

Data Preparation Doku

Wichtige Funktionen und Konzepte

Inhaltsverzeichnis

<u>WOHLFÜHL STARTEINSTELLUNGEN</u>	<u>3</u>
<u>KAP. 1 AUFGABE 1</u>	<u>3</u>
A 1.2	3
<u>KAP. 1 AUFGABE 2</u>	<u>3</u>
A2.1	3
A2.2	4
<u>KAP.1 AUFGABE 3</u>	<u>4</u>
A3.1	4
A3.2	4
<u>KAP. 1 AUFGABE 4</u>	<u>5</u>
A4.2	5
A4.3	5
A4.4	5
A4.5	5
A4.7	5
<u>KAP1. AUFGABE 6</u>	<u>5</u>
A6.1	5
A6.2	6
A6.3	7
A6.6	7
A 7	8
<u>KAP.2 AUFGABE 1</u>	<u>8</u>
A2	9
A4	9
A5	9
A7	10
A9	11
<u>VERKETTUNGEN</u>	<u>14</u>

<u>PYTHON UMGEBUNG COPY_ON_WRITE</u>	17
<u>ERRORS.....</u>	18
<u>ANDERE FUNKTIONEN.....</u>	19
<u>GROUPS</u>	19
<u>AGGREGATIONEN</u>	19
<u>GROUPED AGGREGATION.....</u>	20
<u>QUANTILE</u>	21
<u>BINS.....</u>	21
<u>NONE.....</u>	21
<u>INTERPOLIEREN.....</u>	23
<u>STRING OPERATIONEN SERIES.....</u>	23
<u>EINSATZ VON REGULAR EXPRESSIONS (RE REGEX)</u>	25
<u>TIDY DATA DEFINITION</u>	28
<u>TIDY DATA LONG AND WIDE</u>	28
<u>JOIN ARTEN</u>	31
<u>VEKTOR OPERATIONEN.....</u>	32

Wohlfühl Starteinstellungen

```
import pandas as pd  
%mode minimal  
pd.set_option("mode.copy_on_write", True)
```

Kap. 1 Aufgabe 1

A 1.2

Data Frame erzeugen

```
pd.DataFrame(zB Dict, JSON...)
```

Zeilen und Spaltenauswahl

```
Df.loc[df[„spalte“] == Wert, „auszugebende Spalte“]
```

Einzelnen Wert erhalten ohne weitere Infos (Wert = df.Projektion)

```
Wert.item()
```

Index setzen einer Spalte

```
Df.set_index(„Spaltennamen“)
```

Zwei oder mehrere Spalten als Index

```
Df.set_index([„Spalte1“, „Spalte2“])
```

Aus einem DF die einzeln Werte pro Zeile ausgeben lassen

```
for year,temp in dfTemp.iterrows():
```

Projektion einer Spalte

```
dfTemp[“temp”]
```

Durchschnitt berechnen einer Spalte

```
df.mean()
```

Kap. 1 Aufgabe 2

A2.1

Datentyp bekommen

```
df.dtypes
```

Datentyp ändern

```
df[„Spalte1“].astype(„float64“)
```

Zeilen und Spalten Anzahl

```
df.shape
```

Transponieren eines DataFrames

```
df.T
```

Den Index ausgeben lassen

```
df.index
```

A2.2

Spalten Ebene

Bei doppelten Spaltennamen eine ‚Zeile‘ auswählen.

	Flag, name and postal abbreviation[8]	Cities		Ratification or admission[A]	Population (2020) [10]	Total area[11]		Reps.	
	Flag, name and postal abbreviation[8]	Flag, name and postal abbreviation[8].1	Capital	Largest[12]	Ratification or admission[A]	Population (2020) [10]	mi2	km2	Unnamed: 8_level_1
0	Alabama		AL	Montgomery	Huntsville	Dec 14, 1819	5024279	52420	135767
1	Alaska		AK	Juneau	Anchorage	Jan 3, 1959	733391	665384	1723337

```
df.columns.get_level_values(1)
```

Kap.1 Aufgabe 3

A3.1

Spaltennamen umbenennen

```
df.rename(columns={"Wind_max": "Windspitze"}, inplace=True)
```

Kopie erstellen von DF

```
Df.copy()
```

A3.2

Konzept: Neue Spalte erstellen aus Berechnung von zwei Spalten

```
wetter["NeueSpalte"] = wetter["Spalte1"] +*/ wetter["Spalte2"]
```

Spalte löschen

```
wetter.drop(columns=["Taupunkt"], inplace=True)
```

Mehrere Spalten löschen

```
wetter.drop(columns=["Taupunkt", "Luftdruck"], inplace=True)
```

Wenn bei der Verkettung auf eine Spalte zugegriffen werden soll bsp. Datentyp ändern

wird `.assign()` verwendet und der Spaltenname wird ausgeschrieben bsp.

```
(wetter.rename(columns={"Wind_max": "WindGeschwindigkeit"})
    .drop(columns=["Regen", "Sonne"])
    .assign(
        Taupunkt = wetter["Taupunkt"].astype("float64"),
        Spread = wetter["Temperatur"] - wetter["Taupunkt"]
```

```
)
```

Kap. 1 Aufgabe 4

A4.2

Den Index sortieren

```
df.sort_index()
```

A4.3

Selektion mit Bedingung

```
df.loc[(df["Spalte"]==3) | (df["Spalte"]==0)]
```

Logik Operatoren: and (&), or(|), negieren !=

A4.4

Tabelle nach einer Spalte sortieren

```
df. sort_values("Spalte")
```

A4.5

Zeilen nach Index auswählen

```
df.loc[start:ende]
```

Zeilen nach Index mit Bedingung auswählen

```
df.loc[(df.index > 42)& (df["Spalte"] == 0)]
```

A4.7

Wert in Spalte einfügen nach einer Bedingung

```
wetter.loc[wetter["Sonne"] > 0, "Text"] = "April"
```

Werte in Spalte einfügen nach Bedingung und überprüfung auf NA Werte

```
wetter.loc[wetter["Text"].isna(), "Text"] = "NaN"
```

Kap1. Aufgabe 6

A6.1

Gruppieren einer Spalte

```
df.groupby('Spalte')
```

Gruppieren mehreren Spalten

```
df.groupby(['Spalte1','Spalte2'])
```

Funktionen anwenden auf df/series

```
apply(funktion, axis=0)
```

axis=0 [vertikal auf Spalte]

axis=1 [horizontal auf Zeilen]

Werte summieren

```
.sum()
```

Werte zählen

```
.count()
```

Einzigartige Werte aus Spalte

```
Spalte.unique()
```

Liste sortieren

```
Liste.sort()
```

Länge der Liste ermitteln

```
len(Liste)
```

Findet Maxwert:

```
df.max()
```

A6.2

Findet Minwert:

```
df.min()
```

Findet Index von Maxwert

```
df.idxmax()
```

Bsp:

```
t.loc[list(t_class_group["Fare"].idxmin()), 'Name']
```

```
wetter.loc[wetter['Wind_max'].idxmax(), 'Stunde']
```

Findet Index von Minwert:

```
df.idxmin()
```

```
Finden der max/ min Werte : `nlargest()`
```

`1`: Dies gibt an, dass du nur den größten Wert (oder die größten Werte) aus jeder Gruppe haben möchtest.

