

Bausteine Computergestützter Datenanalyse

Lukas Arnold Simone Arnold Florian Bagemihl
Matthias Baitsch Marc Fehr Franca Hollmann
Maik Poetzsch Sebastian Seipel

2026-02-20

Inhaltsverzeichnis

Werkzeugbaustein Pandas	4
Voraussetzungen	4
Lernziele	5
1 Einleitung	6
1.1 Die Datenstrukturen Series und DataFrame	6
1.1.1 Series	7
1.1.2 Aufgabe Series	9
1.1.3 DataFrame	9
1.1.4 Aufgabe DataFrame	15
1.2 Deskriptive Datenanalyse mit Pandas	16
1.3 Slicing	19
1.3.1 Slice Operator	19
1.3.2 Slicing mit Pandas-Methoden	21
1.3.3 Indexbasiertes Slicing mit .iloc[]	23
1.4 Aufgaben Slicing	24
1.5 Datenstrukturen verbinden	27
1.6 Einfügen und löschen in Datenstrukturen	28
1.7 Aufgaben verbinden und löschen	29
Quellen	31
2 Operationen	32
2.1 Zeilen- und spaltenweise Operationen	33
2.1.1 arithmetische Funktionen	34
2.1.2 summarische Funktionen	35
2.1.3 boolsche Funktionen	36
2.2 Einzelwerte oder Liste	36
2.3 NumPy-Array	37
2.4 Series	38
2.5 DataFrame	39
2.5.1 Verwendung der Methoden .agg() und .apply()	41
2.6 Funktion	42
2.7 Funktionsname	43
2.8 Liste von Funktionen	43
2.9 Dictionary von Funktionen	43

2.10	Aufgaben Operationen	45
2.11	Suchen und ersetzen	47
2.12	Aufgaben suchen und ersetzen	48
2.13	Sortieren	49
2.14	Aufgaben Sortieren	52
2.15	GroupBy	53
2.16	DataFrame meerschweinchen	53
2.17	meerschweinchen gruppiert nach Verabreichungsart	54
2.18	Länge nach Verabreichungsart	54
2.19	Länge nach Verabreichungsart und Dosis	54
2.20	Aufgaben GroupBy	55
3	Grafikerstellung	58
3.1	Series	58
3.2	DataFrame	60
3.3	subplots	62
4	Datentypen	66
5	Zeitreihen	70
5.1	Datums- und Zeitinformationen in Python	70
5.1.1	Naive und bewusste Datetime-Objekte	71
5.1.2	Alles ist relativ: die Epoche	87
5.1.3	Zeitumstellung - Daylight Saving Time	88
5.1.4	Kalender	89
5.2	datetime in Pandas	89
5.3	timedelta in Pandas	92
5.4	Zugriff auf Zeitreihen	93
5.5	Aufgaben	96
6	Dateien lesen und schreiben	98
6.1	Zeitreihen einlesen	103
6.2	Aufgaben Zeitreihen einlesen	104
6.3	Schwierige Dateien einlesen	107

Werkzeugbaustein Pandas



Bausteine Computergestützter Datenanalyse von Lukas Arnold, Simone Arnold, Florian Bagemihl, Matthias Baitsch, Marc Fehr, Franca Hollmann, Maik Poetzsch und Sebastian Seipel. Werkzeugbaustein Pandas von Marc Fehr und Maik Poetzsch ist lizenziert unter [CC BY 4.0](https://creativecommons.org/licenses/by/4.0/). Das Werk ist abrufbar auf [GitHub](https://github.com/bausteine-der-datenanalyse/w-pandas). Ausgenommen von der Lizenz sind alle Logos Dritter und anders gekennzeichneten Inhalte. 2025

Zitiervorschlag

Arnold, Lukas, Simone Arnold, Matthias Baitsch, Marc Fehr, Franca Hollmann, Maik Poetzsch, und Sebastian Seipel. 2025. „Bausteine Computergestützter Datenanalyse. Werkzeugbaustein Pandas“. <https://github.com/bausteine-der-datenanalyse/w-pandas>.

BibTeX-Vorlage

```
@misc{BCD-w-pandas-2025,  
  title={Bausteine Computergestützter Datenanalyse. Werkzeugbaustein Pandas},  
  author={Arnold, Lukas and Arnold, Simone and Baitsch, Matthias and Fehr, Marc and Hollmann, Franca and Poetzsch, Maik and Seipel, Sebastian},  
  year={2025},  
  url={https://github.com/bausteine-der-datenanalyse/w-pandas}}
```

Voraussetzungen

Die Bearbeitungszeit dieses Bausteins beträgt circa 8 Stunden. Folgende Bausteine sollte Sie bereits bearbeitet haben:

- w-Python
- w-NumPy

In diesem Baustein werden die folgenden Module verwendet:

- NumPy
- Pandas
- Matplotlib
- zoneinfo (optional)

Im Baustein werden folgende Daten verwendet:

- Zahnwachstum bei Meerschweinchen [CSV-Datei](#)
- Fahrzeugdaten aus der Zeitschrift Motor Trend [GitHub](#)
- Vermessung von Pinguinen an der Palmer Station [GitHub](#)
- Kursdaten der Microsoft-Aktie [kaggle](#)

Lernziele

In diesem Baustein lernen Sie ...

- die Datenstrukturen des Moduls Pandas Series und DataFrame kennen.
- wie Operationen in Pandas ausgeführt werden.
- wie Grafiken mit Pandas erstellt werden.
- Dateien einzulesen (und zu schreiben).

1 Einleitung

Das Modul Pandas wurde für die Arbeit mit strukturierten Daten konzipiert. Pandas erleichtert die Analyse insbesondere von in Tabellenform vorliegenden Daten, da es mit dem DataFrame eine leicht zu benutzende Struktur für die Verarbeitung unterschiedlicher Datentypen und fehlenden Werten bietet. Wie NumPy erlaubt Pandas vektorisierte Operationen, ohne mit Hilfe einer Schleife jedes Element eines Sammeltyps durchlaufen zu müssen. Pandas integriert darüber hinaus Funktionalitäten anderer Module und bietet unter anderem einen einheitlichen Zugang zu:

- Datumsinformationen und Zeitreihen
- Grafikerstellung
- Einlesen von Dateien

Das Modul Pandas wird mit dem Befehl `import pandas` geladen. Als Kürzel hat sich `pd` etabliert. Da Pandas auf dem Modul NumPy aufbaut, werden häufig beide Module geladen. Viele Funktionen und Methoden von NumPy und Pandas sind miteinander kompatibel.

```
import numpy as np
import pandas as pd
```

1.1 Die Datenstrukturen Series und DataFrame

Pandas führt die zwei Klassen `Series` und `DataFrame` ein.

- `Series` sind eindimensionale Arrays, die genau einen Datentyp haben.
- `DataFrame` sind zweidimensionale Arrays, die spaltenweise aus `Series` bestehen und so verschiedene Datentypen enthalten können. (Durch hierarchische Indexierung sind mehrdimensionale Datenstrukturen möglich, siehe [MultiIndex](#).)

Beide Datenstrukturen verfügen über einen Index, der in der Ausgabe angezeigt wird.

Der Index beginnt wie in der Pythonbasis bei 0.

```
0    Frühschicht
1    Frühschicht
2    Spätschicht
dtype: string
```

Der Index ist standardmäßig numerisch, kann aber mit beliebigen Werten versehen werden.

Der Index kann angepasst werden.

```
Montag    Frühschicht
Dienstag  Frühschicht
Mittwoch  Spätschicht
dtype: string
```

1.1.1 Series

Series werden mit der Funktion `pd.Series(data)` erstellt. `data` kann ein Einzelwert, ein Sammeltyp oder ein NumPy-Array sein.

```
einzelwert_series = pd.Series('Hallo Welt!')
print(f"Series aus Einzelwert:\n{einzelwert_series}")

numerische_series = pd.Series([1, 2, 3])
print(f"\nSeries aus Liste:\n{numerische_series}")

alphanumerische_series = pd.Series(('a', '5', 'g'))
print(f"\nSeries aus Tupel:\n{alphanumerische_series}")

boolean_series = pd.Series(np.array([True, False, True])) # NumPy-Array
print(f"\nSeries aus NumPy-Array:\n{boolean_series}")
```

Series aus Einzelwert:

```
0    Hallo Welt!
dtype: object
```

Series aus Liste:

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

Series aus Tuplel:

```
0    a
1    5
2    g
dtype: object
```

Series aus NumPy-Array:

```
0    True
1   False
2    True
dtype: bool
```

Beim Anlegen einer `pd.Series` können verschiedene Parameter übergeben werden:

- `pd.Series(data, dtype = 'float')` legt den Datentyp der Series fest.
- `pd.Series(data, index = ['A1', 'B2', 'C3'])` übergibt Werte für den Index.
- `pd.Series(data, name = 'der Name')` legt einen Namen für die Series fest.

```
numerische_series = pd.Series([1, 2, 3], dtype = 'float', index = ['A1', 'B2', 'C3'], name =  
print(numerische_series)
```

```
A1    1.0
B2    2.0
C3    3.0
Name: Gleitkommazahlen, dtype: float64
```

Für eine bestehende Series können Name und Index über entsprechende Attribute aufgerufen und geändert werden. Um den Datentyp zu ändern, wird die Methode `pd.Series.astype()` verwendet. Eine Übersicht der in Pandas verfügbaren Datentypen finden Sie in der [Pandas-Dokumentation](#).

```
print(f"Name der Series: {numerische_series.name}")
numerische_series.name = 'Fließkommazahlen'

print(f"Index der Series: {numerische_series.index}")
numerische_series.index = ['eins', 'zwei', 'drei']

numerische_series = numerische_series.astype('string')
print(f"\nDie geänderte Series:\n{numerische_series}")
```

```
Name der Series: Gleitkommazahlen
Index der Series: Index(['A1', 'B2', 'C3'], dtype='object')
```

Die geänderte Series:

```
eins    1.0
zwei    2.0
drei    3.0
Name: Fließkommazahlen, dtype: string
```

1.1.2 Aufgabe Series

Ändern Sie den Datentyp des Objekts 'numerische_series' in Ganzzahl und wählen Sie einen neuen Namen für die Series aus. `numerische_series = pd.Series([1, 2, 3], dtype = 'float', index = ['A1', 'B2', 'C3'], name = 'Gleitkommazahlen')`

 Tipp 1: Musterlösung dtype

```
numerische_series = pd.Series([1, 2, 3], dtype = 'float', index = ['A1', 'B2', 'C3'], name
numerische_series.name = 'Ganzzahlen'

# falls numerische_series vorher vom dtype string ist,
# muss erst in dtype float konvertiert werden
# numerische_series = numerische_series.astype('float')

numerische_series = numerische_series.astype('int')

print(numerische_series)

A1    1
B2    2
C3    3
Name: Ganzzahlen, dtype: int64
```

1.1.3 DataFrame

Ein `DataFrame` wird mit der Funktion `pd.DataFrame([data])` angelegt. `data` ist listenartig, kann aber aus einem Einzelwert, einer Series, einem Numpy-Array oder aus mehreren Series und Sammeltypen bestehen.

```

einzelwert_df = pd.DataFrame(['Hallo Welt!'])
print(einzelwert_df, "\n")

df_aus_listen = pd.DataFrame([[1, 2, 3], [4, 5, 6]])
print(df_aus_listen, "\n")

df_aus_series = pd.DataFrame([alphanumerische_series, boolean_series])
print(df_aus_series, "\n")

df_aus_verschieden = pd.DataFrame([np.array([True, False, True]), alphanumerische_series, [1, 2, 3]])
print(df_aus_verschieden)

```

```

      0
0  Hallo Welt!

```

```

      0  1  2
0  1  2  3
1  4  5  6

```

```

      0      1      2
0      a      5      g
1  True  False  True

```

```

      0      1      2
0  True  False  True
1      a      5      g
2      1      2      3

```

Beim Anlegen eines DataFrames können ebenfalls verschiedene Parameter übergeben werden:

- `pd.DataFrame(data, dtype = 'float')` legt den Datentyp des DataFrames für alle Werte fest. Wird der Parameter nicht übergeben, wählt Pandas einen passenden Datentyp für jede Spalte aus.
- `pd.DataFrame(data, index = ['A1', 'B2', 'C3'])` übergibt Werte für den Index.
- `pd.DataFrame(data, columns = ['Spalte1', 'Spalte2'])` übergibt Werte für die Spaltenbeschriftung.

Um Daten spaltenweise einzutragen, kann der DataFrame zum einen mit dem Attribut `DataFrame.T` transponiert werden. Dabei müssen die Spaltenbeschriftungen als Argument `index` und die Indexbeschriftung als Argument `columns` übergeben werden.

```
df_transponiert = pd.DataFrame([[1, 2, 3], [True, False, True]], index = ['Spalte 1', 'Spalte 2'])
print(df_transponiert)
```

	Spalte 1	Spalte 2
Zeile 1	1	True
Zeile 2	2	False
Zeile 3	3	True

Eine direkte Zuordnung der Beschriftungen ist möglich, indem zuerst der transponierte DataFrame angelegt und anschließend über die Attribute `.index` und `.columns` die Beschriftungen eingetragen werden.

```
df_transponiert = pd.DataFrame([[1, 2, 3], [True, False, True]]).T
df_transponiert.columns = ['Spalte 1', 'Spalte 2']
df_transponiert.index = ['Zeile 1', 'Zeile 2', 'Zeile 3']
print(df_transponiert)
```

	Spalte 1	Spalte 2
Zeile 1	1	True
Zeile 2	2	False
Zeile 3	3	True

Das Anlegen von transponierten DataFrames hat den Nachteil, dass Pandas die Datentypen der eingegebenen Daten spaltenweise verwaltet. Bei der zeilenweise erfolgenden Eingabe von Daten unterschiedlichen Datentyps wird ein für alle Spalten passender Datentyp gewählt. Im folgenden Beispiel wird deshalb von Pandas der Datentyp `object` für gemischte Datentypen gewählt.

```
df_transponiert = pd.DataFrame([[1, 2, 3], ['a', 'b', 'c']], index = ['Zahlen', 'Buchstaben'])
print(df_transponiert)
print(f"\n{df_transponiert.dtypes}")
```

	Zahlen	Buchstaben
0	1	a
1	2	b
2	3	c

```
Zahlen      object
Buchstaben   object
dtype: object
```

Zum anderen kann ein DataFrame direkt aus einem Dictionary erzeugt werden. Dabei werden die Schlüssel als Spaltenbeschriftung verwendet.

```
df = pd.DataFrame({'Spalte 1': [1, 2, 3], 'Spalte 2': [4.1, 5.6, 6.0]}, index = ['oben', 'mitte', 'unten'])
print(df)
```

	Spalte 1	Spalte 2
oben	1	4.1
mitte	2	5.6
unten	3	6.0

Außerdem kann ein DataFrame durch Zuweisung von Daten erweitert werden.

```
# einen leeren DataFrame erzeugen
df = pd.DataFrame()

# Zuweisung von Daten
df['Spaltenbeschriftung'] = [1, 2, 3]
df['zweite Spalte'] = alphanumerische_series

print(df)
```

	Spaltenbeschriftung	zweite Spalte
0	1	a
1	2	5
2	3	g

💡 Tipp 2: Der Index

In den meisten Fällen ist der von 0 bis n-1 reichende Index am praktischsten. Der numerische Index hilft bei der Auswahl von Indexbereichen (Slicing) und der Arbeit mit mehreren Datenstrukturen. Probieren Sie einmal aus, was passiert, wenn Sie einen DataFrame aus zwei Series mit unterschiedlichen Indizes erstellen.

Auch widerspricht das Auslagern beschreibender oder gemessener Variablen in den Index dem Konzept tidy data, einem System zum Strukturieren von Datensätzen, das Sie im [Methodenbaustein Einlesen strukturierter Datensätze](#) kennenlernen.

Bestehende DataFrames können ähnlich wie Series modifiziert werden. Um den Datentyp einer oder mehrerer Spalten zu ändern, wird die Methode `pd.DataFrame.astype()` verwendet.

```
df = pd.DataFrame({'Spalte 1': ['1', '2', '3'], 'Spalte 2': [True, False, True]})
print(f"Die Datentypen von df:\n{df.dtypes}")

# Datentyp von Spalte 1 ändern
df['Spalte 1'] = df['Spalte 1'].astype('string')
print(f"\nDie Datentypen von df:\n{df.dtypes}")
```

Die Datentypen von df:

```
Spalte 1    object
Spalte 2     bool
dtype: object
```

Die Datentypen von df:

```
Spalte 1    string[python]
Spalte 2         bool
dtype: object
```

Ebenso kann allen Spalten eines DataFrames ein Datentyp zugewiesen werden.

```
df = df.astype('string')
print(f"\nDie Datentypen von df:\n{df.dtypes}")
```

Die Datentypen von df:

```
Spalte 1    string[python]
Spalte 2    string[python]
dtype: object
```

Um unterschiedliche Datentypen zuzuweisen, wird ein Dictionary verwendet.

```
df = df.astype({'Spalte 1': 'int', 'Spalte 2': 'bool'})
print(f"\nDie Datentypen von df:\n{df.dtypes}")
```

Die Datentypen von df:

```
Spalte 1    int64
Spalte 2     bool
dtype: object
```

Spaltennamen und Index eines bestehenden `DataFrame` können über entsprechende Attribute oder Methoden verändert werden. Die Spaltennamen können über das Attribut `pd.DataFrame.columns` geändert werden. Dabei wird eine Liste übergeben, deren Länge der Spaltenanzahl entsprechen muss. Der Index kann über das Attribut `pd.DataFrame.index` geändert werden, indem eine Liste zugewiesen wird. Dabei muss die Länge der Liste der Anzahl Zeilen entsprechen.

```
# ändern der Spaltennamen über das Attribut .columns
df.columns = ['Spalte1', 'Spalte2']
df.index = [1, 2, 3]
print(df)
```

	Spalte1	Spalte2
1	1	True
2	2	True
3	3	True

Mit der Methode `pd.DataFrame.rename(columns = {"alt1": "neu1", "alt2": "neu2"}, index = {"alt1": "neu1", "alt2": "neu2"}, inplace = True)` können Spalten- und Zeilenbeschriftungen als Dictionary übergeben werden. Auf diese Weise können alle oder ausgewählte Beschriftungen geändert werden. Durch das Argument `inplace = True` erfolgt die Zuweisung direkt ohne Neuzuweisung des Objekts.

```
df.rename(columns = {'Spalte1': 'Spalte_1', 'Spalte2': 'Spalte_2'}, index = {1: 'A1', 2: 'B2'})
print(df)
```

	Spalte_1	Spalte_2
A1	1	True
B2	2	True
C3	3	True

Mit der Methode `pd.DataFrame.reset_index(inplace = True, drop = True)` wird der Index auf die Standardwerte zurückgesetzt. Wird der Parameter `drop = False` gesetzt, wird der alte Index als Spalte an Indexposition 0 in den `DataFrame` eingefügt.

```
df.reset_index(inplace = True, drop = True)
print(df)
```

	Spalte_1	Spalte_2
0	1	True
1	2	True
2	3	True

1.1.4 Aufgabe DataFrame

Erstellen Sie einen DataFrame.

- Die erste Spalte soll die Zahlen von 1 bis 12 enthalten und mit 'Nummer' beschriftet werden. Die zweite Spalte soll die Monatsnamen des Jahres enthalten und mit 'Monat' beschriftet werden.
- Fügen Sie nachträglich die Series 'ferien' als dritte Spalte mit der Spaltenbeschriftung 'Ferien' ein.
ferien = [False, False, False, True, False, True, True, True, False, True, False, True]

💡 Tipp 3: Musterlösung

```
ferien = [False, False, False, True, False, True, True, True, False, True, False, True]

df = pd.DataFrame({
    'Nummer': list(range(1,13)),
    'Monat': ['Januar', 'Februar', 'März', 'April', 'Mai', 'Juni', 'Juli', 'August', 'September', 'Oktober', 'November', 'Dezember'],
})

df['Ferien'] = ferien

print(df)
```

	Nummer	Monat	Ferien
0	1	Januar	False
1	2	Februar	False
2	3	März	False
3	4	April	True
4	5	Mai	False
5	6	Juni	True
6	7	Juli	True
7	8	August	True
8	9	September	False
9	10	Oktober	True
10	11	November	False
11	12	Dezember	True

1.2 Deskriptive Datenanalyse mit Pandas

Pandas bietet einige praktische Funktionen, um den Aufbau eines Datensatzes und die enthaltenen Daten zu beschreiben. Als Beispieldatensatz dienen Daten zur Länge zahnbildender Zellen bei Meerschweinchen, die Vitamin C direkt (VC) oder in Form von Orangensaft (OJ) in unterschiedlichen Dosen erhielten.

