05_pandas_filled

October 18, 2020

0.1 Einführung Pandas

Pandas kann man sich als eine sehr mächtige Version von Excel vorstellen, welche viel mehr Werkzeuge liefert. Hier behandeln: - Series - Dataframes - Missing Data - GroupBy - Merging, Joining und Concatenating - Operationen - Dateneingabe und -ausgabe

** Auch hier müssen wir Pandas erstmal installieren! **

Dazu Anaconda Comand Prompt -> via poetry: `poetry add pandas`

0.1.1 Series

Eine Folge in Pandas ist fast genau wie ein Numpy Array und fast genau wie eine Liste mit einem individuellen Index. Allerdings erlaubt eine Pandas Serie im Gegensatz zu einem Numpy Array, dass die Achsen beschriftet bzw gelabelt sein können. Es kann zudem jedes beliebige Python Objekt beinhalten.

[1]:
```
# imports

import numpy as np
import pandas as pd
```

[2]:
```
labels = ["a","b","c"]
liste = [1,2,3]
arr = np.array([10,20,30])
d = {"a":10, "b":20, "c":30}
```

Pandas Serie aus einer Liste

[3]:
```
pd.Series(data=liste)
```

[3]:
```
0    1
1    2
2    3
dtype: int64
```

[4]:
```
pd.Series(data=liste,index=labels)
```

[4]:
```
a    1
b    2
c    3
dtype: int64
```

[5]:
```
# hier nochmal: parameter-namen müssen nicht angegeben werden, wenn man die␣
 ↪reihenfolge der parameter einhält!
pd.Series(liste,labels)
```

[5]:
```
a    1
b    2
c    3
dtype: int64
```

Pandas Serie aus einem Numpy Array

[6]:
```
pd.Series(arr)
```

[6]:
```
0    10
1    20
2    30
dtype: int64
```

[7]:
```
pd.Series(arr,labels)
```

[7]:
```
a    10
b    20
c    30
dtype: int64
```

Pandas Serie aus einem Dictionary

[8]:
```
pd.Series(d)
```

[8]:
```
a    10
b    20
c    30
dtype: int64
```

Index Die Möglichkeit Indexs zu verwenden macht Pandas Serien sehr nützlich. Beispiel:

[9]:
```
serie = pd.Series([1,2,3,4], index=["Sven", "Arthur", "Rainer", "Georg"])
```

[10]:
```
serie
```

[10]:
```
Sven      1
Arthur    2
Rainer    3
Georg     4
dtype: int64
```

[11]:
```
serie2 = pd.Series([1,2,5,4], index = ["Sven", "Arthur", "Rainer", "Georg"])
```

[12]:
```
serie2
```

[12]:
```
Sven      1
Arthur    2
Rainer    5
Georg     4
dtype: int64
```

[13]:
```
serie2["Georg"]
```

[13]:
```
4
```

Operationen orientieren sich am Index

[14]:
```
serie2 + serie
```

[14]:
```
Sven      2
Arthur    4
Rainer    8
Georg     8
dtype: int64
```

0.2 Pandas DataFrames

DataFrames ist das wohl wichtigste Konzept, wenn es um Data Science mit Python geht. Im Grunde sind DataFrames mehrere einzelne Serien welche den gleichen Index teilen und somit eine Art Tabelle (DataFrame) bilden.

[15]:
```
rng=np.random.default_rng()
```

[16]:
```
df = pd.DataFrame(rng.random((6,5)), index='A B C D E F'.split(), columns="V W X␣
↪Y Z".split())
```

[17]:
```
df
```

[17]:

	V	W	X	Y	Z
A	0.355577	0.105731	0.103734	0.182056	0.523912
B	0.555777	0.550569	0.781812	0.820208	0.603625
C	0.905441	0.071661	0.547346	0.429186	0.999948

```
D  0.784713  0.827490  0.143071  0.222107  0.526393
E  0.312108  0.594676  0.324236  0.306128  0.250516
F  0.853728  0.466379  0.624880  0.557055  0.184242
```

Selecting und Indexing

[18]:
```
df["X"]
```

[18]:
```
A    0.103734
B    0.781812
C    0.547346
D    0.143071
E    0.324236
F    0.624880
Name: X, dtype: float64
```

Gibt uns die Spalte mit dem Index X zurück, welche nichts anderes als eine Pandas Serie ist!

[19]:
```
# man kann auch gleich mehrere Spalten ausgeben lassen, indem man eine Liste mit␣
 ↪den Indexen übergibt
df[["W","X"]]
```

[19]:
```
          W         X
A  0.105731  0.103734
B  0.550569  0.781812
C  0.071661  0.547346
D  0.827490  0.143071
E  0.594676  0.324236
F  0.466379  0.624880
```

[20]:
```
# es gibt auch eine andere Syntax, welche sich and SQL orientiert
df.W
# Allerdings werden wir diesen nicht verwenden!
```

[20]:
```
A    0.105731
B    0.550569
C    0.071661
D    0.827490
E    0.594676
F    0.466379
Name: W, dtype: float64
```

