Series & Dataframes

by Woche 6

Pandas stellt uns zwei wichtige Datenstrukturen zur Verfügung: Series und Dataframes. Series sind dabei eher wie Spalten in einer Tabelle zu verstehen, während Dataframes die gesamte Tabelle repräsentieren.

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

Series

Eine Series ist eine eindimensionale Datensammlungsstruktur. Nachdem wir bereits eine Vielzahl von Datensammlungsstrukturen wie Listen, Tuples, Sets, Dictionaries und numpy Arrays kennengelernt haben, stellt sich nun die berechtigte Frage:

Warum brauchen wir überhaupt eine weitere eindimensionale Datenstruktur wie die pandas Series?

Jede dieser Strukturen bringt ihre eigenen Vorteile und Einschränkungen mit sich. Listen und Tuples bieten Flexibilität durch ihre Fähigkeit, heterogene Daten zu speichern, wobei Tuples unveränderlich sind. Sets hingegen sind ideal für die Verwaltung einzigartiger Elemente, während Dictionaries effizienten Zugriff auf Werte über Schlüssel ermöglichen. Alle diese Strukturen haben gemeinsam, dass sie in der Standardbibliothek von Python verfügbar sind und grundlegende Funktionen für die Datenspeicherung und -manipulation bieten.

Mit der Einführung von numpy haben wir einen tiefen Einblick in die Handhabung homogener Datensätze erhalten, insbesondere in Bezug auf numerische Berechnungen. numpy Arrays revolutionieren die Art und Weise, wie wir mit großen Datenmengen umgehen, durch ihre Effizienz und die Möglichkeit, komplexe mathematische Operationen mit Leichtigkeit auszuführen. Diese Arrays sind allerdings primär auf numerische Daten ausgerichtet und bieten nicht dieselbe Flexibilität wie die eingebauten Python-Sammlungen, wenn es um die Arbeit mit tabellarischen Daten geht, die sowohl numerische als auch nicht-numerische Werte enthalten können.

Hier setzt pandas mit seinen Series und DataFrames an, die speziell für die Analyse und Manipulation von realen Daten konzipiert wurden. Eine pandas Series ist mehr als nur eine weitere eindimensionale Datenstruktur; sie ist eine natürliche Erweiterung eines numpy Arrays, das nicht nur für numerische Daten, sondern für Daten jeglicher Art – einschließlich Text und Zeiten – optimiert ist. Jedes Element in einer Series kann mit

einem Index versehen werden, der weit über die einfache numerische Indizierung hinausgeht, was die Arbeit mit Daten wesentlich intuitiver und flexibler macht. Darüber hinaus sind Series in der Lage, fehlende Daten auf eine Weise zu handhaben, die in reinen numpy Arrays nicht direkt verfügbar ist.

Wie schon bei den numpy Arrays für Zahlen ist ein signifikanter Vorteil der Series gegenüber den herkömmlichen Sammlungsstrukturen ihre Integration in das pandas-Ökosystem, das reichhaltige Funktionen für Datenimport, -bereinigung, -manipulation und -analyse bietet. Während man mit Listen oder Tuples mühsam durch Daten iterieren und manuelle Bereinigungen durchführen müsste, ermöglichen Series und DataFrames die Anwendung von hochgradig optimierten Operationen auf ganzen Datensätzen mit minimalen Codezeilen.

Kurz gesagt, pandas Series füllen eine wichtige Lücke in der Python-Datenlandschaft, indem sie die leistungsstarken numerischen Fähigkeiten von numpy mit der Flexibilität und Benutzerfreundlichkeit kombinieren, die für die Arbeit mit realen, oft unvollständigen oder heterogenen Datensätzen erforderlich sind.

Erstellen

Eine Series kann mittels `pd.Series()` aus einer Vielzahl von Datenstrukturen erstellt bzw. umgewandelt werden, darunter Listen, Tuples, Dictionaries und numpy Arrays.

