Wirtschaftsinformatik 2

Pandas DataFrames 2

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

Hochschule Bochum

Wintersemester 2026/2027

Vorbemerkungen zum Block Pandas DataFrames GroupBy

In den letzten beiden Blöcken haben wir uns intensiv mit der `DataFrame` Struktur beschäftigt. Zuletzt ging es dabei um das Filtern und Aggregieren von Daten. In diesem Abschnitt geht es nun einen Schritt weiter in dem wir Daten gruppiert zusammenfassen wollen. Etwas sehr ähnliches haben wir bereits in der Wirtschaftsinformatik 1 Vorlesung gemacht, als es bei den Datenbank (SQL) um das Thema *GROUP BY* ging.
Das entsprechende `groupby` in Pandas tut genau das gleiche, lässt sich jedoch noch leicht um weitere (eigene) Aggregationsfunktionen erweitern. Wir werden in diesem Teil jedoch nur mit den grundlegenden Funktionen rechnen.

Der zweite Teil dieses Blocks ist die Behandlung von Datumsangaben. Diese enthalten oft Zeitzonen-Informationen und es ist durchaus wichtig zu wissen, wie Zeit im Computer behandelt wird und auf welche Dinge man dort achten muss.

DataFrames bieten Funktionen für Aggregate

Wir betrachten einen Datensatz über Kunden aus dem Schoko Online-Shop. Die Daten wurden beim BO Familientag erhoben und enthalten Informationen über die Kundengruppe, Altersgruppe sowie Anzahl der bestellten Artikel und den Gesamtbetrag der Bestellung:

KundeNr	Kundengruppe	Altersgruppe	Artikel	Betrag
282	Schulkind	6-10	4	0.15
269	Mitarbeiter	41-50	2	0.1
261	KigaKind	0-5	2	0.05
280	Schulkind	6-10	3	0.1

Aus betriebswirtschaftlicher Sicht sind wir z.B. interessiert an Fragen wie

- Wie war die Gesamtzahl an Einkäufen?
- Wie ist die gesamte Anzahl verkaufter Produkte?

Der Datensatz ist über eine URL als CSV-Datei verfügbar und kann einfach in einen DataFrame eingelesen werden:

```
import pandas as pd

kd = pd.read_csv('https://data.hsbo.de/kunden.csv')
```

Mit den Aggregationsfunktionen, wie z.B. `sum()` können wir auf einfache Weise die Spaltensummen berechnen (vgl. letzter Foliensatz):

```
summe = kd.sum()
```

Wenn wir das mit Filtern kombinieren, lassen sich so auch Teilsummen z.B. für eine bestimmte Kundengruppe berechnen:

```
kinder = kd[ kd['Kundengruppe'] == 'KigaKind' ]
summe_kinder = kinder.sum()
```

Häufige Fragestellung: Wie ist die Summe pro Kategorie?

In der Regel möchten wir eine Auswertung wie z.B. der Gesamtumsatz allerdings nicht nur für einen Teil (KigaKind), sondern für alle Teile einer Kategorie.
Soetwas hatten wir in Wirtschaftsinformatik 1 bereits für Datenbanken kennengelernt (GROUP BY).
Auch Pandas DataFrames unterstützen eine Aufteilung/Gruppierung nach Spalten/Werten:

```
gruppiert = kd.groupby("Kundengruppe")
# das Ergebnis ist vom Typ DataFrameGroupBy
```

Attribut `.groups` enthält die Gruppen und zugehörigen Zeilennummern:

```
print(gruppiert.groups)
#
# {'KigaKind': [7, 13,..],'Student': [0, 2,.],.. }
```

Jede Gruppe repräsentiert einen eigenen DataFrame

KundeNr	Kundengruppe	Altersgruppe	Artikel	Betrag
282	Schulkind	6-10	4	0.15
269	Mitarbeiter	41-50	2	0.1
261	KigaKind	0-5	2	0.05
280	Schulkind	6-10	3	0.1

name='Student'

KundeNr	Kundengruppe	Altersgruppe	Artikel	Betrag
244	Student	21-30	2	0.05
267	Student	11-20	4	0.05
235	Student	21-30	2	0.05

name='KigaKind'