Code-Block 1.1

```
dateipfad = "01-daten/ToothGrowth.csv"
meerschweinchen = pd.read_csv(filepath_or_buffer = dateipfad, sep = ',', header = 0, \
    names = ['ID', 'len', 'supp', 'dose'], dtype = {'ID': 'int', 'len': 'float', 'dose': 'float'})
```

Crampton, E. W. 1947. „THE GROWTH OF THE ODONTOBLASTS OF THE INCISOR TOOTH AS A CRITERION OF THE VITAMIN C INTAKE OF THE GUINEA PIG“. The Journal of Nutrition 33 (5): 491–504. <https://doi.org/10.1093/jn/33.5.491>

Der Datensatz kann in R mit dem Befehl “ToothGrowth” aufgerufen werden.

Ein Ausschnitt des Datensatzes:

	ID	len	supp	dose
0	1	4.2	VC	0.5
10	11	16.5	VC	1.0
20	21	23.6	VC	2.0
30	31	15.2	OJ	0.5
40	41	19.7	OJ	1.0
50	51	25.5	OJ	2.0

Die Methode `pd.DataFrame.info()` erzeugt eine Beschreibung des Datensatzes.

```
meerschweinchen.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 60 entries, 0 to 59
Data columns (total 4 columns):
#   Column  Non-Null Count  Dtype
---  -
0   ID      60 non-null      int64
```

```

1   len      60 non-null    float64
2   supp     60 non-null    category
3   dose     60 non-null    float64
dtypes: category(1), float64(2), int64(1)
memory usage: 1.7 KB

```

Die Dimensionen einer Series oder eines DataFrame können mit dem Attribut **shape** abgerufen werden. Der DataFrame hat 60 Zeilen und 4 Spalten.

```
print(meerschweinchen.shape)
```

```
(60, 4)
```

Die Methode `pd.DataFrame.describe()` erzeugt eine beschreibende Statistik für einen DataFrame. Standardmäßig werden alle numerischen Spalten berücksichtigt. Mit dem Parameter `include` können die zu berücksichtigenden Spalten vorgegeben werden. `include = all` berücksichtigt alle Spalten, was nicht unbedingt sinnvoll ist, da auf diese Weise auch die Spalte mit den ID-Nummern der Meerschweinchen ausgewertet wird.

```
print(meerschweinchen.describe(include = 'all'))
```

	ID	len	supp	dose
count	60.000000	60.000000	60	60.000000
unique	NaN	NaN	2	NaN
top	NaN	NaN	OJ	NaN
freq	NaN	NaN	30	NaN
mean	30.500000	18.813333	NaN	1.166667
std	17.464249	7.649315	NaN	0.628872
min	1.000000	4.200000	NaN	0.500000
25%	15.750000	13.075000	NaN	0.500000
50%	30.500000	19.250000	NaN	1.000000
75%	45.250000	25.275000	NaN	2.000000
max	60.000000	33.900000	NaN	2.000000

Mit dem Parameter `include` kann eine Liste zu berücksichtigender Datentypen übergeben werden. Der Parameter `exclude` schließt auf die gleiche Weise Datentypen von der Ausgabe aus.

```
print(meerschweinchen.describe(include = ['float']))
```

	len	dose
count	60.000000	60.000000
mean	18.813333	1.166667
std	7.649315	0.628872
min	4.200000	0.500000
25%	13.075000	0.500000
50%	19.250000	1.000000
75%	25.275000	2.000000
max	33.900000	2.000000

```
print(meerschweinchen.describe(include = ['category']))
```

	supp
count	60
unique	2
top	0J
freq	30

Die Methode `pd.DataFrame.count()` zählt alle nicht fehlenden Werte in jeder Spalte oder mit `pd.DataFrame.count(axis = 'columns')` in jeder Zeile.

```
meerschweinchen.count(axis = 'rows') # der Standardwert von axis ist 'rows'
```

```
ID      60
len      60
supp     60
dose     60
dtype: int64
```

Die Methode `pd.Series.value_counts()` zählt die Anzahl der Merkmalsausprägungen in einer Series. Die Methode kann auch auf einen DataFrame angewendet werden, dann wird die Häufigkeit jeder einzigartigen Zeile gezählt (was hier nicht sinnvoll ist).

```
meerschweinchen['dose'].value_counts()
```

```
dose
0.5    20
1.0    20
2.0    20
Name: count, dtype: int64
```

Die Methode `pd.unique()` listet alle einzigartigen Werte einer Series auf.

```
meerschweinchen['dose'].unique()
```

```
array([0.5, 1. , 2. ])
```

1.3 Slicing

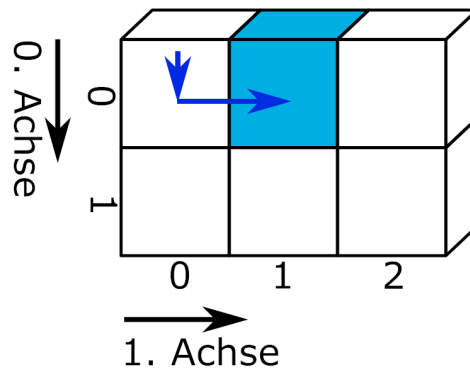


Abbildung 1.1: zweidimensionaler Datensatz

slicing von Marc Fehr ist lizenziert unter [CC-BY-4.0](#) und abrufbar auf [GitHub](#). Die Grafik wurde auf den gezeigten Teil beschnitten und die obenstehende Beschriftung entfernt. 2024

Pandas bringt eigene Werkzeuge für die Auswahl von Indexbereichen mit. Der Slice Operator aus der Pythonbasis wird deshalb nur kurz vorgestellt.

1.3.1 Slice Operator

Mit dem Slice Operator können wie bei einer Liste Indexbereiche aus einer Series ausgewählt werden.

```
zehn_zahlen = pd.Series(range(0, 10))  
print(zehn_zahlen[3:6])
```

```
3    3
4    4
5    5
dtype: int64
```

Mit dem Slice Operator werden die Zeilen eines DataFrames ausgewählt.

```
print(meerschweinchen[7:12])
```

	ID	len	supp	dose
7	8	11.2	VC	0.5
8	9	5.2	VC	0.5
9	10	7.0	VC	0.5
10	11	16.5	VC	1.0
11	12	16.5	VC	1.0

Durch Angabe eines Spaltennamens wird die entsprechende Spalte ausgewählt, die als Series zurückgegeben wird. Durch das Anfügen eines zweiten Slice Operators ist es möglich, wie bei einem eindimensionalen Datensatz die Werte in einem bestimmten Indexbereich abzurufen. Dies wird verkettete Indexierung genannt.

```
print(meerschweinchen['dose'][10:15], "\n")
print(type(meerschweinchen['dose'][10:15]))
```

```
10    1.0
11    1.0
12    1.0
13    1.0
14    1.0
Name: dose, dtype: float64
```

```
<class 'pandas.core.series.Series'>
```

Warning 1: Verkettete Indexierung

Die verkettete Indexierung erzeugt in Pandas abhängig vom Kontext eine Kopie des Objekts oder greift auf den Speicherbereich des Objekts zu. Mit Pandas 3.0 wird die verkettete Indexierung nicht mehr unterstützt, das Anlegen einer Kopie wird zum Standard werden. Weitere Informationen erhalten Sie im zitierten Link.

“Whether a copy or a reference is returned for a setting operation, may depend on the context. This is sometimes called **chained assignment** and should be avoided. See [Returning a View versus Copy](#).”

([Pandas Dokumentation](#))

1.3.2 Slicing mit Pandas-Methoden

Für das Slicing von Series und DataFrames werden in Pandas die Methoden `.iloc[]` und `.loc[]` verwendet.

- `.loc[]` arbeitet mit Index- oder Spaltenbeschriftungen, akzeptiert aber auch ein boolesches Array.
- `.iloc[]` arbeitet mit Ganzzahlen, akzeptiert aber auch ein boolesches Array.

Für das Slicing von Series wird eine Bereichsangabe übergeben, bspw. `pd.Series.iloc[5:8]`. Für das Slicing von DataFrames werden zwei durch ein Komma getrennte Bereichsangaben übergeben, wobei an erster Stelle die Zeilen und an zweiter Stelle die Spalten ausgewählt werden, bspw. `pd.DataFrame.iloc[5, 2:4]`. Um alle Zeilen oder Spalten auszuwählen kann der Doppelpunkt verwendet werden, etwa `pd.DataFrame.iloc[5, :]`.

Beschriftungsbasiertes Slicing mit `.loc[]`

Für eine Series interpretiert `.loc` übergebene Zeichen als Indexbeschriftung. Buchstaben und andere Zeichen werden wie strings in Anführungszeichen übergeben, bspw. `.loc['e']`, Zahlen ohne Anführungszeichen. Neben Einzelwerten ('a' oder 0) können Listen oder Arrays (['a', 'b', 'c'] oder [1, 2, 3]) und Slices übergeben werden ('a':'c' oder 0:2). Das Slicing mit einem Einzelwert führt zur Rückgabe eines Einzelwerts (sog. Skalar).

Warning 2: inklusives Slicing

Anders als die Pythonbasis und das Slicing mit `.iloc[]` zählt Pandas beim beschriftungsbasierten Slicing inklusiv, gibt also die letzte ausgewählte Position mit aus.

```
# Nummern
zehn_zahlen = pd.Series(range(0, 10))
print("Rückgabe eines Einzelwerts:", zehn_zahlen.loc[5]) # Einzelwert
print(zehn_zahlen.loc[[2, 4, 7]]) # Liste
print(zehn_zahlen.loc[5:7], "\n") # Slice

# Buchstaben und andere Zeichen
sechs_zahlen = pd.Series(list(range(0, 6)), index = ['a', 'b', 'c', 'd', 'e', 'f'])
print("Rückgabe eines Einzelwerts:", sechs_zahlen.loc['c']) # Einzelwert
print(sechs_zahlen.loc[['c', 'f', 'a']]) # Liste
print(sechs_zahlen.loc['c':'e']) # Slice
```

Rückgabe eines Einzelwerts: 5

```
2    2
4    4
7    7
dtype: int64
5    5
6    6
7    7
dtype: int64
```

Rückgabe eines Einzelwerts: 2

```
c    2
f    5
a    0
dtype: int64
c    2
d    3
e    4
dtype: int64
```

Die Interpretation als Beschriftung bedeutet, dass bei einem nicht numerischen Index als Slice übergebene Zahlen nicht gefunden werden. Hier muss mit der übergebenen Beschriftung gearbeitet werden.

```
try:
    print(sechs_zahlen.loc[2:4])
except Exception as error:
    print(error)

print("\n", sechs_zahlen.loc['c':'e'], sep = "\n")
```

cannot do slice indexing on Index with these indexers [2] of type int

```
c    2
d    3
e    4
dtype: int64
```

Für DataFrames funktioniert das Slicing genauso.

```
print(meerschweinchen.loc[18:22, ['len', 'dose']])
```

	len	dose
18	18.8	1.0
19	15.5	1.0
20	23.6	2.0
21	18.5	2.0
22	33.9	2.0

1.3.3 Indexbasiertes Slicing mit .iloc[]

Die Methode `.iloc[]` ermöglicht die Auswahl von Ausschnitten basierend auf Indexpositionen. Die Methode akzeptiert die gleichen Eingaben wie die Methode `.loc[]`.

Warning 3: exklusives Slicing

Beim Slicing mit der Methode `.iloc[]` zählt Pandas wie die Pythonbasis exklusiv.

Das Slicing mit Einzelwerten führt zur Ausgabe eines Einzelwertes. Die Methode akzeptiert ebenfalls eine Liste oder ein Slice.

```
print("Rückgabe eines Einzelwerts:", meerschweinchen.iloc[27, 2]) # Einzelwerte
print(meerschweinchen.iloc[[27, 29, 52], 2:4]) # Liste und Slice
```

```
Rückgabe eines Einzelwerts: VC
   supp  dose
27   VC   2.0
29   VC   2.0
52   OJ   2.0
```

Die Methoden `.head()` und `.tail()`

Vereinfachte Varianten des indexbasierten Slicings sind die Methoden `.head(n=5)` und `.tail(n=5)`, mit denen die ersten bzw. letzten `n` Zeilen eines DataFrame oder einer Series ausgegeben werden können. Über den optionalen Parameter `n` kann die Anzahl der angezeigten Zeilen gesteuert werden. Die Methoden eignen sich gut, um sich einen ersten Eindruck von einem Datensatz zu verschaffen.

```
print(meerschweinchen.head(3), "\n")
print(meerschweinchen.tail(3))
```

	ID	len	supp	dose
0	1	4.2	VC	0.5
1	2	11.5	VC	0.5
2	3	7.3	VC	0.5

	ID	len	supp	dose
57	58	27.3	OJ	2.0
58	59	29.4	OJ	2.0
59	60	23.0	OJ	2.0

Ebenso können Series damit betrachtet werden.

```
print(meerschweinchen['len'].tail(3))
```

```
57    27.3
58    29.4
59    23.0
Name: len, dtype: float64
```

1.4 Aufgaben Slicing

Gegeben ist eine Pandas Series ‘temperatures_2021’ mit den durchschnittlichen Monatstemperaturen.

```
temperatures_2021 = pd.Series([2, 4, 7, 12, 19, 23, 25, 23, 18, 15, 9, 5],
                               index = ['Jan', 'Feb', 'Mär', 'Apr', 'Mai', 'Jun',
                                         'Jul', 'Aug', 'Sep', 'Okt', 'Nov', 'Dez'])
```

1. Wählen Sie die Temperaturen für die Frühlingsmonate (März bis Mai) aus.

2. Wählen Sie die Temperaturen für die letzten drei Monate des Jahres einmal mit Hilfe des Slicing Operators und einmal mit Hilfe der Pandas-Methoden aus.
3. Wählen Sie mit der Methode `.loc[]` die Spalten 'dose' und 'len' des DataFrame meerschweinchen aus und geben Sie die ersten 4 und die letzten 3 Zeilen aus. (Code zum Einlesen der Datei siehe Code-Block 1.1)
4. Die Methoden `.loc[]` und `.iloc[]` akzeptieren auch ein boolesches Array als Eingabe. Geben Sie aus der Spalte 'dose' des DataFrame meerschweinchen alle Zeilen mit dem Wert 2.0 aus.

Tipp 4: Musterlösung Slicing

Aufgabe 1

```
print(temperaturen_2021.loc[['Mär', 'Apr', 'Mai']])
```

```
Mär      7
Apr     12
Mai     19
dtype: int64
```

Aufgabe 2

```
print(temperaturen_2021[-3:], "\n")
print(temperaturen_2021.iloc[-3:])
```

```
Okt     15
Nov      9
Dez      5
dtype: int64
```

```
Okt     15
Nov      9
Dez      5
dtype: int64
```

Aufgabe 3

```
print(meerschweinchen.loc[:, ['dose', 'len']].head(n = 4), "\n")
print(meerschweinchen.loc[:, ['dose', 'len']].tail(n = 3))
```

```
dose    len
```

```
0    0.5    4.2
1    0.5   11.5
2    0.5    7.3
3    0.5    5.8
```

```
      dose    len
57    2.0   27.3
58    2.0   29.4
59    2.0   23.0
```

Aufgabe 4

```
# Slice aus Series
# print(meerschweinchen['dose'].loc[meerschweinchen['dose'] == 2.0])

# Slice aus DataFrame
print(meerschweinchen.loc[meerschweinchen['dose'] == 2.0, ['dose']])
```

```
      dose
20    2.0
21    2.0
22    2.0
23    2.0
24    2.0
25    2.0
26    2.0
27    2.0
28    2.0
29    2.0
50    2.0
51    2.0
52    2.0
53    2.0
54    2.0
55    2.0
56    2.0
57    2.0
58    2.0
59    2.0
```

1.5 Datenstrukturen verbinden

DataFrames sind flexible Datenspeicher. Mit der Funktion `pd.concat()` können Series und DataFrames zusammengeführt werden.

- Mit dem Argument `pd.concat(ignore_index = True)` wird ein neuer Index generiert.
- Mit dem Argument `pd.concat(axis = 1)` werden die übergebenen Objekte spaltenweise zusammengeführt.

```
series_1 = pd.Series([1, 2])
series_2 = pd.Series([4, 5])
print(pd.concat([series_1, series_2]), "\n")
print(pd.concat([series_1, series_2], ignore_index = True), "\n")
print(pd.concat([series_1, series_2], ignore_index = True, axis = 1))
```

```
0    1
1    2
0    4
1    5
dtype: int64
```

```
0    1
1    2
2    4
3    5
dtype: int64
```

```
   0  1
0  1  4
1  2  5
```

Gleichermaßen können DataFrames verbunden werden.

```
temperaturen_2021 = pd.Series([2, 4, 7, 12, 19, 23, 25, 23, 18, 15, 9, 5],
                              index = ['Jan', 'Feb', 'Mär', 'Apr', 'Mai', 'Jun',
                                       'Jul', 'Aug', 'Sep', 'Okt', 'Nov', 'Dez'])

temperaturen_2022 = pd.Series([3, 6, 9, 13, 18, 21, 24, 23, 19, 14, 8, 4],
                              index = ['Jan', 'Feb', 'Mär', 'Apr', 'Mai', 'Jun',
                                       'Jul', 'Aug', 'Sep', 'Okt', 'Nov', 'Dez'])
```

```

temperaturen_2023 = pd.Series([-3, -1, 4, 9, 15, 20, 20, 19, 16, 15, 7, 6],
                               index = ['Jan', 'Feb', 'Mär', 'Apr', 'Mai', 'Jun',
                                         'Jul', 'Aug', 'Sep', 'Okt', 'Nov', 'Dez'])

temperaturen_2024 = pd.Series([-1, 2, 5, 8, 17, 24, 25, 20, 17, 14, 9, 2],
                               index = ['Jan', 'Feb', 'Mär', 'Apr', 'Mai', 'Jun',
                                         'Jul', 'Aug', 'Sep', 'Okt', 'Nov', 'Dez'])

# Series zu DataFrame verbinden
df1 = pd.concat([temperaturen_2021, temperaturen_2022], axis = 1)
df2 = pd.concat([temperaturen_2023, temperaturen_2024], axis = 1)

# DataFrames verbinden
temperaturen = pd.concat([df1, df2], axis = 1)
temperaturen.columns = [2021, 2022, 2023, 2024]
print(temperaturen)

```

	2021	2022	2023	2024
Jan	2	3	-3	-1
Feb	4	6	-1	2
Mär	7	9	4	5
Apr	12	13	9	8
Mai	19	18	15	17
Jun	23	21	20	24
Jul	25	24	20	25
Aug	23	23	19	20
Sep	18	19	16	17
Okt	15	14	15	14
Nov	9	8	7	9
Dez	5	4	6	2

1.6 Einfügen und löschen in Datenstrukturen

Der Operator `del` aus der Pythonbasis löscht Spalten aus einem DataFrame, zum Beispiel: `del DataFrame['Spaltenname']`. Pandas bringt aber auch eigene Methoden mit, um Einträge zeilen- / oder spaltenweise zu ergänzen und zu löschen.

- `pd.DataFrame.drop(labels = None, axis = 0, index = None, columns = None, inplace = False)` entfernt Zeilen oder Spalten nach den mit dem Parameter `labels` als Einzelwert ('Spalte 1') oder als Liste (['Spalte 1', 'Spalte 2']) übergebenen Beschriftungen. Der Parameter `axis` steuert, ob Zeilen oder Spalten (`axis = 1`) entfernt

werden sollen. Die Parameter `index` und `columns` sind alternative Möglichkeiten, Index- oder Spaltenbeschriftungen direkt zu übergeben und ersetzen die Parameter `labels` und `axis`.

