Open in Colab

Time Series Analysis: Theory and Practice with Python

What Is a Time Series?

A **time series** is a sequence of observations collected **over time**, usually at regular intervals.

Examples:

- Daily stock prices
- Hourly electricity consumption
- Monthly sales revenue
- Sensor measurements every second

Why Time Series Are Different

In time series data:

- Observations are **dependent**
- The order of data points **cannot be shuffled**
- Past values influence future values

This dependency structure is the core challenge of time series analysis.

Core Characteristics of Time Series Data

1 Trend

Long-term upward or downward movement.

Examples:

- Growing revenue over years
- Declining costs due to efficiency

2 Seasonality

Regular, repeating patterns tied to calendar cycles.

Examples:

- Higher sales every December
- Daily electricity peaks in the evening

3 Cyclic Behavior

Longer, irregular economic or business cycles.

4 Noise (Residuals)

Random variation not explained by structure.

Time Series Decomposition

We often assume:

$$y_t = \text{Trend}_t + \text{Seasonality}_t + \text{Noise}_t$$

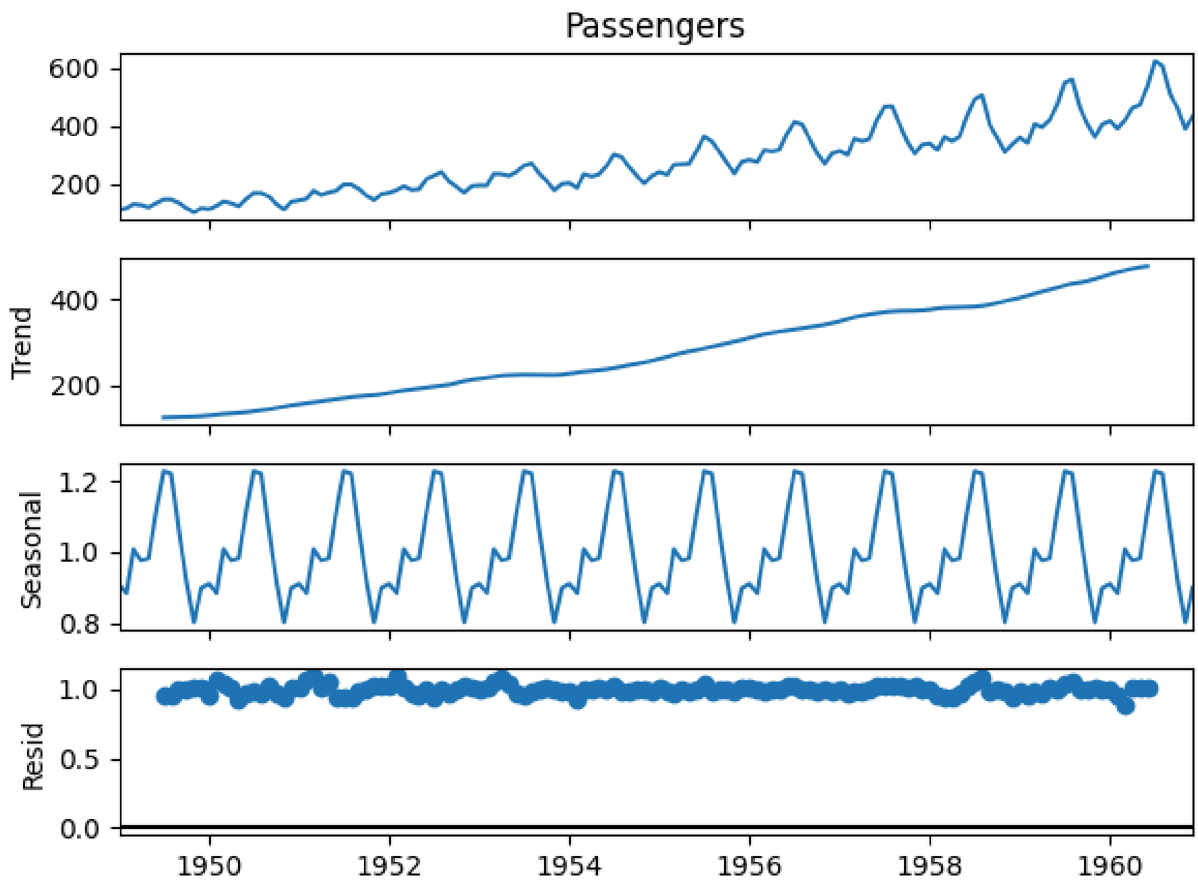
or (multiplicative form):

$$y_t = \text{Trend}_t \times \text{Seasonality}_t \times \text{Noise}_t$$

Python Example: Decomposition

In [2]:

```
import pandas as pd
import matplotlib.pyplot as plt
from statsmodels.tsa.seasonal import seasonal_decompose

# Load example data
data = pd.read_csv(
    "https://raw.githubusercontent.com/jbrownlee/Datasets/master/airline-passengers
    parse_dates=["Month"],
    index_col="Month"
)

# Decompose
decomposition = seasonal_decompose(data["Passengers"], model="multiplicative")

# Plot
decomposition.plot()
plt.show()
```



Why this matters

- Makes patterns visible
- Makes seasonality explicit
- Separates structure from noise
- Helps decide which model to use

Stationarity (Critical Concept)

Many statistical models assume **stationarity**.