KundeNr	Kundengruppe	Altersgruppe	Artikel	Betrag
261	KigaKind	0-5	2	0.05
251	KigaKind	0-5	3	0.1
283	KigaKind	0-5	3	0.1
253	KigaKind	0-5	1	0.05

name='Mitarbeiter'

KundeNr	Kundengruppe	Altersgruppe	Artikel	Betrag
272	Mitarbeiter	41-50	3	0.1
236	Mitarbeiter	41-50	2	0.05
269	Mitarbeiter	41-50	2	0.1
240	Mitarbeiter	31-40	1	0.05

Dabei erhält jeder Teil-DataFrame einen Namen, der dem Wert der jeweiligen Gruppe entspricht, also in unserem Fall *Student*, *KigaKind*, usw.

Aggregate auf DataFrameGroupBy liefern DataFrame

Auf der Gruppierung lassen sich nun Aggregationsfunktionen wie `sum()` anwenden, die dann für jeden Teil-DataFrame separat berechnet werden.

```
# Kategorie hat die Werte 'KigaKind', 'Schulkind',...
gruppiert = kd.groupby("Kundengruppe")
gruppiert.sum()
```

Das Ergebnis ist ein `DataFrame`:

	KundeNr	Altersgruppe	Artikel	Betrag
Gast	742	31-4021-3041-50	8	0.3
KigaKind	1751	0-50-5..	19	0.6
Mitarbeiter	3171	41-5041-5..	29	0.9
Schulkind	5059	6-100-511-2..	59	2.1
Student	746	21-3..	8	0.15

Aggregate auf DataFrames

Die Summenberechnung ergibt für Spalten, die keine Zahlen, sondern Text enthalten (z.B. *Altersgruppe*) natürlich nicht viel Sinn. In dem aggregierten DataFrame sieht man z.B. in der Spalte Altersgruppe alle Werte hintereinander geschrieben:

	KundeNr	Altersgruppe	Artikel	Betrag
Gast	742	31-4021-3041-50	8	0.3
KigaKind	1751	0-50-5..	19	0.6
Mitarbeiter	3171	41-5041-5..	29	0.9
Schulkind	5059	6-100-511-2..	59	2.1
Student	746	21-3..	8	0.15

Hier ist es sinnvoll, die entsprechenden Spalten vorab herauszufiltern, z.B. indem nur die Spalte(n) für die Gruppierung und die Aggregation ausgewählt werden:

```
kunden = kd.loc[:,['Kundengruppe', 'Artikel', 'Betrag']]
```

Einfache Aggregat-Funktionen

Natürlich funktionieren auch andere Aggregationsfunktionen wie z.B. `count()` oder `mean()`. Wir betrachten wegen der besseren Darstellung wieder nur die Spalten *Kundengruppe*, *Artikel* und *Betrag*:

```
kunden = kd.loc[:,['Kundengruppe', 'Artikel', 'Betrag']]
```

```
grps = kunden.groupby('Kundengruppe')
grps.count()
```

	Artikel	Betrag
Gast	3	3
KigaKind	7	7
Mitarbeiter	13	13
Schulkind	19	19
Student	3	3

```
grps = kunden.groupby("Kategory")
grps.mean()
```

	Artikel	Betrag
Gast	2.67	0.1
KigaKind	2.71	0.09
Mitarbeiter	2.23	0.07
Schulkind	3.11	0.11
Student	2.67	0.05

Pandas und Datumsangaben

Datumsangaben (Hintergrund)

Wir haben im Rahmen den bisherigen Kapiteln verschiedene Datentypen kennengelernt. Die einfachsten waren `int` für ganze Zahlen und `str` für Text-Werte. Mit Werten vom Typ `int` sind arithmetische Operationen (+,-,*,/) usw. möglich - bei Textwerten können wir auf einzelne Buchstaben zugreifen oder ähnliches.
Um mit Datumsangaben umzugehen, benötigen wir ein Konzept, wie Zeitpunkte im Computer dargestellt werden können. Die zentrale Idee dazu ist, Zeitpunkte in Abhängigkeit von einem fixen Datum festzulegen. Damit kann jeder Zeitpunkt z.B. als Anzahl der Sekunden seit diesem Fixpunkt gespeichert werden[1].
Der Fixpunkt für die Zeitdarstellung im Computer ist der 1.1.1970 um 0:00 Uhr.