- `pd.DataFrame.insert(loc, column, value)` fügt eine Spalte an Position `loc` mit dem Spaltennamen `column` und dem Inhalt `value` ein. Falls `value` eine Series mit abweichendem Index ist, kann über das Attribut `value = Series.values` auf die enthaltenen Werte der Series zugegriffen und diese in den bestehenden Index eingefügt werden (andernfalls gleicht Pandas die Indizes der Series und des DataFrames ab und fügt nur die Werte übereinstimmender Indizes ein).
- Werte können zeilenweise mit der Methode `pd.DataFrame.loc[index] = value` eingefügt werden. Falls `value` eine Series ist, muss über das Attribut `value = Series.values` auf die enthaltenen Werte der Series zugegriffen werden, da Pandas andernfalls versucht, den Index der Series mit den Spaltennamen des DataFrames abzugleichen. Wird als `value` ein Einzelwert übergeben, füllt dieser die gesamte Zeile aus.

1.7 Aufgaben verbinden und löschen

Legen Sie einen leeren DataFrame `df = pd.DataFrame()` an.

1. Fügen Sie die Spalten 'len' und 'dose' aus dem DataFrame 'meerschweinchen' ein.
2. Löschen Sie alle ungeraden Zeilen aus dem DataFrame `df`.
3. Benutzen Sie die Indizes der verbleibenden geraden Zeilen des DataFrame `df`, um die entsprechenden Zeilen aus der Spalte 'ID' des DataFrame 'meerschweinchen' auszuwählen. Fügen Sie diese als Spalte an Indexposition 0 in den DataFrame `df` ein.

Tipp 5: Musterlösung verbinden und löschen

1. Aufgabe

```
df = pd.DataFrame()

# Alternative 1
df['len'] = meerschweinchen['len']

# Alternative 2
df.insert(loc = 1, column = 'dose', value = meerschweinchen['dose'])

print(df.head(), "\n", df.shape)
```

```
len  dose
```

```

0    4.2    0.5
1   11.5    0.5
2    7.3    0.5
3    5.8    0.5
4    6.4    0.5
(60, 2)

```

2. Aufgabe

```

df = df.drop(index = range(1, len(df), 2))

print(df.head(), "\n", df.shape)

```

```

      len  dose
0    4.2    0.5
2    7.3    0.5
4    6.4    0.5
6   11.2    0.5
8    5.2    0.5
(30, 2)

```

3. Aufgabe

```

df.insert(loc = 0, column = 'ID', value = meerschweinchen.loc[df.index, 'ID'])

print(df.head(), "\n")
print(df.tail(), "\n")
print("df.shape:", df.shape)

```

```

      ID  len  dose
0     1  4.2    0.5
2     3  7.3    0.5
4     5  6.4    0.5
6     7 11.2    0.5
8     9  5.2    0.5

      ID  len  dose
50    51 25.5    2.0
52    53 22.4    2.0
54    55 24.8    2.0
56    57 26.4    2.0

```

```
58 59 29.4 2.0
```

```
df.shape: (30, 3)
```

Quellen

https://pandas.pydata.org/docs/user_guide/dsintro.html https://pandas.pydata.org/docs/user_guide/basics.html

2 Operationen

Pandas erlaubt wie NumPy vektorisierte Operationen, das heißt, Berechnungen mit einer Series oder einem DataFrame werden auf jedes Element angewendet. So können die Rechenoperatoren direkt verwendet werden.

```
print("Temperaturen in Celsius:")
print(27 * "=")
print(temperatures, "\n")

print("Temperaturen in Fahrenheit:")
print(27 * "=")
print(temperatures * 9/5 + 32)
```

Temperaturen in Celsius:

```
=====
      2021  2022  2023  2024
Jan      2    3   -3   -1
Feb      4    6   -1    2
Mär      7    9    4    5
Apr     12   13    9    8
Mai     19   18   15   17
Jun     23   21   20   24
Jul     25   24   20   25
Aug     23   23   19   20
Sep     18   19   16   17
Okt     15   14   15   14
Nov      9    8    7    9
Dez      5    4    6    2
```

Temperaturen in Fahrenheit:

```
=====
      2021  2022  2023  2024
Jan   35.6  37.4  26.6  30.2
Feb   39.2  42.8  30.2  35.6
Mär   44.6  48.2  39.2  41.0
```

Apr	53.6	55.4	48.2	46.4
Mai	66.2	64.4	59.0	62.6
Jun	73.4	69.8	68.0	75.2
Jul	77.0	75.2	68.0	77.0
Aug	73.4	73.4	66.2	68.0
Sep	64.4	66.2	60.8	62.6
Okt	59.0	57.2	59.0	57.2
Nov	48.2	46.4	44.6	48.2
Dez	41.0	39.2	42.8	35.6

Auch boolsche Operationen können direkt ausgeführt werden.

```
print("Minusgrade:")
print(27 * "=")
print(temperaturen < 0)
```

```
Minusgrade:
=====
      2021    2022    2023    2024
Jan  False  False   True   True
Feb  False  False   True  False
Mär  False  False  False  False
Apr  False  False  False  False
Mai  False  False  False  False
Jun  False  False  False  False
Jul  False  False  False  False
Aug  False  False  False  False
Sep  False  False  False  False
Okt  False  False  False  False
Nov  False  False  False  False
Dez  False  False  False  False
```

2.1 Zeilen- und spaltenweise Operationen

Pandas umfasst eine Vielzahl von Methoden, die arithmetische, summarische, boolsche und Indexfunktionen umsetzen. Eine vollständige Übersicht finden Sie hier: <https://pandas.pydata.org/docs/reference/index.html>.

In der Regel werden die Funktionen standardmäßig spaltenweise angewendet. Mit dem Argument `axis = 1` wird die jeweilige Funktion zeilenweise ausgeführt. Die Funktionen sind auch für Series verfügbar.

Im Folgenden werden einige Methoden exemplarisch vorgestellt.

2.1.1 arithmetische Funktionen

Die Methoden `pd.DataFrame.add()`, `pd.DataFrame.sub()`, `pd.DataFrame.mul()`, `pd.DataFrame.div()`, `pd.DataFrame.floordiv()`, `pd.DataFrame.mod()` und `pd.DataFrame.pow()` entsprechen den Grundrechenarten mit den Operatoren `+`, `-`, `*`, `/`, `//`, `%`, `**`. Sie eignen sich gut für verkettete Operationen.

```
print("Temperaturen in Fahrenheit:")
print(27 * "=")
print(temperaturen.mul(9).div(5).add(32))
```

Temperaturen in Fahrenheit:

```
=====
      2021  2022  2023  2024
Jan   35.6  37.4  26.6  30.2
Feb   39.2  42.8  30.2  35.6
Mär   44.6  48.2  39.2  41.0
Apr   53.6  55.4  48.2  46.4
Mai   66.2  64.4  59.0  62.6
Jun   73.4  69.8  68.0  75.2
Jul   77.0  75.2  68.0  77.0
Aug   73.4  73.4  66.2  68.0
Sep   64.4  66.2  60.8  62.6
Okt   59.0  57.2  59.0  57.2
Nov   48.2  46.4  44.6  48.2
Dez   41.0  39.2  42.8  35.6
```

Außerdem kann mit dem Parameter `fill_value` ein Füllwert für fehlende Werte spezifiziert werden (dieser wird vor der Operation eingesetzt). Wie NumPys `np.nan` umfasst auch Pandas einen speziellen fehlenden Wert: `pd.NA` (achten Sie auf den Datentyp der Ausgabe). Der Umgang mit fehlenden Werten wird ausführlich im [Methodenbaustein Einlesen strukturierter Datensätze](#) behandelt.

```
missing_value = pd.Series([1, pd.NA, 3])
print(missing_value.add(1, fill_value = -999), "\n")
print(missing_value.add(1, fill_value = np.nan), "\n")
print(missing_value.add(1, fill_value = pd.NA))
```

```
0      2
1    -998
2      4
dtype: int64
```

```
0      2.0
1     NaN
2      4.0
dtype: float64
```

```
0      2
1    <NA>
2      4
dtype: object
```

2.1.2 summarische Funktionen

- `pd.DataFrame.mean()` ermittelt den Durchschnitt.
- `pd.DataFrame.median()` ermittelt den Median.
- `pd.DataFrame.mode()` ermittelt den Modus.
- `pd.DataFrame.sum()` ermittelt die Summe.
- `pd.DataFrame.cumsum()` ermittelt die kummulierte Summe.
- `pd.DataFrame.min()` und `pd.DataFrame.max()` ermitteln Minimum bzw. Maximum.
- `pd.DataFrame.cummin()` und `pd.DataFrame.cummax()` ermittelt das kummulierte Minimum bzw. Maximum.

```
# spaltenweise
print("Mittlere Jahrestemperaturen")
print(27 * "=")
print(temperatures.mean(), "\n")

# zeilenweise
print("Monatliche Mindesttemperatur")
print(28 * "=")
print(temperatures.min(axis = 1))
```

```
Mittlere Jahrestemperaturen
=====
2021      13.500000
2022      13.500000
2023      10.583333
```

```
2024    11.833333
dtype: float64
```

Monatliche Mindesttemperatur

=====

```
Jan    -3
Feb    -1
Mär     4
Apr     8
Mai    15
Jun    20
Jul    20
Aug    19
Sep    16
Okt    14
Nov     7
Dez     2
dtype: int64
```

2.1.3 boolsche Funktionen

Pandas bietet wie die Pythonbasis verschiedene boolsche Funktionen.

`pd.DataFrame.isin(values)` prüft für jedes Element des DataFrame, ob dieses in `values` enthalten ist. Mit dem Operator `~` kann geprüft werden, ob die Elemente eines DataFrame nicht in `values` enthalten sind: `~pd.DataFrame.isin(values)`.

Die Funktionsausführung ist abhängig vom Datentyp des in `values` übergebenen Objekts.

- Wenn `values` eine Liste oder ein NumPy-Array ist, ist das Ergebnis True, wenn es eine Übereinstimmung mit einem der enthaltenen Elemente gibt.
- Ist `value` eine Series oder ein DataFrame, wird die Übereinstimmung positionsbasiert überprüft (siehe Beispiel).

i Hinweis 1: klassenabhängige Funktionsausführung

2.2 Einzelwerte oder Liste

Für Einzelwerte oder eine Liste wird die Übereinstimmung elementweise überprüft.

```
print(temperaturen, "\n")
print(temperaturen.isin([2, 3]))
```

	2021	2022	2023	2024
Jan	2	3	-3	-1
Feb	4	6	-1	2
Mär	7	9	4	5
Apr	12	13	9	8
Mai	19	18	15	17
Jun	23	21	20	24
Jul	25	24	20	25
Aug	23	23	19	20
Sep	18	19	16	17
Okt	15	14	15	14
Nov	9	8	7	9
Dez	5	4	6	2

	2021	2022	2023	2024
Jan	True	True	False	False
Feb	False	False	False	True
Mär	False	False	False	False
Apr	False	False	False	False
Mai	False	False	False	False
Jun	False	False	False	False
Jul	False	False	False	False
Aug	False	False	False	False
Sep	False	False	False	False
Okt	False	False	False	False
Nov	False	False	False	False
Dez	False	False	False	True

2.3 NumPy-Array

Für ein NumPy-Array wird die Übereinstimmung elementweise überprüft (vergleiche zum nächsten Reiter).

```
print(type(temperaturen[2021].values), "\n")

print(temperaturen.isin(temperaturen[2021].values))
```

```
<class 'numpy.ndarray'>
```

	2021	2022	2023	2024
Jan	True	False	False	False

Feb	True	False	False	True
Mär	True	True	True	True
Apr	True	False	True	False
Mai	True	True	True	False
Jun	True	False	False	False
Jul	True	False	False	True
Aug	True	True	True	False
Sep	True	True	False	False
Okt	True	False	True	False
Nov	True	False	True	True
Dez	True	True	False	True

2.4 Series

Für eine Series wird die Übereinstimmung positionswise geprüft (vergleiche zum vorherigen Reiter). Der Index muss übereinstimmen.

```
print(temperaturen.isin(temperaturen[2021]), "\n")

temperaturen_2021_falscher_index = pd.Series([2, 4, 7, 12, 19, 23, 25, 23, 18, 15, 9, 5])
temperaturen_2021_falscher_index.index = ['A', 'B', 'C', 'D', 'E', 'F', 'Jul', 'Aug', 'Sep', 'Okt', 'Nov', 'Dez']

print("Der Index der Series lautet:\n['A', 'B', 'C', 'D', 'E', 'F', 'Jul', 'Aug', 'Sep', 'Okt', 'Nov', 'Dez']")
print(temperaturen.isin(temperaturen_2021_falscher_index))
```

	2021	2022	2023	2024
Jan	True	False	False	False
Feb	True	False	False	False
Mär	True	False	False	False
Apr	True	False	False	False
Mai	True	False	False	False
Jun	True	False	False	False
Jul	True	False	False	True
Aug	True	True	False	False
Sep	True	False	False	False
Okt	True	False	True	False
Nov	True	False	False	True
Dez	True	False	False	False

Der Index der Series lautet:
 ['A', 'B', 'C', 'D', 'E', 'F', 'Jul', 'Aug', 'Sep', 'Okt', 'Nov', 'Dez'].

Das Ergebnis an den Indexpositionen A-F ist immer False.

	2021	2022	2023	2024
Jan	False	False	False	False
Feb	False	False	False	False
Mär	False	False	False	False
Apr	False	False	False	False
Mai	False	False	False	False
Jun	False	False	False	False
Jul	True	False	False	True
Aug	True	True	False	False
Sep	True	False	False	False
Okt	True	False	True	False
Nov	True	False	False	True
Dez	True	False	False	False

2.5 DataFrame

Für einen DataFrame wird die Übereinstimmung positionsweise geprüft. Index und Spaltennamen müssen übereinstimmen (Index siehe Reiter Series).

```
temperaturen_2021_df = pd.DataFrame(temperaturen[2021])
print(temperaturen.isin(temperaturen_2021_df), "\n")

temperaturen_2021_df.columns = [2035]
print(temperaturen.isin(temperaturen_2021_df), "\n")
```

	2021	2022	2023	2024
Jan	True	False	False	False
Feb	True	False	False	False
Mär	True	False	False	False
Apr	True	False	False	False
Mai	True	False	False	False
Jun	True	False	False	False
Jul	True	False	False	False
Aug	True	False	False	False
Sep	True	False	False	False
Okt	True	False	False	False
Nov	True	False	False	False
Dez	True	False	False	False

	2021	2022	2023	2024
--	------	------	------	------

Jan	False	False	False	False
Feb	False	False	False	False
Mär	False	False	False	False
Apr	False	False	False	False
Mai	False	False	False	False
Jun	False	False	False	False
Jul	False	False	False	False
Aug	False	False	False	False
Sep	False	False	False	False
Okt	False	False	False	False
Nov	False	False	False	False
Dez	False	False	False	False

💡 Tipp 6: Überraschungen vermeiden

Eine klassenabhängige Funktionsausführung kann, wenn das Verhalten unbemerkt bleibt, die Ergebnisse einer Datenanalyse verfälschen. Um dies zu verhindern, sollten Sie 3 allgemeine Ratschläge befolgen:

1. Schauen Sie in die Dokumentation der jeweiligen Funktion. Python und viele Module entwickeln sich dynamisch, sodass sich das Verhalten einer Funktion verändern kann.
2. Gehen Sie schrittweise vor und lassen sich die Zwischenergebnisse von Arbeitsschritten mit der Funktion `print()` ausgeben.
3. Bei großen Datenmengen ist es häufig einfacher, mit eigens erzeugten Testdaten zu arbeiten. Ein zehnzeiliger DataFrame mit den Datentypen und der Struktur der Arbeitsdaten, ist leichter zu überblicken. Nutzen Sie einen solchen Testdatensatz um die von Ihnen verwendeten Funktionen zu überprüfen.

Eine Gruppe von Funktionen setzt logische Vergleiche um.

Funktion	Vergleich
<code>pd.DataFrame.lt(other)</code>	kleiner
<code>pd.DataFrame.le(other)</code>	kleiner gleich
<code>pd.DataFrame.eq(other)</code>	gleich
<code>pd.DataFrame.ne(other)</code>	ungleich
<code>pd.DataFrame.ge(other)</code>	größer gleich
<code>pd.DataFrame.gt(other)</code>	größer

```
print(temperaturen.le(2), "\n")
print(temperaturen[2021].gt(5))
```

	2021	2022	2023	2024
Jan	True	False	True	True
Feb	False	False	True	True
Mär	False	False	False	False
Apr	False	False	False	False
Mai	False	False	False	False
Jun	False	False	False	False
Jul	False	False	False	False
Aug	False	False	False	False
Sep	False	False	False	False
Okt	False	False	False	False
Nov	False	False	False	False
Dez	False	False	False	True

Jan	False
Feb	False
Mär	True
Apr	True
Mai	True
Jun	True
Jul	True
Aug	True
Sep	True
Okt	True
Nov	True
Dez	False

Name: 2021, dtype: bool

2.5.1 Verwendung der Methoden .agg() und .apply()

Pandas bringt zwei eigene Methoden mit, um Operationen zeilen- oder spaltenweise auszuführen. `DataFrame.agg()` (oder auch `DataFrame.aggregate()`) aggregiert einen `DataFrame` zeilen- oder spaltenweise durch eine Funktion. Die Pandas-Methode `DF.apply()` wendet eine Funktion zeilen- oder spaltenweise auf einen `DataFrame` an. Die Methoden sind also sehr ähnlich und führen in den meisten Fällen zum selben Ergebnis.

Beide Funktionen führen mit dem Argument `axis = 1` Operationen zeilenweise aus.

2.6 Funktion

```
def my_plus_ten(x):  
    y = x + 10  
    return y  
  
print(temperaturen.agg(my_plus_ten), "\n")  
print(temperaturen.apply(my_plus_ten))
```

	2021	2022	2023	2024
Jan	12	13	7	9
Feb	14	16	9	12
Mär	17	19	14	15
Apr	22	23	19	18
Mai	29	28	25	27
Jun	33	31	30	34
Jul	35	34	30	35
Aug	33	33	29	30
Sep	28	29	26	27
Okt	25	24	25	24
Nov	19	18	17	19
Dez	15	14	16	12

	2021	2022	2023	2024
Jan	12	13	7	9
Feb	14	16	9	12
Mär	17	19	14	15
Apr	22	23	19	18
Mai	29	28	25	27
Jun	33	31	30	34
Jul	35	34	30	35
Aug	33	33	29	30
Sep	28	29	26	27
Okt	25	24	25	24
Nov	19	18	17	19
Dez	15	14	16	12

2.7 Funktionsname

```
print(temperaturen.agg("sum"), "\n")
print(temperaturen.apply("sum"))
```

```
2021    162
2022    162
2023    127
2024    142
dtype: int64
```

```
2021    162
2022    162
2023    127
2024    142
dtype: int64
```

2.8 Liste von Funktionen

```
print(temperaturen.agg(["sum", "mean", "median"]), "\n")
print(temperaturen.apply(["sum", "mean", "median"]))
```

	2021	2022	2023	2024
sum	162.0	162.0	127.000000	142.000000
mean	13.5	13.5	10.583333	11.833333
median	13.5	13.5	12.000000	11.500000

	2021	2022	2023	2024
sum	162.0	162.0	127.000000	142.000000
mean	13.5	13.5	10.583333	11.833333
median	13.5	13.5	12.000000	11.500000

2.9 Dictionary von Funktionen

```
print(temperaturen.agg({2021: "sum", 2022: "mean", 2023: "median", 2024: "min"}), "\n")
print(temperaturen.apply({2021: "sum", 2022: "mean", 2023: "median", 2024: "min"}), "\n")
```

```

2021    162.0
2022     13.5
2023     12.0
2024     -1.0
dtype: float64

```

```

2021    162.0
2022     13.5
2023     12.0
2024     -1.0
dtype: float64

```

Besonders nützlich ist die Möglichkeit, Funktionen, die normalerweise auf eine Series angewendet werden, auf jedes Element der Series anzuwenden. Dafür wird die lambda Syntax verwendet: `lambda x: x + 1`. lambda ist ein Platzhalter und kann als “für jedes x tue:” gelesen werden. So kann beispielsweise die Anzahl der Zeichen in jeder Zeile bestimmt werden.

```

# Auf die Series angewendet
print(len(str(temperaturen[2021])), "\n")

# Elementweise angewendet
print(temperaturen[2021].agg(lambda x: len(str(x))), "\n") # deprecated
print(temperaturen[2021].apply(lambda x: len(str(x))), "\n")

```

```

144

```

```

Jan     1
Feb     1
Mär     1
Apr     2
Mai     2
Jun     2
Jul     2
Aug     2
Sep     2
Okt     2
Nov     1
Dez     1
Name: 2021, dtype: int64

```

```

Jan     1
Feb     1

```

```

Mär    1
Apr    2
Mai    2
Jun    2
Jul    2
Aug    2
Sep    2
Okt    2
Nov    1
Dez    1
Name: 2021, dtype: int64

```

Details zur Verwendung des [Lambda-Ausdrucks](#) finden Sie in der Dokumentation.

