Wirtschaftsinformatik 2

Pandas DataFrames 2

Prof. Dr. Christian Bockermann, Prof. Dr. Volker Klingspor

Hochschule Bochum

Wintersemester 2026/2027

Vorbemerkungen zum Block Pandas DataFrames 2

Wir haben im letzten Block den Einstieg in Pandas' `DataFrame` Struktur kennengelernt. Das ist in der Regel der Datentyp, mit dem wir eine Vielzahl von Berechnungen und Analysen durchführen können. Pandas' `DataFrame` Objekte bieten eine unglaubliche Vielzahl an Funktionalitäten, die wir im in der Vorlesung nicht vollumfänglich behandeln können. Daher geht es in diesem Block darum, ein paar weitere, grundlegende Konzepte und Ideen von Pandas zu vertiefen. Dazu gehört insbesondere

- der Zugriff auf Zeilen/Spalten/Zellen über die Position,
- das Filtern von Daten,
- und das Aggregieren von Daten

Vorwort

Daten in tabellarischer Form kommen in einer Vielzahl von betriebswirtschaftlichen Anwendungen vor. Ein Beispiel ist die nachfolgende Tabelle mit Bestellungen aus dem Schoko-Kugel Shop, bei dem jede Zeile einer Bestellung entspricht.

Datum	BestellungNr	KundeNr	Anzahl_Gold	Anzahl_Rot	Anzahl_Blau	Anzahl_Gruen	Betrag
2023-10-12	77	78	1	0	0	0	0.050
2023-10-12	78	79	1	2	0	0	0.150
2023-10-12	79	80	1	1	1	0	0.150
2023-10-12	80	81	1	0	1	1	0.150
2023-10-12	81	82	1	1	1	0	0.150
2023-10-12	82	83	0	0	1	0	0.050

Die Tabelle enthält Spalten für die Bestellnummer, den Zeitpunkt der Bestellung, sowie die Anzahl der jeweils bestellten Produkte und den Gesamtbetrag der Bestellung.

Solche Tabellen können in Pandas einfach geladen und analysiert werden.
Genau das ist das Ziel der restlichen Vorlesungen/Blöcke in *Wirtschaftsinformatik 2*.
Ein kleines Beispiel als *Schnellstart*:

```python
# Pandas importieren
import pandas as pd

# Tabelle mit Bestellungen einlesen
url = 'https://data.hsbo.de/Bestellungen.csv'
bestellungen = pd.read_csv(url)

# Statistiken/Analysen/Ergebnisse berechnen
bestellungen.describe()
```

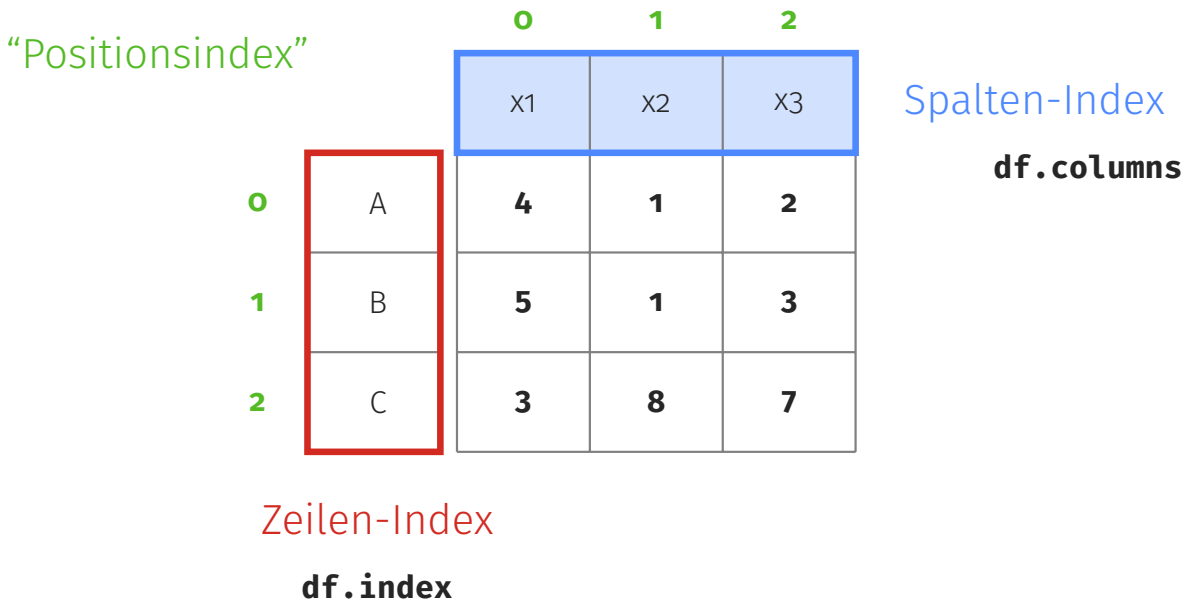


Probieren Sie es im Notebook aus!

