Lecture

In [1]:
```
import numpy as np

import IPython.display as dp
import matplotlib.pyplot as plt
import seaborn as sns

dp.set_matplotlib_formats("retina")
sns.set(style="whitegrid", font_scale=1.5)
sns.despine()

%matplotlib inline
```

```
/var/folders/33/j0cl7y453td68qb96j7bqcj4cf41kc/T/ipykernel_823/4025387531.
py:7: DeprecationWarning: `set_matplotlib_formats` is deprecated since IPy
thon 7.23, directly use `matplotlib_inline.backend_inline.set_matplotlib_f
ormats()`
  dp.set_matplotlib_formats("retina")
<Figure size 640x480 with 0 Axes>
```

Reminder

Suppose that Z_t is a WN with mean zero and variance σ_Z^2, then:

- $MA(q)$ process is

$$X_t = \beta_0 Z_t + \beta_1 Z_{t-1} + \ldots + \beta_q Z_{t-q},$$

  where βs are constants. Typically, the series is scaled so that $\beta_0 = 1$.

  Using backward shift operator B^j defined as $B^j X_t = X_{t-j}$, as follows:

$$X_t = (\beta_0 + \beta_1 B + \ldots + \beta_q B^q) Z_t = \theta(B) Z_t$$

- $MA(q)$ is always stationary, and sometimes **invertible**, when the complex roots of the equation $\theta(x) = 0$ lie outside the unit circles

ACF and stationarity of MA(1)

Consider $MA(1)$ process with $\beta_0 = 1, \beta_1 = \theta$:

$$X_t = Z_t + \theta Z_{t-1}$$

The ACF of $MA(q)$ is given by:

$$\rho(k) = \begin{cases} 1, & k = 0, \\ \sum\limits_{i=0}^{q-k} \beta_i \beta_{i+k} \Big/ \sum\limits_{i=0}^{q} \beta_i^2, & 1 \leqslant k \leqslant q, \\ 0, & k > q, \\ \gamma(-k), & k < 0 \end{cases}$$

The ACF of our $MA(1)$ is then given by:

$$\rho(k) = \begin{cases} 1, & k = 0, \\ \theta/(1+\theta^2), & k = \pm 1, \\ 0, & \text{else} \end{cases}$$

Check ACF for $\tilde{\theta} = \frac{1}{\theta}$.

The process expressed with backward shift operator is:

$$X_t = (\beta_0 + \beta_1 B + \ldots + \beta_q B^q) Z_t = \theta(B) Z_t$$

Then the polynomial for our $MA(1)$ is:

$$\theta(B) = 1 + \theta B$$

Its root is $-\frac{1}{\theta}$, and it lies outside unit circle if $|\theta| < 1$. So $MA(1)$ is invertible if $|\theta| < 1$.

Reminder

- $AR(p)$ process is

  $$X_t = \alpha_1 X_{t-1} + \ldots + \alpha_p X_{t-p} + Z_t$$

  Using backward shift operator B^j:

  $$(1 - \alpha_1 B - \ldots - \alpha_p B^p) X_t = \phi(B) X_t = (1 + \beta_1 B + \beta_2 B^2 + \ldots)^{-1} Z_t = f(B).$$

- $AR(p)$ is not always stationary! There are two ways to verify stationarity:

  1. Verify that ACVF/ACF only depends on $t_2 - t_1$
  2. Verify that the complex roots of $\phi(x) = 0$ lie outside of the unit circle

ACF and stationarity of AR(1)

Consider $AR(1)$ process:

$$X_t = \alpha X_{t-1} + Z_t$$

With backward shift operator, we get

$$Z_t = (1 - \alpha B) X_t$$

or

$$X_t = (1 - \alpha B)^{-1} Z_t = (1 + \alpha B + \alpha^2 B + \ldots) Z_t$$

From this representation:

$$\begin{aligned} \mathbb{E}[X_t] &= 0, \\ \mathrm{Var}(X_t) &= \sigma_Z^2 (1 + \alpha^2 + \alpha^4 + \ldots) = \sigma_Z^2 / (1 - \alpha^2) \end{aligned}$$

ACVF is given by:

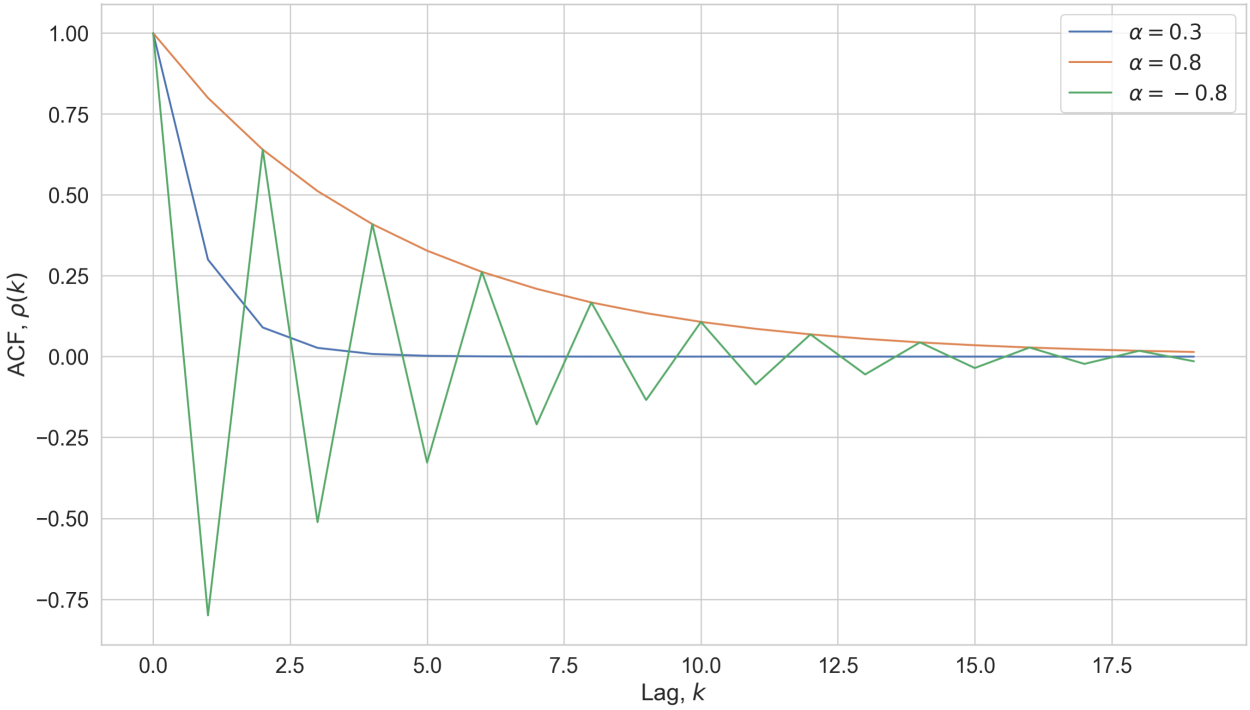
$$\gamma(k) = \mathbb{E}[X_t X_{t+k}] = \mathbb{E}\left[\left(\sum_{i=0}^{\infty} \alpha^i Z_{t-i} \right) \left(\sum_{j=0}^{\infty} \alpha^j Z_{t+k-j} \right) \right] = \sigma_Z^2 \sum_{i=0}^{\infty} \alpha^i \alpha^{k+i} = \sigma_Z^2 \alpha^k$$

ACF is then given by

$$\rho(k) = \gamma(k) / \sigma_X^2 = \alpha^k$$

The process is stationary provided that $|\alpha| < 1$.

In [2]:
```
kk = np.arange(20)
acf_ar_1_03 = 0.3 ** kk
acf_ar_1_08 = 0.8 ** kk
acf_ar_1_m08 = (- 0.8) ** kk
fig, ax = plt.subplots(figsize=(16,9))
ax.plot(kk, acf_ar_1_03, label="$\\alpha = 0.3$")
ax.plot(kk, acf_ar_1_08, label="$\\alpha = 0.8$")
ax.plot(kk, acf_ar_1_m08, label="$\\alpha = - 0.8$")
ax.set_xlabel("Lag, $k$")
ax.set_ylabel("ACF, $\\rho(k)$")
ax.legend();
```



ACF and stationarity of AR(2)

Consider $AR(2)$ process

$$X_t = \alpha_1 X_{t-1} + \alpha_2 X_{t-2} + Z_t$$

With backward shift operator, we get

$$Z_t = (1 - \alpha B - \alpha^2 B^2) X_t$$

As discussed last time, the ACF can be found from Yule-Walker equations

$$\rho(k) = \alpha_1 \rho(k-1) + \ldots + \alpha_p \rho(k-p)$$

for which the general solution is

$$\rho(k) = A_1 \pi_1^{|k|} + \ldots + A_p \pi_p^{|k|}$$

where π_i are the roots of the so-called auxiliary equation

$$y^p - \alpha_1 y^{p-1} - \ldots - \alpha_p = 0$$

In case of $AR(2)$, the auxiliary equation writes

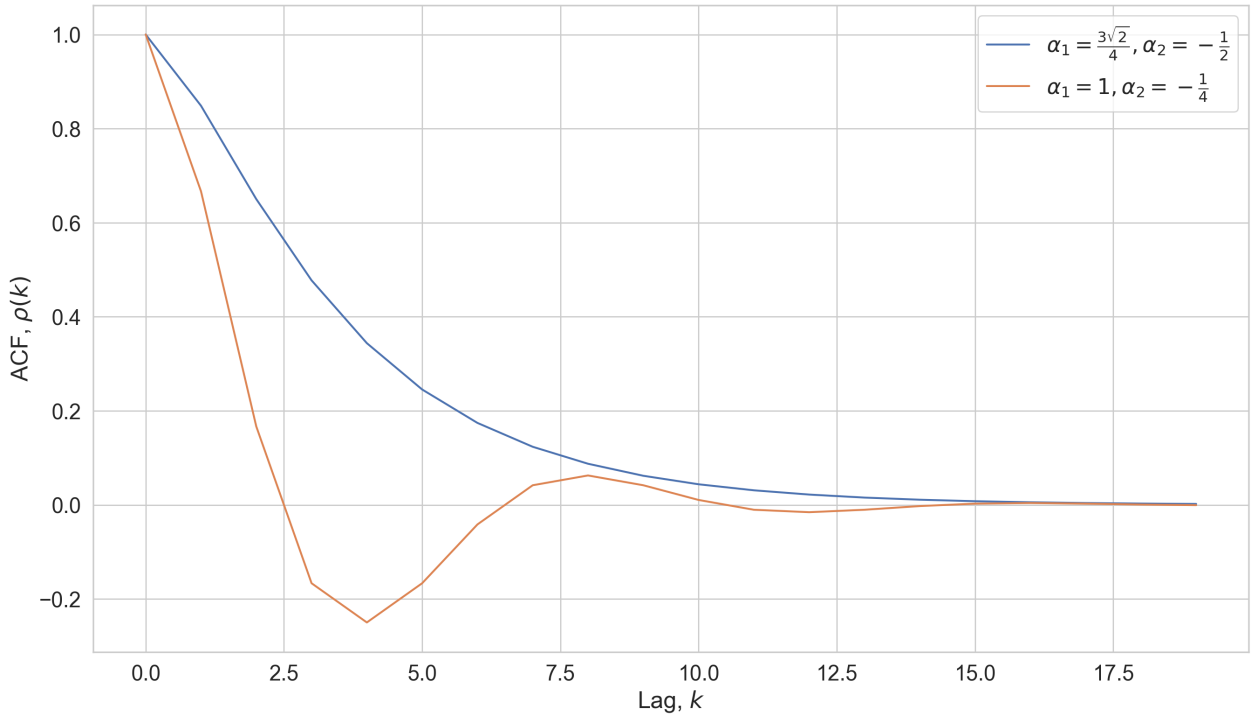
$$y^2 - \alpha_1 y - \alpha_2 = 0$$

So we get stationarity if

$$\left|\frac{\alpha_1 \pm \sqrt{\alpha_1^2 + 4\alpha_2}}{2}\right| < 1 \Leftrightarrow \begin{cases} \alpha_1 + \alpha_2 & < 1, \\ \alpha_1 - \alpha_2 & > -1, \\ \alpha_2 & > -1 \end{cases}$$

If the discriminant is positive, thes roots are real, and the ACF behaves like exponential decay. If discriminant is negative, the roots are complex, and ACF behaves like damped sinusoid.

In [3]:
```
kk = np.arange(20)

alpha_1 = 3 * np.sqrt(2) / 4
alpha_2 = - 0.25
pi_1 = (alpha_1 + np.sqrt(alpha_1 ** 2 + 4 * alpha_2)) / 2
pi_2 = (alpha_1 - np.sqrt(alpha_1 ** 2 + 4 * alpha_2)) / 2
a_1 = (alpha_1 / (1 - alpha_2) - pi_2) / (pi_1 - pi_2)
a_2 = 1 - a_1
acf_ar_2_sqrt2_m14 = a_1 * pi_1 ** kk + a_2 * pi_2 ** kk

acf_ar_2_1_m12 = (1 / np.sqrt(2)) ** kk * (np.cos(np.pi * kk / 4) + np.si
fig, ax = plt.subplots(figsize=(16,9))
ax.plot(kk, acf_ar_2_sqrt2_m14, label="$\\alpha_1 = \\frac{3 \\sqrt{2}}{4
ax.plot(kk, acf_ar_2_1_m12, label="$\\alpha_1 = 1, \\alpha_2 = - \\frac{1
ax.set_xlabel("Lag, $k$")
ax.set_ylabel("ACF, $\\rho(k)$")
ax.legend();
```



ARMA

A mixed autoregressive/moving-average process containing p AR terms and q MA

terms is said to be an $ARMA(p, q)$:

$$X_t = \alpha_1 X_{t-1} + \ldots + \alpha_p X_{t-p} + Z_t + \beta_1 Z_{t-1} + \ldots + \beta_q Z_{t-q}$$

Using backshift operator:

$$\phi(B) X_t = \theta(B) Z_t$$

where

$$\phi(B) = 1 - \alpha_1 B - \ldots - \alpha_p B^p,$$
$$\theta(B) = 1 + \beta_1 B + \ldots + \beta_q B^q$$

The conditions for stationarity and invertibility are the same as for a pure AR or pure MA process, namely, ARMA will be stationary if roots of

$$\phi(B) = 0$$

lie outside the unit circle, and will be invertible if roots of

$$\theta(B) = 0$$

lie outside the unit circe.

A note on mean

If mean of the series is $\mu \neq 0$, we can do the following e.g. for $AR(p)$ process:

$$X_t - \mu = \alpha_1 (X_{t-1} - \mu) + \ldots + \alpha_p (X_{t-p} - \mu) + Z_t$$

or

$$X_t = \alpha_0 + \alpha_1 X_{t-1} + \ldots + \alpha_p X_{t-p} + Z_t$$

where $\alpha_0 = \mu(1 - \alpha_1 - \ldots - \alpha_p)$.

All properties persist. Analagous reasoning for MA and ARMA.

Wold's theorem

A famous result states that every stationary time series can be written as the sum of two infinite time series, one purely deterministic and one purely stochastic, i.e. an infinite-order ARMA process. It should be taken with a grain of salt, because in reality we do not use inifinite order processes, as there will be too many parameters to estimate. However, we can hope that we can approximate a stationary process with a sufficiently high-order ARMA process with good approximation error.

Differencing

In practice most time series are non-stationary. In order to fit a stationary model to a non-stationary series X_t, we need to remove non-stationarity. One quick way to deal with certain types of non-stationarity is to perform differencing of order d, i.e. replace the process with its differences $\nabla^d X_t = X_t - X_{t-d}$. If we then fit a model on the differenced time series, such a model would be called an **integrated** model, as the outputs of this model have to be summed or "integrated" to provide a model for the original series.

