```python
"""
Forecasting stock prices using the ARIMA model and optimizing stock portfolios using Mean-
Variance Optimization (MVO).

This module includes functions for:

1. Testing for stationarity in time series data.
2. Forecasting future stock prices with the ARIMA model.
3. Calculating daily returns from stock price data.
4. Computing key statistics such as mean returns, correlation, and covariance matrices.
5. Evaluating portfolio performance.
6. Optimizing a portfolio to balance return and risk based on user-defined preferences.
7. Integrating data downloading, forecasting, and optimization to determine the optimal
stock portfolio.

Functions:
- adf_test(series)
- forecast_arima(data, steps=183)
- calculate_daily_returns(data)
- calculate_statistics(returns)
- portfolio_performance(weights, mean_returns, cov_matrix)
- objective_function(weights, mean_returns, cov_matrix, risk_preference)
- check_sum(weights)
- optimize_portfolio(mean_returns, cov_matrix, risk_preference)
- get_optimal_portfolio(tickers, start_date, risk_preference)
"""

def adf_test(series):
    """
    Perform the Augmented Dickey-Fuller (ADF) test to check for stationarity in the time
series.

    Args:
        series (pd.Series): The time series data to test.

    Returns:
        float: The p-value of the ADF test. A p-value less than 0.05 indicates stationarity.
    """
    result = adfuller(series, autolag='AIC')
    return result[1]

def forecast_arima(data, steps=183):
    """
    Forecast future stock prices using the ARIMA model for each ticker in the dataset.

    Args:
        data (pd.DataFrame): DataFrame containing the historical stock price data.
        steps (int): Number of periods to forecast into the future (default is 183 days).

    Returns:
        tuple: A tuple containing:
            - pd.DataFrame: DataFrame with the forecasted stock prices.
            - dict: A dictionary containing the ARIMA parameters (p, d, q) used for each
ticker.
    """
    print("The forecast is being calculated. Please wait.")
    forecasted_data = pd.DataFrame()
    arima_params = {}

    for ticker in data.columns:
        series = data[ticker].dropna()

        if adf_test(series) > 0.05:
            d = 1
        else:
            d = 0
```

```
        best_aic = float("inf")
        best_order = (0, d, 0)
        model = None

        try:
            p, q = 0, 0
            while True:
                try:
                    temp_model = auto_arima(series, start_p=p, start_q=q, max_p=p+1,
max_q=q+1, d=d, seasonal=False,
                                            stepwise=False, suppress_warnings=True,
error_action="ignore", trace=False)
                    temp_aic = temp_model.aic()

                    if temp_aic < best_aic:
                        best_aic = temp_aic
                        best_order = temp_model.order
                        model = temp_model
                        p, q = p + 1, q + 1
                    else:
                        break
                except Exception as e:
                    print(f"Failed to fit ARIMA model for {ticker} with p={p}, q={q}: {e}")
                    break

            forecast = model.predict(n_periods=steps)
            forecasted_data[ticker] = forecast
            arima_params[ticker] = best_order
        except Exception as e:
            print(f"Failed to fit ARIMA model for {ticker}: {e}")

    return forecasted_data, arima_params

def calculate_daily_returns(data):
    returns = data.pct_change().dropna()
    return returns

def calculate_statistics(returns):
    mean_returns = returns.mean()
    corr_matrix = returns.corr()
    cov_matrix = returns.cov()
    return mean_returns, corr_matrix, cov_matrix

def portfolio_performance(weights, mean_returns, cov_matrix):
    portfolio_return = np.dot(weights, mean_returns)
    portfolio_stddev = np.sqrt(np.dot(weights.T, np.dot(cov_matrix, weights)))
    return portfolio_return, portfolio_stddev

def objective_function(weights, mean_returns, cov_matrix, risk_preference):
    portfolio_return, portfolio_stddev = portfolio_performance(weights, mean_returns,
cov_matrix)
    return -(portfolio_return * risk_preference - portfolio_stddev * (1 - risk_preference))

def check_sum(weights):
    return np.sum(weights) - 1

def optimize_portfolio(mean_returns, cov_matrix, risk_preference):
    num_assets = len(mean_returns)
    args = (mean_returns, cov_matrix, risk_preference)

    constraints = {'type': 'eq', 'fun': check_sum}
    bounds = tuple((0, 1) for asset in range(num_assets))

    result = minimize(objective_function, num_assets * [1. / num_assets], args=args,
                      method='SLSQP', bounds=bounds, constraints=constraints)
    return result
```

```
def get_optimal_portfolio(tickers, start_date, risk_preference):
    data = download_data_fillna(tickers, start_date="2023-01-03",
end_date=datetime.today()-timedelta(days=1))
    forecasted_data, arima_params = forecast_arima(data)
    combined_data = pd.concat([data, forecasted_data], axis=0)
    daily_returns = calculate_daily_returns(combined_data)
    mean_returns, corr_matrix, cov_matrix = calculate_statistics(daily_returns)

    optimal_portfolio = optimize_portfolio(mean_returns, cov_matrix,
risk_preference=risk_preference)
```