```python
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
@author: Iago Puente Esteban
@document: Individual Assignment AM02

The assingment is composed of 2 questions:
Q1 Coin Toss Game
Q2 Rolling Window Regression Analysis
"""

import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from itertools import product

import yfinance as yf
from sklearn.linear_model import LinearRegression


#%% Q1 COIN TOSS ------------------------------------------------------------:
'''
The coin flip game is set as follows:

GAME RULES:
Win £2 if you get heads
Lose £1 if you get tails

SINGLE COIN TOSS (GAME 1)
Expected payoff is:
    0.5*2 + 0.5*(-1) = 0.5
Std dev of payoff is:
    sqrt( (0.5 * (2 - 0.5) ** 2) + (0.5 * (- 1 - 0.5) ** 2) ) = 1.5

What if we split the game over 2 coin tosses?
TWO COIN TOSSES (GAME 2)
Expected payoff is the same as for the 1 coin toss game:
    0.25 * (1 + 1) + 2 * 0.25 * (1 - 0.5) + 0.25 * (- 0.5 - 0.5) = 0.5
Std dev of payoff is reduced compared to the 1 coin toss game:
    np.sqrt(0.25 * (2 - 0.5) ** 2 + 0.5 * (0.5 - 0.5) ** 2 + 0.25 * (-1 - 0.5)**2) = 1.06

As we divide the the bet over more an more coin tosses, risk decreases quite
rapidly and in the limit (infinite number of bets) we can even eliminate it
completely.

Q1 aims for you to investigate this property of the game, using n tosses.
'''


#%% SHOW HOW TO USE PYTHON AS A CALCULATOR ONLY -----------------------------:
# --- 1a)
'''
Simply transfer equations describing the rules of the game into Python code.
'''
# --- GAME 1: Single coin toss
n = 1    # number of coin tosses
p = 0.5  # pWin = pLoss for the toss

H = 2    # heads wins £2
T = -1   # tails loses -£1
# s = (H, T) set of all possible outcomes
```

```
# Payoff and mean payoff
payoff1 = p * H + p * T
mean1 = payoff1

# Standard deviation
std_PO_1 = np.sqrt(p * (H - mean1) ** 2 + p * (T - mean1) ** 2)

print("Game 1 - Single Toss")
print("Payoff:", payoff1, "StdPayoff:", '%.2f' % std_PO_1)


# --- GAME 2: Two coin tosses
n = 2     # number of coin tosses
p = 0.25  # 0.25 pWin = pLoss on each toss

H = 1     #  £1   heads wins
T = -0.5  # -£0.5 tails loses

# Defining outcomes explicitly
HH = H + H       # Two heads
HT = H + T       # One head, one tail
TH = T + H       # One tail, one head
TT = T + T       # Two tails

payoff2 = p * HH + (2 * p) * HT + p * TT
mean2 = payoff2

# Standard deviation
std_PO_2 = np.sqrt(p * (HH - mean2) ** 2 + (2 * p) * (HT - mean2) ** 2 + p * (TT - mean2) ** 2)

print("\nGame 2 - Two Tosses")
print("Payoff:", payoff2, "StdPayoff:", '%.2f' % std_PO_2)

'''
Observation:
As the number of tosses increases, the standard deviation of the payoff decreases.
This is because the risk (variability) is spread across more independent tosses.
While the mean payoff remains constant, the standard deviation of the total payoff
grows more slowly compared to the increase in the number of tosses.

Mathematically, the standard deviation of the **average payoff** scales as \( 1/\sqrt{n} \),
where \( n \) is the number of tosses. This reduction in risk is a key concept in the
law of large numbers, which states that as more independent outcomes are averaged,
the variability of the average decreases.

- For Game 1 (1 toss), the standard deviation is 1.5.
- For Game 2 (2 tosses), the standard deviation reduces to ~1.06.
This trend will continue as the number of tosses increases, leading to a further
reduction in variability.
'''

#%% USE FOR LOOPS TO PLAY GAME WITH N TOSSES --------------------------------:
# --- 1b)
# Generalising above solution to n tosses.
# Note: since for loop are not as efficient ndarrays we will use n < 20.
# In the later question you will sovle this with ndarray.
'''
ALSO! There is a significant overhang for converting 'product()' function
output to other data types, such as lists, arrays and dataframes.
Therefore only go up to n=20.