[1] vgl. `https://de.wikipedia.org/wiki/Unixzeit`

Datumsangaben (Hintergrund)

Wir haben im Rahmen den bisherigen Kapiteln verschiedene Datentypen kennengelernt. Die einfachsten waren `int` für ganze Zahlen und `str` für Text-Werte. Mit Werten vom Typ `int` sind arithmetische Operationen (+,-,*,/) usw. möglich - bei Textwerten können wir auf einzelne Buchstaben zugreifen oder ähnliches.
Um mit Datumsangaben umzugehen, benötigen wir ein Konzept, wie Zeitpunkte im Computer dargestellt werden können. Die zentrale Idee dazu ist, Zeitpunkte in Abhängigkeit von einem fixen Datum festzulegen. Damit kann jeder Zeitpunkt z.B. als Anzahl der Sekunden seit diesem Fixpunkt gespeichert werden[1].
Der Fixpunkt für die Zeitdarstellung im Computer ist der 1.1.1970 um 0:00 Uhr.



[1] vgl. `https://de.wikipedia.org/wiki/Unixzeit`

Datumsangaben (Hintergrund)

Wir haben im Rahmen den bisherigen Kapiteln verschiedene Datentypen kennengelernt. Die einfachsten waren `int` für ganze Zahlen und `str` für Text-Werte. Mit Werten vom Typ `int` sind arithmetische Operationen (+,-,*,/) usw. möglich - bei Textwerten können wir auf einzelne Buchstaben zugreifen oder ähnliches.
Um mit Datumsangaben umzugehen, benötigen wir ein Konzept, wie Zeitpunkte im Computer dargestellt werden können. Die zentrale Idee dazu ist, Zeitpunkte in Abhängigkeit von einem fixen Datum festzulegen. Damit kann jeder Zeitpunkt z.B. als Anzahl der Sekunden seit diesem Fixpunkt gespeichert werden[1].
Der Fixpunkt für die Zeitdarstellung im Computer ist der 1.1.1970 um 0:00 Uhr.



[1] vgl. `https://de.wikipedia.org/wiki/Unixzeit`

Datum im Computer – Unixzeit

Traditionell wird die Unix-Zeit in Sekunden angegeben. Für eine genauere Zeitauflösung werden häufig auch Milli- oder Nanosekunden benutzt. Geht man von Sekunden aus, hat man für eine Stunde 3600 Sekunden und somit für einen Tag $24 \cdot 3600 = 86400$ Sekunden:

```
# 18.11.2024 um 9:00 Uhr
t = 1731924000

t2 = t + 86400
# t2 entspricht 18.11.2024 um 10:00 Uhr
```

Natürlich ist ein Wert wie 1731924000 nicht wirklich intuitiv für das menschliche Auge nutzbar. Daher benutzt Pandas den Datentyp `datetime64`, der einen Zeitpunkt darstellt.

Datum in Pandas

In Daten finden sich daher häufig Darstellungen z.B. in der Form `18.11.2024 10:00:00`. Mit der Funktion `to_datetime(..)` lässt sich daraus ein `datetime64` Wert berechnen:

```
import pandas as pd

datum = pd.to_datetime("18.11.2024 10:00:00")
```

Die Variable `datum` enthält jetzt den `datetime64` Wert, über den z.B. das Jahr, der Monat, Tag usw. abgefragt werden kann:

```
tag = datum.day                # tag ist 18
tage = datum.days_in_month     # tage ist 30 (November)
stunde = datum.hour            # stunde hat den Wert 10
```

Datum in Pandas und Zeitzonen

Wir können jetzt also mit der Funktion `to_datetime(..)` aus einem Text der ein Datum enthält einen *richtigen* Datumswert machen. Betrachten wir nochmal das Datum:

18.11.2024 10:00

Dieser Wert wird von Python intern auf die Anzahl der Millisekunden seit dem 1.1.1970 00:00 Uhr umgerechnet. Aber in welcher Zeitzone?

