6. Datenanalyse mit Pandas

- Einführung in Pandas
  - Was ist Pandas?
  - Was sind Dataframes?
  - Grundlegende Zugriffe auf Dataframes
- Übersetzung von SQL nach Pandas
  - Projektion, Selektion, Sortieren
  - GroupBy und Aggregationen
  - Joins
  - Insert, Update, Delete
- Erweiterte Konzepte in Pandas
  - Einlesen und Schreiben von Daten
  - Transformationen und Filterung
  - Analysebeispiele Covid-Datenbank

Überblick zu Pandas

- Data Science Modul für die Python Programmiersprache
- Anwendung in der Datenanalyse, -exploration und –aufbereitung
  - Direkte Verarbeitung von Daten im Python-Programmcode
- Einlesen aus verschiedenen Quellen/Formaten:
  - Datenbanken, CSV, Excel, JSON, etc.
- Unterstützung von Tabellen durch `DataFrame` Datenstruktur
  - Auch von (Zeit-)reihen über `Series` Datenstruktur (hier nicht Fokus)
  - Datenstrukturen bauen auf dem `NumPy` Modul auf
- Viele Operationen direkt über bereitgestellte Funktionen möglich

Dataframes

- Grundlegende Datenstruktur in Pandas
- Tabellarische Datenspeicherung ähnlich einer Microsoft Excel Tabelle
- Beispielausgabe eines Dataframe:

	a1	a2	a3
0	5	2	8
1	3	2	5
2	8	1	9

- Indexierte Zeilen und Spalten (in SQL nur Spalten!)
- Jede Zeile und Spalte stellt selbst eine `Series` dar (ähnlich Array)

Dataframe Erzeugung

- Einbindung des Moduls im Python Code als `pd`:

```
import pandas as pd
```

- Mit Dataframe Erzeugung im Code:

```
import pandas as pd
from pandas import DataFrame
zeilen = [[5,2,8], [3,2,5], [8,1,9]]
spalten = ['a1', 'a2', 'a3']
df = DataFrame(zeilen, columns=spalten)
```

	a1	a2	a3
0	5	2	8
1	3	2	5
2	8	1	9

- Oder direkt mit Einlesen von z.B. CSV-Daten:

```
import pandas as pd
df = pd.read_csv('data.csv')
```

Simple Dataframe Statistiken

- Grundlegende Informationen nach Einlesen von Daten sammeln
- Form eines Dataframe herausfinden: `df.shape`
  - Liefert hier: `(3,3)` mit (#zeilen, #spalten)
- Erste zwei Zeilen ausgeben: `df.head(2)`
  - Liefert hier: mit *gelb* hinterlegt im Beispiel
- Spaltennamen herausfinden: `df.columns`
  - Liefert hier: `['a1', 'a2', 'a3']`
- Datentypen in den Spalten: `df.dtypes`
  - Liefert hier: `int64` für jede Spalte
- Wertebereiche der Spalten: `df.describe()`
  - Liefert hier: Übersicht *rechts* gezeigt
  - Auch nützlich: `df.info()`

	a1	a2	a3
0	5	2	8
1	3	2	5
2	8	1	9

	a1	a2	a3
count	3.000000	3.000000	3.000000
mean	5.333333	1.666667	7.333333
std	2.516611	0.577350	2.081666
min	3.000000	1.000000	5.000000
25%	4.000000	1.500000	6.500000
50%	5.000000	2.000000	8.000000
75%	6.500000	2.000000	8.500000
max	8.000000	2.000000	9.000000

Spaltenzugriff I

- Einzelne Spalten selektierbar über Label:

```
a2 = df['a2']
```

  - Werden dann wie `Series` behandelt

	a1	a2	a3
0	5	2	8
1	3	2	5
2	8	1	9

Spaltenzugriff II

- Mehrere Spalten selektierbar:

```
df23 = df[['a2', 'a3']]
```

  - Werden dann wie neuer `Dataframe` behandelt

	a1	a2	a3
0	5	2	8
1	3	2	5
2	8	1	9

Zeilenzugriff I

- Zeilenzugriff über Index (Position):

```
idx1 = df.iloc[1]
```

  - Werden dann wie `Series` behandelt

- Index kann auch auf Labels erweitert werden:

```
df.index = ['A', 'B', 'C']
```

- Zugriff dann zusätzlich über das Label möglich:

```
idx1 = df.loc['B']
```

	a1	a2	a3
0/A	5	2	8
1/B	3	2	5
2/C	8	1	9

Zeilenzugriff II

- Statt `iloc` kann bei Auswahl über Slicing mit Integern auch direkt per z.B. `df[1:3]` auf Zeilen zugegriffen werden, da für Spalten Strings existieren
  - Gibt jedoch immer `Dataframe` Format zurück!

