Module 4 : Time Series Fundamentals for Commodities

Course: Bayesian Regression and Time Series Forecasting for Commodities Trading

---

Learning Objectives

By the end of this module, you will be able to:

1. **Test** for stationarity using ADF and KPSS tests
2. **Decompose** time series into trend, seasonality, and residual components
3. **Interpret** ACF and PACF plots for model identification
4. **Apply** differencing and detrending to achieve stationarity
5. **Identify** seasonal patterns in agricultural commodities
6. **Recognize** commodity-specific time series characteristics
7. **Prepare** time series data for Bayesian forecasting models

---

Why This Matters for Trading

Time series analysis is the foundation of commodity price forecasting. Unlike cross-sectional dat
series have **memory**:

- **Yesterday's price influences today's**: Autocorrelation matters
- **Seasonal patterns repeat**: Corn peaks pre-harvest every year
- **Regimes shift**: Oil price dynamics change with OPEC policy
- **Trends emerge**: Climate change affects agricultural productivity

Why Stationarity Matters

Stationary series: Statistical properties (mean, variance) constant over time
Non-stationary series: Mean/variance changes over time (trends, breaks)

The problem: Most statistical models assume stationarity. Fitting them to non-stationary data lea

- **Spurious correlations**: Finding patterns that don't exist
- **Invalid forecasts**: Predictions diverge to infinity
- **Poor out-of-sample performance**: Models fail in live trading

What You'll Learn to Spot

- **Unit roots**: Prices have infinite memory (use returns instead)
- **Seasonality**: Predictable annual patterns (harvestable edge)
- **Volatility clustering**: High volatility follows high volatility
- **Mean reversion**: Prices pull back to long-run average

- **Structural breaks**: Regime changes (COVID-19, policy shifts)

Trading edge: Correctly identifying these features lets you build models that actually work out-o

---

1. Stationarity: The Foundation of Time Series Analysi

Definition: Weak Stationarity

A time series $\{y_t\}$ is **weakly stationary** if:

1. **Constant mean**: $E[y_t] = \mu$ for all t
2. **Constant variance**: $\text{Var}(y_t) = \sigma^2$ for all t
3. **Time-invariant autocovariance**: $\text{Cov}(y_t, y_{t+k})$ depends only on lag k, not tin

Examples

Stationary:

- White noise: $y_t \sim N(0, 1)$ i.i.d.
- AR(1) with $|\phi| < 1$: $y_t = \phi y_{t-1} + \epsilon_t$
- Daily returns of most assets

Non-stationary:

- Random walk: $y_t = y_{t-1} + \epsilon_t$ (unit root)
- Trending series: $y_t = \alpha + \beta t + \epsilon_t$
- Price levels of most commodities

Why Prices Are Non-Stationary But Returns Are Stationary

Price: $P_t = P_{t-1} + \epsilon_t$ (random walk, unit root)
Return: $r_t = \frac{P_t - P_{t-1}}{P_{t-1}} \approx \log(P_t) - \log(P_{t-1})$ (stationary)

Key insight: First differencing (computing returns) often induces stationarity.

In [ ]:
```
# Setup: Import libraries
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from scipy import stats
from statsmodels.tsa.stattools import adfuller, kpss
from statsmodels.graphics.tsaplots import plot_acf, plot_pacf
from statsmodels.tsa.seasonal import seasonal_decompose
import warnings
warnings.filterwarnings('ignore')

# Set random seed for reproducibility
np.random.seed(42)

# Plotting style
plt.style.use('seaborn-v0_8-whitegrid')
plt.rcParams['figure.figsize'] = (12, 6)
```

```python
plt.rcParams['font.size'] = 11

print("Libraries loaded successfully!")
```