A time series is stationary if:

- Mean is constant over time
- Variance is constant
- Autocorrelation structure is stable

Why Stationarity Matters

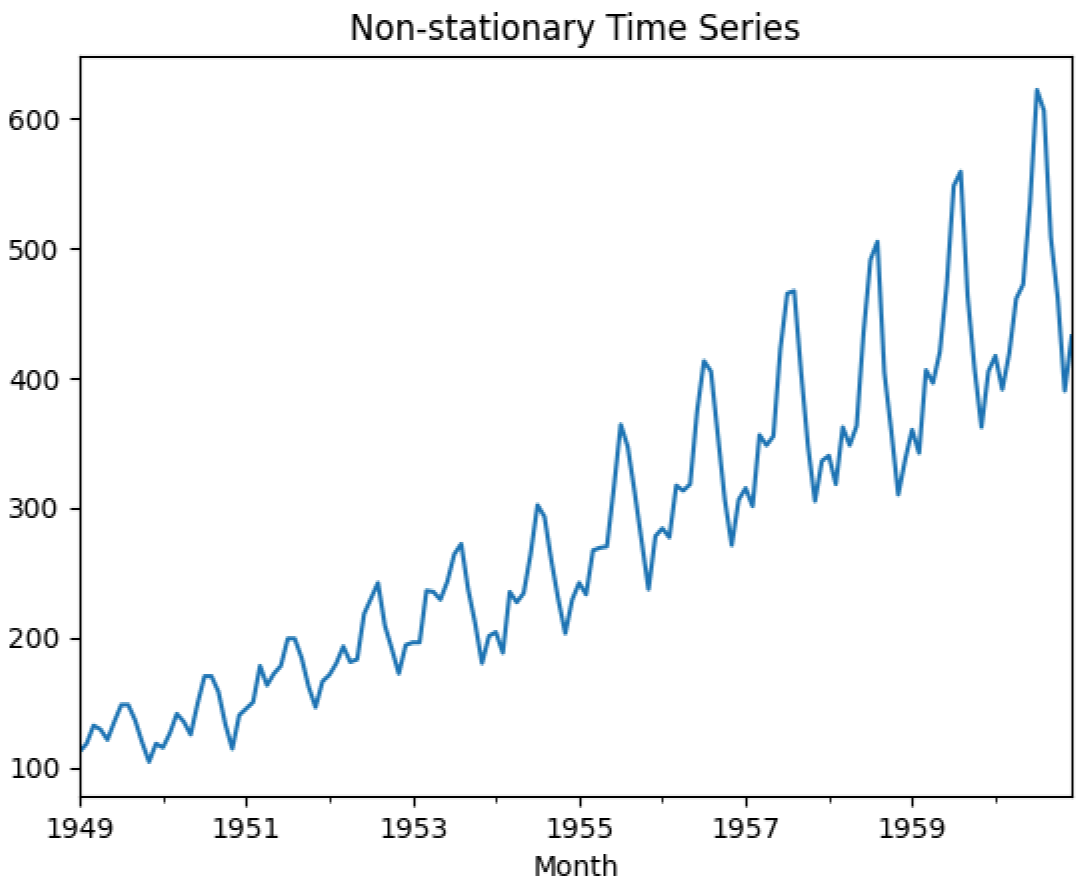
Many statistical models assume stable statistical properties. Without stationarity:

- Model parameters become unreliable

- Forecasts degrade quickly

Visual Example

In [4]:
```
data["Passengers"].plot(title="Non-stationary Time Series")
plt.show()
```

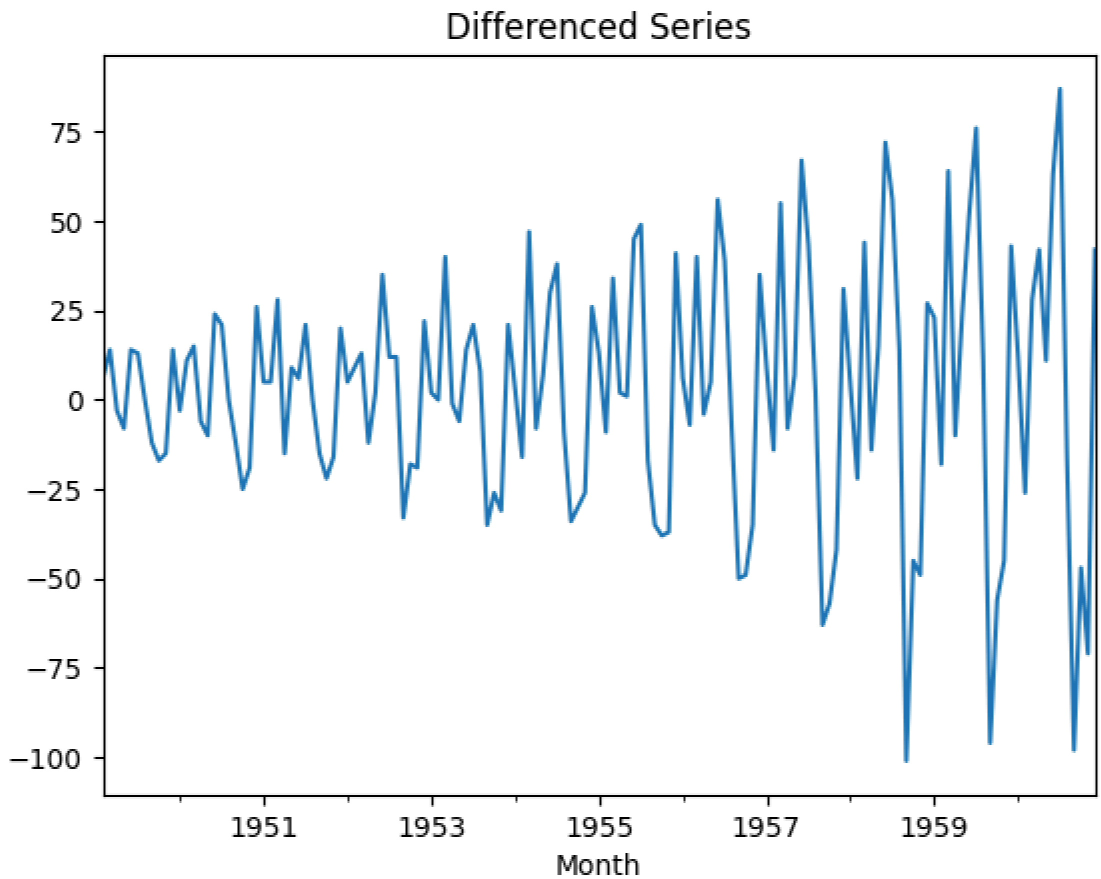


Based on the plot, the time series data shows a clear upward trend, with some seasonal variations. The data also shows some volatility, with some periods of high volatility and some periods of low volatility.