- Mehrere Zeilen selektierbar über Slicing:

```
df_idx12 = df[1:3]
```

  - Werden dann wie neuer `Dataframe` behandelt

	a1	a2	a3
0	5	2	8
1	3	2	5
2	8	1	9

- Ebenso über bool'sche Liste:

```
df_idx12 = df[[False, True, True]]
```

Kombinierter Zugriff

- Erlaubt Auswahl von bestimmten Teilen des Dataframe:

```
df = df[['a2','a3']][1:3]
```

- Auch über `loc` Funktion:

```
df = df.loc['B':'C',['a2','a3']]
```

	a1	a2	a3
0/A	5	2	8
1/B	3	2	5
2/C	8	1	9

Genauer Wertezugriff

- Durch Kombination auch genauer Wertezugriff:

```
x = df['a2'][1]
```

  - Gibt einzelnen Wert mit Datentyp der Spalte

	a1	a2	a3
0/A	5	2	8
1/B	3	2	5
2/C	8	1	9

Filtern

- Zugriff über bestimmte Werte-Einschränkungen

- Beispiel für Zeilen mit Wert in Spalte a1 kleiner 5:

```
a1 = df['a1']  # Series aus Spalte a1
xs = a1 < 5            # xs = [False, True, False]
df = df[ xs ]
```

- Kurz: `df = df[ df['a1'] < 5 ]`

	a1	a2	a3
0/A	5	2	8
1/B	3	2	5
2/C	8	1	9

6. Datenanalyse mit Pandas

- Einführung in Pandas
  - Was ist Pandas?
  - Was sind Dataframes?
  - Grundlegende Zugriffe auf Dataframes
- Übersetzung von SQL nach Pandas
  - Projektion, Selektion, Sortieren
  - GroupBy und Aggregationen
  - Joins
  - Insert, Update, Delete
- Erweiterte Konzepte in Pandas
  - Einlesen und Schreiben von Daten
  - Transformationen und Filterung
  - Analysebeispiele Covid-Datenbank

Nutzung von SQL in Pandas

- Bekannte SQL-Ausdrücke sind auch in Pandas realisierbar
  - Teilweise direkt als konkrete Funktionen verfügbar
  - Sonst durch Verkettung mehrerer Funktionen (Method-Chaining)
  - Einiges auch unter zu Hilfenahme von Python Konzepten
- Aber auch Einbindung von SQL in Pandas
  - Um bestimmten Teil einer Datenbank als Dataframe einzulesen
  - Um SQL-Queries auf Dataframes zu stellen über z.B. das `pandasql`-Modul
- Wichtig: Pandas-Anfragen geben Lösungs-Dataframe meist als eigene Kopie
  - Beispiel: `df2 = df[['a2', 'a3']]`
  - “df2” wird ein neuer Dataframe mit Inhalt der Anfrage, also nur Spalten “a2” und “a3”
  - Originaler Dataframe “df” bleibt hier jedoch unverändert

Projektion

SQL

```
SELECT a2, a3 FROM df_r
```

Pandas

```
df_r[['a2', 'a3']]
```

Beispiel

df_r

	a1	a2	a3
0	5	2	8
1	3	2	5
2	8	1	9

Selektion

SQL

```
SELECT * FROM df_r WHERE a3 < 9
```

Pandas

```
df_r[ df_r['a3'] < 9 ]
```

Beispiel

df_r

	a1	a2	a3
0	5	2	8
1	3	2	5
2	8	1	9

Selektion mit mehreren Filtern

SQL

```
SELECT * FROM df_r WHERE a3 < 9 AND a1 = 3
```

Pandas

```
df_r[ (df_r['a3'] < 9) & (df_r['a1'] == 3) ]
```

Beispiel

df_r

	a1	a2	a3
0	5	2	8
1	3	2	5
2	8	1	9

Selektion mit Berechnung

SQL

```
SELECT *, a1*a2*a3 AS mult FROM df_r
```

Pandas

```
df_r['mult'] = df_r['a1'] * df_r['a2'] * df_r['a3']
```

Beispiel

df_r

	a1	a2	a3	mult
0	5	2	8	80
1	3	2	5	30
2	8	1	9	72

Sortieren

SQL

```
SELECT * FROM df_r ORDER BY a1 ASC
```

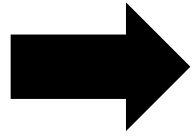
Pandas

```
df_r.sort_values('a1', ascending=True)
```

