Teil IV

Pandas

18 Einführung in Pandas

Bild 18.1 Pandabären und DataFrame

Auch wenn Panda-Bären allgemein als süß und niedlich angesehen werden, bleiben wir in diesem Kapitel bei der Programmierung. Bei Pandas handelt es sich um eine Bibliothek zur Datenanalyse mit Python. Der Name Pandas ist ein Akronym für „panel data“. In der Ökonometrie versteht man darunter Datensätze, die sowohl eine zeitliche als auch eine nichtzeitliche Dimension aufweisen. Man kann diese Daten auch als Matrix betrachten, in der beispielsweise die Spaltenrichtung als Individualdimension und die Zeilenrichtung als Zeitdimension angesehen wird. Dies ist aber bereits ein sehr spezieller Anwendungsfall. Besser ist es, Pandas als ein System zu sehen, das auf Tabellen, so wie sie bei der Tabellenkalkulation verwendet werden, beruht, so wie etwa in dem bekanntesten Tabellenkalkulationsprogramm „Excel“. Eine besondere Stärke von Pandas liegt auch darin, dass es direkt CSV-, DSV- und Excel-Dateien einlesen und schreiben kann.

Oft gibt es Verwirrung darüber, ob Pandas nicht eine Alternative zu NumPy, SciPy und Matplotlib sei. Die Wahrheit ist aber, dass Pandas auf NumPy aufbaut. Das bedeutet auch, dass NumPy für Pandas Voraussetzung ist. SciPy und Matplotlib werden von Pandas nicht grundlegend benötigt, sind aber eine wertvolle Ergänzung. Deshalb listet das Pandas-Projekt diese auch als „optionale Abhängigkeiten“.

18.1 Datenstrukturen

Die wichtigen Datenstrukturen von Pandas sind:

- Series und
- DataFrame

Da der Datentyp `DataFrame` auf dem Typ `Series` basiert, beginnen wir mit `Series`.

18.2 Series

Ein Series-Objekt kann man wie die Spalte in einer Excel-Tabelle plus dem zugehörigen Index sehen. Anders ausgedrückt: Eine Series ist ein eindimensionales Array-ähnliches Objekt mit einem Index. Während bei einem Array der Index den natürlichen Zahlen von 0 bis zur Länge des Arrays (exklusive) entspricht, kann der Index einer Series beliebig sein, solange er hashable ist.

Sowohl der Index als auch die Werte einer Series müssen einen einheitlichen Datentyp aufweisen, also beispielsweise nur Integers, Floats, Strings usw.

Eine Series kann als eine Datenstruktur mit zwei Arrays angesehen werden: Ein Array fungiert als Index, d.h. als Bezeichner (Label), und ein Array beinhaltet die aktuellen Daten (Werte).

Wir definieren im folgenden Beispiel ein einfaches Series-Objekt, indem wir dieses Objekt mit einer Liste instanziieren. Wir werden später sehen, dass wir auch andere Daten-Objekte verwenden können, z.B. NumPy-Arrays und Dictionaries.

```
import pandas as pd
S = pd.Series([11, 28, 72, 3, 5, 8])
print(S)
```

Ausgabe:

```
0    11
1    28
2    72
3     3
4     5
5     8
dtype: int64
```

Wir haben in unserem Beispiel keinen Index definiert. Trotzdem sehen wir zwei Spalten in der Ausgabe: Die rechte Spalte zeigt unsere Daten, die linke Spalte stellt den Index dar. Pandas erstellt einen Default-Index, der bei 0 beginnt und bis 5 läuft.

Wir können direkt auf die Indizes und die Werte der Series S zugreifen:

```
print(S.index)
print(S.values)
```

Ausgabe:

```
RangeIndex(start=0, stop=6, step=1)
[11 28 72  3  5  8]
```

Wenn wir dies mit der Erstellung eines Arrays in NumPy vergleichen, stellen wir viele Gemeinsamkeiten fest:

```
import numpy as np
X = np.array([11, 28, 72, 3, 5, 8])
print(X)
print(S.values)
# both are the same type:
print(type(S.values), type(X))
```

Ausgabe:

```
[11 28 72  3  5  8]
[11 28 72  3  5  8]
<class 'numpy.ndarray'> <class 'numpy.ndarray'>
```