Differencing to Achieve Stationarity

Differencing is a common technique used in time series analysis to remove trends and seasonality from the data. It returns a new series by subtracting the previous value from the current value.
For example, if the series is [1, 2, 3, 4, 5], then the differenced series is [1, 1, 1, 1]

In [ ]:
```
diff_series = data["Passengers"].diff().dropna()

diff_series.plot(title="Differenced Series")
plt.show()
```



The plot after differencing shows a more stable pattern, which is a characteristic of a stationary time series. The mean and variance are more stable over time.

Autocorrelation and Partial Autocorrelation

1. **ACF**: correlation with past values
2. **PACF**: direct correlation excluding intermediate lags

Autocorrelation Function (ACF)

What ACF Measures

The **ACF** measures how strongly a time series is correlated with **its own past values**.

At lag (k):

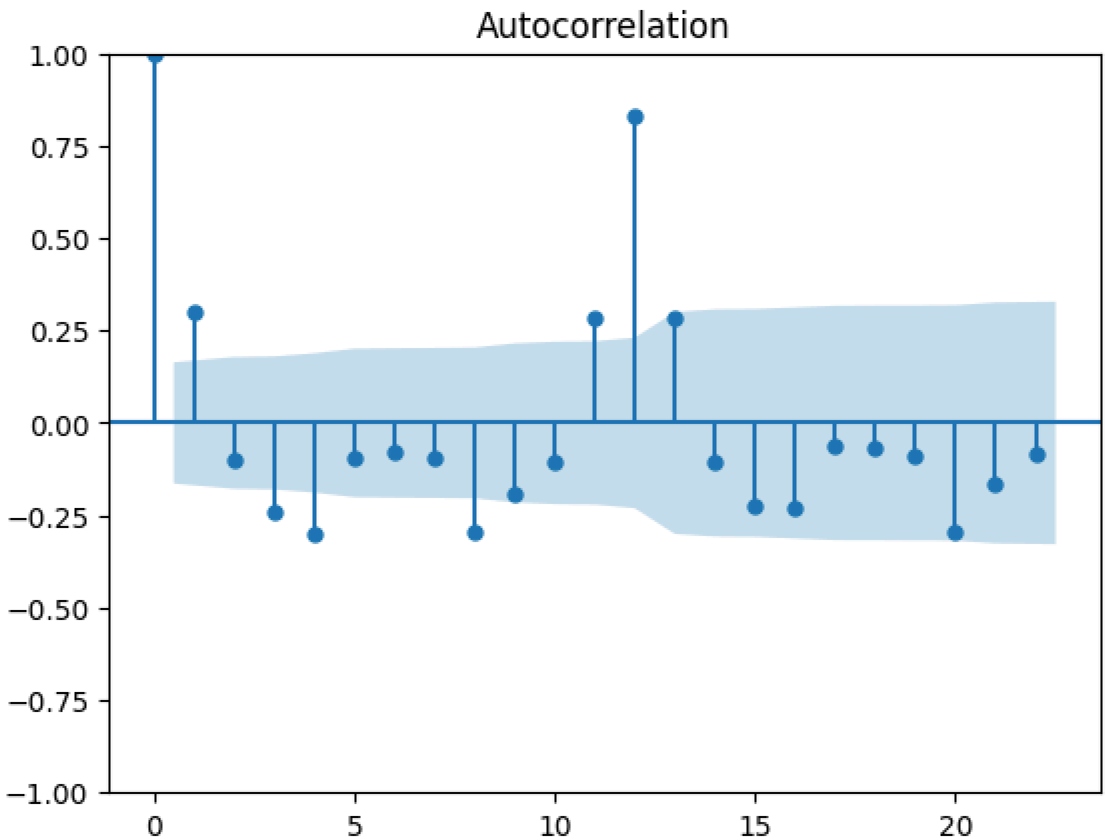
$$\mathrm{ACF}(k) = \mathrm{corr}(y_t, y_{t-k})$$

Intuition

- “How much does today look like *k* periods ago?”
- Measures **total correlation**, including indirect effects

Python Example

In [32]:
```
from statsmodels.graphics.tsaplots import plot_acf

plot_acf(diff_series)
plt.show()
```



How to Interpret ACF

Pattern	Interpretation
Slow decay	Series is non-stationary
Sharp cutoff after lag q	MA(q) process
Spikes at seasonal lags	Seasonal MA component
All values near zero	White noise

Key Insight: ACF is primarily used to identify the **MA (Moving Average)** part of ARIMA.