Der Vollständigkeit wegen ist zu erwähnen, dass mit den Methoden `.map()` und `.transform()` weitere, sehr ähnliche Alternativen bestehen. Bei Interesse können Sie die Unterschiede [in diesem Artikel](#) nachlesen.

```

# print(temperaturen[2021].map(lambda x: len(str(x))))
# print(temperaturen[2021].transform(lambda x: len(str(x))), "\n")

```

2.10 Aufgaben Operationen

Gegeben ist der DataFrame `temperaturen`.

```

# Temperaturdaten
temperaturen_2021 = pd.Series([2, 4, 7, 12, 19, 23, 25, 23, 18, 15, 9, 5])
temperaturen_2022 = pd.Series([3, 6, 9, 13, 18, 21, 24, 23, 19, 14, 8, 4])
temperaturen_2023 = pd.Series([-3, -1, 4, 9, 15, 20, 20, 19, 16, 15, 7, 6])
temperaturen_2024 = pd.Series([-1, 2, 5, 8, 17, 24, 25, 20, 17, 14, 9, 2])

# DataFrame erzeugen
temperaturen = pd.concat([temperaturen_2021, temperaturen_2022, temperaturen_2023, temperaturen_2024])
temperaturen.columns = [2021, 2022, 2023, 2024]
temperaturen.index = ['Jan', 'Feb', 'Mär', 'Apr', 'Mai', 'Jun', 'Jul', 'Aug', 'Sep', 'Okt', 'Nov', 'Dez']

```

1. Bestimmen Sie für den Dataframe `temperaturen` die monatliche Mediantemperatur.
2. Ermitteln Sie die Monate mit einer Mediantemperatur größer gleich 21 Grad.
3. Geben Sie die Indexbeschriftung dieser Monate aus.

💡 Tipp 7: Musterlösung Aufgaben Operationen

1. Aufgabe

```
print(temperaturen.mean(axis = 1))
```

```
Jan      0.25
Feb      2.75
Mär      6.25
Apr     10.50
Mai     17.25
Jun     22.00
Jul     23.50
Aug     21.25
Sep     17.50
Okt     14.50
Nov      8.25
Dez      4.25
dtype: float64
```

2. Aufgabe

```
print(temperaturen.mean(axis = 1).ge(21))
```

```
Jan      False
Feb      False
Mär      False
Apr      False
Mai      False
Jun       True
Jul       True
Aug       True
Sep      False
Okt      False
Nov      False
Dez      False
dtype: bool
```

3. Aufgabe

```
print(temperaturen.index[temperaturen.mean(axis = 1).ge(21)], "\n")

# als Liste
print(list(temperaturen.index[temperaturen.mean(axis = 1).ge(21)]), "\n")

Index(['Jun', 'Jul', 'Aug'], dtype='object')

['Jun', 'Jul', 'Aug']
```

2.11 Suchen und ersetzen

Um die Indexposition eines bestimmten Werts zu bestimmen, kann die Numpy-Funktion `np.where()` verwendet werden. Diese gibt zwei Arrays mit den jeweiligen Zeilen- und Spaltennummern zurück.

```
print(np.where(temperaturen == 4))
```

```
(array([ 1,  2, 11]), array([0, 2, 1]))
```

Unter anderem befindet sich der Wert 4 in Zeile 1 in Spalte 0 oder auch in Zeile 2 in Spalte 2.

```
print(temperaturen.iloc[1, 0])
print(temperaturen.iloc[2, 2])
```

```
4
4
```

Pandas bietet zwei Methoden, um Werte zu ersetzen.

- `pd.DataFrame.replace(to_replace, value, *, inplace = False)` ersetzt `to_replace` mit `value`. Mit dem Argument `inplace = True` erfolgt dies direkt im Objekt.
- `pd.where(cond, other = nan, inplace = False)` behält `cond` und ersetzt alle anderen Werte mit `other` (standardmäßig ein Platzhalter für fehlende Werte). Mit dem Argument `inplace = True` erfolgt dies direkt im Objekt.

Die Syntax beider Funktionen unterscheidet sich leicht, wie im folgenden Beispiel zu sehen ist.

```
print(temperaturen.replace(to_replace = 25, value = 1000), "\n")
print(temperaturen.where(temperaturen == 25, other = 1000))
```

	2021	2022	2023	2024
Jan	2	3	-3	-1
Feb	4	6	-1	2
Mär	7	9	4	5
Apr	12	13	9	8
Mai	19	18	15	17
Jun	23	21	20	24
Jul	1000	24	20	1000
Aug	23	23	19	20
Sep	18	19	16	17
Okt	15	14	15	14
Nov	9	8	7	9
Dez	5	4	6	2

	2021	2022	2023	2024
Jan	1000	1000	1000	1000
Feb	1000	1000	1000	1000
Mär	1000	1000	1000	1000
Apr	1000	1000	1000	1000
Mai	1000	1000	1000	1000
Jun	1000	1000	1000	1000
Jul	25	1000	1000	25
Aug	1000	1000	1000	1000
Sep	1000	1000	1000	1000
Okt	1000	1000	1000	1000
Nov	1000	1000	1000	1000
Dez	1000	1000	1000	1000

2.12 Aufgaben suchen und ersetzen

1. Bestimmen Sie die Position der Werte im DataFrame 'temperaturen', die kleiner als 0 sind und geben Sie die Werte aus.
2. Ersetzen Sie alle Werte im DataFrame 'temperaturen', die kleiner sind als 0 durch den Wert 0 und geben Sie das Ergebnis aus.

💡 Tipp 8: Musterlösung suchen und ersetzen

1. Aufgabe

```
print(np.where(temperaturen <= 0))
print("Anzahl Werte:", len(np.where(temperaturen <= 0)[0]))

for i in range(len(np.where(temperaturen <= 0)[0])):
    print(temperaturen.iloc[np.where(temperaturen <= 0)[0][i], np.where(temperaturen <= 0)[1][i]])
```

```
(array([0, 0, 1]), array([2, 3, 2]))
Anzahl Werte: 3
-3
-1
-1
```

2. Aufgabe

```
print(temperaturen.where(temperaturen > 0, other = 0))
```

	2021	2022	2023	2024
Jan	2	3	0	0
Feb	4	6	0	2
Mär	7	9	4	5
Apr	12	13	9	8
Mai	19	18	15	17
Jun	23	21	20	24
Jul	25	24	20	25
Aug	23	23	19	20
Sep	18	19	16	17
Okt	15	14	15	14
Nov	9	8	7	9
Dez	5	4	6	2

2.13 Sortieren

Die Methode `DataFrame.sort_index(axis = 0, ascending = True, inplace = False)` sortiert entlang einer Achse, standardmäßig aufsteigend nach dem Index. Durch die Übergabe des Arguments `axis = 1` werden die Spalten sortiert. Mit dem Argument `ascending = False` wird absteigend sortiert. Das Argument `inplace = True` sorgt, wie gewohnt, dafür, dass das

Ergebnis des Sortiervorgangs direkt im Objekt gespeichert wird.

```
print(temperaturen.sort_index(), "\n")
print(temperaturen.sort_index(axis = 1, ascending = False))
```

	2021	2022	2023	2024
Apr	12	13	9	8
Aug	23	23	19	20
Dez	5	4	6	2
Feb	4	6	-1	2
Jan	2	3	-3	-1
Jul	25	24	20	25
Jun	23	21	20	24
Mai	19	18	15	17
Mär	7	9	4	5
Nov	9	8	7	9
Okt	15	14	15	14
Sep	18	19	16	17

	2024	2023	2022	2021
Jan	-1	-3	3	2
Feb	2	-1	6	4
Mär	5	4	9	7
Apr	8	9	13	12
Mai	17	15	18	19
Jun	24	20	21	23
Jul	25	20	24	25
Aug	20	19	23	23
Sep	17	16	19	18
Okt	14	15	14	15
Nov	9	7	8	9
Dez	2	6	4	5

Die Methode `DataFrame.sort_values(by, *, axis = 0, ascending = True, inplace = False)` sortiert Werte entlang einer Achse, standardmäßig entlang des Index (`axis = 0`). Dem Parameter `by` sind [laut Dokumentation](#) der Spaltenname als string bzw. eine Liste von Spaltennamen als string zu übergeben, nach denen sortiert werden soll. Wie im folgenden Code-Beispiel zu sehen ist, muss die numerische Spaltenbeschriftung jedoch auch in numerischer Form übergeben werden.

Wird mit dem Argument `axis = 1` entlang der zweiten Dimension sortiert, werden entsprechend Indexbeschriftungen übergeben.

```
# Sortieren nach numerischen Spaltenbeschriftungen
print(temperaturen.sort_values(by = 2021), "\n")
print(temperaturen.sort_values(by = [2021, 2023]), "\n")

# Sortieren nach als string übergebenen Spaltenbeschriftungen
# führt zu KeyError, die Fehlermeldung wird nicht vollständig abgefangen
try:
    print(temperaturen.sort_values(by = '2021'))
except Exception as error:
    print(error)
```

	2021	2022	2023	2024
Jan	2	3	-3	-1
Feb	4	6	-1	2
Dez	5	4	6	2
Mär	7	9	4	5
Nov	9	8	7	9
Apr	12	13	9	8
Okt	15	14	15	14
Sep	18	19	16	17
Mai	19	18	15	17
Jun	23	21	20	24
Aug	23	23	19	20
Jul	25	24	20	25

	2021	2022	2023	2024
Jan	2	3	-3	-1
Feb	4	6	-1	2
Dez	5	4	6	2
Mär	7	9	4	5
Nov	9	8	7	9
Apr	12	13	9	8
Okt	15	14	15	14
Sep	18	19	16	17
Mai	19	18	15	17
Aug	23	23	19	20
Jun	23	21	20	24
Jul	25	24	20	25

'2021'

2.14 Aufgaben Sortieren

1. Sortieren Sie den DataFrame 'meerschweinchen' absteigend nach der Zahnlänge ('len'). Welches Meerschweinchen hat die längste zahnbildende Zelle (gesucht ist die ID)?
2. Welches Meerschweinchen, welches die Dosis 1.0 erhielt, hat die längste zahnbildende Zelle (gesucht ist die ID)?

💡 Tipp 9: Musterlösung Sortieren

1. Aufgabe

```
print(meerschweinchen.sort_values(by = 'len', ascending = False).head(), "\n")  
print("Die ID lautet:", meerschweinchen.sort_values(by = 'len', ascending = False).iloc[0,
```

	ID	len	supp	dose
22	23	33.9	VC	2.0
25	26	32.5	VC	2.0
55	56	30.9	OJ	2.0
29	30	29.5	VC	2.0
58	59	29.4	OJ	2.0

Die ID lautet: 23

2. Aufgabe

```
dose_1 = meerschweinchen[meerschweinchen['dose'] == 1.0]  
print(dose_1.sort_values(by = 'len', ascending = False).head(), "\n")  
print("Die ID lautet:", dose_1.sort_values(by = 'len', ascending = False).iloc[0, 0])
```

	ID	len	supp	dose
49	50	27.3	OJ	1.0
43	44	26.4	OJ	1.0
46	47	25.8	OJ	1.0
45	46	25.2	OJ	1.0
42	43	23.6	OJ	1.0

Die ID lautet: 50

2.15 GroupBy

Die Methode `pd.groupby()` teilt einen DataFrame (oder eine Series) in Gruppen auf und gibt ein GroupBy-Objekt zurück. Das GroupBy-Objekt hat dieselben Spalten- und Zeilenbeschriftungen wie der DataFrame, das GroupBy-Objekt ist aber nach der Gruppeneinteilung sortiert. Operationen, die auf das GroupBy-Objekt angewendet werden, werden für jede Gruppe separat ausgeführt.

Dies kann am Datensatz ‘meerschweinchen’ im folgenden Panel nachvollzogen werden.

1. Reiter: Der Datensatz enthält 60 Einträge. Die ersten 30 Einträge haben in der Spalte ‘supp’ die Ausprägung VC für Vitamin C, die letzten 30 Einträge die Ausprägung OJ für Orangensaft.
2. Reiter: Mit der Methode `pd.groupby('supp')` kann der Datensatz nach den Merkmalsausprägungen in der Spalte ‘dose’ (0.5, 1 und 2) gruppiert werden.
3. Reiter: Auf das Groupby-Objekt können Operationen ausgeführt werden. Beispielsweise kann die Spalte ‘len’ ausgewählt und mit der Methode `.mean()` die mittlere Länge der zahnbildenden Zelle bestimmt werden.
4. Reiter: Ebenso kann nach den Ausprägungen mehrerer Merkmale gruppiert werden, indem diese als Liste übergeben werden `pd.groupby(by = ['supp', 'dose'])`.

2.16 DataFrame meerschweinchen

```
print(meerschweinchen.head(n = 12))
```

	ID	len	supp	dose
0	1	4.2	VC	0.5
1	2	11.5	VC	0.5
2	3	7.3	VC	0.5
3	4	5.8	VC	0.5
4	5	6.4	VC	0.5
5	6	10.0	VC	0.5
6	7	11.2	VC	0.5
7	8	11.2	VC	0.5
8	9	5.2	VC	0.5
9	10	7.0	VC	0.5
10	11	16.5	VC	1.0
11	12	16.5	VC	1.0

2.17 meerschweinchen gruppiert nach Verabreichungsart

Für die Methode `.head()` wurde das Argument `n` halbiert, um die gleiche Zeilenzahl in der Ausgabe anzeigen zu lassen, da auch diese Methode für jede der beiden Gruppen (VC und OJ) ausgeführt wird.

```
print(meerschweinchen.groupby('supp').head(n = 6))
```

	ID	len	supp	dose
0	1	4.2	VC	0.5
1	2	11.5	VC	0.5
2	3	7.3	VC	0.5
3	4	5.8	VC	0.5
4	5	6.4	VC	0.5
5	6	10.0	VC	0.5
30	31	15.2	OJ	0.5
31	32	21.5	OJ	0.5
32	33	17.6	OJ	0.5
33	34	9.7	OJ	0.5
34	35	14.5	OJ	0.5
35	36	10.0	OJ	0.5

2.18 Länge nach Verabreichungsart

```
print(meerschweinchen.groupby(by = 'supp')['len'].mean())
```

```
supp
OJ    20.663333
VC    16.963333
Name: len, dtype: float64
```

2.19 Länge nach Verabreichungsart und Dosis

```
print(meerschweinchen.groupby(by = ['supp', 'dose'])['len'].mean())
```

```
supp  dose
OJ    0.5    13.23
```

```

      1.0      22.70
      2.0      26.06
VC     0.5       7.98
      1.0      16.77
      2.0      26.14
Name: len, dtype: float64

```

2.20 Aufgaben GroupBy

Der Datensatz Motor Trend Car Road Tests (mtcars) stammt aus der us-amerikanischen Zeitschrift Motor Trend von 1974 und enthält Daten für 32 Autos.

Code-Block 2.1

```

mtcars = pd.read_csv(filepath_or_buffer = "01-daten/mtcars.csv", sep = ",")
mtcars.rename(columns = {'Unnamed: 0': 'car'}, inplace = True)

mtcars.head()

```

	car	mpg	cyl	disp	hp	drat	wt	qsec	vs	am	gear	carb
0	Mazda RX4	21.0	6	160.0	110	3.90	2.620	16.46	0	1	4	4
1	Mazda RX4 Wag	21.0	6	160.0	110	3.90	2.875	17.02	0	1	4	4
2	Datsun 710	22.8	4	108.0	93	3.85	2.320	18.61	1	1	4	1
3	Hornet 4 Drive	21.4	6	258.0	110	3.08	3.215	19.44	1	0	3	1
4	Hornet Sportabout	18.7	8	360.0	175	3.15	3.440	17.02	0	0	3	2

Spalte	Bedeutung
mpg	Kraftstoffverbrauch in Meilen pro Gallone
cyl	Anzahl Zylinder
disp	Hubraum in Kubikzoll
hp	Pferdestärken
drat	Hinterachsübersetzung
wt	Gewicht in 1000 Pfund
qsec	Zeit auf der Viertelmeile
vs	Motor (0 = V-Motor, 1 = Reihenmotor)
am	Schaltung (0 = Automatik, 1 = Handschaltung)
gear	Anzahl der Vorwärtsgänge
carb	Anzahl der Vergaser

Henderson and Velleman 1981. Building multiple regression models interactively. Biometrics 37: 391–411. Der Datensatz ist abrufbar auf [GitHub](#) und in R verfügbar.

1. Gruppieren Sie den Datensatz nach der Anzahl Zylinder und ermitteln Sie den durchschnittlichen Kraftstoffverbrauch für jede Gruppe.
2. Wie viele Liter auf 100 Kilometer sind es?
3. Gruppieren Sie den Datensatz nach der Anzahl der Zylinder und der Vergaser. Welche Gruppe ist am schnellsten auf der Viertelmeile?

Tipp 10: Musterlösung GroupBy

1. Aufgabe

```
mtcars.groupby(by = 'cyl')['mpg'].mean()
```

```
cyl
4    26.663636
6    19.742857
8    15.100000
Name: mpg, dtype: float64
```

2. Aufgabe

```
# 1 Meile = 1.60934 Kilometer
# 1 Gallone = 3.78541 Liter

mpg = mtcars.groupby(by = 'cyl')['mpg'].mean()

liter_100km = 1 / mpg.mul(1.60934).div(3.78541).div(100)

print(liter_100km)
```

```
cyl
4     8.821567
6    11.913932
8    15.577156
Name: mpg, dtype: float64
```

3. Aufgabe

```
print(mtcars.groupby(by = ['cyl', 'carb'])['qsec'].mean(), "\n")
print(mtcars.groupby(by = ['cyl', 'carb'])['qsec'].mean().index[-1], "\n")
```

```
cyl  carb
4    1    19.378000
     2    18.936667
6    1    19.830000
     4    17.670000
     6    15.500000
8    2    17.060000
     3    17.666667
     4    16.495000
     8    14.600000
Name: qsec, dtype: float64
```

```
(np.int64(8), np.int64(8))
```

Die Gruppe mit 8 Zylindern und 8 Vergasern ist am schnellsten. (Hinweis: Es handelt sich hierbei um einen sogenannten [MultiIndex](#).)