In [ ]:
```python
# Generate examples of stationary vs non-stationary series
n = 500

# Stationary: AR(1) with phi = 0.7
phi = 0.7
ar1 = np.zeros(n)
for t in range(1, n):
    ar1[t] = phi * ar1[t-1] + np.random.normal(0, 1)

# Non-stationary: Random walk
random_walk = np.cumsum(np.random.normal(0, 1, n))

# Non-stationary: Trend
trend_series = 0.1 * np.arange(n) + np.random.normal(0, 1, n)

# Visualize
fig, axes = plt.subplots(1, 3, figsize=(16, 4))

axes[0].plot(ar1, linewidth=1.5)
axes[0].axhline(0, color='red', linestyle='--', alpha=0.5)
axes[0].set_title('Stationary: AR(1) with φ=0.7', fontsize=12, fontweight
axes[0].set_xlabel('Time')
axes[0].set_ylabel('Value')
axes[0].grid(alpha=0.3)

axes[1].plot(random_walk, linewidth=1.5, color='orange')
axes[1].set_title('Non-Stationary: Random Walk', fontsize=12, fontweight=
axes[1].set_xlabel('Time')
axes[1].set_ylabel('Value')
axes[1].grid(alpha=0.3)

axes[2].plot(trend_series, linewidth=1.5, color='green')
axes[2].set_title('Non-Stationary: Linear Trend', fontsize=12, fontweight
axes[2].set_xlabel('Time')
axes[2].set_ylabel('Value')
axes[2].grid(alpha=0.3)

plt.tight_layout()
plt.show()

print("Key observations:")
print("  - AR(1): Fluctuates around mean (0), bounded")
print("  - Random walk: Wanders, no mean reversion")
print("  - Trend: Systematic increase over time")
```

2 . Testing for Stationarity

Visual inspection isn't enough. We need **statistical tests**.

2 . 1 Augmented Dickey-Fuller (ADF) Test

Null hypothesis ($H_ 0 $): Series has a unit root (non-stationary)

Alternative ($H_ 1 $): Series is stationary

Test statistic: More negative → stronger evidence against H_0

Decision rule:

- If p-value < 0.05: Reject H_0 → series is **stationary**
- If p-value ≥ 0.05: Fail to reject H_0 → series is **non-stationary**

Regression form: $$\Delta y_t = \alpha + \beta t + \gamma y_{t-1} + \sum_{i=1}^{p} \delta_i y_{t-i} + \epsilon_t$$

Test if $\gamma = 0$ (unit root) vs $\gamma < 0$ (stationary)

2.2 KPSS Test

Null hypothesis (H_0): Series is **stationary**
Alternative (H_1): Series has a unit root (non-stationary)

Why both tests?: They're complementary!

ADF p-value	KPSS p-value	Interpretation
< 0.05	> 0.05	**Stationary**
> 0.05	< 0.05	**Non-stationary** (unit root)
< 0.05	< 0.05	Difference stationary
> 0.05	> 0.05	Inconclusive, needs more investigation

In [ ]:
```python
def test_stationarity(series, name="Series"):
    """
    Perform ADF and KPSS tests for stationarity.
    """
    print("="*70)
    print(f"STATIONARITY TESTS: {name}")
    print("="*70)

    # ADF test
    adf_result = adfuller(series, autolag='AIC')
    print(f"\nAugmented Dickey-Fuller Test:")
    print(f"  Test Statistic: {adf_result[0]:.4f}")
    print(f"  p-value: {adf_result[1]:.4f}")
    print(f"  Critical Values: {adf_result[4]}")

    if adf_result[1] < 0.05:
        print(f"  → Reject H0: Series is STATIONARY (p < 0.05)")
    else:
        print(f"  → Fail to reject H0: Series is NON-STATIONARY (p ≥ 0.05

    # KPSS test
    kpss_result = kpss(series, regression='c', nlags='auto')
    print(f"\nKPSS Test:")
    print(f"  Test Statistic: {kpss_result[0]:.4f}")
    print(f"  p-value: {kpss_result[1]:.4f}")
    print(f"  Critical Values: {kpss_result[3]}")

    if kpss_result[1] > 0.05:
```

```python
        print(f"  → Fail to reject H0: Series is STATIONARY (p > 0.05)")
    else:
        print(f"  → Reject H0: Series is NON-STATIONARY (p ≤ 0.05)")

    print()

# Test our example series
test_stationarity(ar1, "AR(1) - Stationary")
test_stationarity(random_walk, "Random Walk - Non-Stationary")
test_stationarity(trend_series, "Trend - Non-Stationary")
```

In [ ]:
```python
# Real example: Test stationarity of simulated commodity prices vs return
np.random.seed(42)
n_days = 500

# Simulate crude oil prices (random walk with drift)
price_init = 70
returns = np.random.normal(0.0002, 0.02, n_days)  # Small drift, 2% daily
log_prices = np.log(price_init) + np.cumsum(returns)
oil_prices = np.exp(log_prices)

# Test prices (non-stationary)
test_stationarity(oil_prices, "Oil Prices (Levels)")

# Test returns (stationary)
oil_returns = np.diff(np.log(oil_prices))
test_stationarity(oil_returns, "Oil Returns (First Difference)")

# Visualize
fig, axes = plt.subplots(2, 1, figsize=(14, 8))

axes[0].plot(oil_prices, linewidth=1.5)
axes[0].set_title('Oil Prices: Non-Stationary (Random Walk)', fontsize=12
axes[0].set_xlabel('Day')
axes[0].set_ylabel('Price ($/barrel)')
axes[0].grid(alpha=0.3)

axes[1].plot(oil_returns, linewidth=1)
axes[1].axhline(0, color='red', linestyle='--', alpha=0.5)
axes[1].set_title('Oil Returns: Stationary (After First Differencing)', f
axes[1].set_xlabel('Day')
axes[1].set_ylabel('Log Return')
axes[1].grid(alpha=0.3)

plt.tight_layout()
plt.show()

print("\nKey Insight: ALWAYS work with returns (or differenced prices) fo
```