ARIMA

Consider $W_t = \nabla^d X_t = (1-B)^d X_t$, and then an $ARMA(p,q)$ model of W_t:

$$W_t = \alpha_1 W_{t-1} + \ldots + \alpha_p W_{t-p} + Z_t + \beta_1 Z_{t-1} + \ldots + \beta_q Z_{t-q}$$

Using backward shift operator, we write:

$$\phi(B) W_t = \theta(B) Z_t$$

or

$$\phi(B)(1-B)^d X_t = \theta(B) Z_t$$

The former equation is $ARMA(p,q)$ of W_t, while the latter equation is $ARIMA(p,d,q)$ of X_t.

SARIMA

ARIMA models can be generalized to include seasonal terms. Consider series X_t with seasonal cycle of length S. Let's take $ARMA(p,q)$ model:

$$X_t = \alpha_1 X_{t-1} + \ldots + \alpha_p X_{t-p} + Z_t + \beta_1 Z_{t-1} + \ldots + \beta_q Z_{t-q}$$

and add P AR components:

$$+\alpha_S X_{t-S} + \alpha_{2S} X_{t-2S} + \ldots + \alpha_{PS} X_{t-PS}$$

and add Q MA components:

$$+\beta_S Z_{t-S} + \beta_{2S} Z_{t-2S} + \ldots + \beta_{PS} Z_{t-QS}$$

Such model is called $SARMA(p,q) \times (P,Q)$. We can get model $SARIMA(p,d,q) \times (P,D,Q)$, if we apply additionally differencing of order d and

seasonal differencing of order D to the series.

SARIMA(p, d, q) x (P, D, Q)

$$\begin{aligned} X_t =& \alpha_0 + \alpha_1 X_{t-1} + \ldots + \alpha_p X_{t-p} + Z_t + \beta_1 Z_{t-1} + \ldots + \beta_q Z_{t-q} + \\ &+ \alpha_S X_{t-S} + \alpha_{2S} X_{t-2S} + \ldots + \alpha_{PS} X_{t-PS} + \\ &+ \beta_S Z_{t-S} + \beta_{2S} Z_{t-2S} + \ldots + \beta_{PS} Z_{t-QS} \end{aligned}$$

where $\alpha_0 = \mu(1 - \alpha_1 - \ldots - \alpha_p)$

Estimating parameters

- α_0, α, β
- d, D
- q, Q
- p, P

Estimating the mean

$$\hat{\mu} = \frac{1}{T} \sum_{t=1}^{T} X_t$$

Must be taken with care:

- Only makes sense for stationary processes
- If there is autocorrelation, the variance of the estimator depends on it
- Not necessarily that $\hat{\mu} \to \mathbb{E}[X_t]$, requires property called **ergodicity**, usually OK for stationary processes

Estimating AR process

Consider $AR(p)$ process

$$X_t = \alpha_0 + \alpha_1 X_{t-1} + \ldots + \alpha_p X_{t-p} + Z_t$$

Note that it is linear in X_t. Thanks to that, we have many ways to do the estimation:

- Least squares:

$$\min_{\alpha_0, \alpha_1, \ldots, \alpha_p} \sum_{t=p+1}^{T} \left[X_t - \alpha_0 - \alpha_1 X_{t-1} - \ldots - \alpha_p X_{t-p} \right]^2$$

- Maximum likelihood:

$$\min_{\alpha_0,\alpha_1,\dots,\alpha_p} \log \mathcal{L}(\alpha|X)$$

$$\mathcal{L}(\alpha|X) = \left(\prod_{t=p+1}^{T} p(X_t|X_{t-1},\dots,X_{p+1},\theta) \right) p(X_1,\dots,X_p|\theta)$$

$$\log \mathcal{L}(\alpha|X) = \underbrace{\sum_{t=p+1}^{T} \log p(X_t|X_{t-1},\dots,X_{p+1},\theta)}_{\text{conditional ll}} + \underbrace{\log p(X_1,\dots,X_p|\theta)}_{\text{marginal ll}}$$

  Will coincide with LSE if the noise is from Normal distribution.

Estimating AR process

Only good for large N and strong stationary properties:

- As ordinary regression treating each X_{t-k} as independent variable:

$$X_t \sim \alpha_0 + \alpha_1 X_{t-1} + \ldots + \alpha_p X_{t-p} + Z_t$$

- Using Yule-Walker equations:

$$\rho(k) = \alpha_1 \rho(k-1) + \ldots + \alpha_p \rho(k-p)$$

$$R\hat{\alpha} = r$$

  where r is a vector of sample autocorrelations and R is a matrix of them.

Example: AR(1)

Consider $AR(1)$ process

$$X_t = \mu + \alpha_1 X_{t-1} + Z_t$$

The estimation using least squares will give:

$$\hat{\mu} = \frac{\bar{X}_{(2)} - \hat{\alpha}_1 \bar{X}_{(1)}}{1 - \hat{\alpha}_1}$$

where $\bar{X}_{(1)}$ and $\bar{X}_{(2)}$ are the means of the first and last $(N-1)$ observations respectively. Obviously they are close, so often used estimator is the sample mean:

$$\hat{\mu} \approx \bar{X}$$

$$\hat{\alpha}_1 = \frac{\sum_{t=1}^{N-1}(X_t - \hat{\mu})(X_{t+1} - \hat{\mu})}{\sum_{t=1}^{N-1}(X_t - \hat{\mu})^2} \approx \frac{\sum_{t=1}^{N-1}(X_t - \bar{X})(X_{t+1} - \bar{X})}{\sum_{t=1}^{N-1}(X_t - \bar{X})^2} \approx \frac{\sum_{t=1}^{N-1}(X_t - \bar{X}}{\sum_{t=1}^{N}(X}$$

Estimating MA process

Consider $MA(q)$ process:

$$X_t = \alpha_0 + \beta_1 Z_{t-1} + \ldots + \beta_q Z_{t-q}$$

Unfortunately, for such a process, we can not express the residual as a linear function of observed X_ts. Hence, only the maximum likelihood method is applicable.

$$\min_{\alpha_0, \alpha_1, \ldots, \alpha_p} \log \mathcal{L}(\alpha|X)$$

$$\mathcal{L}(\alpha|X) = \left(\prod_{t=p+1}^{T} p(X_t|X_{t-1}, \ldots, X_{p+1}, \theta) \right) p(X_1, \ldots, X_p|\theta)$$

$$\log \mathcal{L}(\alpha|X) = \underbrace{\sum_{t=p+1}^{T} \log p(X_t|X_{t-1}, \ldots, X_{p+1}, \theta)}_{\text{conditional ll}} + \underbrace{\log p(X_1, \ldots, X_p|\theta)}_{\text{marginal ll}}$$

Estimating ARMA process

As a composition of $AR(p)$ and $MA(q)$, $ARMA(p, q)$ should be estimated with maximum likelihood method. Optimization of conditional likelihood is what happens under the hood of most practical methods.

Estimating parameters

- α_0, α, β
- d, D
- q, Q
- p, P

Estimating differencing order

- d and D are chosen such that the series is stationary.
- If there is seasonality, it is recommended to start with seasonal differencing, it may already make the series stationary, as we've seen last time.

- The less differencing, the better, because
  - More data
  - Less variance due to back transformation

Estimating MA and AR order

- We absolutely can't estimate p and q with maximum likelihood, the higher the order, the higher $\mathcal{L}$
- We can select the order by looking at ACF and PACF
- We can select the order automatically with information criteria

PACF

Partial autocorrelation function is defined as follows:

$$\pi(h,h) = \begin{cases} \rho(X_{t+1}, X_t), & h = 1, \\ \rho(X_{t+h} - X_{t+h}^{h-1}, X_t - X_t^{h-1}), & h > 1 \end{cases}$$

where X_t^{h-1} is a result of regressing X_t on its $h-1$ previous values.

In words: When fitting an $AR(p)$ model, the last coefficient α_p will be denoted by π_p and measures the excess correlation at lag p which is not accounted for by an $AR(p-1)$ model. It is called the p-th partial autocorrelation coefficient and, when plotted against p, gives the partial ACF.

Estimating MA and AR order with ACF and PACF

- We know that for $MA(q)$ process, the ACF becomes zero after q lags.
- We also know that for $AR(p)$ process, the ACF is a mixture of exponential decay and sinusoid.
- It can be shown that the PACF has reverse properties to those above.

So by looking at ACF we define the order q and by looking at PACF we define the order p.

Also the seaonal order can be defined, the lags will be significant at $P \cdot S$ and $Q \cdot S$.

Estimating MA and AR order with information criteria

Information criteria:

- Akaike

$$AIC = -2\mathcal{L} + 2k$$

- Corrected Akaike (for small samples)

$$AICc = -2\mathcal{L} + \frac{2k^2}{T - k - 1}$$

- Bayesian

$$BIC = -2\mathcal{L} + k(\log T - 2)$$

here $k = P + Q + p + q + 1$ is the number of parameters in the model.s

Residuals

When a model has been fitted to a time series, it is advisable to check that the model really does provide an adequate description of the data. As with most statistical models, this is usually done by looking at the residuals:

$$\text{residual} = \text{observation} - \text{fitted value}$$

For a univariate time-series model, the fitted value is the one-step-ahead forecast so that the residual is the one-step-ahead forecast error.

The properties of residuals, e.g. lack of structure, quanitfy the goodness of the model.

Necessary properties of residuals

- Unbiasedness; to be checked with standard statistical tests for the mean, e.g. Student's t-test
- Stationarity; to be checked with visual analysis, KPSS test
- Uncorrelatedness; to be checked with correlogram, Ljung-Box-Pierce Q-criterion

Desired properties of residuals

- Normality; to be checked with QQ-plot, Shapiro test
- Homoscedacity; to be checked with visual analysis, Breusch–Pagan test

Webinar

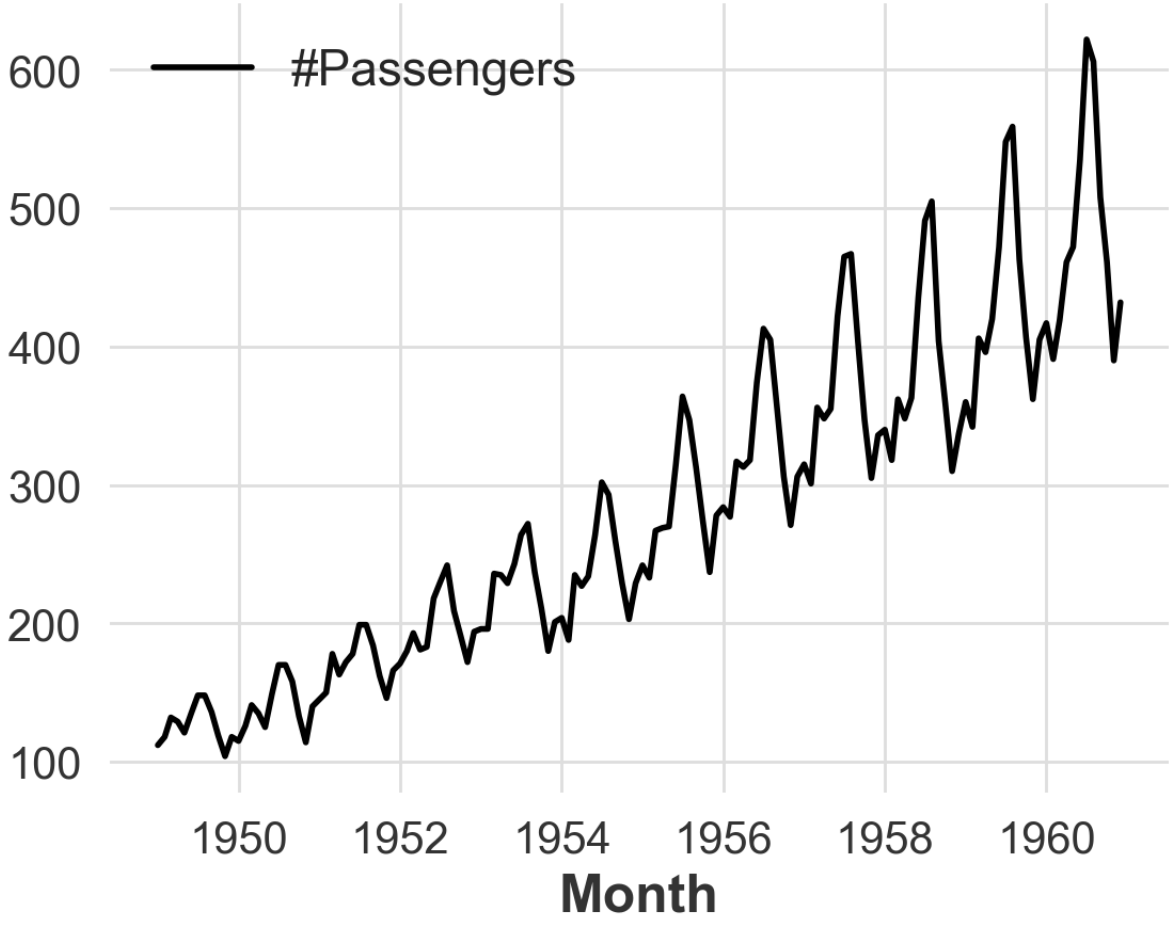
Testing stationarity

There are two main tools to test if series is stationary or not. There are more, but these two are sufficient.

- KPSS (Kwiatkowski-Philips-Schmidt-Shin) test. Null hypothesis is that series is stationary, alternative is that there is trend.
- Dickey–Fuller test. Null hypothesis is that series is non-stationary, alternative is that series is stationary.

In [7]:
```
from darts import TimeSeries
import darts.datasets as ds
import scipy.stats as sts
import statsmodels.api as sm
import statsmodels.tsa.api as tsa
from darts.utils.statistics import plot_acf, plot_pacf
```

In [5]:
```
air_pax = ds.AirPassengersDataset().load()
air_pax.plot();
```



In [8]:
```
_, p_value, _, _ = tsa.stattools.kpss(air_pax.values())
print("rejected" if p_value < 0.05 else "not rejected")
```

```
rejected
```

```
/var/folders/33/j0cl7y453td68qb96j7bqcj4cf41kc/T/ipykernel_823/3112241267.
py:1: InterpolationWarning: The test statistic is outside of the range of
p-values available in the
look-up table. The actual p-value is smaller than the p-value returned.

  _, p_value, _, _ = tsa.stattools.kpss(air_pax.values())
```

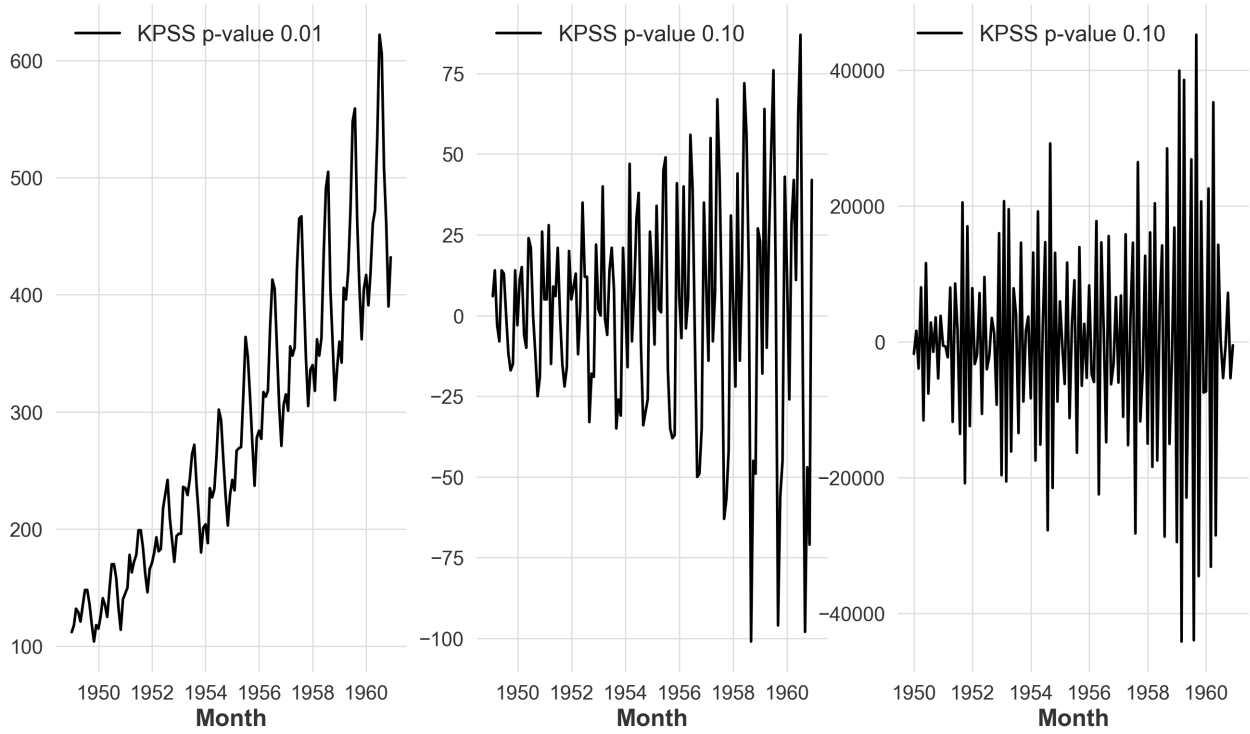
In [10]:
```
_, p_value, _, _, _, _ = tsa.stattools.adfuller(air_pax.values())
print("rejected" if p_value < 0.05 else "not rejected")
```

```
not rejected
```

