Time Series

Plan

1. Intro: what is a Time Series?
2. Why are standard ML tools not sufficient?
3. Concept: **Decomposition**
4. Concept: **Stationarity**
5. Concept: **Autocorrelation**
6. **AR** (Auto regressive processes)
7. **MA** (Moving average processes)
8. **ARMA** models
9. **ARIMA** models
10. **SARIMA** models (Optional)
11. Additional models

1️⃣ Introduction

Traditional dataset: Semi-structured data

	C1	C2	C3
Row1	23	55	88
Row2	32	1	31
Row3	12	93	80
Row4	44	0	49

=

	C1	C2	C3
Row1	23	55	88
Row4	44	0	49
Row3	12	93	80
Row2	32	1	31

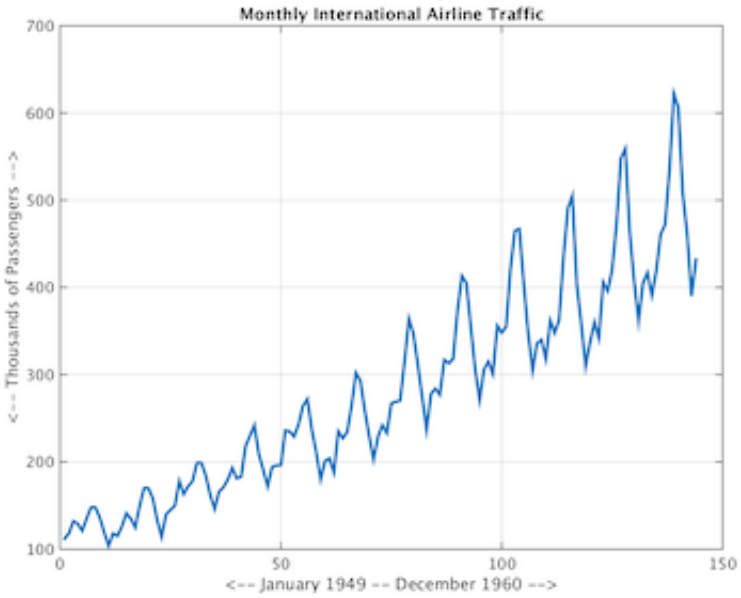
Time series: Highly structured data

Date	Value
Today	10
Yesterday	8
2 days ago	7
...	...
Last week	2

Order matters

What is a Time Series?

- A Time Series is a series of repeated observations considered within a certain time interval
- The observations are taken at equal (regular/evenly spaced) time intervals



Time series data examples:

- Stock prices: 💹
- Weather data: ☁️🌦️🌧️🌞
- Sales data: 💰
- Heavy machinery logging data: 🏗️
- Telemetry data: 🏎️
- User activity: 👤

What are our goals with Time Series data?

👉 "Understand" a TS = decompose, explain the behavior of a Time Series

👉 "Forecast" = predict future values based solely on existing/past observations

Here is our dataset (https://wagon-public-datasets.s3.amazonaws.com/05-Machine-Learning/09-Time-Series/new_drugs.csv):

In [ ]:
```
# Reading data
df = pd.read_csv('data/drugs.csv')

# Converting dates to datetime objects
df['date'] = pd.to_datetime(df['date'], infer_datetime_format=True)

# Setting dates as the index
df.set_index(['date'], inplace=True)

df
```

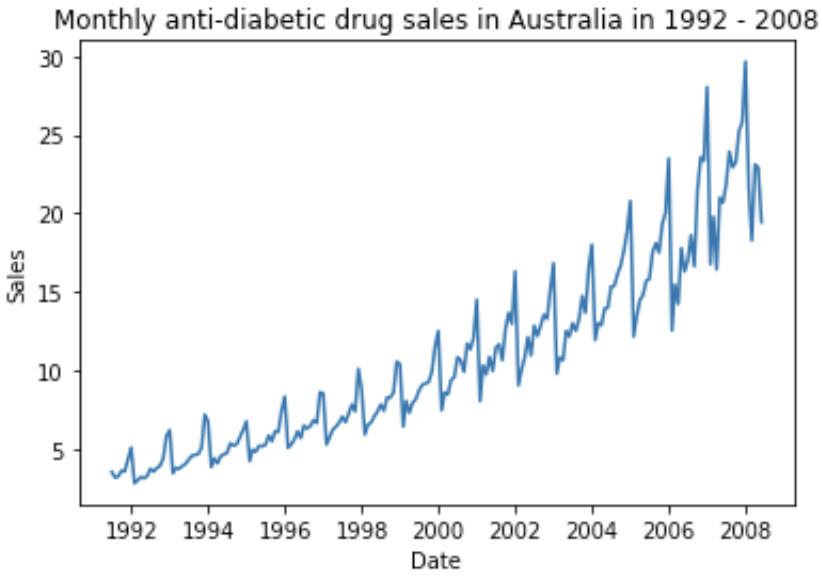
Out[ ]:

	value
date	
1991-07-01	3.526591
1991-08-01	3.180891
1991-09-01	3.252221
1991-10-01	3.611003
1991-11-01	3.565869
...	...
2008-02-01	21.654285
2008-03-01	18.264945
2008-04-01	23.107677
2008-05-01	22.912510
2008-06-01	19.431740

204 rows × 1 columns

date	value
1991-07-01	3.526591
1991-08-01	3.180891
1991-09-01	3.252221
1991-10-01	3.611003
1991-11-01	3.565869
...	...
2008-02-01	21.654285
2008-03-01	18.264945
2008-04-01	23.107677
2008-05-01	22.912510
2008-06-01	19.431740

204 rows × 1 columns



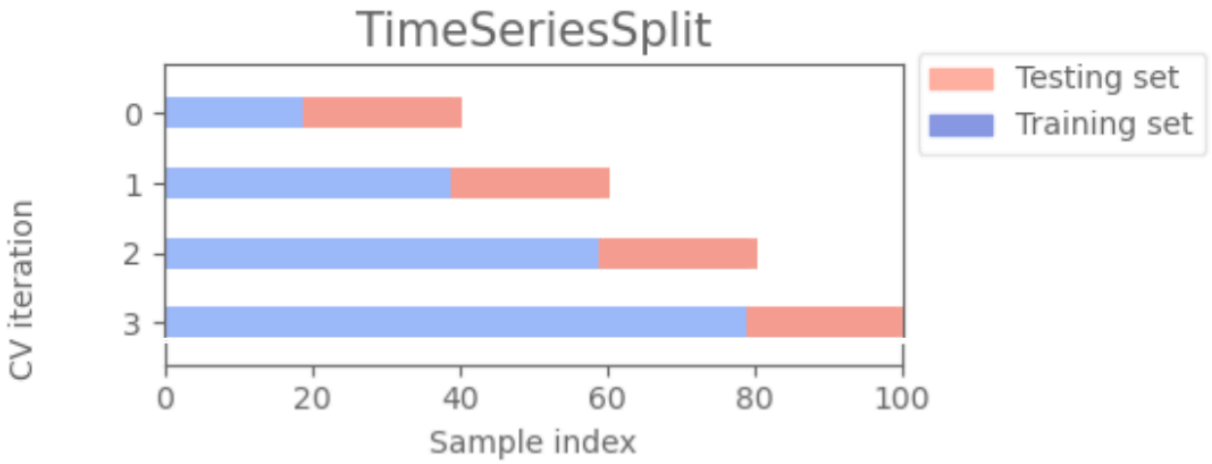
2️⃣ Classical ML approach to Time Series

❓ What is our (X, y) dataset

- `y = df['value']`
- `X` = only information available = past `y` values
- Each row/observation is `X[t] = [y[t-1], y[t-2], ....]`

date	value	t - 1	t - 2	t - 3
1992-07-01	3.737851	3.270523	3.127578	3.204780
1992-08-01	3.558776	3.737851	3.270523	3.127578
1992-09-01	3.777202	3.558776	3.737851	3.270523
1992-10-01	3.924490	3.777202	3.558776	3.737851
1992-11-01	4.386531	3.924490	3.777202	3.558776

⚠️ `train_test_split`



❌ We can't use the traditional Holdout method, a regular `train_test_split`

⚠️ It would cause data leakage as we can't use future values to predict past values...

✅ Contiguous `train_test_split` mandatory

In [ ]:
```
# Let's keep the last 40% of the values out for testing purposes
train_size = 0.6
index = round(train_size*df.shape[0])

df_train = df.iloc[:index]
df_test = df.iloc[index:]
```

2.1 Predict NEXT datapoint (1-month horizon)

Baseline (simplest model)

- 1 feature only: `X[t] = y[t-1]`
- Predict the previous value!

In [ ]:
```
y_pred = df_test.shift(1)
y_pred
```

Out[ ]:

date	value
2001-09-01	NaN
2001-10-01	10.647060
2001-11-01	12.652134
2001-12-01	13.674466
2002-01-01	12.965735
...	...
2008-02-01	29.665356
2008-03-01	21.654285
2008-04-01	18.264945
2008-05-01	23.107677
2008-06-01	22.912510

82 rows × 1 columns

In [ ]:
```
from sklearn.metrics import r2_score

y_pred = df_test.shift(1).dropna()
y_true = df_test[1:]

print(f"R2: {r2_score(y_true, y_pred)}")
```

```
R2: 0.5069517261286796
```

Linear model with 12 autoregressive features

Let's build our dataset X with 12 autoregressive features

In [ ]:
```
df2 = df.copy(); df2_train = df_train.copy(); df2_test = df_test.cop
y()

for i in range(1, 13):
    df2_train[f't - {i}'] = df_train['value'].shift(i)
    df2_test[f't - {i}'] = df_test['value'].shift(i)

df2_train.dropna(inplace=True)
df2_test.dropna(inplace=True)

df2_train.head()
```

Out[ ]:

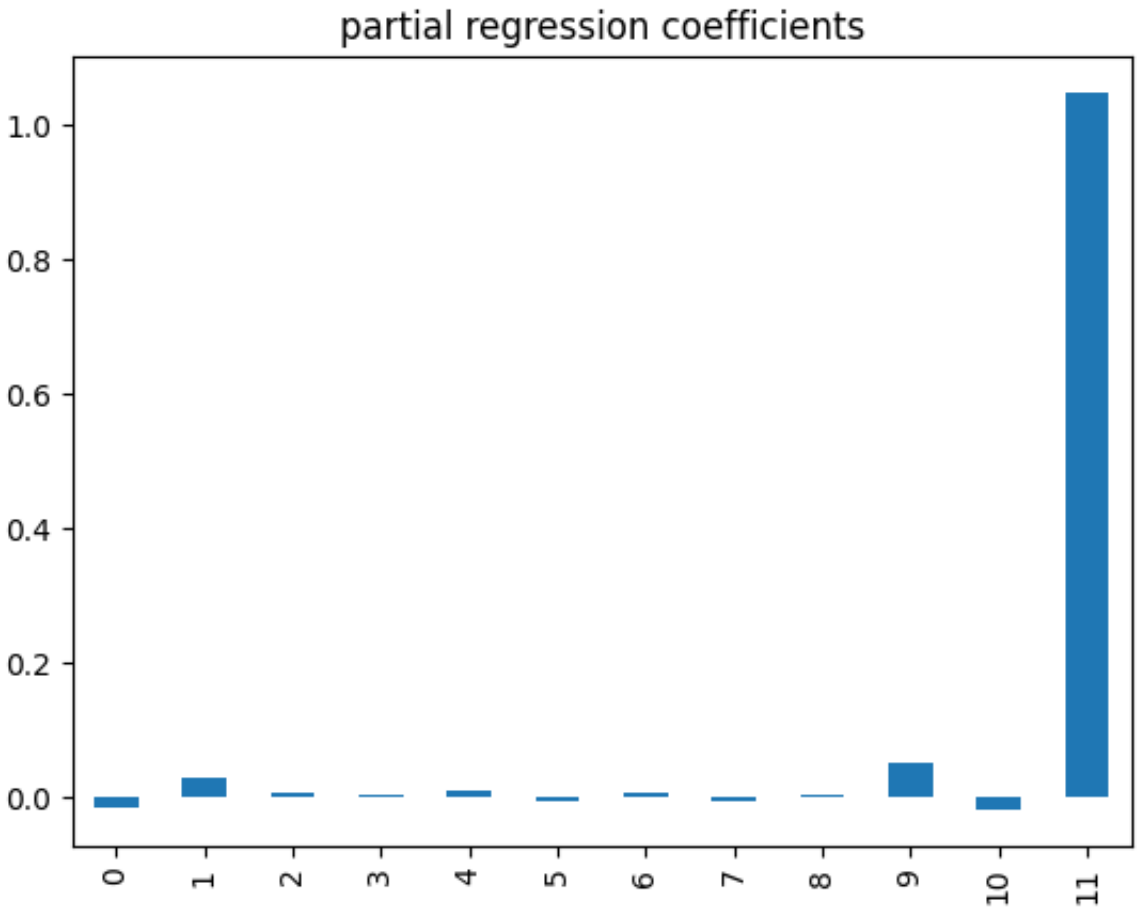
date	value	t - 1	t - 2	t - 3	t - 4	t - 5	t - 6	t - 7	t -
1992-07-01	3.737851	3.270523	3.127578	3.204780	2.985811	2.814520	5.088335	4.306371	3.5658
1992-08-01	3.558776	3.737851	3.270523	3.127578	3.204780	2.985811	2.814520	5.088335	4.3063
1992-09-01	3.777202	3.558776	3.737851	3.270523	3.127578	3.204780	2.985811	2.814520	5.0883
1992-10-01	3.924490	3.777202	3.558776	3.737851	3.270523	3.127578	3.204780	2.985811	2.8145
1992-11-01	4.386531	3.924490	3.777202	3.558776	3.737851	3.270523	3.127578	3.204780	2.9858

In [ ]:
```
# Train Test Split
X2_train = df2_train.drop(columns = ['value'])
y2_train = df2_train['value']
X2_test = df2_test.drop(columns = ['value'])
y2_test = df2_test['value']

print(X2_train.shape,y2_train.shape, X2_test.shape,y2_test.shape)
```

```
(110, 12) (110,) (70, 12) (70,)
```

In [ ]:
```
# Predict and measure R2
model = LinearRegression()
model = model.fit(X2_train, y2_train)

print('R2: ', r2_score(y2_test, model.predict(X2_test)))
pd.Series(model.coef_).plot(kind='bar')
plt.title('partial regression coefficients');
```

```
R2:  0.8580874548863758
```



- Only the 12th coefficient conveys information to the model (a Linear Regression based on these 12 autoregressive features)

2.2 Predict a longer time horizon?

We need to train **one model per forecast horizon!**

Horizon = 1

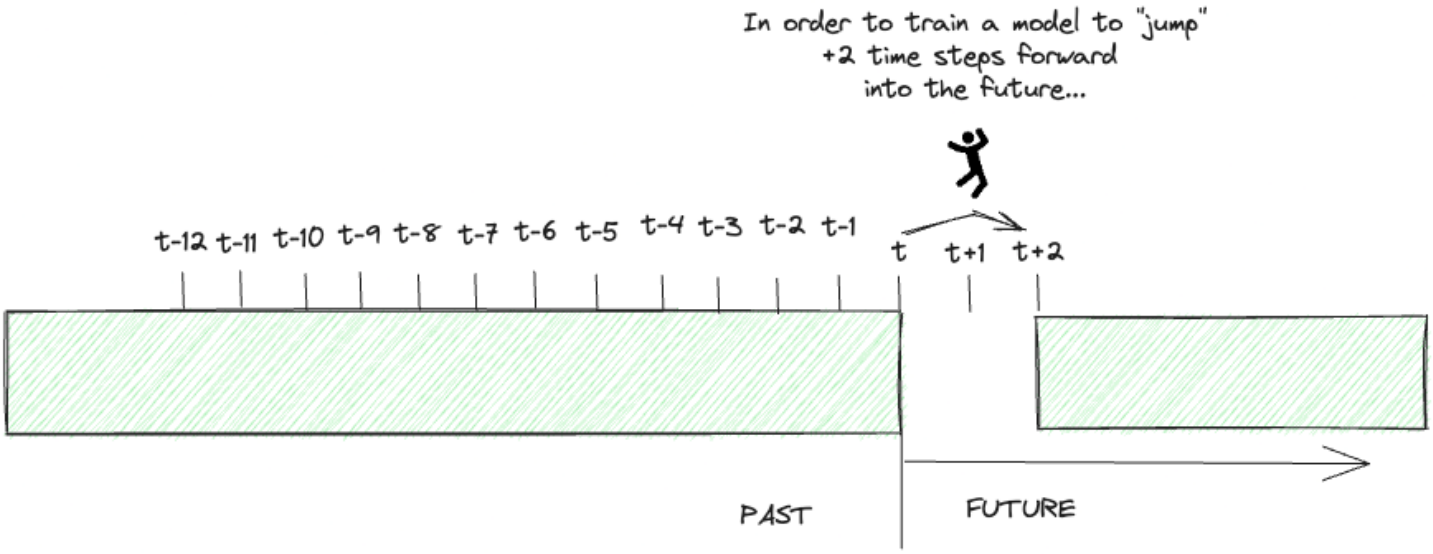
date	value	t - 1	t - 2	t - 3	t - 4	t - 5	t - 6	t - 7	t - 8	t - 9	t - 10	t - 11	t - 12
1992-07-01	3.737851	3.270523	3.127578	3.204780	2.985811	2.814520	5.088335	4.306371	3.565869	3.611003	3.252221	3.180891	3.526591
1992-08-01	3.558776	3.737851	3.270523	3.127578	3.204780	2.985811	2.814520	5.088335	4.306371	3.565869	3.611003	3.252221	3.180891
1992-09-01	3.777202	3.558776	3.737851	3.270523	3.127578	3.204780	2.985811	2.814520	5.088335	4.306371	3.565869	3.611003	3.252221
1992-10-01	3.924490	3.777202	3.558776	3.737851	3.270523	3.127578	3.204780	2.985811	2.814520	5.088335	4.306371	3.565869	3.611003
1992-11-01	4.386531	3.924490	3.777202	3.558776	3.737851	3.270523	3.127578	3.204780	2.985811	2.814520	5.088335	4.306371	3.565869

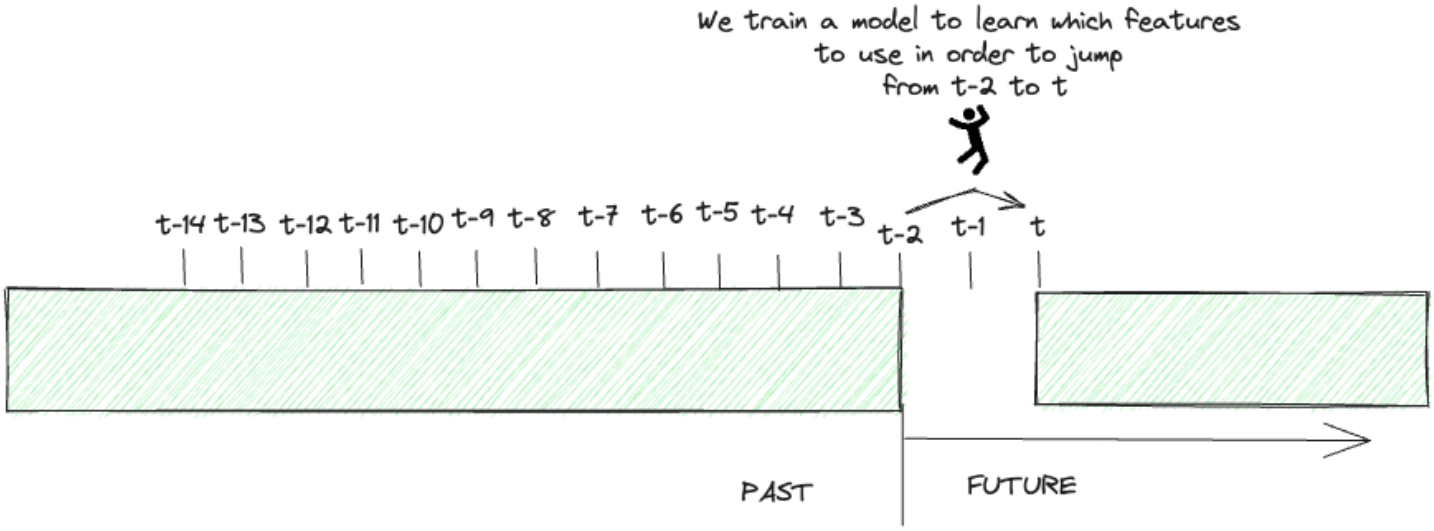
Horizon = 2

date	value	t - 2	t - 3	t - 4	t - 5	t - 6	t - 7	t - 8	t - 9	t - 10	t - 11	t - 12	t - 13
1992-08-01	3.558776	3.270523	3.127578	3.204780	2.985811	2.814520	5.088335	4.306371	3.565869	3.611003	3.252221	3.180891	3.526591
1992-09-01	3.777202	3.737851	3.270523	3.127578	3.204780	2.985811	2.814520	5.088335	4.306371	3.565869	3.611003	3.252221	3.180891
1992-10-01	3.924490	3.558776	3.737851	3.270523	3.127578	3.204780	2.985811	2.814520	5.088335	4.306371	3.565869	3.611003	3.252221
1992-11-01	4.386531	3.777202	3.558776	3.737851	3.270523	3.127578	3.204780	2.985811	2.814520	5.088335	4.306371	3.565869	3.611003
1992-12-01	5.810549	3.924490	3.777202	3.558776	3.737851	3.270523	3.127578	3.204780	2.985811	2.814520	5.088335	4.306371	3.565869