Beispiel

df_r

	a1	a2	a3
0	5	2	8
1	3	2	5
2	8	1	9



	a1	a2	a3
0	3	2	5
1	5	2	8
2	8	1	9

GroupBy mit Aggregation

SQL

```
SELECT a2, count(*) FROM df_r GROUP BY a2
```

Pandas

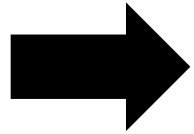
zählt NULL: `df_r.groupby('a2')['a2'].size()`

ohne NULL: `df_r.groupby('a2')['a2'].count()`

Beispiel

df_r

	a1	a2	a3
0	5	2	8
1	3	2	5
2	8	1	9



a2	
1	1
2	2

in pandas sortiert `groupby` standardmäßig!

GroupBy mit mehreren Aggregation

SQL

```
SELECT a2, count(a3), avg(a3) FROM df_r GROUP BY a2
```

Pandas

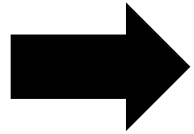
Mehrere Aggregationen über `agg`-Funktion:

```
df_r.groupby('a2')['a3']
    .agg(['count', 'mean'])
```

Beispiel

df_r

	a1	a2	a3
0	5	2	8
1	3	2	5
2	8	1	9



	count	mean
a2		
1	1	9.0
2	2	6.5

Inner Join

SQL

```
SELECT * FROM df_r INNER JOIN df_s
    ON df_r.a3 = df_s.a3
```

Pandas

```
pd.merge(df_r, df_s, on='a3')
```

`merge` nutzt Standardmäßig Inner Join

Hinweis: Pandas nimmt auch `NULL`-Werte als Join-Partner!

Beispiel

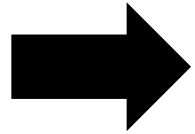
df_r

	a1	a2	a3
0	5	2	8
1	3	2	5
2	8	1	9

df_s

⋈

	a3	a4	a5
0	8	1	5
1	4	6	7
2	5	9	3



	a1	a2	a3	a4	a5
0	5	2	8	1	5
1	3	2	5	9	3

Natural Join

SQL

```
SELECT * FROM df_r NATURAL JOIN df_s
```

Pandas

```
pd.merge(df_r, df_s)
```

Beispiel

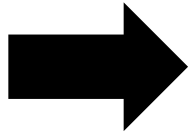
df_r

	a1	a2	a3
0	5	2	8
1	3	2	5
2	8	1	9

df_s

⋈

	a3	a4	a5
0	8	1	5
1	4	6	7
2	5	9	3



	a1	a2	a3	a4	a5
0	5	2	8	1	5
1	3	2	5	9	3

Join verschiedener Attributnamen

SQL

```
SELECT * FROM df_r INNER JOIN df_s
    ON df_r.a3 = df_s.a4
```

Pandas

```
pd.merge(df_r, df_s, left_on='a3', right_on='a4')
```

Beispiel

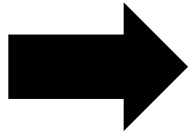
df_r

	a1	a2	a3
0	5	2	8
1	3	2	5
2	8	1	9

df_s

⋈

	a4	a5	a6
0	8	1	5
1	4	6	7
2	5	9	3



	a1	a2	a3	a4	a5
0	5	2	8	1	5
1	3	2	5	9	3

Full Outer Join

SQL

```
SELECT * FROM df_r FULL OUTER JOIN df_s
    ON df_r.a3 = df_s.a3
```

Pandas

```
pd.merge(df_r, df_s, on='a3', how='outer')
```

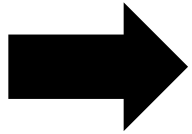
Beispiel

df_r

	a1	a2	a3
0	5	2	8
1	3	2	5
2	8	1	9

df_s ⋈

	a3	a4	a5
0	8	1	5
1	4	6	7
2	5	9	3



	a1	a2	a3	a4	a5
0	5	2	8	1	5
1	3	2	5	9	3
2	8	1	9	N	N
3	N	N	4	6	7

Left Outer Join

SQL

```
SELECT * FROM df_r LEFT OUTER JOIN df_s
    ON df_r.a3 = df_s.a3
```

Pandas

```
pd.merge(df_r, df_s, on='a3', how='left')
```

(äquivalent für Right Outer Join `right` einfügen)