Differencing

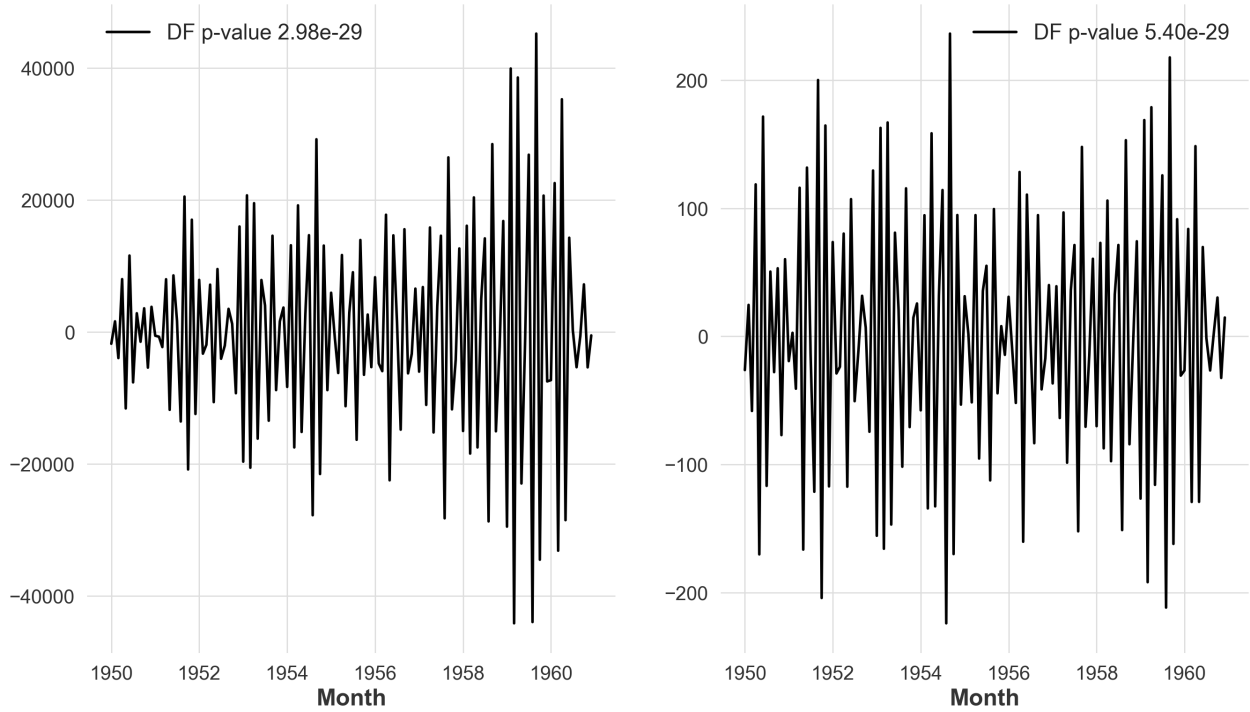
In [12]:
```
fig, ax = plt.subplots(1,3,figsize=(16,9))
air_pax.plot(ax=ax[0], label=f"KPSS p-value {tsa.stattools.kpss(air_pax.v
air_pax.diff().plot(ax=ax[1], label=f"KPSS p-value {tsa.stattools.kpss(ai
air_pax.diff(12).plot(ax=ax[2], label=f"KPSS p-value {tsa.stattools.kpss(
```

```
/var/folders/33/j0cl7y453td68qb96j7bqcj4cf41kc/T/ipykernel_823/540728647.p
y:2: InterpolationWarning: The test statistic is outside of the range of p
-values available in the
look-up table. The actual p-value is smaller than the p-value returned.

  air_pax.plot(ax=ax[0], label=f"KPSS p-value {tsa.stattools.kpss(air_pax.
values())[1]:.2f}");
/var/folders/33/j0cl7y453td68qb96j7bqcj4cf41kc/T/ipykernel_823/540728647.p
y:3: InterpolationWarning: The test statistic is outside of the range of p
-values available in the
look-up table. The actual p-value is greater than the p-value returned.

  air_pax.diff().plot(ax=ax[1], label=f"KPSS p-value {tsa.stattools.kpss(a
ir_pax.diff().values())[1]:.2f}");
/var/folders/33/j0cl7y453td68qb96j7bqcj4cf41kc/T/ipykernel_823/540728647.p
y:4: InterpolationWarning: The test statistic is outside of the range of p
-values available in the
look-up table. The actual p-value is greater than the p-value returned.

  air_pax.diff(12).plot(ax=ax[2], label=f"KPSS p-value {tsa.stattools.kps
s(air_pax.diff(12).values())[1]:.2f}");
```



In [13]:
```
air_pax_log = air_pax.map(lambda ts, x: sts.boxcox(x, lmbda=0.1))
```

In [15]:
```
fig, ax = plt.subplots(1,2,figsize=(16,9))
air_pax.diff(12).plot(ax=ax[0], label=f"DF p-value {tsa.stattools.adfulle
air_pax_log.diff(12).plot(ax=ax[1], label=f"DF p-value {tsa.stattools.adf
```



Estimating p and q

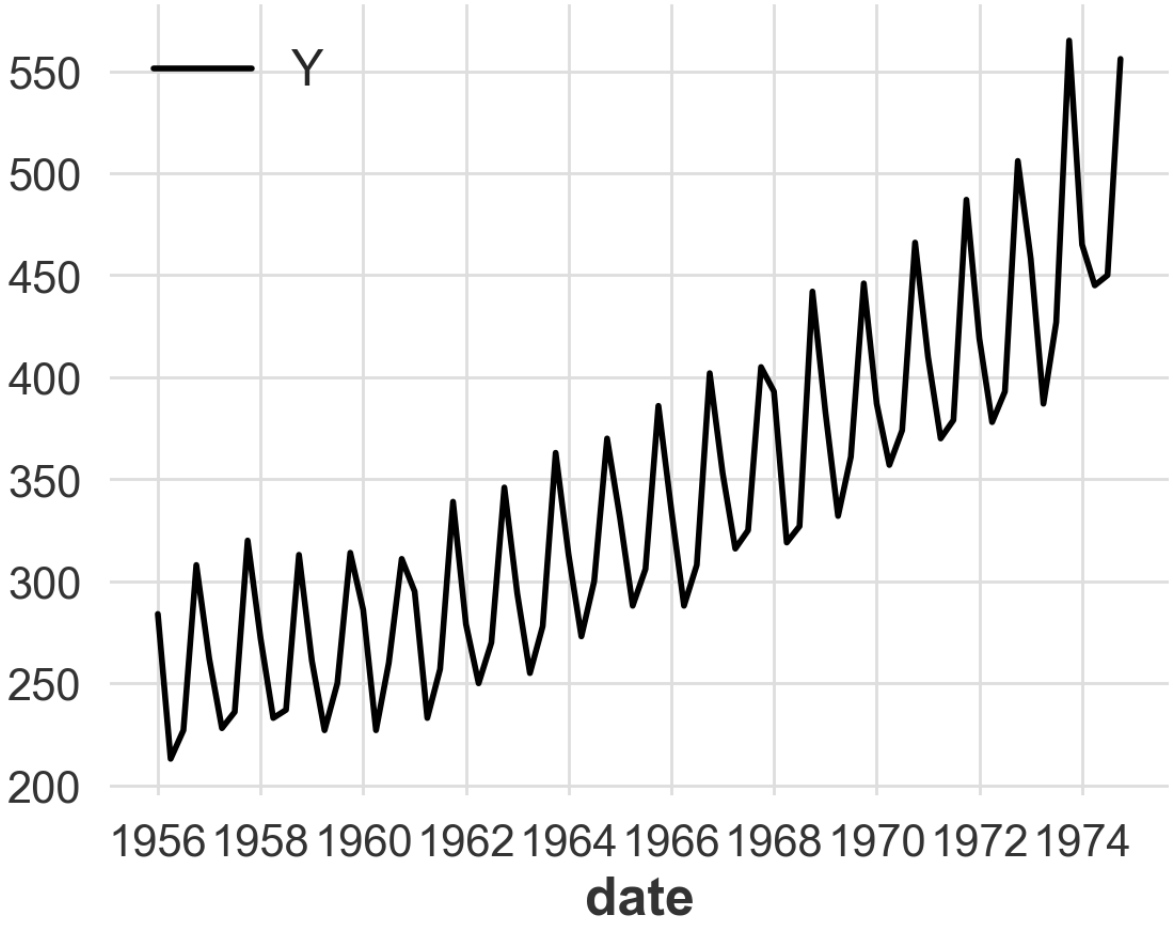
In [77]:
```
beer_dataset = ds.AusBeerDataset().load()
```

In [81]:
```
import pandas as pd
```

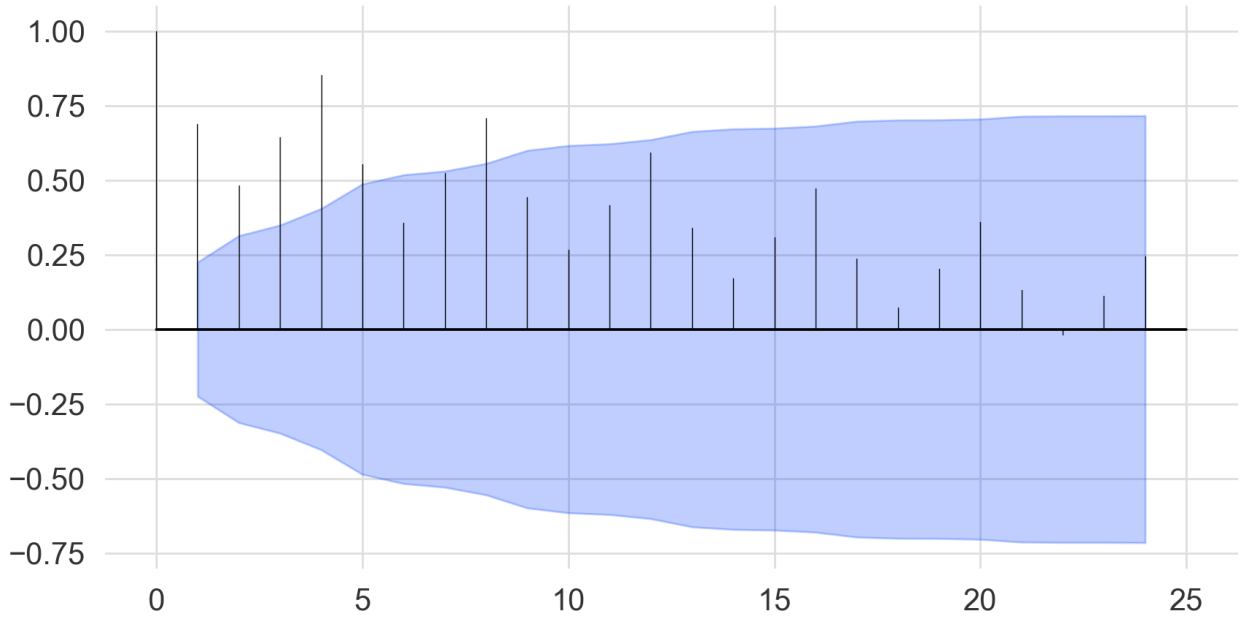
In [82]:
```
beer_dataset, _ = beer_dataset.split_before(pd.Timestamp("1975-01-01"))
```