Neue Spalte erzeugen

[21]:
```
df["neu"] = df["X"] + df["Z"]
```

[22]:
```
df["neu"]
```

[22]:
```
A    0.627646
B    1.385437
C    1.547294
D    0.669464
E    0.574752
F    0.809122
Name: neu, dtype: float64
```

[23]:
```
df
```

[23]:

	V	W	X	Y	Z	neu
A	0.355577	0.105731	0.103734	0.182056	0.523912	0.627646
B	0.555777	0.550569	0.781812	0.820208	0.603625	1.385437
C	0.905441	0.071661	0.547346	0.429186	0.999948	1.547294
D	0.784713	0.827490	0.143071	0.222107	0.526393	0.669464
E	0.312108	0.594676	0.324236	0.306128	0.250516	0.574752
F	0.853728	0.466379	0.624880	0.557055	0.184242	0.809122

Spalten löschen

[24]:
```
df.drop("neu",axis=1)
```

[24]:

	V	W	X	Y	Z
A	0.355577	0.105731	0.103734	0.182056	0.523912
B	0.555777	0.550569	0.781812	0.820208	0.603625
C	0.905441	0.071661	0.547346	0.429186	0.999948
D	0.784713	0.827490	0.143071	0.222107	0.526393
E	0.312108	0.594676	0.324236	0.306128	0.250516
F	0.853728	0.466379	0.624880	0.557055	0.184242

[25]:
```
# es wird nur aus der Ansicht gelöscht, aber nicht aus dem eigentlichen␣
↪DataFrame (das muss explizit spezifiziert werden)!
df
```

[25]:

	V	W	X	Y	Z	neu
A	0.355577	0.105731	0.103734	0.182056	0.523912	0.627646
B	0.555777	0.550569	0.781812	0.820208	0.603625	1.385437
C	0.905441	0.071661	0.547346	0.429186	0.999948	1.547294
D	0.784713	0.827490	0.143071	0.222107	0.526393	0.669464
E	0.312108	0.594676	0.324236	0.306128	0.250516	0.574752
F	0.853728	0.466379	0.624880	0.557055	0.184242	0.809122

[26]:
```
# um die Spalte 'komplett' zu löschen muss der inplace-Parameter auf True␣
↪gesetzt werden
df.drop("neu",axis=1,inplace=True)
```

[27]:
```
df
```

[27]:

	V	W	X	Y	Z
A	0.355577	0.105731	0.103734	0.182056	0.523912
B	0.555777	0.550569	0.781812	0.820208	0.603625
C	0.905441	0.071661	0.547346	0.429186	0.999948
D	0.784713	0.827490	0.143071	0.222107	0.526393
E	0.312108	0.594676	0.324236	0.306128	0.250516
F	0.853728	0.466379	0.624880	0.557055	0.184242

Zeilen löschen Funktioniert genauso, nur muss der axis-Parameter auf 0 gesetzt werden

[28]:
```
df.drop("F",axis=0)
```

[28]:

	V	W	X	Y	Z
A	0.355577	0.105731	0.103734	0.182056	0.523912
B	0.555777	0.550569	0.781812	0.820208	0.603625
C	0.905441	0.071661	0.547346	0.429186	0.999948
D	0.784713	0.827490	0.143071	0.222107	0.526393
E	0.312108	0.594676	0.324236	0.306128	0.250516

Reihen auswählen

[29]:
```
df.loc["A"]
```

[29]:
```
V    0.355577
W    0.105731
X    0.103734
Y    0.182056
Z    0.523912
Name: A, dtype: float64
```

[30]:
```
# man kann auch die Position, anstelle des Index-Namens benutzen
df.iloc[4]
```

[30]:
```
V    0.312108
W    0.594676
X    0.324236
Y    0.306128
Z    0.250516
Name: E, dtype: float64
```

Teilmengen von Zeilen und Spalten auswählen

[31]:
```
df.loc["B","Z"]
```

[31]:
```
0.6036251114966996
```

[32]:
```
df.loc[["A","B"],["Y","Z"]]
```

[32]:

	Y	Z
A	0.182056	0.523912
B	0.820208	0.603625

Bedingte Auswahl

[33]:
```
df>0
```

[33]:

	V	W	X	Y	Z
A	True	True	True	True	True
B	True	True	True	True	True
C	True	True	True	True	True
D	True	True	True	True	True
E	True	True	True	True	True
F	True	True	True	True	True

[34]:
```
df[df>0]
```

[34]:

	V	W	X	Y	Z
A	0.355577	0.105731	0.103734	0.182056	0.523912
B	0.555777	0.550569	0.781812	0.820208	0.603625
C	0.905441	0.071661	0.547346	0.429186	0.999948
D	0.784713	0.827490	0.143071	0.222107	0.526393
E	0.312108	0.594676	0.324236	0.306128	0.250516
F	0.853728	0.466379	0.624880	0.557055	0.184242

[35]:
```
df[df["V"]>0]
```

[35]:

	V	W	X	Y	Z
A	0.355577	0.105731	0.103734	0.182056	0.523912
B	0.555777	0.550569	0.781812	0.820208	0.603625
C	0.905441	0.071661	0.547346	0.429186	0.999948
D	0.784713	0.827490	0.143071	0.222107	0.526393
E	0.312108	0.594676	0.324236	0.306128	0.250516
F	0.853728	0.466379	0.624880	0.557055	0.184242