3 . Time Series Decomposition

Most commodity price series can be decomposed into three components:

$$y_t = T_t + S_t + R_t$$

where:

- T_t = **Trend**: Long-term increase/decrease

- S_t = **Seasonality**: Regular, predictable patterns (annual, quarterly)
- R_t = **Residual**: Irregular, random fluctuations

Additive vs Multiplicative

Additive: $y_t = T_t + S_t + R_t$ (seasonal variation constant)
Multiplicative: $y_t = T_t \times S_t \times R_t$ (seasonal variation proportional to level)

Commodity rule: Use multiplicative for prices (% changes), additive for log-prices

Why Decompose?

1. **Identify seasonality**: When does corn peak?
2. **Detrend**: Remove long-term drift to find cycles
3. **Forecast components**: Model trend, seasonality, residuals separately
4. **Anomaly detection**: Large residuals = unusual events

In [ ]:

```
# Generate synthetic corn prices with seasonality
np.random.seed(42)
n_years = 5
n_days = 365 * n_years
t = np.arange(n_days)

# Components
trend = 400 + 0.02 * t  # Slow upward trend
seasonal = 50 * np.sin(2 * np.pi * t / 365 - np.pi/2)  # Annual cycle, pe
noise = np.random.normal(0, 15, n_days)

corn_prices = trend + seasonal + noise

# Create date index
dates = pd.date_range('2019-01-01', periods=n_days, freq='D')
corn_series = pd.Series(corn_prices, index=dates)

# Decompose
decomposition = seasonal_decompose(corn_series, model='additive', period=

# Plot decomposition
fig, axes = plt.subplots(4, 1, figsize=(14, 12))

decomposition.observed.plot(ax=axes[0], title='Observed', color='blue')
axes[0].set_ylabel('Price (¢/bushel)')
axes[0].grid(alpha=0.3)

decomposition.trend.plot(ax=axes[1], title='Trend', color='red')
axes[1].set_ylabel('Price (¢/bushel)')
axes[1].grid(alpha=0.3)

decomposition.seasonal.plot(ax=axes[2], title='Seasonal', color='green')
axes[2].set_ylabel('Price (¢/bushel)')
axes[2].grid(alpha=0.3)

decomposition.resid.plot(ax=axes[3], title='Residual', color='purple')
axes[3].axhline(0, color='black', linestyle='--', alpha=0.5)
axes[3].set_ylabel('Price (¢/bushel)')
axes[3].set_xlabel('Date')
```

```
axes[3].grid(alpha=0.3)

plt.suptitle('Time Series Decomposition: Corn Prices', fontsize=14, fontw
plt.tight_layout()
plt.show()

print("="*70)
print("DECOMPOSITION INSIGHTS")
print("="*70)
print(f"\nTrend: Prices rising from ~{decomposition.trend.dropna().iloc[0
print(f"Seasonality: Amplitude = {decomposition.seasonal.max():.0f} ¢/bus
print(f"             Peak occurs around day {decomposition.seasonal.idxma
print(f"             Trough occurs around day {decomposition.seasonal.idx
print(f"Residual: Std = {decomposition.resid.std():.1f} ¢/bushel (unexpla
print(f"\nTrading implication: Buy in late fall, sell mid-year to capture
```

4 . ACF and PACF: Understanding Autocorrelation

Autocorrelation Function (ACF)

Definition: Correlation between y_t and y_{t-k} for lag k

$$\rho_k = \frac{\text{Cov}(y_t, y_{t-k})}{\text{Var}(y_t)}$$

Interpretation:

- $\rho_k > 0 $: Positive correlation at lag k (similar values)
- $\rho_k < 0 $: Negative correlation (oscillating)
- $|\rho_k|$ near 0 : No linear relationship at lag k

Partial Autocorrelation Function (PACF)

Definition: Correlation between y_t and y_{t-k} **controlling for** lags 1 through $k- 1 $

Why it matters: PACF tells you the **direct** effect of lag k, removing indirect effects through intermediate lags.