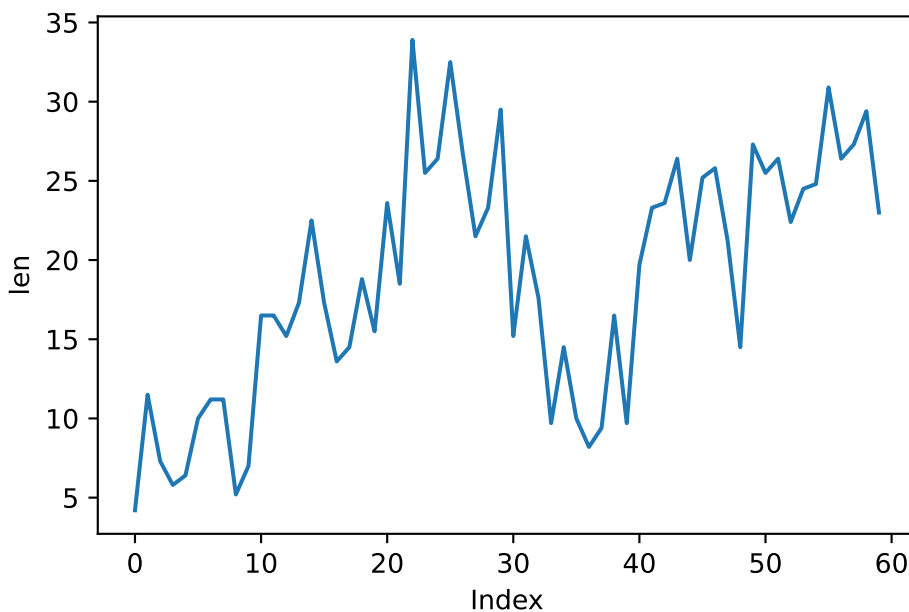
3 Grafikerstellung

Mit der Methode `pd.plot()` können Series und DataFrame grafisch dargestellt werden. Dazu greift Pandas auf matplotlib zurück. Die Syntax von Pandas ist einfacher, dafür bietet das Modul weniger Möglichkeiten als matplotlib. Jedoch können Befehle aus Pandas und aus matplotlib zur Grafikerstellung kombiniert werden (siehe [Werkzeugbaustein Matplotlib](#)).

3.1 Series

Eine Series wird gegen den Index geplottet. Standardmäßig wird ein Liniendiagramm gezeichnet. Mit den Parametern `xlabel` und `ylabel` können Achsenbeschriftungen eingetragen werden.

```
meerschweinchen['len'].plot(xlabel = 'Index', ylabel = 'len')
```



Mit dem Parameter `kind` kann der Grafiktyp geändert werden. Einige Möglichkeiten sind:

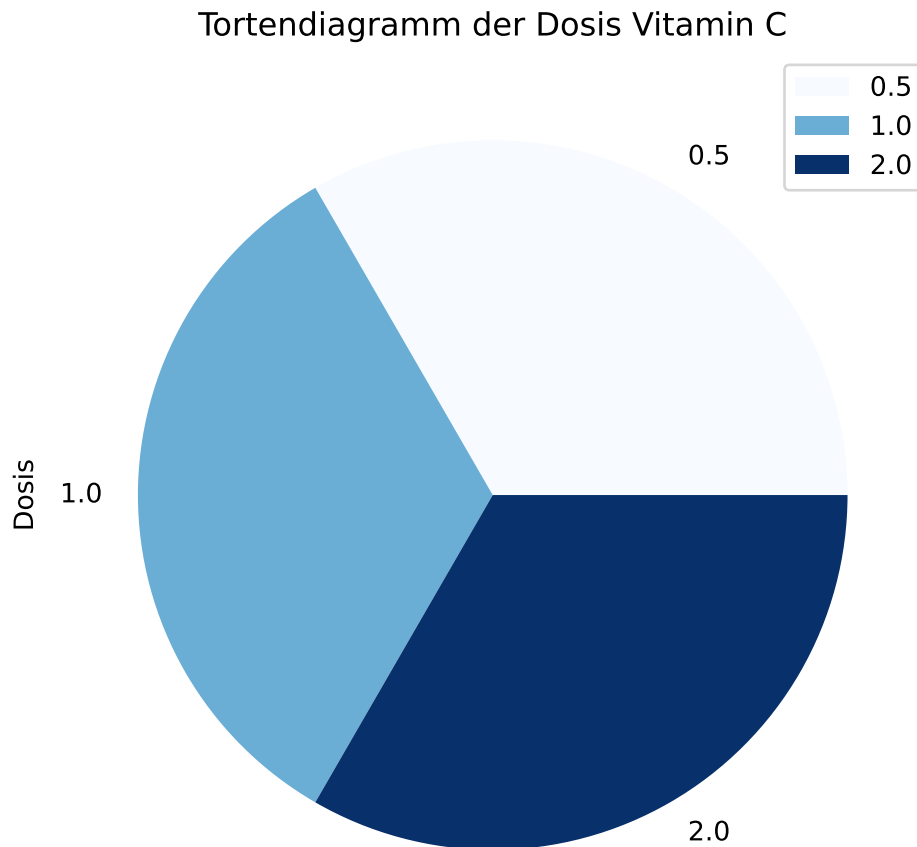
- `'line'`: Standardwert Liniendiagramm

- 'bar': vertikales Balkendiagramm
- 'hbar': horizontales Balkendiagramm
- 'hist': Histogramm
- 'box': Boxplot
- 'pie': Tortendiagramm

Einige sinnvolle Parameter sind:

- `colormap = palette` ändert die Farbpalette. Eine Liste der in matplotlib verfügbaren Paletten finden Sie in der [Dokumentation](#).
- `figsize = (Breite, Höhe)` Tupel der Bildgröße in Zoll
- `legend = True` zeichnet eine Legende ein.
- `title = 'Titel'` trägt einen Titel ein.
- `grid = True` fügt Gitternetzlinien ein.
- `xlim = (min, max) / ylim = (min, max)` setzt den Wertebereich der x- bzw. y-Achse.

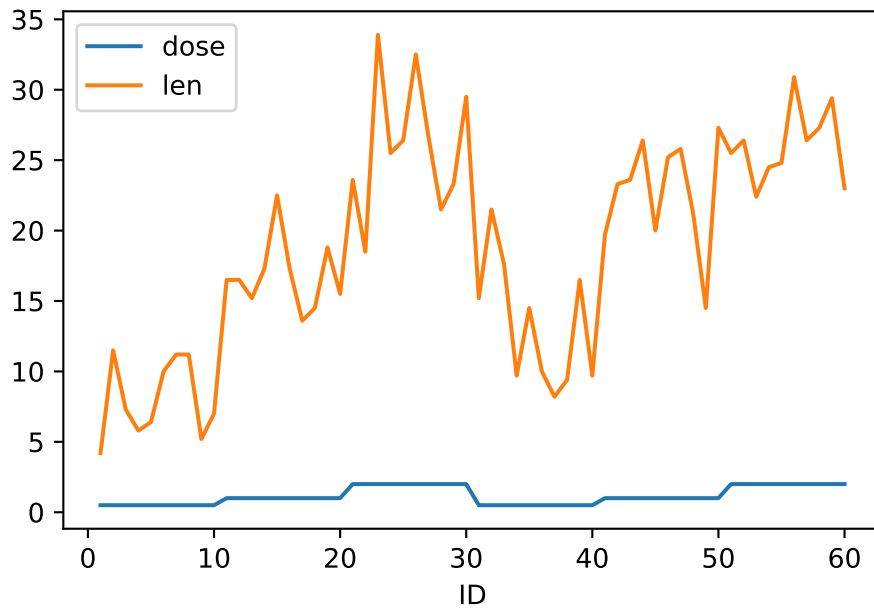
```
meerschweinchen['dose'].value_counts().plot(kind = 'pie', ylabel = 'Dosis', colormap = 'Blue
```



3.2 DataFrame

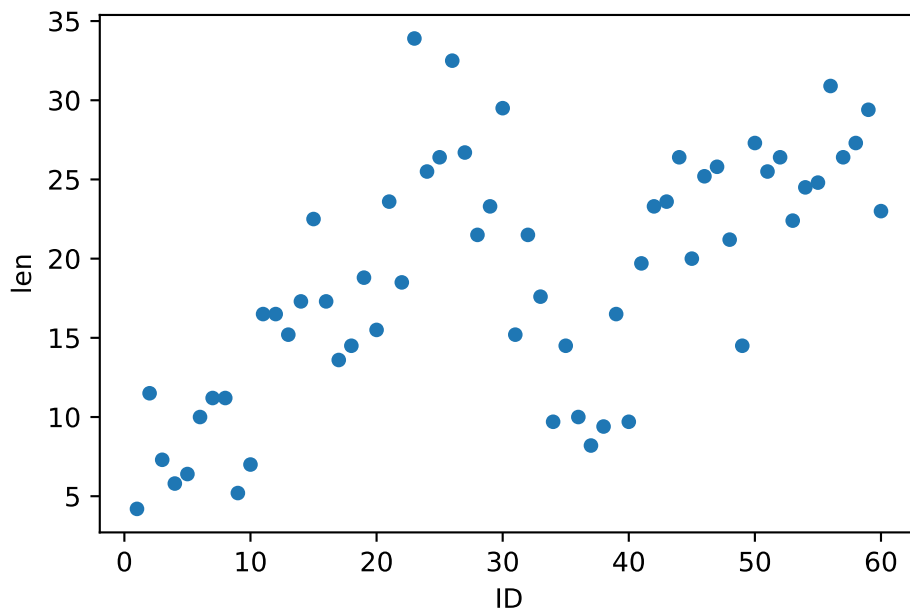
Für DataFrames sind weitere Optionen verfügbar. Mittels der Parameter `x` und `y` können Spalten oder Zeilen ausgewählt werden, die auf den jeweiligen Achsen aufgetragen werden sollen. `y` kann dabei auch eine Liste mit mehreren Einträgen enthalten.

```
meerschweinchen.plot(x = 'ID', y = ['dose', 'len'])
```



Für DataFrames ist das Streudiagramm als Diagrammtyp verfügbar.

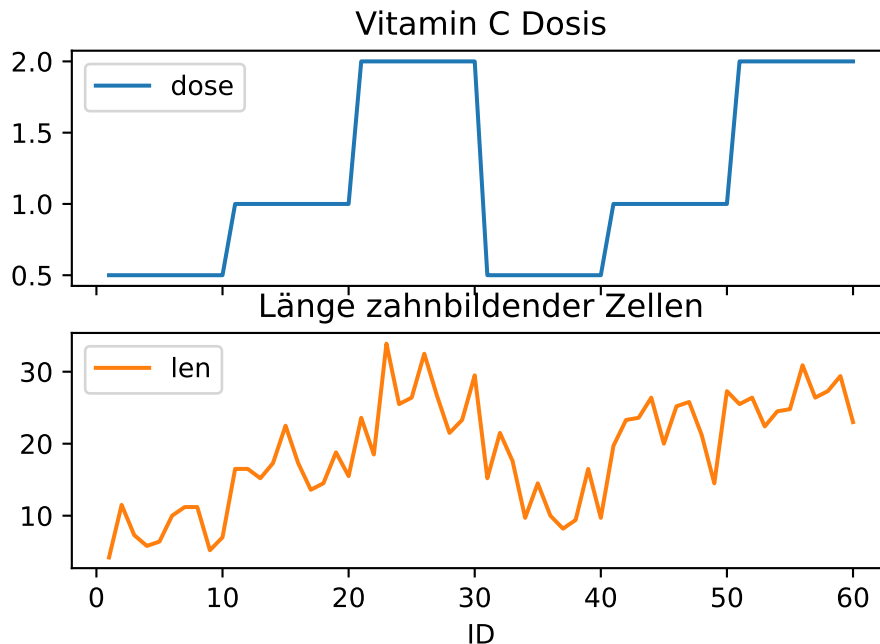
```
meerschweinchen.plot(x = 'ID', y = 'len', kind = 'scatter')
```



3.3 subplots

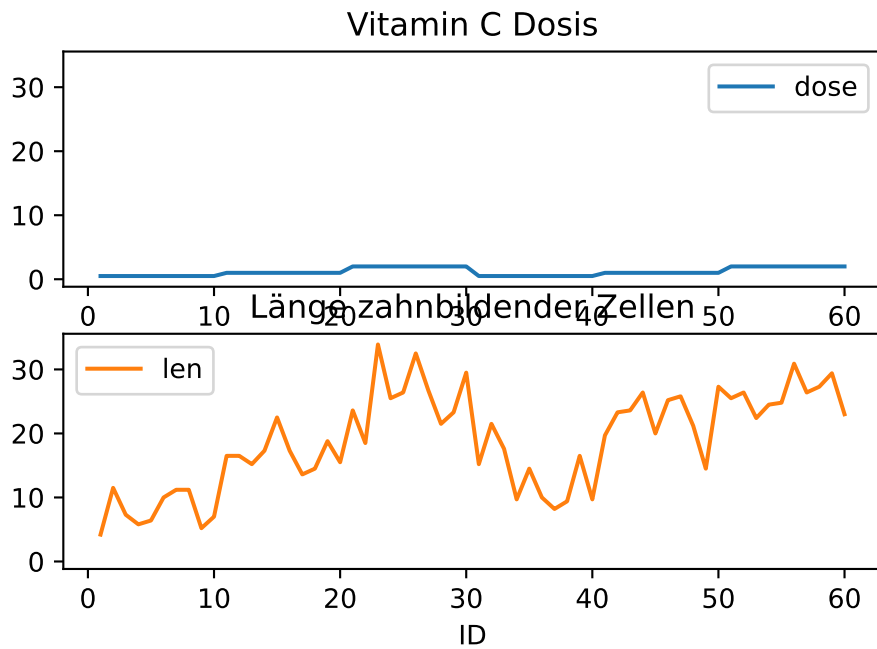
Mit dem Argument `subplots = True` werden Teilgrafiken erstellt. Dabei wird automatisch das Argument `sharex = True` gesetzt, sodass beide Teilgrafiken eine gemeinsame x-Achse nutzen teilen. Dem Parameter `title` können Überschriften für jede Teilgrafik als Liste übergeben werden.

```
meerschweinchen.plot(x = 'ID', y = ['dose', 'len'], subplots = True, title = ['Vitamin C Dosis', 'Länge zahnbildender Zellen'])
```



Ebenso ist das Argument `sharey = True` verfügbar. Das Argument `sharex` wird auf `False` gesetzt.

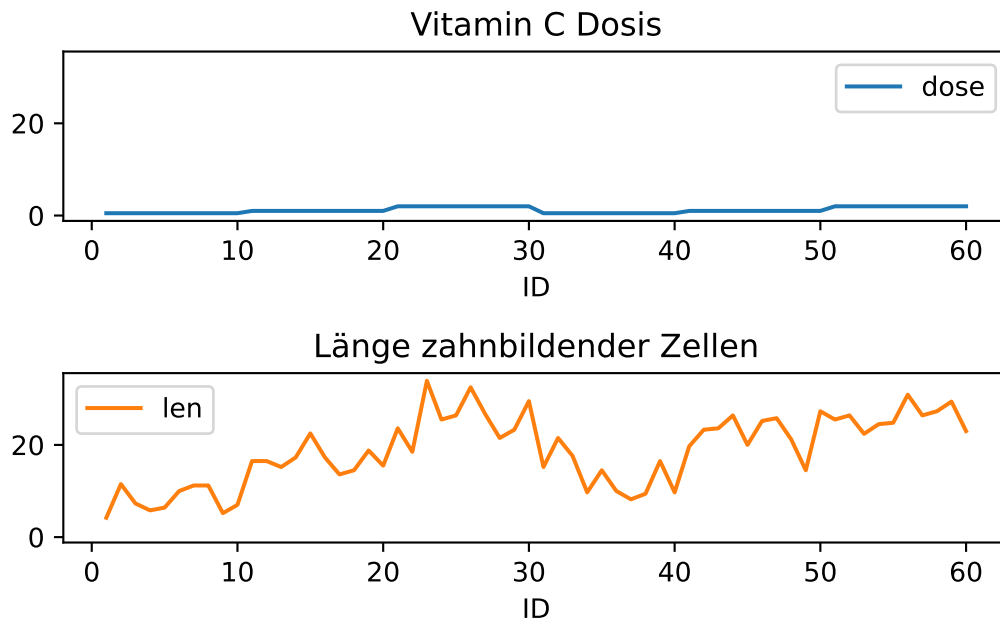
```
meerschweinchen.plot(x = 'ID', y = ['dose', 'len'], subplots = True, sharex = False, sharey = True)
```



Um die überdeckte Beschriftung der x-Achse zu beheben, muss auf den Befehl `plt.tight_layout()` aus dem Modul `matplotlib.pyplot` zurückgegriffen werden.

```
import matplotlib.pyplot as plt

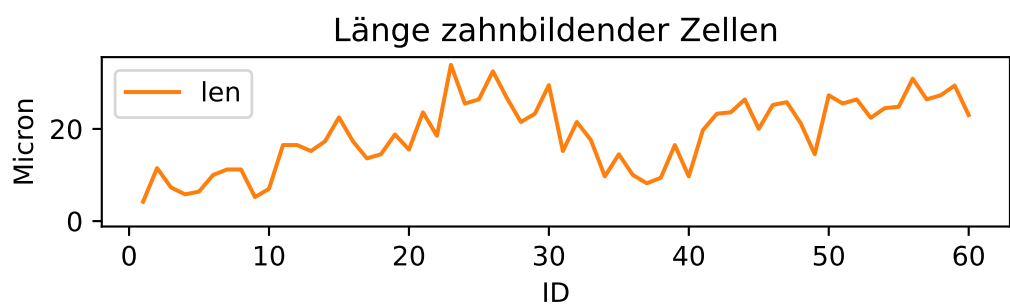
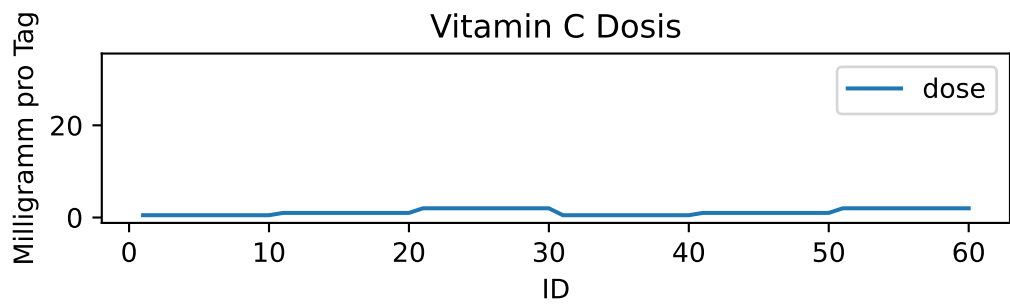
meerschweinchen.plot(x = 'ID', y = ['dose', 'len'], subplots = True, sharex = False, sharey = False)
plt.tight_layout()
```



Auch um für jede Teilgrafik eine y-Achsenbeschriftung zu setzen, muss auf matplotlib zurückgegriffen werden.

```
mein_plot = meerschweinchen.plot(x = 'ID', y = ['dose', 'len'], subplots = True, sharex = Fa

mein_plot[0].set_ylabel('Milligramm pro Tag')
mein_plot[1].set_ylabel('Micron')
plt.tight_layout()
plt.show()
```



4 Datentypen

Das Modul Pandas ist auf den Umgang mit unterschiedlichen Datentypen spezialisiert. Ein Dataframe kann unterschiedliche Datentypen enthalten (bspw. Zahlen und Wahrheitswerte).

NumPy unterstützt folgende Datentypen:

Datentyp NumPy-Array	Datentyp in Python
int_	int
double	float
cdouble	complex
bytes_	bytes
str_	str
bool_	bool
datetime64	datetime.datetime
timedelta64	datetime.timedelta

Die NumPy-Datentypen haben betriebssystemabhängige Synonyme. Beispielsweise wird für ein Array vom Datentyp `int_` der Datentyp `int` ausgegeben, für ein Array aus Gleitkommazahlen der Datentyp `float64`.

```
skalar = np.array([2])
print(skalar.dtype, "\n")

skalar = np.array([2.1])

print(skalar.dtype)
```

int64

float64

Außerdem gibt es für jeden NumPy-Datentyp ein Kürzel, das aus einem einzigen Buchstaben besteht. Beispielsweise wird für Zeichenfolgen das Kürzel U für Unicode-Zeichen und die Anzahl der Stellen ausgegeben (Für alle anderen Datentypen repräsentiert die Zahl die Anzahl der Bytes, die im Speicher benötigt werden.). Der Ausgabe ist ein Zeichen zur Kodierung der [Byte-Reihenfolge](#) im Speicher vorangestellt '>' (big-endian), '<' (little-endian) oder '=' (Systemstandard).

```
skalar = np.array(['2'])
print(skalar.dtype, "\n")

skalar = np.array(['2.1'])
print(skalar.dtype, "\n")

# Ein Datentyp mit mehr Speicherplatzbedarf kann zugewiesen werden
skalar = np.array([2], dtype = 'U3')
print(skalar.dtype)
```

<U1

<U3

<U3

Alle Synonyme und Kürzel können Sie der [NumPy-Dokumentation](#) entnehmen.