`keep='all'`: Dies ist wichtig, wenn es mehrere gleich große "größte Werte" in der Gruppe gibt. Normalerweise gibt nlargest nur den ersten dieser Werte zurück, wenn

```
sie gleich groß sind. Aber mit keep='all' sorgst du dafür, dass alle diese gleich großen maximalen Werte beibehalten werden.
```

Bsp:

```
idx = t.groupby(by="Pclass")["Fare"].nlargest(1, keep='all')
```

```
Pclass
1    258    512.3292
      679    512.3292
      737    512.3292
2     72    73.5000
     120    73.5000
     385    73.5000
     655    73.5000
     665    73.5000
3    159    69.5500
    180    69.5500
    201    69.5500
    324    69.5500
    792    69.5500
    846    69.5500
    863    69.5500
Name: Fare, dtype: float64
```

A6.3

Quantile Info

quantile()

Die Methode `quantile()` in Pandas berücksichtigt keine NaN (Null)-Werte bei der Berechnung. Das bedeutet, dass NaN-Werte standardmäßig ausgeschlossen werden.

Anzahl von Zeilen berechnen:

df.size()

Bsp.:

```
Nach der Gruppierung berechnet diese Methode die Größe jeder Gruppe, d.h. die Anzahl der Zeilen in `t`, die zu jeder eindeutigen Kombination von `Ticket` und `Cabin` gehören.
```

```
s_t = t.groupby(by=["Ticket", "Cabin"]).size()
```

```
Ticket          Cabin
110152          B77     2
                  B79     1
110413          E67     2
                  E68     1
110465          A14     1
                  ..
SC/AH Basle 541  D      1
SC/Paris 2163   D      1
SOTON/0.0. 392078 E10     1
W.E.P. 5734    E31     1
WE/ZP 5735    B22     2
Length: 161, dtype: int64
```

A6.6

```
In SQL wird das HAVING-Schlüsselwort verwendet, um Bedingungen nach einer Aggregation anzuwenden. In Pandas können Sie eine ähnliche Funktionalität erreichen, indem Sie die Methode groupby() verwenden, gefolgt von einer
```

Aggregationsfunktion und dann die resultierenden Gruppen mit einer Bedingung filtern.

Zusammenfassend lässt sich sagen, dass Sie in Pandas das HAVING-Schlüsselwort aus SQL durch die Kombination von groupby(), agg() und Bedingungen mit loc[] abbilden können.

A 7

Mergen von zwei dfs

```
`df1.merge(df2, how='Join Art zb 'inner'', on='gemeinsame Spalte zB 'id'')`
```

Gibt Index des höchsten Wertes: `.argmax()`

Bsp.

```
max_budget = op_grouped.sum().argmax()
```

Ersetzen von NaN Werten :

```
df[['Embarked']].fillna('Kein Hafen', inplace=True)
```

Überprüfen ob nicht NaN

```
Überprüfen ob es kein NaN Wert ist: `pd.notna(`a)`
```

Bsp.

```
t['Age'].apply(lambda a: print(f'Ich bin {a} alt') if pd.notna(a) else print(f'Kein Alter'))
```

Überprüfen ob NaN

```
pd.isnan(Row); df.isna()
```

Kap.2 Aufgabe 1

Lamda mit mehreren if's :

```
`lambda c: 'F' if c == 'female' else('M' if c== 'male' else 'X')`
```

Bins setzen:

```
Pd.qcut()
```

Bsp.:

```
quantile_labels = ['niedrig', 'normal', 'hoch', 'sehr hoch']
t['Quantil'], grenzen = pd.qcut(t['Fare'], q=4,
labels=quantile_labels, retbins=True)
```

```
retbins=True → Grenzen der quantile
duplicates='drop' → Duplikate entfernen
```

A2

Embarked Unique Werte aus df in anderen df mappen

```
embarked_names = list(t2["Joined"].unique())
embarked_names
```

```
def hafen_setzten(c):
    if c == embarked_names[0][0]: return embarked_names[0]
    if c == embarked_names[1][0]: return embarked_names[1]
    if c == embarked_names[2][0]: return embarked_names[2]
    if c == embarked_names[3][0]: return embarked_names[3]
```

```
t['Embarked'] = t['Embarked'].map(hafen_setzten)
```

Gleitkommazahl auf zweite Ziffer

```
t['Fare'] = t['Fare'].map(lambda f: round(f, 2))
```

A4

Funktion mit if Verkettung

```
def alone(c):
    if c['Parch'] > 0 or c['SibSp'] > 0: return 0
    if c['Parch'] == 0 or c['SibSp'] == 0: return 1
```

A5

Seaborn (import seaborn as sns)

Copy_on_write= True gibt manchmal einen Error → False einstellen

Histplot() → Histogramm

```
sns.histplot(data=t, x=t['Age'], bins=10)
```

bins für die Anzahl der Balken

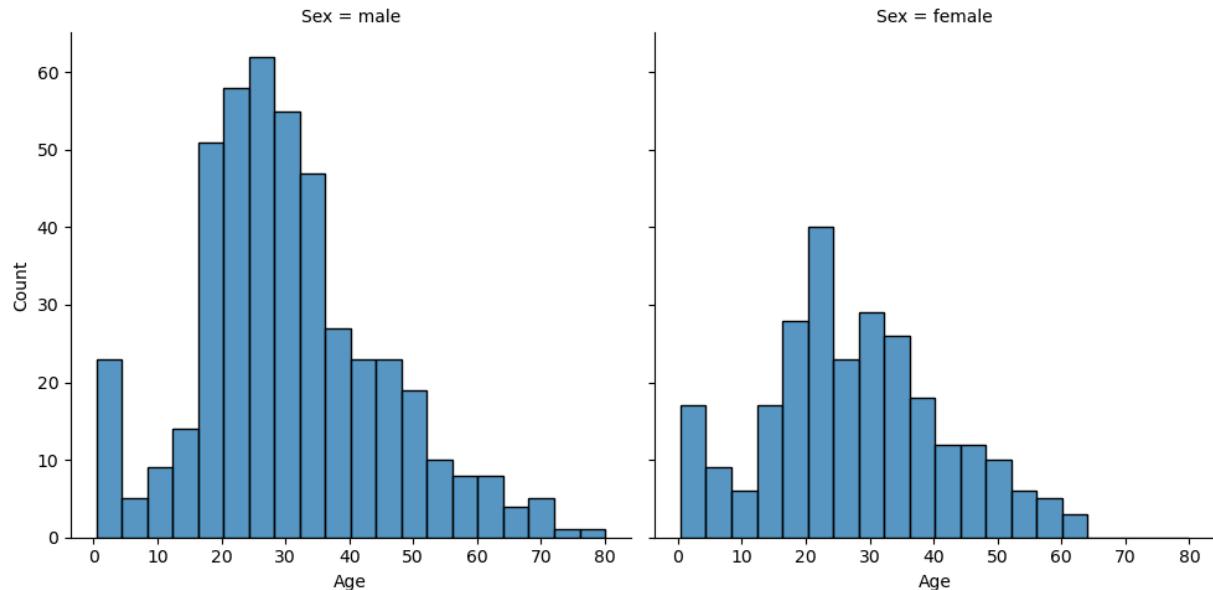
KDE (Kernel Density Estimate) auf False gesetzt