[36]:
```
df[df["V"]>0]["Z"]
```

[36]:
```
A    0.523912
B    0.603625
C    0.999948
D    0.526393
E    0.250516
F    0.184242
Name: Z, dtype: float64
```

[37]:
```
df[df["V"]>0][["Y","Z"]]
```

[37]:

	Y	Z
A	0.182056	0.523912
B	0.820208	0.603625
C	0.429186	0.999948
D	0.222107	0.526393
E	0.306128	0.250516
F	0.557055	0.184242

Weitere Möglichkeiten

[38]:
```
df
```

[38]:

	V	W	X	Y	Z
A	0.355577	0.105731	0.103734	0.182056	0.523912
B	0.555777	0.550569	0.781812	0.820208	0.603625
C	0.905441	0.071661	0.547346	0.429186	0.999948
D	0.784713	0.827490	0.143071	0.222107	0.526393
E	0.312108	0.594676	0.324236	0.306128	0.250516
F	0.853728	0.466379	0.624880	0.557055	0.184242

[39]:
```
# Index zurücksetzen auf 0,1,...,n Index
df.reset_index()
# wieder nicht inplace, da Parameter nicht gesetzt
```

[39]:

	index	V	W	X	Y	Z
0	A	0.355577	0.105731	0.103734	0.182056	0.523912
1	B	0.555777	0.550569	0.781812	0.820208	0.603625
2	C	0.905441	0.071661	0.547346	0.429186	0.999948
3	D	0.784713	0.827490	0.143071	0.222107	0.526393
4	E	0.312108	0.594676	0.324236	0.306128	0.250516
5	F	0.853728	0.466379	0.624880	0.557055	0.184242

[40]:
```
index_neu = "BW BY HE TH RP SAA".split()
```

[41]:
```
df['Laender'] = index_neu
```

[42]:
```
df
```

[42]:

	V	W	X	Y	Z	Laender
A	0.355577	0.105731	0.103734	0.182056	0.523912	BW
B	0.555777	0.550569	0.781812	0.820208	0.603625	BY
C	0.905441	0.071661	0.547346	0.429186	0.999948	HE
D	0.784713	0.827490	0.143071	0.222107	0.526393	TH
E	0.312108	0.594676	0.324236	0.306128	0.250516	RP
F	0.853728	0.466379	0.624880	0.557055	0.184242	SAA

[43]:
```
# Laender als Index auswählen
df.set_index("Laender")
```

[43]:

Laender	V	W	X	Y	Z
BW	0.355577	0.105731	0.103734	0.182056	0.523912
BY	0.555777	0.550569	0.781812	0.820208	0.603625
HE	0.905441	0.071661	0.547346	0.429186	0.999948
TH	0.784713	0.827490	0.143071	0.222107	0.526393
RP	0.312108	0.594676	0.324236	0.306128	0.250516
SAA	0.853728	0.466379	0.624880	0.557055	0.184242

[44]:
```
df
```

[44]:

	V	W	X	Y	Z	Laender
A	0.355577	0.105731	0.103734	0.182056	0.523912	BW
B	0.555777	0.550569	0.781812	0.820208	0.603625	BY
C	0.905441	0.071661	0.547346	0.429186	0.999948	HE
D	0.784713	0.827490	0.143071	0.222107	0.526393	TH
E	0.312108	0.594676	0.324236	0.306128	0.250516	RP
F	0.853728	0.466379	0.624880	0.557055	0.184242	SAA

[45]:
```
df.set_index('Laender',inplace=True)
```

[46]:
```
df
```

[46]:

Laender	V	W	X	Y	Z
BW	0.355577	0.105731	0.103734	0.182056	0.523912
BY	0.555777	0.550569	0.781812	0.820208	0.603625
HE	0.905441	0.071661	0.547346	0.429186	0.999948
TH	0.784713	0.827490	0.143071	0.222107	0.526393
RP	0.312108	0.594676	0.324236	0.306128	0.250516
SAA	0.853728	0.466379	0.624880	0.557055	0.184242

0.2.1 Multi-Index und Index Hierarchie

[47]:
```
# Index Level
hoch = ["L1","L1","L1","L2","L2","L2"]
niedrig = [1,2,3,1,2,3]
index_hier = list(zip(hoch,niedrig))
index_hier = pd.MultiIndex.from_tuples(index_hier)
```

[48]:
```
index_hier
```

[48]:
```
MultiIndex([('L1', 1),
            ('L1', 2),
            ('L1', 3),
            ('L2', 1),
            ('L2', 2),
            ('L2', 3)],
           )
```

[49]:
```
df = pd.DataFrame(np.random.randn(6,2),index=index_hier,columns=["A","B"])
df
```

[49]:
```
              A         B
L1 1  0.537690 -1.118663
   2  0.940684 -0.681561
   3 -1.378170 -0.047193
L2 1  0.400464 -0.148059
   2 -0.457143  0.395144
   3  0.190999 -1.344541
```