Pattern Recognition for Model Selection

Process	ACF Pattern	PACF Pattern
White Noise	All ≈ 0	All ≈ 0
AR(p)	Decays exponentially	Cuts off after lag p
MA(q)	Cuts off after lag q	Decays exponentially
ARMA(p,q)	Decays exponentially	Decays exponentially

Trading use: ACF/PACF help identify the right model order for forecasting.

In [ ]:
```
# Generate examples of different processes
np.random.seed(42)
n = 500

# White noise
```

```python
white_noise = np.random.normal(0, 1, n)

# AR(1) process
ar1_proc = np.zeros(n)
phi = 0.8
for t in range(1, n):
    ar1_proc[t] = phi * ar1_proc[t-1] + np.random.normal(0, 1)

# MA(1) process
theta = 0.8
errors = np.random.normal(0, 1, n)
ma1_proc = np.zeros(n)
for t in range(1, n):
    ma1_proc[t] = errors[t] + theta * errors[t-1]

# Plot ACF and PACF
fig, axes = plt.subplots(3, 3, figsize=(16, 12))

processes = [
    (white_noise, 'White Noise'),
    (ar1_proc, 'AR(1) with φ=0.8'),
    (ma1_proc, 'MA(1) with θ=0.8')
]

for i, (process, name) in enumerate(processes):
    # Time series plot
    axes[i, 0].plot(process[:200], linewidth=1)
    axes[i, 0].set_title(f'{name}', fontsize=11, fontweight='bold')
    axes[i, 0].set_ylabel('Value')
    axes[i, 0].grid(alpha=0.3)

    # ACF
    plot_acf(process, lags=40, ax=axes[i, 1], alpha=0.05)
    axes[i, 1].set_title(f'ACF: {name}', fontsize=11, fontweight='bold')

    # PACF
    plot_pacf(process, lags=40, ax=axes[i, 2], alpha=0.05, method='ywm')
    axes[i, 2].set_title(f'PACF: {name}', fontsize=11, fontweight='bold')

axes[2, 0].set_xlabel('Time')
axes[2, 1].set_xlabel('Lag')
axes[2, 2].set_xlabel('Lag')

plt.tight_layout()
plt.show()

print("="*70)
print("ACF/PACF PATTERN INTERPRETATION")
print("="*70)
print("\nWhite Noise:")
print("  - ACF: All near zero (no correlation)")
print("  - PACF: All near zero")
print("  → No predictive structure")

print("\nAR(1):")
print("  - ACF: Exponential decay (slow decline)")
print("  - PACF: Sharp cutoff after lag 1")
print("  → Use AR(1) model")

print("\nMA(1):")
```

```python
print("  - ACF: Sharp cutoff after lag 1")
print("  - PACF: Exponential decay")
print("  → Use MA(1) model")
```

In [ ]:
```python
# Apply to real commodity returns
# Use the oil returns from earlier

fig, axes = plt.subplots(1, 2, figsize=(14, 5))

plot_acf(oil_returns, lags=40, ax=axes[0], alpha=0.05)
axes[0].set_title('ACF: Oil Returns', fontsize=12, fontweight='bold')

plot_pacf(oil_returns, lags=40, ax=axes[1], alpha=0.05, method='ywm')
axes[1].set_title('PACF: Oil Returns', fontsize=12, fontweight='bold')

plt.tight_layout()
plt.show()

print("\nObservation: Oil returns show minimal autocorrelation")
print("This suggests efficient markets - hard to predict returns from pas
print("Need to incorporate other information (fundamentals, seasonality,
```

5 . Differencing and Detrending Strategies

When you have a non-stationary series, you need to transform it to achieve stationarity.

Strategy 1 : First Differencing

$$\Delta y_t = y_t - y_{t-1}$$

When to use: Series has a unit root (random walk)
Effect: Removes stochastic trend

For prices: $\Delta \log(P_t) \approx$ return

Strategy 2 : Seasonal Differencing

$$\Delta_s y_t = y_t - y_{t-s}$$

where s is the seasonal period (e.g., 12 for monthly data, 365 for daily)

When to use: Series has seasonal unit root
Effect: Removes seasonal pattern

Strategy 3 : Linear Detrending

1. Fit: $y_t = \alpha + \beta t + \epsilon_t$
2. Extract residuals: $\tilde{y}_t = y_t - (\hat{\alpha} + \hat{\beta} t)$

When to use: Series has deterministic (linear) trend
Effect: Removes deterministic trend