Beispiel

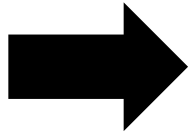
df_r

	a1	a2	a3
0	5	2	8
1	3	2	5
2	8	1	9

df_s

	a3	a4	a5
0	8	1	5
1	4	6	7
2	5	9	3

⋈



	a1	a2	a3	a4	a5
0	5	2	8	1	5
1	3	2	5	9	3
2	8	1	9	N	N

Insert / Update / Delete

Insert

- Neue Spalten und Zeilen durch direkte Zuweisung hinzufügbar

```
df['neue_spalte'] = [1,2,3]
df.loc[len(df)] = [1,2,3]
```

Update – *Preis verdoppeln wo Gewinn < 5*

- Auswahl der gesuchten Werte und Zuweiseung der neuen Werte

```
df.loc[df['gewinn'] < 5, 'preis'] *= 2
```

Delete – *Löschung wo Gewinn kleiner 5*

- Auswahl der bleibenden Zeilen statt Löschung

```
df = df.loc[df['gewinn'] >= 5]
```

6. Datenanalyse mit Pandas

- Einführung in Pandas
  - Was ist Pandas?
  - Was sind Dataframes?
  - Grundlegende Zugriffe auf Dataframes
- Übersetzung von SQL nach Pandas
  - Projektion, Selektion, Sortieren
  - GroupBy und Aggregationen
  - Joins
  - Insert, Update, Delete
- Erweiterte Konzepte in Pandas
  - Einlesen und Schreiben von Daten
  - Transformationen und Filterung
  - Analysebeispiele Covid-Datenbank

Laden und Speichern von Daten

- Einlesen von verschiedenen Datentypen
  - Microsoft Excel: `df = pd.read_excel('data.xlsx')`
  - CSV: `df = pd.read_csv('data.csv')`
  - JSON: `df = pd.read_json('data.json')`
  - SQL: `df = pd.read_sql(query_or_tablename_str, conn)`
  - Zwischenablage: `df = pd.read_clipboard()`
  - Auch aus Dateien über Links durch Eingabe des URL-String
  - und mehr…
- Schreiben aus Dataframes
  - Microsoft Excel: `df.to_excel('data.xlsx')`
  - CSV: `df.to_csv('data.csv')`
  - JSON: `df.to_json('data.json')`
  - SQL: `df.to_sql(tablename_str, conn)`
  - und mehr…

Lambda-Funktionen

- Auch bekannt unter der Bezeichnung Anonyme Funktion
- Funktionskonzept welches ohne explizite bezeichnende Definition auskommt
  - Meist wenn nur einmalige Verwendung und begrenzter Umfang
  - Nutzung dann in höherwertigen Funktionen, welche Funktion als Eingabe erwarten
  - u.A. verfügbar in Python, Java, …
- Aufbau: `lambda` Schlüsselwort, Eingabeparameter, Funktionsausdruck

```
lambda arg1(, arg2, arg3): arg1 * 2
```

- Über Variable Funktionsnamen für Aufrufe möglich

```
fn = lambda x,y: x+y
res = fn(2,4)
```

  - Weiteres Beispiel auf der nächsten Folie

Apply-Funktionen

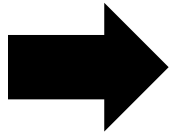
- `apply(func, axis,...)`
  - Funktion **für ganze Achsen** des Dataframes ausführen
    - `axis=0` entspricht jeder Spalte, `axis=1` entspricht jeder Zeile
- `applymap(func, na_action=None,...)`
  - Funktion **für jedes Element** des Dataframes ausführen
- Beispiel: Datum zusammenführen

```
fn = lambda row: str(row.year) +'-'+ str(row.month) +'-'+ str(row.day)

date_df['datum'] = date_df.apply(fn, axis=1)
```

	year	month	day
0	2020	2	24
1	2020	2	25

date_df



	year	month	day	datum
0	2020	2	24	2020-2-24
1	2020	2	25	2020-2-25

date_df

- Hinweis: `apply` Funktionen sind langsamer als vektorisierte Lösungen

Filterungs-Funktionen

- `filter(items, like, regex, axis,…)`
  - Indexierung/Label aus dem Dataframe auf Basis von Filtern selektieren
  - Spalten: `df.filter(items=['day','month'])` – Spalten "day" und "month"
  - Like: `df.filter(like='mon', axis=0)` – Zeilenindex die "mon" enthalten
  - Regex: `df.filter(regex='th$', axis=1)` – Spalten die auf "th" enden
- `query(expr)`
  - Selektieren aus Dataframe basierend auf einfachen String Queries
  - `df.query("A > 4 and E == 5")`
  - `values = [22,23,24]` dann z.B. `df.query("B not in @value")`
- Konkrete Filter Methoden
  - Die meisten Operationen können auch als eigene Funktion genutzt werden
  - `df[ df['day'].str.contains('mon') ]`
  - `df[ df['preis'].isin([22,23,24]) ]`