Hue='Age' → Farbe von Plot

`Displot()` → Mehrere Histogramme in einer Ausgabe

```
sns.displot(data=t, x='Age', col='Sex')
```

`col='Sex'` → Einteilen der zwei Histogramme in männlich und weiblich

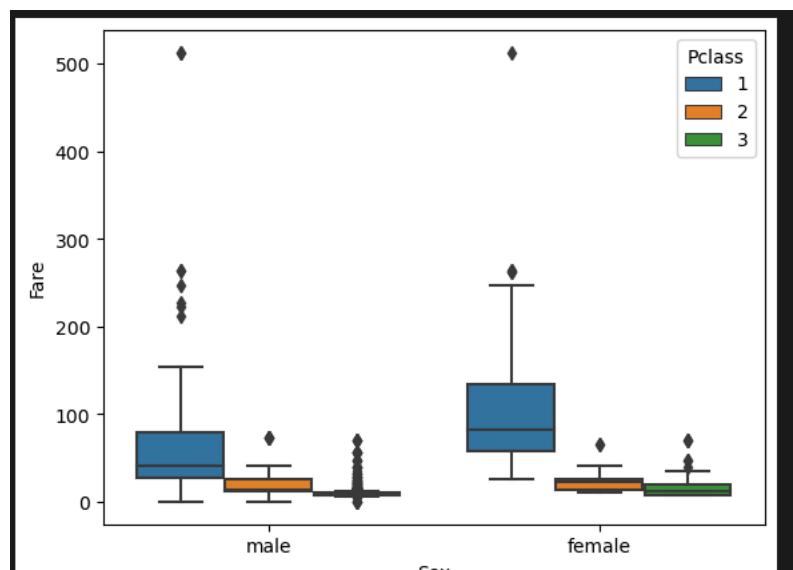


Boxplot erstellen

```
sns.boxplot()
```

Bsp.

```
sns.boxplot(data=t, x='Sex', y='Fare', hue='Pclass')
```



A7

komplexe Transformationen auf Daten in einem DataFrame anzuwenden, oft in Verbindung mit der groupby-Methode.

```
group.transform('median')
```

A9

Nach Anfangsbuchstaben extrahieren:

```
`col.startswith("A")`
```

Spalten löschen:

```
`b.drop(`Spalte`)"`
```

Datum ISO Format:

```
`pd.to_datetime(`b["Spalte"]`)`
```

```
w['Date'] = pd.to_datetime(w[['year', 'month', 'days']])
```

```
w['Date'] = pd.to_datetime(w['year'].astype(str) + '-' + w['month'].astype(str) + '-' + w['days'].astype(str))
```

Mit Zeiträumen rechnen:

```
`pd.to_timedelta(`b['Woche']` , unit='w')` <br> UNITS:
```

M - Monate,

Y - Jahre,

w - Wochen,

d - Tage,

h - Stunden,

m oder min - Minuten,

s - Sekunden

Bsp. :

```
b['end-date'] = pd.to_datetime(b['start-date']) + pd.to_timedelta(b['Woche'] ,  
unit='w')
```

Long Format in wide Format → Geht von "long" zu "wide". Mehr Spalten, weniger Zeilen. Jede Kategorie bekommt ihre eigene Spalte.

```
df.pivot() / df. pivot_table()
```

Bsp:

```
df.pivot_table(index='name', columns='property', values='values')
```

	name	property	values	property	age	height
0	Philipp Woods	age	45			
1	Philipp Woods	height	186	name		
2	Philipp Woods	age	50	Jessica Cordero	37.0	156.0
3	Jessica Cordero	age	37	Philipp Woods	47.5	186.0
4	Jessica Cordero	height	156			

Wide Format in Long Format → Geht von "wide" zu "long". Weniger Spalten, mehr Zeilen. Kategorien werden in einer Spalte gestapelt.

df.melt()

Bsp.:

```
df_tidy = df_wide.melt(id_vars=['pregnant'], var_name='gender', value_name='count')
```

```
pandas.melt(frame, id_vars=None, value_vars=None, var_name=None,
value_name='value', col_level=None)
```

- **frame**: Der DataFrame, den du umformen möchtest.
- **id_vars**: Die Spalte(n) des DataFrames, die als Identifier dienen und nicht umgeformt werden sollen.
- **value_vars**: Die Spalten, die umgeformt werden sollen. Wenn nicht angegeben, werden alle Spalten, die nicht **id_vars** sind, umgeformt.
- **var_name**: Der Name der neuen Spalte, die die Spaltennamen der **value_vars** enthält. Wenn nicht angegeben, wird der Name **variable** verwendet.
- **value_name**: Der Name der neuen Spalte, die die Werte enthält. Wenn nicht angegeben, wird der Name **value** verwendet

Einzigartige Werte in einer Spalte

set()

Range of values

Simple trick to find out all occurring values in a column: convert the column to a Python **set**:

```
set(titanic["Sex"])
{'female', 'male'}
```

```
set(titanic["Embarked"])
{'C', 'Q', 'S', nan}
```

Anzahl einzigartiger Werte
Value_counts()

```
titanic["Sex"].value_counts()

male      577
female    314
```

Listenwerte in einer Zeile in einzelne Werte in mehreren Zeilen reinschreiben.
Explode() Vertikal

3c) A list of values in one column (explode vertically)

In this situation, the column containing a list models a 1:n relationship, e.g. student applied for a course.

First, we need to transform the string with commas into a real list:

The diagram illustrates the transformation of a single row with a list in the 'course' column into multiple rows for each course in the list. It shows three stages of data transformation:

- Initial Data:** A single row with columns 'matrikel', 'name', and 'course'. The 'course' column contains a list: [ERP, DB1, Prog].
- Intermediate Step:** The 'course' column is modified using the code: `students["course_list"] = students["course"].str.split(',')`. The 'course' column is now empty, and a new column 'course_list' is added, containing the list [ERP, DB1, Prog].
- Final Result:** The data is exploded into multiple rows. For each entry in the 'course_list' column, a new row is created with the same values for 'matrikel' and 'name', and the corresponding course value from the list. The 'course' column is now empty, and the 'course_list' column contains the individual course names: [ERP], [DB1], and [Prog].