Der Referenzwert 1.1.1970 00:00 Uhr ist in UTC (*coordinated universal time*), was der Zeitzone von London, UK entspricht. Wenn es sich bei der obigen Zeit um die Deutsche Zeit handelt (Sommerzeit? Winterzeit?) ist die Anzahl der Millisekunden ggf. um 1 oder 2 Stunden verschoben.



Datum in Pandas und Zeitzonen

Wir können jetzt also mit der Funktion `to_datetime(..)` aus einem Text der ein Datum enthält einen *richtigen* Datumswert machen. Betrachten wir nochmal das Datum:

18.11.2024 10:00

Dieser Wert wird von Python intern auf die Anzahl der Millisekunden seit dem 1.1.1970 00:00 Uhr umgerechnet. Aber in welcher Zeitzone?

Der Referenzwert 1.1.1970 00:00 Uhr ist in UTC (*coordinated universal time*), was der Zeitzone von London, UK entspricht. Wenn es sich bei der obigen Zeit um die Deutsche Zeit handelt (Sommerzeit? Winterzeit?) ist die Anzahl der Millisekunden ggf. um 1 oder 2 Stunden verschoben.



Datum und Zeitzonen

Um eine vollständige Zeitangabe zu haben, muss klar sein, in welcher Zeitzone die Zeitangabe gemacht wird. Dazu gibt es unterschiedliche Formate, z.B.:

18.11.2024 10:00 +01:00

Dabei ist die Zeitverschiebung mit +01 Stunde angegeben - es handelt sich also in diesem Fall um 10 Uhr deutsche Zeit, was z.B. 9 Uhr London-Zeit entspricht. Wenn wir mit `to_datetime(..)` ein Datum einlesen, wird die lokal konfigurierte Zeitzone angenommen. Mit `utc=True` erzwingen wir UTC Zeit:

```
text = "18.11.2024 10:00 +01:00"

datum = pd.to_datetime(text)
datum.hour           # 10 Uhr
datum.timestamp()    # 1731924000

utc = pd.to_datetime(text, utc=True)
utc.hour             #  9 Uhr
utc.timestamp()      # 1731924000
```

Datum in DataFrames

Häufig haben wir Datumsangaben in Tabellenspalten, die beim Einlesen erstmal nur reiner Text sind. Mit `to_datetime(..)` können wir aus einer Textspalte mit Datum eine richtige Datumsspalte berechnen. Hier am Beispiel der Bitcoin Kursdaten:

```
btc = pd.read_csv('https://data.hsbo.de/bitcoin.csv')
```

Daraus ergibt sich der folgende DataFrame `btc`, der in der Spalte *Date* jedoch lediglich Text und keinen richtigen Datumswert enthält:

	Date	Open	High	Low	Close
0	**2024-10-25**	**62946**	**63490**	**61020**	**61770**
1	**2024-10-24**	**61757**	**63050**	**61485**	**62930**
2	**2024-10-23**	**62475**	**62748**	**60525**	**61737**
3	**2024-10-22**	**62573**	**62707**	**61555**	**62500**

Datum in DataFrames

Mit `to_datetime(..)` lässt sich aus der gesamten Spalte *Date* eine richtige Datumsspalte extrahieren:

```
datums_spalte = pd.to_datetime( btc['Date'] )
btc['Datum'] = datums_spalte
```

Jetzt enthält jede Zelle der neuen Spalte *Datum* einen richtigen Datumswert vom Typ `datetime64`:

	Date	Open	High	Low	Close	Datum
0	2024-10-25	62946	63490	61020	61770	2024-10-25 00:00:00
1	2024-10-24	61757	63050	61485	62930	2024-10-24 00:00:00
2	2024-10-23	62475	62748	60525	61737	2024-10-23 00:00:00
3	2024-10-22	62573	62707	61555	62500	2024-10-22 00:00:00