Bis hierhin unterscheiden sich die Series noch nicht wirklich von den ndarrays aus NumPy. Das ändert sich aber, sobald wir Series-Objekte mit individuellen Indizes definieren:

```
fruits = ['apples', 'oranges', 'cherries', 'pears']
quantities = [20, 33, 52, 10]
S = pd.Series(quantities, index=fruits)
print(S)
```

Ausgabe:

```
apples      20
oranges     33
cherries    52
pears       10
dtype: int64
```

Eine großer Vorteil gegenüber NumPy-Arrays ist hier ganz offensichtlich: Wir können beliebige Indizes verwenden.

Wenn wir zwei Series-Objekte mit denselben Indizes addieren, so erhalten wir ein neues Series-Objekt mit diesem Index, und die Werte entsprechen den Summen der entsprechenden Werte aus den beiden Series-Objekten.

```
fruits = ['apples', 'oranges', 'cherries', 'pears']

S = pd.Series([20, 33, 52, 10], index=fruits)
S2 = pd.Series([17, 13, 31, 32], index=fruits)
print(S + S2)
print("Summe aus S: ", sum(S))
```

Ausgabe:

```
apples      37
oranges     46
cherries    83
pears       42
dtype: int64
Summe aus S:  115
```

Die Indizes müssen für die Addition von Series-Typen nicht identisch sein. Der resultierende Index ist eine „Vereinigung“ beider Indizes. Wenn ein Index nicht in beiden Series-Objekten vorkommt, so wird der entsprechende Wert auf NaN gesetzt:

```
fruits = ['peaches', 'oranges', 'cherries', 'pears']
fruits2 = ['raspberries', 'oranges', 'cherries', 'pears']

S = pd.Series([20, 33, 52, 10], index=fruits)
S2 = pd.Series([17, 13, 31, 32], index=fruits2)
print(S + S2)
```

Ausgabe:

```
cherries       83.0
oranges        46.0
peaches         NaN
pears          42.0
raspberries     NaN
dtype: float64
```

Prinzipiell können die Indizes auch komplett verschieden sein, wie im folgenden Beispiel:

```
fruits = ['apples', 'oranges', 'cherries', 'pears']

fruits_tr = ['elma', 'portakal', 'kiraz', 'armut']
```

```
S = pd.Series([20, 33, 52, 10], index=fruits)
S2 = pd.Series([17, 13, 31, 32], index=fruits_tr)
print(S + S2)
```

Ausgabe:

```
apples     NaN
armut      NaN
cherries   NaN
elma       NaN
kiraz      NaN
oranges    NaN
pears      NaN
portakal   NaN
dtype: float64
```

18.2.1 Indizierung

Es ist möglich, auf einzelne Werte eines Series-Objekts zuzugreifen:

```
print(S['apples'])
```

Ausgabe:

```
20
```

Man kann auch auf mehrere Indizes gleichzeitig zugreifen, wenn man ein Listen- oder ein Array-ähnliches Objekt übergibt:

```
print(S[['apples', 'oranges', 'cherries']])
```

Ausgabe:

```
apples      20
oranges     33
cherries    52
dtype: int64
```

Filterung mit einem Booleschen Array:

```
S[S>30]
```

Ausgabe:

```
oranges     33
cherries    52
dtype: int64
```

Wie bei NumPy sind auch Operationen mit Skalaren oder die Anwendung von mathematischen Funktionen auf ein Series-Objekt möglich:

```
import numpy as np
print((S + 3) * 4)
print("======================")
print(np.sin(S))
```

Ausgabe:

```
apples       92
oranges     144
cherries    220
pears        52
```

```
dtype: int64
======================
apples      0.912945
oranges     0.999912
cherries    0.986628
pears      -0.544021
dtype: float64
```

18.2.2 pandas.Series.apply

```
Series.apply(func, convert_dtype=True, args=(), **kwds)
```

Die Funktion „func“ wird auf das Series-Objekt angewendet und liefert, in Abhängigkeit von „func“, entweder ein Series-Objekt oder ein DataFrame-Objekt zurück.