Beispiel Covid-19-Daten

Datenquelle https://github.com/owid/covid-19-data/tree/master/public/data

SQL unter https://lots.uni-leipzig.de/sql-training/ (Stand 07.04.2022)

CSV unter https://dbs.uni-leipzig.de/file/cov_lots.txt (Stand 07.04.2022)

cov_df

	tag	monat	jahr	land	faelle	tote	population	kontinent
0	24	2	2020	Afghanistan	5.0	NaN	39835428	Asia
1	25	2	2020	Afghanistan	0.0	NaN	39835428	Asia
2	26	2	2020	Afghanistan	0.0	NaN	39835428	Asia
3	27	2	2020	Afghanistan	0.0	NaN	39835428	Asia
4	28	2	2020	Afghanistan	0.0	NaN	39835428	Asia
...	...	...	...	...	...	...	...	...

Datenstand: 07.04.2022

Beispiel GroupBy (Covid)

Group-by kontinent + jahr (drill-down, 2-dimens.)

```
SELECT kontinent, jahr, SUM(faelle) AS faelle, SUM(todesfaelle) AS tote
FROM (cov NATURAL JOIN land NATURAL JOIN zeit)
WHERE NOT (faelle IS NULL AND todesfaelle IS NULL)
GROUP BY kontinent, jahr ORDER BY kontinent, jahr LIMIT 16
```

kontinent	jahr	faelle	tote
Africa	2020	2,760,926	65,507
Africa	2021	6,977,423	163,031
Africa	2022	1,826,358	23,460
Asia	2020	19,898,890	338,282
Asia	2021	64,274,703	914,735
Asia	2022	58,294,580	154,200
Europe	2020	23,928,285	546,454
Europe	2021	65,198,522	985,155
Europe	2022	94,914,027	251,507
North_America	2020	23,180,679	509,168
North_America	2021	41,865,982	770,693
North_America	2022	29,877,422	213,438
Oceania	2020	48,428	1,060
Oceania	2021	540,424	3,462
Oceania	2022	5,433,304	5,051

Datenstand: 07.04.2022

Beispiel GroupBy (Covid)

Group-by kontinent + jahr (drill-down, 2-dimens.)

```
df = (cov_df.groupby(['kontinent','jahr'])
            .agg({'faelle': 'sum', 'tote': 'sum'}))
```

		faelle	tote
kontinent	jahr		
Africa	2020	2760926.0	65507.0
	2021	6977423.0	163031.0
	2022	1826358.0	23460.0
Asia	2020	19898890.0	338282.0
	2021	64274703.0	914735.0
	2022	58294580.0	154200.0
Europe	2020	23928285.0	546454.0
	2021	65198522.0	985155.0
	2022	94914027.0	251507.0
North_America	2020	23180679.0	509168.0
	2021	41865982.0	770693.0
	2022	29877422.0	213438.0
Oceania	2020	48428.0	1060.0
	2021	540424.0	3462.0
	2022	5433304.0	5051.0
South_America	2020	13230145.0	414318.0
	2021	26448688.0	765948.0
	2022	16481334.0	86500.0

Datenstand: 07.04.2022

Rank-Beispiel (Covid-Datenbank)

Ranking der Länder basierend auf den gesamt gezählten Fällen und Toden

```
SELECT landname AS land, SUM(COALESCE(faelle, 0)) AS faelle,
SUM(COALESCE(todesfaelle, 0)) AS tote,
RANK() OVER w1 AS "RankCases", RANK() OVER w2 AS "RankDeaths"
FROM (cov NATURAL JOIN land)
WHERE NOT (faelle IS NULL AND todesfaelle IS NULL) GROUP BY land
WINDOW w1 AS (ORDER BY SUM(COALESCE(faelle, 0)) DESC),
       w2 AS (ORDER BY SUM(COALESCE(todesfaelle, 0)) DESC) LIMIT 16
```