Partial Autocorrelation Function (PACF)

What PACF Measures

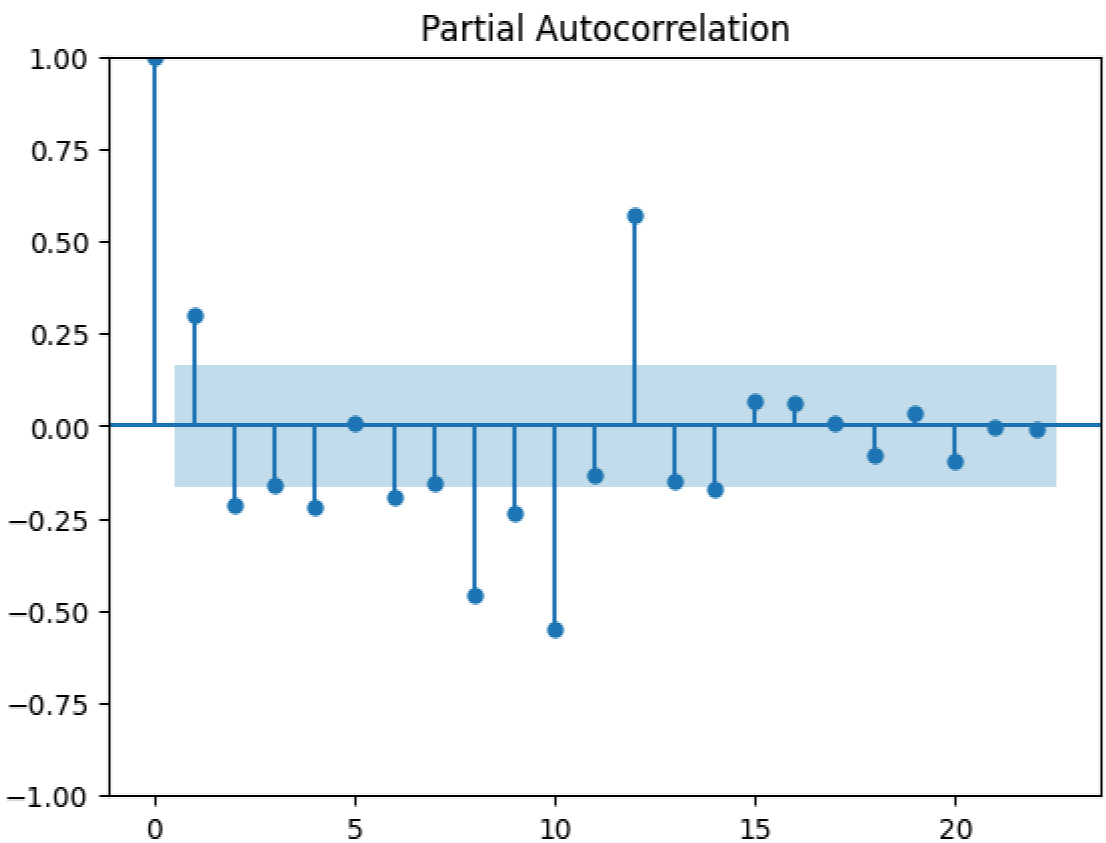
The **PACF** measures the **direct** relationship between (y_t) and (y_{t-k}), **removing the influence of intermediate lags**.

Intuition

- “Does lag k matter *on its own*, or only because of closer lags?”

Python Example

In [33]:
```
from statsmodels.graphics.tsaplots import plot_pacf

plot_pacf(diff_series)
plt.show()
```



How to Interpret PACF

Pattern	Interpretation
Sharp cutoff after lag p	AR(p) process
Gradual decay	MA behavior
Seasonal spikes	Seasonal AR component

Key Insight: PACF is primarily used to identify the **AR (Autoregressive)** part of ARIMA.

Why ACF and PACF Are Different (Critical Understanding)

- **ACF** includes **direct + indirect** correlations
- **PACF** isolates **direct** correlations only

Example: If:

- (y_t) depends on (y_{t-1})
- (y_{t-1}) depends on (y_{t-2})

Then:

- ACF at lag 2 may be high
- PACF at lag 2 may be near zero

This distinction is what allows AR and MA terms to be identified separately.

Summary

- **ACF** shows how similar the current month is to past months. → Strong spikes, especially at **12 months**, mean the data has a **clear yearly seasonal pattern**. → Correlations stay high for many lags, showing the series has a **trend** and is **not stationary**.

- **PACF** shows which past months **directly influence** the current month. → A strong spike at **lag 1** means this month depends on **last month**. → A strong spike at **lag 12** means this month also depends on **the same month last year**.

- **What we learn from both plots:** Passenger numbers depend on **recent history (last month)** and **seasonal history (last year)**.

- **Why SARIMA is a good choice:** SARIMA is built for time-series data that has a **trend** and **repeating seasonal patterns**. It can handle both **short-term changes** and **yearly seasonality**, which is exactly what the ACF and PACF plots reveal in the AirPassengers dataset.

CLASSICAL STATISTICAL MODELS

ARIMA Models

Model Structure

ARIMA(p, d, q)

Parameter	Meaning	How Chosen
p	AR terms	From PACF
d	Differencing	From stationarity
q	MA terms	From ACF

p = 2 means we need to use the last 2 data points to predict the next data point
d = 1 means we need to difference the data once, for example if we have 100 data points, we will have 99 data points after differencing
q = 2 means we need to use the last 2 data points to predict the next data point

Mathematical Form

$y_t = c + \sum_{i=1}^{p} \phi_i y_{t-i} + \sum_{j=1}^{q} \theta_j \varepsilon_{t-j} + \varepsilon_t$

This equation shows that the current value y_t is made up of a constant term (c), plus a contribution from **past values** of the series $(\sum_{i=1}^{p} \phi_i, y_{t-i})$ (where (y_{t-i}) are earlier observations and (ϕ_i) are their weights), plus a contribution from **past errors/shocks** $(\sum_{j=1}^{q} \theta_j, \varepsilon_{t-j})$ (where (ε_{t-j}) are previous forecast errors and (θ_j) are their weights), and finally the **current random noise** (ε_t). This structure aligns with what we see in the ACF and PACF plots—dependence on recent history (e.g., lag 1) and seasonal repetition (e.g., lag 12)—which is why a seasonal extension such as SARIMA is appropriate for the AirPassengers series.