[50]:
```
df.loc["L1"]
```

[50]:
```
          A         B
1  0.537690 -1.118663
2  0.940684 -0.681561
3 -1.378170 -0.047193
```

[51]:
```
df.loc["L1"].loc[2]
```

[51]:
```
A    0.940684
B   -0.681561
Name: 2, dtype: float64
```

[52]:
```
# andere Möglichkeit für df.loc['G1'] - cross-section
df.xs("L1")
```

[52]:
```
          A         B
1  0.537690 -1.118663
2  0.940684 -0.681561
3 -1.378170 -0.047193
```

[53]:
```
df.xs(['L1',1])
```

[53]:
```
A    0.537690
B   -1.118663
Name: (L1, 1), dtype: float64
```

0.3 Missing Data

In Data Science ist es wichtig im Schritt Data Preprocessing mit 'Missing Data' (also fehlenden Daten) entsprechend umzugehen.

[54]:
```
df = pd.DataFrame({"A":[1,2,np.nan],
                   "B":[5,np.nan,np.nan],
                   "C":[1,2,3]})
```

[55]:
```
df
```

[55]:
```
     A    B  C
0  1.0  5.0  1
1  2.0  NaN  2
2  NaN  NaN  3
```

[56]:
```
# Lösche die Zeilen, die fehlende Daten haben
df.dropna()
```

[56]:
```
     A    B  C
0  1.0  5.0  1
```

[57]:
```
# Lösche die Spaalten die fehlende Daten haben
df.dropna(axis=1)
```

[57]:
```
   C
0  1
1  2
2  3
```

[58]:
```
# Setze ein Threshold wieviele fehlende Daten geduldet werden
# thresh=2 bedeutet max ein Wert darf fehlen
df.dropna(thresh=2)
```

[58]:
```
     A    B  C
0  1.0  5.0  1
1  2.0  NaN  2
```

[59]:
```
# meistens überlegt man sich eine Strategie, wie man mit fehlenden Werten umgeht.
 ↪ Bei Zahlen könnte man sich überlegen
# den Mittelwert zu nehmen
df["A"].fillna(value=df["A"].mean())
```

[59]:
```
0    1.0
1    2.0
2    1.5
Name: A, dtype: float64
```

0.3.1 GroupBy

[60]:
```
# Daten für das DataFrame
data = {"Uni":["UniMa","UniMa","LMU","LMU","KIT","KIT"],
       "Person":["Simon","Fred","Felix","Sofie","Sarah","Celine"],
       "Spende":[2500,50000,3000,750,1500,500000]}
```

[61]:
```
df = pd.DataFrame(data)
```

[62]:
```
df
```

[62]:

	Uni	Person	Spende
0	UniMa	Simon	2500
1	UniMa	Fred	50000
2	LMU	Felix	3000
3	LMU	Sofie	750
4	KIT	Sarah	1500
5	KIT	Celine	500000

[63]:
```
by_uni = df.groupby("Uni")
```

[64]:
```
by_uni
```

[64]:
```
<pandas.core.groupby.generic.DataFrameGroupBy object at 0x1169e5a30>
```

[65]:
```
# Durchschnitt pro Uni
by_uni.mean()
```

[65]:

Uni	Spende
KIT	250750
LMU	1875
UniMa	26250

[66]:
```
# Geht natürlich auch direkt
df.groupby("Uni").mean()
```

[66]:

Uni	Spende
KIT	250750
LMU	1875
UniMa	26250

[67]:
```
# Standardabweichung
df.groupby("Uni").std()
```

[67]:
```
              Spende
Uni
KIT    352492.730421
LMU      1590.990258
UniMa   33587.572106
```

[68]:
```
# Min
df.groupby('Uni').min()
```

[68]:
```
       Person  Spende
Uni
KIT    Celine    1500
LMU     Felix     750
UniMa    Fred    2500
```

[69]:
```
# Max
df.groupby("Uni").max()
```

[69]:
```
      Person  Spende
Uni
KIT    Sarah  500000
LMU    Sofie    3000
UniMa  Simon   50000
```

[70]:
```
# Anzahl
df.groupby("Uni").count()
```

[70]:
```
       Person  Spende
Uni
KIT         2       2
LMU         2       2
UniMa       2       2
```

[71]:
```
# ganz wichtige Funktion: describe()
df.groupby("Uni").describe()
```

[71]:
```
      Spende                                                              \
       count      mean            std     min       25%       50%       75%
Uni
KIT      2.0  250750.0  352492.730421  1500.0  126125.0  250750.0  375375.0
LMU      2.0    1875.0    1590.990258   750.0    1312.5    1875.0    2437.5
UniMa    2.0   26250.0   33587.572106  2500.0   14375.0   26250.0   38125.0


             max
Uni
KIT    500000.0
```

```
LMU      3000.0
UniMa   50000.0
```

[72]:
```
df.groupby("Uni").describe().transpose()
```

[72]:
```
Uni                    KIT          LMU         UniMa
Spende count       2.000000     2.000000      2.000000
       mean   250750.000000  1875.000000  26250.000000
       std    352492.730421  1590.990258  33587.572106
       min      1500.000000   750.000000   2500.000000
       25%    126125.000000  1312.500000  14375.000000
       50%    250750.000000  1875.000000  26250.000000
       75%    375375.000000  2437.500000  38125.000000
       max    500000.000000  3000.000000  50000.000000
```