Häufig verwendet das Modul Pandas die NumPy-Datentypen. Pandas führt aber auch einige zusätzliche Datentypen ein. Eine vollständige Liste finden Sie in der [Pandas Dokumentation](#). Die wichtigsten zusätzlichen Datentypen sind:

- [Kategorie](#) dtype = 'category' für kategoriale, also ungeordnete, Daten.
- [Zeitzonebewusstes Datumsformat](#) dtype = 'datetime64[ns, US/Eastern]'
- Erweiterungen der NumPy-Datentypen mit Unterstützung fehlender Werte. Diese sind an der Großschreibung zu erkennen.

```
# NumPy-Datentyp int
series = pd.Series([1, 2, 3], dtype = 'int')
print(series, "\n")

# NumPy-Datentyp int unterstützt fehlende Werte nicht
try:
    series = pd.Series([1, 2, 3, np.nan], dtype = 'int')
```

```
except Exception as error:
    print(error, "\n")

# Pandas-Datentyp Int64 unterstützt fehlende Werte
series = pd.Series([1, 2, 3, np.nan], dtype = 'Int64')
print(series)
```

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

cannot convert float NaN to integer

```
0      1
1      2
2      3
3    <NA>
dtype: Int64
```

⚠ Warning 4: Pandas-Datentyp string

Pandas nutzt wie die Pythonbasis den Datentyp 'string', der unveränderlich (immutable) ist. Das bedeutet, es gibt keine Methode, die eine angelegte Zeichenkette verändern kann. Operationen mit diesem Datentyp geben ein neues Objekt mit dem Datentyp 'string' zurück.

Die Übergabe des Datentyps 'str' führt zur Verwendung des NumPy-Datentyps string (dtype = 'str'), der veränderlich (mutable) ist.

Je nach Situation kann die Verwendung des einen oder des anderen Datentyps nützlich sein. Beispielsweise kann der NumPy-Datentyp 'str' mit der Methode `pd.Series.sum()` verkettet werden.

```
# mit NumPy-Datentyp 'str'
string_series = pd.Series(['H', 'a', 'l', 'l', 'o', '!'], dtype = 'str')
print(f"Mit NumPy-Datentyp 'str': {string_series.sum()}")

# mit Pandas-Datentyp 'string'
try:
    string_series.astype('string').sum()
except Exception as error:
    print("\nMit Pandas-Datentyp 'string':")
    print(error)
```

Mit NumPy-Datentyp 'str': Hallo!

Mit Pandas-Datentyp 'string':
Cannot perform reduction 'sum' with string dtype

5 Zeitreihen

Die Verarbeitung von Datums- und Zeitinformationen wird in Python durch verschiedene Module ermöglicht. Einleitend werden einige dieser Module kurz vorgestellt, da in der Dokumentation gelegentlich auf diese verwiesen wird. Pandas bietet einen einheitlichen Zugang zu den meisten dieser Funktionen und verwendet die NumPy Datentypen `datetime64` und `timedelta64`.

- Der Datentyp `datetime64` beschreibt einen bestimmten Zeitpunkt an einem bestimmten Datum und gehört zu der Klasse `Timestamp`. Der Datentyp hat die Einheit Nanosekunden und kann Informationen über die Zeitzone speichern.
- Der Datentyp `timedelta64` beschreibt eine absolute Zeitdauer in der Einheit Nanosekunden und gehört zu der Klasse `Timedelta`.

5.1 Datums- und Zeitinformationen in Python

In Python gibt es einige Module zur Verarbeitung von Datums- und Zeitinformationen.

- Das Modul `time` stellt Zeit- und Datumsoperationen mit Objekten vom Typ `struct_time` bereit. ([Dokumentation des Moduls time](#))
- Das Modul `datetime` führt die Datentypen `datetime` und `timedelta`, zusätzliche Methoden für die Bearbeitung und die Ausgabe von Datums- und Zeitinformationen ein. Das Modul kann Jahreszahlen von 1 bis 9999 nach unserer Zeitrechnung im Gregorianischen Kalender verarbeiten. ([Dokumentation des Moduls datetime](#))
- Das Modul `calendar` führt verschiedene Kalenderfunktionen ein und erweitert den verarbeitbaren Zeitraum. Basierend auf dem Gregorianischen Kalender reicht dieser in beide Richtungen ins Unendliche. ([Dokumentation des Moduls calendar](#))
- Das Modul `pytz` führt die IANA-Zeitzonendatenbank (Internet Assigned Numbers Authority) für Anwendungsprogramme und Betriebssysteme ein (auch Olsen-Datenbank genannt). Die IANA-Datenbank beinhaltet die Zeitzonen und Änderungen der Zeit seit 1970. ([Wikipedia](#)) Das Modul `pytz` sorgt für eine korrekte Berechnung von Zeiten zum Ende der Zeitumstellung (Ende Sommerzeit) über Zeitzonen hinweg. ([Dokumentation pytz](#))

- NumPy führt die Datentypen `datetime64` und `timedelta64` ein. Diese basieren auf dem Gregorianischen Kalender und reichen in beide Richtungen ins Unendliche. <https://numpy.org/doc/stable/reference/arrays.datetime.html>
- Pandas nutzt die NumPy-Datentypen `datetime64` und `timedelta64` und ergänzt zahlreiche Funktionen zur Verarbeitung von Datums- und Zeitinformationen aus anderen Paketen. https://pandas.pydata.org/docs/user_guide/timeseries.html

NumPy und Pandas können Datetime-Objekte anderer Module in den Datentyp `datetime64` umwandeln.

5.1.1 Naive und bewusste Datetime-Objekte

Datetime-Objekte werden abhängig davon, ob sie Informationen über Zeitzonen enthalten, als *naiv* (naive) oder als *bewusst* (aware) bezeichnet. Naiven Datetime-Objekten fehlt diese Information, bewusste Datetime-Objekte enthalten diese. Objekte der Module `time`, `datetime` und Pandas verfügen über ein Zeitzonenattribut, sind also bewusst. `np.datetime64` ist seit NumPy-Version 1.11.0 ein naiver Datentyp, unterstützt aber Zeitzonen aus Gründen der Rückwärtskompatibilität.

“Deprecated since version 1.11.0: NumPy does not store timezone information. For backwards compatibility, `datetime64` still parses timezone offsets, which it handles by converting to UTC±00:00 (Zulu time). This behaviour is deprecated and will raise an error in the future.” [NumPy Dokumentation](#)

Zeitzonen

Pandas kann mit Zeitzonen umgehen und `datetime`-Objekte von einer in eine andere Zeitzone umwandeln. Über das Argument `tz` kann in verschiedenen Funktionen die Zeitzone angegeben werden.

```
zeitreihe = pd.Series(pd.date_range(start = "2023-03-26T00:00", end = "2023-03-27T00:00", fr
zeitreihe
```

```
0    2023-03-26 00:00:00+03:00
1    2023-03-26 03:00:00+03:00
2    2023-03-26 06:00:00+03:00
3    2023-03-26 09:00:00+03:00
4    2023-03-26 12:00:00+03:00
5    2023-03-26 15:00:00+03:00
6    2023-03-26 18:00:00+03:00
7    2023-03-26 21:00:00+03:00
```

```
8    2023-03-27 00:00:00+03:00
dtype: datetime64[ns, Turkey]
```

Mit der Funktion `pd.to_datetime(arg, utc = True)` kann die Zeitzone in die koordinierte Universalzeit UTC umgewandelt werden.

```
pd.to_datetime(zeitreihe, utc = True)
```

```
0    2023-03-25 21:00:00+00:00
1    2023-03-26 00:00:00+00:00
2    2023-03-26 03:00:00+00:00
3    2023-03-26 06:00:00+00:00
4    2023-03-26 09:00:00+00:00
5    2023-03-26 12:00:00+00:00
6    2023-03-26 15:00:00+00:00
7    2023-03-26 18:00:00+00:00
8    2023-03-26 21:00:00+00:00
dtype: datetime64[ns, UTC]
```

Eine Umwandlung in beliebige Zeitzonen ist mit der Methode `pd.Series.dt.tz_convert(tz = 'utc')` möglich.

```
zeitreihe.dt.tz_convert(tz = 'portugal')
```

```
0    2023-03-25 21:00:00+00:00
1    2023-03-26 00:00:00+00:00
2    2023-03-26 04:00:00+01:00
3    2023-03-26 07:00:00+01:00
4    2023-03-26 10:00:00+01:00
5    2023-03-26 13:00:00+01:00
6    2023-03-26 16:00:00+01:00
7    2023-03-26 19:00:00+01:00
8    2023-03-26 22:00:00+01:00
dtype: datetime64[ns, Portugal]
```

i Hinweis 2: verfügbare Zeitzonen ermitteln

Der folgende Code gibt die in Python verfügbaren Zeitzonen aus.

```
from zoneinfo import available_timezones

for timezone in sorted(available_timezones()):
    print(timezone)
```

```
Africa/Abidjan
Africa/Accra
Africa/Addis_Ababa
Africa/Algiers
Africa/Asmara
Africa/Asmera
Africa/Bamako
Africa/Bangui
Africa/Banjul
Africa/Bissau
Africa/Blantyre
Africa/Brazzaville
Africa/Bujumbura
Africa/Cairo
Africa/Casablanca
Africa/Ceuta
Africa/Conakry
Africa/Dakar
Africa/Dar_es_Salaam
Africa/Djibouti
Africa/Douala
Africa/El_Aaiun
Africa/Freetown
Africa/Gaborone
Africa/Harare
Africa/Johannesburg
Africa/Juba
Africa/Kampala
Africa/Khartoum
Africa/Kigali
Africa/Kinshasa
Africa/Lagos
Africa/Libreville
Africa/Lome
Africa/Luanda
Africa/Lubumbashi
```

Africa/Lusaka
Africa/Malabo
Africa/Maputo
Africa/Maseru
Africa/Mbabane
Africa/Mogadishu
Africa/Monrovia
Africa/Nairobi
Africa/Ndjamena
Africa/Niamey
Africa/Nouakchott
Africa/Ouagadougou
Africa/Porto-Novo
Africa/Sao_Tome
Africa/Timbuktu
Africa/Tripoli
Africa/Tunis
Africa/Windhoek
America/Adak
America/Anchorage
America/Anguilla
America/Antigua
America/Araguaina
America/Argentina/Buenos_Aires
America/Argentina/Catamarca
America/Argentina/ComodRivadavia
America/Argentina/Cordoba
America/Argentina/Jujuy
America/Argentina/La_Rioja
America/Argentina/Mendoza
America/Argentina/Rio_Gallegos
America/Argentina/Salta
America/Argentina/San_Juan
America/Argentina/San_Luis
America/Argentina/Tucuman
America/Argentina/Ushuaia
America/Aruba
America/Asuncion
America/Atikokan
America/Atka
America/Bahia

America/Bahia_Banderas
America/Barbados
America/Belem
America/Belize
America/Blanc-Sablon
America/Boa_Vista
America/Bogota
America/Boise
America/Buenos_Aires
America/Cambridge_Bay
America/Campo_Grande
America/Cancun
America/Caracas
America/Catamarca
America/Cayenne
America/Cayman
America/Chicago
America/Chihuahua
America/Ciudad_Juarez
America/Coral_Harbour
America/Cordoba
America/Costa_Rica
America/Coyhaique
America/Creston
America/Cuiaba
America/Curacao
America/Danmarkshavn
America/Dawson
America/Dawson_Creek
America/Denver
America/Detroit
America/Dominica
America/Edmonton
America/Eirunepe
America/El_Salvador
America/Ensenada
America/Fort_Nelson
America/Fort_Wayne
America/Fortaleza
America/Glace_Bay
America/Godthab

America/Goose_Bay
America/Grand_Turk
America/Grenada
America/Guadeloupe
America/Guatemala
America/Guayaquil
America/Guyana
America/Halifax
America/Havana
America/Hermosillo
America/Indiana/Indianapolis
America/Indiana/Knox
America/Indiana/Marengo
America/Indiana/Petersburg
America/Indiana/Tell_City
America/Indiana/Vevay
America/Indiana/Vincennes
America/Indiana/Winamac
America/Indianapolis
America/Inuvik
America/Iqaluit
America/Jamaica
America/Jujuy
America/Juneau
America/Kentucky/Louisville
America/Kentucky/Monticello
America/Knox_IN
America/Kralendijk
America/La_Paz
America/Lima
America/Los_Angeles
America/Louisville
America/Lower_Princes
America/Maceio
America/Managua
America/Manaus
America/Marigot
America/Martinique
America/Matamoros
America/Mazatlan
America/Mendoza

America/Menominee
America/Merida
America/Metlakatla
America/Mexico_City
America/Miquelon
America/Moncton
America/Monterrey
America/Montevideo
America/Montreal
America/Montserrat
America/Nassau
America/New_York
America/Nipigon
America/Nome
America/Noronha
America/North_Dakota/Beulah
America/North_Dakota/Center
America/North_Dakota/New_Salem
America/Nuuk
America/Ojinaga
America/Panama
America/Pangnirtung
America/Paramaribo
America/Phoenix
America/Port-au-Prince
America/Port_of_Spain
America/Porto_Acre
America/Porto_Velho
America/Puerto_Rico
America/Punta_Arenas
America/Rainy_River
America/Rankin_Inlet
America/Recife
America/Regina
America/Resolute
America/Rio_Branco
America/Rosario
America/Santa_Isabel
America/Santarem
America/Santiago
America/Santo_Domingo

America/Sao_Paulo
America/Scoresbysund
America/Shiprock
America/Sitka
America/St_Barthelemy
America/St_Johns
America/St_Kitts
America/St_Lucia
America/St_Thomas
America/St_Vincent
America/Swift_Current
America/Tegucigalpa
America/Thule
America/Thunder_Bay
America/Tijuana
America/Toronto
America/Tortola
America/Vancouver
America/Virgin
America/Whitehorse
America/Winnipeg
America/Yakutat
America/Yellowknife
Antarctica/Casey
Antarctica/Davis
Antarctica/DumontDUrville
Antarctica/Macquarie
Antarctica/Mawson
Antarctica/McMurdo
Antarctica/Palmer
Antarctica/Rothera
Antarctica/South_Pole
Antarctica/Syowa
Antarctica/Troll
Antarctica/Vostok
Arctic/Longyearbyen
Asia/Aden
Asia/Almaty
Asia/Amman
Asia/Anadyr
Asia/Aqtau

Asia/Aqtobe
Asia/Ashgabat
Asia/Ashkhabad
Asia/Atyrau
Asia/Baghdad
Asia/Bahrain
Asia/Baku
Asia/Bangkok
Asia/Barnaul
Asia/Beirut
Asia/Bishkek
Asia/Brunei
Asia/Calcutta
Asia/Chita
Asia/Choibalsan
Asia/Chongqing
Asia/Chungking
Asia/Colombo
Asia/Dacca
Asia/Damascus
Asia/Dhaka
Asia/Dili
Asia/Dubai
Asia/Dushanbe
Asia/Famagusta
Asia/Gaza
Asia/Harbin
Asia/Hebron
Asia/Ho_Chi_Minh
Asia/Hong_Kong
Asia/Hovd
Asia/Irkutsk
Asia/Istanbul
Asia/Jakarta
Asia/Jayapura
Asia/Jerusalem
Asia/Kabul
Asia/Kamchatka
Asia/Karachi
Asia/Kashgar
Asia/Kathmandu

Asia/Katmandu
Asia/Khandyga
Asia/Kolkata
Asia/Krasnoyarsk
Asia/Kuala_Lumpur
Asia/Kuching
Asia/Kuwait
Asia/Macao
Asia/Macau
Asia/Magadan
Asia/Makassar
Asia/Manila
Asia/Muscat
Asia/Nicosia
Asia/Novokuznetsk
Asia/Novosibirsk
Asia/Omsk
Asia/Oral
Asia/Phnom_Penh
Asia/Pontianak
Asia/Pyongyang
Asia/Qatar
Asia/Qostanay
Asia/Qyzylorda
Asia/Rangoon
Asia/Riyadh
Asia/Saigon
Asia/Sakhalin
Asia/Samarkand
Asia/Seoul
Asia/Shanghai
Asia/Singapore
Asia/Srednekolymsk
Asia/Taipei
Asia/Tashkent
Asia/Tbilisi
Asia/Tehran
Asia/Tel_Aviv
Asia/Thimbu
Asia/Thimphu
Asia/Tokyo

Asia/Tomsk
Asia/Ujung_Pandang
Asia/Ulaanbaatar
Asia/Ulan_Bator
Asia/Urumqi
Asia/Ust-Nera
Asia/Vientiane
Asia/Vladivostok
Asia/Yakutsk
Asia/Yangon
Asia/Yekaterinburg
Asia/Yerevan
Atlantic/Azores
Atlantic/Bermuda
Atlantic/Canary
Atlantic/Cape_Verde
Atlantic/Faeroe
Atlantic/Faroe
Atlantic/Jan_Mayen
Atlantic/Madeira
Atlantic/Reykjavik
Atlantic/South_Georgia
Atlantic/St_Helena
Atlantic/Stanley
Australia/ACT
Australia/Adelaide
Australia/Brisbane
Australia/Broken_Hill
Australia/Canberra
Australia/Currie
Australia/Darwin
Australia/Eucla
Australia/Hobart
Australia/LHI
Australia/Lindeman
Australia/Lord_Howe
Australia/Melbourne
Australia/NSW
Australia/North
Australia/Perth
Australia/Queensland

Australia/South
Australia/Sydney
Australia/Tasmania
Australia/Victoria
Australia/West
Australia/Yancowinna
Brazil/Acre
Brazil/DeNoronha
Brazil/East
Brazil/West
CET
CST6CDT
Canada/Atlantic
Canada/Central
Canada/Eastern
Canada/Mountain
Canada/Newfoundland
Canada/Pacific
Canada/Saskatchewan
Canada/Yukon
Chile/Continental
Chile/EasterIsland
Cuba
EET
EST
EST5EDT
Egypt
Eire
Etc/GMT
Etc/GMT+0
Etc/GMT+1
Etc/GMT+10
Etc/GMT+11
Etc/GMT+12
Etc/GMT+2
Etc/GMT+3
Etc/GMT+4
Etc/GMT+5
Etc/GMT+6
Etc/GMT+7
Etc/GMT+8

Etc/GMT+9
Etc/GMT-0
Etc/GMT-1
Etc/GMT-10
Etc/GMT-11
Etc/GMT-12
Etc/GMT-13
Etc/GMT-14
Etc/GMT-2
Etc/GMT-3
Etc/GMT-4
Etc/GMT-5
Etc/GMT-6
Etc/GMT-7
Etc/GMT-8
Etc/GMT-9
Etc/GMT0
Etc/Greenwich
Etc/UCT
Etc/UTC
Etc/Universal
Etc/Zulu
Europe/Amsterdam
Europe/Andorra
Europe/Astrakhan
Europe/Athens
Europe/Belfast
Europe/Belgrade
Europe/Berlin
Europe/Bratislava
Europe/Brussels
Europe/Bucharest
Europe/Budapest
Europe/Busingen
Europe/Chisinau
Europe/Copenhagen
Europe/Dublin
Europe/Gibraltar
Europe/Guernsey
Europe/Helsinki
Europe/Isle_of_Man

