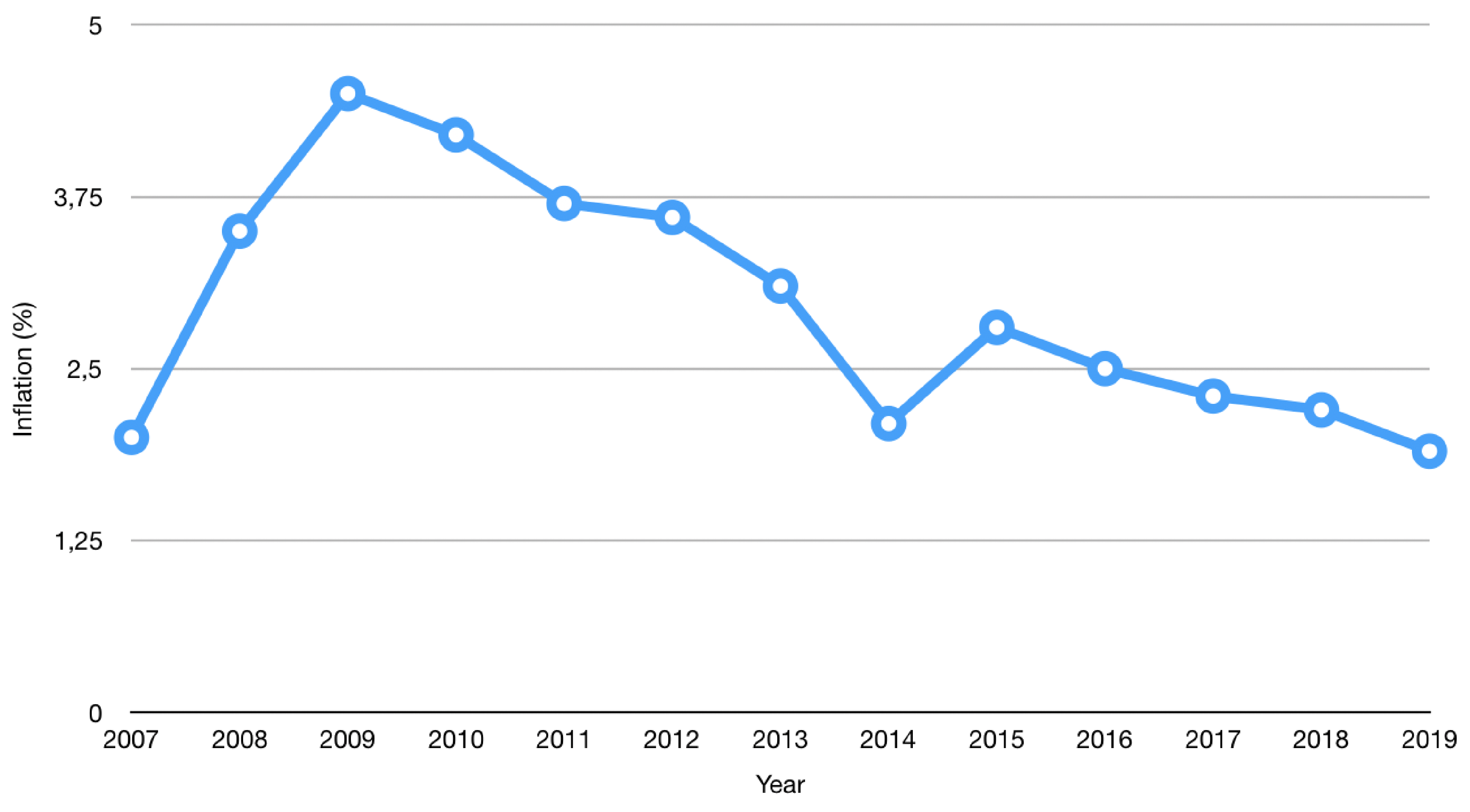
01- Lecture for Time Series

1. What are time series ?

Time series are series of data collected with the same unit over several successive periods.

Examples of time series include :

- stock market value
- yearly inflation
- consumption of a certain good per month
- hourly temperature
- daily exchange rate ...



2. Understanding a time series

To illustrate the main concepts related to time series, we'll analyze the monthly anti-diabetic sales index on the Australian market between 1991 and 2008.