land	faelle	tote	RankCases	RankDeaths
United_States	80,289,236	1,065,464	1	1
India	43,033,067	514,200	2	3
Brazil	29,990,246	661,228	3	2
France	27,026,588	143,473	4	10
Germany	22,478,759	131,391	5	13
United_Kingdom	20,737,905	165,501	6	7
Russia	17,693,468	363,455	7	4
Italy	15,106,214	160,433	8	8
South_Korea	14,983,693	18,754	9	46
Turkey	14,116,034	98,311	10	19
Spain	11,636,410	103,456	11	17
Vietnam	10,070,692	42,918	12	24
Argentina	9,049,250	128,158	13	14
Netherlands	8,054,934	22,197	14	39
Iran	7,183,808	140,492	15	11
Japan	6,885,874	28,531	16	32

Datenstand: 07.04.2022

Rank-Beispiel (Covid-Datenbank)

Ranking der Länder basierend auf den gesamt gezählten Fällen und Toden

```
df = cov_df[['land', 'faelle', 'tote']]
df = (df.groupby('land')
        .agg({'faelle': 'sum', 'tote': 'sum'})
        .sort_values(['faelle', 'tote'], ascending=False))
df[['RankCases',
    'RankDeaths']] = df[['faelle','tote']].rank(ascending=False)
print(df.head(16))
```

land	faelle	tote	RankCases	RankDeaths
United_States	80289236.0	1065464.0	1.0	1.0
India	43033067.0	514200.0	2.0	3.0
Brazil	29990246.0	661228.0	3.0	2.0
France	27026588.0	143473.0	4.0	10.0
Germany	22478759.0	131391.0	5.0	13.0
United_Kingdom	20737905.0	165501.0	6.0	7.0
Russia	17693468.0	363455.0	7.0	4.0
Italy	15106214.0	160433.0	8.0	8.0
South_Korea	14983693.0	18754.0	9.0	46.0
Turkey	14116034.0	98311.0	10.0	19.0
Spain	11636410.0	103456.0	11.0	17.0
Vietnam	10070692.0	42918.0	12.0	24.0
Argentina	9049250.0	128158.0	13.0	14.0
Netherlands	8054934.0	22197.0	14.0	39.0
Iran	7183808.0	140492.0	15.0	11.0
Japan	6885874.0	28531.0	16.0	32.0

Datenstand: 07.04.2022

Beispiel Moving Average (Covid)

Infektionen pro Tag vs. Tagesdurchschnitt sowie Inzidenz pro 100,000 der letzten Woche in D.

```
SELECT tag, monat, jahr, COALESCE(faelle, 0) AS faelle,
ROUND(AVG(COALESCE(faelle, 0)) OVER w1, 0) AS "avgCasesWeek",
ROUND(100000 * SUM(COALESCE(faelle, 0)) OVER w1 / population, 0) AS "WeekIncidence"
FROM (cov NATURAL JOIN land NATURAL JOIN zeit) WHERE landname='Germany'
WINDOW w1 AS (ORDER BY jahr, monat, tag ROWS BETWEEN 6 PRECEDING AND 0 FOLLOWING)
```

tag	monat	jahr	faelle	avgCasesWeek	WeekIncidence
19	3	2022	260,239	221,926	1,851
20	3	2022	131,792	219,809	1,833
21	3	2022	92,314	219,800	1,833
22	3	2022	222,080	223,113	1,861
23	3	2022	283,732	226,133	1,886
24	3	2022	318,387	229,484	1,914
25	3	2022	296,498	229,292	1,913
26	3	2022	252,026	228,118	1,903
27	3	2022	111,224	225,180	1,878
28	3	2022	67,501	221,635	1,849
29	3	2022	237,352	223,817	1,867
30	3	2022	268,477	221,638	1,849
31	3	2022	565,139	256,888	2,143
1	4	2022	0	214,531	1,789
2	4	2022	196,456	206,593	1,723
3	4	2022	115,182	207,158	1,728
4	4	2022	0	197,515	1,647
5	4	2022	180,397	189,379	1,580
6	4	2022	416,714	210,555	1,756
7	4	2022	175,263	154,859	1,292

Datenstand: 07.04.2022

Beispiel Moving Average (Covid)