In [89]:
```
beer_dataset.plot()
```

Out[89]:
```
<Axes: xlabel='date'>
```

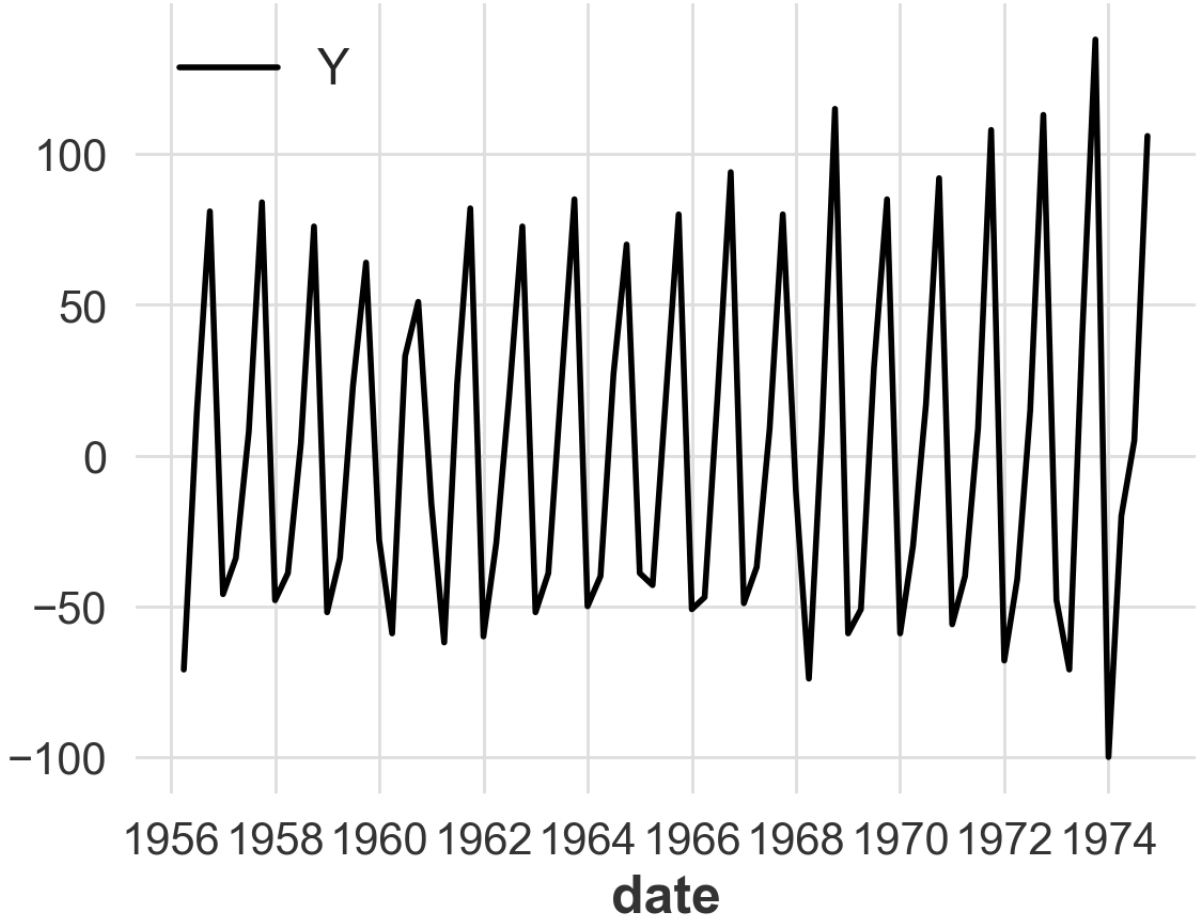


In [90]:
```
plot_acf(beer_dataset)
```

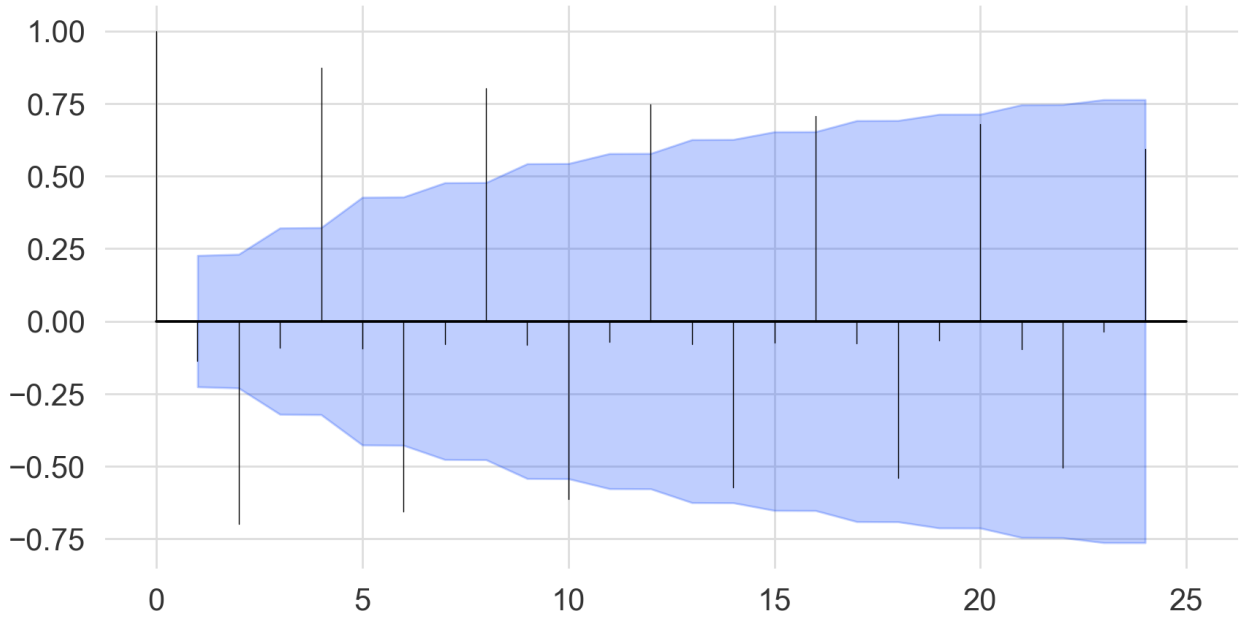


In [87]:
```
beer_dataset_diff = beer_dataset.diff()
beer_dataset_diff.plot()
```

Out[87]: <Axes: xlabel='date'>



In [88]:
```
plot_acf(beer_dataset_diff)
```

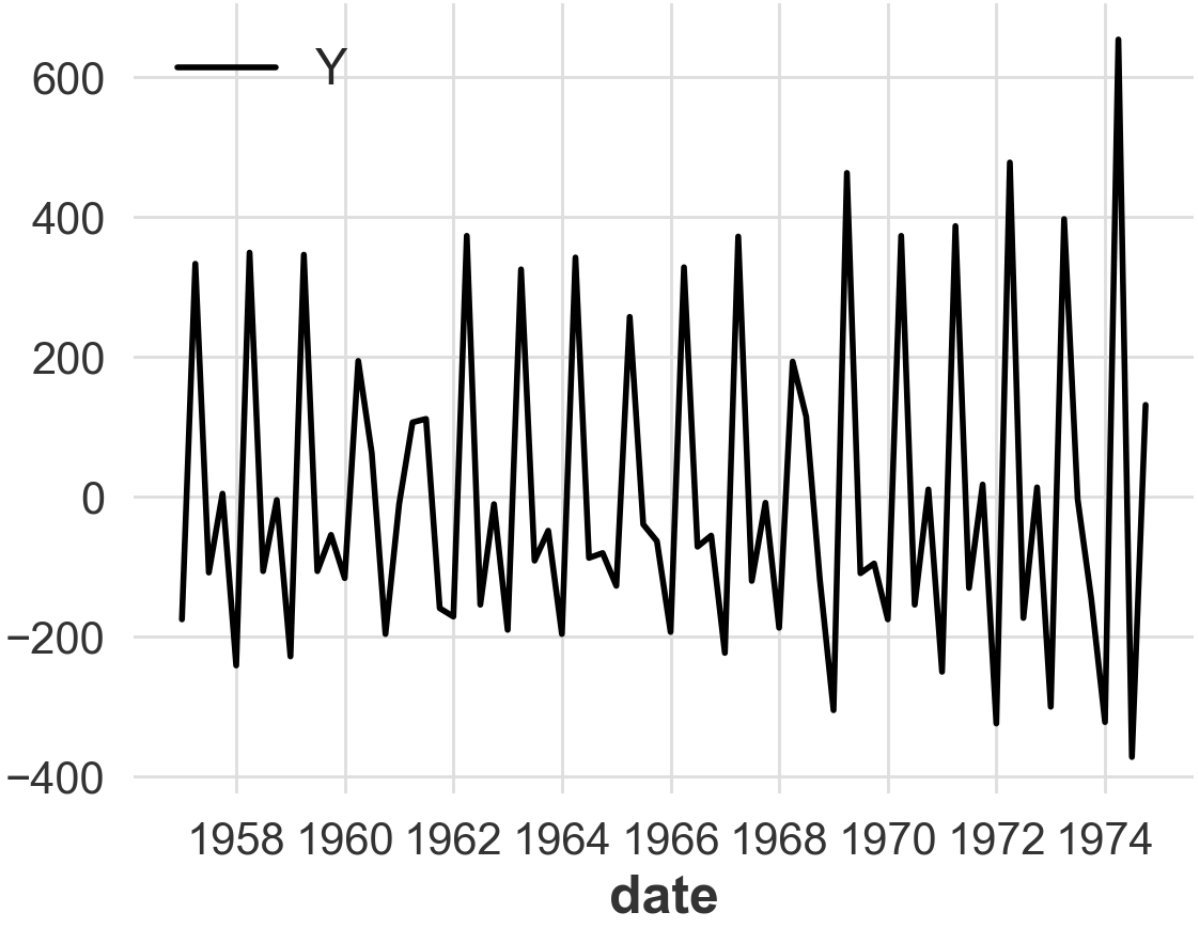


In [ ]:
```
# s = 4
```

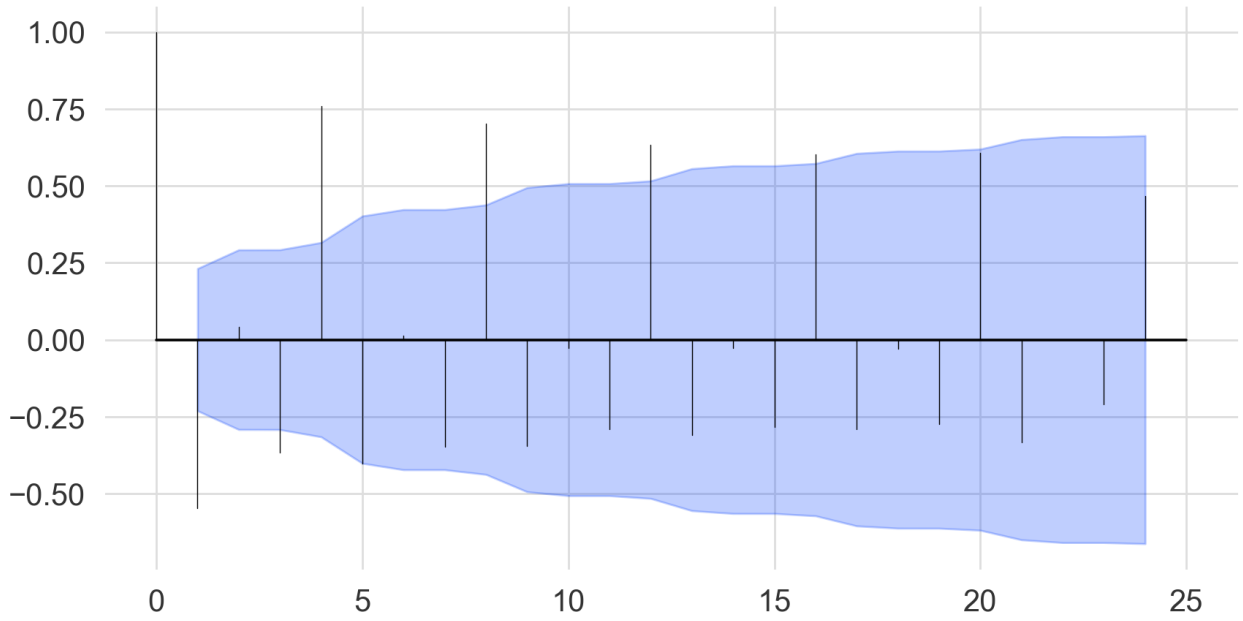
In [91]:
```
beer_dataset_sdiff = beer_dataset.diff(4)
```

In [92]:
```
beer_dataset_sdiff.plot()
```

Out[92]:
```
<Axes: xlabel='date'>
```

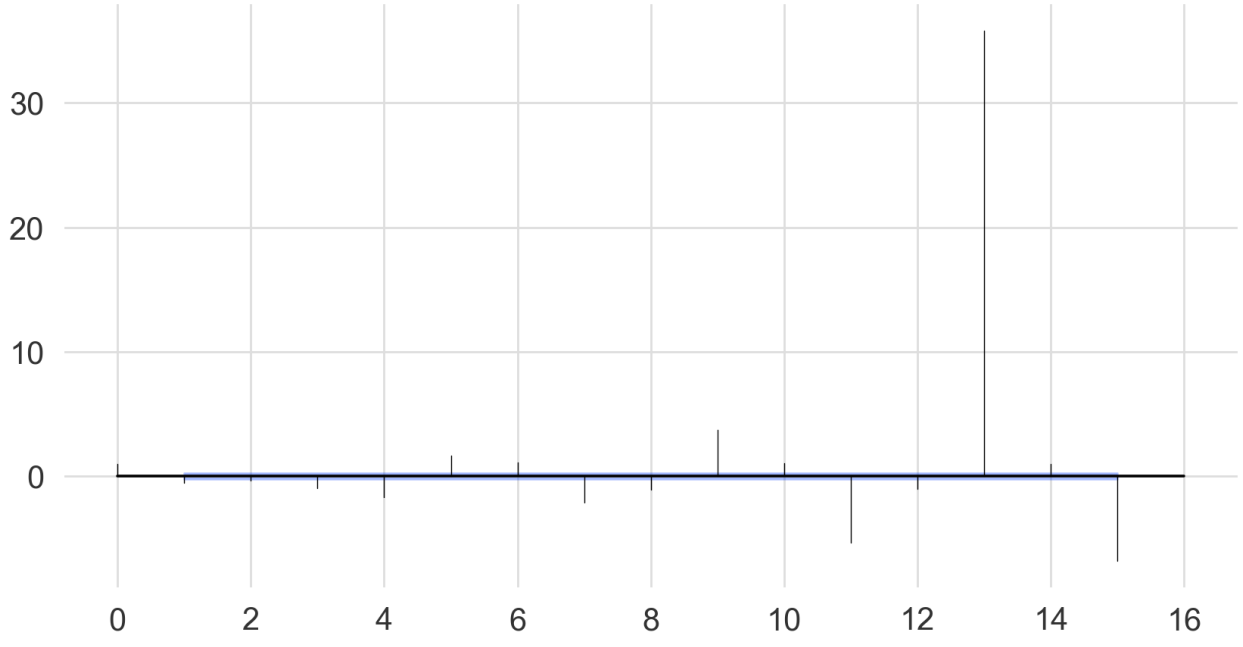


In [93]:
```
plot_acf(beer_dataset_sdiff)
```



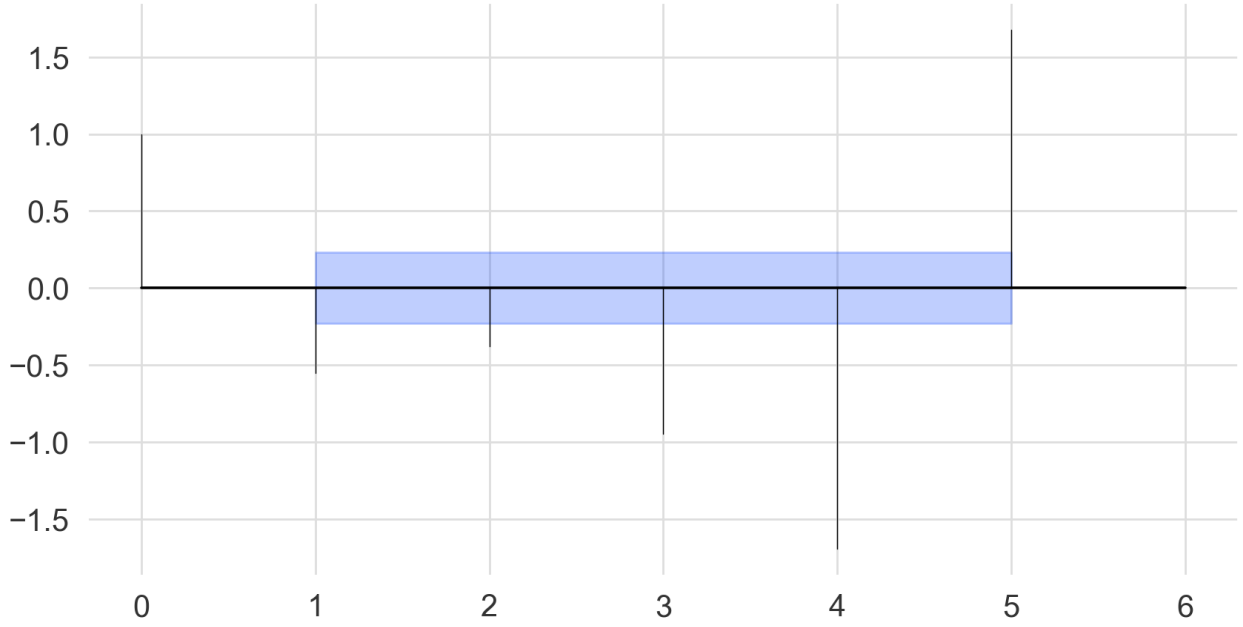
In [ ]:
```
# Q = 1
# q = 1
```

In [101…
```
plot_pacf(beer_dataset_sdiff, max_lag=15)
```



In [ ]:
```
# P = 0
```

In [98]:
```
plot_pacf(beer_dataset_sdiff, max_lag=5)
```



In [ ]:
```
# p = 1
```

In [ ]:
```
# ARIMA(1, 0, 1)x(0, 1, 1, 4)
```

In [102…
```
model = ARIMA(p=1, d=0, q=1, seasonal_order=(0, 1, 1, 4))
model.fit(beer_dataset)
```

```
/Users/nstulov/miniconda3/envs/msai/lib/python3.12/site-packages/statsmode
ls/tsa/statespace/sarimax.py:966: UserWarning: Non-stationary starting aut
oregressive parameters found. Using zeros as starting parameters.
  warn('Non-stationary starting autoregressive parameters'
/Users/nstulov/miniconda3/envs/msai/lib/python3.12/site-packages/statsmode
ls/tsa/statespace/sarimax.py:978: UserWarning: Non-invertible starting MA
parameters found. Using zeros as starting parameters.
  warn('Non-invertible starting MA parameters found.'
```