Start off by importing the following packages :

```
### General import
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn import preprocessing

from pandas.plotting import register_matplotlib_converters
register_matplotlib_converters()
from IPython.display import Image
```

```
df = pd.read_csv('file.csv', parse_dates=['date'], index_col='date')
df.head(20)
```

```
df.describe()
```

	value
count	204.000000
mean	10.694430
std	5.956998
min	2.814520
25%	5.844095
50%	9.319345
75%	14.289964
max	29.665356

```
df.index = pd.to_datetime(df.index)
```

2.1. Plotting and understanding a time series

To plot a time series in Python, we have 2 options :

- either the index is a date, and we can plot the column (value) directly
- either the index is not a date, and we need to specify a date (x-axis) and a value (y-axis)

```
plt.figure(figsize=(14,10))
plt.plot(df['value'], label="value")
plt.title("Monthly anti-diabetic sales index on the Australian market between 1991 and 2008")
plt.legend()
plt.show()
```

We can also simply plot dots. It might be useful to identify patterns we do not except, but generally we prefer plotting lines.

```
plt.figure(figsize=(12,8))
plt.plot(df['value'], linewidth = 0.5, linestyle = "None", marker='.')
plt.title("Monthly anti-diabetic sales index on the Australian market between 1991 and 2008")
plt.show()
```

The aim of this section is to get used to analyze patterns and trends in time series. What can we notice ?

- there is an upward trend over the years
- there seems to be a seasonal trend repeated each year
- there is a rising variance over time

2.2. Change the scale

To further explore a time series, it might be necessary to change scale and zoom on a certain period.

We can zoom on a given year to observe the trend :

```
plt.figure(figsize=(12,8))
plt.plot(df[:35]['value'])

plt.title("Value of the index between 1994 and 1997")
plt.show()
```

This confirms that there is a pick in December-January of each year !

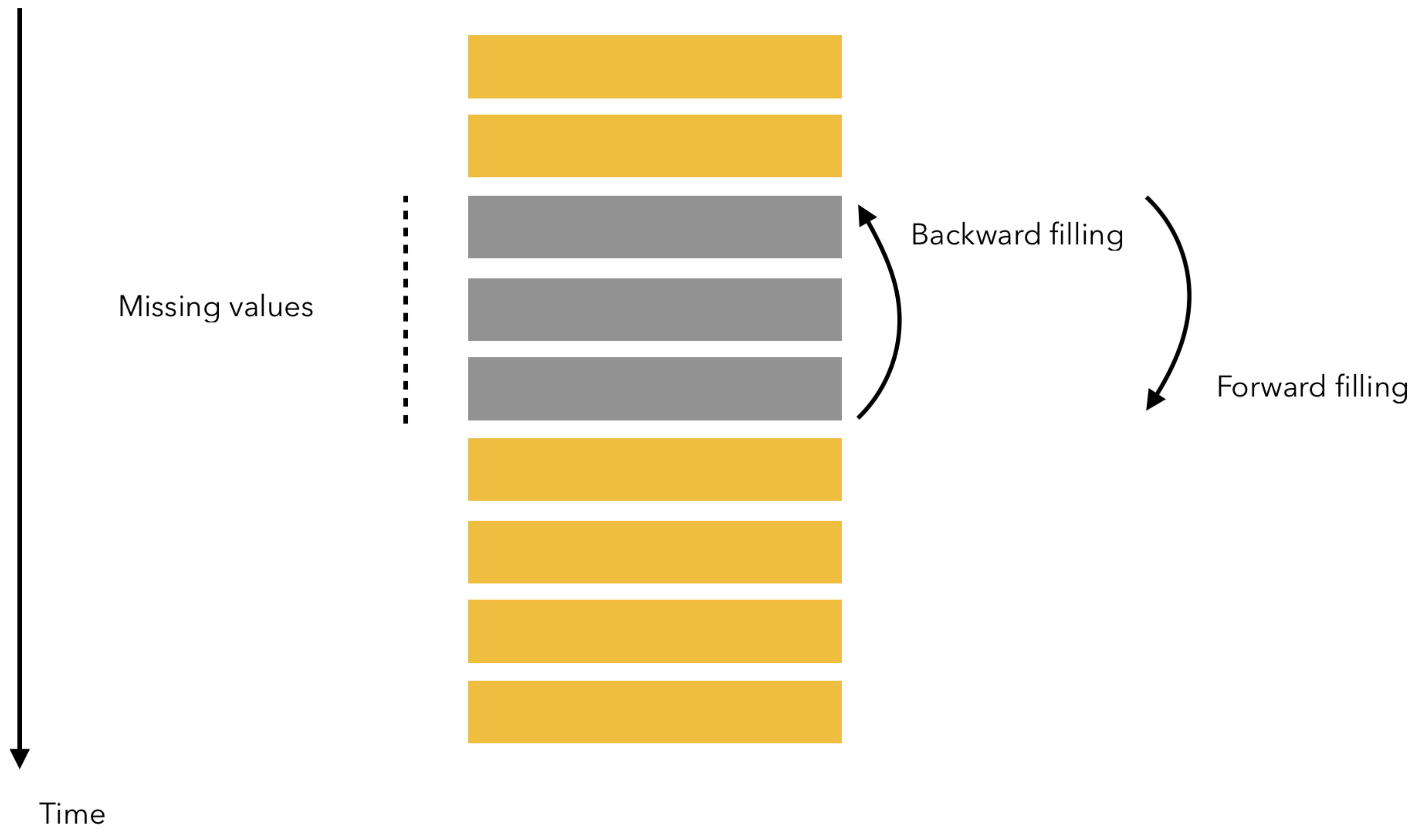
2.3. Fill missing values

```
na_data = df[:35].copy()
na_data.iloc[11:14] = np.nan
na_data.iloc[19:25] = np.nan

plt.figure(figsize=(12,8))
plt.plot(na_data)
plt.show()
```