Combined Strategies

For series with trend AND seasonality:

1. Remove trend (detrend or first difference)
2. Remove seasonality (seasonal difference or subtract seasonal component)
3. Model residuals

In [ ]:
```
# Demonstrate differencing on non-stationary series
# Use random walk with drift and seasonality

np.random.seed(42)
n = 365 * 3
t = np.arange(n)

# Random walk with drift
drift = 0.01
random_component = np.cumsum(np.random.normal(drift, 0.5, n))

# Add seasonality
seasonal_component = 10 * np.sin(2 * np.pi * t / 365)

# Combined
nonstationary_series = 100 + random_component + seasonal_component

# Apply transformations
first_diff = np.diff(nonstationary_series)
seasonal_diff = nonstationary_series[365:] - nonstationary_series[:-365]

# Detrend
from scipy import signal
detrended = signal.detrend(nonstationary_series)

# Plot
fig, axes = plt.subplots(2, 2, figsize=(14, 10))

axes[0, 0].plot(nonstationary_series, linewidth=1.5)
axes[0, 0].set_title('Original: Non-Stationary (Trend + Seasonality)', fo
axes[0, 0].set_ylabel('Value')
axes[0, 0].grid(alpha=0.3)

axes[0, 1].plot(first_diff, linewidth=1)
axes[0, 1].axhline(0, color='red', linestyle='--', alpha=0.5)
axes[0, 1].set_title('First Differencing: Δy_t = y_t - y_{t-1}', fontsize
axes[0, 1].set_ylabel('Value')
axes[0, 1].grid(alpha=0.3)

axes[1, 0].plot(seasonal_diff, linewidth=1, color='green')
axes[1, 0].axhline(0, color='red', linestyle='--', alpha=0.5)
axes[1, 0].set_title('Seasonal Differencing: y_t - y_{t-365}', fontsize=1
axes[1, 0].set_xlabel('Time')
axes[1, 0].set_ylabel('Value')
axes[1, 0].grid(alpha=0.3)

axes[1, 1].plot(detrended, linewidth=1, color='purple')
axes[1, 1].axhline(0, color='red', linestyle='--', alpha=0.5)
axes[1, 1].set_title('Linear Detrending', fontsize=11, fontweight='bold')
```

```
axes[1, 1].set_xlabel('Time')
axes[1, 1].set_ylabel('Value')
axes[1, 1].grid(alpha=0.3)

plt.tight_layout()
plt.show()

# Test stationarity of transformations
print("\nStationarity after transformations:\n")

adf_first = adfuller(first_diff)
print(f"First Difference ADF p-value: {adf_first[1]:.4f} → {'Stationary'

adf_seasonal = adfuller(seasonal_diff)
print(f"Seasonal Difference ADF p-value: {adf_seasonal[1]:.4f} → {'Statio

adf_detrend = adfuller(detrended)
print(f"Detrended ADF p-value: {adf_detrend[1]:.4f} → {'Stationary' if ad
```

6 . Commodity-Specific Time Series Characteristics

Different commodities have unique time series features that must be understood for effective mo

Agricultural Commodities (Corn, Wheat, Soybeans)

Seasonality:

- **Planting season** (spring): Uncertainty about crop size
- **Growing season** (summer): Weather-driven volatility
- **Harvest** (fall): Prices drop as supply floods market
- **Storage** (winter): Carrying costs influence futures curve

Key features:

- Strong annual seasonality (365 -day cycle)
- Weather shocks create outliers
- Government policy affects long-term trends

Energy Commodities (Crude Oil, Natural Gas)

Crude Oil:

- Geopolitical risk (supply disruptions)
- OPEC production decisions create structural breaks
- Dollar-denominated (USD strength affects prices)
- Inventory levels matter (EIA reports move markets)

Natural Gas:

- Strong seasonal pattern (winter heating, summer cooling)
- Storage capacity constraints
- Extreme volatility during cold snaps

Metals (Gold, Silver, Copper)

Precious Metals (Gold, Silver):

- Safe-haven demand (spikes during uncertainty)
- Inflation hedge (negative correlation with real rates)
- Limited industrial use (store of value dominates)

Industrial Metals (Copper):

- Economic cycle sensitivity ("Dr. Copper")
- China demand dominates
- Supply disruptions from mining strikes