Python Example

The code below shows the application of an **ARIMA(2,1,2)** model to the AirPassengers time series. The model first applies one level of differencing to remove the overall upward trend and achieve approximate stationarity. It then captures short-term dependencies by modeling the influence of passenger counts from the previous two months (autoregressive terms) and the impact of recent shocks or errors from the past two periods (moving-average terms). This approach focuses on modeling non-seasonal, month-to-month dynamics in the data and does not explicitly account for the strong yearly seasonal pattern characteristic of the AirPassengers series.

In [ ]:
```
from statsmodels.tsa.arima.model import ARIMA

model = ARIMA(data["Passengers"], order=(2,1,2))
model_fit = model.fit()

print(model_fit.scummary())
```

```
                               SARIMAX Results
==============================================================================
Dep. Variable:             Passengers   No. Observations:                  144
Model:                 ARIMA(2, 1, 2)   Log Likelihood                -671.673
Date:                Tue, 16 Dec 2025   AIC                           1353.347
Time:                        23:10:20   BIC                           1368.161
Sample:                    01-01-1949   HQIC                          1359.366
                         - 12-01-1960
Covariance Type:                  opg
==============================================================================
                 coef    std err          z      P>|z|      [0.025      0.975]
------------------------------------------------------------------------------
ar.L1          1.6850      0.020     83.059      0.000       1.645       1.725
ar.L2         -0.9548      0.017    -55.419      0.000      -0.989      -0.921
ma.L1         -1.8432      0.125    -14.798      0.000      -2.087      -1.599
ma.L2          0.9953      0.135      7.374      0.000       0.731       1.260
sigma2       665.9568    114.104      5.836      0.000     442.316     889.597
===================================================================================
Ljung-Box (L1) (Q):                   0.30   Jarque-Bera (JB):                 1.84
Prob(Q):                              0.59   Prob(JB):                         0.40
Heteroskedasticity (H):               7.38   Skew:                             0.27
Prob(H) (two-sided):                  0.00   Kurtosis:                         3.14
===================================================================================

Warnings:
[1] Covariance matrix calculated using the outer product of gradients (complex-ste
p).
```

```
c:\Users\me\venv\Lib\site-packages\statsmodels\tsa\base\tsa_model.py:473: ValueWarni
ng: No frequency information was provided, so inferred frequency MS will be used.
  self._init_dates(dates, freq)
c:\Users\me\venv\Lib\site-packages\statsmodels\tsa\base\tsa_model.py:473: ValueWarni
ng: No frequency information was provided, so inferred frequency MS will be used.
  self._init_dates(dates, freq)
c:\Users\me\venv\Lib\site-packages\statsmodels\tsa\base\tsa_model.py:473: ValueWarni
ng: No frequency information was provided, so inferred frequency MS will be used.
  self._init_dates(dates, freq)
c:\Users\me\venv\Lib\site-packages\statsmodels\base\model.py:607: ConvergenceWarnin
g: Maximum Likelihood optimization failed to converge. Check mle_retvals
  warnings.warn("Maximum Likelihood optimization failed to "
```

What this *really* is (in one sentence)

ARIMA is just regression on time, where the predictors are **past values and past errors instead of columns in a table**.

Connect it to regular regression (intuition first)

In ordinary regression you write:

> $y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \text{error}$

In ARIMA, you write:

> $y_t = \text{constant} + (\text{past } y\text{'s}) + (\text{past errors}) + \text{error}$

Same idea. Different inputs.

Translate the output into regression language

1. The coefficients (`ar.L1` , `ar.L2` , `ma.L1` , `ma.L2`)

These are just **regression coefficients**.

- `ar.L1 = 1.685` → "Last month strongly predicts this month"
- `ar.L2 = -0.955` → "Two months ago has a correcting (negative) effect"
- `ma.L1` and `ma.L2` → "If the model made errors recently, those errors help adjust today's prediction"

This is no different from saying:

> feature_1 coefficient = 1.68 feature_2 coefficient = −0.95

The "features" just happen to be **lags of the same variable**.

2. p-values and z-scores

Exactly the same as regression:

- Low p-value → coefficient matters
- High p-value → coefficient doesn't matter

Here, all p-values ≈ 0 → **the model found strong structure in time**.

3. Residual diagnostics = model sanity checks

Think of these as **post-fit validation**, not magic.

- **Ljung–Box** → "Did we leave obvious patterns behind?" (No → good)
- **Jarque–Bera** → "Do residuals look roughly normal?" (Yes → OK)
- **Heteroskedasticity test** → "Does variance change over time?" (Yes → data grows over time, known issue with AirPassengers)

Same role as:

- residual plots
- normality checks
- constant variance checks

Why it *feels* alien

1. **No feature matrix**

    - Features are created internally from time lags.
2. **Statistical language**
    - Econometrics emphasizes inference, not prediction accuracy.
3. **Diagnostics-heavy**
    - Much more focus on *why* a model works than ML usually gives.

How this fits into machine learning

ARIMA is:

- ✔ supervised learning
- ✔ parametric
- ✔ linear model
- ✔ maximum likelihood–based

It sits closer to:

- linear regression
- ridge regression
- Kalman filters

than to:

- neural networks
- random forests

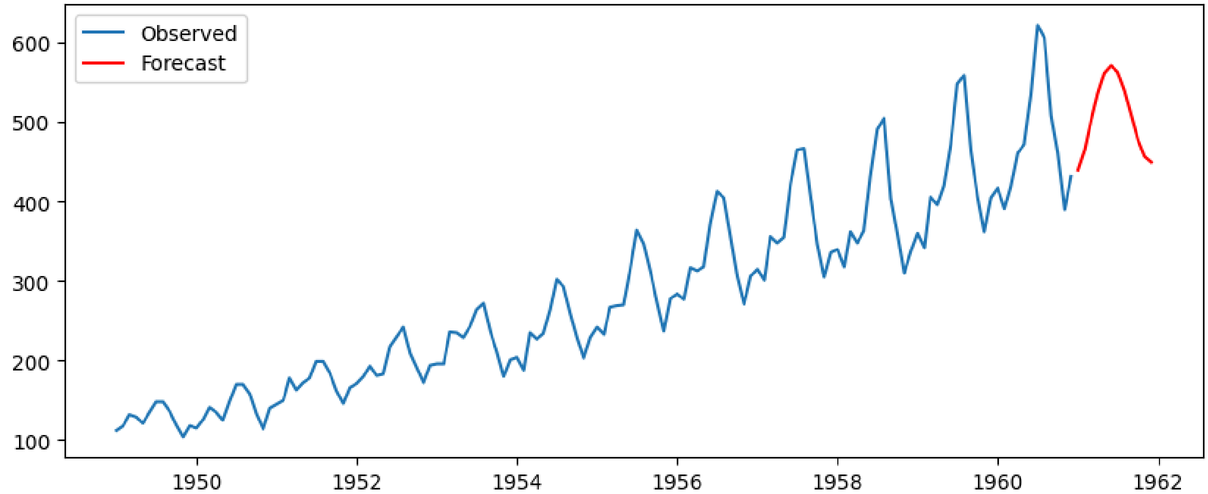
One grounding sentence you can remember

> ARIMA is just linear regression where the predictors are the past of the same variable, and the output is wrapped in statistical diagnostics instead of ML metrics.

Forecasting with ARIMA

In [11]:

```
"""
This cell performs time series forecasting using the previously fitted ARIMA model.
It generates a 12-step forecast and then visualizes both the historical 'Passengers
and the new forecast on a single plot for comparison.
"""
forecast = model_fit.forecast(steps=12)

plt.figure(figsize=(10,4))
plt.plot(data.index, data["Passengers"], label="Observed")
plt.plot(forecast.index, forecast, label="Forecast", color="red")
plt.legend()
plt.show()
```



Strengths

- Interpretable
- Statistically grounded

Limitations

- Linear
- Sensitive to assumptions
- Poor with many external variables such as holidays, promotions, etc.

Seasonal ARIMA (SARIMA)

Handles seasonality explicitly.

In [13]:
```
from statsmodels.tsa.statespace.sarimax import SARIMAX

model = SARIMAX(
    data["Passengers"],
    order=(1,1,1),
    seasonal_order=(1,1,1,12)
)

fit = model.fit()
fit.summary()
```

```
c:\Users\me\venv\Lib\site-packages\statsmodels\tsa\base\tsa_model.py:473: ValueWarni
ng: No frequency information was provided, so inferred frequency MS will be used.
  self._init_dates(dates, freq)
c:\Users\me\venv\Lib\site-packages\statsmodels\tsa\base\tsa_model.py:473: ValueWarni
ng: No frequency information was provided, so inferred frequency MS will be used.
  self._init_dates(dates, freq)
```

Out[13]:

SARIMAX Results

Dep. Variable:	Passengers	**No. Observations:**	144
Model:	SARIMAX(1, 1, 1)x(1, 1, 1, 12)	**Log Likelihood**	-506.149
Date:	Tue, 16 Dec 2025	**AIC**	1022.299
Time:	21:57:56	**BIC**	1036.675
Sample:	01-01-1949	**HQIC**	1028.140
	- 12-01-1960		
Covariance Type:	opg		

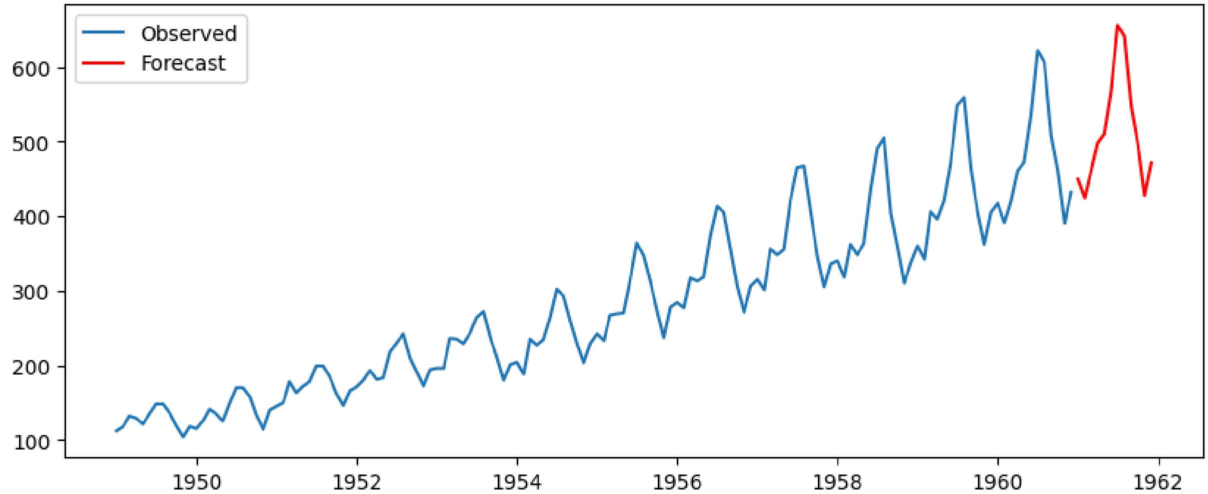
	coef	**std err**	**z**	**P>|z|**	**[0.025**	**0.975]**
ar.L1	-0.1272	0.356	-0.358	0.721	-0.825	0.570
ma.L1	-0.2149	0.325	-0.660	0.509	-0.853	0.423
ar.S.L12	-0.9272	0.214	-4.341	0.000	-1.346	-0.509
ma.S.L12	0.8394	0.309	2.717	0.007	0.234	1.445
sigma2	130.7826	15.420	8.481	0.000	100.559	161.006

Ljung-Box (L1) (Q):	0.00	**Jarque-Bera (JB):**	7.05
Prob(Q):	0.99	**Prob(JB):**	0.03
Heteroskedasticity (H):	2.65	**Skew:**	0.13
Prob(H) (two-sided):	0.00	**Kurtosis:**	4.11

Warnings:
[1] Covariance matrix calculated using the outer product of gradients (complex-step).