Europe/Istanbul
Europe/Jersey
Europe/Kaliningrad
Europe/Kiev
Europe/Kirov
Europe/Kyiv
Europe/Lisbon
Europe/Ljubljana
Europe/London
Europe/Luxembourg
Europe/Madrid
Europe/Malta
Europe/Mariehamn
Europe/Minsk
Europe/Monaco
Europe/Moscow
Europe/Nicosia
Europe/Oslo
Europe/Paris
Europe/Podgorica
Europe/Prague
Europe/Riga
Europe/Rome
Europe/Samara
Europe/San_Marino
Europe/Sarajevo
Europe/Saratov
Europe/Simferopol
Europe/Skopje
Europe/Sofia
Europe/Stockholm
Europe/Tallinn
Europe/Tirane
Europe/Tiraspol
Europe/Ulyanovsk
Europe/Uzhgorod
Europe/Vaduz
Europe/Vatican
Europe/Vienna
Europe/Vilnius
Europe/Volgograd

Europe/Warsaw
Europe/Zagreb
Europe/Zaporozhye
Europe/Zurich
Factory
GB
GB-Eire
GMT
GMT+0
GMT-0
GMT0
Greenwich
HST
Hongkong
Iceland
Indian/Antananarivo
Indian/Chagos
Indian/Christmas
Indian/Cocos
Indian/Comoro
Indian/Kerguelen
Indian/Mahe
Indian/Maldives
Indian/Mauritius
Indian/Mayotte
Indian/Reunion
Iran
Israel
Jamaica
Japan
Kwajalein
Libya
MET
MST
MST7MDT
Mexico/BajaNorte
Mexico/BajaSur
Mexico/General
NZ
NZ-CHAT
Navajo

PRC
PST8PDT
Pacific/Apia
Pacific/Auckland
Pacific/Bougainville
Pacific/Chatham
Pacific/Chuuk
Pacific/Easter
Pacific/Efate
Pacific/Enderbury
Pacific/Fakaofu
Pacific/Fiji
Pacific/Funafuti
Pacific/Galapagos
Pacific/Gambier
Pacific/Guadalcanal
Pacific/Guam
Pacific/Honolulu
Pacific/Johnston
Pacific/Kanton
Pacific/Kiritimati
Pacific/Kosrae
Pacific/Kwajalein
Pacific/Majuro
Pacific/Marquesas
Pacific/Midway
Pacific/Nauru
Pacific/Niue
Pacific/Norfolk
Pacific/Noumea
Pacific/Pago_Pago
Pacific/Palau
Pacific/Pitcairn
Pacific/Pohnpei
Pacific/Ponape
Pacific/Port_Moresby
Pacific/Rarotonga
Pacific/Saipan
Pacific/Samoa
Pacific/Tahiti
Pacific/Tarawa

```
Pacific/Tongatapu
Pacific/Truk
Pacific/Wake
Pacific/Wallis
Pacific/Yap
Poland
Portugal
ROC
ROK
Singapore
Turkey
UCT
US/Alaska
US/Aleutian
US/Arizona
US/Central
US/East-Indiana
US/Eastern
US/Hawaii
US/Indiana-Starke
US/Michigan
US/Mountain
US/Pacific
US/Samoa
UTC
Universal
W-SU
WET
Zulu
localtime
```

5.1.2 Alles ist relativ: die Epoche

Python speichert Zeit relativ zu einem zeitlichen Bezugspunkt, der Unix-Zeit, der sogenannten Epoche. Die Epoche kann mit der Funktion `pd.to_datetime(0)` ausgegeben werden. Die Funktion konvertiert Argumente in Zeitpunkte (Timestamp). Ganzzahlen werden dabei als Nanosekunden seit der Epoche interpretiert. Die Funktion werden wir später noch ausführlicher behandeln.

```
import pandas as pd
print(pd.to_datetime(0))
```

1970-01-01 00:00:00

⚠ Warning 5: Zeit - atomar, koordiniert oder universal?

NumPy nutzt die Internationale Atomzeit (abgekürzt TAI für französisch Temps Atomique International). Diese nimmt für jeden Kalendertag eine Länge von 86.400 Sekunden an, kennt also keine Schaltsekunde. Die Atomzeit bildet die Grundlage für die koordinierte Weltzeit UTC.

UTC steht für Coordinated Universal Time (auch bekannt als Greenwich Mean Time). Das Kürzel UTC ist ein Kompromiss für die englische und die französische Sprache. Die koordinierte Weltzeit gleicht die Verlangsamung der Erdrotation (astronomisch gemessen als Universalzeit, Universal Time UT) durch Schaltsekunden aus, um die geringfügige Verlängerung eines Tages auszugleichen. Die TAI geht deshalb gegenüber der UTC vor. Seit 1972 unterscheiden sich beide Zeiten um eine ganzzahlige Anzahl von Sekunden. Aktuell (2024) geht die TAI 37 Sekunden gegenüber UTC vor.

Eine Umwandlung in die koordinierte Weltzeit ist in NumPy bislang noch nicht umgesetzt. ([Dokumentation NumPy](#), [Wikipedia](#)).

5.1.3 Zeitumstellung - Daylight Saving Time

“DST is Daylight Saving Time, an adjustment of the timezone by (usually) one hour during part of the year. DST rules are magic (determined by local law) and can change from year to year. The C library has a table containing the local rules (often it is read from a system file for flexibility) and is the only source of True Wisdom in this respect.” ([Dokumentation time](#))

Pandas arbeitet standardmäßig mit der koordinierten Weltzeit UTC. Die UTC selbst ist keine Zeitzone und kennt deshalb keine Zeitumstellung. Die Zeitumstellung wird abhängig von der Zeitzone berücksichtigt. Beispielsweise wurde die Zeitumstellung in der Türkei 2016 abgeschafft (und die Sommerzeit dauerhaft eingeführt).

In den folgenden Beispielen wird am Tag vor der Zeitumstellung um 9 Uhr eine Zeitdifferenz von 24 Stunden addiert. Da über die Nacht (der Morgen des Folgetages) die Uhr um eine Stunde vorgestellt wird, zeigt der resultierende Zeitstempel die Uhrzeit 10 Uhr an, sofern die Zeitumstellung gilt.

```
print("Keine Zeitumstellung in UTC:")
print(pd.Timestamp("2025-03-29T09:00") + pd.Timedelta(24, "h"), "\n")

print("Zeitzone mit Zeitumstellung:")
```

```
print(pd.Timestamp("2025-03-29T09:00", tz="Europe/Berlin") + pd.Timedelta(24, "h"), "\n")

print("Heute keine Zeitumstellung in Türkei:")
print(pd.Timestamp("2025-03-29T09:00", tz="Turkey") + pd.Timedelta(24, "h"), "\n")

print("Türkei vor der Abschaffung der Zeitumstellung:")
print(pd.Timestamp("2014-03-30T09:00", tz="Turkey") + pd.Timedelta(24, "h"))
```

Keine Zeitumstellung in UTC:
2025-03-30 09:00:00

Zeitzone mit Zeitumstellung:
2025-03-30 10:00:00+02:00

Heute keine Zeitumstellung in Türkei:
2025-03-30 09:00:00+03:00

Türkei vor der Abschaffung der Zeitumstellung:
2014-03-31 10:00:00+03:00

Eine Liste der Zeitzonen finden Sie auf Wikipedia: https://en.wikipedia.org/wiki/List_of_tz_database_time_zones

5.1.4 Kalender

Die Module calendar, NumPy und Pandas verwenden den um die Zeit vor seiner Einführung 1582 erweiterten Gregorianische Kalender, den [proleptischen Gregorianischen Kalender](#). Während das Modul date nur die Jahre 1-9999 nach unserer Zeit unterstützt, erlaubt der Datentyp `datetime64` auch Jahre vor unserer Zeit in [astronomischer Jahresnumerierung](#). Das bedeutet, es gibt ein Jahr 0 (das erste Jahr vor unserer Zeit) und vorausgehende Jahre werden mit negativen Zahlen dargestellt (-1 ist das zweite Jahr vor unserer Zeit). [NumPy Dokumentation](#)

5.2 datetime in Pandas

Pandas nutzt den NumPy-Datentyp `datetime64`, um Datums- und Zeitinformationen zu verarbeiten. In Pandas werden `datetime64`-Objekte mit den Funktionen `pd.to_datetime()` oder `pd.date_range()` angelegt.

Hinweis: Eine weitere Möglichkeit ist die Funktion `pd.Timestamp()`, die umfangreichere Möglichkeiten zur Erzeugung eines Zeitpunkts bietet, aber kein string-parsing unterstützt.

`pd.to_datetime()` erzeugt Werte des Datentyps `datetime64[ns]` (mit `pd.to_datetime()` erzeugte Skalare (Einzelwerte) werden als Timestamp (Zeitpunkt) ausgegeben, die kein Attribut `dtype` haben). Die Funktion `pd.to_datetime()` akzeptiert als Eingabewerte:

- `datetime`-Objekte anderer Module.
- Zahlen und eine Zeiteinheit `pd.to_datetime(1, unit = None)` (Standard sind Nanosekunden). Das Argument `unit` nimmt die Werte `'ns'`, `'ms'`, `'s'`, `'m'`, `'h'`, `'D'`, `'W'`, `'M'`, `'Y'` für Nanosekunde, Millisekunde, Sekunde, Minute, Stunde, Tag, Woche, Monat bzw. Jahr entgegen. Erzeugt wird ein Zeitpunkt relativ zur Epoche.

```
print(pd.to_datetime(1000, unit = 'D'))
print(pd.to_datetime(1000 * 1000, unit = 'h'))
print(pd.to_datetime(1000 * 1000 * 1000, unit = 's'))
```

```
1972-09-27 00:00:00
2084-01-29 16:00:00
2001-09-09 01:46:40
```

- Zeichenketten, die ein Datum oder ein Datum mit Uhrzeit ausdrücken, formatiert nach [ISO 8601](#).

```
print(pd.to_datetime('2017'))
print(pd.to_datetime('2017-01-01T00'))
print(pd.to_datetime('2017-01-01 00:00:00'))
```

```
2017-01-01 00:00:00
2017-01-01 00:00:00
2017-01-01 00:00:00
```

- Anders formatierte Zeichenketten mit dem Argument `format = "%d/%m/%Y"` (siehe [Dokumentation `strptime` zur string-Formatierung](#)).

```
print(pd.to_datetime('Monday, 12. August `24', format = "%A, %d. %B `%y"))
print(pd.to_datetime('Monday, 12. August 2024, 12:15 Uhr CET', format = "%A, %d. %B %Y, %H:%M"))
```

```
2024-08-12 00:00:00
2024-08-12 12:15:00+02:00
```

- Dictionary oder `DataFrame`.

```
print(pd.to_datetime({'year':[2020, 2024], 'month': [1, 11], 'day': [1, 21]}), "\n")
print(pd.to_datetime(pd.DataFrame({'year':[2020, 2024], 'month': [1, 11], 'day': [1, 21]})))
```

```
0    2020-01-01
1    2024-11-21
dtype: datetime64[ns]
```

```
0    2020-01-01
1    2024-11-21
dtype: datetime64[ns]
```

Die Funktion `pd.date_range()` erzeugt ein Array vom Typ `DatetimeIndex` mit dtype `datetime64`. Genau drei der folgenden vier Argumente sind für die Erzeugung erforderlich:

- **start:** Beginn der Reihe.
- **end:** Ende der Reihe (inklusive)
- **freq:** Schrittweite (bspw. Jahr, Tag, Geschäftstag, Stunde oder Vielfache wie '6h' - siehe [Liste verfügbarer strings](#))
- **periods:** Anzahl der zu erzeugenden Werte.

```
print(pd.date_range(start = '2017', end = '2024', periods = 3), "\n")
print(pd.date_range(start = '2017', end = '2024', freq = 'Y'), "\n")
print(pd.date_range(end = '2024', freq = 'h', periods = 3))
```

```
DatetimeIndex(['2017-01-01', '2020-07-02', '2024-01-01'], dtype='datetime64[ns]', freq=None)
```

```
DatetimeIndex(['2017-12-31', '2018-12-31', '2019-12-31', '2020-12-31',
               '2021-12-31', '2022-12-31', '2023-12-31'],
              dtype='datetime64[ns]', freq='YE-DEC')
```

```
DatetimeIndex(['2023-12-31 22:00:00', '2023-12-31 23:00:00',
               '2024-01-01 00:00:00'],
              dtype='datetime64[ns]', freq='h')
```

Warning 6: pd.date_range()

Die Funktion `pd.date_range()` wird künftig das Kürzel 'Y' nicht mehr unterstützen. Stattdessen können die Kürzel 'YS' (Jahresbeginn) oder 'YE' (Jahresende) verwendet werden. Ebenso wird das Kürzel 'M' künftig durch 'MS' (Monatsstart), 'ME' (Monatsende) ersetzt.

5.3 timedelta in Pandas

Zeitdifferenzen werden mit der Funktion `pd.Timedelta()` erzeugt. Zeitdifferenzen können zum einen durch Angabe einer Ganzzahl und einer Zeiteinheit angelegt werden. Außerdem ist die Übergabe mit Argumenten möglich (zulässige Argumente sind: weeks, days, hours, minutes, seconds, milliseconds, microseconds, nanoseconds).

```
print(pd.Timedelta(1, 'D'))
print(pd.Timedelta(days = 1, hours = 1))
```

```
1 days 00:00:00
1 days 01:00:00
```

Wichtig: Anders als in NumPy werden Zeitdifferenzen in Monaten und Jahren nicht mehr von Pandas unterstützt.

```
try:
    print(pd.Timedelta(1, 'Y'))
except ValueError as error:
    print(error)
else:
    print(pd.Timedelta(1, 'Y'))
```

Units 'M', 'Y', and 'y' are no longer supported, as they do not represent unambiguous timedeltas.

Zum anderen können Zeitdifferenzen mit einer Zeichenkette erzeugt werden.

```
print(pd.Timedelta('10sec'))
print(pd.Timedelta('10min'))
print(pd.Timedelta('10hours'))
print(pd.Timedelta('10days'))
print(pd.Timedelta('10w'))
```

```
0 days 00:00:10
0 days 00:10:00
0 days 10:00:00
10 days 00:00:00
70 days 00:00:00
```

Mit Hilfe einer Zeitdifferenz können Zeitreihen leicht verschoben werden.

```
pd.date_range(start = '2024-01-01T00:00', end = '2024-01-01T02:00', freq = '15min') + pd.Timedelta('10min')
```

```
DatetimeIndex(['2024-01-01 00:30:00', '2024-01-01 00:45:00',
               '2024-01-01 01:00:00', '2024-01-01 01:15:00',
               '2024-01-01 01:30:00', '2024-01-01 01:45:00',
               '2024-01-01 02:00:00', '2024-01-01 02:15:00',
               '2024-01-01 02:30:00'],
              dtype='datetime64[ns]', freq='15min')
```

5.4 Zugriff auf Zeitreihen

Pandas bietet zahlreiche Attribute und Methoden, um Informationen aus `datetime64`-Objekten auszulesen. Eine Übersicht aller verfügbaren Attribute und Methoden liefert `dir(pd.to_datetime(0))` bzw. der im folgenden Beispiel gezeigte Code.

```
# Attribute
print("Jahr:", pd.to_datetime(0).year)
print("Monat:", pd.to_datetime(0).month)
print("Tag:", pd.to_datetime(0).day)
print("Stunde:", pd.to_datetime(0).hour)
print("Minute:", pd.to_datetime(0).minute)
print("Sekunde:", pd.to_datetime(0).second)
print("Tag des Jahres:", pd.to_datetime(0).dayofyear)
print("Wochentag:", pd.to_datetime(0).dayofweek)
print("Tage im Monat:", pd.to_datetime(0).days_in_month)
print("Schaltjahr:", pd.to_datetime(0).is_leap_year)

# Methoden
print("\nDatum:", pd.to_datetime(0).date())
print("Zeit:", pd.to_datetime(0).time())
print("Wochentag (0-6):", pd.to_datetime(0).weekday())
print("Monatsname:", pd.to_datetime(0).month_name())
```

```
Jahr: 1970
Monat: 1
Tag: 1
Stunde: 0
Minute: 0
Sekunde: 0
Tag des Jahres: 1
Wochentag: 3
Tage im Monat: 31
Schaltjahr: False
```

Datum: 1970-01-01
Zeit: 00:00:00
Wochentag (0-6): 3
Monatsname: January

Hinweis 3: Attribute und Methoden eines datetime-Objekts

[illegible]

Für `pd.Series` erfolgt der Zugriff über den `.dt`-Operator (siehe [.dt accessor](#)). Der Zugriff auf verschiedene Informationen über ein Attribut (ohne Klammern) oder über eine Methode (mit Klammern) unterscheidet sich jedoch teilweise (siehe folgendes Beispiel).

Der dt-Operator

```
# Attribute
print("Datum:", pd.Series(pd.to_datetime(0)).dt.date) # Unterschied
print("Zeit:", pd.Series(pd.to_datetime(0)).dt.time) # Unterschied
print("Jahr", pd.Series(pd.to_datetime(0)).dt.year)
print("Monat", pd.Series(pd.to_datetime(0)).dt.month)
print("Tag", pd.Series(pd.to_datetime(0)).dt.day)
print("Stunde", pd.Series(pd.to_datetime(0)).dt.hour)
print("Minute", pd.Series(pd.to_datetime(0)).dt.minute)
print("Sekunde", pd.Series(pd.to_datetime(0)).dt.second)

print("\nTag des Jahres", pd.Series(pd.to_datetime(0)).dt.dayofyear)
print("Wochentag:", pd.Series(pd.to_datetime(0)).dt.dayofweek)
print("Wochentag:", pd.Series(pd.to_datetime(0)).dt.weekday) # Unterschied
print("Tage im Monat:", pd.Series(pd.to_datetime(0)).dt.days_in_month)
print("Schaltjahr:", pd.Series(pd.to_datetime(0)).dt.is_leap_year)

# Methoden
print("\nName des Monats:", pd.Series(pd.to_datetime(0)).dt.month_name())
```

```
Datum: 0    1970-01-01
dtype: object
Zeit: 0    00:00:00
dtype: object
Jahr 0    1970
dtype: int32
Monat 0    1
dtype: int32
Tag 0    1
dtype: int32
Stunde 0    0
dtype: int32
Minute 0    0
dtype: int32
Sekunde 0    0
dtype: int32
```

```
Tag des Jahres 0    1
dtype: int32
Wochentag: 0    3
dtype: int32
```

```
Wochentag: 0      3
dtype: int32
Tage im Monat: 0    31
dtype: int32
Schaltjahr: 0      False
dtype: bool

Name des Monats: 0      January
dtype: object
```

5.5 Aufgaben

1. Wie alt sind Sie in Tagen? Wie alt in Sekunden?
2. An welchem Wochentag war ihr Geburtstag?
3. Wie viele Tage sind es noch bis Weihnachten?
4. Erstellen Sie eine Liste aller Schaltjahre im 20. Jahrhundert.

Tipp 11: Musterlösung

Aufgabe 1

Ersetzen sie in der Lösung die Zeichenkette 'YYYY-MM-DD' bzw., wenn Sie die Uhrzeit Ihrer Geburt kennen, die Zeichenkette 'YYYY-MM-DDTHH:MM' durch Ihren Geburtstag. In Pandas werden die Schlüsselwörter `pd.to_datetime('today')` und `pd.to_datetime('now')` in Nanosekunden aufgelöst.

```
print((pd.to_datetime('today') - pd.to_datetime('YYYY-MM-DD')).days)
print(pd.to_datetime('now') - pd.to_datetime('YYYY-MM-DDTHH:MM')).total_seconds())
```

Aufgabe 2

```
print(pd.to_datetime('YYYY-MM-DD').day_of_week)
```

Aufgabe 3

```
(pd.to_datetime('2025-12-25') - pd.to_datetime('now')).days
```

Aufgabe 4

```
schaltjahre = pd.date_range(start = '1901', end = '2000', freq = 'YE')
schaltjahre = schaltjahre[schaltjahre.is_leap_year]
print(schaltjahre.year)
```

```
Index([1904, 1908, 1912, 1916, 1920, 1924, 1928, 1932, 1936, 1940, 1944, 1948,  
       1952, 1956, 1960, 1964, 1968, 1972, 1976, 1980, 1984, 1988, 1992, 1996],  
      dtype='int32')
```

6 Dateien lesen und schreiben

Pandas bietet eine Reihe von Funktionen, um Dateien einzulesen und zu schreiben, deren Namensgebung einem einheitlichen Schema folgt. Funktionen zum Lesen von Dateien werden in der Form `pd.read_csv()` und Funktionen zum Schreiben in der Form `pd.to_csv()` aufgerufen. Mit Pandas können auch Dateien aus dem Internet abgerufen werden `pd.read_csv(URL)`.