There are several ways to handle missing values in time series. The first one would be :

- to take the last know value and make a "forward fill", i.e to fill the values with the last known value until a new value is met
- the take the first value after the missing values, and full the values backward



```
ffill = na_data.ffill()
bfill = na_data.bfill()
interp = na_data.interpolate()

plt.figure(figsize=(12, 6))
plt.plot(na_data, 'ro', label='original')
plt.plot(ffill, label='ffill')
plt.plot(bfill, label='bfill')
plt.plot(interp, label='interpolate')
plt.legend()
plt.show()
```

```
ffill = na_data.ffill()
bfill = na_data.bfill()
interp = na_data.interpolate()

plt.figure(figsize=(12, 6))
plt.plot(na_data[9:16], 'ro', label='original')
plt.plot(ffill[9:16], label='ffill')
plt.plot(bfill[9:16], label='bfill')
plt.plot(interp[9:16], label='interpolate')
plt.legend()
plt.show()
```

3. Key concepts in time series

3.1. Auto-correlation

The auto-correlation ρ is defined as the correlation of the series over time, i.e how much the value at time t depends on the value at time $t-j$ for all j.

3.2. Partial Auto-correlation

The partial auto-correlation function (PACF) gives the partial correlation of a stationary time series with its own lagged values, regressed the values of the time series at all shorter lags. It is a regression of the series against its past lags.

If a series is significantly autocorrelated, that means, the previous values of the series (lags) may be helpful in predicting the current value.

To plot the auto-correlation and the partial auto-correlation, we can use statsmodel package :

```
import statsmodels.api as sm

fig, axes = plt.subplots(1, 2, figsize=(15,8))

fig = sm.graphics.tsa.plot_acf(df['value'], lags=12, ax=axes[0])
fig = sm.graphics.tsa.plot_pacf(df['value'], lags=12, ax=axes[1])
```

3.3. Ergodicity

Ergodicity is the process by which we forget the initial conditions. This is reached when auto-correlation of order k tends to 0 as k tends to ∞.

