

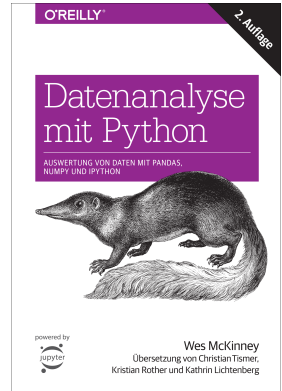
Das Pandas Modul

- **Pandas** ist ein Python-Modul für die Datenanalyse
- Ursprünglich entwickelt von Wes McKinney
- Buch: **Datenanalyse mit Python**
(Wes McKinney)

Weitere Dokumentation:

<https://pandas.pydata.org>

- Als Online-Buch in Hochschul Bib erhältlich!!



Die Vorlesung 3 behandelt Teile von Kapitel 5.1 - 5.3

DATA SCIENCE 1

VORLESUNG 3 - DATENANALYSE MIT PANDAS

PROF. DR. CHRISTIAN BOCKERMANN

HOCHSCHULE BOCHUM

WINTERSEMESTER 2025/2026

- 1 Datenanalyse mit Pandas
 - Das Pandas Modul
 - Series - Ein Datentyp für Meßreihen
 - DataFrame - Ein Datentyp für Tabellen

- 2 Daten Lesen und Explorieren

pandas ist ein zentrales Modul für die Datenverarbeitung in Python



- Datentypen für Tabellen (**DataFrame**) und Zeitreihen (**Series**)
- Funktionen zum Lesen/Transformieren von Daten
- Unterstützung einer Vielzahl von Formaten: CSV, Excel, Datenbanken, usw.

```
import pandas as pd

# Lesen von Excel-Daten
df = pd.read_excel('meine_Daten.xlsx')
```

Einbinden des Moduls in ein Python Program

```
import pandas as pd  
  
# optional dazu noch:  
from pandas import Series, DataFrame
```

- Importiert Pandas als Modul mit Namensraum **pd**
- **pd** ist allgemein verbreiteter Namensraum für Pandas
- Häufig werden **Series** und **DataFrame** noch in den globalen Namensraum importiert (optional)

Pandas bietet Datentypen und Funktionen

- `pandas.read_csv`
- `pandas.read_excel`
- `pandas.to_csv`
- `Series.plot`
- `Series + Series`
- `DataFrame.plot`
- `DataFrame + DataFrame`
- ...

Funktionen

0	4
1	5
2	8
3	2

Series

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

DataFrame

Pandas Datentypen unterstützen Rechenoperationen

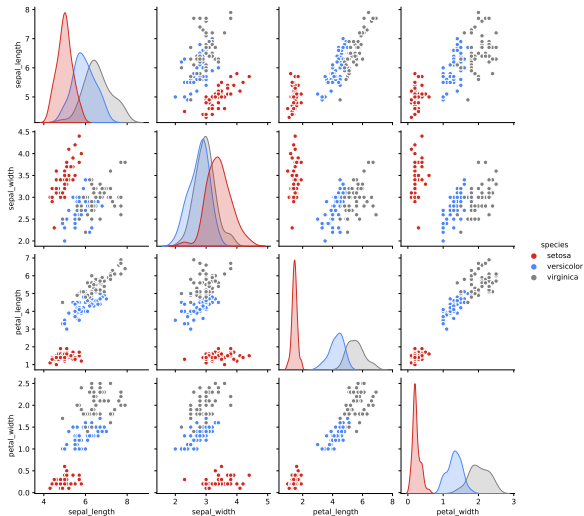
Zum Beispiel:

0	4		0	8
1	5		1	10
2	8	*	2	16
3	2	=	3	4

```
a = Series([4,5,8,2])  
b = a * 2
```

Kompatibilität zu anderen Modulen

- Pandas baut auf *NumPy* Strukturen auf
- SciKit-Learn Algorithmen (ML)
unterstützen NumPy/Pandas Datentypen
- Seaborn kann Pandas Daten plotten



Datenanalyse mit Pandas

SERIES - EIN DATENTYP FÜR MESSREIHEN

Datentyp für Reihe von Meßwerten: **Series**

Index	Werte
0	4
1	5
2	8
3	2

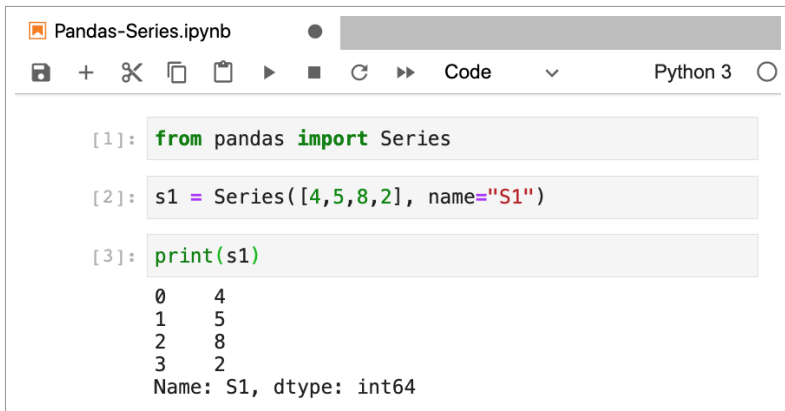
Datentyp für Reihe von Meßwerten: **Series**

Index	Werte
0	4
1	5
2	8
3	2

Ein **Series** Objekt aus einer Liste von Werten erzeugen:

```
data = Series([4,5,8,2])
```

Series Objekt im Jupyter Notebook:



The screenshot shows a Jupyter Notebook window titled "Pandas-Series.ipynb". The toolbar includes icons for saving, adding, deleting, copying, pasting, running, and other standard Jupyter actions. The code cell contains three lines of Python code: importing Series from pandas, creating a Series named s1 with values [4, 5, 8, 2] and name "S1", and printing s1. The output shows the index values 0 to 3 and their corresponding values 4, 5, 8, and 2, followed by the Series name and dtype.

```
[1]: from pandas import Series

[2]: s1 = Series([4,5,8,2], name="S1")

[3]: print(s1)
```

0	4
1	5
2	8
3	2

Name: S1, dtype: int64

(Hier mit optionalem Parameter **name**.)

Series Objekte haben häufig Werte eines Typs

Aber auch Series mit unterschiedlichen Werte-Typen möglich

```
s1 = Series([4,5,8,2])  
print(s1.dtypes)    # ergibt: int64  
  
s2 = Series([1,2,'a','b'])  
print(s2.dtypes)    # ergibt: object
```

Der Zugriff auf Elemente aus einer **Series** erfolgt über den Index:

```
s1 = Series([4,5,8,2])  
x = s1[2]  
print(x)  
  
s1[2] = 7      # Veraendern von Werten
```

Die Method **Series.items()** liefert die Elemente der Series:

```
zs = s1.items()  
# zs ist ein zip Objekt - zip Funktion!  
list(zs)
```

```
[(0, 4), (1, 5), (2, 7), (3, 2)]
```

Iterieren über die Series Werte

Über **zip** Objekte kann mit **for** iteriert werden:

```
s1 = Series([4,5,8,2])

sum = 0
for index,value in s1.items():
    sum = sum + value
```

Die Klasse **Series** hat aber auch das Attribute **values**, mit dem man direkt auf die Werte zugreifen kann:

```
sum = 0
for value in s1.values:
    sum = sum + value
```

Slicing von Series Objekten

```
s1 = Series([4,5,8,2])  
  
s2 = s1[2:4]  
# s2 ist nun ein Teil von s1
```

Slicing von Series Objekten

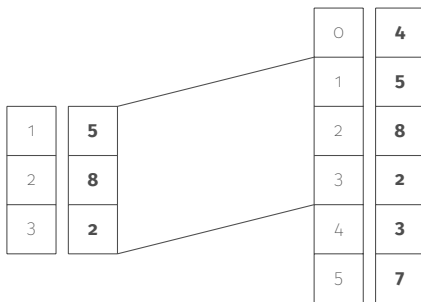
```
s1 = Series([4,5,8,2])  
  
s2 = s1[2:4]  
# s2 ist nun ein Teil von s1
```

Welche Index-Werte hat die neue Teil Serie?



Probieren Sie es im Notebook aus!

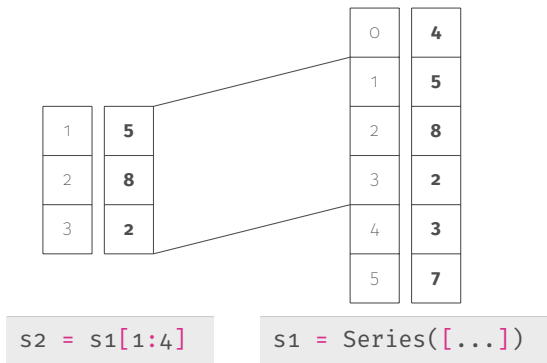
Was passiert beim Slicing?



```
s2 = s1[1:4]
```

```
s1 = Series([...])
```

Was passiert beim Slicing?



Slicing erzeugt **View - beide Series Objekte teilen sich den gleichen Speicherbereich**

Was passiert beim Verändern?

```
s1 = Series([4,5,8,2,3,7])  
  
s2 = s1[1:4]  
s2[2] = -1
```

- Wie wirkt sich die Zuweisung auf s1 aus?
- Was ist der Wert von **s2[0]** ?
- Welche Index-Werte hat **s2** ?



◀ Probieren Sie es im Notebook aus!

Der **Index** eines **Series** Objektes läßt sich ebenfalls festlegen:

```
s1 = Series([4,5,8,2], index=['a','b','c','d'])
```

führt zu folgendem **Series** Objekt:

a	4
b	5
c	8
d	2

Series unterstützt eine Reihe von Operationen

```
a = Series([4,5,8,2])  
b = Series([1,3,0,2])  
  
s = a + b      # ergibt s = [5,8,8,4]  
w = a > 3      # ergibt w = [True, True, True, False]
```

Ergebnisse selbst wieder **Series** Objekte!

Series Objekte unterstützen das Filtern:

```
a = Series([4,5,8,2])
```

```
c = a[a > 4]
```

Series Objekte unterstützen das Filtern:

```
a = Series([4,5,8,2])  
c = a[a > 4]
```

Wie sehen die Index-Werte von c aus?



◀ Probieren Sie es im Notebook aus!

Filtern funktioniert mit **True/False** Sequenzen

Hier bauen wir eine True/False Liste, einen Teil der Series-Elemente auszuwählen:

```
s = Series([4,5,8])  
  
anfang = s[ [True,True,False] ]
```

Mit der **<** Operation passiert quasi das Gleiche:

```
filter = s < 8    # filter ist ~ [True,True,False]  
  
anfang = s[filter]    # Auswaehlen
```



Probieren Sie es im Notebook aus!

Operationen auf **Series** nutzen Index für “Join”

Was passiert im folgenden Fall?

0	4		1	3
1	5		2	2
2	8	+	3	7
3	2		4	4
			5	8
				= ?

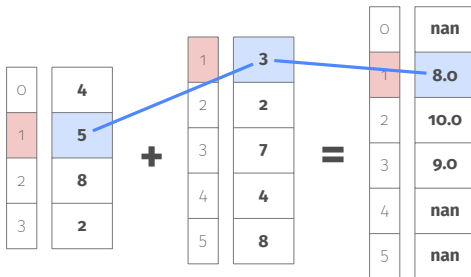
Operationen auf **Series** nutzen Index für “Join”

Was passiert im folgenden Fall?

0	4		1	3		0	nan
1	5		2	2		1	8.0
2	8		3	7		2	10.0
3	2		4	4		3	9.0
			5	8		4	nan
						5	nan

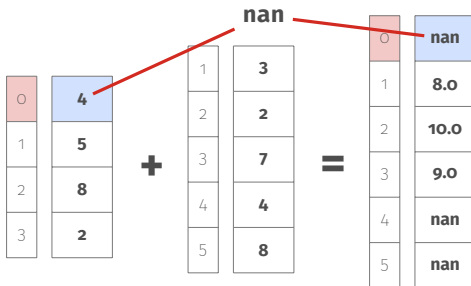
Operationen auf **Series** nutzen Index für “Join”

Was passiert im folgenden Fall?



Operationen auf **Series** nutzen Index für “Join”

Was passiert im folgenden Fall?

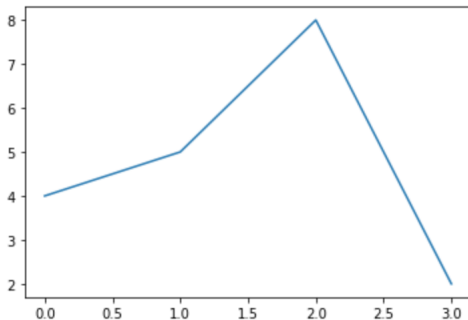


Fehlende Werte werden als
nan (not-a-number) behandelt!

Ein **Series**-Objekt lässt sich leicht in Jupyter-Notebooks plotten:

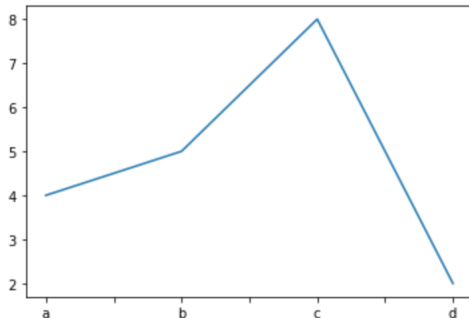
```
s1 = Series([4,5,8,2])  
s1.plot()
```

Out[3]: <AxesSubplot:>



Auch mit einem eigenen Index, lassen sich **Series** Objekte plotten:

Out[5]: <AxesSubplot:>



Datenanalyse mit Pandas

DATAFRAME - EIN DATENTYP FÜR TABELLEN

Datentyp für Tabellen: **DataFrame**

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

- Besitzt Zeilen- und Spalten-Index
- Jede Spalte wie ein **Series** Objekt
- Wie ein **dict** Objekt mit **Series** Werten

Definition eines DataFrame

Ein **DataFrame** wird z.B. als Liste von Zeilen definiert:

```
z0 = [4,1,2,9]    # zeile 0
z1 = [5,1,3,6]    # zeile 1
z2 = [3,8,7,4]    # zeile 2

df = DataFrame([z0, z1, z2],
               columns=['a1', 'a2', 'a3', 'a4'])
```

Ohne **columns** werden die Spalten durchnummeriert (0, 1,..)

Die Form eines DataFrame

df =		a1	a2	a3	a4
	0	4	1	2	9
	1	5	1	3	6
	2	3	8	7	4

`df.shape` liefert die Form von `df`: (zeilen,spalten)

'shape' des obigen DataFrames df

zeilen, spalten = `df.shape` *# zeilen=3, spalten=4*

Einzelne Spalten sind **Series** Objekte

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

Einzelne Spalten sind **Series** Objekte

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

```
a2 = df['a2'] # Spalte 'a2' selektieren  
type(a2)      # -> pandas.core.series.Series  
  
print(a2[2])  # gibt 8 aus
```

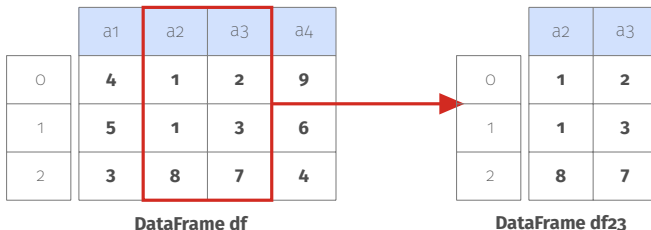
Wenn die Spalten **Series** Objekte sind...

... dann kann man mit mehreren Series-Objekten auch einen DataFrame bauen:

```
s1 = Series([1,1,1])  
s2 = Series([2,2,2])  
s3 = Series([3,3,3])  
  
d = DataFrame()  
d['s1'] = s1  
d['s2'] = s2  
d['s3'] = s3
```

	s1	s2	s3
0	1	2	3
1	1	2	3
2	1	2	3

Mehrere Spalten ergeben wieder einen DataFrame



```
# z0,z1,z2 wie in Folie 28
df = DataFrame([z0, z1, z2],
                columns=['a1', 'a2', 'a3', 'a4'])
df23 = df[['a2', 'a3']]
```

Einzelne Zeilen ergeben ebenfalls **Series** Objekte

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

DataFrame

Einzelne Zeilen ergeben ebenfalls **Series** Objekte

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

DataFrame

```
z2 = df.iloc[1]    # Zeile 1 selektieren
type(z2)           # -> pandas.core.series.Series
print(z2['a1'])    # gibt 5 aus
```

Einzelne Zeilen ergeben ebenfalls **Series** Objekte

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

DataFrame



a1	5
a2	1
a3	3
a4	6

Series

```
z2 = df.iloc[1]    # Zeile 1 selektieren
type(z2)           # -> pandas.core.series.Series
print(z2['a1'])    # gibt 5 aus
```

Zeilen-Series haben Spaltennamen als Index!!

Der Index kann um Label erweitert werden

Betrachten wir den DataFrame von zuvor und fügen Label hinzu:

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

		a1	a2	a3	a4
0	A	4	1	2	9
1	B	5	1	3	6
2	C	3	8	7	4

Der Index kann um Label erweitert werden

Betrachten wir den DataFrame von zuvor und fügen Label hinzu:

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

		a1	a2	a3	a4
0	A	4	1	2	9
1	B	5	1	3	6
2	C	3	8	7	4

← df.loc['B']

```
zeileB = df.loc['B'] # Zugriff per Label
```

Zugriff auf Zeilen/Spalten über LocationIndexer

```
df.iloc[...]
```



Zugriff über **Positionsindex**

- `df.iloc[0]`
- `df.iloc[0:2]`
- `df.iloc[[True, True, False]]`
- `df.iloc[0:2, 0:2]`

```
df.loc[...]
```



Zugriff über **Label**

- `df.loc['A']`
- `df.loc['A':'B']`
- `df.loc[[True, True, False]]`
- `df.loc['A':'B', ['a1', 'a2']]`

Zugriff auf Zeilen/Spalten über LocationIndexer



Keine Sorge, das lernen
Sie über das Semester...!

Zugriff über **Positionsindex**

- `df.iloc[0]`
- `df.iloc[0:2]`
- `df.iloc[[True, True, False]]`
- `df.iloc[0:2, 0:2]`

Zugriff über **Label**

- `df.loc['A']`
- `df.loc['A':'B']`
- `df.loc[[True, True, False]]`
- `df.loc['A':'B', ['a1', 'a2']]`

Der Zugriff über `df[..]`

Zeilenauswahl auch über bool'sche Listen/Sequenzen möglich:

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

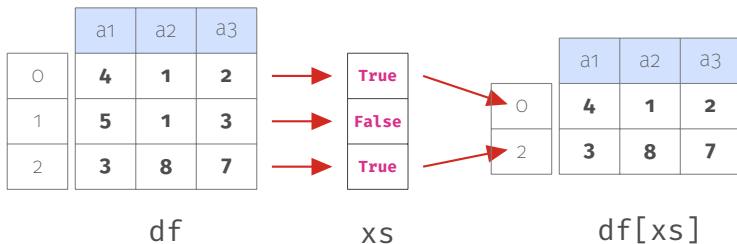
df

True
False
True

xs

Der Zugriff über df[**..**]

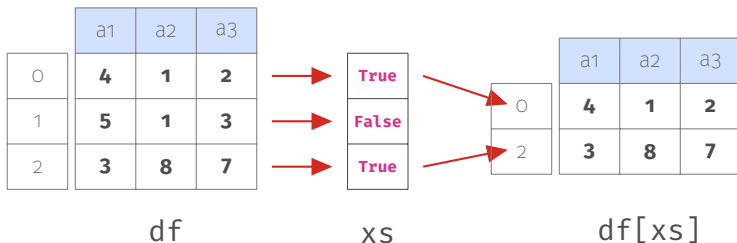
Zeilenauswahl auch über bool'sche Listen/Sequenzen möglich:



Woher kriegen wir bool'sche Sequenzen?

Der Zugriff über df[**..**]

Zeilenauswahl auch über bool'sche Listen/Sequenzen möglich:



Woher kriegen wir bool'sche Sequenzen?

Über Series Objekte!

Erinnern wir uns an **Series** Operationen (Folie 21)

Series Objekte unterstützen viele Operationen

```
# 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
```

Alternativ (kürzer):

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

Zeit für eine **Pause** + Kaffee/Tee!?

- Wenn wir das in den nächsten Übungen nutzen, werden Sie sich langsam daran gewöhnen
- Auf dem Notebook-Server im Verzeichnis **Kurse/DataScience1** liegt die Datei

Pandas-DataFrame.ipynb



◀ Probieren Sie es im Notebook aus!



Daten Lesen und Explorieren

Pandas enthält Funktionen zum Lesen von DataFrames

- `pd.read_csv` - Lesen aus CSV-Datei
- `pd.read_excel` - Lesen aus Excel-Datei

DataFrame aus CSV-Datei lesen:

```
# Lesen aus der Datei 'meine-daten.csv'  
df = pd.read_csv("meine-daten.csv")  
  
# Funktioniert auch mit URLs  
u = "https://datascience.hs-bochum.de/data/iris.csv"  
df = pd.read_csv(u)
```

CSV ist weit verbreitetes Datenformat (*comma separated values*)

Aufbau einer CSV Datei:

```
NR,a1,a2,a3,a4,x1,"Art der Pflanze"  
1,4,1,2,9,2.0,"setosa"  
2,5,1,3,6,0.2,"versicolor"  
3,3,8,7,4,13.0,"virginica"
```

- Text-Datei, eine Zeile pro Datenzeile
- Spalten durch Komma getrennt (auch: Semicolon, Tabulator)
- Manchmal Spaltennamen in erster Zeile (Header)

Pandas unterstützt verschiedene Optionen

```
pd.read_csv( "datei.csv", sep=",",  
             header="infer", names=None)
```

sep	Trennzeichen der Spalten (z.B. ';')
header	Wie werden Spaltennamen bestimmt?
names	Liste mit Spaltennamen

Lesen einer CSV-Datei mit Semikolon, ohne Header:

```
columns = ["a1", "a2", "a3"]  
df = pd.read_csv("datei.csv", sep=";", names=columns)
```

Probieren Sie es aus!

- Im Verzeichnis `Kurse/DataScience1/data` liegt der Iris Datensatz als CSV Datei:
`toy-data.csv`
- Erstellen Sie ein neues Notebook und lesen die Datei ein!



◀ Probieren Sie es im Notebook aus!



Nachdem Laden – Was ist drin, im DataFrame?

- Wie sieht der DataFrame aus? → `df.head(5)`
- Welche Spalten/Datentypen? → `df.columns` / `df.dtypes`
- Wertebereiche der Spalten? → `df.describe()`

```
import pandas as pd
df = pd.read_csv( "Kurse/DataScience1/data/iris.csv")

# Anfang anzeigen (ersten 5 Zeilen)
df.head(5)

# Spalten-Statistiken
df.describe()
```

Spalten-Statistiken mit `describe()`

`describe()` berechnet Statistiken für numerische Spalten

```
# Statistiken berechnen:  
df.describe()
```

	a1	a2	a3
count	3.00	3.00	3.00
mean	4.00	3.33	4.00
std	1.00	4.04	2.65
min	3.00	1.00	2.00
25%	3.50	1.00	2.50
50%	4.00	1.00	3.00
75%	4.50	4.50	5.00
max	5.00	8.00	7.00

Spalten-Statistiken mit `describe()`

`describe()` berechnet Statistiken für numerische Spalten

```
# Statistiken berechnen:  
df.describe()
```

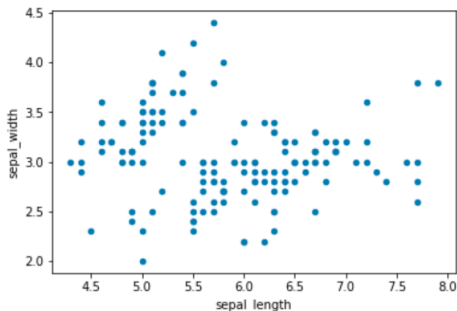
```
# Und das Ergebnis  
# von df.describe()?  
stats = df.describe()
```

Ist natürlich wieder ein DataFrame

	a1	a2	a3
count	3.00	3.00	3.00
mean	4.00	3.33	4.00
std	1.00	4.04	2.65
min	3.00	1.00	2.00
25%	3.50	1.00	2.50
50%	4.00	1.00	3.00
75%	4.50	4.50	5.00
max	5.00	8.00	7.00

Einfache Plots mit DataFrame

```
iris = pd.read_csv('data/iris.csv')  
iris.plot.scatter(x='sepal_length', y='sepal_width')
```



Einfache Plots mit DataFrame in Farbe

- Farbe wird durch Spalte/Series bestimmt
- Series mit Farbe für jeweilige Zeile berechnen
- Farb-Series als Parameter `c` für plot angeben

```
# Farben fuer Werte definieren
farben = { 'setosa': 'red',
           'versicolor': 'green',
           'virginica': 'blue'}

# Spalte colors aus der Spalte species berechnen
colors = iris['species'].map(farben)
```

Einfache Plots mit DataFrame in Farbe

```
# Plot mit c=colors Series (zuvor berechnet!)  
iris.plot.scatter(x='sepal_length', y='sepal_width',  
                  c=colors)
```