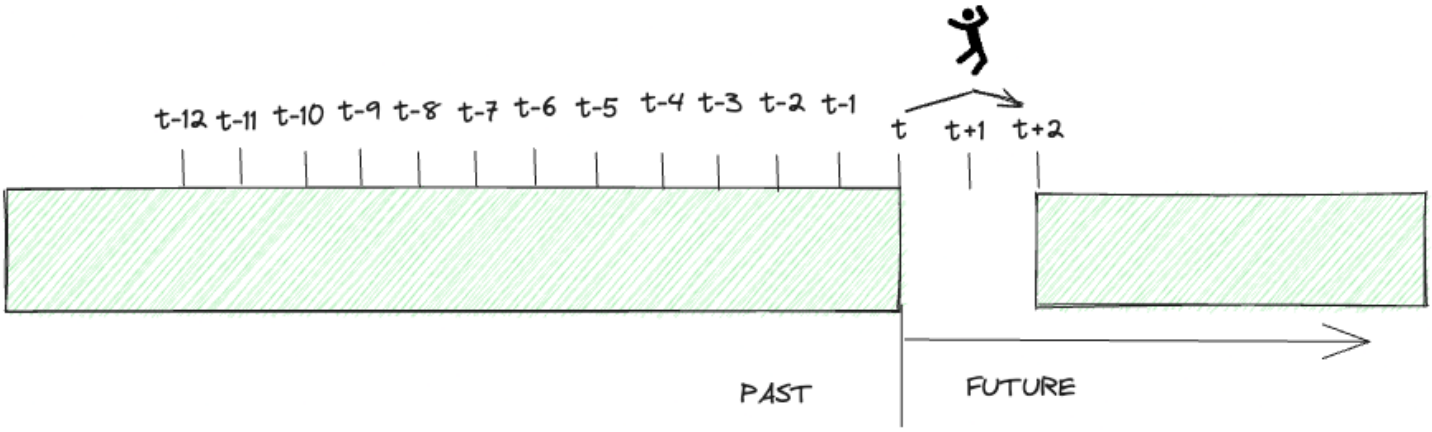
Horizon = 3

date	value	t - 3	t - 4	t - 5	t - 6	t - 7	t - 8	t - 9	t - 10	t - 11	t - 12	t - 13	t - 14
1992-09-01	3.777202	3.270523	3.127578	3.204780	2.985811	2.814520	5.088335	4.306371	3.565869	3.611003	3.252221	3.180891	3.526591
1992-10-01	3.924490	3.737851	3.270523	3.127578	3.204780	2.985811	2.814520	5.088335	4.306371	3.565869	3.611003	3.252221	3.180891
1992-11-01	4.386531	3.558776	3.737851	3.270523	3.127578	3.204780	2.985811	2.814520	5.088335	4.306371	3.565869	3.611003	3.252221
1992-12-01	5.810549	3.777202	3.558776	3.737851	3.270523	3.127578	3.204780	2.985811	2.814520	5.088335	4.306371	3.565869	3.611003
1993-01-01	6.192068	3.924490	3.777202	3.558776	3.737851	3.270523	3.127578	3.204780	2.985811	2.814520	5.088335	4.306371	3.565869

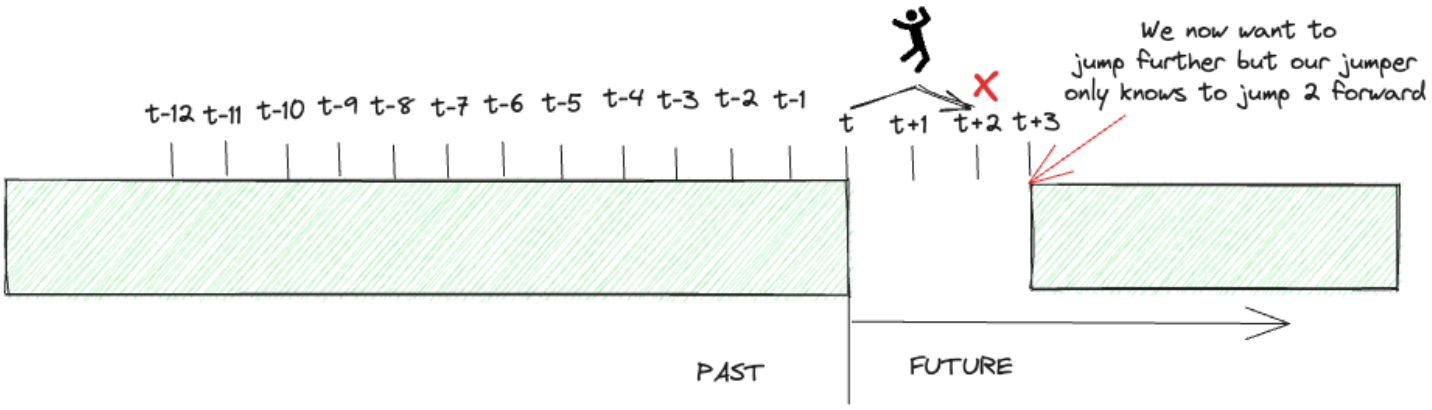

In order to train a model to "jump"
+2 time steps forward
into the future...
t-12 t-11 t-10 t-9 t-8 t-7 t-6 t-5 t-4 t-3 t-2 t-1
t t+1 t+2
PAST
FUTURE

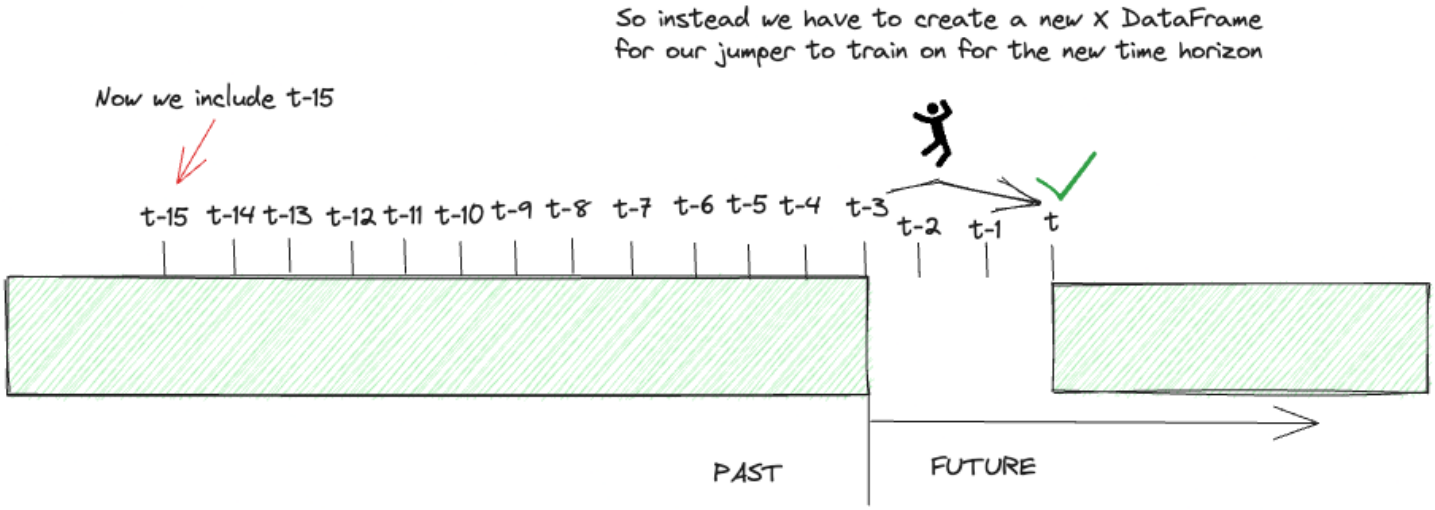

We train a model to learn which features
to use in order to jump
from t-2 to t
t-14 t-13 t-12 t-11 t-10 t-9 t-8 t-7 t-6 t-5 t-4 t-3
t-2 t-1 t
PAST
FUTURE

Until finally we use all of
our available data when we predict for real

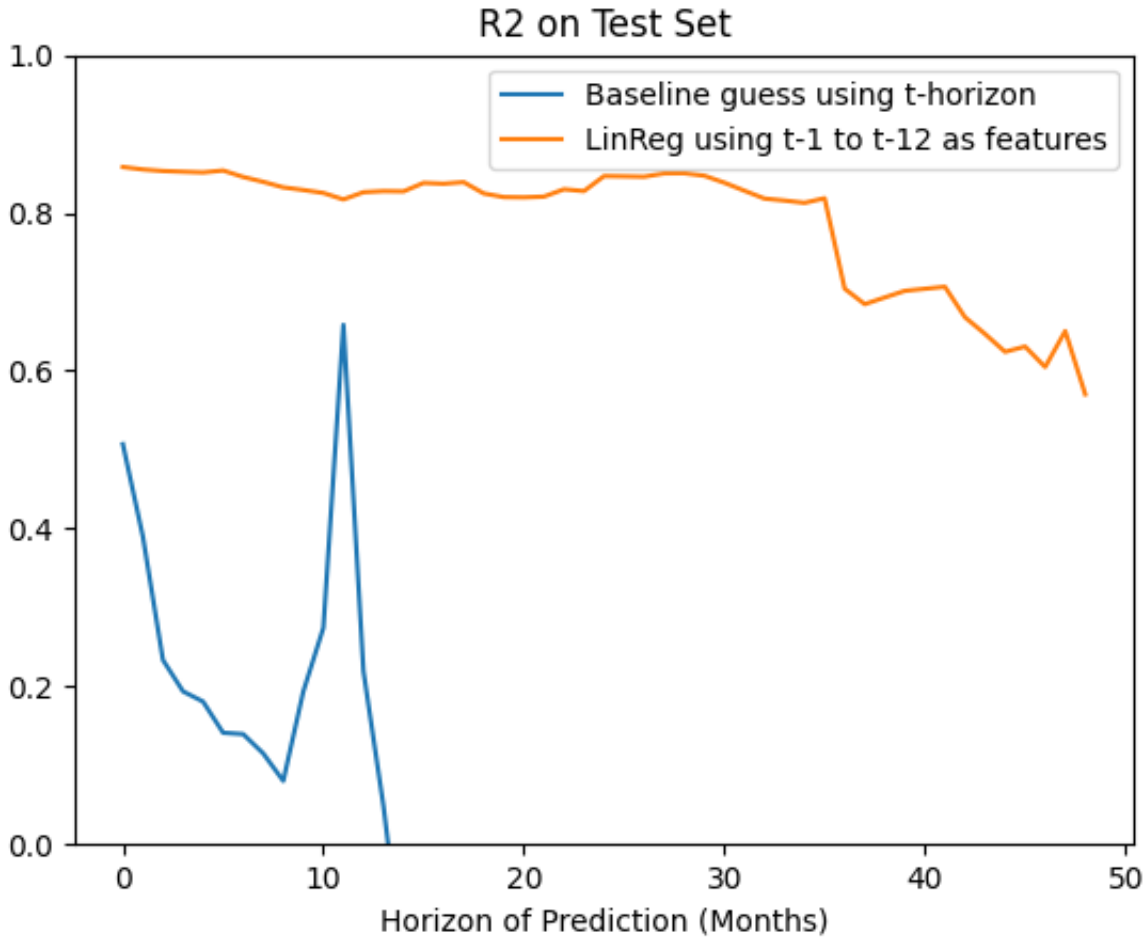


The problem is we can't easily change our horizon.
Our model will have to be refitted with new data
and learn new coefficients.




So instead we have to create a new X DataFrame
for our jumper to train on for the new time horizon
Now we include t-15
t-15 t-14 t-13 t-12 t-11 t-10 t-9 t-8 t-7 t-6 t-5 t-4 t-3 t-2 t-1 t
PAST
FUTURE

Results after 50 models trained!



❗ Model performance drops quickly as the horizon increases
❓ Need to train one model per horizon?

👉 **Use tools that are specific to Time Series**

3️⃣ Decomposition

Most Time Series can be decomposed into several components:

1. **Trend**
2. **Seasonal** (calendar)/**Periodic** (non-calendar)
3. **Irregularities**

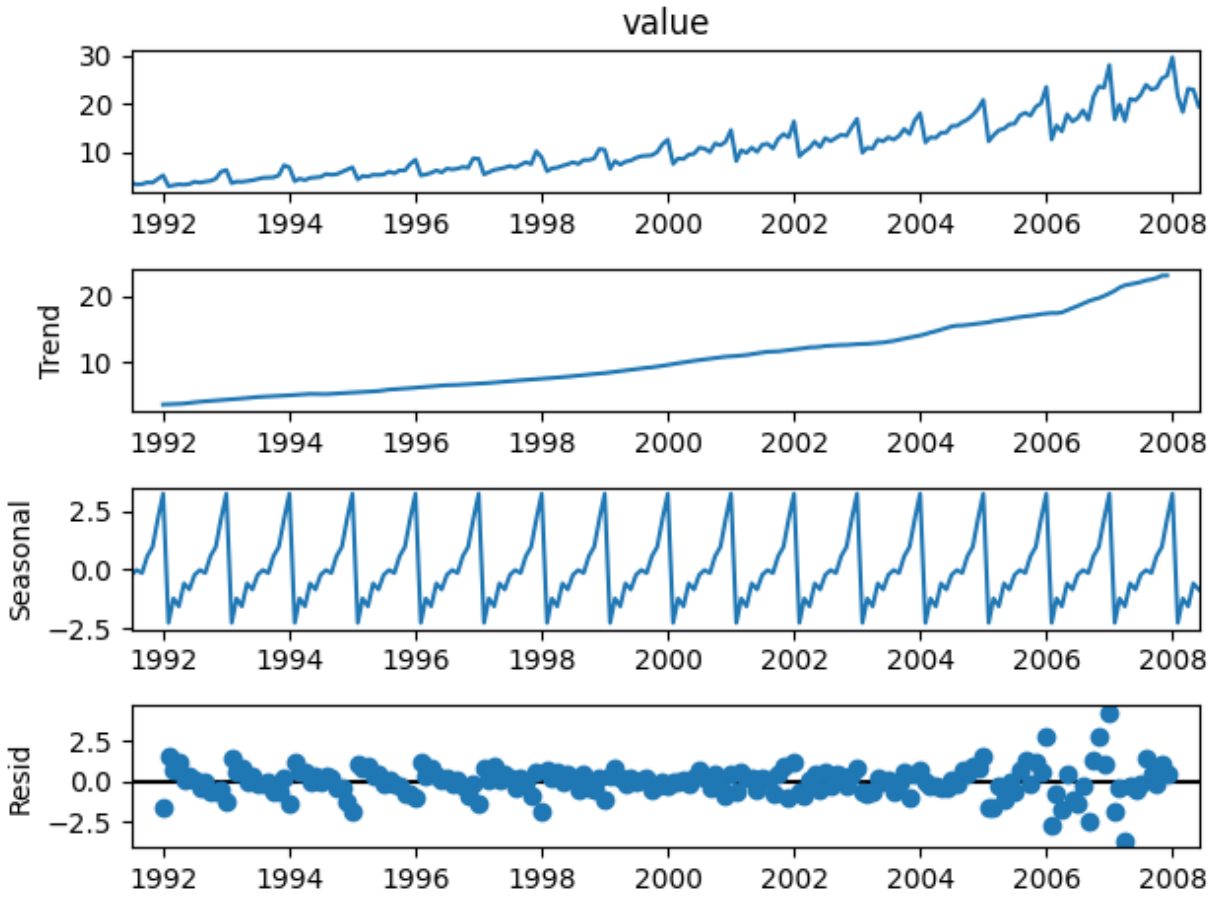
These components can be additive or multiplicative

How to decompose them?

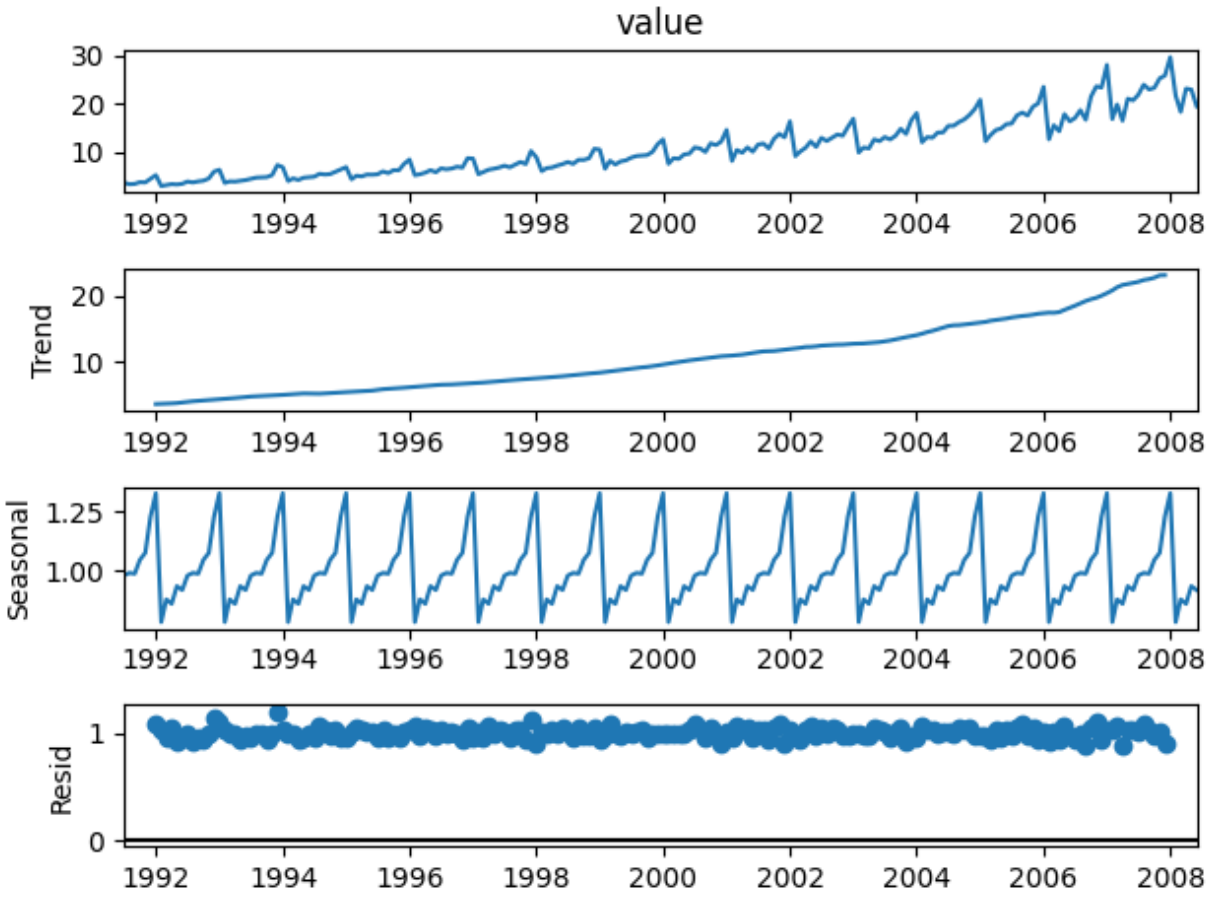
- Manually (using piece-wise OLS regressions, for instance)
- Using `statsmodels.tsa`

In [ ]:
```
from statsmodels.tsa.seasonal import seasonal_decompose
```

In [ ]:
```
# Additive Decomposition (y = Trend + Seasonal + Residuals)
result_add = seasonal_decompose(df['value'], model='additive')
result_add.plot();
```

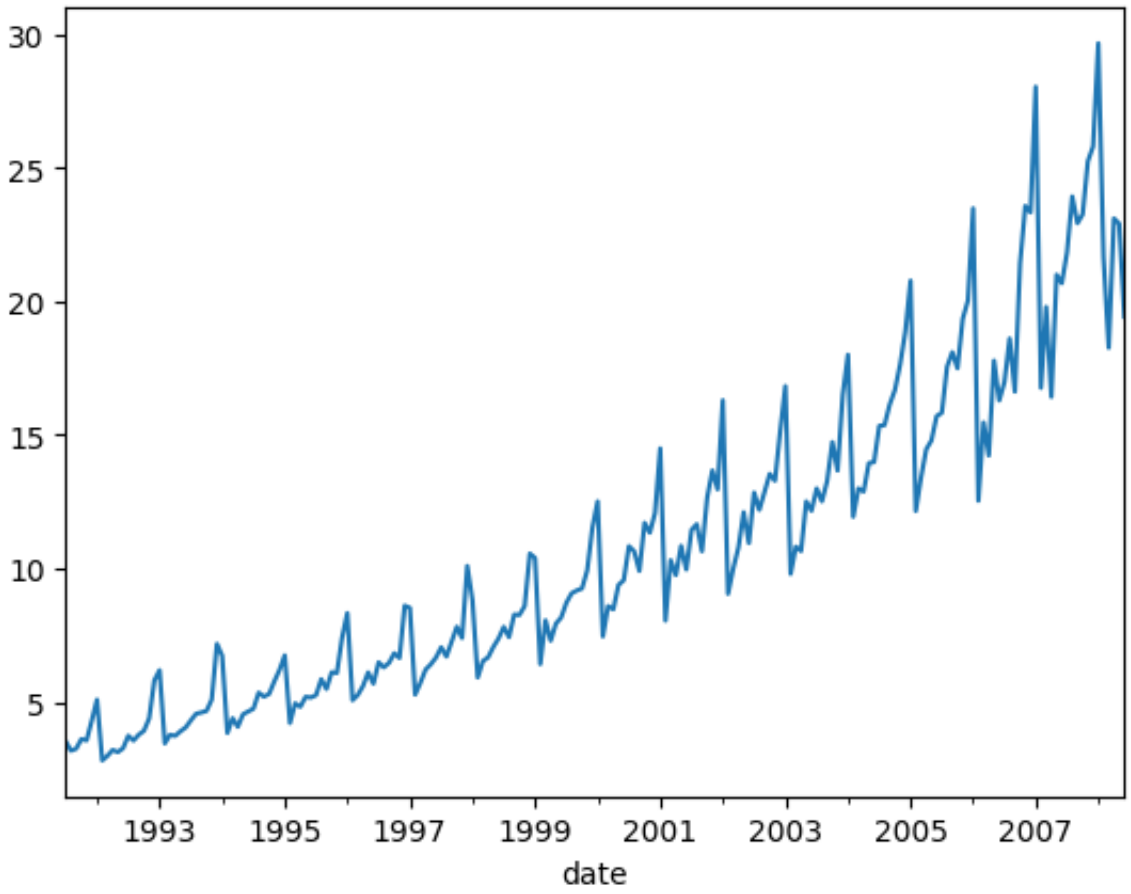

value
30
20
10
1992
1994
1996
1998
2000
2002
2004
2006
2008
Trend
20
10
Seasonal
2.5
0.0
−2.5
Resid
2.5
0.0
−2.5

In [ ]:
```
# Multiplicative Decomposition (y = Trend * Seasonal * Residuals)
result_mul = seasonal_decompose(df['value'], model='multiplicative')
result_mul.plot();
```



In [ ]:
```
df["value"].plot()
```

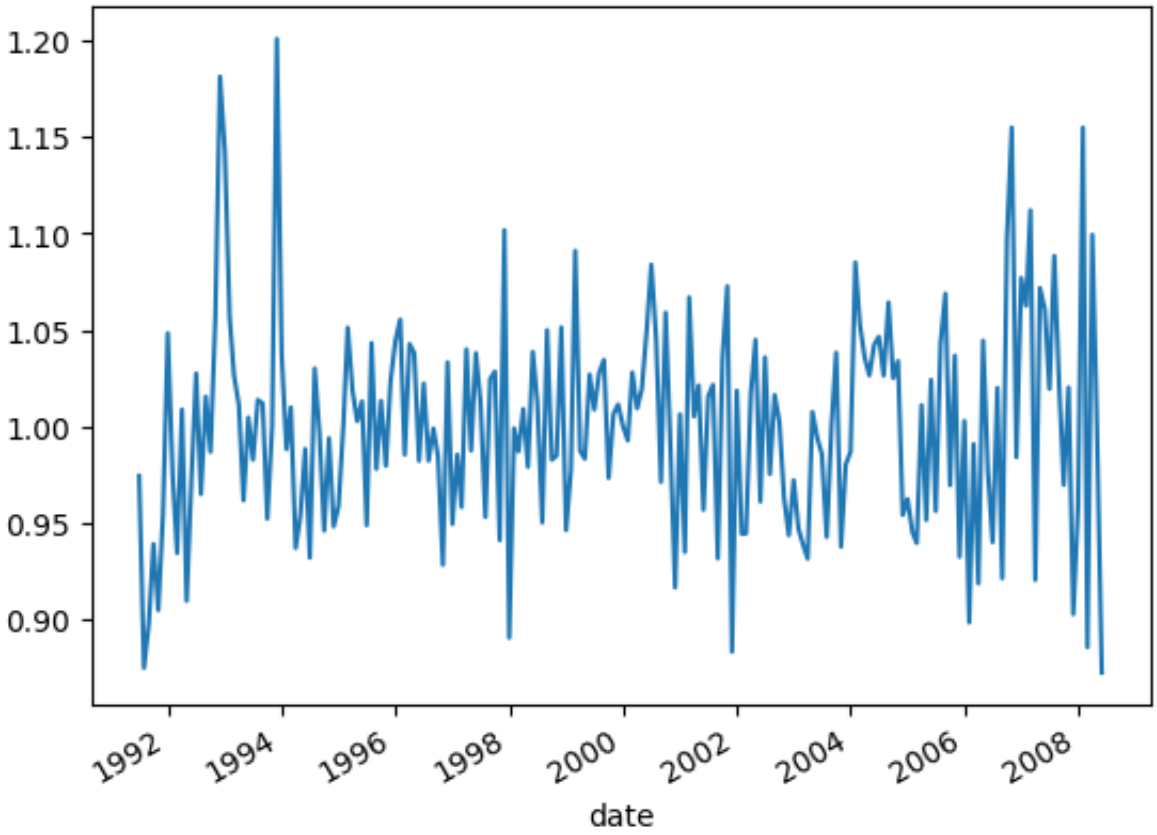
Out[ ]: <AxesSubplot:xlabel='date'>



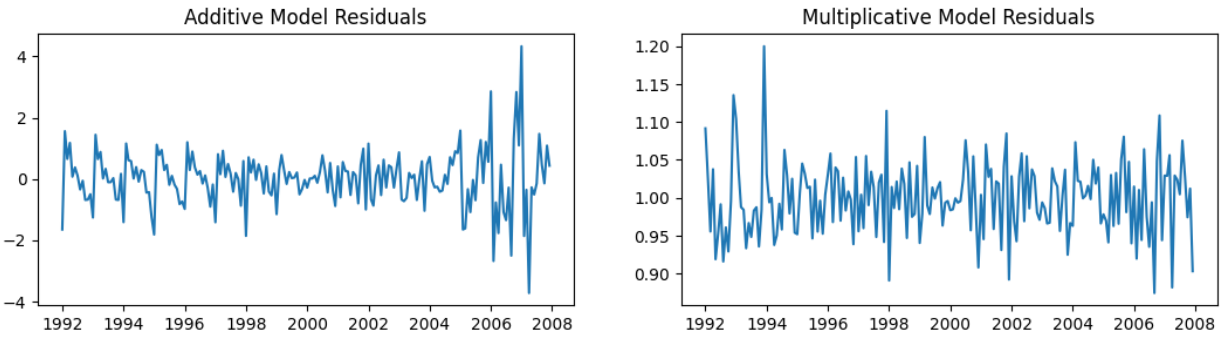
In [ ]:
```
# Use rolling mean to smooth our data (decompose uses a filter for tre
nd)
df["trend"] = df["value"].rolling(12).mean()
df["trend"].fillna(method='bfill', inplace=True)

# Divide our original 'value' by the monthly mean
df["trend_stripped"] = df["value"] / df["trend"]

# Group the remainder by month and take the mean for each month
monthly_means = df.groupby(df.index.month)[["trend_stripped"]].mean().
reset_index()
monthly_means.columns = ["month", "monthly_value"]

# Add these values back onto our original DataFrame
df["month"] = df.index.month
joined = df.reset_index().merge(monthly_means, on = "month")

# Divide our values one more time by our monthly component
joined["residuals"] = joined["trend_stripped"] / joined["monthly_valu
e"]
joined.set_index("date", inplace = True)
```

In [ ]:
```
joined["residuals"].plot()
```

Out[ ]: <AxesSubplot:xlabel='date'>



In [ ]:
```
# Plot the residuals with "result_add.resid" to decide
f, (ax1, ax2) = plt.subplots(1,2, figsize=(13,3))
ax1.plot(result_add.resid); ax1.set_title("Additive Model Residuals")
ax2.plot(result_mul.resid); ax2.set_title("Multiplicative Model Residuals");
```



☝️ Multiplicative residuals seem to have **"less notion of time"** ➔ better

🤔 Why? Models work best when forecasting TS that **do not exhibit meaningful statistical changes over time** (so that we can capture these statistical properties and project/extrapolate them into the future)

In time series we call this **`stationarity`** .

✅ Most statistical forecasting methods are designed to work on **stationary Time Series**

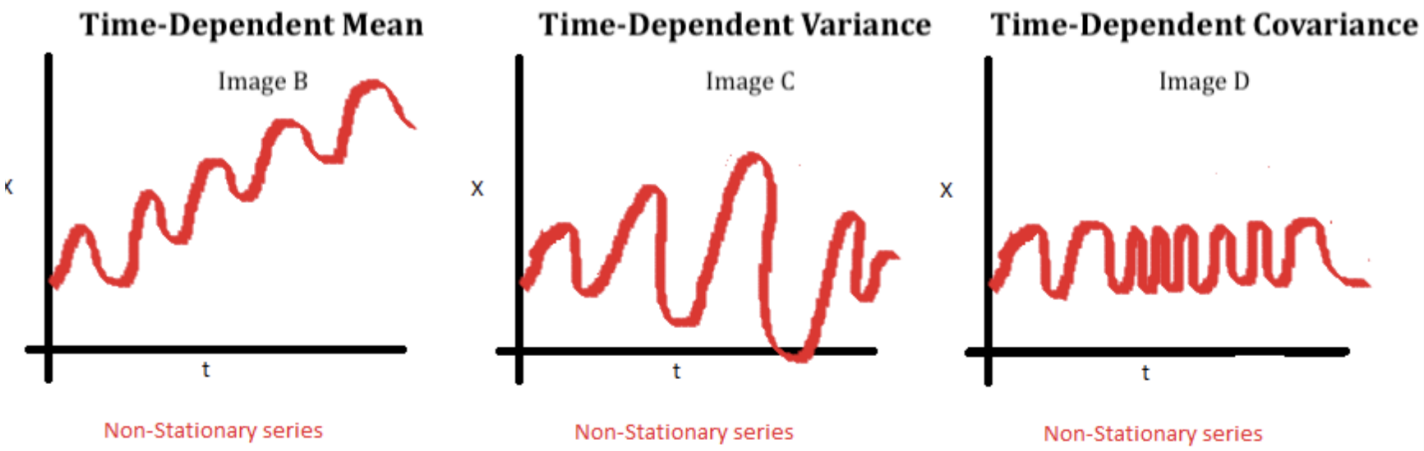
1. **Convert** a non-stationary TS to stationary
2. **Model** our stationary TS and understand its inner workings
3. **Extrapolate** that TS into the future
4. **Reintroduce** seasonality and the rest

4️⃣ Stationarity

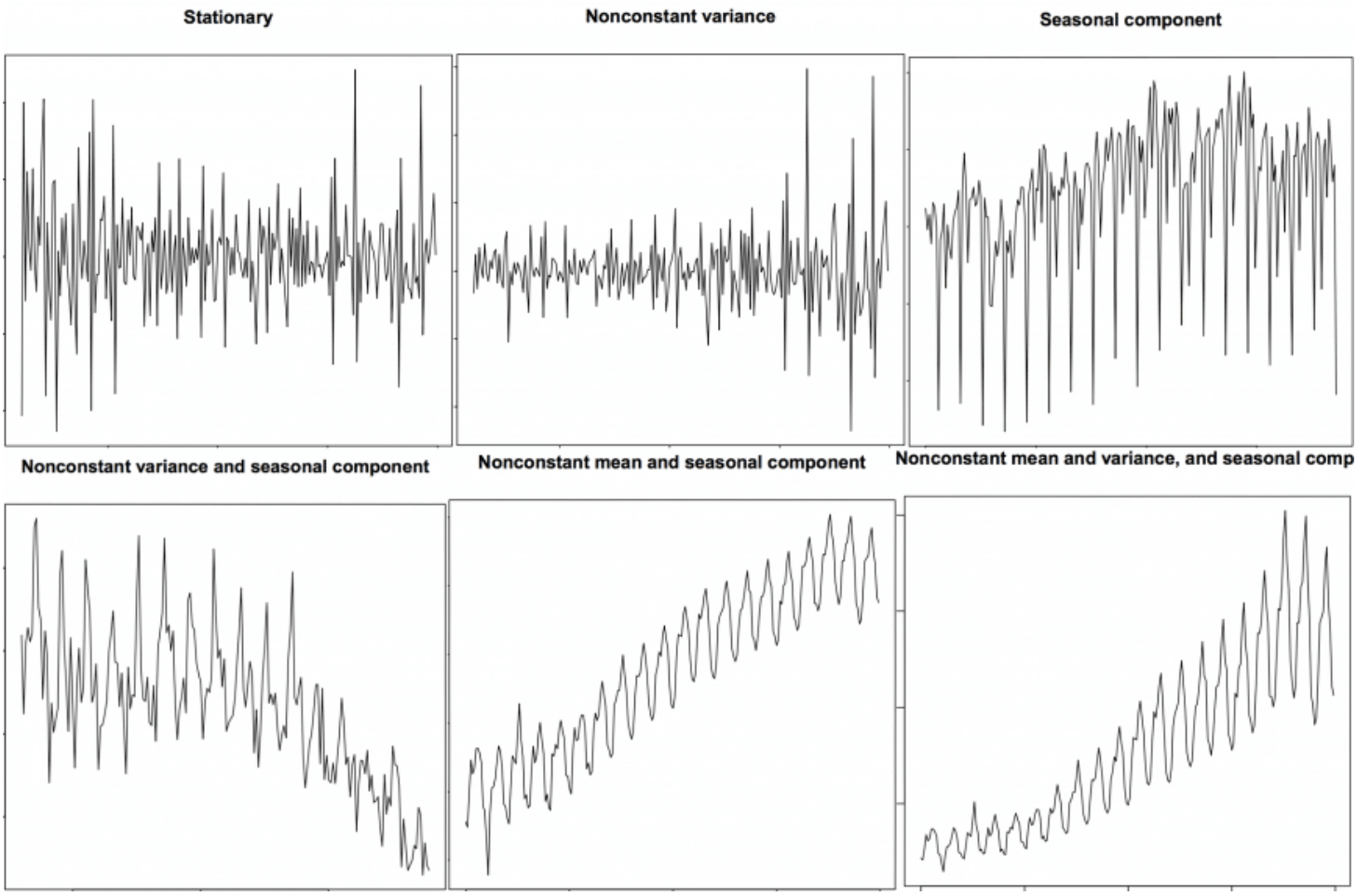
🤔 When is a TS considered **stationary**?

👉 When time does not influence the statistical properties of a TS dataset, such as:

- mean
- variance
- autocorrelation (covariance with its lagged terms)



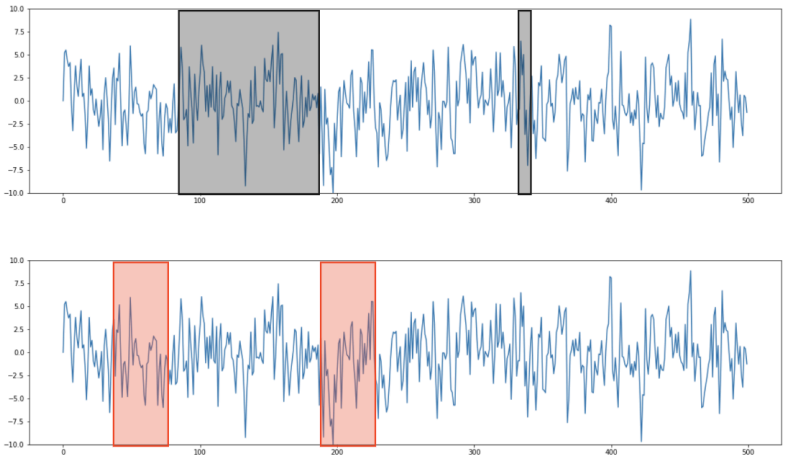
Examples of stationary and non-stationary series



Stationary TS still contain a lot of information!