3.4. Training data

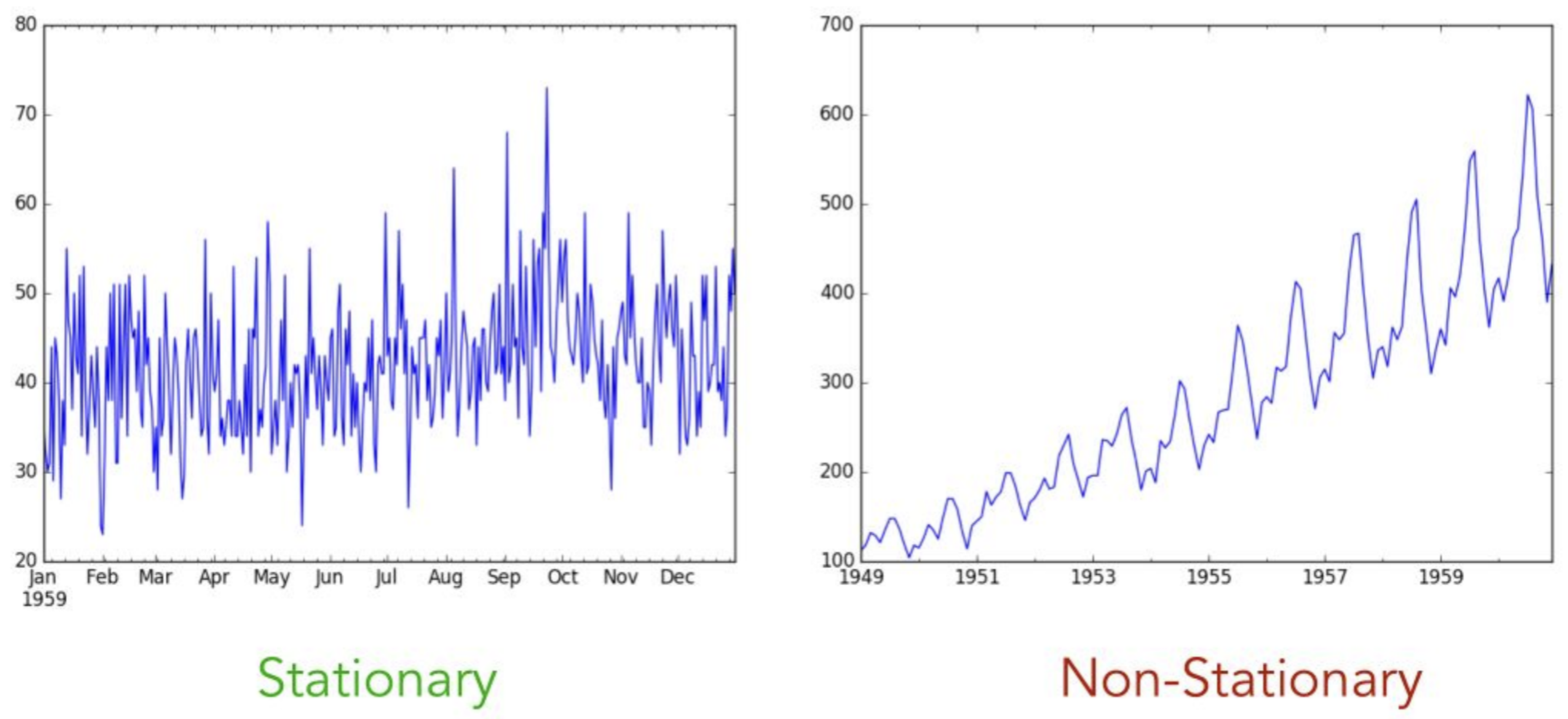
To assess the performance of a model fit, it is important to notice that :

- **We cannot shuffle the data in a random order** !
- We take the training data up to a certain point in time, and then compute the mean squared error (for example) between the prediction and the test
- Make a cross validation to identify the best model parameters to reduce the Mean-Squared Error

4. Decomposition and forecasting

We should only try to predict stationary time series :

A stationary time series is one whose statistical properties such as mean, variance, autocorrelation, etc. are all constant over time



There are 2 scenarios :

- either the time series **is stationary**, and we can make a direct prediction
- either the time series is **not stationary**, and in this case, we need to **decompose** it into several components : trend, seasonality and residual, the residual being a stationary series. Once decomposed, we can forecast the stationary part and rebuild the time series.

There are 2 main models to re-build a time series :

- **Additive model** : Time Series = Constant + Trend + Seasonality + Residual
- **Multiplicative model** : Time Series = Constant *Trend* Seasonality * Residual

We are now able to make predictions using time series. Then, we will apply any forecasting model on the remaining noise, and rebuild a series the following way :TimeSeries=Constant+Trend+Seasonality+Residual

4.1. Test for stationarity

How can we test if a time series is stationary ?

- look at the plots
- look at summary statistics and box plots. A simple trick is to cut the data set in 2, look at mean and variance for each split, and plot the distribution of values for both splits.
- perform statistical tests, using the (Augmented) Dickey-Fuller test

Augmented Dickey-Fuller

Implementation in Python

```
from statsmodels.tsa.stattools import adfuller

result = adfuller(df['value'].ffill(0))
print('ADF Statistic: %f' % result[0])
print('p-value: %f' % result[1])
print('Critical Values:')
for key, value in result[4].items():
    print('\t%s: %.3f' % (key, value))
```