Format Type	Data Description	Reader	Writer
text	CSV	<code>read_csv</code>	<code>to_csv</code>
text	Fixed-Width Text File	<code>read_fwf</code>	NA
text	JSON	<code>read_json</code>	<code>to_json</code>
text	HTML	<code>read_html</code>	<code>to_html</code>
text	LaTeX	<code>Styler.to_latex</code>	NA
text	XML	<code>read_xml</code>	<code>to_xml</code>
text	Local clipboard	<code>read_clipboard</code>	<code>to_clipboard</code>
binary	MS Excel	<code>read_excel</code>	<code>to_excel</code>
binary	OpenDocument	<code>read_excel</code>	NA
binary	HDF5 Format	<code>read_hdf</code>	<code>to_hdf</code>
binary	Feather Format	<code>read_feather</code>	<code>to_feather</code>
binary	Parquet Format	<code>read_parquet</code>	<code>to_parquet</code>
binary	ORC Format	<code>read_orc</code>	<code>to_orc</code>
binary	Stata	<code>read_stata</code>	<code>to_stata</code>
binary	SAS	<code>read_sas</code>	NA
binary	SPSS	<code>read_spss</code>	NA
binary	Python Pickle Format	<code>read_pickle</code>	<code>to_pickle</code>
SQL	SQL	<code>read_sql</code>	<code>to_sql</code>

([Pandas Dokumentation](#))

Im Folgenden wird der Datensatz `palmerpenguins` mit Pandas eingelesen.

`palmerpenguins`

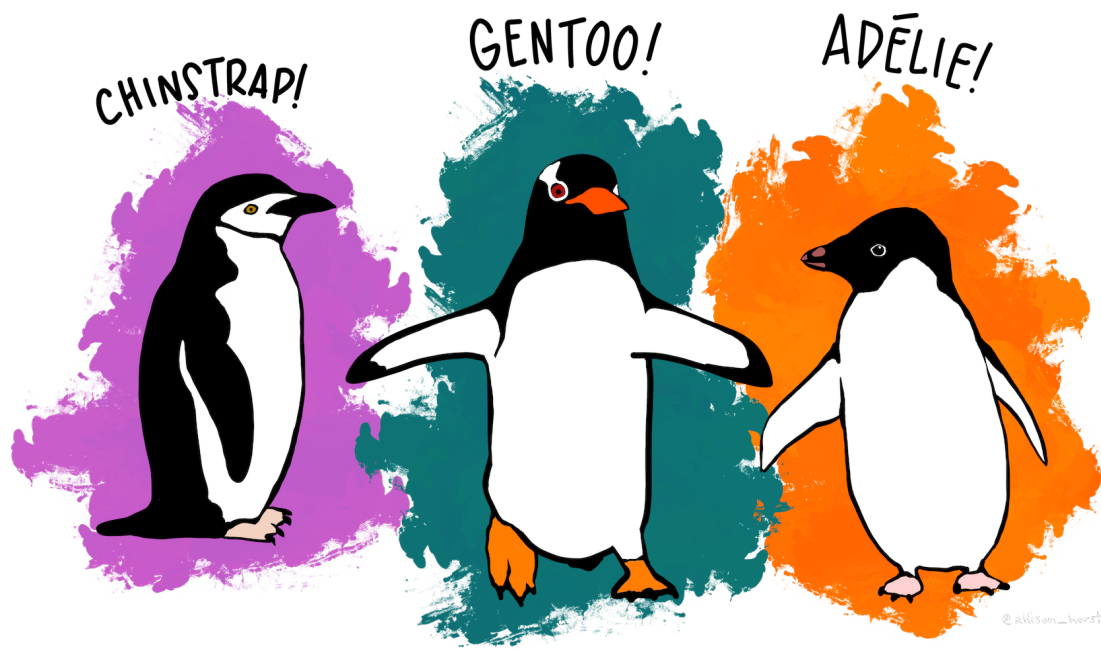


Abbildung 6.1: Pinguine des Palmer-Station-Datensatzes

Meet the Palmer penguins von @allison_horst steht unter der Lizenz [CC0-1.0](#) und ist auf [GitHub](#) abrufbar. 2020

Der Datensatz steht unter der Lizenz [CCO](#) und ist in R sowie auf [GitHub](#) verfügbar. 2020

```
# R Befehle, um den Datensatz zu laden
install.packages("palmerpenguins")
library(palmerpenguins)
```

Horst AM, Hill AP und Gorman KB. 2020. palmerpenguins: Palmer Archipelago (Antarctica) penguin data. R package version 0.1.0. <https://allisonhorst.github.io/palmerpenguins/>. doi: 10.5281/zenodo.3960218.

Die Funktionen zum Lesen von Dateien erwarten eine Pfadangabe, die positional oder mit einem Schlüsselwort übergeben werden kann. Das Schlüsselwort für die Pfadangabe variiert abhängig vom Dateityp und lautet für eine kommaseparierte CSV-Datei `filepath_or_buffer`.

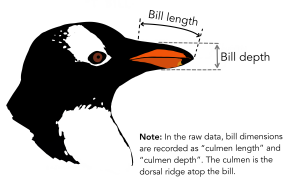
```
penguins = pd.read_csv(filepath_or_buffer = '01-daten/penguins.csv')
```

Ein Blick auf die Daten mit der Methode `penguins.head()`:

```
print(penguins.head())
```

	species	island	bill_length_mm	bill_depth_mm	flipper_length_mm	\
0	Adelie	Torgersen	39.1	18.7	181.0	
1	Adelie	Torgersen	39.5	17.4	186.0	
2	Adelie	Torgersen	40.3	18.0	195.0	
3	Adelie	Torgersen	NaN	NaN	NaN	
4	Adelie	Torgersen	36.7	19.3	193.0	

	body_mass_g	sex	year
0	3750.0	male	2007
1	3800.0	female	2007
2	3250.0	female	2007
3	NaN	NaN	2007
4	3450.0	female	2007



{fig-alt:“illustrative Grafik des Pinguinschnabels”}

Bill dimensions von @allison__horst steht unter der Lizenz [CC0-1.0](#) und ist auf [GitHub](#) abrufbar. 2020

Einen Überblick über den Datensatz verschafft die Methode `DataFrame.info()`.

```
print(penguins.info())
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 344 entries, 0 to 343
Data columns (total 8 columns):
#   Column              Non-Null Count  Dtype
---  -
0   species             344 non-null   object
```

```

1  island          344 non-null  object
2  bill_length_mm  342 non-null  float64
3  bill_depth_mm   342 non-null  float64
4  flipper_length_mm 342 non-null  float64
5  body_mass_g     342 non-null  float64
6  sex             333 non-null  object
7  year            344 non-null  int64
dtypes: float64(4), int64(1), object(3)
memory usage: 21.6+ KB
None

```

Einige Datentypen wurden nicht erkannt. Den betreffenden Spalten wurde der Sammeltyp `object` zugeordnet. Den Funktionen zum Einlesen von Daten kann mit dem Argument `dtype` der Datentyp übergeben werden. Für mehrere Spalten ist dies in Form eines Dictionaries in der Form `{'Spaltenname': 'dtype'}` möglich. Mit der Methode `DataFrame.astype()` ist dies auch nachträglich möglich.

```

penguins = pd.read_csv(filepath_or_buffer = '01-daten/penguins.csv', dtype = {'species': 'category'})

# nachträglich
# penguins = penguins.astype({'species': 'category', 'island': 'category', 'sex': 'category'})

print(penguins.info())

```

```

<class 'pandas.core.frame.DataFrame'>
RangeIndex: 344 entries, 0 to 343
Data columns (total 8 columns):
#   Column          Non-Null Count  Dtype
---  -
0   species         344 non-null   category
1   island          344 non-null   category
2   bill_length_mm  342 non-null   float64
3   bill_depth_mm   342 non-null   float64
4   flipper_length_mm 342 non-null   float64
5   body_mass_g     342 non-null   float64
6   sex             333 non-null   category
7   year            344 non-null   int64
dtypes: category(3), float64(4), int64(1)
memory usage: 15.0 KB
None

```

Einige Spalten weisen ungültige Werte auf. Die Tiere mit unvollständigen Werten sollen aus dem Datensatz entfernt werden.

- Mit der Methode `DataFrame.apply(pd.isna)` werden fehlende Werte bestimmt.
- Mit der Methode `DataFrame.any(axis = 1)` wird das Ergebnis zeilenweise aggregiert. `any` gibt `True` zurück, wenn mindestens ein Element `True` ist.
- Mit der Methode `sum()` wird die Anzahl der Zeilen mit fehlenden Werten bestimmt.
- Mit `np.where()` wird deren Indexposition bestimmt.
- Mit der Methode `DataFrame.drop()` werden die betreffenden Zeilen entfernt.

```
# Fehlende Werte bestimmen
print(penguins.apply(pd.isna).head(), "\n")

# zeilenweise aggregieren
print(penguins.apply(pd.isna).any(axis = 1).head(), "\n")

# Anzahl der Zeilen mit fehlenden Werten
print(f"Für {penguins.apply(pd.isna).any(axis = 1).sum()} Pinguine liegen unvollständige Werte vor")

# Indexpositionen bestimmen
print(np.where(penguins.apply(pd.isna).any(axis = 1))[0])

# Zeilen entfernen
penguins.drop(np.where(penguins.apply(pd.isna).any(axis = 1))[0], inplace = True)
```

	species	island	bill_length_mm	bill_depth_mm	flipper_length_mm	\
0	False	False	False	False	False	
1	False	False	False	False	False	
2	False	False	False	False	False	
3	False	False	True	True	True	
4	False	False	False	False	False	

	body_mass_g	sex	year
0	False	False	False
1	False	False	False
2	False	False	False
3	True	True	False
4	False	False	False

```
0    False
1    False
2    False
3     True
4    False
dtype: bool
```

Für 11 Pinguine liegen unvollständige Werte vor.

```
[ 3   8   9  10  11  47 178 218 256 268 271]
```

Kontrolle:

```
print(penguins.info())
```

```
<class 'pandas.core.frame.DataFrame'>
Index: 333 entries, 0 to 343
Data columns (total 8 columns):
 #   Column                Non-Null Count  Dtype  
---  -
 0   species               333 non-null   category
 1   island                333 non-null   category
 2   bill_length_mm        333 non-null   float64
 3   bill_depth_mm         333 non-null   float64
 4   flipper_length_mm     333 non-null   float64
 5   body_mass_g           333 non-null   float64
 6   sex                   333 non-null   category
 7   year                  333 non-null   int64   
dtypes: category(3), float64(4), int64(1)
memory usage: 17.0 KB
None
```

6.1 Zeitreihen einlesen

Mit Pandas ist es leicht möglich, Zeitreihen einzulesen. Durch string parsing können beliebige Zeichenketten als datetime interpretiert werden.

Wenn der innere Aufbau einer Datei bekannt ist, können die notwendigen Parameter direkt beim Einlesen beispielsweise mit `pd.read_csv()` übergeben werden. Dazu werden die Parameter `parse_dates` und `date_format` verwendet.

`parse_dates` gibt an, an welcher Stelle sich datetime-Informationen befinden. Es können verschiedene Argumente übergeben werden.

- `parse_dates = True` bewirkt, dass der Index als datetime interpretiert wird.
- Eine Liste von Ganzzahlen oder Spaltenbeschriftungen bewirkt, dass diese Spalten jeweils als eigene Spalte in datetime übersetzt werden, bspw `parse_dates = [1, 2, 3]`.

- Eine von einer Liste umschlossene Liste bewirkt, dass die übergebenen Spalten in einer einzigen Spalte zusammengeführt werden, bspw. `parse_dates = [[1, 2, 3]]`. Die Werte der Spalten werden mit einem Leerzeichen getrennt und anschließend interpretiert.

Pandas interpretiert die Zeichenketten nach [ISO 8601](#) als Repräsentation eines Datums in der festgelegten Reihenfolge Jahr, Monat, Tag, Stunde, Minute, Sekunde, Millisekunde im Format `YYYY-MM-DD 12:00:00.000`. Als Zeichentrenner zwischen Datum und Uhrzeit sind ein Leerzeichen oder der Buchstabe T zulässig. Der Datentyp und die kleinste verwendete Einheit werden im Attribut `dtype` gespeichert.

Andere Formate werden mit dem Parameter `date_format` spezifiziert. Mit Hilfe der [strftime-Dokumentation](#) kann das Datumsformat übergeben werden.

Datumsinformationen können aber auch nachträglich als solche deklariert werden. Dafür wird die Funktion `pd.to_datetime(arg, format = " ... ")` verwendet. Mit dem Parameter `arg` wird die zu konvertierende Spalte übergeben. Mit dem Parameter `format` kann wie mit dem Parameter `date_format` ein von der ISO8601 abweichendes Datumsformat spezifiziert werden.

6.2 Aufgaben Zeitreihen einlesen

Unter dem Pfad '01-daten/Microsoft_Stock.csv' sind Kursdaten der Microsoft-Aktie gespeichert.

```
stock = pd.read_csv(filepath_or_buffer = '01-daten/Microsoft_Stock.csv')

print(stock.head(), "\n")
print(stock.info())
```

	Date	Open	High	Low	Close	Volume
0	4/1/2015 16:00:00	40.60	40.76	40.31	40.72	36865322
1	4/2/2015 16:00:00	40.66	40.74	40.12	40.29	37487476
2	4/6/2015 16:00:00	40.34	41.78	40.18	41.55	39223692
3	4/7/2015 16:00:00	41.61	41.91	41.31	41.53	28809375
4	4/8/2015 16:00:00	41.48	41.69	41.04	41.42	24753438

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1511 entries, 0 to 1510
Data columns (total 6 columns):
#   Column   Non-Null Count  Dtype
---  ---
0   Date     1511 non-null   object
```

```

1   Open      1511 non-null   float64
2   High      1511 non-null   float64
3   Low       1511 non-null   float64
4   Close     1511 non-null   float64
5   Volume    1511 non-null   int64
dtypes: float64(4), int64(1), object(1)
memory usage: 71.0+ KB
None

```

Microsoft Stock- Time Series Analysis von Vijay V Venkitesh steht unter der Lizenz [CC0](#) und ist auf [kaggle](#) abrufbar. 2021

In der Spalte 'Date' sind Datums- und Zeitinformationen in der Form 'Monat/Tag/Jahr Stunde:Minute:Sekunde' verzeichnet, die von Pandas nicht automatisch erkannt wurden. Die Spalte hat deshalb den Datentyp object erhalten.

1. Übergeben Sie der Funktion `pd.read_csv()` die erforderlichen Argumente, um die Spalte Date korrekt als datetime einzulesen.
2. Berechnen Sie die Höchstkurse für jede Woche (intraday).

Tipp 12: Musterlösung Zeitreihen einlesen

1. Aufgabe

```

stock = pd.read_csv(filepath_or_buffer = '01-daten/Microsoft_Stock.csv',
                    parse_dates = ['Date'], # alternativ: [0]
                    date_format = '%m/%d/%Y %H:%M:%S')

print(stock.head(), "\n")
print(stock.info())

```

	Date	Open	High	Low	Close	Volume
0	2015-04-01 16:00:00	40.60	40.76	40.31	40.72	36865322
1	2015-04-02 16:00:00	40.66	40.74	40.12	40.29	37487476
2	2015-04-06 16:00:00	40.34	41.78	40.18	41.55	39223692
3	2015-04-07 16:00:00	41.61	41.91	41.31	41.53	28809375
4	2015-04-08 16:00:00	41.48	41.69	41.04	41.42	24753438

```

<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1511 entries, 0 to 1510
Data columns (total 6 columns):

```

```

#   Column  Non-Null Count  Dtype
---  -
0   Date    1511 non-null    datetime64[ns]
1   Open    1511 non-null    float64
2   High    1511 non-null    float64
3   Low     1511 non-null    float64
4   Close   1511 non-null    float64
5   Volume  1511 non-null    int64
dtypes: datetime64[ns](1), float64(4), int64(1)
memory usage: 71.0 KB
None

```

2. Aufgabe

Die Pandas-Methode `Series.dt.weekofyear()` wird seit einiger Zeit nicht mehr unterstützt ([siehe Dokumentation](#)). Die Funktion wurde durch `Series.dt.isocalendar().week` ersetzt.

```

# Jahr und Woche isolieren
print(stock['Date'].dt.isocalendar().week.head(), "\n")
print(stock['Date'].dt.isocalendar().year.tail())

# Jahr und Woche in den DataFrame einfügen
stock.insert(loc = 1, column = 'week', value = stock['Date'].dt.isocalendar().week)
stock.insert(loc = 1, column = 'year', value = stock['Date'].dt.isocalendar().year)

# Maximum für jede Woche mit groupby bestimmen
print(stock.groupby(by = ['year', 'week'])['High'].max())

# grafisch darstellen
stock.groupby(by = ['year', 'week'])['High'].max().plot(ylabel = 'Wochenhöchstkurs (intraday)')

```

```

0      14
1      14
2      15
3      15
4      15
Name: week, dtype: UInt32

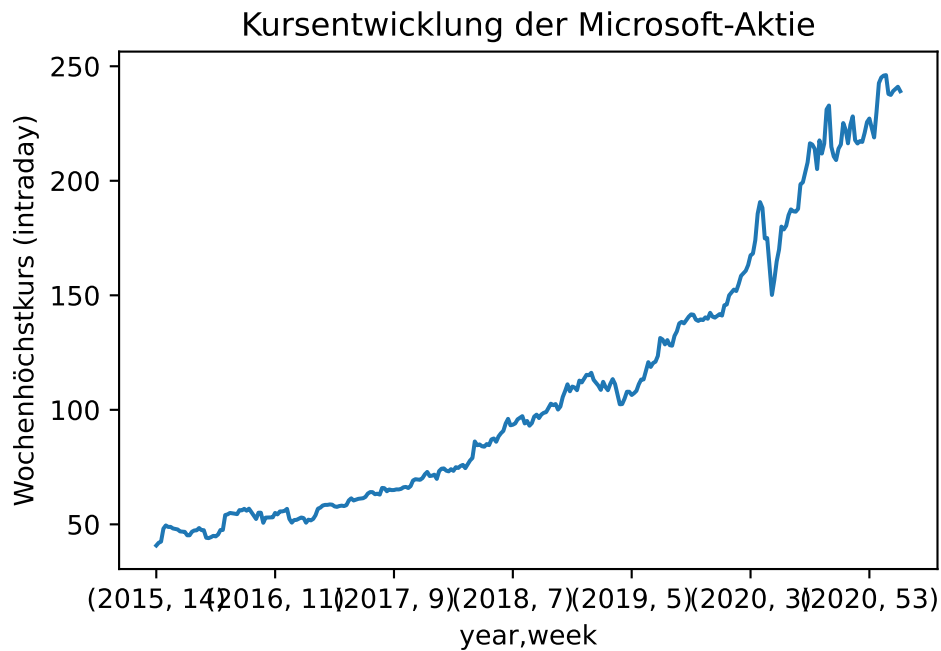
1506      2021
1507      2021
1508      2021

```

```

1509     2021
1510     2021
Name: year, dtype: UInt32
year  week
2015   14      40.76
      15      41.95
      16      42.46
      17      48.14
      18      49.54
      ...
2021    9      237.47
      10      239.17
      11      240.06
      12      241.05
      13      239.10
Name: High, Length: 314, dtype: float64

```



6.3 Schwierige Dateien einlesen

Das Einlesen von Dateien ist nicht immer einfach. Werkzeuge und Strategien zur Bewältigung schwieriger Fälle finden Sie im [Methodenbaustein Einlesen strukturierter Datensätze](#). Dort

wird auch der Umgang mit fehlenden Werten ausführlich behandelt.