Then, we can explode the list into a separate row for each entry in the list:

```
(  
    students.explode("course_list")  
    .drop(columns="course")  
    .rename(columns={"course_list" : "course"})  
)  
  
.dropna()  
.groupby(["matrikel", "name"]).agg(course=("course", ', '.join))
```

2 Data preparation

The final transformed data frame has columns 'matrikel', 'name', and 'course'. It contains 6 rows, with the last row (index 3) having a value of '<NA>' in the 'course' column. The 'course' column values are: ERP, DB1, Prog, DB2, lazy, and crazy.

© Prof. Dr. K. Verbarg

44

Explode() Horizontal

3d) explode a list horizontally

Alternatively, we can "explode" the list horizontally,
i.e. make a new column for each list element

	matrikel	name	course	course_list
0	123	freak	ERP, DB1, Prog	[ERP, DB1, Prog]
1	42	doe	DB2	[DB2]
2	888	lazy		[]
3	999	crazy	<NA>	<NA>

```
students.merge(  
    students["course_list"].apply(pd.Series), fillna=pd.NA)  
, left_index=True, right_index=True  
)
```

	matrikel	name	course	course_list	0	1	2
0	123	freak	ERP, DB1, Prog	[ERP, DB1, Prog]	ERP	DB1	Prog
1	42	doe	DB2	[DB2]	DB2	<NA>	<NA>
2	888	lazy		[]		<NA>	<NA>
3	999	crazy	<NA>	<NA>	<NA>	<NA>	<NA>

Verkettungen

No grouping – do aggregation over all rows

Use the aggregation functions on a column

```
(  
    titanic["Age"].mean().round(1),  
    titanic["Survived"].sum()  
)
```

(29.7, 342)

```
SELECT AVG("Age"),  
       SUM("Survived")  
FROM "titanic";
```

SQL

Warning: check whether or not the desired function is defined on your pandas-object (SeriesGroupBy, DataFrame,...).

Make a chain or pipeline

To avoid this, we can chain all operations in one big step.

- We need to be able to write each operation as a function.
- We use brackets to be able to put each function into a separate line.

```
itanic.csv"
df = [
    pd.read_csv(url)
    .convert_dtypes()
    .rename(columns={"Pclass": "Klasse", "Age": "Alter"})
    .drop(columns="PassengerId")
    .set_index(keys=["Cabin", "Name"])
]

SibSp  Parch
```

assign das „weiche“ apply. Gut geeignet für Verkettung

Extended projection

Alternatively, use `assign()`:

```
df.assign(mortality = df["death"] / df["cases"] * 100)

  year state cases death mortality
0 2000    MV      0     0       NaN
1 2001    MV      1     0  0.000000
2 2002  Saxen     55     7  12.727273
3 2000  Bayern    33     4  12.121212
```

Rename a column

```
df.rename(columns={"mortality": "m", "year": "y"})

      y state cases death      m
0  2000    MV      0     0       NaN
1  2001    MV      1     0  0.000000
2  2002  Saxen     55     7  12.727273
3  2000  Bayern    33     4  12.121212
```

Remember that the original DataFrame `df` is not modified.

Con: Only simple identifiers are possible.

Pro: It can be chained with other transformations of a DataFrame. Multiple calculated columns are possible in one command (like in SQL).

```
SELECT "year" AS "y",
       state,
       cases,
       death,
       "mortality" AS "m"
FROM df;
SQL
```

Make a chain or pipeline

To avoid this, we can chain all operations in one big step.

- We need to be able to write each operation as a function.
- We use brackets to be able to put each function into a separate line.

```
itanic.csv"
df = (
    pd.read_csv(url)
    .convert_dtypes()
    .rename(columns={"Pclass" : "Klasse", "Age" : "Alter"})
    .drop(columns="PassengerId")
    .set_index(keys=["Cabin", "Name"])
)

SibSp  Parch
```

Bsp.:

```
( df.loc[df["year"] == 2000]
    .assign(incidence = df["cases"] / df["population"] * 1_000_000)
    [ ["country", "incidence"]]
)
✓ 0.9s
```

	country	incidence
1	Afghanistan	129.446633
3	Brazil	461.236337
5	China	166.948788

Python Umgebung copy_on_write

Shallow vs. deep copy (SettingWithCopyWarning)

Achtung shallow oder deep copy

Bei einer shallow copy warnt ggf. IPython, wenn man anschließend eine Zuweisung macht. Diese wird dennoch ausgeführt und verändert dann auch df.

```
shallow = df[["state", "cases"]]
shallow["cases"] += 100
<ipython-input-77-9e91dec35f3>:1: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead
See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/user_guide/indexing.html#why-does-assignment-fail-when-using-chained-indexing
df
```

	state	cases	death
0	MV	100	0
1	MV	101	0
2	Sachsen	155	7
3	Bayern	133	4

Simple assignment is doing only a shallow copy (= reference = a view).

So sometimes, the outcome (view or copy) is unpredictable and that is the reason for the SettingWithCopyWarning.

https://pandas.pydata.org/pandas-docs/stable/user_guide/indexing.html#why-does-assignment-fail-when-using-chained-indexing

Normally, we don't have to worry too much about it. Usually, we will subsequently assign intermediate results to new variables and not tamper with "old" variables any more.

To avoid the warning,
do an explicit
deep copy earlier in your code.

```
deep = df[["state", "cases"]].copy()
deep["cases"] += 2000
df
```

	state	cases	death
0	MV	100	0
1	MV	101	0
2	Sachsen	155	7
3	Bayern	133	4

NEW: Always do a **Copy on Write**

<https://phofl.github.io/cow-introduction.html>

```
pd.set_option("mode.copy_on_write", True)
```

Errors

Loc[] Error Slicing

For a MultiIndex having both, row- and column-indexer gives an error (two many commas):

```
gbi.loc['2007-01-01', 17000, ['PRTR1000','DXTR1000'], ["City", "Country"]]  
IndexError: list index out of range
```

This does not work in combination with slicing:

```
: gbi.loc['2007-01-01', 17000:18000, ['PRTR1000','DXTR1000']], ["City", "Country"]  
File "<ipython-input-51-99581a08af90>", line 1  
    gbi.loc['2007-01-01', 17000:18000, ['PRTR1000','DXTR1000']], ["City", "Country"]  
SyntaxError: invalid syntax
```

The elements of a MultiIndex are Tuples: so we better write it with parenthesis:

```
gbi.loc[('2007-01-01', 17000, ['PRTR1000','DXTR1000']), ["City", "Country"]]
```

Date	Customer	Product	City	Country
2007-01-01	17000	PRTR1000	Hannover	DE
		DXTR1000	Hannover	DE

Solution for MultiIndex and slicing at the same time: use "pd.IndexSlice":

```
gbi.loc[pd.IndexSlice['2007-01-01', 17000:18000, ['PRTR1000','DXTR1000']], ["City", "Country"]]  
Date Customer Product City Country  
2007-01-01 17000 PRTR1000 Hannover DE  
DXTR1000 Hannover DE
```