In [ ]:
```
# Compare seasonal patterns across commodities
np.random.seed(42)
n_years = 3
n_days = 365 * n_years
t = np.arange(n_days)

# Corn: Peak mid-year (pre-harvest fear), trough post-harvest
corn_seasonal = 400 + 50 * np.sin(2 * np.pi * t / 365 - np.pi/2) + np.ran

# Natural Gas: Peak winter (heating demand), trough summer
natgas_seasonal = 3.0 + 1.2 * np.sin(2 * np.pi * t / 365 + np.pi) + np.ra

# Gold: Weak seasonality, but rises during Q4 (jewelry demand for holiday
gold_seasonal = 1800 + 80 * np.sin(2 * np.pi * t / 365 + 3*np.pi/4) + np.

# Create date index
dates = pd.date_range('2021-01-01', periods=n_days, freq='D')

# Plot
fig, axes = plt.subplots(3, 1, figsize=(14, 12))

axes[0].plot(dates, corn_seasonal, linewidth=1.5, color='green')
axes[0].set_title('Corn: Strong Seasonality (Peak Pre-Harvest)', fontsize
axes[0].set_ylabel('Price (¢/bushel)')
axes[0].grid(alpha=0.3)

axes[1].plot(dates, natgas_seasonal, linewidth=1.5, color='red')
axes[1].set_title('Natural Gas: Winter Peak (Heating Demand)', fontsize=1
axes[1].set_ylabel('Price ($/MMBtu)')
axes[1].grid(alpha=0.3)

axes[2].plot(dates, gold_seasonal, linewidth=1.5, color='gold')
axes[2].set_title('Gold: Weak Seasonality (Q4 Jewelry Demand)', fontsize=
axes[2].set_ylabel('Price ($/oz)')
axes[2].set_xlabel('Date')
axes[2].grid(alpha=0.3)

plt.tight_layout()
plt.show()

# Extract and compare seasonal components
corn_decomp = seasonal_decompose(pd.Series(corn_seasonal, index=dates), m
```

```
gas_decomp = seasonal_decompose(pd.Series(natgas_seasonal, index=dates),
gold_decomp = seasonal_decompose(pd.Series(gold_seasonal, index=dates), m

print("="*70)
print("SEASONAL AMPLITUDE COMPARISON")
print("="*70)
print(f"\nCorn:")
print(f"  Seasonal range: {corn_decomp.seasonal.max() - corn_decomp.seaso
print(f"  % of mean: {100 * (corn_decomp.seasonal.max() - corn_decomp.sea
print(f"  Peak: {corn_decomp.seasonal.idxmax().strftime('%B')} (pre-harve

print(f"\nNatural Gas:")
print(f"  Seasonal range: {gas_decomp.seasonal.max() - gas_decomp.seasona
print(f"  % of mean: {100 * (gas_decomp.seasonal.max() - gas_decomp.seaso
print(f"  Peak: {gas_decomp.seasonal.idxmax().strftime('%B')} (winter hea

print(f"\nGold:")
print(f"  Seasonal range: {gold_decomp.seasonal.max() - gold_decomp.seaso
print(f"  % of mean: {100 * (gold_decomp.seasonal.max() - gold_decomp.sea
print(f"  Peak: {gold_decomp.seasonal.idxmax().strftime('%B')} (holiday d

print(f"\nKey insight: Agricultural and energy commodities have MUCH stro
print(f"             seasonality than precious metals!")
```

7 . Practical Application: Analyzing Corn Futures

Let's apply all the tools we've learned to a comprehensive analysis of corn futures.

Analysis Workflow

1. **Visual inspection**: Plot the series
2. **Stationarity testing**: ADF and KPSS on levels and returns
3. **Decomposition**: Extract trend, seasonality, residuals
4. **ACF/PACF**: Identify autocorrelation structure
5. **Transformation**: Apply differencing/detrending if needed
6. **Model preparation**: Create stationary series ready for forecasting