Infektionen pro Tag vs. Tagesdurchschnitt sowie Inzidenz pro 100,000 der letzten Woche in D.

```
df = cov_df[cov_df['land']=='Germany'][['tag','monat','jahr','faelle']]
df['avgCasesWeek'] = cov_df['faelle'].fillna(0).rolling(7).mean().round()
pop = cov_df[cov_df['land']=='Germany']['population'].head(1).values[0]
df['WeekIncidence'] = (cov_df['faelle'].fillna(0).rolling(7).sum() * 100000 / pop).round()
print(df[(df['jahr']==2022) & (df['monat']==3) & (df['tag']>=19)])
```

tag	monat	jahr	faelle	avgCasesWeek	WeekIncidence
19	3	2022	260239.0	221926.0	1852.0
20	3	2022	131792.0	219809.0	1834.0
21	3	2022	92314.0	219800.0	1834.0
22	3	2022	222080.0	223113.0	1861.0
23	3	2022	283732.0	226133.0	1887.0
24	3	2022	318387.0	229484.0	1915.0
25	3	2022	296498.0	229292.0	1913.0
26	3	2022	252026.0	228118.0	1903.0
27	3	2022	111224.0	225180.0	1879.0
28	3	2022	67501.0	221635.0	1849.0
29	3	2022	237352.0	223817.0	1867.0
30	3	2022	268477.0	221638.0	1849.0
31	3	2022	565139.0	256888.0	2143.0

Datenstand: 07.04.2022

Beispiel Plotting (Covid)

Visualisierung der täglich gezählten Corona-Fälle in Deutschland

1.) Kombinierte Spalte für Datum hinzufügen

```
df = cov_df[cov_df['land']=='Germany']
df['datum']
  = (df[['jahr','monat','tag']]
       .astype('str')
       .apply(lambda row:
                pd.to_datetime(row.jahr + ' ' +
                               row.monat + ' ' +
                               row.tag), axis=1))
print(df)
```

	tag	monat	jahr
	27	1	2020
	28	1	2020
	29	1	2020
	30	1	2020
	31	1	2020
	...	...	...
	3	4	2022
	4	4	2022
	5	4	2022
	6	4	2022
	7	4	2022

	datum
2020-01-27	
2020-01-28	
2020-01-29	
2020-01-30	
2020-01-31	
...	
2022-04-03	
2022-04-04	
2022-04-05	
2022-04-06	
2022-04-07	