Werte sortieren

```
tips.sort_index(inplace=True)  
tips.head(4)
```

	total_bill	tip	sex	smoker	day	time	size
0	16.99	1.01	Female	No	Sun	Dinner	2
1	10.34	1.66	Male	No	Sun	Dinner	3
2	21.01	3.50	Male	No	Sun	Dinner	3
3	23.68	3.31	Male	No	Sun	Dinner	2
4	24.59	3.61	Female	No	Sun	Dinner	4

or using some columns (also works for the index)

```
tips.sort_values(by=["total_bill", "sex"], ascending=[True, False], inplace=True)  
tips  
total_bill tip sex smoker day time size  
67 3.07 1.00 Female Yes Sat Dinner 1  
92 5.75 1.00 Female Yes Fri Dinner 2  
172 7.25 5.15 Male Yes Sun Dinner 2  
111 7.25 1.00 Female No Sat Dinner 1  
149 7.51 2.00 Male No Thur Lunch 2  
... ... ... ... ... ... ...  
T TOP 5 *  
SELECT * FROM tips  
ORDER BY "total_bill", "sex" DESC;  
SAP HANA SQL
```

Keys von einer Gruppe .groups().keys()

Grouping

The simplest form of grouping is done by specifying the grouping attribute (the "keys"):

```
grouped = t.groupby(by=['Pclass', 'Sex'])  
grouped  
#pandas.core.groupby.generic.DataFrameGroupBy object at 0x0000020288033190  
This returns a pandas "GroupBy" object.  
It has the groups calculated. For each group, it contains a DataFrame with all (!) attributes of the original.  
Sorting within the groups is preserved.  
You can display this with the groups-Attribute: it is a Dictionary of the grouping keys and the indexes of that group.  
grouped.groups
```

```
((1, 'female'): [1, 3, 11, 31, 52, 61, 88,  
307, 309, 310, 311, 318, 319, 325, 329, 31,  
7, 549, 540, 556, 558, 571, 577, 581, 584]
```

That helps to output the groups alone:

```
grouped.groups.keys()  
dict_keys([(1, 'female'), (1, 'male'), (2, 'female'), (2, 'male'), (3, 'female'), (3, 'male')])
```

We can use **projection** on all grouping and non-grouping attributes. (We use head() to display something for each group.)

```
grouped[['Pclass', 'Sex', 'Age', 'Cabin', 'Name']].head(1)  
Pclass Sex Age Cabin Name  
0 3 male 22.0 NaN Braund, Mr. Owen Harris  
1 1 female 38.0 C/S Cumings, Mrs. John Bradley (Florence Briggs Th...  
2 3 female 26.0 NaN Heikkinen, Miss. Laina  
6 1 male 54.0 E46 McCarthy, Mr. Timothy J  
9 2 female 14.0 NaN Nasser, Mrs. Nicholas (Adele Achem)  
17 2 male NaN NaN Williams, Mr. Charles Eugene
```

```
len(grouped)  
6
```

Andere Funktionen

Any(axis=0/1)

Die Methode `any()` in Pandas wird verwendet, um zu überprüfen, ob mindestens ein Element in einer Achse (entweder in einer Reihe oder Spalte eines DataFrame oder in einer Serie) wahr ist.

```
regen_vorhanden = wetter['Regen'].any()
```

Groups

```
spezifische_gruppe = gruppierung.get_group(270)
```

Aggregationen

Aggregation

The size of each group (including Nulls)

```
grouped.size()
```

```
Pclass  Sex
1    female    94
     male     122
2    female    76
     male     108
3    female   144
     male     347
```

```
SELECT "Pclass", "Sex", COUNT(*)
FROM "titanic"
GROUP BY "Pclass", "Sex";
SQL
```

The number of non-Null values for "Age" of each group

```
grouped["Age"].count()
```

```
Pclass  Sex
1    female    85
     male     101
2    female    74
     male     99
3    female   102
     male     253
```

```
SELECT "Pclass", "Sex", COUNT("Age")
FROM "titanic"
GROUP BY "Pclass", "Sex";
SQL
```

Some standard aggregation methods:

size	Number of rows in group
count	Number of non-Null values
sum	Sum of non-Null values
mean	Average of non-Null values
median	Median of non-Null values
min, max	Minimum, maximum of non-Null values
nunique	Number of distinct values

Aggregation mit Verkettung

No grouping – do aggregation over all rows

Use the aggregation functions on a column

```
(  
    titanic["Age"].mean().round(1),  
    titanic["Survived"].sum()  
)
```

(29.7, 342)

```
SELECT AVG("Age"),  
       SUM("Survived")  
FROM "titanic";
```

SQL

Warning: check whether or not the desired function is defined on your pandas-object (SeriesGroupBy, DataFrame,...).

Grouped Aggregation

```
Group[, 'Fare'].sum()
```

```
grouped.gg()
```

Bsp.:

```
tt_gclass[['Fare', 'Age']].agg(['mean', 'std'])
```

To have more control and do multiple aggregations at the same time, the following syntax helps:

```
grouped.agg(
    nof_persons = ("Survived", "size"),
    nof_survived = ("Survived", "sum"),
    avg_fare = ("Fare", "mean"),
    min_fare = ("Fare", "min")
).assign(survival_rate = lambda grp: grp["nof_survived"] / grp["nof_persons"])
```

Pclass	Sex		nof_persons	nof_survived	avg_fare	min_fare	survival_rate
1	female		94	91	106.125798	25.9292	0.968085
	male		122	45	67.226127	0.0000	0.368852
2	female		76	70	21.970121	10.5000	0.921053
	male		108	17	19.741782	0.0000	0.157407
3	female		144	72	16.118810	6.7500	0.500000
	male		347	47	12.661633	0.0000	0.135447

Quantile

Kap.2 A6

```
fare_95 = t['Fare'].quantile(0.95)
```

```
t['QuantileFare'], grenzen= pd.qcut(t['Fare'], q=4, retbins=True)
```

```
temperatur_50_quantil = wetter['Temperatur'].quantile(0.50)
```

Bins

Funktion einteilen bins

```
def alter_bins(alter):
    if pd.isna(alter): return 'kein Alter'
    if alter < 11: return 'child'
    if alter < 18 and alter >= 12: return 'teenager'
    if alter < 66 and alter > 17: return 'adult'
    if alter > 65: return 'adult'
```

None

Pd.Nan is the way

The truth about pd.NA

There is a bunch of values, which might serve as a database NULL equivalent:

- o The Python None value
- o In NumPy there is the notion of a not-a-number np.NaN. This is a float datatype and cannot be used across all datatypes. It was used in pandas in the past.
- o Pandas introduced a new value pd.NA