[73]:
```
df.groupby("Uni").describe().transpose()["UniMa"]
```

[73]:
```
Spende  count        2.000000
        mean     26250.000000
        std      33587.572106
        min       2500.000000
        25%      14375.000000
        50%      26250.000000
        75%      38125.000000
        max      50000.000000
Name: UniMa, dtype: float64
```

0.4 Merging, Joining und Concatenating

[74]:
```
# data für df1
df1 = pd.DataFrame({"A": ["A0", "A1", "A2", "A3"],
                    "B": ["B0", "B1", "B2", "B3"],
                    "C": ["C0", "C1", "C2", "C3"],
                    "D": ["D0", "D1", "D2", "D3"]},
                    index=[0, 1, 2,3])
```

[75]:
```
# data für df2
df2 = pd.DataFrame({"A": ["A4", "A5", "A6", "A7"],
                        "B": ["B4", "B5", "B6", "B7"],
                        "C": ["C4", "C5", "C6", "C7"],
                        "D": ["D4", "D5", "D6", "D7"]},
                         index=[4, 5, 6, 7])
```

[76]:
```
# data für df2
df3 = pd.DataFrame({"A": ["A8", "A9", "A10", "A11"],
                        "B": ["B8", "B9", "B10", "B11"],
```

```
                        "C": ["C8", "C9", "C10", "C11"],
                        "D": ["D8", "D9", "D10", "D11"]},
                         index=[8, 9, 10, 11])
```

Concatenation (Verkettung) Verkettet DataFrames. Die Länge der Achse, auf die verkettet wird, muss übereinstimmen. Zum Verketten kann man pd.concat verwenden.

[77]:
```
pd.concat([df1,df2,df3])
```

[77]:

	A	B	C	D
0	A0	B0	C0	D0
1	A1	B1	C1	D1
2	A2	B2	C2	D2
3	A3	B3	C3	D3
4	A4	B4	C4	D4
5	A5	B5	C5	D5
6	A6	B6	C6	D6
7	A7	B7	C7	D7
8	A8	B8	C8	D8
9	A9	B9	C9	D9
10	A10	B10	C10	D10
11	A11	B11	C11	D11

[78]:
```
# mit axis=1 verketten
pd.concat([df1,df2,df3],axis=1)
```

[78]:

	A	B	C	D	A	B	C	D	A	B	C	D
0	A0	B0	C0	D0	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
1	A1	B1	C1	D1	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
2	A2	B2	C2	D2	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
3	A3	B3	C3	D3	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
4	NaN	NaN	NaN	NaN	A4	B4	C4	D4	NaN	NaN	NaN	NaN
5	NaN	NaN	NaN	NaN	A5	B5	C5	D5	NaN	NaN	NaN	NaN
6	NaN	NaN	NaN	NaN	A6	B6	C6	D6	NaN	NaN	NaN	NaN
7	NaN	NaN	NaN	NaN	A7	B7	C7	D7	NaN	NaN	NaN	NaN
8	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	A8	B8	C8	D8
9	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	A9	B9	C9	D9
10	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	A10	B10	C10	D10
11	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	A11	B11	C11	D11

Beispiel-DataFrames

[79]:
```
left = pd.DataFrame({"key": ["K0", "K1", "K2", "K3"],
                     "A": ["A0", "A1", "A2", "A3"],
                     "B": ["B0", "B1", "B2", "B3"]})
```

```
right = pd.DataFrame({"key": ["K0", "K1", "K2", "K3"],
                      "C": ["C0", "C1", "C2", "C3"],
                      "D": ["D0", "D1", "D2", "D3"]})
```

[80]:
```
left
```

[80]:
```
  key   A   B
0  K0  A0  B0
1  K1  A1  B1
2  K2  A2  B2
3  K3  A3  B3
```

[81]:
```
right
```

[81]:
```
  key   C   D
0  K0  C0  D0
1  K1  C1  D1
2  K2  C2  D2
3  K3  C3  D3
```

Merging Die **merge** Funktion erlaubt es DataFrames basierend auf einem Kriterium zu fusionieren.

[82]:
```
pd.merge(left,right,how="inner",on="key")
```

[82]:
```
  key   A   B   C   D
0  K0  A0  B0  C0  D0
1  K1  A1  B1  C1  D1
2  K2  A2  B2  C2  D2
3  K3  A3  B3  C3  D3
```

[83]:
```
# komplizierteres Beispiel
left = pd.DataFrame({"key1": ["K0", "K0", "K1", "K2"],
                     "key2": ["K0", "K1", "K0", "K1"],
                        "A": ["A0", "A1", "A2", "A3"],
                        "B": ["B0", "B1", "B2", "B3"]})

right = pd.DataFrame({"key1": ["K0", "K1", "K1", "K2"],
                              "key2": ["K0", "K0", "K0", "K0"],
                                 "C": ["C0", "C1", "C2", "C3"],
                                 "D": ["D0", "D1", "D2", "D3"]})
```