In [15]:

```
forecast = fit.forecast(steps=12)

plt.figure(figsize=(10,4))
plt.plot(data.index, data["Passengers"], label="Observed")
plt.plot(forecast.index, forecast, label="Forecast", color="red")
plt.legend()
plt.show()
```



PART II — MACHINE LEARNING APPROACHES

Why Machine Learning for Time Series?

Statistical models:

- Focus on *past values only*
- Require strong assumptions

ML models:

- Handle non-linearity
- Incorporate many features
- Scale well with data

Feature Engineering for ML

ML models **do not understand time**. Time information must be converted into features.

Lag Features

In [17]:
```
df = data.copy()

df["lag_1"] = df["Passengers"].shift(1)
df["lag_12"] = df["Passengers"].shift(12)

df.dropna(inplace=True)
```

Display new dataframe with lags

In [20]:
```
df
```

Out[20]:

Month	Passengers	lag_1	lag_12
1950-01-01	115	118.0	112.0
1950-02-01	126	115.0	118.0
1950-03-01	141	126.0	132.0
1950-04-01	135	141.0	129.0
1950-05-01	125	135.0	121.0
...	...	...	...
1960-08-01	606	622.0	559.0
1960-09-01	508	606.0	463.0
1960-10-01	461	508.0	407.0
1960-11-01	390	461.0	362.0
1960-12-01	432	390.0	405.0

132 rows × 3 columns

Train-Test Split (Time-Aware)

In [ ]:
```
train = df.iloc[:-12] # All data except last 12 months
test = df.iloc[-12:] # last 12 months for testing

X_train = train.drop("Passengers", axis=1) # Drop the target column
y_train = train["Passengers"] # Target column

X_test = test.drop("Passengers", axis=1)
y_test = test["Passengers"]
```

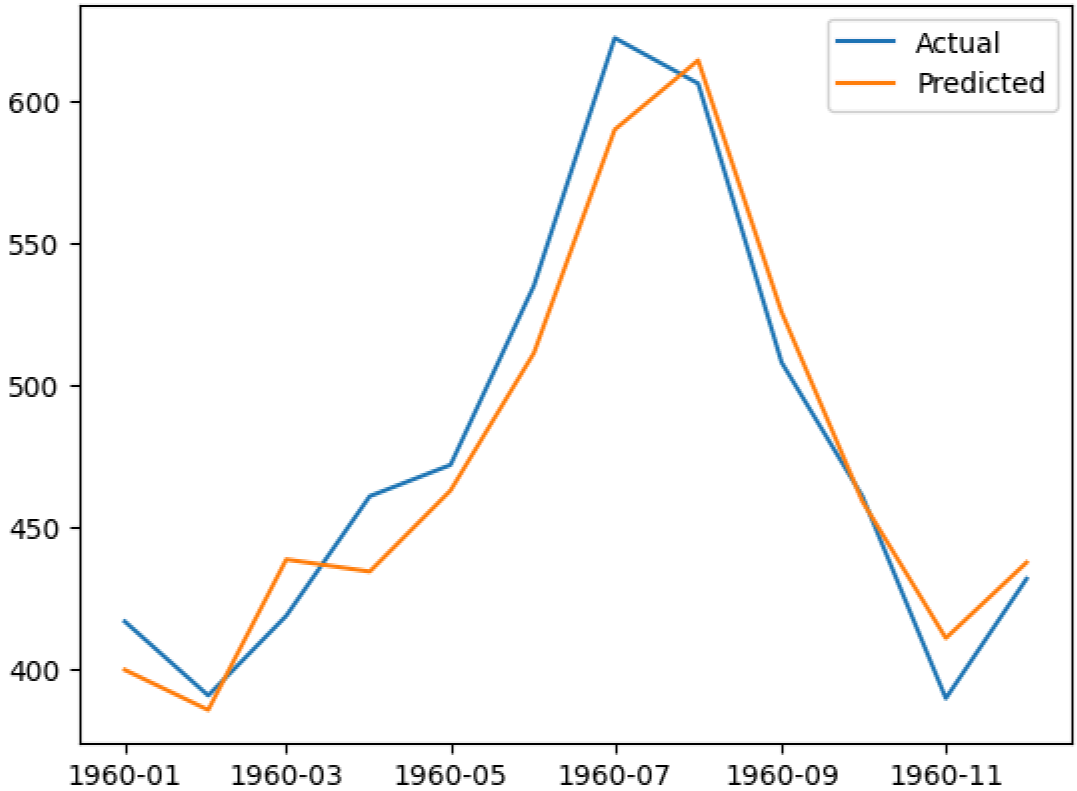
Regression Model (Baseline ML)

In [21]:
```
from sklearn.linear_model import LinearRegression

model = LinearRegression()
model.fit(X_train, y_train)

predictions = model.predict(X_test)
```

Plot the actual vs predicted values

In [22]:
```
plt.plot(test.index, y_test, label="Actual")
plt.plot(test.index, predictions, label="Predicted")
```

```
plt.legend()
plt.show()
```



Evaluate the performance of the model

In [29]:
```
import numpy as np

# Compute R-Squared
from sklearn.metrics import r2_score
r2 = r2_score(y_test, predictions)
print(f'R-squared: {r2:.4f}')

# Compute RMSE
from sklearn.metrics import mean_squared_error
rmse = np.sqrt(mean_squared_error(y_test, predictions))
print(f'RMSE: {rmse:.4f}')
```