We recommend using pd.NA only (displayed as "<NA>" / "NaN" / "NA").

```
import numpy as np

df = pd.DataFrame({'col': [1, "text", None, pd.NA, np.NaN]})
```

col	col
0 1	0 False
1 text	1 False
2 None	2 True
3 <NA>	3 True
4 NaN	4 True

1 Python pandas

```
None == None
```

True

```
not(np.NaN == np.NaN)
```

True

In contrast, pd.NA implements the expected logic:

```
pd.NA == pd.NA
```

<NA>

```
pd.NA is pd.NA
```

True

```
1 + pd.NA
```

<NA>

```
1 > pd.NA
```

<NA>

```
pd.NA & True
```

<NA>

```
pd.NA | True
```

True

© Prof. Dr. K. Verbarg

53

Problems with pd.NA

Python bool only has the values True / False

```
: s = pd.Series([1,2,pd.NA])
s == 2

: 0    False
 1    True
 2    False
dtype: bool

: s == pd.NA
```

Even worse:

```
if(pd.NA < 13):
    print('kleiner')
else:
    print('groß')

TypeError: boolean value of NA is ambiguous
```

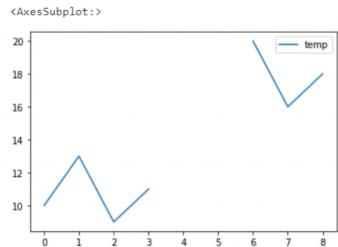
Solution: explicitly check for Null-values using pd.isna()

Interpolieren

Interpolate missing values

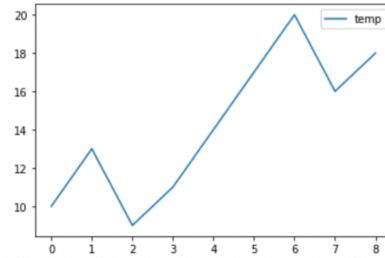
Missing values can be linear interpolated:

```
df = pd.DataFrame({"temp": [10, 13, 9, 11, None, None, 20, 16, 18]}, dtype="float64")  
df.plot()
```



```
df.interpolate().plot()
```

<AxesSubplot:>



Interpolation can also be axis-aware (for time series) or use various methods (e.g. spline, quadratic, piecewise polynomial).

String Operationen Series

```
titanic["Name"].str.lower()
```

```
0          braund, mr. owen harris  
1    cumings, mrs. john bradley (florence briggs th...  
2          heikkinen, miss. laina  
3    futrelle, mrs. jacques heath (lily may peel)  
4          allen, mr. william henry
```

String vector operations

To find available string operations, check the pandas documentation under `pandas.Series.str`.

This is an incomplete overview.

We start with simple methods which usually originate from a Python scalar pendant (example: `str.lower`).

How is Null-handling different for vector methods compared to element-wise
`.map()`?

<code>len</code>	Length of string (or tuple or list)
<code>lower, upper, title,</code> <code>capitalize, swapcase</code>	Modify the case
<code>isalnum, isalpha, isdigit,</code> <code>isspace, istitle, islower,</code> <code>isupper, isnumeric,</code> <code>isdecimal</code>	Returns True / False based on the checked condition
<code>startswith, endswith</code>	
<code>strip, lstrip, rstrip</code>	Remove trailing whitespace
<code>pad</code>	Pad with fill character
<code>translate</code>	Map characters based on mapping table
<code>find, rfind, index,</code> <code>rindex, slice</code>	Find substring
<code>split</code>	Split string at separator (<code>expand=True</code> makes df-columns instead of list)
<code>join</code> <code>cat</code>	Concatenate strings in list Concatenate strings of two Series
<code>repeat</code>	Duplicates values: <code>Series.str.repeat(3)</code> is equivalent to <code>x * 3</code>
<code>wrap</code>	Wrap long texts into lines with maximal length

Ersten Buchstaben extrahieren:

```
t['Cabin'].str.get(0)
```

Einsatz von Regular Expressions (re regex)

Import re

Grundlegende Metacharaktere:

- `.` : Passt auf jedes einzelne Zeichen außer auf Zeilenumbrüche.
- `^` : Passt am Anfang eines Strings oder einer Zeile.
- `$` : Passt am Ende eines Strings oder einer Zeile.
- `*` : Passt auf 0 oder mehr Wiederholungen des vorhergehenden Ausdrucks.
- `+` : Passt auf 1 oder mehr Wiederholungen des vorhergehenden Ausdrucks.
- `?` : Macht den vorhergehenden Ausdruck optional (0 oder 1 Wiederholungen).
- `{n}` : Passt genau n Wiederholungen des vorhergehenden Ausdrucks.
- `{n,m}` : Passt n oder mehr Wiederholungen des vorhergehenden Ausdrucks.
- `{,m}` : Passt bis zu m Wiederholungen des vorhergehenden Ausdrucks.
- `{n,m}` : Passt zwischen n und m Wiederholungen des vorhergehenden Ausdrucks.
- `|` : Entspricht einem logischen ODER (matcht entweder den Ausdruck davor oder den danach).

Zeichenklassen:

- `\d` : Passt auf eine Ziffer (äquivalent zu [0-9]).
- `\D` : Passt auf jedes Nicht-Ziffer-Zeichen (entspricht [^0-9]).
- `\w` : Passt auf jedes alphanumerische Zeichen inklusive Unterstrich (äquivalent zu [a-zA-Z0-9_]).
- `\W` : Passt auf alles, was nicht `\w` ist (entspricht [^a-zA-Z0-9_]).
- `\s` : Passt auf jedes whitespace Zeichen (inklusive Leerzeichen, Tabs und Zeilenumbrüche).
- `\S` : Passt auf jedes Nicht-whitespace Zeichen.
- `,` : Komma
- `.` : Punkt

Grenzen:

- `\b` : Passt auf eine Wortgrenze (zwischen einem `\w` und einem `\W`).
- `\B` : Passt auf eine Nicht-Wortgrenze.

Sonstige Ausdrücke:

- `[]` : Ein Set von Zeichen. Alles innerhalb der Klammern wird als eine Menge von Zeichen betrachtet.
- `[^]` : Negation. Passt auf alles, was nicht in den Klammern steht.
- `()` : Definiert eine Gruppe, um Teilreguläre Ausdrücke zu kombinieren.

\: Wird verwendet, um spezielle Zeichen zu entwerten oder spezielle Sequenzen zu signalisieren.

Spezielle Sequenzen:

\A : Passt nur am Anfang des Strings.

\Z : Passt nur am Ende des Strings.

\G : Passt auf die Position, an der das vorherige Such- oder Match-Ende angegeben wurde.

The Python `re`-module provides methods for regular expressions.

First, define a regular expression (a pattern):

```
import re

text = ' foo bar      bar\tbaz  \n\t\nabc'
regex = re.compile(r'\s+') # one or more whitespace chars
```

Then, use it to analyse or modify the text:

```
regex.split(text)
```

```
['', 'foo', 'bar', 'bar', 'baz', 'abc']
```

Capture Groups
markiert durch
Klammern (Muster)

```
regex.findall(text)
```

```
[' ', ' ', ' ', ' ', '\t', ' ', '\n\t\n']
```

```
print(regex.match(text))
```

```
<re.Match object; span=(0, 2), match=' '>
```

```
regex.sub('*', text)
```

```
'*foo*bar*bar*baz*abc'
```

Ersten Char entfernen

(d1) -> (1)

```
w_melted['days'] = w_melted['days'].str.extract('(\d+)').astype(int)
```

Vergleicht ob enthalten

```
str.contains(r'StringVektorOperation')
```

Bsp

```
ok = t['Cabin'].notna() & t["Cabin"].str.contains(r'^\w\d+$')
t.loc[~ok]["Cabin"].shape # Anzahl False Werte
```

Nach einem Wort

```
filter_sonne = wetter[wetter['Bemerkungen'].str.contains(r'Sonne')]
```

\^: Dieses Zeichen markiert den Anfang eines Strings. Es bedeutet, dass das Muster, das wir suchen, am Anfang des Strings stehen muss.