Nach der Erstellung ordnet Pandas der gesamte Series einen spezifischen Datentyp (`dtype`) zu, um die Natur der gespeicherten Informationen zu charakterisieren. So werden beispielsweise Ganzzahlen/Integer als `Int64`, Gleitkommazahlen/Floats als `Float64`, und Boolesche Werte/Booleans als `bool` typisiert. Darüber hinaus können auch spezielle Datentypen für Datum und Zeit (`datetime64`), Zeitdifferenzen (`timedelta64`) und Kategorien (`category`) verwendet werden, welche wir alle später noch genauer kennenlernen. All diese Datentypen unterstützen eine effiziente Datenverarbeitung und Analyse.

```
# Integer: Int64
pd.Series([1, 2, 42])
```

```
0     1
1     2
2    42
dtype: int64
```

```
# Floats: Float64
pd.Series(np.array([1.2, 3.4, 5]))
```

```
0    1.2
1    3.4
2    5.0
dtype: float64
```

```
# Booleans: bool
pd.Series((True, False, True))
```

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

```
# Datum und Zeit: datetime64
pd.Series([
  pd.Timestamp('2021-01-01'),
  pd.Timestamp('2021-04-15')
  ])
```

```
0   2021-01-01
1   2021-04-15
dtype: datetime64[ns]
```

```
# Zeitdifferenzen: timedelta64
pd.Series([
  pd.Timedelta('10 days'),
  pd.Timedelta('2 days')
  ])
```

```
0   10 days
1    2 days
dtype: timedelta64[ns]
```

```
# Kategorien: category
pd.Series(
  ['A', 'B', 'A'],
  dtype="category")
```

```
0    A
1    B
2    A
```

```
dtype: category
Categories (2, object): ['A', 'B']
```

Wenn eine Series jedoch Daten enthält, die nicht eindeutig einem der spezifischen Typen zugeordnet werden können, wie Textstrings oder Listen, verwendet Pandas den Datentyp object. Dieser Catch-all-Typ[1] ermöglicht die Speicherung von komplexen Strukturen, macht Operationen aber potenziell langsamer und weniger speicher-effizient.

```
pd.Series(['A', 'B', 'Hello'])
```

```
0        A
1        B
2    Hello
dtype: object
```

```
pd.Series([[1, 2], [3], [4, 5]])
```

```
0    [1, 2]
1       [3]
2    [4, 5]
dtype: object
```

```
pd.Series([42, {'key': 'value'}, 3.14])
```

```
0                  42
1    {'key': 'value'}
2                3.14
dtype: object
```

Vereinfacht ausgedrückt sind Series mit einem speziellen dtype eine Art “numpy Array mit Index” - also mit erweiterten Funktionalitäten und Flexibilität - wohingegen Series mit dtype `object` eher nur wie eine “Liste mit Index” sind.

Es fällt auf, dass die Series veritkal ausgerichtet ist. Das bedeutet, dass die Daten in einer Series in einer einzigen Spalte angeordnet sind, wobei jedes Element in der Spalte eine Zeile repräsentiert. Dieses vertikale Format ist charakteristisch für tabellarische

[1] “Catch-all-Typ”, wörtlich übersetzt als “Fang-alles-Typ”, bezeichnet in der Programmierung einen Datentyp, der als eine Art Auffangbecken für Daten verschiedenster Art dient. In pandas erfüllt der object Datentyp diese Rolle, indem er es ermöglicht, diverse Datenformen – von Zahlen und Texten bis hin zu komplexeren Objekten wie Listen oder Dictionaries – innerhalb einer Series zu speichern. Diese universelle Anwendbarkeit geht jedoch oft zu Lasten der Speicher- und Verarbeitungseffizienz, besonders im Vergleich zu den spezifischeren, typisierten Alternativen.

Daten und ermöglicht es, die Series als Grundlage (bzw. einzelne Spalte) für die Erstellung von DataFrames (mit mehreren Spalten) zu verwenden, die wir gleich kennenlernen werden.

Außerdem hat jeder Wert, also jede Zeile, einen **Index**. Der Index ist eine eindeutige Kennzeichnung für jede Zeile, die es ermöglicht, auf die Daten in der Series zuzugreifen und sie zu manipulieren. In den Beispielen oben haben wir keinen Index vordefiniert, sodass Pandas automatisch einen numerischen Index von `0` bis `n-1` erstellt hat, wobei `n` die Anzahl der Elemente in der Series ist. Gewissermaßen war also die Indizierung bis hierhin wie gehabt. Tatsächlich kann die Indizierung in Pandas jedoch viel flexibler gestaltet werden und gleicht eher der Schlüssel-Wert-Paar-Struktur eines Dictionaries.