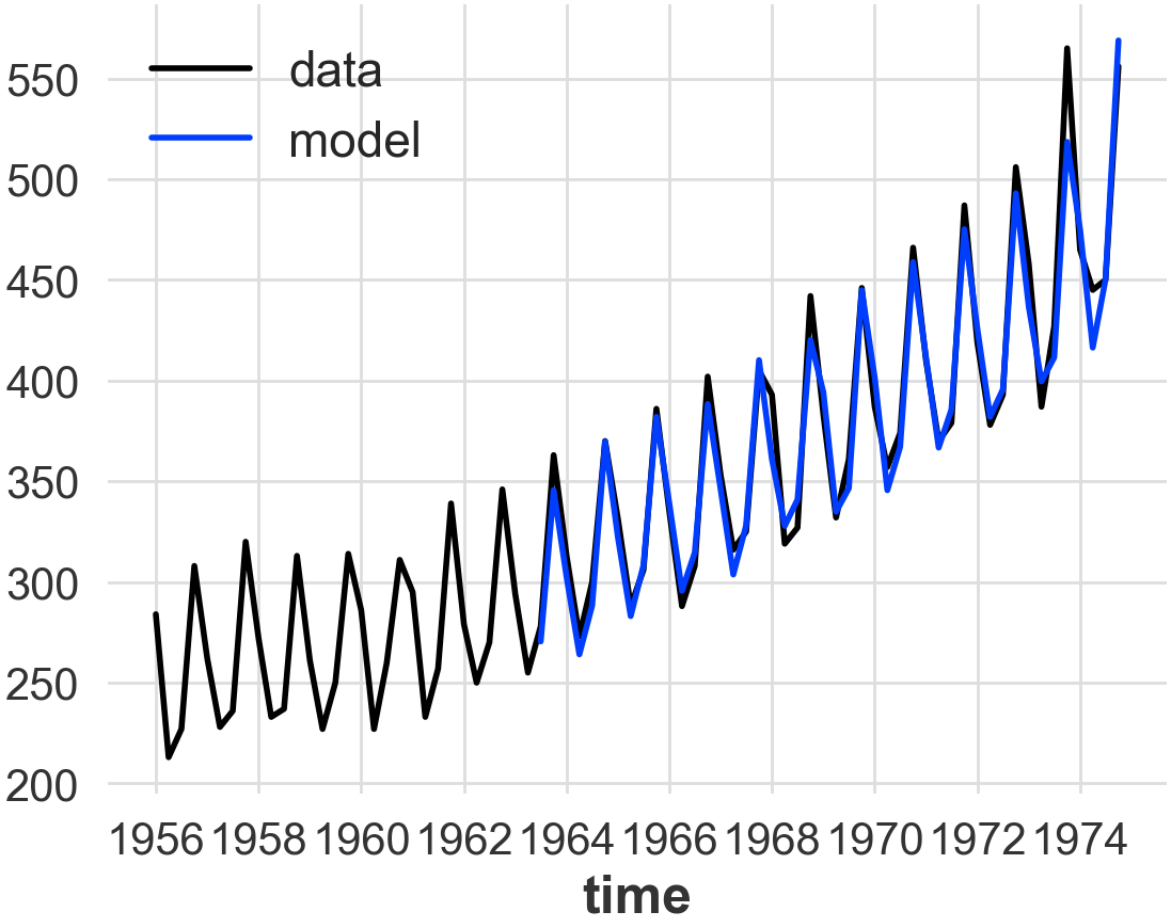
Out[102…
```
ARIMA(p=1, d=0, q=1, seasonal_order=(0, 1, 1, 4), trend=None, random_sta
te=None, add_encoders=None)
```

In [103…
```
fcast = model.historical_forecasts(
    beer_dataset, forecast_horizon=1, start_format="position", show_warni
)
```

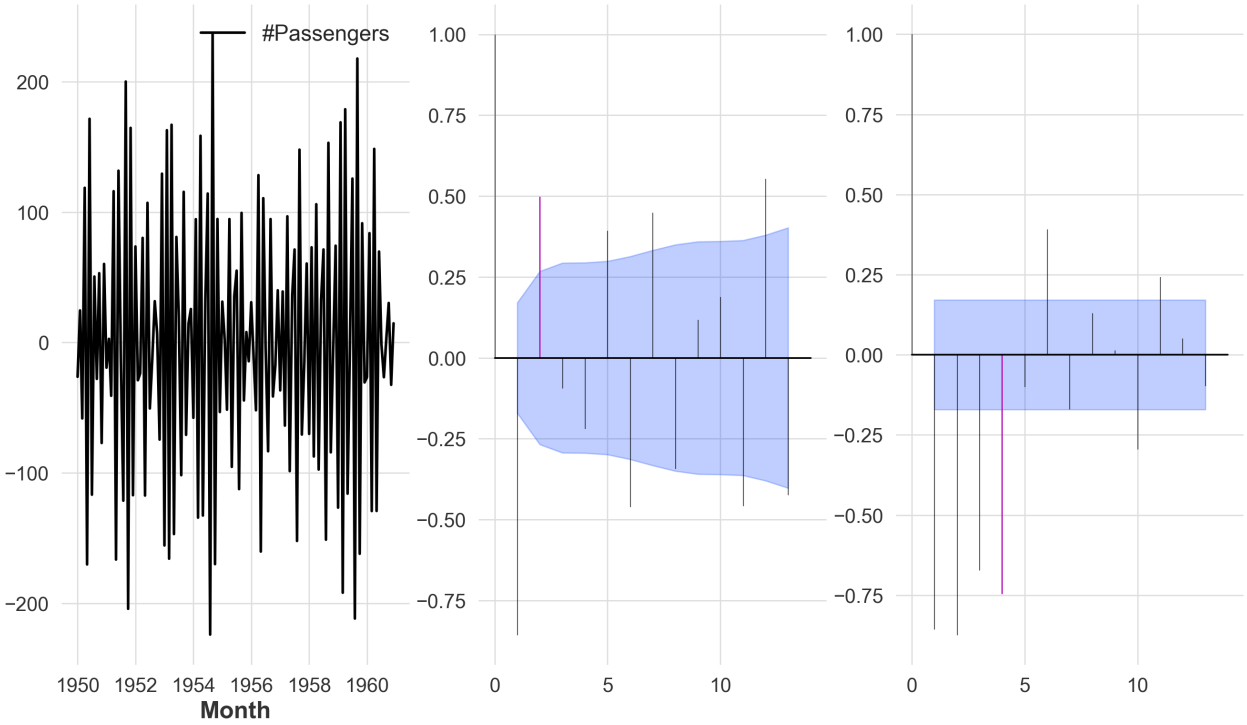
```
/Users/nstulov/miniconda3/envs/msai/lib/python3.12/site-packages/statsmode
ls/tsa/statespace/sarimax.py:966: UserWarning: Non-stationary starting aut
oregressive parameters found. Using zeros as starting parameters.
  warn('Non-stationary starting autoregressive parameters'
/Users/nstulov/miniconda3/envs/msai/lib/python3.12/site-packages/statsmode
ls/tsa/statespace/sarimax.py:978: UserWarning: Non-invertible starting MA
parameters found. Using zeros as starting parameters.
  warn('Non-invertible starting MA parameters found.'
```

In [104…
```
fig, ax = plt.subplots()
beer_dataset.plot(ax=ax, label="data")
fcast.plot(ax=ax, label="model")
```

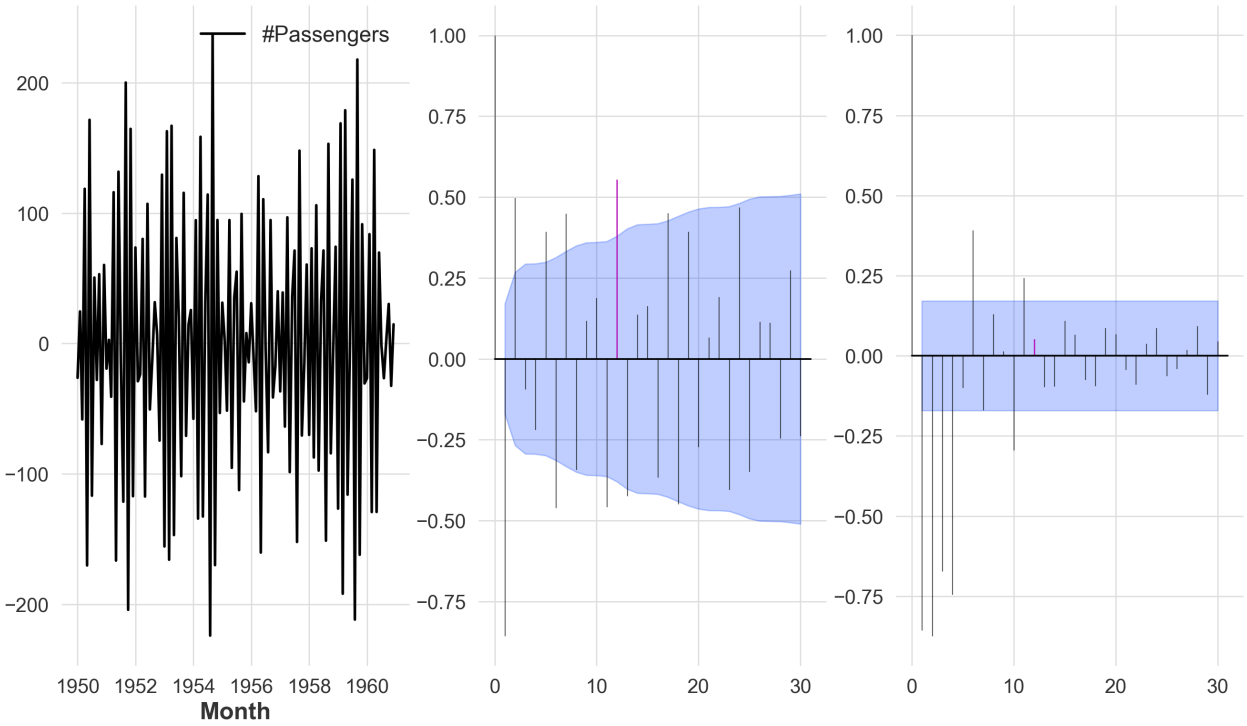
```
ax.legend();
```



In [63]:
```
fig, ax = plt.subplots(1,3,figsize=(16,9))
air_pax_log_diff.plot(ax=ax[0])
plot_acf(air_pax_log_diff, axis=ax[1], max_lag=13, m=2) # q = 2
plot_pacf(air_pax_log_diff, axis=ax[2], max_lag=13, method="ywm", m=4) #
# SARIMA(4, 0, 2)x(0, 1, 0, S=12)
```



In [69]:
```
fig, ax = plt.subplots(1,3,figsize=(16,9))
air_pax_log_diff.plot(ax=ax[0])
plot_acf(air_pax_log_diff, axis=ax[1], max_lag=30, m=12) # Q = 1
plot_pacf(air_pax_log_diff, axis=ax[2], max_lag=30, method="ywm", m=12) #
# SARIMA(4, 0, 2)x(0, 1, 1, S=12)
```