In [ ]:

```
# Generate realistic corn futures data
np.random.seed(42)
n_days = 365 * 5
t = np.arange(n_days)
dates = pd.date_range('2019-01-01', periods=n_days, freq='D')

# Components
base_price = 400
trend = 0.015 * t  # Slow upward trend
seasonal = 60 * np.sin(2 * np.pi * t / 365 - np.pi/2)  # Peak June, troug

# Add random walk component (non-stationary)
random_walk = np.cumsum(np.random.normal(0, 3, n_days))

# Combine + noise
corn_futures = base_price + trend + seasonal + random_walk + np.random.no
corn_futures_series = pd.Series(corn_futures, index=dates)

# Calculate returns
```

```python
corn_returns = corn_futures_series.pct_change().dropna()

print("="*70)
print("COMPREHENSIVE CORN FUTURES ANALYSIS")
print("="*70)
print(f"\nData: {n_days} days ({n_days/365:.1f} years)")
print(f"Range: ${corn_futures_series.min():.2f} - ${corn_futures_series.m
print(f"Mean: ${corn_futures_series.mean():.2f}")

# Step 1: Visual inspection
fig, axes = plt.subplots(2, 1, figsize=(14, 8))

axes[0].plot(dates, corn_futures, linewidth=1.5, color='green')
axes[0].set_title('Corn Futures: Price Levels', fontsize=12, fontweight='
axes[0].set_ylabel('Price (¢/bushel)')
axes[0].grid(alpha=0.3)

axes[1].plot(corn_returns.index, corn_returns.values, linewidth=1, alpha=
axes[1].axhline(0, color='red', linestyle='--', alpha=0.5)
axes[1].set_title('Corn Futures: Daily Returns', fontsize=12, fontweight=
axes[1].set_ylabel('Return')
axes[1].set_xlabel('Date')
axes[1].grid(alpha=0.3)

plt.tight_layout()
plt.show()

# Step 2: Stationarity tests
print("\n" + "="*70)
print("STATIONARITY ANALYSIS")
print("="*70)
test_stationarity(corn_futures_series, "Corn Price Levels")
test_stationarity(corn_returns, "Corn Returns")
```

In [ ]:

```python
# Step 3: Decomposition
decomp = seasonal_decompose(corn_futures_series, model='additive', period

fig = decomp.plot()
fig.set_size_inches(14, 10)
plt.suptitle('Seasonal Decomposition: Corn Futures', fontsize=14, fontwei
plt.tight_layout()
plt.show()

print("\n" + "="*70)
print("DECOMPOSITION RESULTS")
print("="*70)
print(f"\nTrend: Rising from {decomp.trend.dropna().iloc[0]:.1f} to {deco
print(f"Seasonal amplitude: {decomp.seasonal.max() - decomp.seasonal.min(
print(f"Peak month: {decomp.seasonal.idxmax().strftime('%B')}")
print(f"Trough month: {decomp.seasonal.idxmin().strftime('%B')}")
print(f"Residual std: {decomp.resid.std():.1f} ¢/bushel")

# Step 4: ACF/PACF analysis
fig, axes = plt.subplots(2, 2, figsize=(14, 8))

# Prices
plot_acf(corn_futures_series.dropna(), lags=60, ax=axes[0, 0], alpha=0.05
axes[0, 0].set_title('ACF: Corn Prices (Levels)', fontsize=11, fontweight

plot_pacf(corn_futures_series.dropna(), lags=60, ax=axes[0, 1], alpha=0.0
```

```
axes[0, 1].set_title('PACF: Corn Prices (Levels)', fontsize=11, fontweigh

# Returns
plot_acf(corn_returns.dropna(), lags=60, ax=axes[1, 0], alpha=0.05)
axes[1, 0].set_title('ACF: Corn Returns', fontsize=11, fontweight='bold')

plot_pacf(corn_returns.dropna(), lags=60, ax=axes[1, 1], alpha=0.05, meth
axes[1, 1].set_title('PACF: Corn Returns', fontsize=11, fontweight='bold'

plt.tight_layout()
plt.show()

print("\n" + "="*70)
print("MODEL SELECTION INSIGHTS")
print("="*70)
print("\nPrice levels:")
print("  - ACF decays very slowly → non-stationary (confirmed by ADF test
print("  - Strong autocorrelation at annual lags → seasonality present")

print("\nReturns:")
print("  - ACF mostly within confidence bands → weak autocorrelation")
print("  - Some significant lags suggest potential for AR/MA modeling")
print("  - But returns are largely unpredictable from past returns alone"

print("\nConclusion for forecasting:")
print("  1. Work with returns (stationary) or detrended/deseasonalized pr
print("  2. Incorporate seasonal component explicitly in model")
print("  3. Consider exogenous variables (weather, inventory, etc.)")
print("  4. Returns show limited autocorrelation - hard to forecast from
```

8 . Summary: Preparing Time Series for Bayesian Forecasting

The Complete Workflow

Step	Action	Tools
1 . **Visualize**	Plot the data, look for patterns	Time series plot
2 . **Test stationarity**	Formal tests for unit roots	ADF, KPSS
3 . **Decompose**	Extract trend, seasonality, noise	seasonal_decompose
4 . **Transform**	Achieve stationarity	Differencing, detrending
5 . **Check autocorrelation**	Identify model structure	ACF, PACF
6 . **Model**	Fit Bayesian time series model	Next modules!