Beim Erstellen einer Series aus einer Liste oder einem NumPy Array kann das Argument `index=` verwendet werden, um einen expliziten Index anzugeben. Dieser Index kann aus Zahlen, Strings oder Datumsangaben bestehen, wodurch die Datenzugriffe intuitiver und flexibler werden. Man kann aber auch direkt ein Dictionary übergeben, wobei die Keys des Dictionaries als Index verwendet werden.

```
pd.Series([10, 20, 30], index=['a', 'b', 'c'])
```

```
a    10
b    20
c    30
dtype: int64
```

```
pd.Series({'a': 10, 'b': 20, 'c': 30})
```

```
a    10
b    20
c    30
dtype: int64
```

Indizierung

Wir können dann den definierten Index verwenden, um auf die Daten zuzugreifen[2].

```
artikel = pd.Series({'Burger': 4.99, 'Pommes': 2.99, 'Cola': 1.99})
```

[2] Aktuell ist auch möglich auf die Daten über den numerischen Index zuzugreifen, also z.B. `artikel[0]`, was übrigens mit einem Dictionairy nicht möglich ist. Allerdings wird dabei jetzt schon eine `FuturWarning` ausgegeben, dass diese Art des Zugriffs in Zukunft nicht mehr unterstützt wird. Es ist also besser, sich jetzt schon an den Index zu gewöhnen.

```
artikel
```

```
Burger    4.99
Pommes    2.99
Cola      1.99
dtype: float64
```

```
artikel['Burger']
```

```
np.float64(4.99)
```

```
artikel['Pommes':'Cola']
```

```
Pommes    2.99
Cola      1.99
dtype: float64
```

Darüber hinaus können wir auch `.loc()` und `.iloc()` verwenden, um auf die Daten zuzugreifen. Diese Methoden führen wir aber erst gleich für die DataFrames ein, wo man sie ebenfalls verwenden kann.

Verwenden

Abgesehen von den Indices/Labels verhalten sich Series wie numpy Arrays. Das bedeutet, dass wir auch viele der bekannten numpy Funktionen (z.B. `np.sum(series)`) auf Series anwenden können. Gleichzeitig gibt es aber auch die “eingebauten” Methoden, die speziell für Series entwickelt wurden (z.B. `series.sum()`). In diesem Beispiel macht es keinen Unterschied, ob `np.sum(artikel)` oder `artikel.sum()` genutzt wird, denn beide liefern das gleiche Ergebnis. Jedoch, bei komplexeren Datenoperationen kann der Unterschied ins Gewicht fallen: `artikel.sum()` ignoriert beispielsweise standardmäßig Fehlwerte, während `np.sum(artikel)` ggf. schneller ist. Man sollte sich übrigens generell mit dem Gedanken anfreunden, dass es beim Programmieren nicht immer nur eine einzige “richtige” Lösung gibt.

```
np.sum(artikel)
```

```
np.float64(9.97)
```

```
artikel.sum()
```

```
np.float64(9.97)
```

DataFrames

Ein DataFrame ist eine zweidimensionale Datenstruktur, die aus einer Sammlung von Series besteht. Jede Series in einem DataFrame repräsentiert eine Spalte, wobei jede Zeile in einem DataFrame eine separate Beobachtung darstellt. DataFrames sind das Herzstück des pandas-Ökosystems und bilden die Grundlage für die meisten Datenanalysen und -manipulationen.

Erstellen

Hier ist nochmal die Tabelle, die wir im Dictionary-Kapitel erstellt haben - diesmal einmal als Dictionary und einmal als DataFrame:

```
dict_tabelle = {'Name': ['Donald', 'Daisy', 'Mickey'],
                'Jahr': [1934, 1937, 1928],
                'Tier': ['Ente', 'Ente', 'Maus']}
```

```
dict_tabelle
```

```
{'Name': ['Donald', 'Daisy', 'Mickey'], 'Jahr': [1934, 1937, 1928], 'Tier':
['Ente', 'Ente', 'Maus']}
```

```
pd.DataFrame(dict_tabelle)
```

```
     Name  Jahr  Tier
0  Donald  1934  Ente
1   Daisy  1937  Ente
2  Mickey  1928  Maus
```

Hinweis zur Ausgabe von DataFrames

In Jupyter Notebook und Jupyter Lab macht es einen Unterschied ob man sich einen DataFrame mit `print(df)` oder nur `df` ausgeben lässt, da letzteres etwas "schöner" mit html formatiert wird (siehe Screenshot hierunter). Hier in der Online-Dokumentation wird aber immer die nicht-formatierte Ausgabe-Form zu sehen sein, die man mit `print(df)` erhält - auch wenn der gezeigte Befehl `df` ist und demnach beim Ausführen in euren Jupyter Lab Umgebungen die formatierte Ausgabe erscheinen wird.

```
[1]: import pandas as pd

     dict_tabelle = {'Name': ['Donald', 'Daisy', 'Mickey'],
                     'Jahr': [1934, 1937, 1928],
                     'Tier': ['Ente', 'Ente', 'Maus']}

     df = pd.DataFrame(dict_tabelle)
```

```
[2]: print(df)
```

```
     Name  Jahr  Tier
0  Donald  1934  Ente
1   Daisy  1937  Ente
2  Mickey  1928  Maus
```

```
[3]: df
```

[3]:

	Name	Jahr	Tier
0	Donald	1934	Ente
1	Daisy	1937	Ente
2	Mickey	1928	Maus

Es ist also eine klare Tabellenstruktur zu erkennen - die Keys wurden zu Spaltennamen und die Values zum Spalteninhalt. Auch DataFrames haben Indices, die die Zeilen identifizieren - allerdings haben wir oben noch keine expliziten Indices definiert, sodass Pandas wieder einen numerischen Index von `0` bis `n-1` erstellt hat. Für diese Tabelle wäre es wohl angebrachter, die Namen als Index zu verwenden. Hier zwei Möglichkeiten, wie das gemacht werden kann: Namen als Indizes zusätzlich zur Namen-Spalte hinzufügen oder die Namen-Spalte als Index definieren.

```
df1 = pd.DataFrame(dict_tabelle,
  index=dict_tabelle['Name']
)
print(df1)
```

```
          Name  Jahr  Tier
Donald  Donald  1934  Ente
Daisy    Daisy  1937  Ente
Mickey  Mickey  1928  Maus
```

```
df2 = pd.DataFrame(dict_tabelle)
df2 = df2.set_index('Name')

print(df2)
```

```
        Jahr  Tier
Name
Donald  1934  Ente
Daisy   1937  Ente
Mickey  1928  Maus
```

Beim Erzeugen eines DataFrames aus einem Dictionary können die Werte Listen, Tuples, numpy arrays oder pandas Series sein, solange deren Längen gleich sind. Als Ausnahme kann man auch Skalarwerte verwenden, die dann für alle Zeilen gelten. In folgendem Beispiel sind möglichst viele verschiedene Datentypen in einem Dictionary enthalten, um dies zu demonstrieren. Auch lässt sich erkennen, dass der Index aus der einen pandas Series für den DataFrame übernommen wird. Mit der Methode `.types` kann angezeigt werden welcher Datentyp in welcher Spalte vorliegt.

```
dict_tabelle_2 = {
  'Name': ['Donald', 'Daisy', 'Mickey'],
  'Jahr': (1934, 1937, 1928),
  'Gewicht': np.array([50, 40, 50]),
  'Größe': pd.Series([1.2, 1.1, 1.3], index = ['DONALD', 'DAISY', 'MICKEY']),
  'Ursprung': 'Disney',
  'Füße' : 2
}

df = pd.DataFrame(dict_tabelle_2)
```

```
df
```

```
          Name  Jahr  Gewicht  Größe Ursprung  Füße
DONALD  Donald  1934       50    1.2   Disney     2
DAISY    Daisy  1937       40    1.1   Disney     2
MICKEY  Mickey  1928       50    1.3   Disney     2
```

```
df.dtypes
```

```
Name         object
Jahr          int64
Gewicht       int64
Größe       float64
Ursprung     object
Füße          int64
dtype: object
```

ℹ Details für Interessierte

Pandas wählt den Datentyp für jede Serie oder Spalte in einem DataFrame basierend auf den Daten, die sie enthält, um eine optimale Balance zwischen Speichereffizienz und Datenpräzision zu gewährleisten. Die Bezeichnungen `int64` und `int32` repräsentieren Ganzzahltypen/Integer mit 64 bzw. 32 Bit. `int64` kann einen größeren Wertebereich[3] darstellen und ist die Standardwahl auf 64-Bit-Systemen, um die vollständige Präzision und Größenkapazität dieser Plattformen zu nutzen. `int32` hingegen verwendet weniger Speicherplatz und kann auf Systemen mit geringeren Ressourcen oder bei Datensätzen, bei denen der größere Wertebereich von int64 nicht benötigt wird, eine effiziente Wahl darstellen. Die Auswahl zwischen diesen Datentypen erfolgt automatisch, kann aber auch manuell angepasst werden, um spezifische Anforderungen an die Speichergröße oder Kompatibilität mit externen Systemen zu erfüllen.