In [70]:
```
from darts.models import ARIMA
```

In [106…
```
model = ARIMA(p=1, d=0, q=2, seasonal_order=(0, 1, 1, 12))
model.fit(air_pax_log)
fcast = model.historical_forecasts(
```

```
    air_pax_log, forecast_horizon=1, start_format="position", show_warnin
)
```

```
/Users/nstulov/miniconda3/envs/msai/lib/python3.12/site-packages/statsmode
ls/base/model.py:607: ConvergenceWarning: Maximum Likelihood optimization
failed to converge. Check mle_retvals
  warnings.warn("Maximum Likelihood optimization failed to "
/Users/nstulov/miniconda3/envs/msai/lib/python3.12/site-packages/statsmode
ls/tsa/statespace/sarimax.py:866: UserWarning: Too few observations to est
imate starting parameters for seasonal ARMA. All parameters except for var
iances will be set to zeros.
  warn('Too few observations to estimate starting parameters%s.'
/Users/nstulov/miniconda3/envs/msai/lib/python3.12/site-packages/statsmode
ls/tsa/statespace/sarimax.py:966: UserWarning: Non-stationary starting aut
oregressive parameters found. Using zeros as starting parameters.
  warn('Non-stationary starting autoregressive parameters'
/Users/nstulov/miniconda3/envs/msai/lib/python3.12/site-packages/statsmode
ls/base/model.py:607: ConvergenceWarning: Maximum Likelihood optimization
failed to converge. Check mle_retvals
  warnings.warn("Maximum Likelihood optimization failed to "
/Users/nstulov/miniconda3/envs/msai/lib/python3.12/site-packages/statsmode
ls/base/model.py:607: ConvergenceWarning: Maximum Likelihood optimization
failed to converge. Check mle_retvals
  warnings.warn("Maximum Likelihood optimization failed to "
/Users/nstulov/miniconda3/envs/msai/lib/python3.12/site-packages/statsmode
ls/base/model.py:607: ConvergenceWarning: Maximum Likelihood optimization
failed to converge. Check mle_retvals
  warnings.warn("Maximum Likelihood optimization failed to "
/Users/nstulov/miniconda3/envs/msai/lib/python3.12/site-packages/statsmode
ls/base/model.py:607: ConvergenceWarning: Maximum Likelihood optimization
failed to converge. Check mle_retvals
  warnings.warn("Maximum Likelihood optimization failed to "
/Users/nstulov/miniconda3/envs/msai/lib/python3.12/site-packages/statsmode
ls/base/model.py:607: ConvergenceWarning: Maximum Likelihood optimization
failed to converge. Check mle_retvals
  warnings.warn("Maximum Likelihood optimization failed to "
/Users/nstulov/miniconda3/envs/msai/lib/python3.12/site-packages/statsmode
ls/base/model.py:607: ConvergenceWarning: Maximum Likelihood optimization
failed to converge. Check mle_retvals
  warnings.warn("Maximum Likelihood optimization failed to "
/Users/nstulov/miniconda3/envs/msai/lib/python3.12/site-packages/statsmode
ls/base/model.py:607: ConvergenceWarning: Maximum Likelihood optimization
failed to converge. Check mle_retvals
  warnings.warn("Maximum Likelihood optimization failed to "
/Users/nstulov/miniconda3/envs/msai/lib/python3.12/site-packages/statsmode
ls/base/model.py:607: ConvergenceWarning: Maximum Likelihood optimization
failed to converge. Check mle_retvals
  warnings.warn("Maximum Likelihood optimization failed to "
/Users/nstulov/miniconda3/envs/msai/lib/python3.12/site-packages/statsmode
ls/base/model.py:607: ConvergenceWarning: Maximum Likelihood optimization
failed to converge. Check mle_retvals
  warnings.warn("Maximum Likelihood optimization failed to "
/Users/nstulov/miniconda3/envs/msai/lib/python3.12/site-packages/statsmode
ls/tsa/statespace/sarimax.py:1009: UserWarning: Non-invertible starting se
```

```
asonal moving average Using zeros as starting parameters.
  warn('Non-invertible starting seasonal moving average'
/Users/nstulov/miniconda3/envs/msai/lib/python3.12/site-packages/statsmode
ls/base/model.py:607: ConvergenceWarning: Maximum Likelihood optimization
failed to converge. Check mle_retvals
  warnings.warn("Maximum Likelihood optimization failed to "
/Users/nstulov/miniconda3/envs/msai/lib/python3.12/site-packages/statsmode
ls/base/model.py:607: ConvergenceWarning: Maximum Likelihood optimization
failed to converge. Check mle_retvals
  warnings.warn("Maximum Likelihood optimization failed to "
/Users/nstulov/miniconda3/envs/msai/lib/python3.12/site-packages/statsmode
ls/base/model.py:607: ConvergenceWarning: Maximum Likelihood optimization
failed to converge. Check mle_retvals
  warnings.warn("Maximum Likelihood optimization failed to "
/Users/nstulov/miniconda3/envs/msai/lib/python3.12/site-packages/statsmode
ls/base/model.py:607: ConvergenceWarning: Maximum Likelihood optimization
failed to converge. Check mle_retvals
  warnings.warn("Maximum Likelihood optimization failed to "
/Users/nstulov/miniconda3/envs/msai/lib/python3.12/site-packages/statsmode
ls/base/model.py:607: ConvergenceWarning: Maximum Likelihood optimization
failed to converge. Check mle_retvals
  warnings.warn("Maximum Likelihood optimization failed to "
/Users/nstulov/miniconda3/envs/msai/lib/python3.12/site-packages/statsmode
ls/base/model.py:607: ConvergenceWarning: Maximum Likelihood optimization
failed to converge. Check mle_retvals
  warnings.warn("Maximum Likelihood optimization failed to "
/Users/nstulov/miniconda3/envs/msai/lib/python3.12/site-packages/statsmode
ls/base/model.py:607: ConvergenceWarning: Maximum Likelihood optimization
failed to converge. Check mle_retvals
  warnings.warn("Maximum Likelihood optimization failed to "

--------------------------------------------------------------------------
-
KeyboardInterrupt                         Traceback (most recent call last
)
Cell In[106], line 3
      1 model = ARIMA(p=1, d=0, q=2, seasonal_order=(0, 1, 1, 12))
      2 model.fit(air_pax_log)
----> 3 fcast = model.historical_forecasts(
      4     air_pax_log, forecast_horizon=1, start_format="position", show
_warnings=False
      5 )

File ~/.local/lib/python3.12/site-packages/darts/utils/utils.py:178, in _w
ith_sanity_checks.<locals>.decorator.<locals>.sanitized_method(self, *args
, **kwargs)
    175     only_args.pop("self")
    177     getattr(self, sanity_check_method)(*only_args.values(), **only
_kwargs)
--> 178 return method_to_sanitize(self, *only_args.values(), **only_kwargs
```

```
)

File ~/.local/lib/python3.12/site-packages/darts/models/forecasting/forecasting_model.py:1034, in ForecastingModel.historical_forecasts(self, series, past_covariates, future_covariates, num_samples, train_length, start, start_format, forecast_horizon, stride, retrain, overlap_end, last_points_only, verbose, show_warnings, predict_likelihood_parameters, enable_optimization, fit_kwargs, predict_kwargs)
   1025 if retrain_func(
   1026     counter=_counter_train,
   1027     pred_time=pred_time,
   (...)
   1031 ):
   1032     # avoid fitting the same model multiple times
   1033     model = model.untrained_model()
-> 1034     model._fit_wrapper(
   1035         series=train_series,
   1036         past_covariates=past_covariates_,
   1037         future_covariates=future_covariates_,
   1038         **fit_kwargs,
   1039     )
   1040 else:
   1041     # untrained model was not trained on the first trainable timestamp
   1042     if not _counter_train and not model._fit_called:

File ~/.local/lib/python3.12/site-packages/darts/models/forecasting/forecasting_model.py:378, in ForecastingModel._fit_wrapper(self, series, past_covariates, future_covariates, **kwargs)
    373     elif covs is not None:
    374         raise_log(
    375             ValueError(f"Model cannot be fit/trained with `{covs_name}`."),
    376             logger,
    377         )
--> 378 self.fit(series=series, **add_kwargs, **kwargs)

File ~/.local/lib/python3.12/site-packages/darts/models/forecasting/forecasting_model.py:2477, in FutureCovariatesLocalForecastingModel.fit(self, series, future_covariates)
   2469     _, future_covariates = self.generate_fit_encodings(
   2470         series=series,
   2471         past_covariates=None,
   2472         future_covariates=future_covariates,
   2473     )
   2475 super().fit(series)
-> 2477 return self._fit(series, future_covariates=future_covariates)

File ~/.local/lib/python3.12/site-packages/darts/models/forecasting/arima.py:167, in ARIMA._fit(self, series, future_covariates)
    158 self.training_historic_future_covariates = future_covariates
    160 m = staARIMA(
    161     series.values(copy=False),
```

```
    162     exog=future_covariates.values(copy=False) if future_covariates
else None,
   (...)
    165     trend=self.trend,
    166 )
--> 167 self.model = m.fit()
    169 return self

File ~/miniconda3/envs/msai/lib/python3.12/site-packages/statsmodels/tsa/a
rima/model.py:395, in ARIMA.fit(self, start_params, transformed, includes_
fixed, method, method_kwargs, gls, gls_kwargs, cov_type, cov_kwds, return_
params, low_memory)
    392 else:
    393     method_kwargs.setdefault('disp', 0)
--> 395     res = super().fit(
    396         return_params=return_params, low_memory=low_memory,
    397         cov_type=cov_type, cov_kwds=cov_kwds, **method_kwargs)
    398     if not return_params:
    399         res.fit_details = res.mlefit

File ~/miniconda3/envs/msai/lib/python3.12/site-packages/statsmodels/tsa/s
tatespace/mlemodel.py:704, in MLEModel.fit(self, start_params, transformed
, includes_fixed, cov_type, cov_kwds, method, maxiter, full_output, disp,
callback, return_params, optim_score, optim_complex_step, optim_hessian, f
lags, low_memory, **kwargs)
    702         flags['hessian_method'] = optim_hessian
    703     fargs = (flags,)
--> 704     mlefit = super(MLEModel, self).fit(start_params, method=method
,
    705                                        fargs=fargs,
    706                                        maxiter=maxiter,
    707                                        full_output=full_output,
    708                                        disp=disp, callback=callbac
k,
    709                                        skip_hessian=True, **kwargs
)
    711 # Just return the fitted parameters if requested
    712 if return_params:

File ~/miniconda3/envs/msai/lib/python3.12/site-packages/statsmodels/base/
model.py:566, in LikelihoodModel.fit(self, start_params, method, maxiter,
full_output, disp, fargs, callback, retall, skip_hessian, **kwargs)
    563     del kwargs["use_t"]
    565 optimizer = Optimizer()
--> 566 xopt, retvals, optim_settings = optimizer._fit(f, score, start_par
ams,
    567                                                 fargs, kwargs,
    568                                                 hessian=hess,
    569                                                 method=method,
    570                                                 disp=disp,
    571                                                 maxiter=maxiter,
    572                                                 callback=callback,
    573                                                 retall=retall,
```

```
    574                                                 full_output=full_ou
tput)
    575 # Restore cov_type, cov_kwds and use_t
    576 optim_settings.update(kwds)

File ~/miniconda3/envs/msai/lib/python3.12/site-packages/statsmodels/base/
optimizer.py:242, in Optimizer._fit(self, objective, gradient, start_param
s, fargs, kwargs, hessian, method, maxiter, full_output, disp, callback, r
etall)
    239     fit_funcs.update(extra_fit_funcs)
    241 func = fit_funcs[method]
--> 242 xopt, retvals = func(objective, gradient, start_params, fargs, kwa
rgs,
    243                     disp=disp, maxiter=maxiter, callback=callback
,
    244                     retall=retall, full_output=full_output,
    245                     hess=hessian)
    247 optim_settings = {'optimizer': method, 'start_params': start_param
s,
    248                   'maxiter': maxiter, 'full_output': full_output,
    249                   'disp': disp, 'fargs': fargs, 'callback': callba
ck,
    250                   'retall': retall, "extra_fit_funcs": extra_fit_f
uncs}
    251 optim_settings.update(kwargs)

File ~/miniconda3/envs/msai/lib/python3.12/site-packages/statsmodels/base/
optimizer.py:659, in _fit_lbfgs(f, score, start_params, fargs, kwargs, dis
p, maxiter, callback, retall, full_output, hess)
    656 elif approx_grad:
    657     func = f
--> 659 retvals = optimize.fmin_l_bfgs_b(func, start_params, maxiter=maxit
er,
    660                                   callback=callback, args=fargs,
    661                                   bounds=bounds, disp=disp,
    662                                   **extra_kwargs)
    664 if full_output:
    665     xopt, fopt, d = retvals

File ~/miniconda3/envs/msai/lib/python3.12/site-packages/scipy/optimize/_l
bfgsb_py.py:199, in fmin_l_bfgs_b(func, x0, fprime, args, approx_grad, bou
nds, m, factr, pgtol, epsilon, iprint, maxfun, maxiter, disp, callback, ma
xls)
    187 callback = _wrap_callback(callback)
    188 opts = {'disp': disp,
    189         'iprint': iprint,
    190         'maxcor': m,
   (...)
    196         'callback': callback,
    197         'maxls': maxls}
--> 199 res = _minimize_lbfgsb(fun, x0, args=args, jac=jac, bounds=bounds,
    200                         **opts)
    201 d = {'grad': res['jac'],
```

```
    202      'task': res['message'],
    203      'funcalls': res['nfev'],
    204      'nit': res['nit'],
    205      'warnflag': res['status']}
    206 f = res['fun']

File ~/miniconda3/envs/msai/lib/python3.12/site-packages/scipy/optimize/_l
bfgsb_py.py:365, in _minimize_lbfgsb(fun, x0, args, jac, bounds, disp, max
cor, ftol, gtol, eps, maxfun, maxiter, iprint, callback, maxls, finite_dif
f_rel_step, **unknown_options)
    359 task_str = task.tobytes()
    360 if task_str.startswith(b'FG'):
    361     # The minimization routine wants f and g at the current x.
    362     # Note that interruptions due to maxfun are postponed
    363     # until the completion of the current minimization iteration.
    364     # Overwrite f and g:
--> 365     f, g = func_and_grad(x)
    366 elif task_str.startswith(b'NEW_X'):
    367     # new iteration
    368     n_iterations += 1

File ~/miniconda3/envs/msai/lib/python3.12/site-packages/scipy/optimize/_d
ifferentiable_functions.py:286, in ScalarFunction.fun_and_grad(self, x)
    284     self._update_x_impl(x)
    285 self._update_fun()
--> 286 self._update_grad()
    287 return self.f, self.g

File ~/miniconda3/envs/msai/lib/python3.12/site-packages/scipy/optimize/_d
ifferentiable_functions.py:256, in ScalarFunction._update_grad(self)
    254 def _update_grad(self):
    255     if not self.g_updated:
--> 256         self._update_grad_impl()
    257         self.g_updated = True

File ~/miniconda3/envs/msai/lib/python3.12/site-packages/scipy/optimize/_d
ifferentiable_functions.py:173, in ScalarFunction.__init__.<locals>.update
_grad()
    171 self._update_fun()
    172 self.ngev += 1
--> 173 self.g = approx_derivative(fun_wrapped, self.x, f0=self.f,
    174                            **finite_diff_options)

File ~/miniconda3/envs/msai/lib/python3.12/site-packages/scipy/optimize/_n
umdiff.py:505, in approx_derivative(fun, x0, method, rel_step, abs_step, f
0, bounds, sparsity, as_linear_operator, args, kwargs)
    502     use_one_sided = False
    504 if sparsity is None:
--> 505     return _dense_difference(fun_wrapped, x0, f0, h,
    506                              use_one_sided, method)
    507 else:
    508     if not issparse(sparsity) and len(sparsity) == 2:

File ~/miniconda3/envs/msai/lib/python3.12/site-packages/scipy/optimize/_n
```

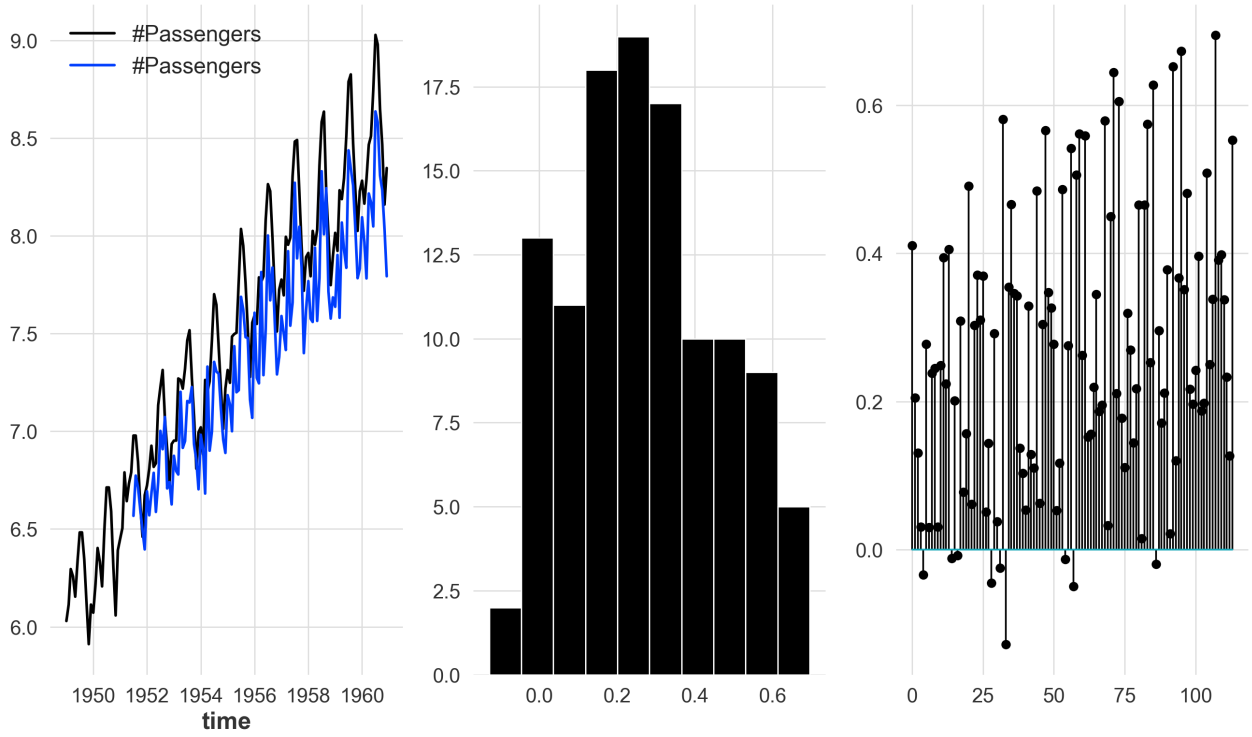
```
umdiff.py:576, in _dense_difference(fun, x0, f0, h, use_one_sided, method)
    574     x = x0 + h_vecs[i]
    575     dx = x[i] - x0[i]  # Recompute dx as exactly representable num
ber.
--> 576     df = fun(x) - f0
    577 elif method == '3-point' and use_one_sided[i]:
    578     x1 = x0 + h_vecs[i]

File ~/miniconda3/envs/msai/lib/python3.12/site-packages/scipy/optimize/_n
umdiff.py:456, in approx_derivative.<locals>.fun_wrapped(x)
    455 def fun_wrapped(x):
--> 456     f = np.atleast_1d(fun(x, *args, **kwargs))
    457     if f.ndim > 1:
    458         raise RuntimeError("`fun` return value has "
    459                            "more than 1 dimension.")

File ~/miniconda3/envs/msai/lib/python3.12/site-packages/scipy/optimize/_d
ifferentiable_functions.py:137, in ScalarFunction.__init__.<locals>.fun_wr
apped(x)
    133 self.nfev += 1
    134 # Send a copy because the user may overwrite it.
    135 # Overwriting results in undefined behaviour because
    136 # fun(self.x) will change self.x, with the two no longer linked.
--> 137 fx = fun(np.copy(x), *args)
    138 # Make sure the function returns a true scalar
    139 if not np.isscalar(fx):

File ~/miniconda3/envs/msai/lib/python3.12/site-packages/statsmodels/base/
model.py:534, in LikelihoodModel.fit.<locals>.f(params, *args)
    533 def f(params, *args):
--> 534     return -self.loglike(params, *args) / nobs

File ~/miniconda3/envs/msai/lib/python3.12/site-packages/statsmodels/tsa/s
tatespace/mlemodel.py:939, in MLEModel.loglike(self, params, *args, **kwar
gs)
    936 if complex_step:
    937     kwargs['inversion_method'] = INVERT_UNIVARIATE | SOLVE_LU
--> 939 loglike = self.ssm.loglike(complex_step=complex_step, **kwargs)
    941 # Koopman, Shephard, and Doornik recommend maximizing the average
    942 # likelihood to avoid scale issues, but the averaging is done
    943 # automatically in the base model `fit` method
    944 return loglike

File ~/miniconda3/envs/msai/lib/python3.12/site-packages/statsmodels/tsa/s
tatespace/kalman_filter.py:1001, in KalmanFilter.loglike(self, **kwargs)
    985 r"""
    986 Calculate the loglikelihood associated with the statespace model.
    987
   (...)
    997     The joint loglikelihood.
    998 """
    999 kwargs.setdefault('conserve_memory',
   1000                   MEMORY_CONSERVE ^ MEMORY_NO_LIKELIHOOD)
```

```
-> 1001 kfilter = self._filter(**kwargs)
   1002 loglikelihood_burn = kwargs.get('loglikelihood_burn',
   1003                                 self.loglikelihood_burn)
   1004 if not (kwargs['conserve_memory'] & MEMORY_NO_LIKELIHOOD):

File ~/miniconda3/envs/msai/lib/python3.12/site-packages/statsmodels/tsa/statespace/kalman_filter.py:921, in KalmanFilter._filter(self, filter_method, inversion_method, stability_method, conserve_memory, filter_timing, tolerance, loglikelihood_burn, complex_step)
    918 kfilter = self._kalman_filters[prefix]
    920 # Initialize the state
--> 921 self._initialize_state(prefix=prefix, complex_step=complex_step)
    923 # Run the filter
    924 kfilter()

File ~/miniconda3/envs/msai/lib/python3.12/site-packages/statsmodels/tsa/statespace/representation.py:1058, in Representation._initialize_state(self, prefix, complex_step)
   1056     if not self.initialization.initialized:
   1057         raise RuntimeError('Initialization is incomplete.')
-> 1058     self._statespaces[prefix].initialize(self.initialization,
   1059                                          complex_step=complex_step)
   1060 else:
   1061     raise RuntimeError('Statespace model not initialized.')

KeyboardInterrupt:
```