\w: Dies steht für "word character" und entspricht in der Regel jedem Buchstaben oder Unterstrich (_). Es beinhaltet alle alphanumerischen Zeichen (Buchstaben und Ziffern) sowie Unterstriche.

\d: Dies steht für "digit" und entspricht jeder Ziffer von 0 bis 9.

\+: Dieser Quantifizierer bedeutet "eins oder mehr" des vorhergehenden Elements. In diesem Fall bedeutet **\d+**, dass eine oder mehrere Ziffern folgen müssen.

\\$: Dieses Zeichen markiert das Ende eines Strings. Es bedeutet, dass das Muster, das wir suchen, am Ende des Strings stehen muss.

```
t['Nachname'] = t['Name'].str.extract(r'^(\w+,\w+)$')
```

Braund, Mr. Owen Harris → Braund

- **\^** ist ein Anker in regulären Ausdrücken, der den Beginn eines Strings angibt.
- **(** und **)** definieren eine Gruppe, die man extrahieren möchte. Alles, was in diesen Klammern gefunden wird, wird als gefundene Gruppe zurückgegeben.
- **[^,]** ist eine negierte Zeichenauswahl, die jedes Zeichen außer einem Komma **,** entspricht.
- **\+** ist ein Quantifizierer, der angibt, dass das vorherige Zeichen oder die Zeichengruppe ein- oder mehrmals wiederholt werden kann.

Titel Extrahieren

```
t['Name'].str.extract(r'(\w+\.\w+.)')
```

Gleikommazahl mit zwei nachkommastellen

```
match=re.search(r'\d\.\d{2}', str(zahl))
```

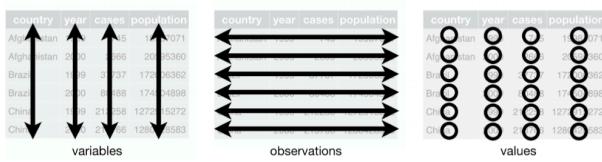
```
if|match|...
3,5464 → 3,54
```

Tidy Data Definition

Definition tidy data

A dataset is **tidy**, if:

1. Each type of observational unit forms a table.
2. Each observation forms a row.
3. Each variable must have its own column.
Sort variables: fixed variables (known in advance) first, followed by measured variables.
4. Each value must have its own cell.



Advantages of tidy data:

- It's easier to learn the commands and work with data structured in a consistent way.
- Leverage vectorized operations with good performance
- Standardize the interface to further processing

How do these rules compare to

- decomposition rules in relational databases?
- the dimensional model for data warehouses?

Synonyms for tidy data:

- **long-form data** (as opposed to wide-form data—or, a crosstab where both, columns and rows, contain levels of a dimension)
- **key figure model** or measure-based model (as opposed to account model with two column, key figure name and value)

References:

Book "R for Data Science", Chapter 12

<https://r4ds.had.co.nz/tidy-data.html#fig:tidy-structure>

Hadley Wickham, "Tidy Data", Journal of Statistical Software, August 2014, volume 59, Issue 10.

<http://dx.doi.org/10.18637/jss.v059.i10>

Tidy Data Long and Wide Wide melt()

Raw data (pew.csv)

religion	<\$10k	\$10-20k	\$20-30k	\$30-40k	\$40-50k	\$50-75k	\$75-100k	\$100-150k	>150k	Don't know/refused
Agnostic	27	34	60	81	76	137	122	109	84	96
Atheist	12	27	37	52	35	70	73	59	74	76
Buddhist	27	21	30	34	33	58	62	39	53	54
Catholic	418	617	732	670	638	1116	949	792	633	1489
Don't know/refused	15	14	15	11	10	35	21	17	18	116
Evangelical Prot	575	869	1064	982	881	1486	949	723	414	1529
Hindu	1	9	7	9	11	34	47	48	54	37
Historically Black Prot	228	244	236	238	197	223	131	81	78	339
Jehovah's Witness	20	27	24	24	21	30	15	11	6	37
Jewish	19	19	25	25	30	95	69	87	151	162

Tidy

religion	income	count
Agnostic	<\$10k	27
Atheist	<\$10k	12
Buddhist	<\$10k	27
Catholic	<\$10k	418
Don't know/refused	<\$10k	15
...

```
pew.melt(id_vars="religion", var_name="income", value_name="count")
```

We have three variables: religion, income, number of people in the survey.

We have to melt everything except religion.
We rename the new columns (income and count).

Long pivot()

2) Variable names in the data cells

Account model

This is just the opposite of case 1)

country	year	key	value
Afghanistan	1999	cases	745
Afghanistan	1999	population	19987071
Afghanistan	2000	cases	2666
Afghanistan	2000	population	20595360
Brazil	1999	cases	37737
Brazil	1999	population	172006362
Brazil	2000	cases	80488
Brazil	2000	population	174504898
China	1999	cases	212258
China	1999	population	1272915272
China	2000	cases	213766
China	2000	population	1280428583

Tidy

We have two key figures (cases and population).

country	year	key	cases	population
Afghanistan	1999	cases	745	19987071
	2000	cases	2666	20595360
Brazil	1999	cases	37737	172006362
	2000	cases	80488	174504898
China	1999	cases	212258	1272915272
	2000	cases	213766	1280428583