Indizierung

Nun können wir also auch mittels Index auf die Daten zugreifen. Dabei ist zu beachten, dass die Indizierung in DataFrames etwas komplexer ist als in Series, da wir sowohl auf die Zeilen als auch auf die Spalten zugreifen können. Prinzipiell gibt es vier Methoden, um auf die Daten zuzugreifen:

- `[ ]` & `.` - basierend auf den Spaltennamen
- `.loc[]` - basierend auf den Labels

[3] `int32`: Werte von −2.147.483.648 to +2.147.483.647; `int64`: Werte von −9.223.372.036.854.775.808 bis +9.223.372.036.854.775.807

- `.iloc[]` - basierend auf den numerischen Indizes

Eine Spalte

```
df2['Jahr']
# oder df2.Jahr
```

```
Name
Donald    1934
Daisy     1937
Mickey    1928
Name: Jahr, dtype: int64
```

```
df2.loc[:, 'Jahr']
```

```
Name
Donald    1934
Daisy     1937
Mickey    1928
Name: Jahr, dtype: int64
```

```
df2.iloc[:, 0]
```

```
Name
Donald    1934
Daisy     1937
Mickey    1928
Name: Jahr, dtype: int64
```

Eine Zeile

```
# nicht möglich
```

```
df2.loc['Donald', :]
```

```
Jahr    1934
Tier    Ente
Name: Donald, dtype: object
```

```
df2.iloc[0, :]
```

```
Jahr    1934
Tier    Ente
Name: Donald, dtype: object
```

Einzelne Zelle

```
df2['Jahr']['Donald']
# oder df2.Jahr['Donald']
```

```
np.int64(1934)
```

```
df2.loc['Donald', 'Jahr']
```

```
np.int64(1934)
```

```
df2.iloc[0, 1]
```

```
'Ente'
```

Alle drei Methoden liefern das gleiche Ergebnis, wenn es darum geht, auf eine Spalte zuzugreifen. Die Methode mit nur den eckigen Klammern ist dabei die kürzeste aber auch die am wenigsten flexible Methode: Wie man sieht, ist es mit ihr nicht möglich, auf eine Zeile zuzugreifen. Auch der Zugriff auf eine einzelne Zelle ist nur mit einem Umweg/Workaround möglich. So nutzen wir erst die eckigen Klammern, um die Spalte `Jahr` als Series zu extrahieren und dann die eckigen Klammern erneut, um innerhalb der Series auf das Element `Donald` zuzugreifen.

Es fällt auf, dass nicht nur Spalten als Series extrahiert werden, sondern auch beim Extrahieren einer Zeile eine Series zurückgegeben wird. Diese wird dann wie immer vertikal ausgerichtet, wobei die ehemaligen Spaltennamen nun als Index verwendet werden. Die Daten wurden also in gewisser Hinsicht rotiert bzw. transponiert.

Die Methoden `.loc[]` und `.iloc[]` sind sich bzgl. der Syntax sehr ähnlich, unterscheiden sich aber in der Art und Weise, wie sie die Indizes interpretieren. `.loc[]` basiert auf den Labels, während `.iloc[]` auf den numerischen Indizes basiert. `.iloc[]` ist daher die effizientere/schnellere Methode, da sie am wenigsten Rechenleistung benötigt. In beiden Methoden können zwei Argumente übergeben werden: das erste Argument bezieht sich auf die Zeilen und das zweite auf die Spalten. Hier wurde jeweils ein `:` verwendet, um alle Zeilen bzw. alle Spalten auszuwählen - also wie beim Slicing in Listen.