In [ ]:
```
fig, ax = plt.subplots()
air_pax_log.plot(ax=ax, label="data")
fcast.plot(ax=ax, label="model")
ax.legend();
```

In [75]:
```
resid = air_pax_log[fcast.start_time():] - fcast
```

In [76]:
```
fig, ax = plt.subplots(1,3,figsize=(16,9))
air_pax_log.plot(ax=ax[0]);
fcast.plot(ax=ax[0]);
ax[1].hist(resid.values());
ax[2].stem(resid.values());
```



Necessary properties of residuals: unbiasedness

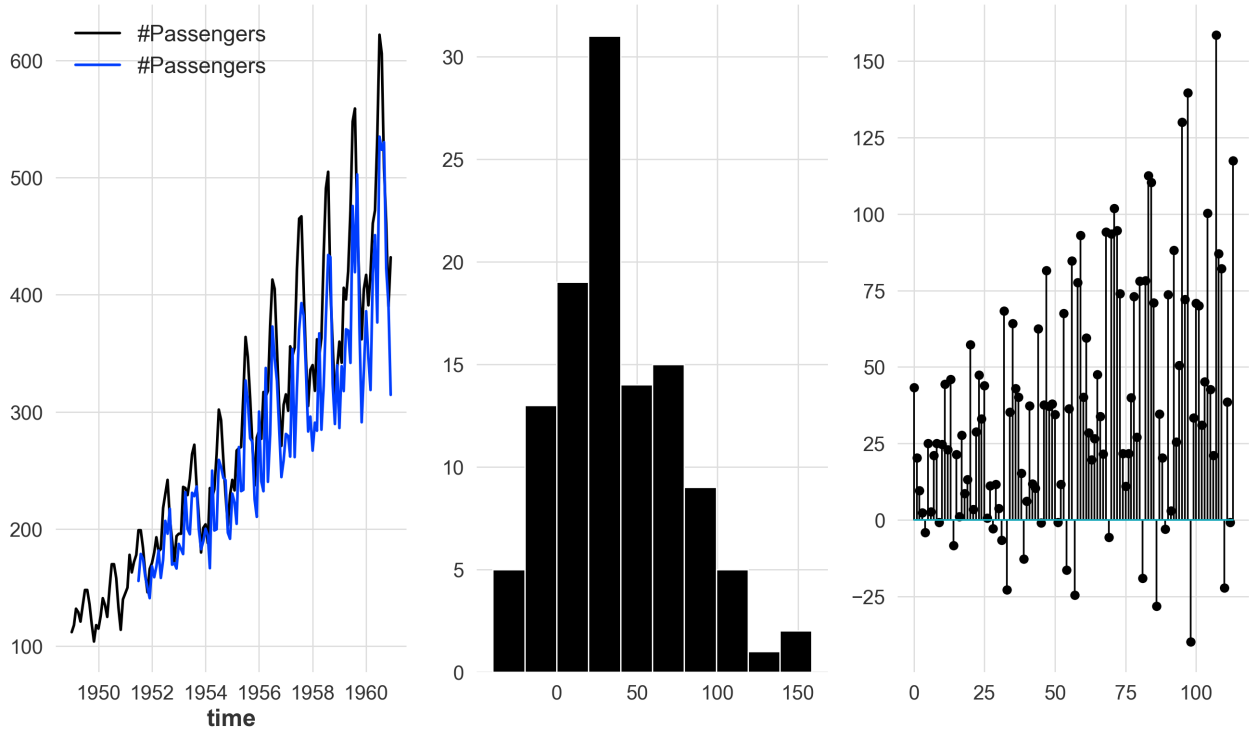
In [16]:
```
from darts.models import ARIMA
```

```
/Users/nstulov/miniconda3/envs/msai/lib/python3.12/site-packages/tqdm/aut
o.py:21: TqdmWarning: IProgress not found. Please update jupyter and ipywi
dgets. See https://ipywidgets.readthedocs.io/en/stable/user_install.html
  from .autonotebook import tqdm as notebook_tqdm
```

In [17]:
```
biased_model = ARIMA(p=0, d=0, q=2, seasonal_order=(0, 0, 0, 0))
fcast = biased_model.historical_forecasts(
    air_pax, forecast_horizon=1, start_format="position", show_warnings=F
)
resid = air_pax[fcast.start_time():] - fcast
```

```
/Users/nstulov/miniconda3/envs/msai/lib/python3.12/site-packages/statsmode
ls/tsa/statespace/sarimax.py:978: UserWarning: Non-invertible starting MA
parameters found. Using zeros as starting parameters.
  warn('Non-invertible starting MA parameters found.'
/Users/nstulov/miniconda3/envs/msai/lib/python3.12/site-packages/statsmode
ls/base/model.py:607: ConvergenceWarning: Maximum Likelihood optimization
failed to converge. Check mle_retvals
  warnings.warn("Maximum Likelihood optimization failed to "
```

In [18]:
```
fig, ax = plt.subplots(1,3,figsize=(16,9))
air_pax.plot(ax=ax[0]);
fcast.plot(ax=ax[0]);
ax[1].hist(resid.values());
ax[2].stem(resid.values());
```



In [19]:
```
pvalue = sts.ttest_1samp(resid.values(), 0).pvalue
rejected = pvalue < 0.05
print("zero mean" if not rejected else "non-zero mean")
```

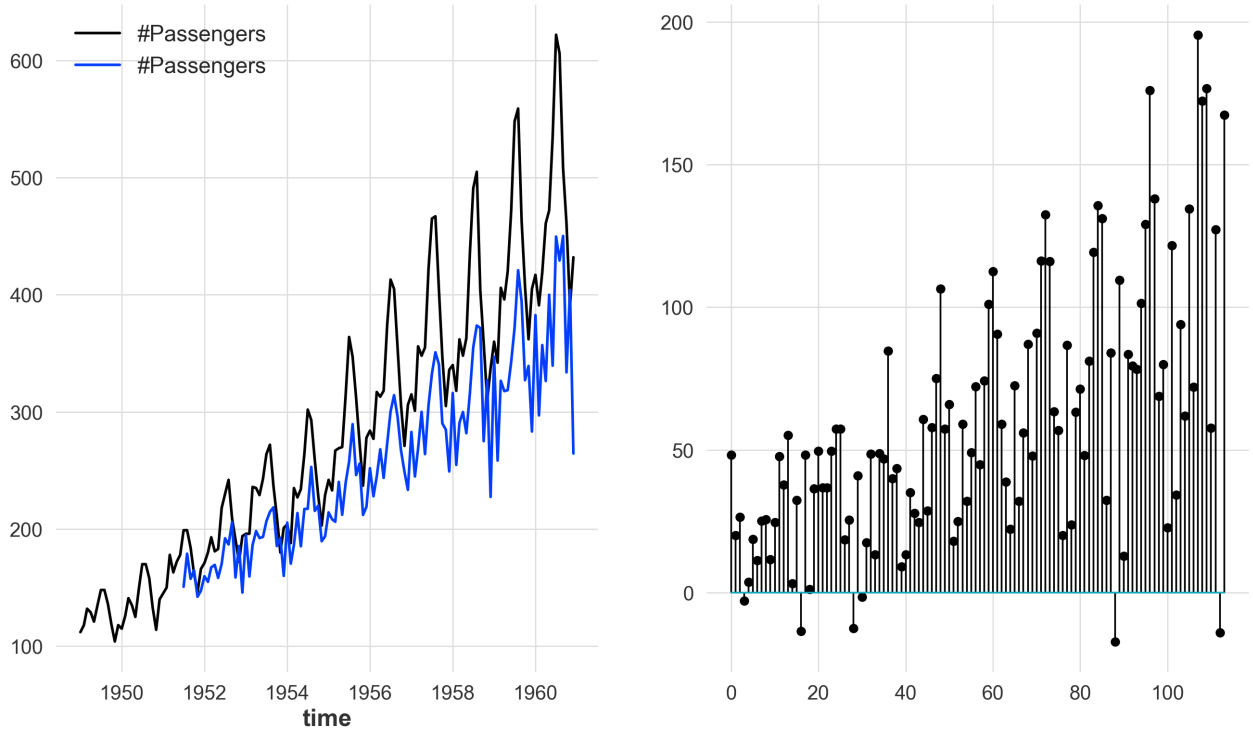
```
non-zero mean
```

Necessary properties of residuals: stationarity

In [20]:
```
unst_model = ARIMA(p=0, d=0, q=1, seasonal_order=(0, 0, 0, 0))
fcast = unst_model.historical_forecasts(
    air_pax, forecast_horizon=1, start_format="position", show_warnings=F
)
resid = air_pax[fcast.start_time():] - fcast
```

```
/Users/nstulov/miniconda3/envs/msai/lib/python3.12/site-packages/statsmode
ls/tsa/statespace/sarimax.py:978: UserWarning: Non-invertible starting MA
parameters found. Using zeros as starting parameters.
  warn('Non-invertible starting MA parameters found.'
```

In [21]:
```
fig, ax = plt.subplots(1,2,figsize=(16,9))
air_pax.plot(ax=ax[0]);
fcast.plot(ax=ax[0]);
ax[1].stem(resid.values());
```



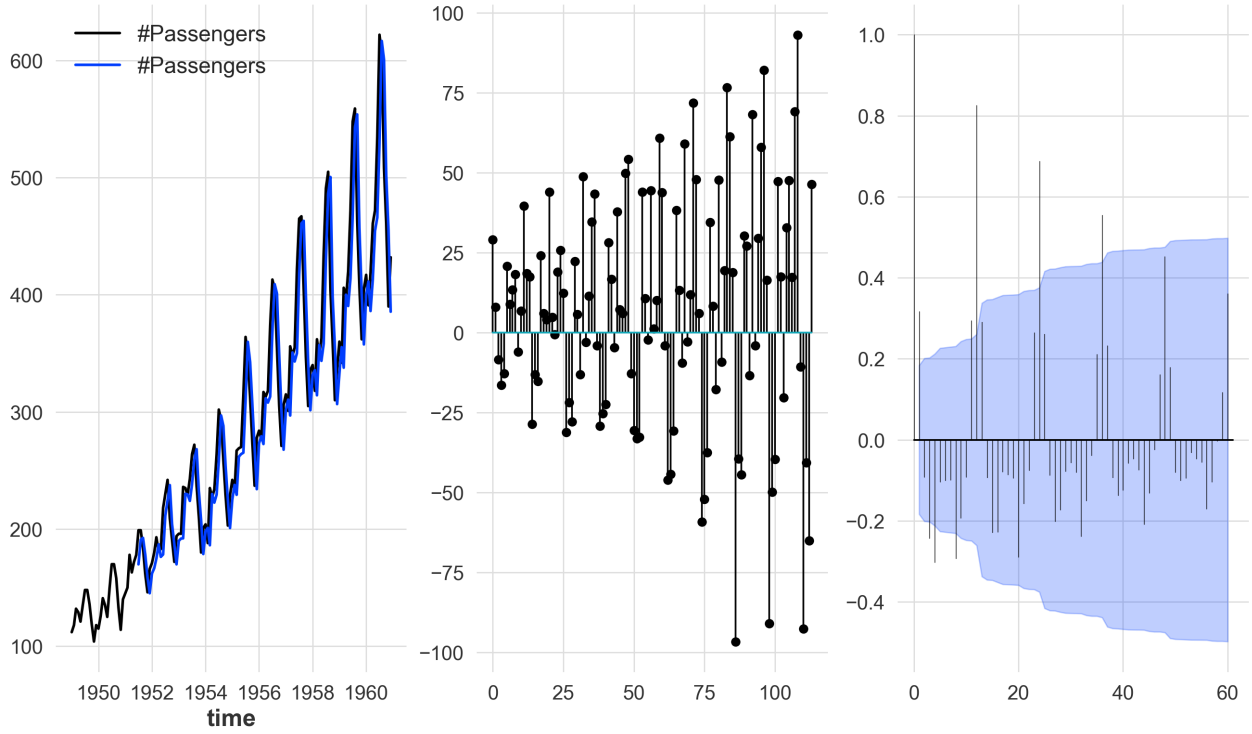
In [22]:
```
pvalue = tsa.stattools.kpss(resid.values())[1]
rejected = pvalue < 0.05
print("stationary" if not rejected else "non-stationary")
```

```
non-stationary
/var/folders/33/j0cl7y453td68qb96j7bqcj4cf41kc/T/ipykernel_823/2228572355.
py:1: InterpolationWarning: The test statistic is outside of the range of
p-values available in the
look-up table. The actual p-value is smaller than the p-value returned.

  pvalue = tsa.stattools.kpss(resid.values())[1]
```

Necessary properties of residuals: uncorrelatedness

In [23]:
```
corr_model = ARIMA(p=1, d=0, q=0, seasonal_order=(0, 0, 0, 0))
fcast = corr_model.historical_forecasts(
    air_pax, forecast_horizon=1, start_format="position", show_warnings=F
)
resid = air_pax[fcast.start_time():] - fcast
```

In [24]:
```
fig, ax = plt.subplots(1,3,figsize=(16,9))
air_pax.plot(ax=ax[0]);
fcast.plot(ax=ax[0]);
ax[1].stem(resid.values());
plot_acf(resid, max_lag=12*5, axis=ax[2])
```



In [25]:
```
sts.ttest_1samp(resid.values(), 0).pvalue
```

Out[25]:
```
array([0.04967464])
```

In [26]:
```
tsa.stattools.kpss(resid.values())[1]
```

```
/var/folders/33/j0cl7y453td68qb96j7bqcj4cf41kc/T/ipykernel_823/3591776226.
py:1: InterpolationWarning: The test statistic is outside of the range of
p-values available in the
look-up table. The actual p-value is greater than the p-value returned.

  tsa.stattools.kpss(resid.values())[1]
```