Im Folgenden werden wir uns im Detail mit `DataFrame`s und deren Konzepten und Benutzung auseinandersetzen...

Zeilen-/Spalten-Index und Positionen

Neben dem Index und den Spaltennamen, hat jede Zeile/Spalte auch eine Position. Dafür gibt es in Pandas die Positionsindizes für Zeilen und Spalten, die – unabhängig von der Art der Spaltenbezeichner oder des Index – immer existieren und die Zeilen/Spalten von 0 an durchnummerieren:
Die folgende Abbildung zeigt einen DataFrame und in grün die jeweiligen Positionsindizes für die Zeilen und Spalten:



Positionsindex mit `.iloc[..]`

Ähnlich wie mit `.loc[..]` gibt es für DataFrames einen Zugriff über `.iloc[..]`, der ebenfalls über Zeilen und Spalten selektieren kann:

```
df.iloc[ ZEILEN , SPALTEN ]
```

Sowohl der Selektor `ZEILEN` als auch `SPALTEN` kann wie bei `.iloc` wieder unterschiedliche Arten von Werten haben, z.B.

- ein einzelner Wert
- eine Liste von Werten
- ein Bereich

Einzelne Zeilen/Spalten sind `Series` Objekte

Beispiele für den Zugriff auf **`df`**. Wenn der spalten Selektor weggelassen wird, bedeutet das, dass jeweils die kompletten Spalten mitselektiert werden.

```
# Zugriff mit Positionsindex:
df.iloc[zeilen,spalten]

df.iloc[3] # die komplette 4. Zeile

df.iloc[:,2] # die komplette 3. Spalte
```

Selektieren mit `iloc[..]`

Die Selektoren für `.iloc` können auch *Bereiche* (Slices) sein. Slices haben die Form `a:b`, wobei `a` und `b` den Start und das Ende des Bereichs darstellen.
Ist für das Slice nur `:` angegeben, so bedeutet dies "von Anfang bis Ende", also die komplette Zeile bzw. Spalte:

```
# die Zeilen 0 und 1:
df.iloc[0:2,:]

# die Spalten 0 und 1:
df.iloc[:,0:2]
```

Was passiert bei `.iloc[..]`?

Hier mal ein Beispiel für das Slicing mit `.iloc[..]`. Wir nehmen das Slicing von `0:2`:

```
df.iloc[0:2]
```

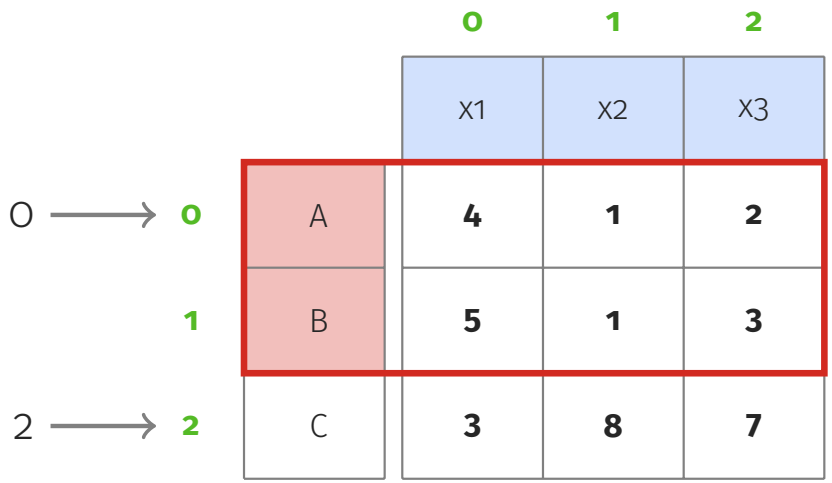
		0	1	2
		X1	X2	X3
0	A	**4**	**1**	**2**
1	B	**5**	**1**	**3**
2	C	**3**	**8**	**7**

Beim Slicing mit `.iloc[..]` gehört die obere Schranke (in diesem Fall die Position 2) nicht mit zur Auswahl! Unterschied zu `.loc`!!