Ergänzende Infos zum `:`

Man kann auch nur ein Argument an `.loc[]` und `.iloc[]` übergeben. Das ist dann automatisch das erste Argument, also das für Zeilen. In diesem Fall wird der Standardwert für das zweite Argument, also das für die Spalten verwendet: ein `:`. Das heißt, dass man anstatt `df2.loc['Donald', :]` auch einfach nur `df2.loc['Donald']` schreiben könnte um zum selben Ergebnis zu kommen. Im Beispiel wurde das zweite Argument aber dennoch explizit angegeben, um den Vergleich der beiden Fälle zu erleichtern.

Zwar funktioniert der `:` hier wie gesagt wie beim Slicing in Listen, allerdings gibt es auch einen Unterschied. Werden die Labels und nicht die numerischen Indizes genutzt, so wird das Ende des Intervalls eingeschlossen. Hier der Unterschied:

```
liste = ["A", "B", "C"]
df3 = pd.DataFrame([1, 2, 3], index=liste)
```

Numerische Indizes schließen Intervallende **aus**

```
liste[0:2]
```

```
['A', 'B']
```

```
df3.iloc[0:2]
```

```
   0
A  1
B  2
```

Label Indizes schließen Intervallende **ein**

```
df3['A':'C']
```

```
   0
A  1
B  2
C  3
```

```
df3.loc['A':'C']
```

```
   0
A  1
```

```
B  2
C  3
```

Als letztes soll hier noch gezeigt werden wie man mehrere Spalten oder Zeilen auswählt ohne Slicing zu verwenden. Dafür kann nämlich eine Liste von Spaltennamen oder Zeilenlabels übergeben werden.

```
df
```

```
          Name  Jahr  Gewicht  Größe Ursprung  Füße
DONALD  Donald  1934       50    1.2   Disney     2
DAISY    Daisy  1937       40    1.1   Disney     2
MICKEY  Mickey  1928       50    1.3   Disney     2
```

```
df[['Jahr', 'Ursprung']]
```

```
        Jahr Ursprung
DONALD  1934   Disney
DAISY   1937   Disney
MICKEY  1928   Disney
```

```
df.loc[:, ['Jahr', 'Ursprung']]
```

```
        Jahr Ursprung
DONALD  1934   Disney
DAISY   1937   Disney
MICKEY  1928   Disney
```

```
df.iloc[:, [1, 4]]
```

```
        Jahr Ursprung
DONALD  1934   Disney
DAISY   1937   Disney
MICKEY  1928   Disney
```

Weitere Ressourcen

- Pandas Tutorial #1 - DataFrames (Python für Data Science) [nur bis 7:27]

Übungen

Führe folgenden Code aus und prüfe was die Befehle machen. Berechne außerdem nach jeder Zeile die Summe von allen Elementen in `daten`.

```
daten = pd.Series([4, 9, -5, 2])
daten['d'] = 9
daten = daten.drop(2)
```

Summe nach Zeile 1: __

Summe nach Zeile 2: __

Summe nach Zeile 3: __

Erstelle einen DataFrame `personen` mit den Spalten `Name`, `Alter` und `Ort`. Die Daten sollen folgende Personen enthalten: Den 34-jährigen Max Müller aus Hamburg, die 30-jährige Trang Nguyen aus Stuttgart und den 1-jährigen Ivo Hernandez aus Köln. Sorge außerdem manuell mit dem `index=` Argument dafür, dass die Initialen der Personen (also z.B. `MM`) als Index/Zeilenlabel verwendet werden. Berechne dann basierend auf dem DataFrame und mithilfe von `.mean()` das mittlere Alter der Personen. Berechne danach nochmal das mittlere Alter, aber diesmal nur für die volljährigen Personen.

- (A) Geschafft

Erstelle einen DataFrame mit 8 Zeilen und 8 Spalten. Die Spaltennamen sollen Die Buchstaben A-H und die Zeilenlabel die Zahlen 1-8 sein. Fülle die Zellen dieser Tabelle so mit Buchstaben, dass es der Startaufstellung eines Schachbretts entspricht. Kürze dabei die Bezeichnung der Figuren (König, Dame, Turm, Läufer, Springer, Bauer) immer nur mit ihrem ersten Buchstaben ab. Felder/Zellen, auf denen keine Figur steht, sollen einen Punkt enthalten. Nutze dann erst `.loc[]` und danach auch nochmal `.iloc[]`, um um jeweils eine 2x2 Tabelle zu extrahieren, in der ausschließlich die Läufer übrig bleiben.

- (A) Geschafft