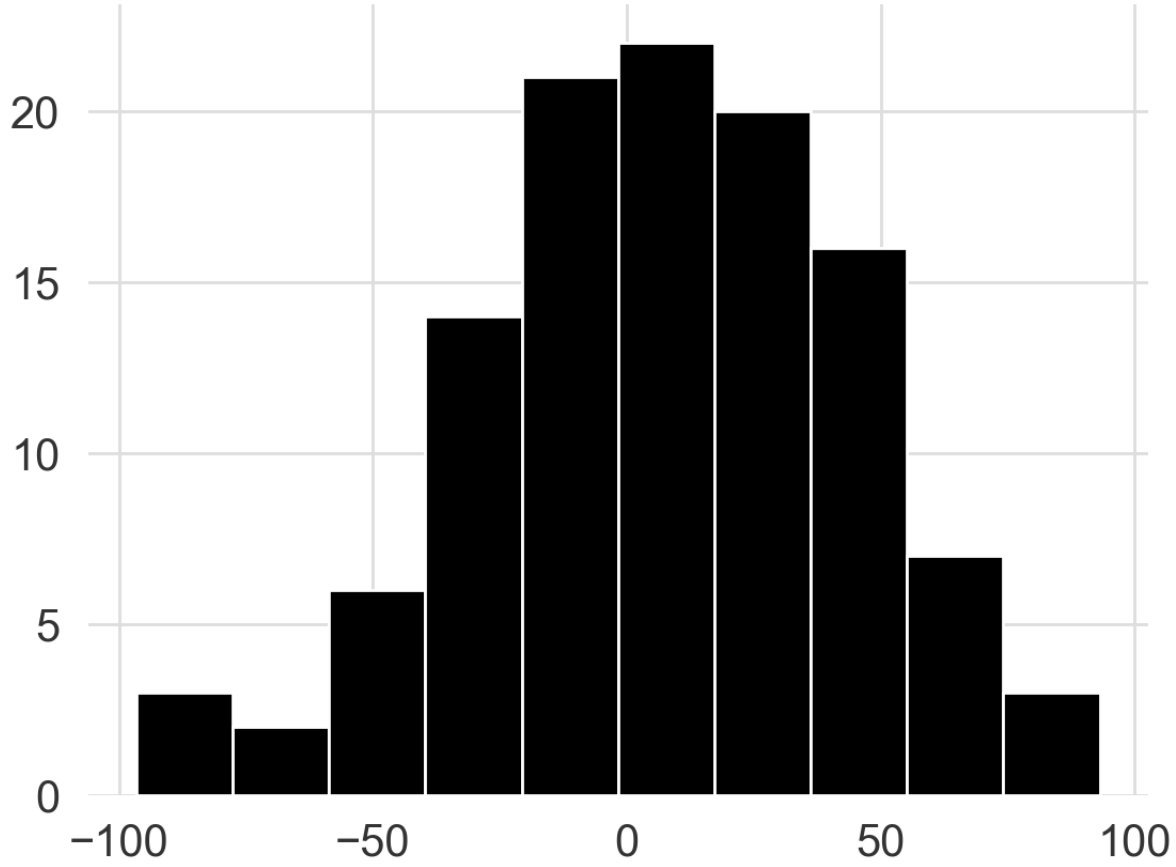
Out[26]:
```
0.1
```

In [27]:
```
sm.stats.acorr_ljungbox(resid.values(), model_df=1)["lb_pvalue"].max()
rejected = pvalue < 0.05
print("there is", "no autocorrelation" if not rejected else "autocorrelat
```

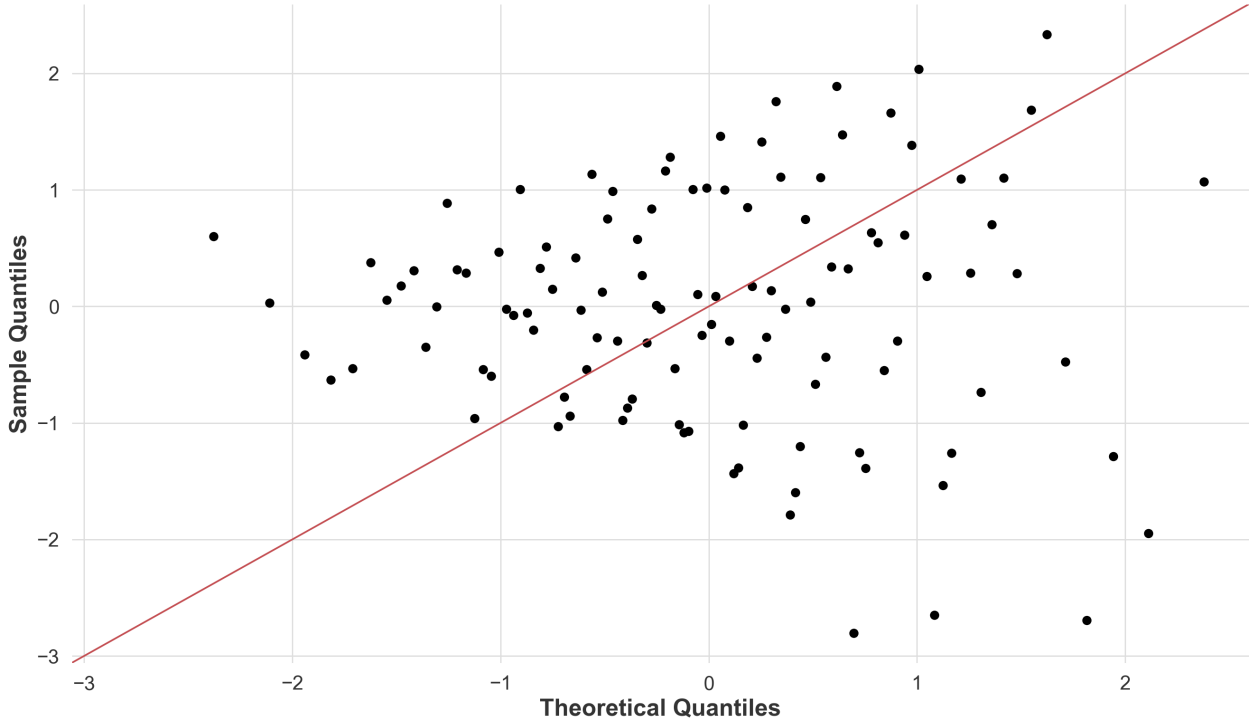
```
there is autocorrelation
```

Desired properties of residuals: normality

In [28]:
```
plt.hist(resid.values());
```



In [29]:
```
fig, ax = plt.subplots(figsize=(16,9))
sm.qqplot(resid.values(), ax=ax, fit=True, line="45");
```



In [30]:
```
pvalue = sts.shapiro(resid.values()).pvalue
rejected = pvalue < 0.05
print("normal" if not rejected else "not normal")
```

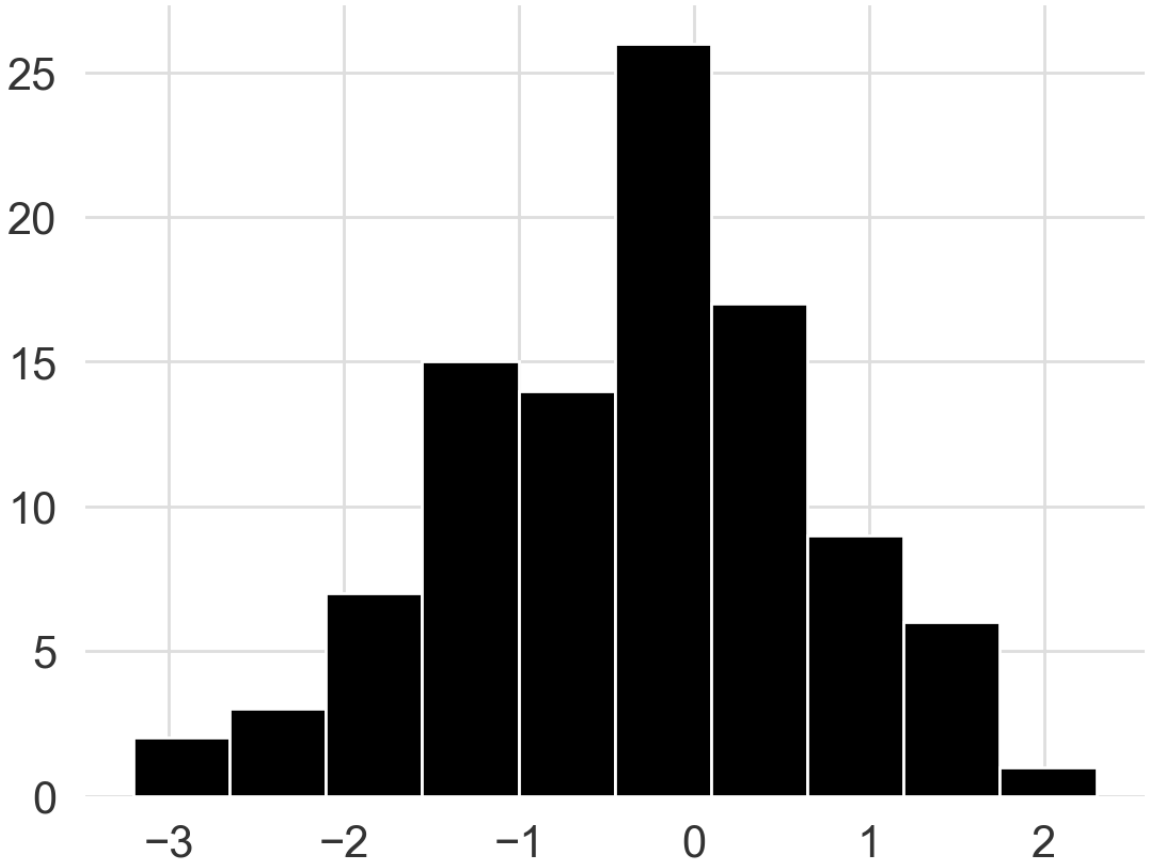
```
normal
```

In [31]:
```
pvalue
```

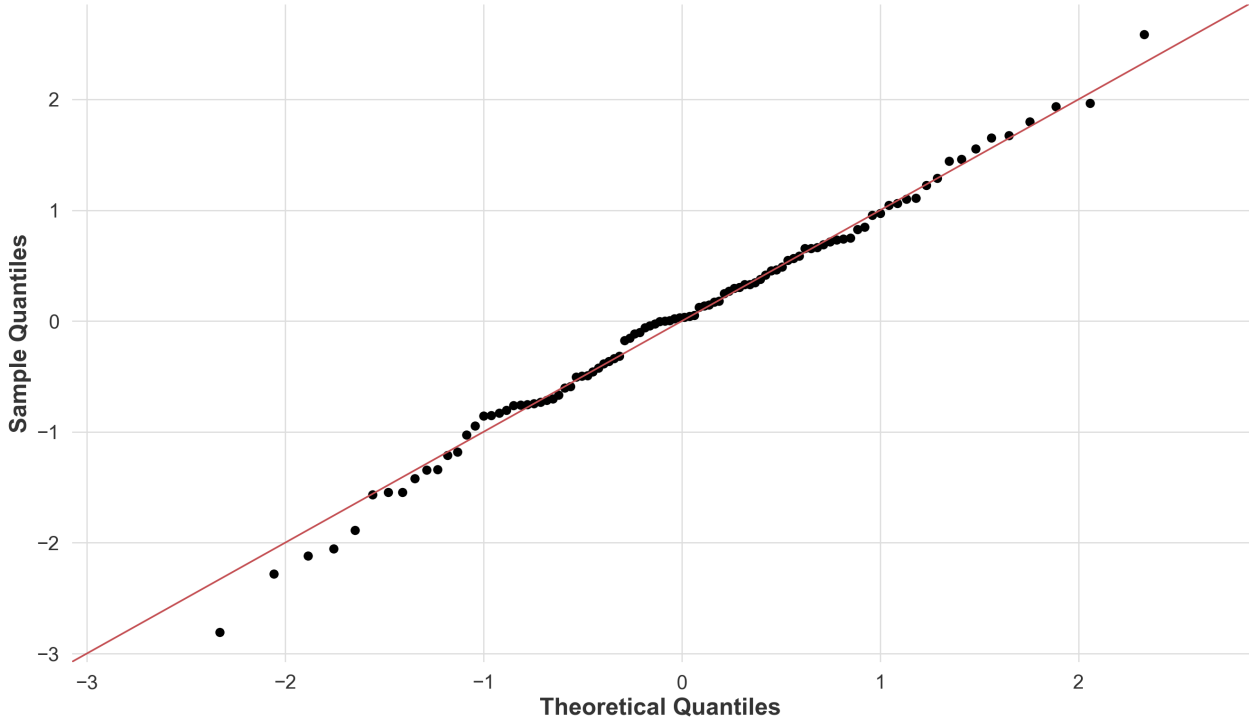
Out[31]:
```
0.5282973647117615
```

In [32]:
```
dummy_resid = sts.norm.rvs(size=100)
```

In [33]:
```
plt.hist(dummy_resid);
```



In [34]:
```
fig, ax = plt.subplots(figsize=(16,9))
sm.qqplot(dummy_resid, ax=ax, fit=True, line="45");
```



In [35]:
```
pvalue = sts.shapiro(dummy_resid).pvalue
rejected = pvalue < 0.05
print("normal" if not rejected else "not normal")
```

```
normal
```

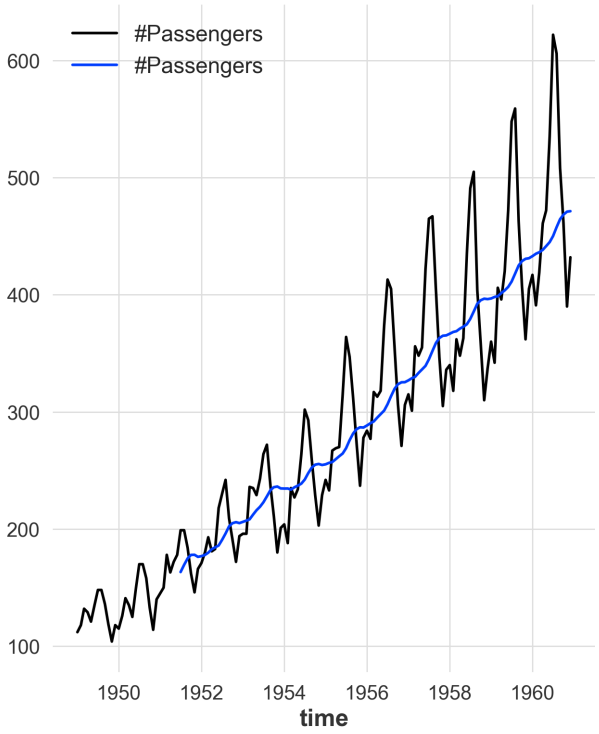
In [36]:
```
pvalue
```

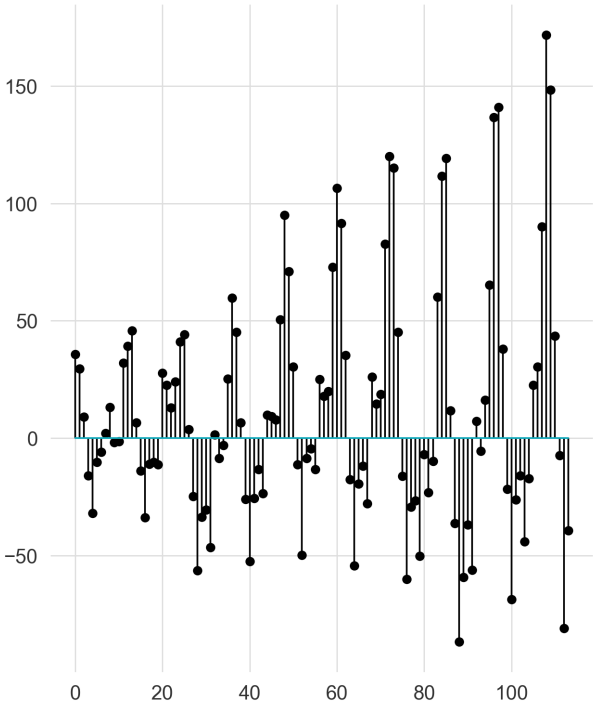
Out[36]:
```
0.9637436866760254
```

Desired properties of residuals: homoscedacity

In [37]:
```
const_model = ARIMA(p=0, d=0, q=0, seasonal_order=(0, 0, 0, 0), trend="ct
fcast = const_model.historical_forecasts(
    air_pax, start=2, forecast_horizon=1, start_format="position", show_w
)
resid = air_pax[fcast.start_time():] - fcast
```

In [38]:
```
fig, ax = plt.subplots(1,2,figsize=(16,9))
air_pax.plot(ax=ax[0]);
fcast.plot(ax=ax[0]);
ax[1].stem(resid.values());
```





In [39]:
```
# pvalue = sm.stats.diagnostic.het_breuschpagan(resid.values(), ...).pval
# rejected = pvalue < 0.05
# print("normal" if not rejected else "not normal")
```

In [ ]: