Python Kurs 2019/2020

9: Berechnungen, Daten, Statistik

Bernhard Mallinger

`b.mallinger [at] gmx.at`

https://totycro.github.io/python-kurs

Mit größeren/externen Daten programmieren

NumPy

- Grundlage für Scientific Computing / Data Science / Big Data in Python
  - Sehr beliebt auch für kleinere Auswertungen (overkill)
- Bietet effiziente Datenstrukturen für (große) numerische Berechnungen
- Verhältnismäßig einfache API (Application Programming Interface)
- Wichtige Erweiterungen (gerne alle gleich installieren):
  - Pandas: Praktische Datenstrukturen und Operationen auf Basis von NumPy
  - Matplotlib: Erzeugt Grafiken
  - SciPy: Weitere nützliche Algorithmen

Python und Excel

NumPy

- NumPy ist nur für numerische Daten gedacht (Messungen, Vektoren, Matrizen, Koordinaten, ...)
- Excel-Dateien können Input und Output sein, aber keine "schönen" Excel-Berichte
- Grafiken exportieren als eigene Bilddateien

openpyxl

- Für "menschenlesbare" Excel-Reports
- Erlaubt auch das Arbeiten mit Excel-Dateien und bleibt dabei in Excel-Welt
- Formeln, Diagramme
- Formattierungen, Zellen mergen, etc.
- Bietet keine numerischen Berechnungsfunktionen

Python und Excel

`openpyxl` und `numpy` auch kombinierbar:

- Daten in `numpy` importieren
- Statistiken berechnen
- Komplexe Grafiken erzeugen
- Mittels `openpyxl` Excel-Report generieren
  - Schön formatiert für ChefIn und Marketing
  - Mit Daten für Excel-basierte KollegInnen

csv

- Für einfachen Datenaustausch gern auch `csv` verwenden
- Universelles Format für einfache Tabellendaten
- Import/Export von Datei direkt als Listen/Dictionaries

Python und NumPy

- NumPy: eigene Zahlendatentypen (kein `int` oder `float`)
  - Schnellere Berechnungen
  - Begrenzte Wertebereiche (Übersicht)
- NumPy: eigener Listentyp `ndarray`
  - Kein Hinzufügen oder Löschen von Elementen
  - Dafür effizienteres Speicherlayout, viel schnellere Berechnungen
  - Erlaubt mehrere Dimensionen (Matrixoperationen)
  - Statistische Berechnungsfunktionen eingebaut
  - Viele Libraries basieren darauf bzw. unterstützen es
  - Anwenden von Operationen elementenweise
- NumPy: Keine Dictionaries

NumPy array

```
>>>        numpy    np
>>> values = [2.3, 4.2, 1.2, 5.6]
>>> np_values = np.array(values)
>>> np_values
array([2.3, 4.2, 1.2, 5.6])
>>> type(np_values)
<class 'numpy.ndarray'>
>>> np_values * 2
array([ 4.6,  8.4,  2.4, 11.2])
>>> np_values ** 2
array([ 5.29, 17.64,  1.44, 31.36])
```

Vgl. in normalem Python:

```
[x ** 2    x    values]
```

⇒ Bei numerischen Berechnnungen ist NumPy eleganter, aber auch eingeschränkt.

NumPy array

```
>>> np_values.dtype
dtype('float64')
>>> np_values.shape
(4,)
>>> np_values.ndim
1
>>> np_values.tolist()
[2.3, 4.2, 1.2, 5.6]
```

Leider schlechter Stil:

- `dtype`: unnötige Abkürzung (`datatype`)
- `ndim`: unnötige Abkürzung (`num_dimensions` oder `dimensions`)
- Stolpersteine: `np.ndarray` bezeichnet den Typ, soll aber nicht direkt verwendet werden!

NumPy array: Einheitliche Datentypen

```
>>> [1, 4.3, 'abc']
[1, 4.3, 'abc']
>>> np.array([1, 4.3])
array([1. , 4.3])  # Konvertierung zu float
>>> np.array([1, 4.3, 'abc'])
array(['1', '4.3', 'abc'], dtype='<U32')  # Konvertierung zu String
```

NumPy array: Gleich wie Python list

```
>>> np_values
array([2.3, 4.2, 1.2, 5.6])
>>> len(np_values)
4
>>> np_values[2]
1.2
>>> np_values[1:3]
array([4.2, 1.2])
>>> np_values[1] = 4.8
>>> np_values
array([2.3, 4.8, 1.2, 5.6])
>>>     index, val    enumerate(np_values):
    print(index, ":", val)
0 : 2.3
1 : 4.8
2 : 1.2
3 : 5.6
>>> np_values.sort()
>>> np_values
array([1.2, 2.3, 4.8, 5.6])
```

NumPy array: Anders als Python list

Kein Hinzufügen/Entfernen zu bestehenden Arrays:

```
>>> np_values.append(2.4)
Traceback (most recent call last):
  File "<pyshell>", line 1,    <module>
AttributeError: 'numpy.ndarray' object has no attribute 'append'
>>>     np_values[2]
Traceback (most recent call last):
  File "<pyshell>", line 1,    <module>
ValueError: cannot delete array elements
```

```
# Aber neue Arrays Erzeugen mit np.append / np.delete:
>>> np_values
array([2.3, 4.2, 1.2, 5.6])
>>> np.append(np_values, 1)
array([2.3, 4.2, 1.2, 5.6, 1. ])
>>> np.append(np_values, [2, 3])  # Achtung, anderer Typ!
array([2.3, 4.2, 1.2, 5.6, 2. , 3. ])
>>> np_values
array([2.3, 4.2, 1.2, 5.6])
```

NumPy array: Anders als Python list

Eigene, effizientere Operationen:

```
>>> sum(np_values)
13.3
>>> np.sum(np_values)
13.3
```

Operationen wirken sich immer auf Elemente aus (*broadcasting*):

```
>>> np.array([1, 2, 3]) + np.array([3, 4, 5])
array([4, 6, 8])
>>> np.array([1, 2, 3]) + 1  # Achtung, anderer Typ!
array([2, 3, 4])
```

NumPy array: Mehr als Python list

```
>>> np_values.mean()
3.325
>>> np_values.var()
2.8768749999999996
>>> np_values.std()
1.6961353129983467
>>> np.percentile(np_values, 20)
1.8599999999999999
>>> np.percentile(np_values, 10)
1.53
>>> np.array([1,2,3]).dot(np.array([2, 3, 4]))
20
```

NumPy: `arange`, `linspace`

Einfache Hilfskontrukte:

- `arange`: Wie `range`, gibt aber `numpy` Array zurück
- `linspace`: Auch ähnlich, man kann sich aber aussuchen, wieviele Werte man in dem Interval haben möchte

```
>>> np.arange(5, 10)
array([5, 6, 7, 8, 9])
>>> np.linspace(5, 10, num=20)
array([ 5.        ,  5.26315789,  5.52631579,  5.78947368,  6.05263158,
        6.31578947,  6.57894737,  6.84210526,  7.10526316,  7.36842105,
        7.63157895,  7.89473684,  8.15789474,  8.42105263,  8.68421053,
        8.94736842,  9.21052632,  9.47368421,  9.73684211, 10.        ])
```

Beispiel: Correlation testing

Berechnung von r^2 mittels `linregress` aus `scipy.stats`

```
       numpy    np
       matplotlib.pyplot    plt
       scipy.stats

# Hier einfache Liste, könnte auch np-Array sein
ausbeute =   [99.9, 70.4, 79.8, 75.2, 95.5, 79.1, 73.6, 98.3]
temperatur = [28.2, 28.3, 28.3, 27.1, 31.1, 27.1, 27.1, 30.1]
plt.scatter(ausbeute, temperatur)

x = np.linspace(68, 103)
slope, intercept, r_value, p_value, std_err = \
    scipy.stats.linregress(ausbeute, temperatur)
# Achtung: Berechnung mit vielen Werten auf einmal:
regression = intercept + slope * x
plt.plot(x, regression)
print("r-value", r_value)
print("r-value squared", r_value ** 2)

plt.show()
```

Matrizen in Python

```
>>> py_matrix = [[1, 2, 3], [4, 5, 6]]
>>> numpy_matrix = np.array(py_matrix)
>>> numpy_matrix
array([[1, 2, 3],
       [4, 5, 6]])
>>> numpy_matrix.shape
(2, 3)
>>> numpy_matrix.ndim
2
>>> len(numpy_matrix)  # nur erste dimension
2
```

```
# Fancy indexing nur mit `numpy` array:
>>> numpy_matrix[1, 2]
6
>>> numpy_matrix[1, (0, 1)] # 2 elemente der 2. dimension ausgewählt
array([4, 5])
>>> numpy_matrix[0:2, 1:3] # slices auch möglich
array([[2, 3],
       [5, 6]])
```

Matrizen in Python

Viele weitere Möglichkeiten in `numpy`:

```
>>> numpy_matrix
array([[1, 2, 3],
       [4, 5, 6]])
>>> numpy_matrix.sum(axis=0)
array([5, 7, 9])
>>> numpy_matrix.sum(axis=1)
array([ 6, 15])
>>> numpy_matrix.T
array([[1, 4],
       [2, 5],
       [3, 6]])
>>> numpy_matrix * np.array([[2], [3]])
array([[ 2,  4,  6],
       [12, 15, 18]])
```

Beispiel: Gleichungssystem (exakte Lösung)

Lösbares Gleichungssystem:

```
1 * x +  3 * y = -7
5 * x + -4 * y = 22
```

Beispiel: Gleichungssystem (exakte Lösung)

Lösbares Gleichungssystem:

```
1 * x +  3 * y = -7
5 * x + -4 * y = 22
```

Gleichungsystem übersetzen und lösen mit [solve]:

```
     numpy.linalg        solve
coefficients = [[1, 3], [5, -4]]
result = [-7, 22]
print(solve(coefficients, result))
```

Ausgabe: `[ 2. -3.]`

⇒ `x = 2`, `y = -3`

Beispiel: Gleichungssystem (näherungsweise)

Überbestimmtes Gleichungssystem:

```
1 * x +  3 * y = -7
5 * x + -4 * y = 22
2 * x +  2 * y =  1
```

Berechnen mit lstsq:

```
     numpy.linalg        lstsq

coefficients = [[1, 3], [5, -4], [2, 2]]
result = [-7, 22, 1]
x, residuals, rank, s = lstsq(coefficients, result, rcond=    )
```

Ergebnis:
`x, y`: `[ 2.35948645 -2.63195435]`
`residuals`: `[4.63480742]`

Beispiel: Curve fitting

Polynome verschiedenen Grades

```
y = np.array([1, 4, 8, 18, 25])
x = np.arange(1, len(y) + 1)

plt.plot(x, y, 'o')

space = np.linspace(x[0] - 0.5, x[-1] + 0.5, 100)

    fitting_degree, style    [(1, '.'), (2, "--"), (4, '-')]:
    coefficients = np.polyfit(x=x, y=y, deg=fitting_degree)
    polynom = np.poly1d(coefficients)
    plt.plot(space, polynom(space), style)

    print("\nPolynom Grad " + str(fitting_degree) + ":")
    print(polynom)

plt.show()
```

Beispiel: Curve fitting

Je nach Parameter kann es mehrere Rückgabewerte geben:

```
>>> coefficients = np.polyfit(x=x, y=y, deg=1)
>>> coefficients, residuals, rank, singular_values, rcond = \
        np.polyfit(x=x, y=y, deg=1, full=    )
```

NumPy: Weiterführende Links

Einfache Einführung:
https://www.python-kurs.eu/numpy.php

Offizielle Dokumentation:
https://numpy.org/doc/1.18/user/quickstart.html

2 komplette Bücher über Python Data Science mit vielen weiterführenden Themen:
https://jakevdp.github.io/PythonDataScienceHandbook/index.html
https://scipy-lectures.org/

NumPy: Übungen

AUFGABE

Rechne die Beispiele auf den letzen Folien mit geänderten Zahlen nach. Schau in der Dokumentation nach, wie die man die Funktionen aufrufen kann!

AUFGABE

Such in der Dokumentation oder den eBooks auf der letzen Folie weitere Funktionen, die für dich interessant sein können und versuche sie anzuwenden

Übungen aus verschiedenen Bereichen:
https://www.w3resource.com/python-exercises/numpy/index.php

Pandas

- Paket für tabellarische Daten (vgl. Excel)
- Bequemes Handling von Spalten und Indices
- Basiert auf `numpy`
- Ähnlichkeiten zu data frame in R

```
# Konvention:
      pandas    pd
```

Hilfreiche Links:

https://www.python-kurs.eu/pandas.php
https://ourcodingclub.github.io/tutorials/pandas-python-intro/

Series

- Grunddatentyp von Pandas
- Kombiniert ein normales `numpy`-Array mit einem Index

```
>>> thickness = pd.Series([4.3, 1.2, 2.2, 9.1])
>>> thickness
0    4.3
1    1.2
2    2.2
3    9.1
dtype: float64
>>> thickness.values
array([4.3, 1.2, 2.2, 9.1])
>>> type(thickness.values)
<class 'numpy.ndarray'>
>>> thickness.index
RangeIndex(start=0, stop=4, step=1)
>>> thickness.mean()  # wie numpy array
4.2
>>> thickness[2]  # wie numpy array
2.2
```

Series

- Versteht aber sinnvolle Indexwerte
- Berücksichtigt diese bei Berechnungen

```
>>> machines = ["Maschine A", "Maschine K", "Maschine L"]
>>> thickness = pd.Series([4.3, 1.2, 2.2], index=machines)
>>> thickness
Maschine A    4.3
Maschine K    1.2
Maschine L    2.2
>>> thickness.index
Index(['Maschine A', 'Maschine K', 'Maschine L'], dtype='object')
>>> speed = pd.Series([100, 50, 80], index=machines)
>>> speed
Maschine A    100
Maschine K     50
Maschine L     80
>>> speed * thickness  # Multiplikation gleicher Indizes!
Maschine A    430.0
Maschine K     60.0
Maschine L    176.0
```

Series

Versteht auch Zugriffe über Index:

```
>>> thickness["Maschine K"]
1.2
>>> thickness[1]
1.2
>>> thickness["Maschine X"] = 5.0
>>> thickness
Maschine A      4.3
Maschine K      1.2
Maschine L      2.2
Maschine X      5.0
```

Viele Werte Indexzugriffsmöglichkeiten (Filterungen, mehrere Zeilen aussuchen)...

Series

Ähnlichkeiten zu Dictionaries:

```
>>> series = pd.Series({'a': 5, 'b': 2, 'c': 4})
>>> series['a']
5
>>> series['d'] = 9
>>> series
a    5
b    2
c    4
d    9
dtype: int64
```

DataFrame

Mehrere Spalten (`Series`) zusammen ergeben eine Tabelle (`DataFrame`)

```
>>> machines_frame = pd.concat(
    {'speed': speed, 'thickness': thickness},
    axis='columns',
)
>>> machines_frame
            speed  thickness
Maschine A    100        4.3
Maschine K     50        1.2
Maschine L     80        2.2
>>> type(machines_frame)
<class 'pandas.core.frame.DataFrame'>
>>> machines_frame['speed']
Maschine A    100
Maschine K     50
Maschine L     80
Name: speed, dtype: int64
>>> machines_frame['speed']['Maschine K']
50
```

DataFrame: Viele nützliche Funktionen

```
>>> machines_frame['speed'].corr(machines_frame['thickness'])
0.9501651394030409
>>> machines_frame['speed'].corr(machines_frame['thickness']) ** 2
0.9028137921368002
>>> machines_frame.describe()
            speed  thickness
count    3.000000   3.000000
mean    76.666667   2.566667
std     25.166115   1.582193
min     50.000000   1.200000
25%     65.000000   1.700000
50%     80.000000   2.200000
75%     90.000000   3.250000
max    100.000000   4.300000

# Eingebautes matplotlib-Interface:
>>> machines_frame.plot.scatter(x="speed", y="thickness")
```

Pandas: Spalten

```
>>> df = pd.DataFrame(
    data={'speed': [100, 50, 80], 'thickness': [4.3, 1.2, 2.2]},
    index=['A', 'K', 'L'],
)
>>> df['price'] = [10000, 5000, 70000]
>>> df
   speed  thickness  price
A    100        4.3  10000
K     50        1.2   5000
L     80        2.2  70000
>>> df.price  # gleiche Bedeutung wie df['price']
A    10000
K     5000
L    70000
Name: price, dtype: int64
>>> # später löschen mit: del df['price']
>>> df[ ['speed', 'price'] ]
   speed  price
A    100  10000
K     50   5000
L     80  70000
```

Pandas: Datensätze auswählen

`.loc[value]` liefert eine Zeile

`.loc[list]` liefert mehrere Zeilen

Achtung: `[]`, nicht `()`!

```
>>> df[ ['speed', 'price'] ].loc['A']
speed      100
price    10000
Name: A, dtype: int64
>>> df[ ['speed', 'price'] ].loc[ ['A', 'K'] ]
   speed  price
A    100  10000
K     50   5000
>>> type(df[ ['speed', 'price'] ].loc['A'])
<class 'pandas.core.series.Series'>
>>> type(df[ ['speed', 'price'] ].loc[ ['A', 'K'] ])
<class 'pandas.core.frame.DataFrame'>
```

Pandas: Daten auswählen

```
>>> df.loc['A', 'price']
10000
>>> df['price']['A']
10000
>>> df.at["A", "price"]
10000
>>> df.loc['A']['price']
10000.0  # Achtung: Konvertierung zu Series
         #          -> Einheitlicher Datentyp (float)
```

`iloc` ähnlich zu `loc`, verwendet aber Zahl, nicht Name (integer location)

Ausführliche Einführung zu Indexing: https://medium.com/dunder-data/selecting-subsets-of-data-in-pandas-6fcd0170be9c

Pandas: Daten auswählen

Ausdruck	Ergebnis
`df[spalte]`	1 Spalte
`df[ [spalte1, spalte2] ]`	Mehrere Spalten
`df[spalte][index]`	1 Zelle
`df.loc[index]`	1 Zeile
`df.loc[index, spalte]`	1 Zelle
`df.loc[ [index1, index2] ]`	Mehrere Zeilen

Ergebnisse dieser Ausdrücke können auch weiter gefiltert werden

Pandas: Daten updaten

`.loc` gibt normalerweise "Ansichten" (views) zurück, über welche Updates auf ursprüngliche Daten möglich sind.

```
>>> df.loc[4, 'Hit & Miss'] =
>>> df.loc[4, 'Hit & Miss']
```

Werden die Daten weiterverwendet, kopiert pandas intern manchmal, wodurch Updates keine Auswirkung mehr haben:

```
>>> df.loc[4]['Hit & Miss'] =
>>> df.loc[4]['Hit & Miss']
```

Pandas: Daten filtern

```
>>> df
   speed  thickness
A    100        4.3
K     50        1.2
L     80        2.2
>>> df.speed > 60  # Operation elementweise angewandt
A
K
L
Name: speed, dtype: bool
>>> df[ [    ,     ,     ] ]
   speed  thickness
A    100        4.3
K     50        1.2
>>> df[ df.speed > 60 ]
   speed  thickness
A    100        4.3
L     80        2.2
```

Excel-Daten einlesen

```
pd.read_excel("file.xlsx")
```

Wichtige Parameter:

- `sheet_name`: Name vom gewünschten Sheet
- `converters`: Dictionary um Inputwerte zu bearbeiten
- `index_col`: Eine Spalte als Index auswählen

```
               (value):
           value == 'x'

df = pd.read_excel(
    "Produktionsdaten.xlsx",
    index_col=0,
    converters={
        'offene Leimfugen': is_equal_to_x,
        'Hit & Miss': is_equal_to_x,
    },
)
```

Dateien in Code referenzieren

Relativ

```
# \\ unter Windows auch möglich statt /
pd.read_excel("file.xlsx")
pd.read_excel("subdirectory/file.xlsx")
pd.read_excel("../file.xlsx")
```

Absolut

```
# Linux / Apple
pd.read_excel("/home/user/python-kurs/file.xlsx")
# Windows
pd.read_excel("C:\\Users\\file.xlsx")
```

Empfohlen: `Path` aus `pathlib` verwenden

Excel-Daten schreiben

```
df.to_excel("a.xlsx")
```

Braucht das Python-Paket `openpyxl`, ggf. über Thonny installieren.

Pandas: Daten filtern

AUFGABE

Filtere ein Excel-File, das du im Studium, beruflich oder privat verwendest, nach interessanten Gesichtspunkten.

Bitte um Zusendung für praxisnahe Beispieldaten!

Bereiche auswählen

Hinweis: Slicing bei Pandas `loc` ist inklusiv

Zeilen auswählen:

```
df.loc[4:7]  # Einträge mit Nummer 4 bis inkl 7
df[4:7]  # 4. bis (exkl.) 7. Eintrag
```

Spalten auswählen:

```
df.loc[:, 'Scanparameter Röntgen':'Scanparameter Farbkamera']
```

Pandas: Funktionen auf Daten anwenden

`apply` auf Series (einzelne Werte verarbeiten):

```
                  (ausbeute):
      ausbeute < 80:
              'niedrig'
       ausbeute < 90:
              'mittel'
       :
              'hoch'

df['Ausbeute qualitativ'] = \
    df['Ausbeute nach Fehlerkappung [%]'].apply(ausbeute_quality)
```

`apply` auf DataFrame (ganze Datensätze verarbeiten):

```
             (row):
    columns = ["Scanparameter Röntgen", "Scanparameter Laser", "Scanp
          row[ columns ].mean()

df['Scanparameter mean'] = df.apply(mean_scan, axis='columns')
```

Pandas: Komplexere Filterausdrücke

Filterung:

```
df[ df['Ausbeute qualitativ'] == "hoch" ]
```

Query:

```
df.query("`Ausbeute qualitativ` == 'hoch'")
```

- Backticks für Spaltennamen
- Jeweils andere Hochkomma für Strings, oder escaping:

  ```
  df.query("`Ausbeute qualitativ` == \"hoch\"")
  ```

Hohe Ausbeute und hohe Luftfeuchtigkeit

Filterung: "und" via Operator `&`

```
# Klammerung wichtig!
df[
  (df['Ausbeute qualitativ'] == "hoch") &
  (df["rel. Luftfeuchtigkeit [%]"] > 72)
]
```

Query: "und" via Python `and`

```
df.query(
  "`Ausbeute qualitativ` == 'hoch' and `rel. Luftfeuchtigkeit [%]` > 72
)
```

"Oder" geht analog mit `|` bzw `or`, auch

Luftfeuchtigkeit von hoher Ausbeute

Kombination Filterung und `loc`:

```
df.loc[df["Ausbeute qualitativ"] == 'hoch', 'rel. Luftfeuchtigkeit [%]
```

Query und Spaltenauswahl:

```
df.query("`Ausbeute qualitativ` == 'hoch'")['rel. Luftfeuchtigkeit [%]
```

Hinweis: Updates nur über `loc` garantiert (sofern keine weiteren Filterungen):

```
# Schreiboperation automatisch auf alle Zellen angewendet:
df.loc[
    df["Ausbeute qualitativ"] == 'hoch',
    'Scanparameter Röntgen':'Scanparameter Farbkamera'
] = 100
```

Scanparameter relativ zueinander

Filterung:

```
df[ df['Scanparameter Röntgen'] < df['Scanparameter Laser'] ]
```

Query:

```
df.query("`Scanparameter Röntgen` < `Scanparameter Laser`")
```

Beispiel: `groupby`

Durchschnittliche Ausbeute je nach Temperatur"gruppe"

```
>>> sheet1 = pd.read_excel("Produktionsdaten.xlsx")
>>> sheet1[
    ["Temperatur [°C]", "Ausbeute nach Fehlerkappung [%]"]
].groupby("Temperatur [°C]").mean()
                 Ausbeute nach Fehlerkappung [%]
Temperatur [°C]
19.1                                   84.649298
19.2                                   84.147770
19.3                                   86.941728
19.4                                   83.435700
19.5                                   82.776405
19.6                                   86.639405
19.7                                   84.452549
19.8                                   83.610708
19.9                                   83.722360
20.0                                   84.518087
20.1                                   87.451870
20.2                                   83.819576
```

Beispiel: `groupby`

Flexiblere Aggregation mit `agg`: Mehrere Spalten mit verschiednene Operationen

```
>>> sheet1.groupby("Temperatur [°C]").agg({
    "Ausbeute nach Fehlerkappung [%]": "mean",
    "rel. Luftfeuchtigkeit [%]": "max",
})
                     Ausbeute [%]  rel. Luftfeuchtigkeit [%]
Temperatur [°C]
19.1                    84.649298                          74
19.2                    84.147770                          74
19.3                    86.941728                          72
19.4                    83.435700                          68
19.5                    82.776405                          67
19.6                    86.639405                          67
19.7                    84.452549                          64
19.8                    83.610708                          63
19.9                    83.722360                          63
20.0                    84.518087                          63
20.1                    87.451870                          61
20.2                    83.819576                          60
```

Beispiel: `groupby`

Flexiblere Aggregation mit `agg`: Verschiedene Operationen auf gleiche Spalten

```
>>> sheet1[
    ["Temperatur [°C]", "Ausbeute nach Fehlerkappung [%]"]
].groupby("Temperatur [°C]").agg([np.min, np.mean, np.max])
                Ausbeute nach Fehlerkappung [%]
                                            amin       mean       amax
Temperatur [°C]
19.1                                   71.013215  84.649298  98.495539
19.2                                   71.269318  84.147770  98.299047
19.3                                   70.299240  86.941728  99.887834
19.4                                   70.134642  83.435700  99.476956
19.5                                   70.069126  82.776405  99.263042
19.6                                   70.310167  86.639405  99.399352
19.7                                   70.411478  84.452549  99.939151
19.8                                   71.062171  83.610708  99.020136
19.9                                   70.220220  83.722360  99.655665
20.0                                   70.071085  84.518087  99.515087
20.1                                   70.697598  87.451870  99.089169
20.2                                   70.384660  83.819576  99.491524
```

Beispiel: `groupby`

Hinweis: Jede dieser Auswertungen kann sofort geplottet werden, einfach `.plot()` am Ende!

Pandas: Achtung bei NaN

NaN bedeutet "not a number" und steht für eine fehlgeschlagene numerische Berechnung.

```
>>> df['Delaminierung %']
Binder Nr.
1        NaN
2        NaN
        ...
460      NaN
461     81.0
Name: Delaminierung %, Length: 461, dtype: float64
>>> df['Delaminierung %'].sum()  # Pandas sum ignoriert "NaN"
848.0
>>> sum(df['Delaminierung %'])  # Achtung: Normale Funktionen nicht
nan
```

Pandas: Achtung bei NaN

```
>>> df['Delaminierung %'].dropna()
Binder Nr.
11     84.0
37     92.0
       ...
413    79.0
461    81.0
Name: Delaminierung %, dtype: float64
>>> sum(df['Delaminierung %'].dropna())
848.0

# Ganze Tabelle entsprechend gefiltert:
>>> df[df['Delaminierung %'].notna()]  # wie dropna(axis='index')
            Lamellenbreite [mm]  ...  seitl. Lamellenversatz
Binder Nr.                       ...
11                          140  ...                    NaN
37                          140  ...                    NaN
       ...
413                         140  ...                    NaN
461                         140  ...                    NaN

[10 rows x 19 columns]
```

Pandas: Achtung bei NaN

Alternativ Default-Wert für NaN vergeben

```
>>> df['Delaminierung %'].fillna(0)
Binder Nr.
1       0.0
2       0.0
3       0.0
4       0.0
5       0.0
       ...
457     0.0
458     0.0
459     0.0
460     0.0
461    81.0
Name: Delaminierung %, Length: 461, dtype: float64
```

Pandas: Index

Anstatt `index_col` bei `read_excel`, später `set_index` verwenden:

```
>>> df
   year  month  measurement
0  2019      4          6.1
1  2019      6          7.8
2  2020      1          9.2
>>> by_year = df.set_index("year")
# Achtung: ursprüngliches df nicht geändert
>>> by_year
      month  measurement
year
2019      4          6.1
2019      6          7.8
2020      1          9.2
>>> by_year.loc[2019]
      month  measurement
year
2019      4          6.1
2019      6          7.8
```

Pandas: MultiIndex

```
>>> by_date = df.set_index(['year', 'month'])
>>> by_date
            measurement
year month
2019 4              6.1
     6              7.8
2020 1              9.2
>>> by_date.loc[2019]
       measurement
month
4              6.1
6              7.8
>>> by_date.loc[2019, 4]
measurement    6.1
Name: (2019, 4), dtype: float64

# Weniger praktisch und langsamer:
>>> df[ (df.year == 2019) & (df.month == 4) ]
   year  month  measurement
0  2019      4          6.1
```

Beispiel: Kombinierte Felder trennen

```
>>> df
                  Messung  Wert
0  Werkstück A, Messung 1   4.3
1  Werkstück A, Messung 2   3.2
2  Werkstück B, Messung 1   1.0
```

Kombinierte Spalte "Messung" auftrennen für einfachere Abfragen.

Möglichkeiten:

- `for`-loop über `df["Messung"]`
- `df["Messung"].apply()`
- `df["Messung"].str`: Funktionen für Text-Spalten (Dokumentation)

Beispiel: Kombinierte Felder trennen

Lösung: `df["Messung"].str.split` mit `expand=True`

`pd.to_numeric` um Text-Spalte zu Integer konvertieren

```
>>> splitted = df["Messung"].str.split(", Messung ", expand=    )
>>> splitted
             0  1
0  Werkstück A  1
1  Werkstück A  2
2  Werkstück B  1
>>> df["Werkstück"] = splitted[0]  # 0 als Spaltenname
>>> df["Messung"] = pd.to_numeric(splitted[1])
>>> df
   Messung  Wert    Werkstück
0        1   4.3  Werkstück A
1        2   3.2  Werkstück A
2        1   1.0  Werkstück B
```

Beispiel: Kombinierte Felder trennen

Getrennte Felder als MultiIndex

```
>>> df = df.set_index(["Werkstück", "Messung"])
>>> df.loc['Werkstück A']
         Wert
Messung
1         4.3
2         3.2
>>> df.loc['Werkstück A', 1]
Wert    4.3
Name: (Werkstück A, 1), dtype: float64
```

Beispiel: Kombinierte Felder trennen

Lösung: `df["Messung"].str.extract` mit regulärem Ausdruck:

- Reguläre Ausdrücke beschreiben Textmuster
- `[a-z]`, `[A-Z]` oder `[0-9]` beschreiben Wertebereiche
- `([a-z])` beschreibt "capture group"

```
>>> df["Messung"].str.extract(r"Werkstück ([A-Z]), Messung ([0-9])")
   0  1
0  A  1
1  A  2
2  B  1
# Auch mit "named capture groups":
>>> df["Messung"].str.extract(
    r"Werkstück (?P<Werkstück>[A-Z]+), Messung (?P<Messung>[0-9]+)"
)
  Werkstück Messung
0         A       1
1         A       2
2         B       1
```

Pandas: Daten zusammenführen

```
>>> sheets[0].shape
(461, 19)
>>> pd.concat(sheets, axis='index')
            Lamellenbreite [mm]  ...  seitl. Lamellenversatz
Binder Nr.                       ...
1                           140  ...                     NaN
2                           140  ...                     NaN
...                         ...  ...                     ...
461                         160  ...                     NaN

[2305 rows x 19 columns]
>>> pd.concat(sheets, axis='columns')
            Lamellenbreite [mm]  ...  seitl. Lamellenversatz
Binder Nr.                       ...
1                           140  ...                     NaN
2                           140  ...                     NaN
...                         ...  ...                     ...
461                         140  ...                     NaN

[461 rows x 95 columns]
```

Pandas

AUFGABE

Wende jede dieser Operationen auf dein eigenes Excel-File an.