When analyzing stationary data, distribution is affected only by the size of the time window, **not** the location of a time window

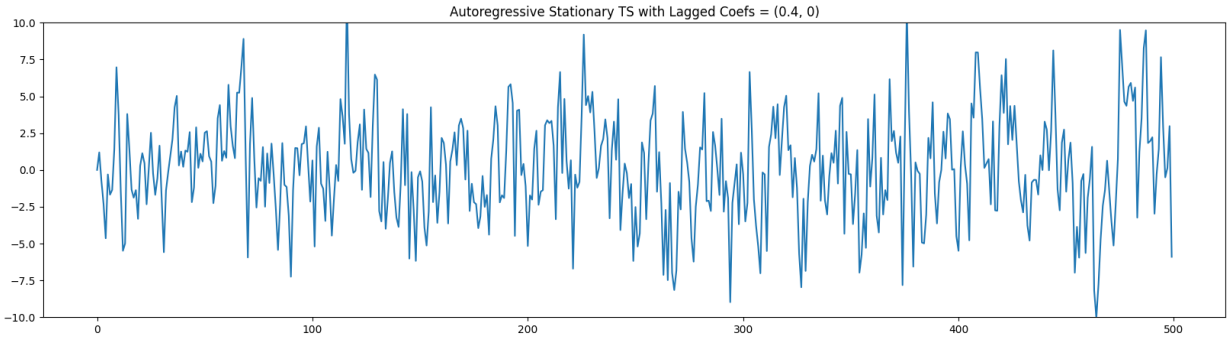
Stationary data



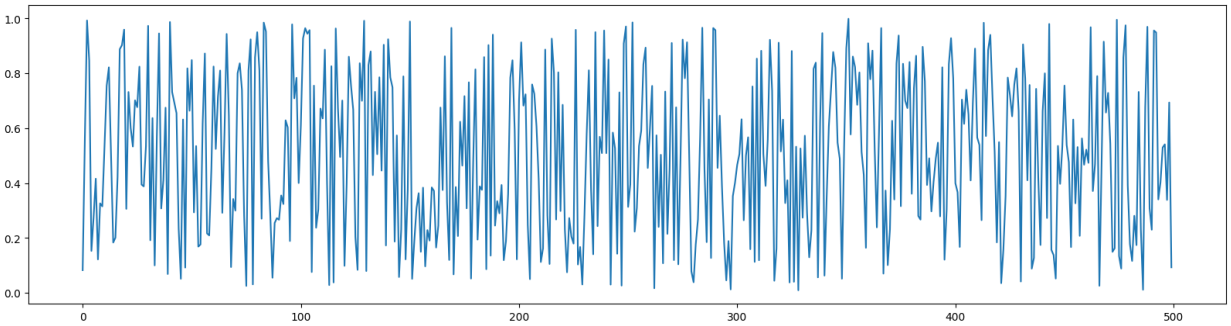
Although stationary, different time window sizes will have different distributions

If stationary and time windows are of equal size- distributions will be equal

In [ ]:
```
# Stationary TS with small autocorrelation but stronger "variance"
plot_stationary_ts(ar1=0.4, sigma=3);
```



In [ ]:
```
# White noise has no information to extract!
plt.figure(figsize=(20,5));
plt.plot(np.arange(500), [scipy.stats.uniform().rvs() for i in np.aran
ge(500)]);
```



How do we test for stationarity?

1. Visually
2. Calculate (mean, variance, and autocorrelation) in various intervals
3. Augmented Dickey Fuller - ADF Test (p-values)

Augmented Dickey Fuller - ADF

ADF tests the following null hypothesis:

- H_0: The series is not-stationary

A `p-value` close to 0 (e.g. p < 0.05) indicates stationarity

In [ ]:
```
from statsmodels.tsa.stattools import adfuller

adfuller(df.value)[1]  # p-value
```

Out[ ]:
```
1.0
```

In [ ]:
```
print('additive resid: ', adfuller(result_add.resid.dropna())[1])
print('multiplicative resid: ', adfuller(result_mul.resid.dropna())[
1])
```

```
additive resid:  0.00028522210547377285
multiplicative resid:  1.7472595795336475e-07
```

Different ways to achieve stationarity

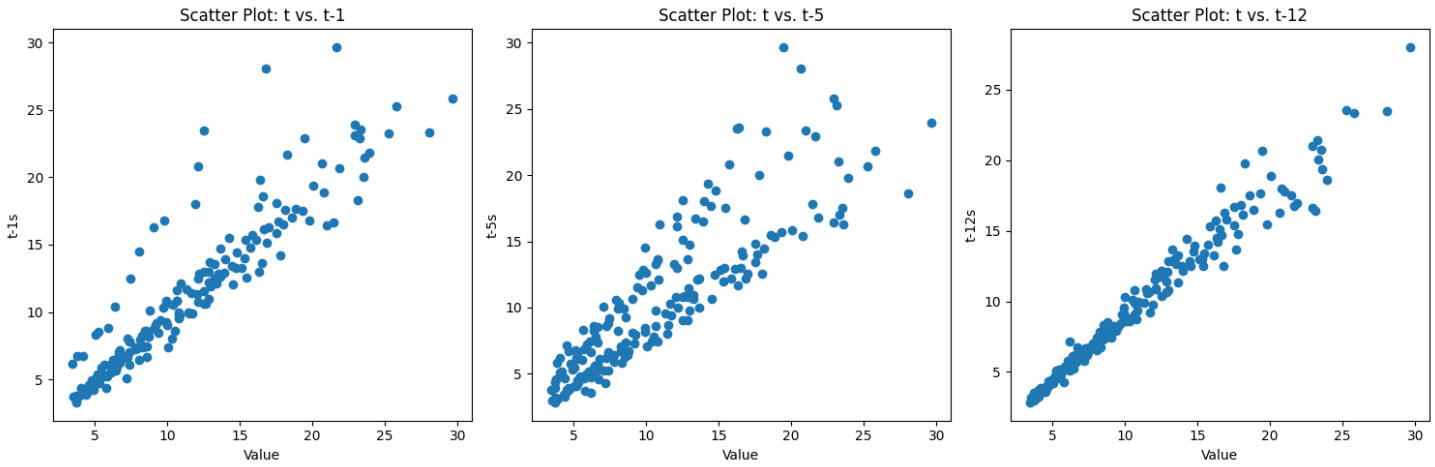
- Decomposition, e.g. $Y = Y_{trend} + Y_{season} + Y_{resid}$ and predict Y_{resid}
- Differencing, e.g. $Y_{diff} = Y_t - Y_{t-1}$ and predict Y_{diff}
- Transformations, e.g. (log, exp, or more advanced variants)

or any combination of these three ways!

5️⃣ Autocorrelation

Let's calculate the correlation between:

- the time series $Y(t)$
- a lagged version of itself $Y(t-i)$



The Autocorrelation Function calculation:

The ACF measures the correlation between a time series and its lagged versions at different time intervals

The ACF at lag "k" is calculated using the following formula:

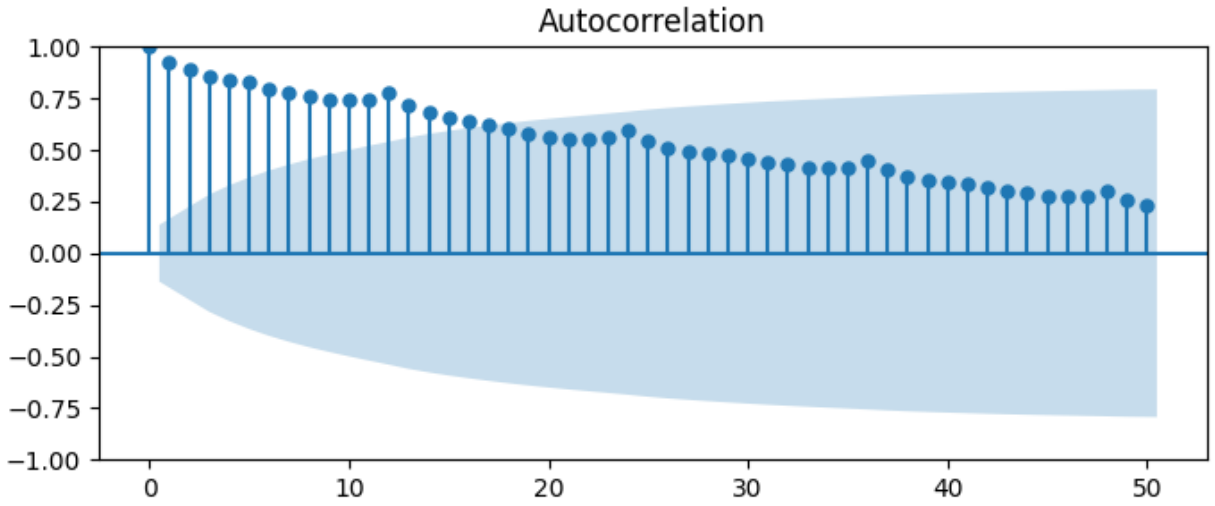
$$\text{ACF}(k) = \frac{\sum_{t=k+1}^{n}(X_t - \bar{X})(X_{t-k} - \bar{X})}{\sum_{t=1}^{n}(X_t - \bar{X})^2}$$

where:

- $ACF(k)$ is the autocorrelation at lag k
- X_t is the value of the time series at time t
- $\bar{X}$ is the mean of the time series.
- X_{t-k} is the value of the time series at time $t-k$, i.e., the value of the series k time steps in the past.
- (n) is the total number of observations in the time series.

Plot each correlation in an autocorrelation graph (ACF) below

In [ ]:
```
fig, ax = plt.subplots(1,1, figsize=(8,3))
from statsmodels.graphics.tsaplots import plot_acf
plot_acf(df.value, lags=50, ax=ax)
plt.show()
```



The blue cone represents a confidence interval (the default is 95%)

Peak inside of cone ➔ not statistically significant

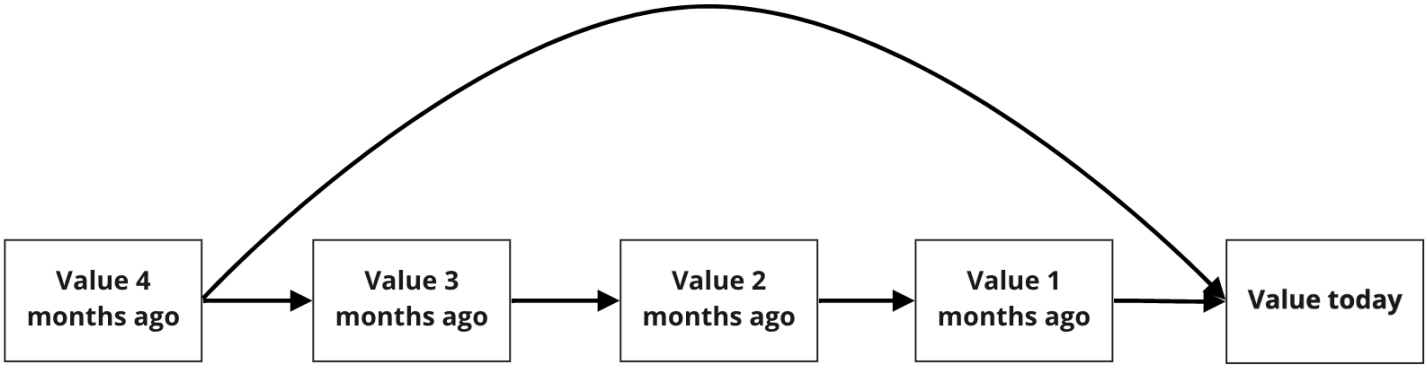
🧐 What do the peaks represent?

Peaks every 12 months indicate the seasonal nature of the data - for example, the value in March 1994 will have a relatively strong correlation with the value in March 1993!

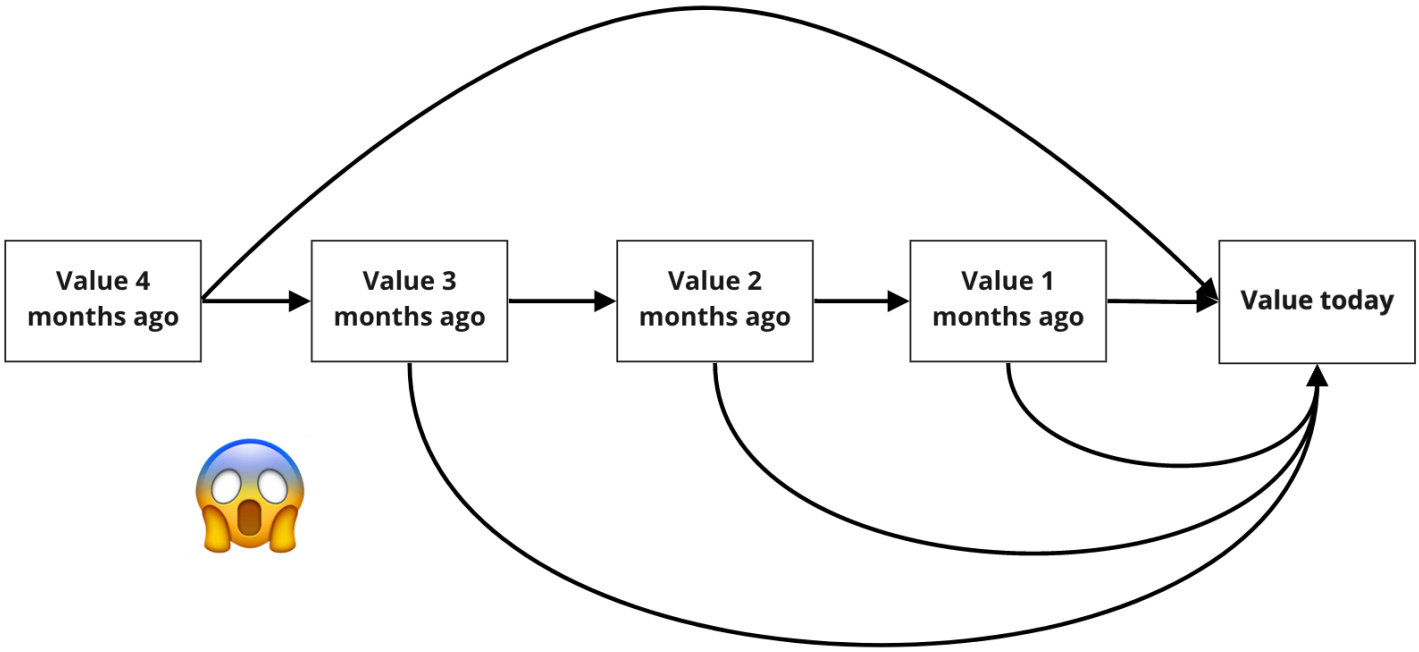
ACF peaks typically decreases slowly: the further back in time you look, the less likely it is that that shift will have an influence

We have to be careful, autocorrelation doesn't just measure the **direct** effect of a lagged point in time...

...it measures the **indirect** effect **as well**!



It can get complicated pretty quickly



🧐 How could we isolate the influence of a specific time lag?

We can try to model our Time Series as an Auto Regressive (AR) process

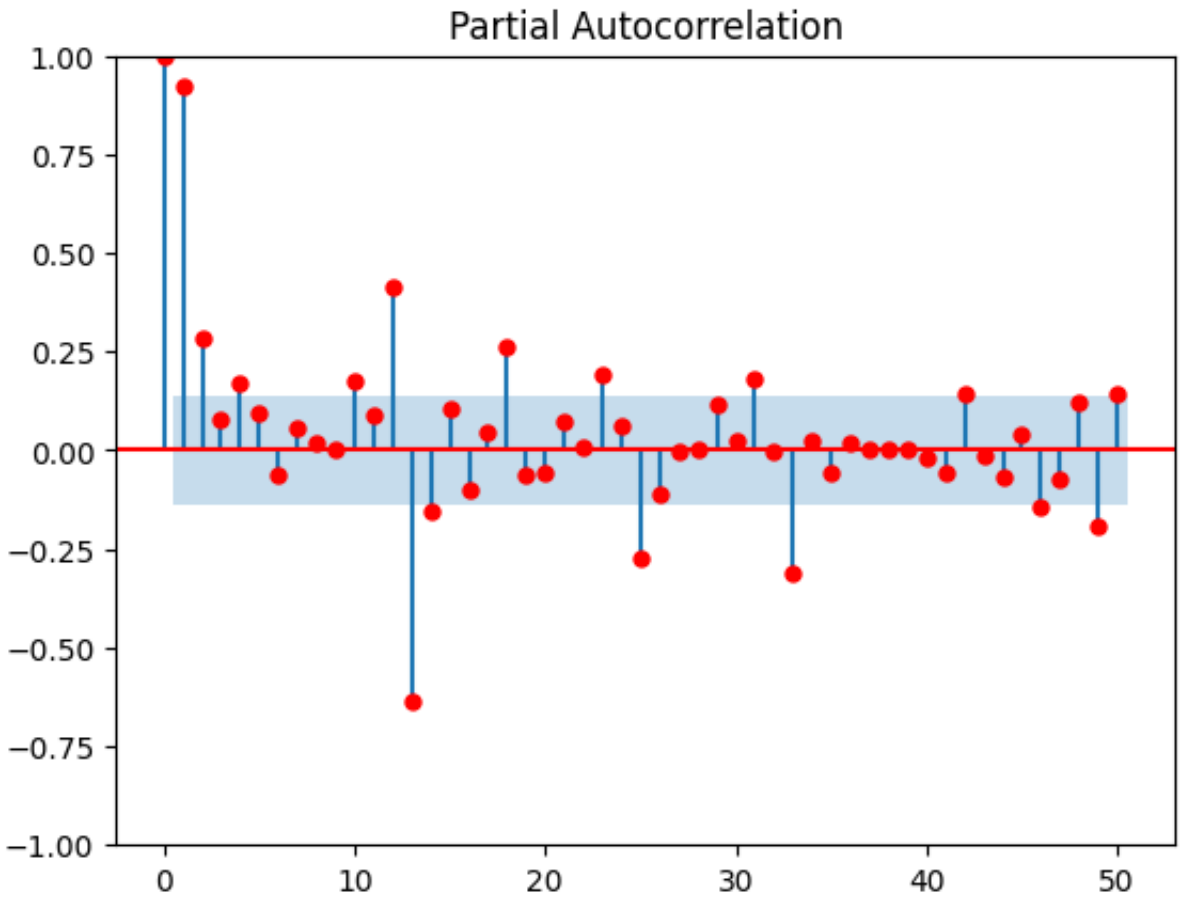
6️⃣ AR (Auto regresssive processes)

$$Y_t = \alpha + \beta_1 Y_{t-1} + \beta_2 Y_{t-2} + \cdots + \beta_p Y_{t-p} + \epsilon_t$$

- This is a **multivariate linear regression** formula
- The **partial** correlation coefficients β_i should give us clues about the **isolated influence of specific time lags**!

We can plot each β_i in a **PACF graph** (Partial ACF)

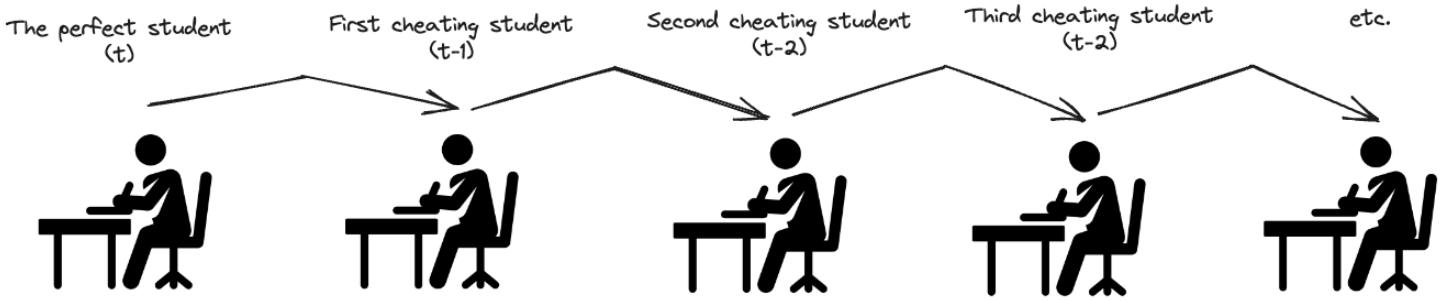
In [ ]:
```
from statsmodels.graphics.tsaplots import plot_pacf
plot_pacf(df.value, lags=50, c='r');
```



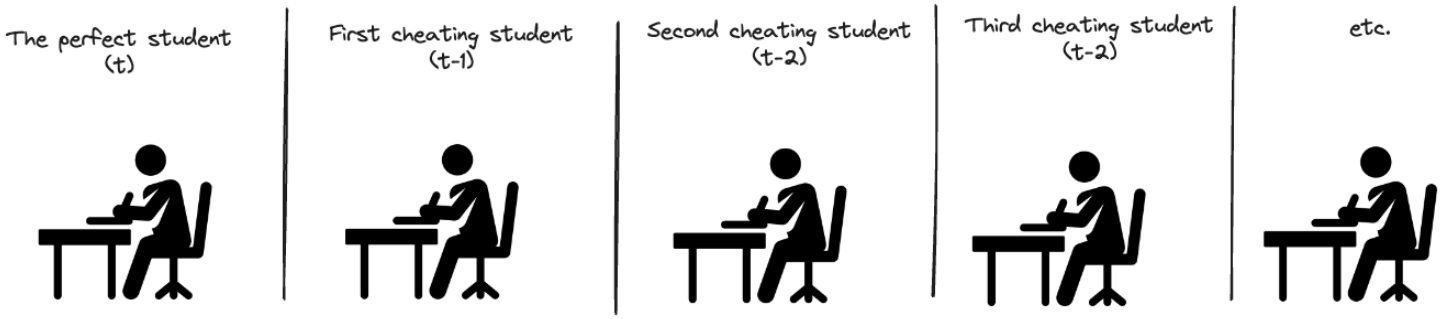
☝️ Typically, we see have a much sharper decline with PACF lag values as opposed to ACF

👍 The PACF plot provides insights into the direct correlation between a specific lagged data point and the current data point, **while removing the influence of intermediate lags!**

Side note: How do magically remove the intermediary correlations?