Was passiert bei `.iloc[..]`?

Hier mal ein Beispiel für das Slicing mit `.iloc[..]`. Wir nehmen das Slicing von `0:2`:

```
df.iloc[0:2]
```



Beim Slicing mit `.iloc[..]` gehört die obere Schranke (in diesem Fall die Position 2) nicht mit zur Auswahl! Unterschied zu `.loc`!!

Datenanalyse mit Pandas (Teil 2)

DataFrame - Filtern von Daten

Der Zugriff über df[..]

Wir hatten uns bereits beim Datentyp `Series` mit dem Filtern von Werten beschäftigt. Dies geschah z.B. über Listen/Sequenzen bool'scher Werte (`True`/`False`):

	a1	a2	a3
0	**4**	**1**	**2**
1	**5**	**1**	**3**
2	**3**	**8**	**7**

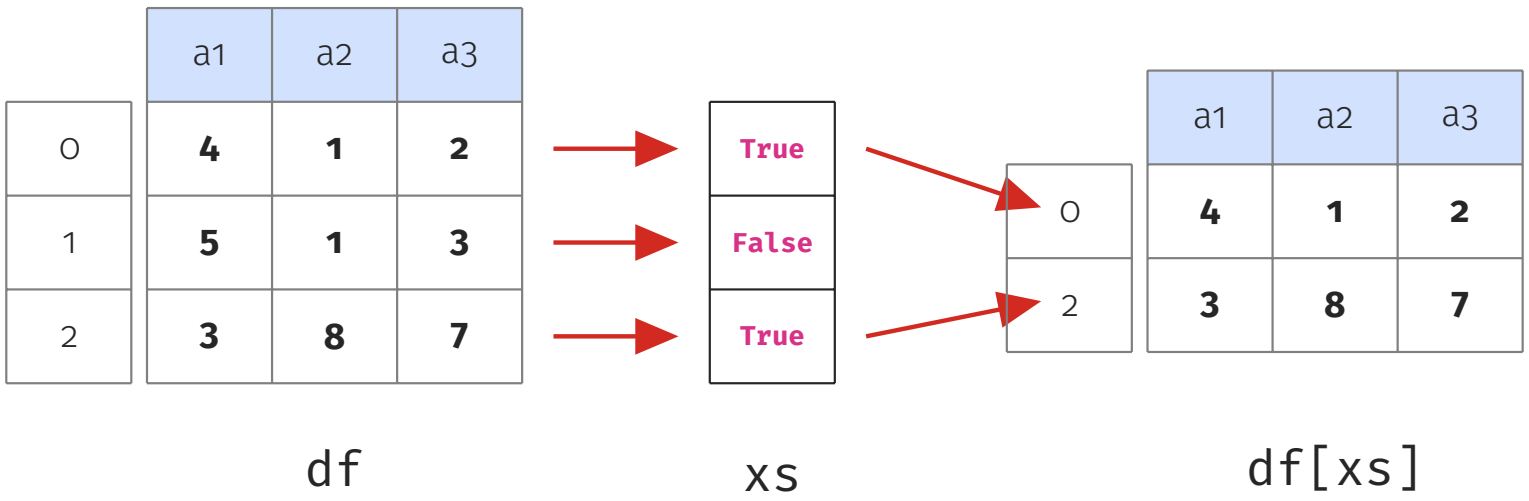
df

True
False
True

xs

Der Zugriff über df[..]