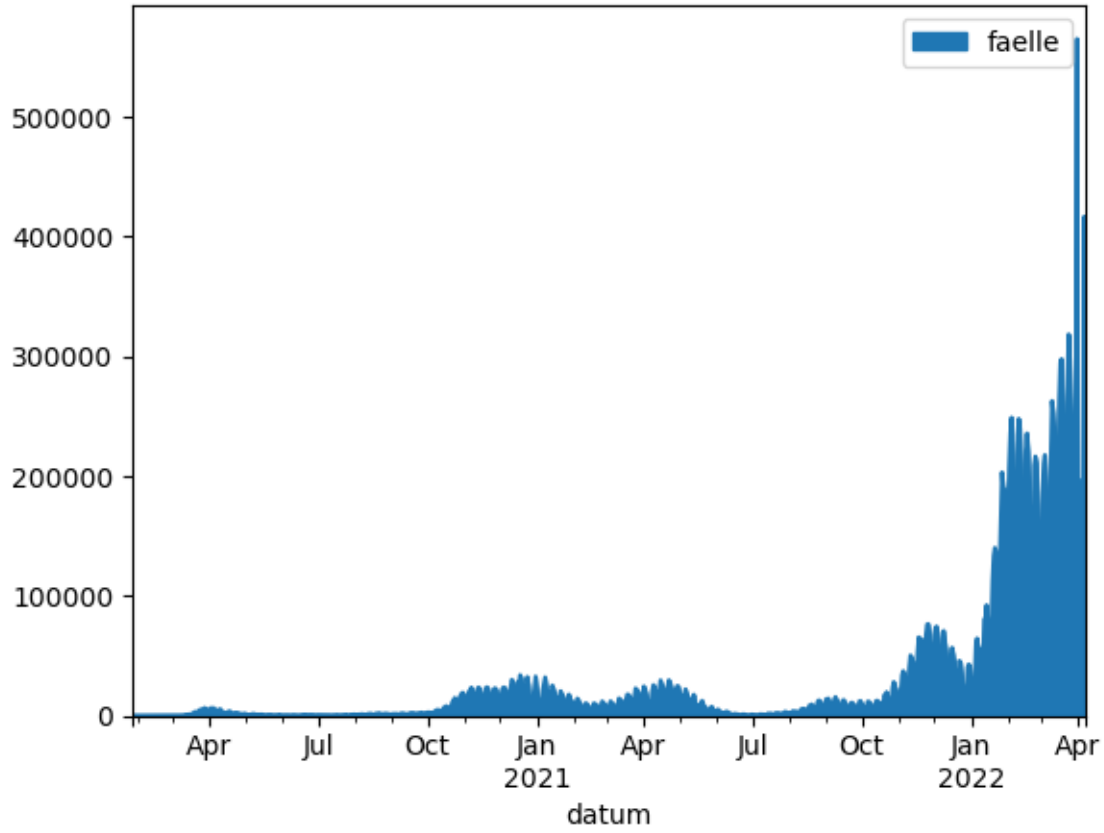
Datenstand: 07.04.2022

Beispiel Plotting (Covid)

Visualisierung der täglich gezählten Corona-Fälle in Deutschland

2.) Plot generieren

```
(df[['datum','faelle']].sort_values('datum')
                       .plot(x='datum', y='faelle', kind='area'))
```



Nutzt das `matplotlib` Modul im Hintergrund

Hinweis: Werte am Ende der Zeitreihe wurden damals teils über Tage kumuliert und sind nach heutigem Stand einzeln geringer.

Datenstand: 07.04.2022

Zusammenfassung

- Schnelle und mächtige Datenanalyse
  - Arbeit auch ohne eigenständige Datenbank sehr gut möglich
  - Umfangreicher Katalog eingebauter Funktionen
    - Durch Python-Umgebung zudem leichte Erweiterung der Funktionalitäten
  - Arbeit auch auf zeilenbasierter Indexierung möglich
    - Aber Probleme mit 3D-Datenverarbeitung
- Durch lokales Halten der Daten eher beschränkte Datenmengen möglich
  - Keine vollwertige Alternative für direkte Arbeit auf Datenbanken
  - Daten bleiben nur zur Anwendungszeit gespeichert, also keine permanente Datenhaltung wie bei Arbeit mit SQL auf dedizierter Datenbank
- Lösung: Kombination von Technologien
  - Pandas als Analysetool für einfache Exploration und Datenbereinigung
  - SQL für Datenbankverwaltung, auch durch z.B. Views / Trigger
  - Außerdem umfangreiche Analysen besser in Datacubes etc. aufgehoben

Covid-Beispiele auch im Moodle-Kurs

- Beispiele für Anfragen auf Covid-Datensatz im Moodle-Kurs
  - Jupyter Notebook oder Python Programm
    - Installation von `Python, pandas` und `matplotlib` nötig
    - Ggf. `Jupyter` für Notebook Nutzung installieren
  - CSV als Datei oder per Link (Der Link funktioniert in Google Colab scheinbar nicht!)
- Gern selbst ausprobieren und alternative Lösungen / Aufgaben finden
  - Beispielsweise die anderen Anfragen aus Kapitel 5

```
# Pandas Modul laden
import pandas as pd
```

```
# Dataframe aus CSV-Daten erstellen
url = ("https://dbs.uni-leipzig.de/file/cov_lots.txt")
cov_df = pd.read_csv(url)
cov_df
```

	tag	monat	jahr	land	faelle	tote	population	kontinent
0	24	2	2020	Afghanistan	5.0	NaN	39835428	Asia
1	25	2	2020	Afghanistan	0.0	NaN	39835428	Asia
2	26	2	2020	Afghanistan	0.0	NaN	39835428	Asia
3	27	2	2020	Afghanistan	0.0	NaN	39835428	Asia
4	28	2	2020	Afghanistan	0.0	NaN	39835428	Asia
...	...	...	...	...	...	...	...	...

Nützliche Literatur und Tutorials

- Dokumentation zum Pandas Modul: `https://pandas.pydata.org`
- Vergleich von Pandas und SQL in der Pandas Dokumentation:
  `https://pandas.pydata.org/docs/getting_started/comparison/comparison_with_sql.html`
- Buch “Datenanalyse mit Python” von Wes McKinney
- Blog “Modern Pandas” von Tom Augspurger:
  `https://tomaugspurger.github.io/posts/modern-1-intro/`
- Data Science 1, Hochschule Bochum von Prof. Dr. Christian Bockermann:
  `https://datascience.hs-bochum.de/Vorlesungen/WS2122/DataScience1/`
- Data Science, Heinrich Heine Universität Düsseldorf von Dr. Konrad Völkel:
  `https://mediathek.hhu.de/playlist/1326`