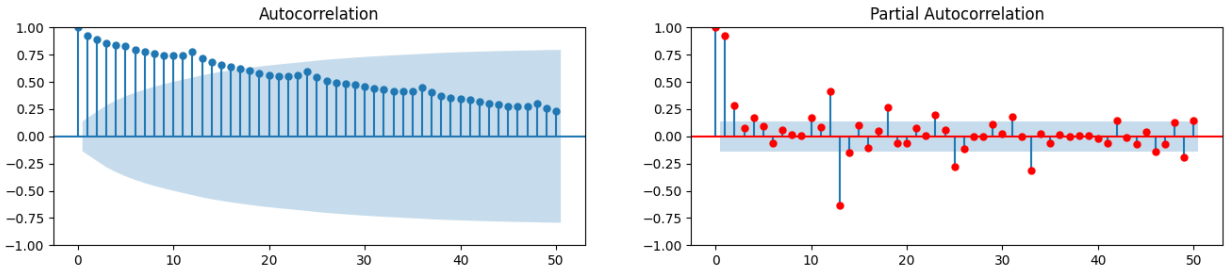
- Imagine a series of students taking a test.
- The student on the left knows 100% of the answers and represents the current time point "t".
- Some students are secretly cheating by looking at their previous neighbor's test papers, but they can only see what the immediate previous student wrote.
- Think of this as the ACF - we'll see a slow decline in their test scores as we go further along (into the past).



- This represents the PACF
- How much does each student **actually** know now that they can't cheat? I.e. how much is each actually correlated with **t** now that the intermediary correlations are taken out?

Advanced: The values for the PACF are calculated by either the Yule-Walker equations (used under the hood in StatsModels) or the Durbin-Levinson Algorithm (uses an iterative approach which is more efficient for longer time series). Both use the values provided by the ACF and assign each a coefficient that is solved for (simultaneously in the YW approach or iteratively in the DL approach).

In [ ]:
```
fig, axes = plt.subplots(1,2, figsize=(16,3))

plot_acf(df.value, lags=50, ax=axes[0]);
plot_pacf(df.value, lags=50, ax=axes[1], color='r');
```



- ACF is simply the correlation of the series with itself
  - slow exponential decrease
  - if $X(t)$ is always correlated with $X(t-1)$, then it is also correlated with $X(t-2)$
- PACF is even more informative
  - it removes intermediary correlations

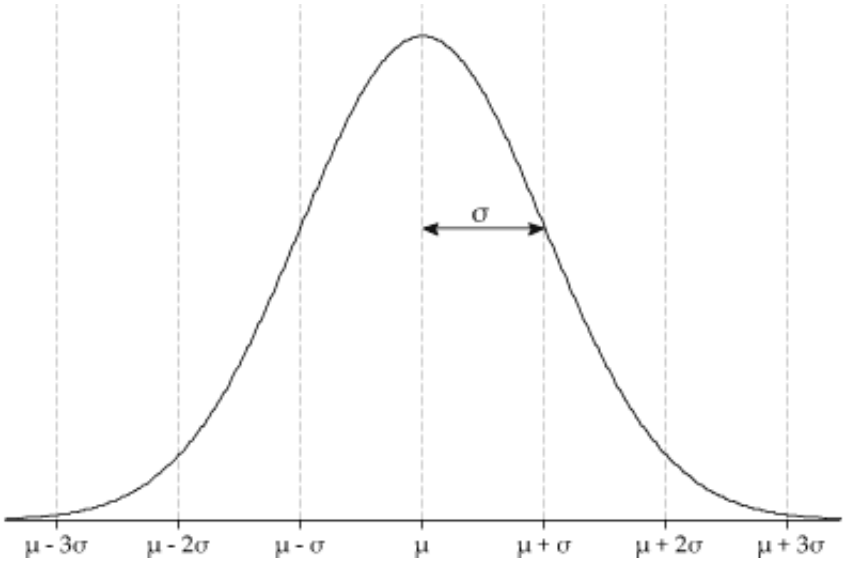
👉 Annual seasonality is confirmed (peaks at `lag = 12`)

What does a pure $AR(p)$ process look like? (intuition)

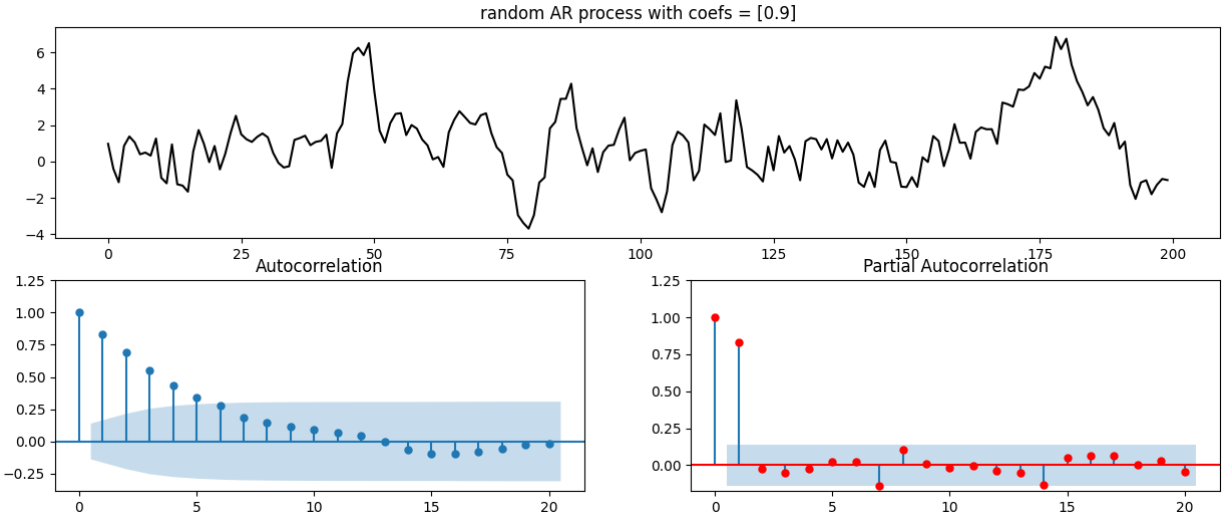
$$Y_t = \alpha + \beta_1 Y_{t-1} + \beta_2 Y_{t-2} + \cdots + \beta_p Y_{t-p} + \epsilon_t$$

$$\text{let's assume random noise } \epsilon \overset{\text{iid}}{\sim} \mathcal{N}(0, \sigma)$$

A note on random noise:



Let's "synthetize" a first order AR(1) process: $Y_t = 0 + 0.9Y_{t-1} + \epsilon_t$ with $Y_0 = Y_1 = 0$



☝ AR process "looks" random but is not! Autoregression makes it "**stickier**" than white noise

☝ We can "capture" information about the "order" p of AR process in the $PACF$ plot by counting the **number of non-null coefs in red** (always excludes first coef of order 0)

💡 Imagine global warming as a simple weather pattern that keeps getting warmer year after year. In a simplified AR process for global warming, the temperature in a given century would depend on the temperature in the previous century. So, if this century is warmer than the last, next century would likely be even warmer, and so on. It's like a 'warming trend' that continues into the future, just based on the recent history of temperatures.

For example, let's consider

$$\Delta\text{GlobalTemp}$$

Global temperature anomalies change from century to century due to natural and anthropogenic factors. Let's imagine we utilize an AR(1) process to model it:

- $\Delta\text{GlobalTemp}(century21) = 0.5 \times \Delta\text{GlobalTemp}(century20) + \epsilon_{\text{century 21}}$
- $\Delta\text{GlobalTemp}(century22) = 0.5 \times \Delta\text{GlobalTemp}(century21) + \epsilon_{\text{century 22}}$
- $\Delta\text{GlobalTemp}(century23) = 0.5 \times \Delta\text{GlobalTemp}(century22) + \epsilon_{\text{century 23}}$

Now, let's imagine a significant increase in greenhouse gas emissions during the 20th and 21st centuries:

- $\epsilon_{century20} = +0.5^\circ C$ (we get an unexpected warming of 0.5 degrees due to emissions)
- $\epsilon_{century21} = +1^\circ C$ (we get another unexpected warming of 1 degree due to emissions)
- $\epsilon_{century22} = 0^\circ C$ (emissions stop so there are no more "shocks")
- $\epsilon_{century23} = 0^\circ C$

ΔGlobalTemperature(century 21) = 0.5 x 0.5 + 1 = +1.25°C 📈 --> An initial temperature rise due to increased greenhouse gases.
ΔGlobalTemperature(century 22) = 0.5 x 1.25 + 0 = +0.625°C 📈 --> The warming effect persists as greenhouse gases continue to accumulate.
ΔGlobalTemperature(century 23) = 0.5 x 0.625 + 0 = +0.31°C 📈
ΔGlobalTemperature(century 24) = 0.5 x 0.31 + 0 = +0.15°C 📈
...

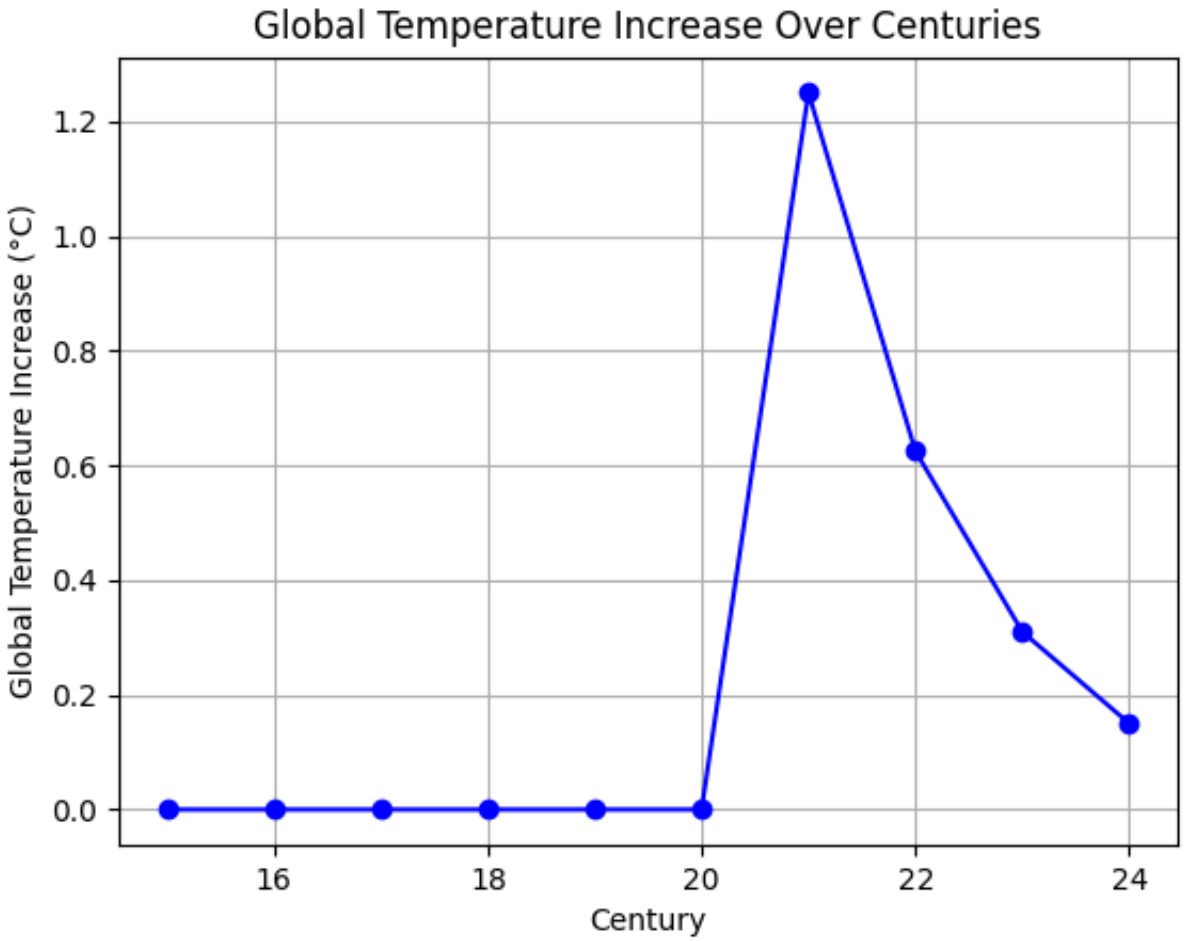
☝️ Unforeseen "shocks" like such as the increase in greenhouse gases and temp in the 20th and 21st century, will have a long-lasting impact. Even in century 24, we still observe a small effect from the original emissions increase of +0.15°C.
👀 N.B. This is a **very** simplistic model - climate modelling involves a lot more variables than those shown here!

📺 5-min video explanation (https://www.youtube.com/watch?v=AN0a58F6cxA)

In [ ]:
```
tp
```

Out[ ]:

Century	GlobalTempIncrease	Shock
15	0.000	0.0
16	0.000	0.0
17	0.000	0.0
18	0.000	0.0
19	0.000	0.0
20	0.000	0.5
21	1.250	1.0
22	0.625	0.0
23	0.310	0.0
24	0.150	0.0



7️⃣ MA (Moving Average processes)

Our previous model was a little simplistic - for many time series, shocks do not propagate far into the future. So another type of model that we can use for a stationary time series is...

- Modeling a **linear combination of consecutive random errors/ shocks** ϵ ?

(MA) process:

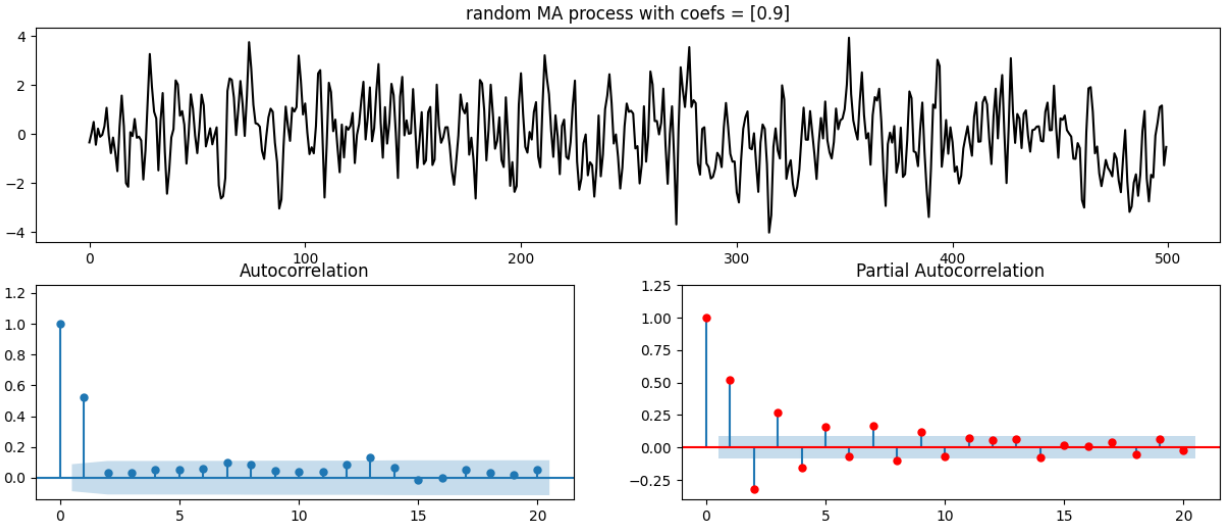
$$Y_t = \alpha + \epsilon_t + \phi_1 \epsilon_{t-1} + \phi_2 \epsilon_{t-2} + \cdots + \phi_q \epsilon_{t-q}$$

with $\epsilon \overset{\text{iid}}{\sim} (0, \sigma)$

- Does not depend on past Y values
- Tends to stay around average value α
- **Any "random shock" will only have a very limited time effect of duration `q`** (contrary to AR processes)

Let's plot a first order random MA(1) process

$Y_t = 0 + \epsilon_t + 0.9\epsilon_{t-1}$ with with $\epsilon \overset{\text{iid}}{\sim} (0, \sigma)$



☝ MA process "look" random but are actually not!

☝ We can deduce information about the "order" q of the MA process by counting the **number of non-null coefs in our** ACF

💡 MA process in real life?

Central Heating System Δ°C:

Scenario: A central heating system responds to environmental changes every minute. The central heating system manufacturer has a **guarantee** that it will adjust to a temperature shock within 2 minutes time!

$$\Delta^\circ C_1 = \epsilon_1 + 0.5\epsilon_0$$
$$\Delta^\circ C_2 = \epsilon_2 + 0.5\epsilon_1$$
$$\Delta^\circ C_3 = \epsilon_3 + 0.5\epsilon_2$$

Imagine someone opens a door at minute 1 and let's in a cold wind! Let's look at our errors:

ϵ_{minute_0} = 0°C
ϵ_{minute_1} = -5°C
ϵ_{minute_2} = 0°C
ϵ_{minute_3} = 0°C

- Minute 0: $\Delta^\circ C_1$ = 0.5 x 0 + 0 = 0 (we assume previous errors have been 0)
- Minute 1: $\Delta^\circ C_2$ = 0.5 x 0 -5 = -5 (our cold breeze enters 🥶)
- Minute 2: $\Delta^\circ C_3$ = 0.5 x -5 + 0 = -2.5 (our system responds to the shock and begins heating the room)
- Minute 3: $\Delta^\circ C_4$ = 0.5 x 0 + 0 = 0 (our room is back to its regular temp -> no further propagation)

📺 Watch this 5min amazing example: Lemonade demand 🍋 (https://www.youtube.com/watch?v=lUhtcP2SUsg)

8️⃣ ARMA (Auto Regressive Moving Average)

Most real world series are actually a combination our two models!