[84]:
```
left
```

[84]:
```
  key1 key2   A   B
0   K0   K0  A0  B0
```

```
1   K0   K1  A1  B1
2   K1   K0  A2  B2
3   K2   K1  A3  B3
```

[85]:
```
right
```

[85]:
```
  key1 key2   C   D
0   K0   K0  C0  D0
1   K1   K0  C1  D1
2   K1   K0  C2  D2
3   K2   K0  C3  D3
```

[86]:
```
pd.merge(left, right, on=["key1", "key2"])
```

[86]:
```
  key1 key2   A   B   C   D
0   K0   K0  A0  B0  C0  D0
1   K1   K0  A2  B2  C1  D1
2   K1   K0  A2  B2  C2  D2
```

[87]:
```
pd.merge(left, right, how='outer', on=["key1", "key2"])
```

[87]:
```
  key1 key2    A    B    C    D
0   K0   K0   A0   B0   C0   D0
1   K0   K1   A1   B1  NaN  NaN
2   K1   K0   A2   B2   C1   D1
3   K1   K0   A2   B2   C2   D2
4   K2   K1   A3   B3  NaN  NaN
5   K2   K0  NaN  NaN   C3   D3
```

[88]:
```
pd.merge(left, right, how="right", on=["key1", "key2"])
```

[88]:
```
  key1 key2    A    B   C   D
0   K0   K0   A0   B0  C0  D0
1   K1   K0   A2   B2  C1  D1
2   K1   K0   A2   B2  C2  D2
3   K2   K0  NaN  NaN  C3  D3
```

[89]:
```
pd.merge(left, right, how='left', on=['key1', 'key2'])
```

[89]:
```
  key1 key2   A   B    C    D
0   K0   K0  A0  B0   C0   D0
1   K0   K1  A1  B1  NaN  NaN
2   K1   K0  A2  B2   C1   D1
3   K1   K0  A2  B2   C2   D2
4   K2   K1  A3  B3  NaN  NaN
```

Joining

[90]:
```
left = pd.DataFrame({"A": ["A0", "A1", "A2"],
                     "B": ["B0", "B1", "B2"]},
                      index=["K0", "K1", "K2"])

right = pd.DataFrame({"C": ["C0", "C2", "C3"],
                      "D": ["D0", "D2", "D3"]},
                      index=["K0", "K2", "K3"])
```

[91]:
```
left.join(right)
```

[91]:

	A	B	C	D
K0	A0	B0	C0	D0
K1	A1	B1	NaN	NaN
K2	A2	B2	C2	D2

[92]:
```
left.join(right, how="outer")
```

[92]:

	A	B	C	D
K0	A0	B0	C0	D0
K1	A1	B1	NaN	NaN
K2	A2	B2	C2	D2
K3	NaN	NaN	C3	D3

0.5 Operationen

Es gibt vesrschiedene wichtige Operationen, die nützlich für uns sein können. Ein paar möchten wir hier kennenlernen. Aber wie immer gilt im programmieren. Dokumentationen der Bibliotheken bzw das Internet zu durchforschen, falls man ein Problem lösen möchte

[93]:
```
df = pd.DataFrame({"col1":[1,2,3,4],"col2":[11,44,66,11],"col3":
 ↪["abc","def","ghi","xyz"]})
df.head()
```

[93]:

	col1	col2	col3
0	1	11	abc
1	2	44	def
2	3	66	ghi
3	4	11	xyz

Eindeutige Werte rausfinden

[94]:
```
df["col2"].unique()
```

[94]:
```
array([11, 44, 66])
```

[95]:
```
#Hier Frage!
#Anzahl eindeutiger Werte rausfinden
#Möglichkeit1
len(df["col2"].unique())
```

[95]:
```
3
```

[96]:
```
# Aber auch hier gibt es bereits eine fertige Methode
df["col2"].nunique()
```

[96]:
```
3
```

[97]:
```
#Anzeigen wie oft die einzelnen Werte vorkommen
df["col2"].value_counts()
```

[97]:
```
11    2
66    1
44    1
Name: col2, dtype: int64
```

[98]:
```
#Selektieren von einem DataFrame basierend auf Kritieren für mehrere Spalten
newdf = df[(df["col1"]>1) & (df["col2"]==44)]
```

[99]:
```
newdf
```

[99]:

	col1	col2	col3
1	2	44	def

Funktionen auf DataFrame anwenden

[100]:
```
def times2(x):
    return x*2
```

[101]:
```
df["col1"].apply(times2)
```

[101]:
```
0    2
1    4
2    6
3    8
Name: col1, dtype: int64
```

[102]:
```
df["col1"].apply(lambda x: x*2)
```

[102]:
```
0    2
1    4
2    6
3    8
```

```
Name: col1, dtype: int64
```

[103]:
```
df["col1"].sum()
```

[103]:
```
10
```

Eine Spalte für immer Löschen

[104]:
```
del df["col1"]
```

[105]:
```
df
```

[105]:
```
   col2 col3
0    11  abc
1    44  def
2    66  ghi
3    11  xyz
```

Spalten- und Indexnamen zurückgeben

[106]:
```
df.columns
```

[106]:
```
Index(['col2', 'col3'], dtype='object')
```

[107]:
```
df.index
```

[107]:
```
RangeIndex(start=0, stop=4, step=1)
```

DataFrame sortieren

[108]:
```
df.sort_values(by="col2") #inplace=False bei default
```

[108]:
```
   col2 col3
0    11  abc
3    11  xyz
1    44  def
2    66  ghi
```

Nullwerte finden

[109]:
```
df.isnull()
```

[109]:
```
    col2   col3
0  False  False
1  False  False
2  False  False
3  False  False
```