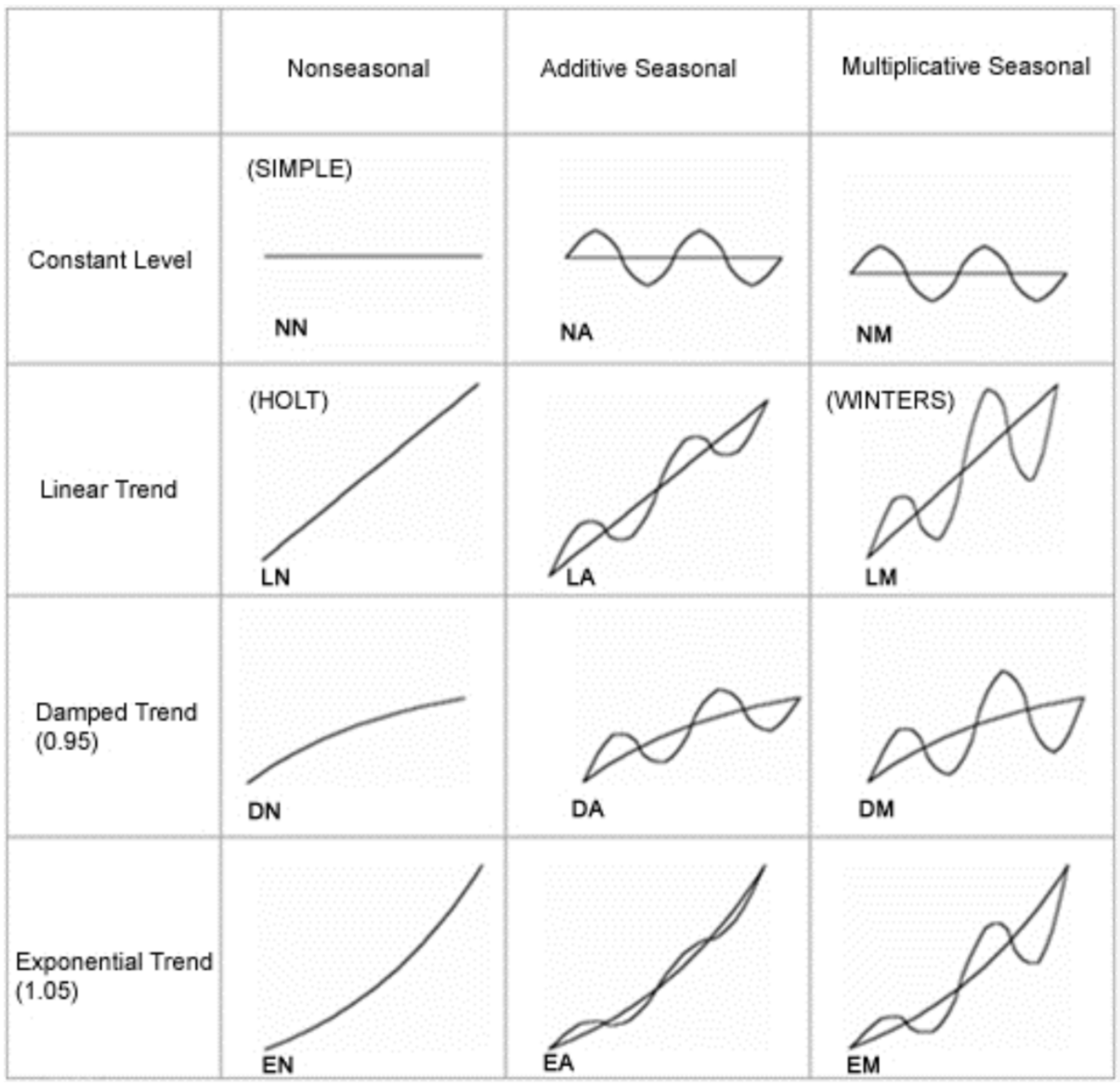
```
ADF Statistic: 3.145186
p-value: 1.000000
Critical Values:
        1%: -3.466
        5%: -2.877
        10%: -2.575
```

4.2. Trends

Trends in time series denote an evolution of the average value over time.

The trends might be :

- constant
- linear
- exponential
- logarithmic (dampened) ...



4.2.1. Logging and de-indexing

The first step toward making a series stationary is to understand the trend. However, sometimes, the variance is modified over time on top of the trend. It is required to de-index or log the series. There are several ways to un-index :

- divide by the index
- transform the series using log, square root... ...

```
plt.figure(figsize=(12,8))
plt.plot(df['value'])
plt.show()
```

In this case, these is a trend that goes upward, but we cannot model through a linear trend the fact that the variance increases over time. Taking the log might completely reduce this effect :

```
plt.figure(figsize=(12,8))
plt.plot(np.log(df['value']))
plt.show()
```

4.2.2. Modeling the trend

To model the trend, we must have a visual sense of what is going on :

```
from sklearn.linear_model import LinearRegression

X = np.array(range(len(df['value'])))
y = np.log(df['value']).ffill(axis=0)

reg = LinearRegression().fit(X.reshape(-1,1), y)
pred_lin = reg.predict(X.reshape(-1,1))
```

```
a_1,b_1 = np.polyfit(np.log(X+1), y, 1)
a_2,b_2 = np.polyfit(X+1, np.log(y), 1)
```

```
pred_log = a_1 * np.log(X+1) + b_1
pred_exp = np.exp(b_2) + np.exp( (X+1) * a_2)
```

```
plt.figure(figsize=(12,8))
plt.plot(np.log(df['value']), label="Index")
plt.plot(df['value'].index, pred_lin, label="linear trend")
plt.plot(df['value'].index, pred_log, label="log trend")
plt.plot(df['value'].index, pred_exp, label="exp trend")
plt.legend()
plt.show()
```

The linear trend seems the most appropriate. Our new series is now :

log(Initial series)−linear trend

To rebuild the final series, you would need :

exp(value)+linear trend at that index

Why should we often use the log transform ?