$ARMA(p,\ q)$

- as a linear combination of p lags of Y
- *plus* a linear combination of q lagged errors

$$AR: Y_t = \alpha + \beta_1 Y_{t-1} + \beta_2 Y_{t-2} + \cdots + \beta_p Y_{t-p} + \epsilon_t$$

$$MA: Y_t = \alpha + \phi_1 \epsilon_{t-1} + \phi_2 \epsilon_{t-2} + \cdots + \phi_q \epsilon_{t-q} + \epsilon_t$$

$$ARMA \rightarrow Y = \alpha + \beta_1 Y_{t-1} + \cdots + \beta_p Y_{t-p} + \phi_1 \epsilon_{t-1} + \cdots + \phi_q \epsilon_{t-q}$$

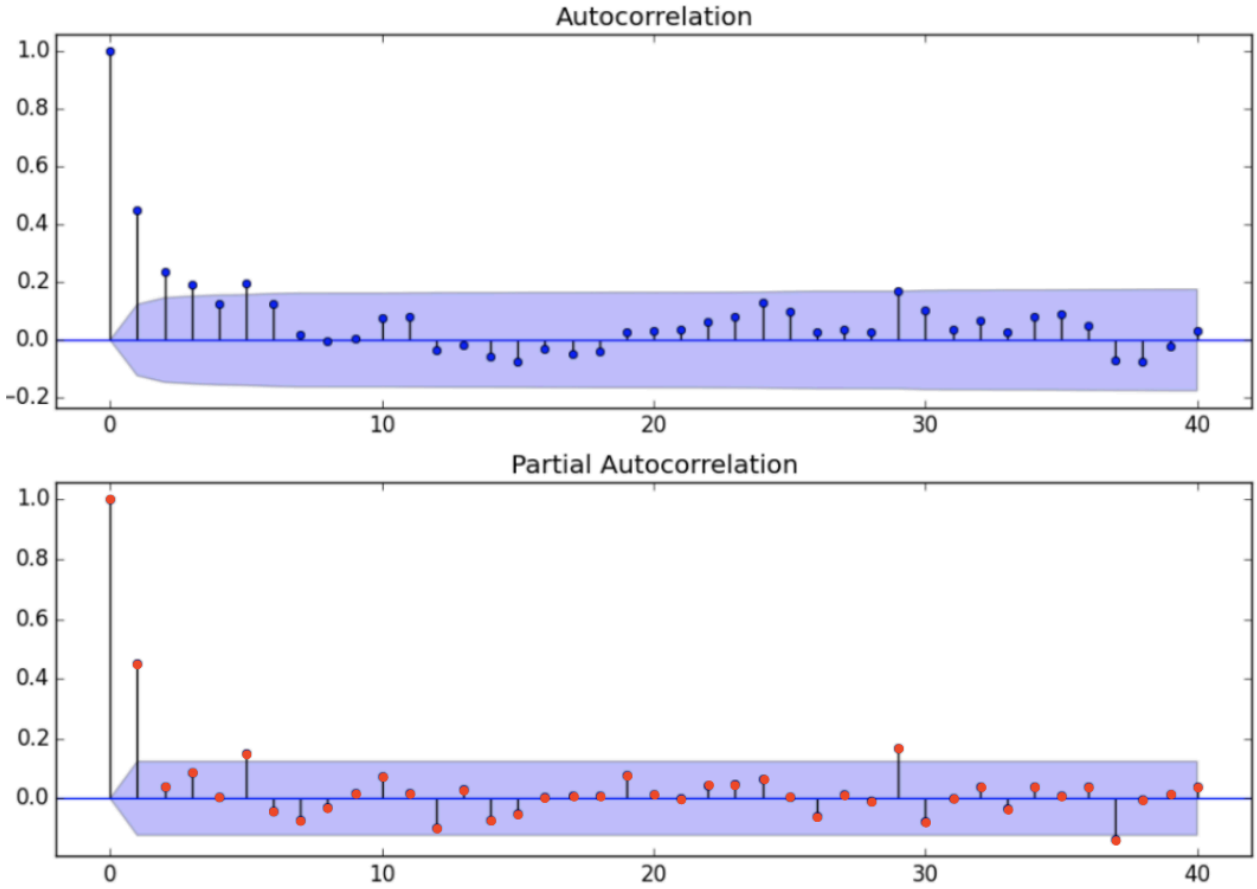
Hyperparameters p & q

p expresses how many lags we want to incorporate into AR

q expresses how many lags we want to incorporate into MA

🧐 How do we know how many lags to use?

Count the number of lags before the values drop below the confidence levels Note that the first lag is ignored, as it represents AC/PAC between y_t and itself



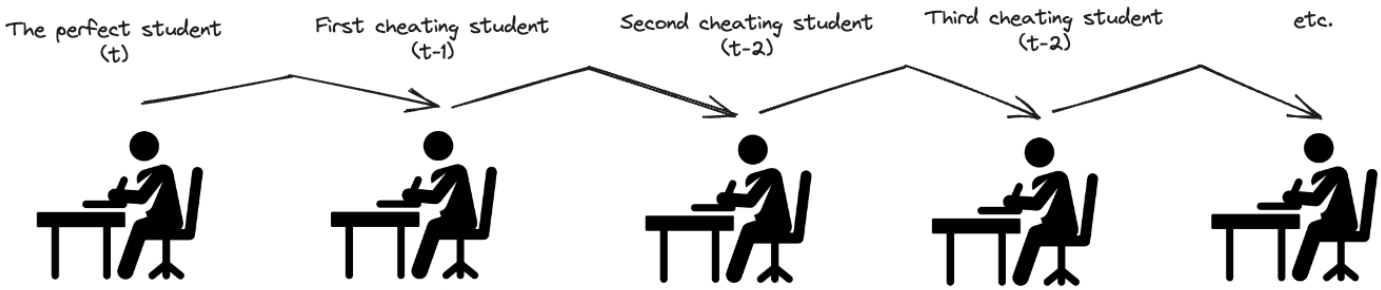
We use the $PACF$ plot to calculate the value for $p = 1$

We use the ACF plot to calculate the value for $q = 3$

$ARMA(1,\ 3)$

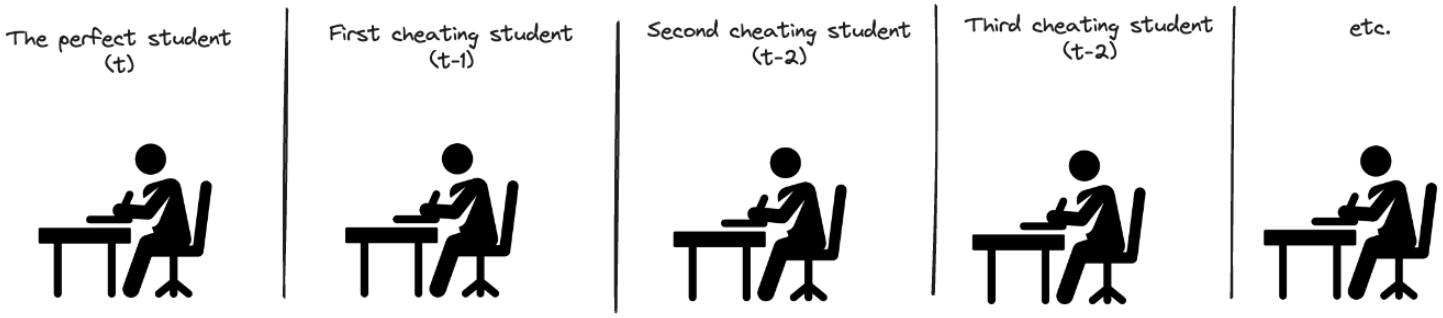
Why pick our q value from our ACF plot?

- Because MA deals with **how our errors propagate**



Why pick our p value from our PACF plot?

- Because AR deals with which terms are **actually** useful to consider to calculate the value at t



9️⃣ ARIMA

(Auto Regressive Integrated Moving Average)

One of the main assumptions for Time Series modeling is that the data must be **stationary**

One way we can achieve that is by doing our decomposition as we've already demonstrated, but we can apply **differencing** to our dataset to turn non-stationary data into stationary data 🎉

Initial series: annual GDP

$$Y_t$$

First-order diff: GDP "growth"

$$Y_t^{(1)} = Y_t - Y_{t-1}$$

Second-order diff: GDP "acceleration"

$$Y_t^{(2)} = Y_t^{(1)} - Y_{t-1}^{(1)}$$

...keep differencing until our data gets stationary...

💻 Use `df.diff` in pandas

In [ ]:
```
non_differenced_data = pd.Series([1, 4, 9, 16, 25, 36])
differenced_1 = non_differenced_data.diff()
print(differenced_1)
```

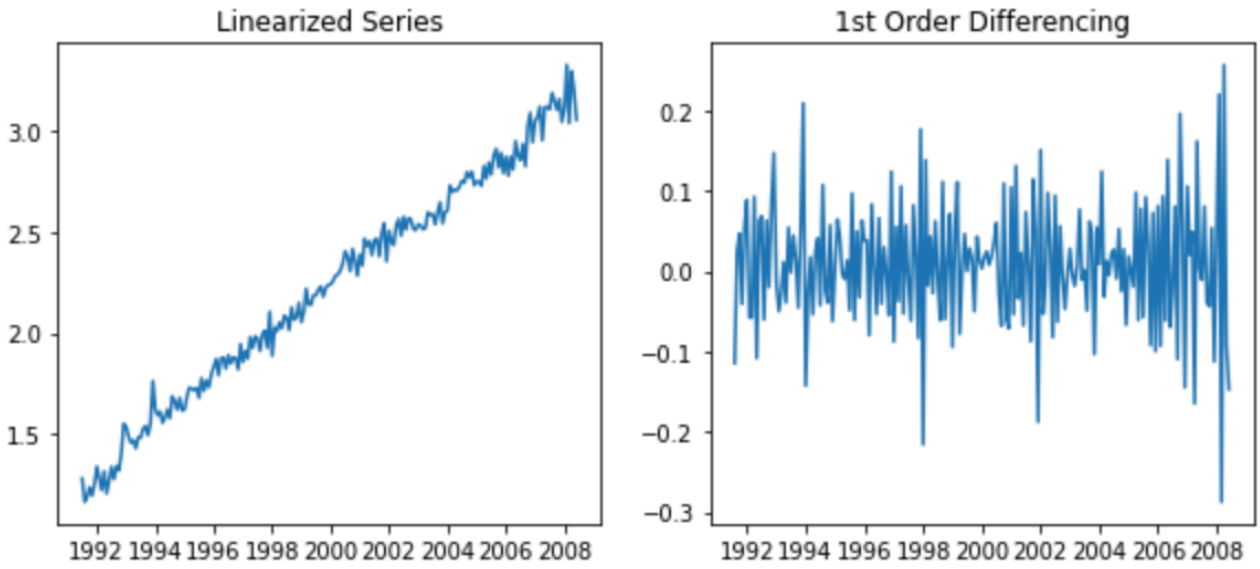
```
0     NaN
1     3.0
2     5.0
3     7.0
4     9.0
5    11.0
dtype: float64
```

We should keep differencing until our data gets stationary

In [ ]:
```
print(differenced_1.diff())
```

```
0    NaN
1    NaN
2    2.0
3    2.0
4    2.0
5    2.0
dtype: float64
```

For instance, here is some non-stationary data that we smoothed out using differencing (removed the function of time in our series) 💪



ARIMA

$$Y_t^{(d)} = \alpha + \beta_1 Y_{t-1}^{(d)} + \cdots + \beta_p Y_{t-p}^{(d)} + \phi_1 \epsilon_{t-1} + \cdots + \phi_q \epsilon_{t-q}$$

AutoRegressive Integrated MovingAverage:

I: Uses differencing to achieve stationarity (i.e. try to predict Y_{diff})

AR: Linear combination between Y and lagged terms Y_{t-i}

MA: Linear combination between Y and residual errors from the AR model above

The AR process is simply a linear combination of p lags

- a one time "shock" will propagate far into the future
- not necessarily stationary

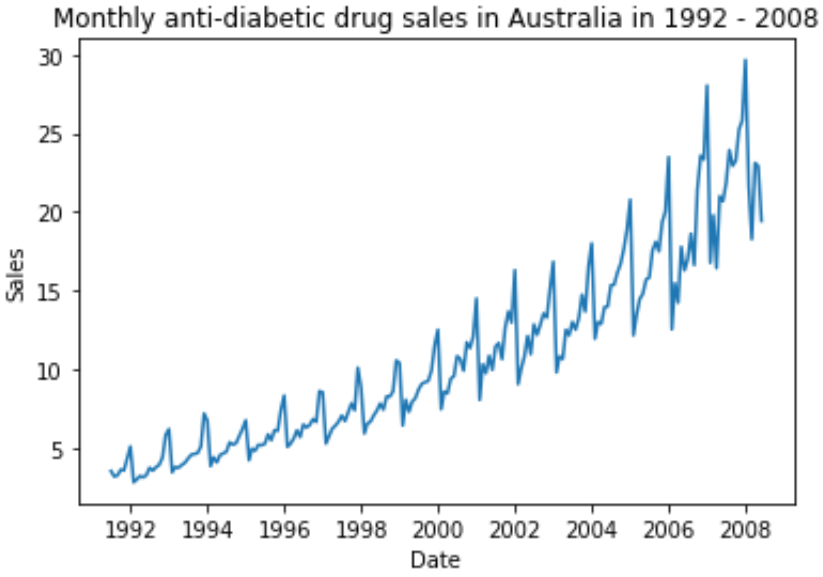
The MA process is a linear combination of q "random shocks"

- any "shock" will have a limited time effect
- always stationary

The I: differencing predicts $Y^{(d)}$ instead of Y

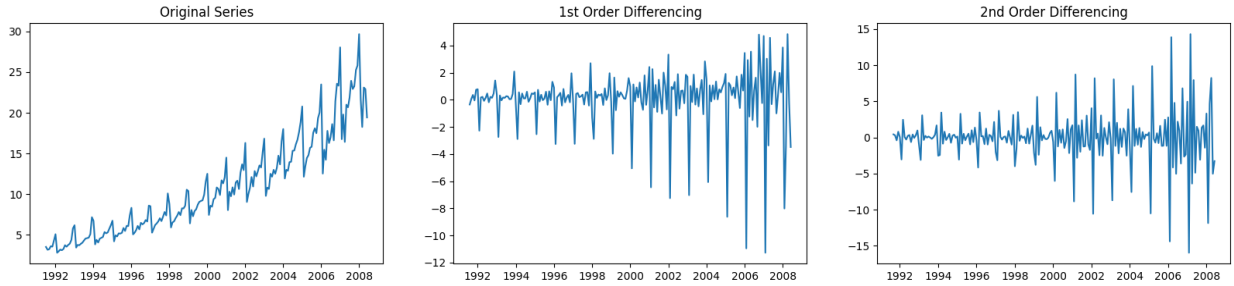
- Differentiate until stationary
- 💡 *differencing* tends to turn AR processes into MA ones
- A GDP *growth* "shock" may not propagate long term but a "GDP shock" will !

Let's get back to our initial dataset and model it with ARIMA



🤔 Can we make it stationary it via differencing?

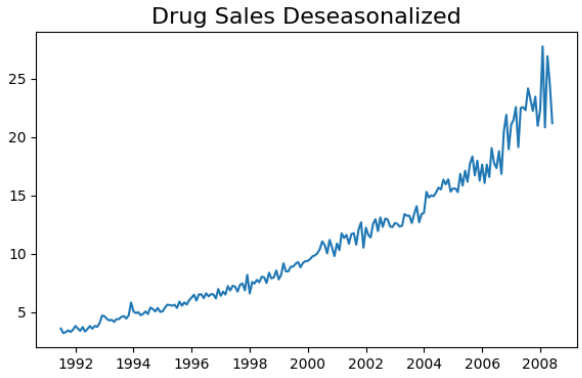
In [ ]:
```
zero_diff = df.value
first_order_diff = df.value.diff(1)
second_order_diff = df.value.diff(1).diff(1)


fig, (ax1, ax2, ax3) = plt.subplots(1, 3, figsize=(20,4))
ax1.plot(zero_diff); ax1.set_title('Original Series')
ax2.plot(first_order_diff); ax2.set_title('1st Order Differencing')
ax3.plot(second_order_diff); ax3.set_title('2nd Order Differencing');
```

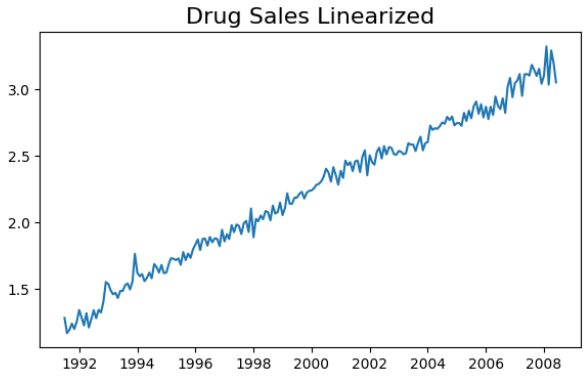


❌ Not stationary: seasonality is still obviously present
❌ **Cannot apply ARIMA** directly (we will see SARIMA later)

👉 We need to *de-seasonalize* our Time Series first, using our *decomposition* tool

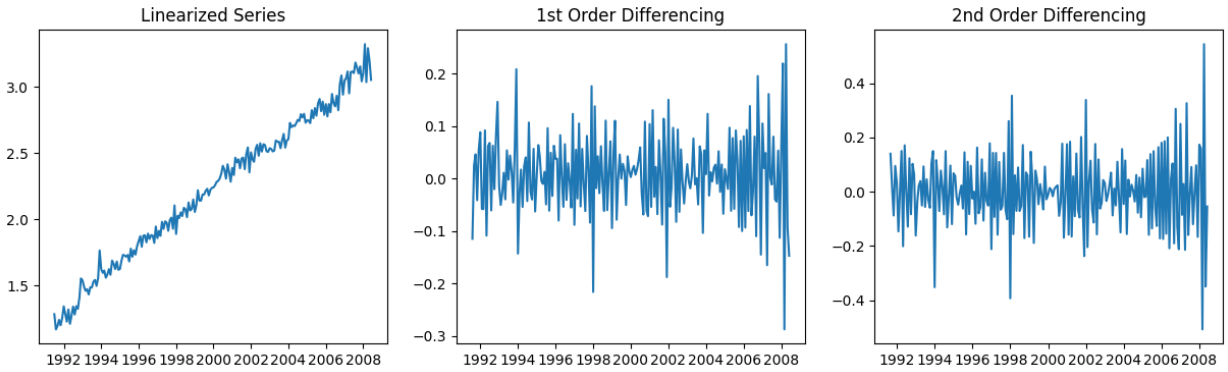
In [ ]:
```
# Let's remove seasons
df['deseasonalized'] = df.value.values/result_mul.seasonal

plt.figure(figsize=(15,4)); plt.subplot(1,2,1); plt.plot(df.deseasonal
ized);
plt.title('Drug Sales Deseasonalized', fontsize=16);

# Also remove exponential trend
df['linearized'] = np.log(df['deseasonalized'])

plt.subplot(1,2,2); plt.plot(df['linearized'])
plt.title('Drug Sales Linearized', fontsize=16);
```





Let's re-run our *differencing* analysis on this new linearized Time Series

In [ ]:
```
# Let's difference this and look at the ACFs
fig, axes = plt.subplots(1, 3,figsize=(15,4))

axes[0].plot(df['linearized']); axes[0].set_title('Linearized Series')
# 1st Differencing
y_diff = df['linearized'].diff().dropna()
axes[1].plot(y_diff); axes[1].set_title('1st Order Differencing')

# 2nd Differencing
y_diff_diff = df['linearized'].diff().diff().dropna()
axes[2].plot(y_diff_diff); axes[2].set_title('2nd Order Differencin
g');
```



In [ ]:
```
# check with ADF Test for stationarity
print('p-value zero-diff: ', adfuller(df['linearized'])[1])
print('p-value first-diff: ', adfuller(df['linearized'].diff().dropn
a())[1])
print('p-value second-diff: ', adfuller(df['linearized'].diff().dif
f().dropna())[1])
```

```
p-value zero-diff:  0.7134623265852306
p-value first-diff:  1.0092820652732304e-09
p-value second-diff:  1.318178239864844e-12
```

No need to over-difference

☝ We should select d=1

Usually, 1-diff is enough! If not, you might have exponential behavior and might want to use a log transformation instead! Alternatively, we can *de-trend* and work with the *de-trended* series without differencing.