Key Decisions for Commodity Traders

Q: Should I model prices or returns?

A: Returns are stationary, but prices have economic meaning. Consider:

- **Returns**: For short-term trading, volatility modeling
- **Prices**: For level forecasting, option pricing (use error correction models)

Q: How do I handle seasonality?
A: Three approaches:

1. **Seasonal differencing**: $y_t - y_{t-365}$
2. **Explicit seasonal terms**: Sine/cosine regressors
3. **Seasonal dummies**: Month indicators

Q: What if my series has structural breaks?
A: Options:

- **Regime-switching models**: Allow parameters to change
- **Rolling windows**: Retrain frequently on recent data only
- **Robust methods**: Student-t likelihood, less sensitive to outliers

Common Mistakes to Avoid

- ❌ Modeling non-stationary series without transformation
- ❌ Ignoring seasonality in agricultural commodities
- ❌ Over-differencing (makes series harder to forecast)
- ❌ Assuming stationarity without testing
- ❌ Ignoring outliers caused by extreme events

What's Next

Now that we can prepare time series data, we're ready to build Bayesian forecasting models:

- Bayesian structural time series (BSTS)
- Dynamic linear models
- Hierarchical time series models
- Gaussian processes for irregular data

Knowledge Check Quiz

Q1: A stationary time series has:

- A) Constant mean and variance over time
- B) No autocorrelation
- C) No seasonality
- D) A linear trend

Q2: The ADF test has null hypothesis:

- A) Series is stationary
- B) Series has a unit root (non-stationary)
- C) Series has no autocorrelation
- D) Series has seasonality

Q 3 : An ACF that decays slowly suggests:

- A) The series is stationary
- B) The series is white noise
- C) The series is non-stationary (likely has unit root)
- D) The series has no predictable structure

Q 4 : For commodity prices, first differencing typically:

- A) Makes the series non-stationary
- B) Converts prices to returns (approximately)
- C) Removes seasonality
- D) Adds a trend

Q 5 : Corn prices typically peak:

- A) During harvest (fall)
- B) Pre-harvest in summer (supply uncertainty)
- C) In winter (storage demand)
- D) Randomly (no seasonality)

In [ ]:
```
# Quiz Answers
print("="*70)
print("QUIZ ANSWERS")
print("="*70)
print("""
Q1: A) Constant mean and variance over time
    Weak stationarity requires constant mean, variance, and time-invarian
    autocovariance. Stationary series CAN have autocorrelation and season

Q2: B) Series has a unit root (non-stationary)
    ADF null = unit root. Low p-value means reject H0 → series is station
    This is opposite of KPSS where null = stationarity.

Q3: C) The series is non-stationary (likely has unit root)
    Slow ACF decay is a classic sign of non-stationarity. Stationary seri
    have ACF that decays quickly to zero.

Q4: B) Converts prices to returns (approximately)
    Δlog(P_t) = log(P_t) - log(P_{t-1}) ≈ (P_t - P_{t-1})/P_{t-1} = retur
    This transformation usually achieves stationarity.

Q5: B) Pre-harvest in summer (supply uncertainty)
    Corn peaks in June-July when weather uncertainty is highest (will cro
    fail?). Prices drop in fall as harvest brings supply certainty.
""")
```

Exercises

Complete these exercises in the `exercises.ipynb` notebook.

Exercise 1 : Multi-Step Differencing (Easy)

Generate a series with trend AND seasonality. Apply (a) first differencing, (b) seasonal differenci
both. Which achieves stationarity?

Exercise 2 : Seasonal Pattern Comparison (Medium)

Compare the seasonal patterns of wheat (winter crop, harvested summer) vs corn (spring crop, harvested fall). How should their seasonal components differ?

Exercise 3 : ACF/PACF Model Selection (Medium)

Generate AR(2), MA(2), and ARMA(1 , 1) processes. Use ACF/PACF to correctly identify ea
model type. Verify by fitting models.

Exercise 4 : Structural Break Detection (Hard)

Create a corn price series with a structural break (e.g., regime change after year 3). How do
stationarity tests behave? Develop a strategy to detect and handle the break.

Next Module Preview

In **Module 5 : Bayesian Linear Regression for Commodities**, we'll learn:

- Building regression models with time series features
- Incorporating economic indicators (inventory, production)
- Handling multicollinearity in commodity relationships
- Forecasting with uncertainty quantification
- Model comparison and selection using Bayesian methods

Module 4 Complete