`account.pivot(index=["country", "year"], columns="key", values="value")`



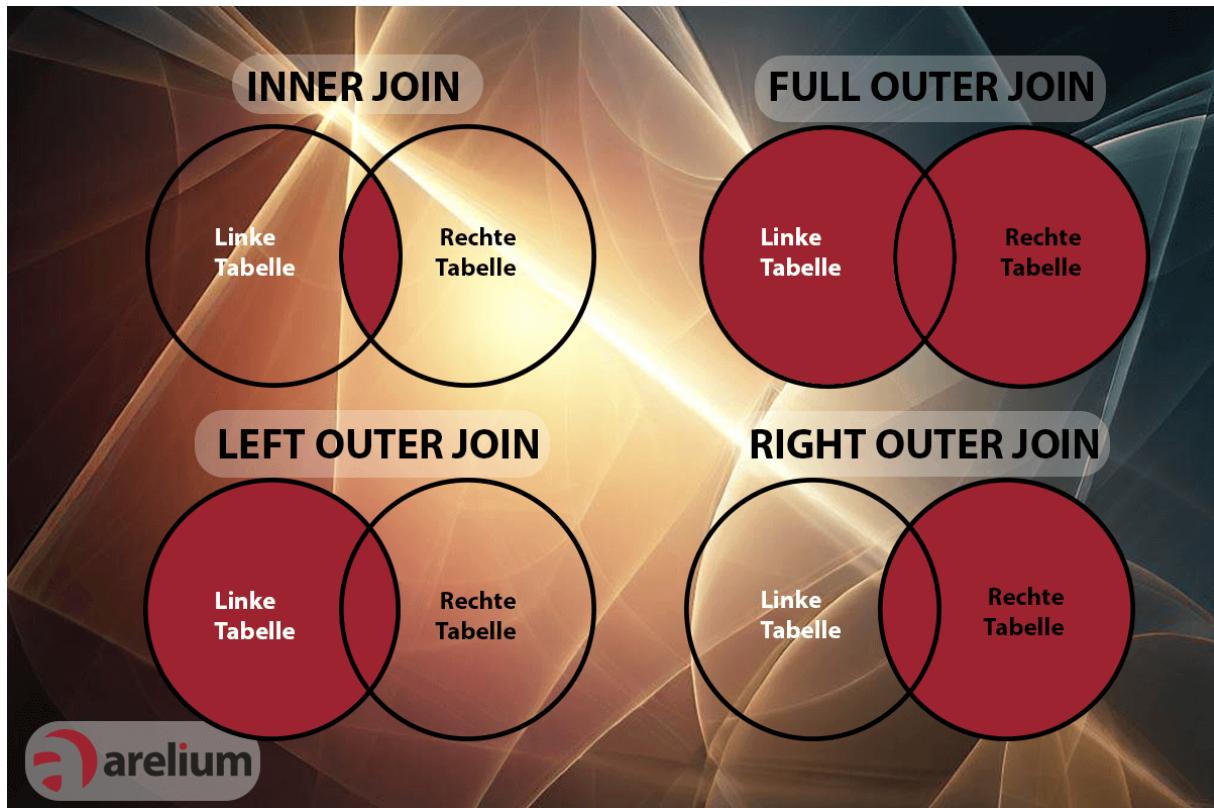
Bsp mit aggregation

```
pivot_tabelle = wetter.pivot_table(values=['Temperatur', 'Taupunkt'],
index='Stunde', aggfunc='mean')
```

Stunde	Taupunkt	Temperatur
1	-2.0	-0.6
2	-2.0	-0.8
3	-2.0	-0.7
4	-2.0	-0.4
5	-2.0	-0.3
6	-2.0	-0.3
7	-1.0	0.6
8	0.0	1.3
9	0.0	1.9
10	1.0	2.3
11	1.0	2.4
12	1.0	3.1
13	2.0	3.4
14	2.0	3.5

```
pivot_luftdruck_windrichtung = wetter.pivot_table(values='Luftdruck',
index='Windrichtung', aggfunc=['max', 'min', 'mean'])
```

Join Arten



1. ****Inner Join**:** `how='inner'
- Standardmäßig verwendet `merge` den Inner Join, wenn nichts anderes angegeben wird.
- Beispiel: `pd.merge(df1, df2, on='Schlüsselspalte', how='inner')`
2. ****Outer Join (Full Outer Join)**:** `how='outer'
- Beispiel: `pd.merge(df1, df2, on='Schlüsselspalte', how='outer')`
3. ****Left Join (Left Outer Join)**:** `how='left'
- Beispiel: `pd.merge(df1, df2, on='Schlüsselspalte', how='left')`
4. ****Right Join (Right Outer Join)**:** `how='right'
- Beispiel: `pd.merge(df1, df2, on='Schlüsselspalte', how='right')`

In jedem Fall ersetzen Sie `df1` und `df2` mit den Namen Ihrer DataFrames und `Schlüsselspalte` mit dem Namen der Spalte(n), nach der/denen gejoint werden soll.

Vektor Operationen

Arithmetische Operationen

Addition: `+`

Subtraktion: `-`

Multiplikation: `*`

Division: `/`

Floor Division: `//`

Modulo: `%`

Exponent: `**`

Vergleichsoperationen

Gleich: `==`

Nicht gleich: `!=`

Größer als: `>`

Kleiner als: `<`

Größer gleich: `>=`

Kleiner gleich: `<=`

Logische Operationen

Logisches UND: `&`

Logisches ODER: `|`

Logisches NICHT: `~`

String-Operationen

String-Operationen werden mit `str`-Accessor durchgeführt und bieten viele Funktionen, die denen von Python-Strings ähneln:

`str.contains()`: Testet, ob jeder String ein bestimmtes Muster enthält.

`str.startswith()`, `str.endswith()`: Testen, ob Strings mit einem bestimmten Muster beginnen oder enden.

`str.lower()`, `str.upper()`: Wandeln Strings in Groß- oder Kleinbuchstaben um.

`str.replace()`: Ersetzt einen Teil des Strings.

`str.split()`, `str.join()`: Teilt Strings oder fügt sie zusammen.

`str.extract()`, `str.extractall()`: Extrahieren von Teilen eines Strings basierend auf einem regulären Ausdruck.

`str.len()`: Gibt die Länge der Strings zurück.

Aggregationsoperationen

Aggregationsoperationen berechnen Statistiken über Elemente einer Serie oder entlang einer Achse eines DataFrame:

`sum()`: Summe der Elemente.

`mean()`: Mittelwert der Elemente.

`min()`, `max()`: Minimum oder Maximum.

`std()`, `var()`: Standardabweichung oder Varianz.

`count()`: Anzahl der nicht-NA/null Werte.

`describe()`: Generiert deskriptive Statistiken.

Handling fehlender Daten

Operationen, um mit fehlenden Daten umzugehen:

`isna()`, `notna()`: Testen auf fehlende Werte.

`fillna()`: Fehlende Daten mit einem Wert oder einer Methode auffüllen.

`dropna()`: Zeilen/Spalten mit fehlenden Daten entfernen.

Zeitreihenoperationen

Für Zeitreihendaten gibt es spezielle Vektoroperationen:

`pd.to_datetime()`: Konvertiert Strings in datetime Objekte.

`dt.year`, `dt.month`, `dt.day`: Extrahiert Jahr, Monat, Tag aus einem datetime Objekt.

`resample()`: Aggregiert zeitbasierte Daten in einem anderen Zeitrahmen.

Bedingte Operationen

`where()`: Ersetzt Werte, bei denen eine Bedingung falsch ist.

`mask()`: Ersetzt Werte, bei denen eine Bedingung wahr ist.

`query()`: Filtern von Daten basierend auf einer Abfragebedingung.

Zuweisungsoperationen

`apply()`: Wendet eine Funktion entlang einer Achse des DataFrames an.

`map()`: Wird verwendet, um Elemente in einer Serie zu ersetzen oder um Werte zu transformieren.

`assign()`: Fügt neue Spalten zu einem DataFrame hinzu.