- Trend measured in natural-log units ≈ percentage growth
- Errors measured in natural-log units ≈ percentage errors

4.3. Seasonality

Seasonality in time series denotes a recurrent pattern over time.

```
series = np.log(df['value']) - pred_lin

plt.figure(figsize=(12,8))
plt.plot(series)
plt.show()
```

Over the course of the years, we observe a regular pattern that looks sinusoidal. The seasonality might take the form of a large pick, a sinusoidal curve...

There are two ways to model seasonality in time series :

- identify patterns that look sinusoidal for example, and fit the right parameters
- the easiest option to remove the trend is to compute the first difference. For example, if there is a yearly seasonality, we can take Yt−Yt−1 (since the measures are made monthly).

```
result = adfuller(series.dropna())
print('ADF Statistic: %f' % result[0])
print('p-value: %f' % result[1])
print('Critical Values:')
for key, value in result[4].items():
    print('\t%s: %.3f' % (key, value))
```

```
ADF Statistic: -3.694921
p-value: 0.004185
Critical Values:
        1%: -3.465
        5%: -2.877
        10%: -2.575
```

Over the course of the years, we observe a regular pattern that looks sinusoidal. The seasonality might take the form of a large pick, a sinusoidal curve...

```
plt.figure(figsize=(12,8))
plt.plot(series - series.shift(12))
plt.show()
```

We have removed most of the trend here, and remain with a stationary series. To make sure that our series is stationary, we can look at the plot : There seems to be no recurrent pattern in the data, a constant variance and mean, no trend...

```
series_stationary = series - series.shift(12)

result = adfuller(series_stationary.dropna())
print('ADF Statistic: %f' % result[0])
print('p-value: %f' % result[1])
print('Critical Values:')
for key, value in result[4].items():
    print('\t%s: %.3f' % (key, value))
```

```
ADF Statistic: -5.214559
p-value: 0.000008
Critical Values:
        1%: -3.467
        5%: -2.878
        10%: -2.575
```

The test now indicates that we must reject the null hypothesis that there is a unit root. The time series is stationary !

4.4. Automatic Decomposition

There are ways to find the trend and the seasonality automatically, using statsmodels.

```
from statsmodels.tsa.seasonal import seasonal_decompose
decomposition = seasonal_decompose(df['value'].ffill(axis=0), period=12, extrapolate_trend='freq')
```

```
plt.figure(figsize=(12,8))
decomposition.plot()
plt.show()
```

```
plt.figure(figsize=(12,8))
plt.plot(df['value'], label="Series")
plt.plot(decomposition.seasonal, label="Seasonality")
plt.plot(decomposition.trend + decomposition.seasonal, label="Trend + Seasonality")
plt.show()
```

According to the documentation, the seasonal component is first removed by applying a convolution filter to the data. The average of this smoothed series for each period is the returned seasonal component.

4.5. Model a stationary series

```
series = np.log(df['value'])

plt.figure(figsize=(12,8))
plt.plot(series, label="Series")
```

```
[]
```

```
series = (np.log(df['value']) - pred_lin)
series = series.dropna()
```

```
plt.figure(figsize=(12,8))
plt.plot(series, label="Series")
```

```
[]
```

There are two main categories of time series models :

- Moving Average (MA) processes
- Auto Regressive (AR) processes

All other models are basically variations and mix of AR and MA.

- The **Moving Average (MA)** model specifies that the output variable depends linearly on the previous values of the series. The model depends on μ, the mean of the series, and on errors made at the previous predictions : θ1, ..., θq are the parameters, and εt, εt−1,..., εt−q are white noise error terms.
- The **Auto Regressive (AR)** model specifies that the output variable depends linearly on its own previous values and on a stochastic term. The model assigns a weight to each variable in the past, and generally gives more weight to recent values than to old ones. One the weight reaches 0, the variables does not influence the present value anymore.