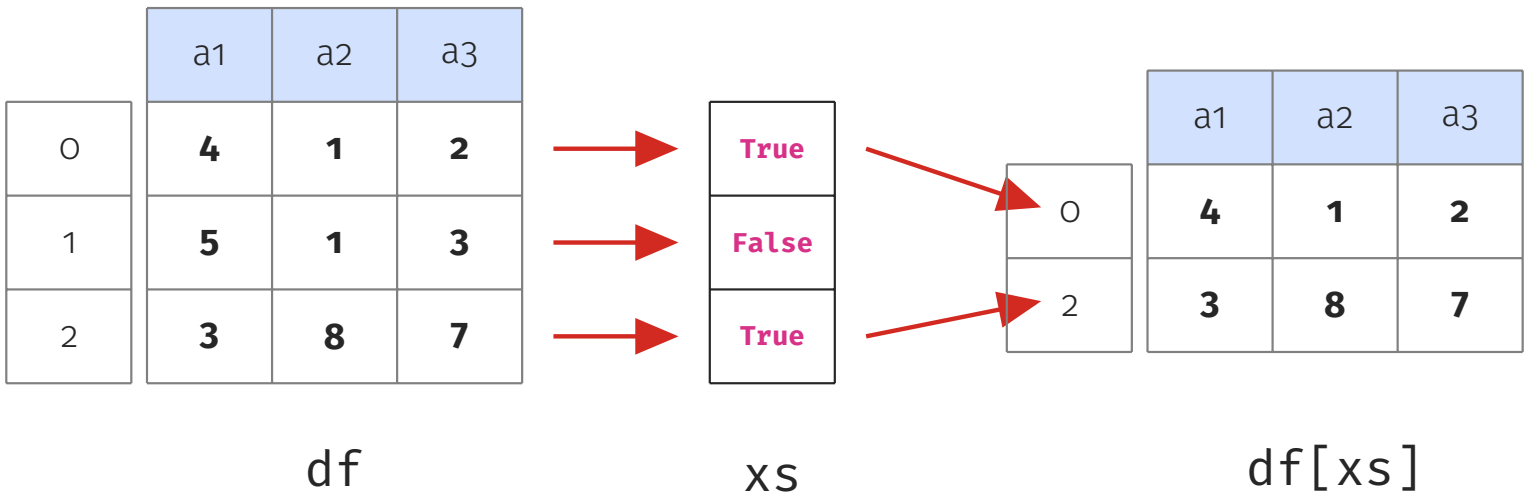
Wir hatten uns bereits beim Datentyp `Series` mit dem Filtern von Werten beschäftigt. Dies geschah z.B. über Listen/Sequenzen bool'scher Werte (`True`/`False`):



Woher kriegen wir bool'sche Sequenzen?

Der Zugriff über df[..]

Wir hatten uns bereits beim Datentyp `Series` mit dem Filtern von Werten beschäftigt. Dies geschah z.B. über Listen/Sequenzen bool'scher Werte (`True`/`False`):



Woher kriegen wir bool'sche Sequenzen?

Über Series Objekte!

Erinnern wir uns an `Series` Operationen (Folie 31 ff.)

Für `Series` Objekte haben wir bereites viele der unterstützten Operationen Vergleichsoperatoren kennengelernt. Damit lassen sich Daten nun Filtern, in dem wir aus dem DateFrame bzw. Spalten des DataFrames mit Vergleichen Series-Objekte berechnen, mit denen wir Zeilen an- und abwählen können:

```
# df sei DataFrame von vorher

df['a1']            # Series erste Spalte
xs = df['a1'] < 5   # Series mit True/False
df = df[ xs ]       # df mit 1. und 3. Zeile
```

die zweite Zeile berechnet aus `df['a1']` mit dem Operator < und der Zahl 5 ein neues Series-Objekt `xs`, das für jede Zeile, in der der Wert in `a1` kleiner als 5 ist `True` enthält, und `False` sonst.
Abgekürzt lässt sich das schreiben als:

```
df = df[ df['a1'] < 5 ]
```

Einfache Aggregationsfunktionen

Aggregieren von Daten

Häufig interessieren Daten nicht ganz im Detail, sondern auf einer anderen Aggregationsebene. Zum Beispiel sei die folgende Tabelle von Bestellungen aus einem Online-Shop gegeben (Ausschnitt der Daten):

Datum	BestellungNr	KundeNr	Anzahl_Gold	Anzahl_Rot	Anzahl_Blau	Anzahl_Gruen	Betrag
2023-10-12	77	78	1	0	0	0	0.050
2023-10-12	78	79	1	2	0	0	0.150
2023-10-12	79	80	1	1	1	0	0.150
2023-10-12	80	81	1	0	1	1	0.150
2023-10-12	81	82	1	1	1	0	0.150
2023-10-12	82	83	0	0	1	0	0.050

Aus betriebswirtschaftlicher Sicht (Marketing, Controlling,..) interessieren uns Fragen wie z.B.

- Welches Produkt wurde insgesamt am häufigsten bestellt?
- Wie hoch ist der durchschnittliche Betrag einer Bestellung?