[110]:
```
# Zeilen mit Nullwerten löschen
df.dropna()
```

[110]:

	col2	col3
0	11	abc
1	44	def
2	66	ghi
3	11	xyz

[ ]:
```

```

0.6 Dateneingabe und -ausgabe

Das Laden von Datensätzen erfolgt über eine der folgenden Funktionen:

- pd.read_csv
- pd.read_excel
- pd.read_html

Für einen Überblick siehe https://pandas.pydata.org/pandas-docs/stable/io.html

Bsp:

```
pd.read_csv('myfile.csv')
```

0.6.1 CSV

CSV Eingabe

[111]:
```
tips_dataset = pd.read_csv("https://raw.githubusercontent.com/pandas-dev/pandas/
↪master/doc/data/tips.csv")
```

[112]:
```
tips_dataset.head()
```

[112]:

	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

CSV Ausgabe

[113]:
```
tips_dataset.to_csv("tips_dataset.csv", index=False)
```

0.6.2 Excel

Pandas kann Excel-Dateien lesen und schreiben. **Aber Pandas kann nur Daten importiert.** Keine Formeln, Bilder, Makros! Kann zum Absturz führen!

Excel Eingabe Es muss das Paket ***xlrd*** installiert sein.

[114]:
```
pd.read_excel("07_pandas_excelbsp.xlsx", sheet_name="BL_7-Tage-Inzidenz")
```

[114]:

	Bundesland	2020-05-06 00:00:00	2020-05-07 00:00:00	\
0	Baden-Württemberg	8.202695	7.037334	
1	Bayern	10.078979	9.528383	
2	Berlin	7.363637	7.070158	
3	Brandenburg	4.976279	4.617987	
4	Bremen	17.423490	18.448402	
5	Hamburg	5.159737	4.616607	
6	Hessen	7.133955	6.703045	
7	Mecklenburg-Vorpommern	1.428860	1.428860	
8	Niedersachsen	4.522422	4.384620	
9	Nordrhein-Westfalen	8.169456	8.353478	
10	Rheinland-Pfalz	5.091994	4.308610	
11	Saarland	9.490070	8.783363	
12	Sachsen	3.653808	3.972597	
13	Sachsen-Anhalt	2.445297	2.264164	
14	Schleswig-Holstein	4.833066	4.902110	
15	Thüringen	9.238759	8.398872	

	2020-05-08 00:00:00	2020-05-09 00:00:00	2020-05-10 00:00:00	\
0	6.007480	6.377866	6.974097	
1	9.038963	9.352497	8.778959	
2	7.256917	7.470356	7.176878	
3	4.657797	5.215140	4.697607	
4	20.498224	22.694462	24.890701	
5	4.670920	4.833859	4.345042	
6	6.766884	6.623247	7.277592	
7	1.366736	1.366736	1.490984	
8	4.397147	5.386819	5.223961	
9	7.689884	8.074657	8.080233	
10	3.843476	3.892438	3.647630	
11	6.461324	5.552701	5.249826	
12	3.800942	4.168775	3.997119	
13	2.354730	2.716996	2.626430	
14	5.143763	6.006810	6.973424	
15	8.352211	8.678834	8.772155	