4.5.1. Moving Average (MA) processes

Moving average processes are processes that depend on a deterministic mean, an error term, and the error term of the previous observation.

```
from sklearn.metrics import mean_squared_error
from statsmodels.tsa.arima_model import ARMA

size = int(len(series) * 0.75)
train, test = series[:size], series[size:]
test = test.reset_index()['value']
history = [x for x in train]
predictions = []
```

```
# walk forward over time steps in test
for t in range(len(test)):
    model = ARMA(history, order=(0,2))
    model_fit = model.fit(disp=0)
    output = model_fit.forecast()
    yhat = output[0]
    predictions.append(yhat)
    obs = test[t]
    history.append(yhat)
```

```
# plot
plt.figure(figsize=(12,8))
plt.plot(test)
plt.title("AR prediction")
plt.plot(predictions, color='red',alpha=0.5)
plt.show()
```

```
mean_squared_error(predictions, test)
```

```
0.02337667112153585
```

4.5.2. Auto Regressive (AR) processes

The **autoregression (AR) method** models the next step in the sequence as a linear function of the observations at prior time steps.

The notation for the model involves specifying the order of the model p as a parameter to the AR function, e.g. AR(p). For example, AR(1) is a first-order autoregression model.

```
from statsmodels.tsa.ar_model import AR

size = int(len(series) * 0.75)
train, test = series[0:size], series[size:len(series)]
test = test.reset_index()['value']
history = [x for x in train]
predictions = []

for t in range(len(test)):
    model = AR(history)
    model_fit = model.fit(disp=0)
    output = model_fit.predict()
    yhat = output[0]
    predictions.append(yhat)
    obs = test[t]
    history.append(yhat)
```

```
plt.figure(figsize=(12,8))
plt.plot(predictions, label="Prediction")
plt.plot(test, label="Series")
plt.title("AR prediction")
plt.legend()
plt.show()
```

```
mean_squared_error(predictions, test)
```

```
0.028324083656471554
```

4.5.3. Auto Regressive Moving Average (ARMA) processes

The **Autoregressive Moving Average (ARMA)** method models the next step in the sequence as a linear function of the observations and resiudal errors at prior time steps.

It combines both Autoregression (AR) and Moving Average (MA) models.

The notation for the model involves specifying the order for the AR(p) and MA(q) models as parameters to an ARMA function, e.g. ARMA(p, q). An ARIMA model can be used to develop AR or MA models.

```
from statsmodels.tsa.arima_model import ARIMA

size = int(len(series) * 0.75)
train, test = series[0:size], series[size:len(series)]
test = test.reset_index()['value']
history = [x for x in train]
```

```
predictions = []

for t in range(len(test)):
    model = ARIMA(history, order=(2,0,1))
    model_fit = model.fit(disp=0)
    output = model_fit.forecast()
    yhat = output[0]
    predictions.append(yhat)
    obs = test[t]
    history.append(yhat)
```

```
plt.figure(figsize=(12,8))
plt.plot(predictions, label="Prediction")
plt.plot(test, label="Series")
plt.title("ARMA prediction")
plt.legend()
plt.show()
```

```
mean_squared_error(predictions, test)
```

```
0.023779785091510824
```

4.5.4. Auto Regressive Integrated Moving Average (ARIMA) processes

The **Autoregressive Integrated Moving Average (ARIMA)** method models the next step in the sequence as a linear function of the differenced observations and residual errors at prior time steps.

It combines both Autoregression (AR) and Moving Average (MA) models as well as a differencing pre-processing step of the sequence to make the sequence stationary, called integration (I).

```
size = int(len(series) * 0.75)
train, test = series[0:size], series[size:len(series)]
test = test.reset_index()['value']
history = [x for x in train]
predictions = []

for t in range(len(test)):
    model = ARIMA(history, order=(1,1,1))
    model_fit = model.fit(disp=0)
    output = model_fit.forecast()
    yhat = output[0]
    predictions.append(yhat)
    obs = test[t]
    history.append(yhat)
```

```
plt.figure(figsize=(12,8))
plt.plot(predictions, label="Prediction")
plt.plot(test, label="Series")
plt.title("ARIMA prediction")
plt.legend()
plt.show()
```

```
mean_squared_error(predictions, test)
```

```
0.02397521944525325
```

4.5.5. Seasonal Autoregressive Integrated Moving-Average (SARIMA) process

The **Seasonal Autoregressive Integrated Moving Average (SARIMA)** method models the next step in the sequence as a linear function of the differenced observations, errors, differenced seasonal observations, and seasonal errors at prior time steps.

It combines the ARIMA model with the ability to perform the same autoregression, differencing, and moving average modeling at the seasonal level.