Another way to determine how many orders of differencing to apply

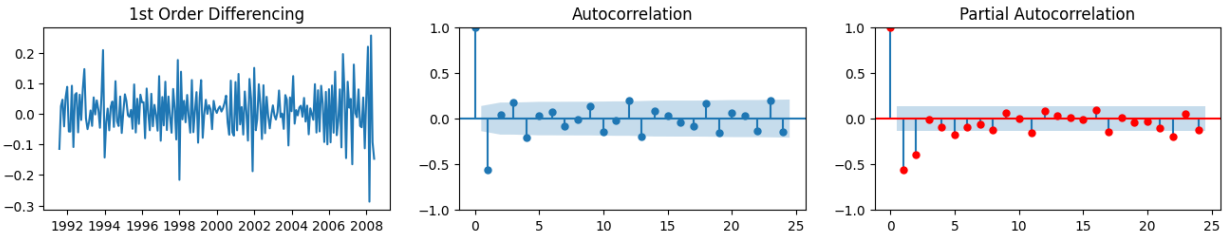
In [ ]:
```
# automatically estimate differencing term
from pmdarima.arima.utils import ndiffs
ndiffs(df['linearized'])
```

Out[ ]: 1

Finding the hyperparameters p&q

- AR(p) ~ number of lags beyond which the $PACF$ plot of $Y^{(d)}$ cuts off
- MA(q) ~ number of lags beyond which the ACF plot of $Y^{(d)}$ cuts off

In [ ]:
```
# ACF / PACF analysis of y_diff linearized
fig, axes = plt.subplots(1,3, figsize=(16,2.5))
axes[0].plot(y_diff); axes[0].set_title('1st Order Differencing')
plot_acf(y_diff, ax=axes[1]);
plot_pacf(y_diff, ax=axes[2], c='r');
```



☝ PACF ➔ `p` = 2
☝ ACF ➔ `q` = 1

$ARIMA(p=2,\ d=1,\ q=1)$

☝ More nuanced rules can be applied to hyperparameter tuning

Use the **Box-Jenkins method**

- IMPORTANT: more than one model might explain your data.

📚 More details can be found here (http://people.duke.edu/~rnau/arimrule.htm) and here (https://people.duke.edu/~rnau/411arim3.htm)

- When in doubt, choose the simpler model!

In [ ]:
```
# from statsmodels.tsa.arima_model import ARIMA #statsmodels 0.11
from statsmodels.tsa.arima.model import ARIMA  #statsmodels 0.12+

arima = ARIMA(df['linearized'], order=(2, 1, 1), trend='t')
arima = arima.fit()
```

In [ ]:
```
arima.summary()
```

Out[ ]:

SARIMAX Results

Dep. Variable:	linearized	**No. Observations:**	204
Model:	ARIMA(2, 1, 1)	**Log Likelihood**	291.326
Date:	Mon, 31 Jul 2023	**AIC**	-572.651
Time:	17:27:06	**BIC**	-556.085
Sample:	07-01-1991	**HQIC**	-565.949
	- 06-01-2008		
Covariance Type:	opg		

	coef	std err	z	P>|z|	[0.025	0.975]
x1	0.0097	0.001	11.608	0.000	0.008	0.011
ar.L1	-0.1419	0.095	-1.491	0.136	-0.328	0.045
ar.L2	0.0290	0.088	0.329	0.742	-0.144	0.202
ma.L1	-0.7891	0.078	-10.162	0.000	-0.941	-0.637
sigma2	0.0033	0.000	12.308	0.000	0.003	0.004

Ljung-Box (L1) (Q):	0.00	**Jarque-Bera (JB):**	26.46
Prob(Q):	0.99	**Prob(JB):**	0.00
Heteroskedasticity (H):	1.33	**Skew:**	0.47
Prob(H) (two-sided):	0.25	**Kurtosis:**	4.50

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

Performance Metrics - Akaike Information Criterion

$$AIC = 2k - 2\log(Likelihood)$$

k = total number of parameters used
e.g. with $p = 2, i = 1, q = 3 (k = 6)$

Acts as penalizing for overfitting

Increases as the model gets better at predicting

We can also use **`auto_arima`** to GridSearch the hyperparameters p, d, q 🥰

In [ ]:
```
import pmdarima as pm
smodel = pm.auto_arima(df['linearized'],
                       start_p=1, max_p=2,
                       start_q=1, max_q=2,
                       trend='t',
                       seasonal=False,
                       trace=True)
```

```
Performing stepwise search to minimize aic
 ARIMA(1,1,1)(0,0,0)[0] intercept   : AIC=-555.440, Time=0.06 sec
 ARIMA(0,1,0)(0,0,0)[0] intercept   : AIC=-453.201, Time=0.02 sec
 ARIMA(1,1,0)(0,0,0)[0] intercept   : AIC=-527.228, Time=0.03 sec
 ARIMA(0,1,1)(0,0,0)[0] intercept   : AIC=-533.804, Time=0.05 sec
 ARIMA(0,1,0)(0,0,0)[0]             : AIC=-453.201, Time=0.02 sec
 ARIMA(2,1,1)(0,0,0)[0] intercept   : AIC=-560.953, Time=0.04 sec
 ARIMA(2,1,0)(0,0,0)[0] intercept   : AIC=-562.744, Time=0.02 sec
 ARIMA(2,1,0)(0,0,0)[0]             : AIC=-562.744, Time=0.02 sec

Best model:  ARIMA(2,1,0)(0,0,0)[0]
Total fit time: 0.260 seconds
```

⚠️ Several models have similar AIC values around -560.

Any of them might be the "true" underlying process

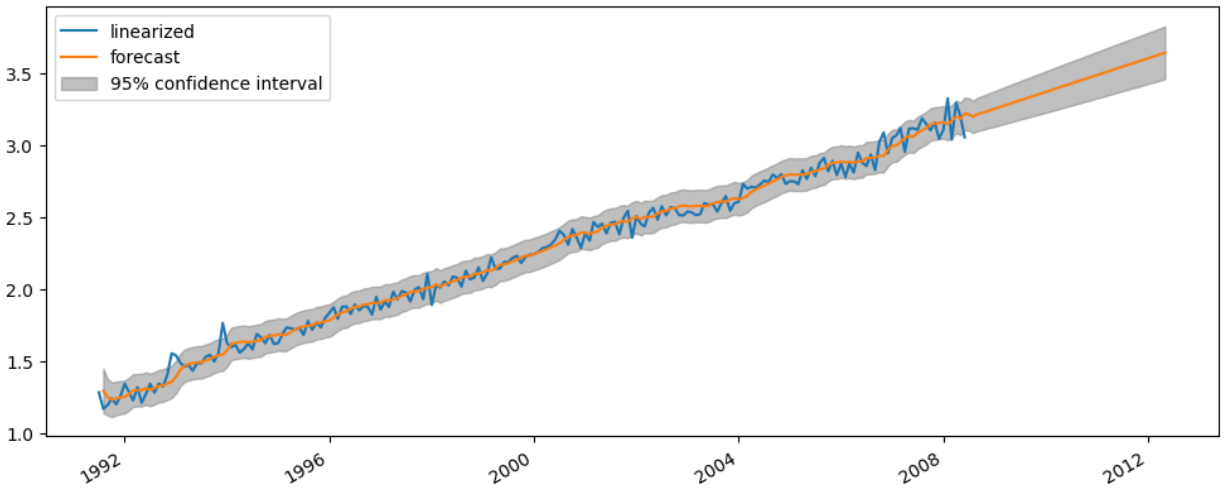
👉 When in doubt, pick the simplest model

$ARIMA(p = 2,\ d = 1,\ q = 0)$

Evaluate Performance

Use `plot_predict()` to glance at our predictions

In [ ]:
```
from statsmodels.graphics.tsaplots import plot_predict

fig, axs = plt.subplots(1, 1, figsize=(12, 5))
axs.plot(df['linearized'], label='linearized')
plot_predict(arima, start=1, end=250, ax=axs);
```

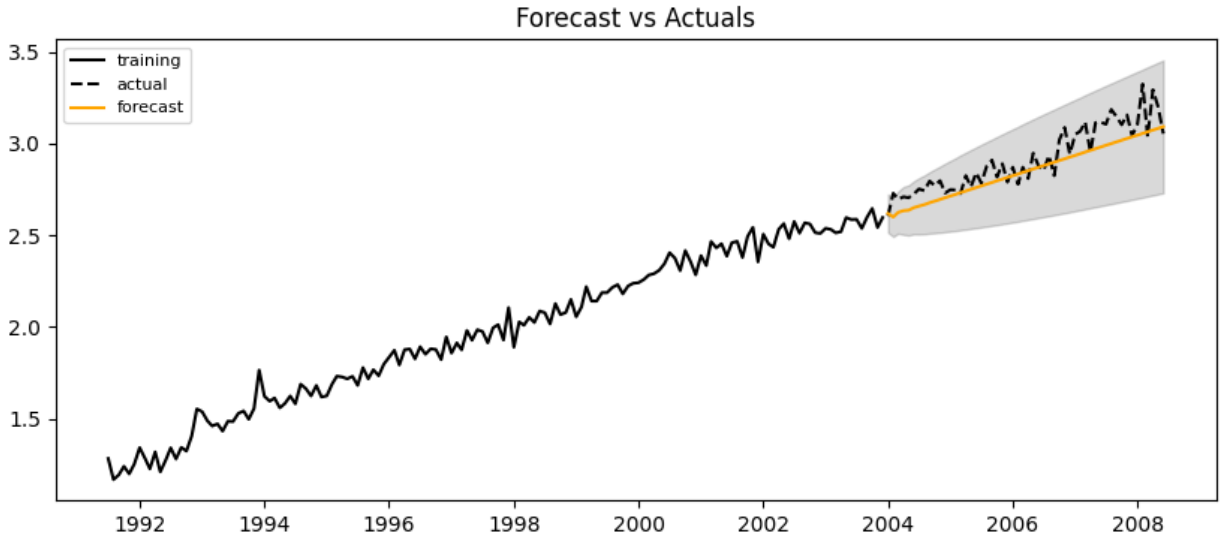


Use `.forecast()` to simply access forecasts
Use `.get_forecast()` to access forecasts and confidence intervals

In [ ]:
```
# Create a correct train_test_split to predict the last 50 points
train = df['linearized'][0:150]
test = df['linearized'][150:]

# Build model
arima = ARIMA(train, order=(2, 1, 0), trend='t')
arima = arima.fit()

## Forecast
# Forecast values
forecast = arima.forecast(len(test), alpha=0.05)  # 95% confidence

# Forecast values and confidence intervals
forecast_results = arima.get_forecast(len(test), alpha=0.05)
forecast = forecast_results.predicted_mean
confidence_int = forecast_results.conf_int().values
```

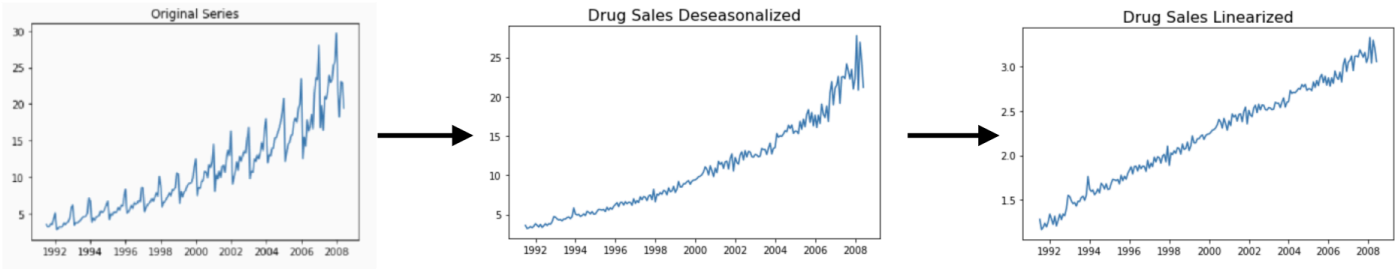
In [ ]:
```
# We define here a "Plot forecast vs. real", which also shows historic
al training set

def plot_forecast(fc, train, test, upper=None, lower=None):
    is_confidence_int = isinstance(upper, np.ndarray) and isinstance(l
ower, np.ndarray)
    # Prepare plot series
    fc_series = pd.Series(fc, index=test.index)
    lower_series = pd.Series(upper, index=test.index) if is_confidence
_int else None
    upper_series = pd.Series(lower, index=test.index) if is_confidence
_int else None

    # Plot
    plt.figure(figsize=(10,4), dpi=100)
    plt.plot(train, label='training', color='black')
    plt.plot(test, label='actual', color='black', ls='--')
    plt.plot(fc_series, label='forecast', color='orange')
    if is_confidence_int:
        plt.fill_between(lower_series.index, lower_series, upper_serie
s, color='k', alpha=.15)
    plt.title('Forecast vs Actuals')
    plt.legend(loc='upper left', fontsize=8);
```

In [ ]:
```
plot_forecast(forecast, train, test, confidence_int[:,0], confidence_int[:,1])
```



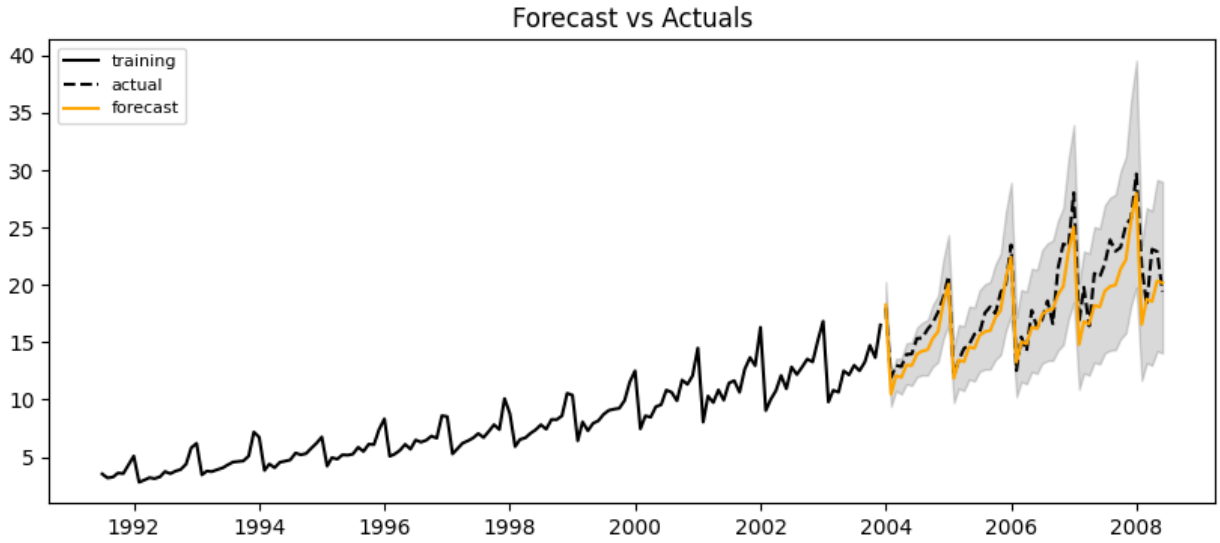
⚠️Remember, this wasn't our original Time Series!

This is the linearized version, we extracted the seasonal and exponential components.



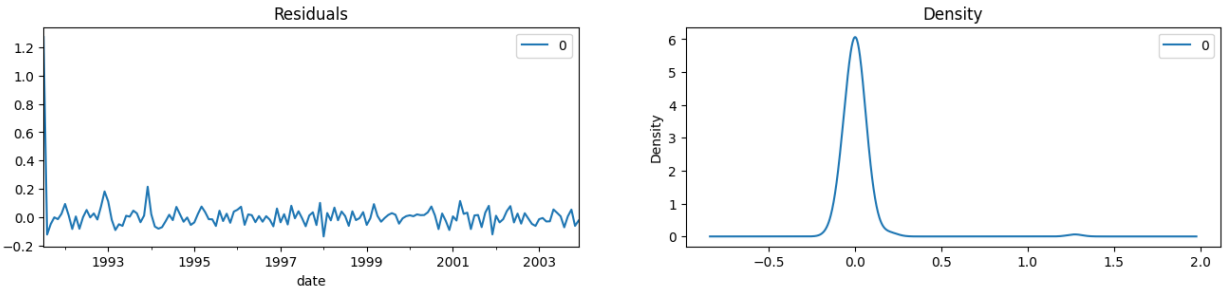
Let's add them back in

In [ ]:
```
# Re-compose back to initial TS

forecast_recons = np.exp(forecast) * result_mul.seasonal[150:]
train_recons = np.exp(train) * result_mul.seasonal[0:150]
test_recons = np.exp(test) * result_mul.seasonal[150:]
lower_recons = np.exp(confidence_int)[:, 0] * result_mul.seasonal[15
0:]
upper_recons = np.exp(confidence_int)[:, 1] * result_mul.seasonal[15
0:]

# Plot
plot_forecast(forecast_recons, train_recons, test_recons, lower_recon
s.values, upper_recons.values)
```



Check residuals for inference validity

In [ ]:
```
residuals = pd.DataFrame(arima.resid)

fig, ax = plt.subplots(1,2, figsize=(16,3))
residuals.plot(title="Residuals", ax=ax[0])
residuals.plot(kind='kde', title='Density', ax=ax[1]);
```



✅ Residuals of equal variance over time (i.e. homoskedastic) ✅ Approximately normally distributed

👉 We can trust our "confidence interval"

What should our full workflow be? The Box Jenkins Method!

1. Apply transformations to make Time Series stationary (*de-trending*, transformations, differencing)
2. If differencing, keep track of order `d` of differencing; don't over-difference!
3. Confirm stationarity (visually, ACF, ADF test)
4. Plot ACF/PACF, identify likely AR and MA orders `p`, `q`.
5. Fit **original** (non-differenced) data with ARIMA model of order `p`, `d`, `q`
6. Try a few other values around these orders
7. If all models with similarly low AIC, pick the least complex one
8. Inspect residuals: if ACF and PACF show white noise (no signal left) ➔ you are done.
9. Otherwise ➔ iterate (try other transformations, change order of differencing, add/remove MA/AR terms)
10. (Optional) Compare with Auto-ARIMA output trace

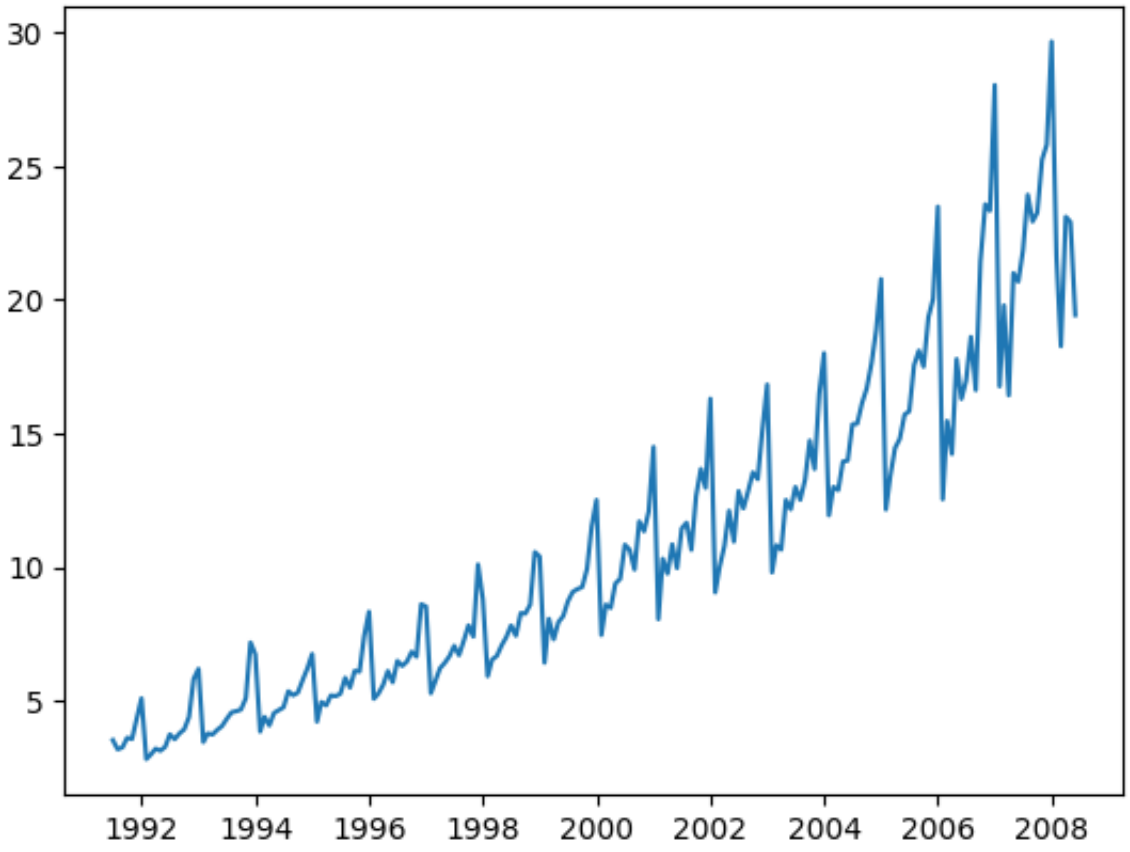
Take a look at the rules here (http://people.duke.edu/~rnau/arimrule.htm) and here (https://people.duke.edu/~rnau/411arim3.htm)