```

```
This means that this operation is performed very fast:
    outcomes = product(s, repeat = n)

But conversion to datatypes we can use (such as list) is very slow for large n:
    list(outcomes)
'''
# FOR LOOPS
n = 3  # number of tosses
p = 0.5   # pWin = pLoss on each toss

H = 1   #  £1 heads wins
T = -0.5   # -£0.5 tails loses
s = (H, T) # set of all possible outcomes

outcomes = list(product(s, repeat = n)) # typecast to list to reveal results

print("Possible outcomes of experiment:")
for result in product("HT", repeat = n):
    print(result)

print("Corresponding probabilities:")
for result in outcomes:
    print(result)

# calculation of payoff payoffN
payoffs = [sum(outcome) for outcome in outcomes]  # Sum the payoffs for each outcome
payoffN = np.mean(payoffs)  # Mean payoff

# calculation of standard deviation of payoff std_PO_N
std_PO_N = np.std(payoffs)  # Standard deviation

# Print results
print("\nGame N -", n, "Tosses")
print("PayoffN:", '%.2f' % payoffN, "StdPayoff:", '%.2f' % std_PO_N)

#%% HOW TO USE NUMPY AND PANDAS LIBRARIES TO PLAY GAME WITH N TOSSES ---------:
# --- 1c)
'''
Repeat the exercise using numpy and pandas libraries to find the solutions for
payoff and std dev of payoff, using n=10 as an example.
'''
# NUMPY AND PANDAS
n = 10       # number of tosses
p = 0.5      # Probability of win or loss

H = 1        # Heads wins £1
T = -0.5     # Tails loses -£0.5
s = (H, T)   # Set of all possible outcomes

# Generate all possible outcomes
outcomes = list(product(s, repeat=n))  # Typecast to list to reveal results

# Create a dataframe of outcomes
df = pd.DataFrame(outcomes)
df['payoff'] = df.sum(axis=1)  # Sum outcomes along rows to compute payoffs
df.head()

# Calculate payoff
payoff = df['payoff'].mean()
```

```
# Calculate standard deviation, stdDev
stdDev = df['payoff'].std()

# Print results
print("PayoffN:", '%.2f' % payoff, "StdPayoff:", '%.2f' % stdDev)

#%% HOW TO CREATE A FUNCTION FOR TESTING 1-N ITERATIVELY ---------------------:
# --- 1d)
'''
Create a function called gameN, which takes in the number of tosses, n and
outputs the payoff's standard deviation.
Use a maximum of 10 tosses and iterate over the function for n=1,...,10 tosses.
Record the values of the standard deviation of the payoff and provide a plot.
Comment on your findings.
'''
# GENERAL GAME: n coin tosses NUMPY AND PANDAS, with function
# Function to compute the standard deviation of the payoff for n tosses
from itertools import product
import numpy as np

def gameN(n):
    """
    Function to compute the standard deviation of the average payoff for n tosses.
    """
    H = 2       # Heads wins £2
    T = -1      # Tails loses £1
    s = (H, T)  # Set of all possible outcomes

    # Generate all possible outcomes for n tosses
    outcomes = list(product(s, repeat=n))

    # Compute average payoffs for each outcome
    average_payoffs = [sum(outcome) / n for outcome in outcomes]

    # Calculate standard deviation of the average payoffs
    stdDev_average = np.std(average_payoffs)
    return stdDev_average

# Loop over tosses n=1 to n=10
n_values = range(1, 11)
std_devs = [gameN(n) for n in n_values]

# Print standard deviation values for verification
for n, std in zip(n_values, std_devs):
    print(f"n={n}, Standard Deviation: {std:.2f}")

# Plot the standard deviation
plt.figure(figsize=(8, 5))
plt.plot(n_values, std_devs, marker='o', linestyle='-', label='Std Dev of Payoff')
plt.xlabel('Number of Tosses (n)')
plt.ylabel('Standard Deviation of Payoff')
plt.title('Reduction in Risk with Increasing Number of Tosses')
plt.grid()
plt.legend()
plt.show()

# INSIDE THE DOCSTRING STORE THE VALUES OF YOUR PAYOFF'S STANDARD DEVIATIONS
# FOR n=1,...,10 TOSSES.
'''
Standard Deviation of Payoffs:
n=1: 1.50
```

```
n=2: 1.06
n=3: 0.87
n=4: 0.75
n=5: 0.67
n=6: 0.61
n=7: 0.57
n=8: 0.53
n=9: 0.50
n=10: 0.48

Comment:
As the number of tosses (n) increases, the standard deviation of the average
payoff decreases. This is consistent with the **Law of Large Numbers**, which
states that averaging independent events reduces variability.

Key Observations:
1. **High Variability at n=1**:
   - For a single toss, the standard deviation is the highest (1.5) due to the extreme outcomes

2. **Rapid Decline for Small n**:
   - The standard deviation decreases significantly as tosses increase (e.g., 1.06 at n=2, 0.75 a

3. **Diminishing Returns**:
   - As n increases beyond 4, the reduction in variability slows, but the trend continues.