Parameters :

- AR(p), I(d), and MA(q) models as parameters to an ARIMA function
- AR(P), I(D), MA(Q) and m (seasonal period) parameters at the seasonal level

```
import warnings
warnings.filterwarnings('ignore')
```

```
# SARIMA example
from statsmodels.tsa.statespace.sarimax import SARIMAX

size = int(len(series) * 0.75)
train, test = series[0:size], series[size:len(series)]
test = test.reset_index()['value']
history = [x for x in train]
predictions = list()

for t in range(len(test)):
    model = SARIMAX(history, order=(1, 1, 1), seasonal_order=(1, 1, 1, 12))
    model_fit = model.fit(disp=False)
    output = model_fit.forecast()
    yhat = output[0]
    predictions.append(yhat)
    obs = test[t]
    history.append(yhat)
```

```
plt.figure(figsize=(12,8))
plt.plot(predictions, label="Prediction")
plt.plot(test, label="Series")
plt.title("SARIMA prediction")
plt.legend()
plt.show()
```

Now we buid the full prediction and compare it to the initial TS. Here we do the inverse of what we did in part 4.2.b

```
history = [x for x in train]

rebuilt_pred = np.concatenate([history, predictions]) + pred_lin
rebuilt_ts = np.concatenate([history, test]) + pred_lin

plt.figure(figsize=(12,8))
plt.plot(np.exp(rebuilt_pred), label='Prediction')
plt.plot(np.exp(rebuilt_ts), label='Test')
plt.legend()
plt.show()
```

We Check the MSEs.

```
mean_squared_error(predictions, test)
```

```
0.0060992448388906455
```

```
mean_squared_error(rebuilt_ts[size:], rebuilt_pred[size:])
```

```
0.006099244838890647
```

4.6. Forecast a time series without pre-processing.

Alright, we now have all the elements to recompose the time series. We demonstrate the power of the SARIMAX model.

```
# SARIMA example
from statsmodels.tsa.statespace.sarimax import SARIMAX

size = int(len(df['value'].dropna()) * 0.75)
train, test = df['value'].dropna()[0:size], df['value'].dropna()[size:len(df['value'].dropna())]
test = test.reset_index()['value']
history = [x for x in train]
predictions = list()

for t in range(len(test)):
    model = SARIMAX(history, order=(1, 1, 1), seasonal_order=(1, 1, 1, 12))
    model_fit = model.fit(disp=False)
    output = model_fit.forecast()
    yhat = output[0]
    predictions.append(yhat)
    obs = test[t]
    history.append(yhat)
```

```
history = [x for x in train]

plt.figure(figsize=(12,8))
plt.plot(np.concatenate([history, predictions]), label='Prediction')
plt.plot(np.concatenate([history, test]), label='Test')
plt.legend()
plt.show()
```

```
mean_squared_error(predictions, test)
```

```
10.129662960416333
```

5. Introducing Prophet by Facebook

Alright, all this introduction was interesting, and this is basically how it is done as soon as you want to do some research on this topic and need to explain the outcome of your models. There is however a way to speed up the process ! Using Prophet by Facebook, an automate time-series forecasting library.

You need to install the library first : `pip install fbprophet`

You may also need to install plotly : `pip install plotly`

```
from fbprophet import Prophet
from pandas.plotting import register_matplotlib_converters
register_matplotlib_converters()
```

```
df.head()
```

	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

```
df = pd.DataFrame(df['value'].dropna()).reset_index().rename(columns={'date': 'ds', 'value': 'y'})
df.head()
```

	ds	y
0	1991-07-01	3.526591
1	1991-08-01	3.180891
2	1991-09-01	3.252221
3	1991-10-01	3.611003
4	1991-11-01	3.565869

```
model = Prophet(interval_width=0.95)
model.fit(df)
```

```
INFO:fbprophet:Disabling weekly seasonality. Run prophet with weekly_seasonality=True to override this.
INFO:fbprophet:Disabling daily seasonality. Run prophet with daily_seasonality=True to override this.
/Users/nmetzger/bin/anaconda/lib/python3.7/site-packages/pystan/misc.py:399: FutureWarning:

Conversion of the second argument of issubdtype from `float` to `np.floating` is deprecated. In future, it will be treated as `np.float64 =
= np.dtype(float).type`.
```

```
future = model.make_future_dataframe(periods=36, freq='MS')
future.head()
```

	ds
0	1991-07-01
1	1991-08-01
2	1991-09-01
3	1991-10-01
4	1991-11-01

```
forecast = model.predict(future)
forecast.head()
```

```
plt.figure(figsize=(18, 8))
model.plot(forecast, xlabel = 'date', ylabel = 'Wind')
plt.title('Index')
plt.show()
```

```
model.plot_components(forecast);
```

Works fine but we get much better results using pre-processing.