	2020-05-11 00:00:00	2020-05-12 00:00:00	2020-05-13 00:00:00	\
0	6.865692	6.992165	6.721151	

```
1              8.801901             9.161318             9.130729
2              6.616601             5.282609             4.535573
3              4.697607             4.180074             2.866337
4             24.158621            22.108799            23.426542
5              4.616607             4.507981             4.019164
6              7.086076             7.054157             6.495570
7              1.615233             1.428860             1.428860
8              5.374291             4.710334             3.820883
9              8.342325             8.559805             8.448277
10             4.186206             4.528937             3.916918
11             5.047910             4.139286             3.533537
12             4.463041             5.125140             4.021641
13             2.581146             2.535863             1.766048
14             5.903245             5.488982             4.418803
15            10.078646             9.098778             8.585513

    2020-05-14 00:00:00  …  2020-10-16 00:00:00  2020-10-17 00:00:00  \
0              6.910861  …            38.070721                  NaN
1              8.526602  …            35.406424                  NaN
2              4.162056  …            73.852205                  NaN
3              1.990512  …            20.302209                  NaN
4             19.619729  …            73.106068                  NaN
5              3.095842  …            32.643065                  NaN
6              6.176377  …            46.755130                  NaN
7              1.428860  …            13.991337                  NaN
8              3.883520  …            27.421910                  NaN
9              7.779106  …            49.812726                  NaN
10             3.721072  …            26.795945                  NaN
11             3.028746  …            44.787296                  NaN
12             4.193297  …            25.368550                  NaN
13             1.675481  …             8.155707                  NaN
14             4.246194  …            12.225474                  NaN
15             7.932268  …            17.624631                  NaN

    2020-10-18 00:00:00  2020-10-19 00:00:00  2020-10-20 00:00:00  \
0                   NaN                  NaN                  NaN
1                   NaN                  NaN                  NaN
2                   NaN                  NaN                  NaN
3                   NaN                  NaN                  NaN
4                   NaN                  NaN                  NaN
5                   NaN                  NaN                  NaN
6                   NaN                  NaN                  NaN
7                   NaN                  NaN                  NaN
8                   NaN                  NaN                  NaN
9                   NaN                  NaN                  NaN
10                  NaN                  NaN                  NaN
11                  NaN                  NaN                  NaN
```

```
12                   NaN                   NaN                   NaN
13                   NaN                   NaN                   NaN
14                   NaN                   NaN                   NaN
15                   NaN                   NaN                   NaN

    2020-10-21 00:00:00  2020-10-22 00:00:00  2020-10-23 00:00:00  \
0                    NaN                   NaN                   NaN
1                    NaN                   NaN                   NaN
2                    NaN                   NaN                   NaN
3                    NaN                   NaN                   NaN
4                    NaN                   NaN                   NaN
5                    NaN                   NaN                   NaN
6                    NaN                   NaN                   NaN
7                    NaN                   NaN                   NaN
8                    NaN                   NaN                   NaN
9                    NaN                   NaN                   NaN
10                   NaN                   NaN                   NaN
11                   NaN                   NaN                   NaN
12                   NaN                   NaN                   NaN
13                   NaN                   NaN                   NaN
14                   NaN                   NaN                   NaN
15                   NaN                   NaN                   NaN

    2020-10-24 00:00:00  2020-10-25 00:00:00
0                    NaN                   NaN
1                    NaN                   NaN
2                    NaN                   NaN
3                    NaN                   NaN
4                    NaN                   NaN
5                    NaN                   NaN
6                    NaN                   NaN
7                    NaN                   NaN
8                    NaN                   NaN
9                    NaN                   NaN
10                   NaN                   NaN
11                   NaN                   NaN
12                   NaN                   NaN
13                   NaN                   NaN
14                   NaN                   NaN
15                   NaN                   NaN

[16 rows x 174 columns]
```

Excel Ausgabe Dazu wird das paket ***openpyxl*** benötigt.

[115]:
```
df.to_excel('beispiel_neu.xlsx',sheet_name='Sheet1')
```

0.6.3 HTML

Je nachdem was man einlesen möchte gibt es hier weiterführende Bibliotheken, die man zuvor installieren muss.

In der Anaconda Command Prompt: - conda install lxml - conda install html5lib - conda install BeautifulSoup4

Jupyter neustarten und dann kann es losgehen!

** Pandas kann allerdings bereits Tabellen aus html extrahieren.** Bsp:

[116]:
```
test = pd.read_html('https://www.taschenhirn.de/politik-und-religion/
↪deutsche-bundeskanzler/')
```

[117]:
```
#Achtung ist eine Liste nach import.
test[0]
```

[117]:

	Deutsche Bundeskanzler	Im Amt	\
0	Konrad Adenauer (CDU)	1949-1963	
1	Ludwig Erhard (CDU)	1963-1966	
2	Kurt Georg Kiesinger (CDU)	1966-1969	
3	Willy Brandt (SPD)	1969-1974	
4	Helmut Schmidt (SPD)	1974-1982	
5	Helmut Kohl (CDU)	1982-1998	
6	Gerhard Schröder (SPD)	1998-2005	
7	Angela Merkel (CDU)	seit 11.2005	

	*/†	\
0	* 5.1.1876 in Köln † 19.4.1967 in Rhöndorf	
1	* 4.2.1897 in Fürth † 5.5.1977 in Bonn	
2	* 6.4.1904 in Ebingen † 9.3.1988 in Tübingen	
3	* 18.12.1913 in Lübeck † 8.10.1992 in Unkel (b…	
4	* 23.12.1918 in Hamburg-Barmbek † 10.11.2015 i…	
5	* 3.4.1930 in Ludwigshafen-Friesenheim † 16.6…	
6	* 7.4.1944 in Mossenberg	
7	*17.7.1954 in Hamburg	

	Kurzbiografie (Auswahl)
0	Konrad Adenauer war der erste deutsche Bundesk…
1	Ludwig Erhard studierte BWL und VWL. Von 1945 …
2	Kurt Georg Kiesinger war der dritte deutsche B…
3	Willy Brandt wurde 1913 als Herbert Ernst Karl…
4	Der studierte Volkswirt Helmut Schmidt war von…
5	Helmut Kohl war ab 1959 Mitglied des Rheinland…
6	Gerhard Schröder war der 6. deutsche Bundeskan…
7	Angela Merkel wurde zwar in Hamburg geboren, v…