Datum in DataFrames

Da die neue Spalte *Datum* nun vom Typ `datetime64` ist, können wir mit Hilfe der Eigenschaft `.dt` auf dieser Spalte nun Bestandteile des Datums berechnen, z.B. das Jahr, den Monat, usw.
Da wir diese Berechnungen auf der gesamten Spalte (Series Objekt) machen, bekommen wir auch immer wieder eine neue Spalte als Ergebnis, die wir zu unserem DataFrame hinzufügen können:

```
jahr = btc['Datum'].dt.year
btc['Jahr']  = jahr                          # Spalte 'Jahr'

# alternativ ohne extra Variable:
btc['Monat'] = btc['Datum'].dt.month   # neue Spalte 'Monat'

# Wochentag (0=Montag, 1=Dienstag,...)
btc['Wochentag'] = btc['Datum'].dt.weekday
```

Datum in DataFrames

Auf diese Weise können wir den DataFrame um eine ganze Reihe neuer Spalten erweitern:

	Date	Open	High	Low	Close	Datum	Jahr	Wochentag
0	2024-10-25	62946	63490	61020	61770	2024-10-25 00:00:00	2024	4
1	2024-10-24	61757	63050	61485	62930	2024-10-24 00:00:00	2024	3
2	2024-10-23	62475	62748	60525	61737	2024-10-23 00:00:00	2024	2
3	2024-10-22	62573	62707	61555	62500	2024-10-22 00:00:00	2024	1

Das eignet sich dann wiederum zum Filtern und Aggregieren:

```
# Wie sind die Durchschnittskurse an Dienstagen?
#
btc_dienstags = btc[ btc['Wochentag'] == 1 ]
btc_dienstags.mean()
```

Kombination mit groupby(..)

Die neu berechneten Spalten lassen sich wie alle anderen Spalten weiter mitbenutzen. Nachdem wir nun die Durchschnittskurse an Dienstagen berechnet haben, bietet es sich natürlich an, dies mit der `groupby(..)` Funktion aus dem ersten Teil des Foliensatzes zu kombinieren:

```
btc.groupby('Wochentag').mean()
```

	Open	High	Low	Close
0	56245.0	57388.46	54550.23	55943.31
1	55946.77	57125.85	54950.38	56039.38
2	56035.31	56800.23	54578.08	55573.38
3	55564.92	56840.92	54574.38	55852.38

Dabei entsprechen die Index-Werte natürlich den Werten der Spalte *Wochentag*, also 0=Montag, 1=Dienstag,...

Kombination mit groupby(..)

Das Beispiel auf der letzten Folie ist ein wenig unglücklich, weil die Wochentage natürlich als Zahlenwert nicht wirklich gut lesbar sind. `Series` hat allerdings eine Funktion `replace(..)` mit der sich die Spalte *Wochentag* etwas umbauen lässt:

```
btc['Wochentag'] = btc['Wochentag'].replace(
   { 0: 'Mo', 1: 'Di', 2: 'Mi', 3: 'Do', 4: 'Fr',
     5: 'Sa', 6: 'So' }  )
```

Damit ersetzen wir die Zahlen in der Spalte *Wochentag* durch Mo/Di/Mi usw.:

	Wochentag	Open	High	Low	Close
0	Fr	62946	63490	61020	61770
1	Do	61757	63050	61485	62930
2	Mi	62475	62748	60525	61737
3	Di	62573	62707	61555	62500

Wenn wir auf der Spalte *Wochentag* mit den neuen Werten jetzt nochmal gruppieren, erhalten wir ein deutlich besser verständliches Ergebnis:

```
btc.groupby('Wochentag').mean()
```

	Open	High	Low	Close
Di	55946.77	57125.85	54950.38	56039.38
Do	55564.92	56840.92	54574.38	55852.38
Fr	56181.43	57882.79	55399.86	56837.93
Mi	56035.31	56800.23	54578.08	55573.38
Mo	56245.0	57388.46	54550.23	55943.31
Sa	56454.38	57071.69	55918.15	56424.31
So	56419.62	56930.08	55607.69	56243.92