Aggregieren von Daten

Pandas unterstützt eine Vielzahl von sogenannten Aggregationsfunktionen, die eine Menge von Daten(zeilen) bekommen und einen einzelnen Wert bzw. eine zusammenfassende Zeile zurückliefern.
Zum Beispiel ergibt der folgenden Aufruf mit dem DataFrame der Online-Bestellungen ein Series Objekt, dass für alle numerischen Spalten die aufsummiert Zeilen enthält:

```
summen = bestellungen.sum()
```



Datum	**2023-10-122023..**
BestellungNr	**39358**
KundeNr	**39469**
Artikel_Gold	**55**
Artikel_Rot	**40**
Artikel_Blau	**29**
Artikel_Gruen	**27**
Betrag	**25.450000000000003**

summen

Aggregationsfunktionen

Wenn die Summen (oder andere Aggregationsfunktionen) berechnet wurden, kann mit dem Ergebnis natürlich weitergerechnet werden.

```
summen = bestellungen.sum()
```



Datum	2023-10-122023..
BestellungNr	39358
KundeNr	39469
Artikel_Gold	55
Artikel_Rot	40
Artikel_Blau	29
Artikel_Gruen	27
Betrag	25.450000000000003

summen

Aus dem Ergebnis von `bestellungen.sum()` lassen sich z.B. auch wieder die einzelnen Werte ausgelesen werde – es handelt sich ja um ein Series Objekt:

```
anzahlGold = summen['Artikel_Gold']
```

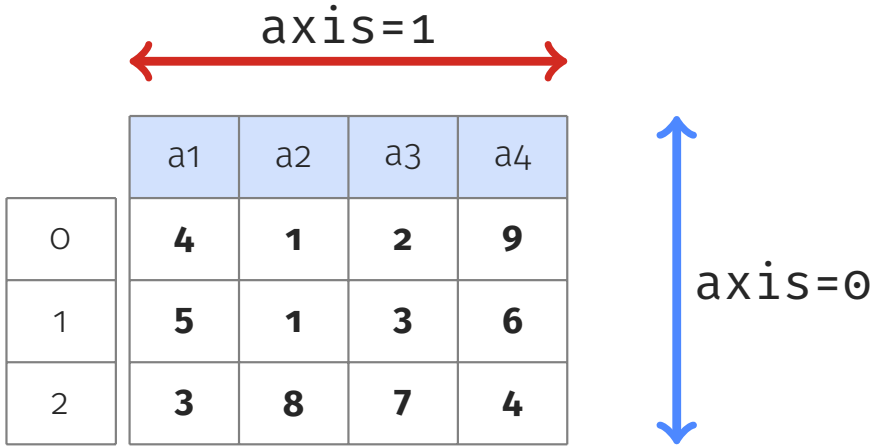
Aggregieren von Daten

In dem Beispiel mit den Bestellungen sieht man, dass die Werte der nicht-numerischen Spalten (z.B. Datum, Uhrzeit) einfach hintereinander gehängt werden, d.h. es entsteht in diesen Fällen ein sehr langer Text aus Datumsangaben.
Neben der Aggregationsfunktion `.sum()` existieren noch viele weitere Funktionen, z.B.

- `count()` – die Anzahl der Werte (NaN=fehlender Wert wird nicht gezählt!)
- `nunique()` – die Anzahl unterschiedlicher Werte

Achsen eines DataFrames

Die meisten Aggregationen sind sowohl für eine zeilen- als auch eine spaltenweise Berechnung sinnvoll. So könnte mit `.sum()` ja auch die Summe der Spalten gemeint sein. Pandas definiert dafür die *Achsen* eines DataFrames:



Der Parameter `axis` kann z.B. beim Aufruf der `.sum()` funktion mit angegeben werden um zu bestimmen, ob die Zeilen- oder die Spaltensumme berechnet werden soll:

```
spalten_summe = df.sum(axis=1)
```

Filtern mit Aggregationen

Natürlich können die Filter-Funktionen mit den Aggregationsfunktionen sehr hilfreich kombiniert werden. Um in der Tabelle mit den Bestellungen die Summen (Anzahl je Produkt, Betrag,...) für den Kunden Nr. 53 zu bekommen, reicht ein einfacher Befehl aus:

```
# Bestellungen von Kunde 53 rausfiltern:
kunde53 = bestellungen[bestellungen['KundeNr'] == 53]

# auf DataFrame kunde53 die Zeilensumme berechnen:
statistik_kunde53 = kunde53.sum()
```