4. **Mathematical Insight**:
   - The standard deviation scales as \( 1/\sqrt{n} \). In the limit, variability approaches zero

Conclusion:
Increasing the number of tosses reduces risk by smoothing out extreme outcomes, making the result
'''
#%% Q2 Rolling Window Regression Analysis
'''
In L4_LAB Question 1, you found the alpha and beta of the Apple Inc stock, i.e.
you ran a single regression to find a single value of alpha and beta.

In practice, the market is dynamic, and the relationship between a stock and
the market index can change over time due to shifts in market conditions,
company performance, economic factors, or macroeconomic events.

Instead of using a static, historical alpha and beta, it is often better to use
rolling alpha and beta values. This means calculating alpha and beta over a
moving window (e.g., the past 30 days or 60 days). The rolling approach
provides a more up-to-date assessment of the stock's behavior, which can be
very useful for trading decisions.

You can copy paste all the relevant answers from the L4_LAB Question 1 and
adapt the solution to the task:

Your task is to create a for loop to loop over the Apple data, using a 3 month
window (use 63 observations at a time). This means you will need to advance the
data input window by 1 time step forward keeping the window size the same,
store the resulting alpha and beta from each window's regression and then plot
the found alphas and betas to see how they evolve with time.

Note there is no need to split the data into training/test for this exercise,
as you are simply fitting regressions and reporting intercepts and slopes.
'''

# Load stock, market, and risk-free rate data
start_date = '2018-01-01'
```

```
end_date = '2024-11-15'

# Download data for AAPL, S&P500 (^GSPC), and risk-free rate (^IRX)
stock_data = yf.download('AAPL' , start=start_date, end=end_date)['Adj Close']
market_data = yf.download('^GSPC' , start=start_date, end=end_date)['Adj Close']
risk_free_data = yf.download('^IRX', start=start_date, end=end_date)['Adj Close']

# Calculate daily returns and risk-free rate
stock_returns = stock_data.pct_change()
market_returns = market_data.pct_change()
risk_free_daily = (risk_free_data / 100) / 252 # Annualized to daily rate

# Combine into a single DataFrame
df = pd.concat([stock_returns, market_returns, risk_free_daily], axis=1)
df.dropna(inplace=True)
df.columns = ['stock_returns', 'market_returns', 'risk_free_daily']

# Calculate excess returns
df['excess_stock_returns'] = df['stock_returns'] - df['risk_free_daily']
df['excess_market_returns'] = df['market_returns'] - df['risk_free_daily']

# Rolling Window Regression
'''
Using a rolling window of 63 observations, calculate alpha and beta for each
window using OLS regression. Store the results in separate lists.
'''
window_size = 63  # 3-month rolling window
alphas = []
betas = []

# Loop through the data with a rolling window
for i in range(len(df) - window_size + 1):
    # Extract the rolling window data
    window = df.iloc[i:i + window_size]
    X = window['excess_market_returns'].values.reshape(-1, 1)  # 2D input
    y = window['excess_stock_returns'].values

    # Fit the linear regression model
    model = LinearRegression()
    model.fit(X, y)

    # Store alpha (intercept) and beta (slope)
    alphas.append(model.intercept_)
    betas.append(model.coef_[0])

# Generate time index for plotting (center of each window)
time_index = df.index[window_size - 1:]

# Line Plot of Alphas over time
plt.figure(figsize=(10, 5))
plt.plot(alphas, label='Alpha', marker='o')
plt.hlines(0, xmin=0, xmax=len(alphas) - 1, colors='red')
plt.xlabel('Time')
plt.ylabel('Values')
plt.title('Rolling Alphas Over Time')
plt.legend()
plt.grid(True)
plt.tight_layout()
plt.show()

# Line Plot of Betas over time
```

```
plt.figure(figsize=(10, 5))
plt.plot(betas, label='Beta', marker='o')
plt.hlines(1, xmin=0, xmax=len(alphas) - 1, colors='red')
plt.xlabel('Time')
plt.ylabel('Values')
plt.title('RollingBetas Over Time')
plt.legend()
plt.grid(True)
plt.tight_layout()
plt.show()

'''
The rolling alphas and betas plots illustrate how the stock's relationship with
the market evolves over time:

1. **Alpha**:
   - Positive alpha indicates that the stock outperformed the market-adjusted risk-free rate.
   - Negative alpha indicates underperformance.
   - Alpha changes over time, reflecting market conditions or company-specific factors.

2. **Beta**:
   - Beta close to 1 means the stock moves in line with the market.
   - Higher beta (>1) indicates higher volatility compared to the market.
   - Lower beta (<1) indicates the stock is less volatile.

By observing the trends, we can see periods where the stock became more/less
sensitive to market movements, which may align with major economic events.
'''
```