🔟 Optional: Seasonal ARIMA (SARIMA)

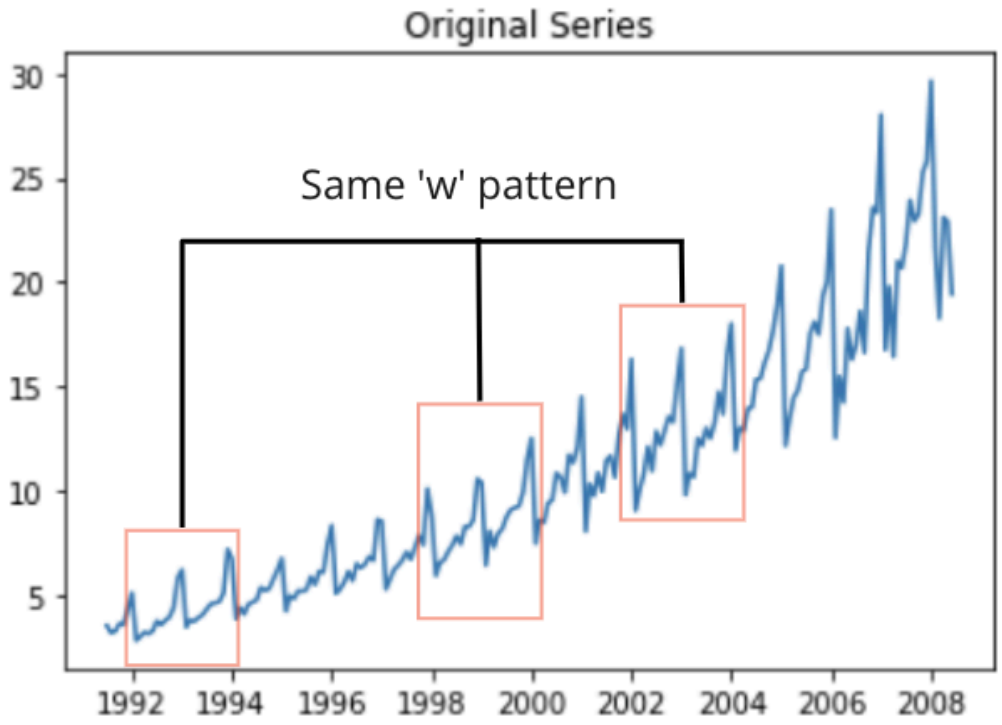
Removes the need to *de-seasonalize* our data

Let's plot our original Time Series one more time:

In [ ]:
```
plt.plot(df.value);
```



Where are the season windows in our data?



We've identified a repeating pattern within our data. The window looks like a window of size 12

This makes sense! Yearly patterns are very common

❌ Not great for stationarity

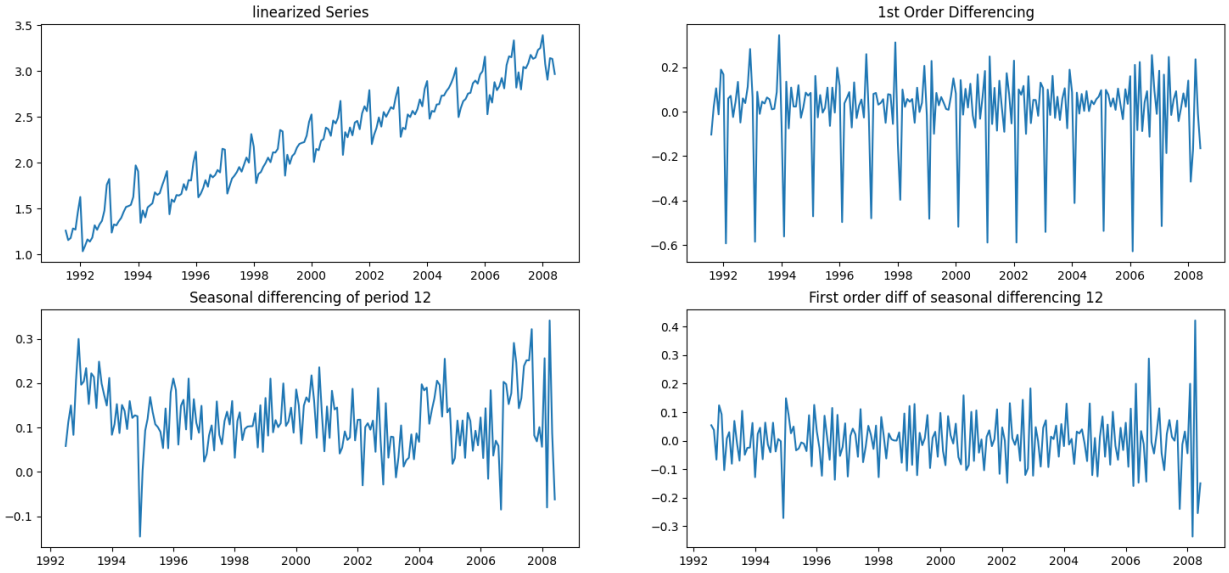
We're going to take seasonal patterns into account by modeling at the individual lag level **and** at the seasonal level

Let's use our differencing as an example of modeling at different time windows.

We'll use $Y_{diff} = Y_t - Y_{t-k}$

- Normal differencing: $Y_{diff} = Y_t - Y_{t-k}$ ⬅ individual lag
- Seasonal differencing: $Y_{sdiff} = Y_t - Y_{t-m}$ (m=12) ⬅ 12 month lag

In [ ]:
```
fig, axs = plt.subplots(2, 2, figsize=(18,8))
# keeping just log transform to stay ~ linear
df['log'] = np.log(df.value)

# linearized series
axs[0,0].plot(df.log); axs[0,0].set_title('linearized Series')

# Normal differencing
axs[0,1].plot(df.log.diff(1)); axs[0,1].set_title('1st Order Differenc
ing')

# Seasonal differencing
axs[1,0].plot(df.log.diff(12))
axs[1,0].set_title('Seasonal differencing of period 12')

# Sesonal + Normal differencing
axs[1,1].plot(df.log.diff(12).diff(1))
axs[1,1].set_title('First order diff of seasonal differencing 12');
```



Just like we applied differencing at two different lag levels, we can apply the other hyperparmeters (p and q) as well!

SARIMA has 7 hyperparameters

```
SARIMA(p,d,q)(P,D,Q)[S]
```

p ~ AR hyperparameter for individual lag level

d ~ Integration hyperparameter for individual lag level

q ~ MA hyperparameter for individual lag level

P ~ AR hyperparameter for seasonal lag level

D ~ Integration hyperparameter for seasonal lag level

Q ~ MA hyperparameter for seasonal lag level

👉 **GridSearch advised**
👉 Must only choose S manually (S = 12, for annual seasonality)

📚 Blog post on hyperparams (https://web.archive.org/web/20210924121131/https://www.datasciencecentral.com/profiles/blogs/tutorial-forecasting-with-seasonal-arima)

In [ ]:
```
# Create a correct Training/Test split to predict the last 50 points
train = df.log[0:150]
test = df.log[150:]
```

In [ ]:
```
smodel = pm.auto_arima(train, seasonal=True, m=12,
                       start_p=0, max_p=1, max_d=1, start_q=0, max_q=
1,
                       start_P=0, max_P=2, max_D=1, start_Q=0, max_Q=
2,
                       trace=True, error_action='ignore', suppress_war
nings=True)
```

```
Performing stepwise search to minimize aic
 ARIMA(0,1,0)(0,0,0)[12] intercept   : AIC=-83.662, Time=0.02 sec
 ARIMA(1,1,0)(1,0,0)[12] intercept   : AIC=inf, Time=0.47 sec
 ARIMA(0,1,1)(0,0,1)[12] intercept   : AIC=inf, Time=0.28 sec
 ARIMA(0,1,0)(0,0,0)[12]             : AIC=-85.171, Time=0.02 sec
 ARIMA(0,1,0)(1,0,0)[12] intercept   : AIC=inf, Time=0.20 sec
 ARIMA(0,1,0)(0,0,1)[12] intercept   : AIC=-190.471, Time=0.12 sec
 ARIMA(0,1,0)(1,0,1)[12] intercept   : AIC=-315.673, Time=0.29 sec
 ARIMA(0,1,0)(2,0,1)[12] intercept   : AIC=-331.977, Time=0.79 sec
 ARIMA(0,1,0)(2,0,0)[12] intercept   : AIC=inf, Time=0.52 sec
 ARIMA(0,1,0)(2,0,2)[12] intercept   : AIC=-326.666, Time=1.23 sec
 ARIMA(0,1,0)(1,0,2)[12] intercept   : AIC=inf, Time=0.98 sec
 ARIMA(1,1,0)(2,0,1)[12] intercept   : AIC=-363.394, Time=0.90 sec
 ARIMA(1,1,0)(1,0,1)[12] intercept   : AIC=-360.691, Time=0.49 sec
 ARIMA(1,1,0)(2,0,0)[12] intercept   : AIC=inf, Time=0.63 sec
 ARIMA(1,1,0)(2,0,2)[12] intercept   : AIC=-366.797, Time=1.42 sec
 ARIMA(1,1,0)(1,0,2)[12] intercept   : AIC=-379.982, Time=0.99 sec
 ARIMA(1,1,0)(0,0,2)[12] intercept   : AIC=inf, Time=0.70 sec
 ARIMA(1,1,0)(0,0,1)[12] intercept   : AIC=-208.145, Time=0.10 sec
 ARIMA(1,1,1)(1,0,2)[12] intercept   : AIC=-398.203, Time=1.10 sec
 ARIMA(1,1,1)(0,0,2)[12] intercept   : AIC=inf, Time=1.00 sec
 ARIMA(1,1,1)(1,0,1)[12] intercept   : AIC=-391.755, Time=0.40 sec
 ARIMA(1,1,1)(2,0,2)[12] intercept   : AIC=-396.249, Time=1.28 sec
 ARIMA(1,1,1)(0,0,1)[12] intercept   : AIC=inf, Time=0.30 sec
 ARIMA(1,1,1)(2,0,1)[12] intercept   : AIC=-387.534, Time=0.80 sec
 ARIMA(0,1,1)(1,0,2)[12] intercept   : AIC=-395.079, Time=1.15 sec
 ARIMA(1,1,1)(1,0,2)[12]             : AIC=-409.107, Time=0.98 sec
 ARIMA(1,1,1)(0,0,2)[12]             : AIC=inf, Time=0.92 sec
 ARIMA(1,1,1)(1,0,1)[12]             : AIC=-396.959, Time=0.25 sec
 ARIMA(1,1,1)(2,0,2)[12]             : AIC=-407.390, Time=1.42 sec
 ARIMA(1,1,1)(0,0,1)[12]             : AIC=-230.235, Time=0.28 sec
 ARIMA(1,1,1)(2,0,1)[12]             : AIC=-394.637, Time=0.94 sec
 ARIMA(0,1,1)(1,0,2)[12]             : AIC=-410.871, Time=1.33 sec
 ARIMA(0,1,1)(0,0,2)[12]             : AIC=inf, Time=0.86 sec
 ARIMA(0,1,1)(1,0,1)[12]             : AIC=-397.771, Time=0.27 sec
 ARIMA(0,1,1)(2,0,2)[12]             : AIC=-408.683, Time=1.38 sec
 ARIMA(0,1,1)(0,0,1)[12]             : AIC=-224.315, Time=0.12 sec
 ARIMA(0,1,1)(2,0,1)[12]             : AIC=inf, Time=0.79 sec
 ARIMA(0,1,0)(1,0,2)[12]             : AIC=inf, Time=1.06 sec
 ARIMA(1,1,0)(1,0,2)[12]             : AIC=inf, Time=1.12 sec

Best model:  ARIMA(0,1,1)(1,0,2)[12]
Total fit time: 27.945 seconds
```

☝️ best model found

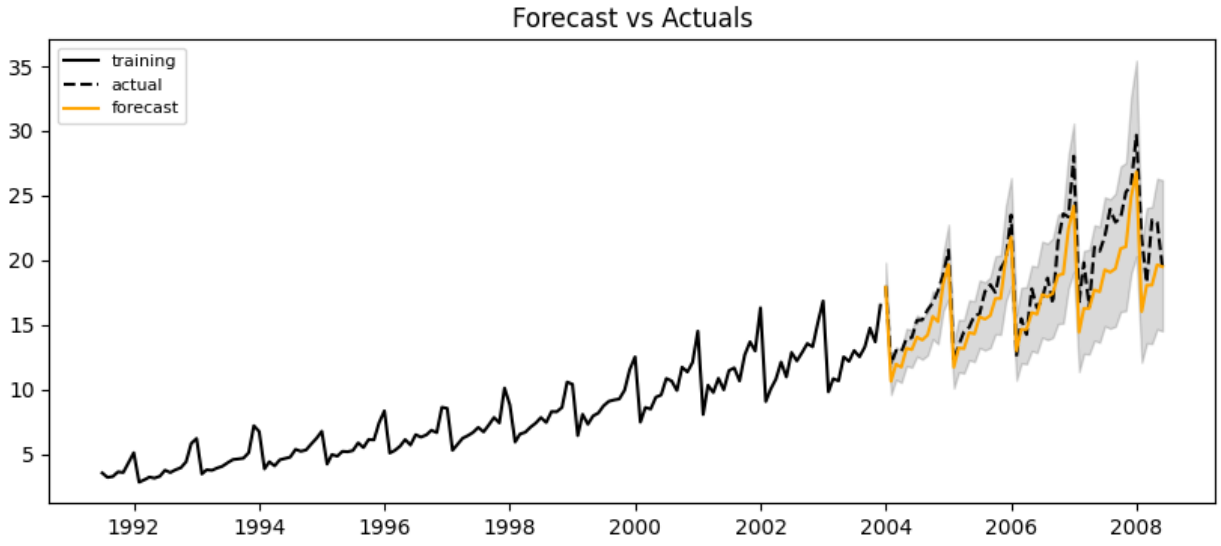
```
SARIMAX(train, order=(0, 1, 1), seasonal_order=(2, 0, 2, 12))
```

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

# Build Model
sarima = SARIMAX(train, order=(0, 1, 1), seasonal_order=(2, 0, 2, 12))
sarima = sarima.fit(maxiter=75)

# Forecast
results = sarima.get_forecast(len(test), alpha=0.05)
forecast = results.predicted_mean
confidence_int = results.conf_int()
```

```
 This problem is unconstrained.

RUNNING THE L-BFGS-B CODE

           * * *

Machine precision = 2.220D-16
 N =            6     M =           10

At X0         0 variables are exactly at the bounds

At iterate    0    f= -6.91380D-01    |proj g|=  4.90088D+00

At iterate    5    f= -1.31919D+00    |proj g|=  1.64706D+00

At iterate   10    f= -1.33494D+00    |proj g|=  2.49931D+00

At iterate   15    f= -1.36738D+00    |proj g|=  2.68968D-01

At iterate   20    f= -1.36776D+00    |proj g|=  6.87559D-02

At iterate   25    f= -1.37303D+00    |proj g|=  2.16324D-01

At iterate   30    f= -1.37506D+00    |proj g|=  1.45971D-02

At iterate   35    f= -1.37649D+00    |proj g|=  4.77421D-01

At iterate   40    f= -1.39074D+00    |proj g|=  8.40582D-01

At iterate   45    f= -1.40149D+00    |proj g|=  3.62746D-01

At iterate   50    f= -1.40228D+00    |proj g|=  1.24331D-01

At iterate   55    f= -1.40544D+00    |proj g|=  1.57332D-01

At iterate   60    f= -1.40574D+00    |proj g|=  5.41910D-03

At iterate   65    f= -1.40575D+00    |proj g|=  2.86790D-02
```

```
At iterate   70    f= -1.40578D+00    |proj g|=  2.54240D-03

           * * *

Tit   = total number of iterations
Tnf   = total number of function evaluations
Tnint = total number of segments explored during Cauchy searches
Skip  = number of BFGS updates skipped
Nact  = number of active bounds at final generalized Cauchy point
Projg = norm of the final projected gradient
F     = final function value

           * * *

   N    Tit     Tnf  Tnint  Skip  Nact     Projg        F
    6     73     103      1     0     0   1.452D-03  -1.406D+00
  F =  -1.4057781232628566

CONVERGENCE: REL_REDUCTION_OF_F_<=_FACTR*EPSMCH
```

In [ ]:
```
# Reconstruct by taking exponential
forecast_recons = pd.Series(np.exp(forecast), index=test.index)
lower_recons = np.exp(confidence_int['lower log']).values
upper_recons = np.exp(confidence_int['upper log']).values

plot_forecast(forecast_recons, np.exp(train), np.exp(test), upper = up
per_recons, lower=lower_recons)
```



In [ ]:
```
sarima.summary()
```

Out[ ]:

SARIMAX Results

Dep. Variable:	log	**No. Observations:**	150
Model:	SARIMAX(0, 1, 1)x(2, 0, [1, 2], 12)	**Log Likelihood**	210.867
Date:	Mon, 31 Jul 2023	**AIC**	-409.733
Time:	17:27:38	**BIC**	-391.710
Sample:	07-01-1991	**HQIC**	-402.411
	- 12-01-2003		
Covariance Type:	opg		

	coef	std err	z	P>|z|	[0.025	0.975]
ma.L1	-0.7951	0.059	-13.544	0.000	-0.910	-0.680
ar.S.L12	1.2089	0.057	21.258	0.000	1.097	1.320
ar.S.L24	-0.2097	0.057	-3.696	0.000	-0.321	-0.098
ma.S.L12	-0.6383	0.091	-7.015	0.000	-0.817	-0.460
ma.S.L24	-0.2543	0.076	-3.352	0.001	-0.403	-0.106
sigma2	0.0026	0.000	8.635	0.000	0.002	0.003

Ljung-Box (L1) (Q):	0.07	**Jarque-Bera (JB):**	10.24
Prob(Q):	0.79	**Prob(JB):**	0.01
Heteroskedasticity (H):	0.75	**Skew:**	-0.27
Prob(H) (two-sided):	0.31	**Kurtosis:**	4.17

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

11) Other types of Time Series models:

- SARIMAX (uses an external time series to help predict our main time series e.g. using weather forecast to predict energy prices)
- Facebook Prophet
- Exponential Smoothing (ETS): moving average with exponentially decaying memory of past values; great for non-linear trends with changing mean over time
- Holt's Trend-Corrected Exponential Smoothing: series has a linear trend with a slope that changes over time
- Holt-Winter's method: adds seasonality

See more (https://otexts.com/fpp2/index.html) Using exogenous features

Bibliography

- Academic slides (https://online.stat.psu.edu/stat510/lesson/2/2.1)
- Great youtube channel (https://www.youtube.com/watch?v=AN0a58F6cxA)
- Time Series Analysis Blog (1/2 : concepts) (https://www.machinelearningplus.com/time-series/time-series-analysis-python/)
- Time Series Analysis Blog (2/2 : arima models) (https://www.machinelearningplus.com/time-series/arima-model-time-series-forecasting-python/)
- Yule Walker equations (https://www.youtube.com/watch?v=PFyp4t16_xk&ab_channel=BarryVanVeen): Advanced!

🚀 Your turn!