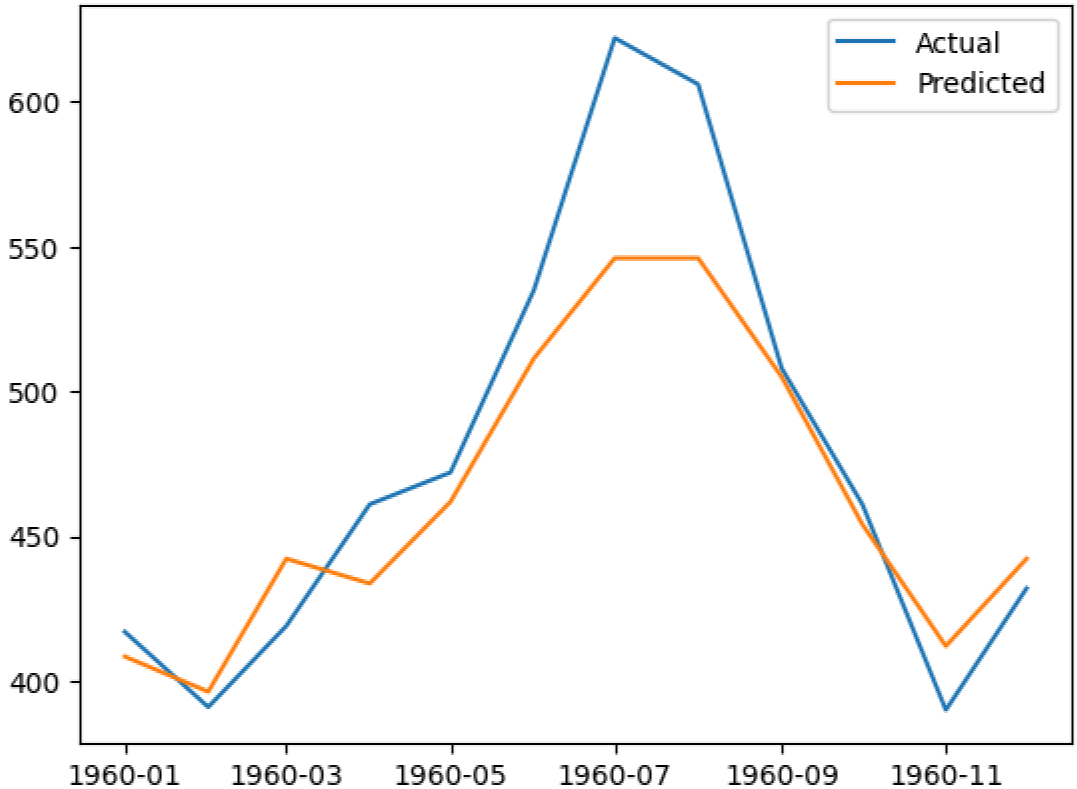
```
R-squared: 0.9406
RMSE: 18.1356
```

Tree-Based Models (Nonlinear Patterns)

In [24]:
```
from sklearn.ensemble import RandomForestRegressor

rf = RandomForestRegressor(n_estimators=200, random_state=42)
rf.fit(X_train, y_train)

rf_preds = rf.predict(X_test)
```

Plot the time series data

In [25]:
```
plt.plot(test.index, y_test, label="Actual")
plt.plot(test.index, rf_preds, label="Predicted")
plt.legend()
plt.show()
```



Evaluate the performance of the model

In [28]:
```
import numpy as np

# Compute R-Squared
from sklearn.metrics import r2_score
r2 = r2_score(y_test, rf_preds)
print(f'R-squared: {r2:.4f}')

# Compute RMSE
from sklearn.metrics import mean_squared_error
rmse = np.sqrt(mean_squared_error(y_test, rf_preds))
print(f'RMSE: {rmse:.4f}')
```

```
R-squared: 0.8184
RMSE: 31.7142
```

Gradient Boosting (XGBoost-style Logic)

Gradient Boosting Regressor

In [30]:
```
from sklearn.ensemble import GradientBoostingRegressor

gbr = GradientBoostingRegressor()
gbr.fit(X_train, y_train)

gbr_preds = gbr.predict(X_test)

import numpy as np

# Compute R-Squared
from sklearn.metrics import r2_score
r2 = r2_score(y_test, gbr_preds)
print(f'R-squared: {r2:.4f}')

# Compute RMSE
from sklearn.metrics import mean_squared_error
rmse = np.sqrt(mean_squared_error(y_test, gbr_preds))
print(f'RMSE: {rmse:.4f}')
```

```
R-squared: 0.8396
RMSE: 29.8079
```

Comparing Models

In [31]:
```
from sklearn.metrics import mean_absolute_error

print("Linear:", mean_absolute_error(y_test, predictions))
print("RF:", mean_absolute_error(y_test, rf_preds))
print("GBR:", mean_absolute_error(y_test, gbr_preds))
```

```
Linear: 15.640331783370726
RF: 23.024166666666677
GBR: 24.80970757043872
```

DEEP LEARNING (CONCEPTUAL)

Recurrent Neural Networks (LSTM)

Deep learning models:

- Learn sequences directly
- Capture long-term dependencies

Key requirements:

- Large datasets
- Careful scaling
- Computational resources

Typical workflow:

1. Window the series
2. Normalize
3. Train LSTM
4. Forecast iteratively

(Implementation usually done with TensorFlow or PyTorch.)

Statistical vs ML: When to Use What

Scenario	Best Approach
Small dataset, interpretability	ARIMA / SARIMA
Strong seasonality	SARIMA
Many external variables	ML
Non-linear patterns	Tree models
Large-scale forecasting	ML / Deep Learning
Regulatory / explainable needs	Statistical

Key Takeaways

- Time series analysis is about **structure + dependency**
- Statistical models explain *why*
- ML models focus on *accuracy*
- Feature engineering is essential for ML
- No single model is universally best