option_type = row['Type']
        strike = float(row['Strike'])
        maturity = parse_maturity(row['Maturity'])
        knockout = float(row['KnockOut'])

        vols = np.array([implied_vols[(stock, maturity)] for stock in stocks])
        price = price_knockout_basket_option(
            option_type, strike, maturity, knockout,
            spot_vec, vols, corr_matrix, risk_free_rate
        )
        price_dict[Id] = round(price, 2)

    print("Id,Price")
    for i in range(1, 37):
        price = price_dict.get(i, 1)
        print(f"{i},{price}")

# Run the code
price_basket_options_from_input(input_data)
```

==

Challenge: Optimal Hedging Strategy
==

```
import sys, numpy as np, pandas as pd, re

A=0.95
L=1e-5
K=25
G=401

def c(s): return re.sub('[^a-z0-9]','',str(s).lower())
def f(df,n):
    t=c(n)
    for col in df.columns:
        if c(col)==t:
            return col
    raise SystemExit

t=sys.stdin.readline().split()
if len(t)<2: sys.exit()
p=np.array(t[1:],float)
n=len(p)

m=pd.read_csv('stocks_metadata.csv')
sid=f(m,'stock_id')
cc=f(m,'capital_cost')
m[cc]=pd.to_numeric(m[cc].astype(str).str.replace(',','',regex=False),errors='coerce')
m=m.dropna(subset=[cc]).set_index(sid)
cost_all=m[cc]

r=pd.read_csv('stocks_returns.csv')

d=f(r,'date')
r=r.set_index(d)
r.index=pd.to_datetime(r.index,errors='coerce')
r=(r.sort_index()/100).iloc[-n:]

common=r.columns.intersection(cost_all.index)
r=r[common]
cost_all=cost_all[common]

corr=r.corrwith(pd.Series(p,index=r.index)).abs()
sel=corr.nlargest(min(K,len(corr))).index
R=r[sel].values
```

```
cvec=cost_all[sel].values

w0=np.linalg.lstsq(R,-p,rcond=None)[0]

best=None
bv=1e100
for s in np.linspace(-5,5,G):
    w=np.round(s*w0).astype(int)
    if not w.any(): continue
    hed=p+R@w
    var=-np.quantile(hed,1-A)
    tot=(np.abs(w)*cvec).sum()
    v=var+L*tot
    if v<bv:
        bv=v
        best=w

if best is None:
    best=np.zeros_like(w0,dtype=int)
    i=int(np.argmax(np.abs(w0)))
    best[i]=1 if w0[i]>=0 else -1

for stk,qty in zip(sel,best):
    if qty!=0:
        print(f'{stk} {qty}')
```