Parameter	Bedeutung
func	Eine Funktion, die auf das gesamte Series-Objekt (NumPy-Funktion) oder nur auf einzelne Werte des Series (Python-Funktion) angewendet wird.
convert_dtype	Ein Boolescher Wert. Wenn dieser auf True gesetzt wird (Standard), so wird versucht, bei der Anwendung einen besseren dtype für die elementweisen Funktionsergebnisse zu finden. Wenn der Parameter auf False gesetzt wird, so wird dtype=objekt verwendet.
args	Positionsargumente, die an die Funktion „func“ übergeben werden, zusätzlich zu den Werten des Series-Objektes.
**kwds	Zusätzliche Schlüsselwortargumente, die als Schlüsselworte an die Funktion übergeben werden.

Beispiel:

```
S.apply(np.log)
```

Ausgabe:

```
apples      2.995732
oranges     3.496508
cherries    3.951244
pears       2.302585
dtype: float64
```

Wir können auch Python-Lambda-Funktionen benutzen. Wir werden nun die Anzahl der Früchte prüfen: Wenn weniger als 50 von einer Sorte vorhanden sind, so soll der Bestand um 10 erhöht werden. Ansonsten lassen wir den Betrag unverändert:

```
S.apply(lambda x: x if x > 50 else x+10 )
```

Ausgabe:

```
apples      30
oranges     43
cherries    52
pears       20
dtype: int64
```

18.2.3 Zusammenhang zu Dictionaries

Ein Series-Objekt kann wie ein geordnetes Python-Dictionary mit einer festen Länge angesehen werden.

Wir können bei der Erstellung eines Series-Objekts ein Dictionary übergeben. Wir erhalten ein Series-Objekt mit den Schlüsseln des Dictionarys als Indizes. Die Indizes werden sortiert.

```
cities = {"London":    8615246,
          "Berlin":    3562166,
          "Madrid":    3165235,
          "Rome":      2874038,
          "Paris":     2273305,
          "Vienna":    1805681,
          "Bucharest": 1803425,
          "Hamburg":   1760433,
          "Budapest":  1754000,
          "Warsaw":    1740119,
          "Barcelona": 1602386,
          "Munich":    1493900,
          "Milan":     1350680}
city_series = pd.Series(cities)
print(city_series)
```

Ausgabe:

```
London       8615246
Berlin       3562166
Madrid       3165235
Rome         2874038
Paris        2273305
Vienna       1805681
Bucharest    1803425
Hamburg      1760433
Budapest     1754000
Warsaw       1740119
Barcelona    1602386
Munich       1493900
Milan        1350680
dtype: int64
```

18.3 NaN – Fehlende Daten

Ein Problem bei Aufgaben in der Datenanalyse besteht in fehlenden Daten.

Schauen wir uns noch einmal das vorherige Beispiel an. Dabei erkennen wir, dass die Indizes der Series mit den Keys des Dictionaries übereinstimmen, aus dem das Series-Objekt cities_series erzeugt wurde. Nehmen wir nun an, dass wir einen Index haben wollen, der sich nicht mit den Keys des Dictionaries überschneidet. Dafür können wir eine Liste oder ein Tupel dem Keyword-Argument 'index' mitgeben, um die Indizes zu definieren. Im nächsten Beispiel übergeben wir eine Liste (oder ein Tupel) als Indizes, welches nicht mit den Keys übereinstimmt. Das bedeutet, dass einige Städte des Dictionaries fehlen und für Stuttgart und Zürich keine Daten vorhanden sind.

```
my_cities = ["London", "Paris", "Zurich", "Berlin",
             "Stuttgart", "Hamburg"]
my_city_series = pd.Series(cities, index=my_cities)
print(my_city_series)
```

Ausgabe:

```
London       8615246.0
Paris        2273305.0
Zurich             NaN
Berlin       3562166.0
Stuttgart          NaN
Hamburg      1760433.0
dtype: float64
```

Abgesehen von den NaN-Werten werden bei den anderen Bevölkerungswerten die Werte in float-Werte gewandelt. Im folgenden Beispiel gibt es keine fehlenden Daten, und damit werden die Werte in Integer-Werte gewandelt:

```
my_cities = ["London", "Paris", "Berlin", "Hamburg"]
my_city_series = pd.Series(cities, index=my_cities)
my_city_series
```

Ausgabe:

```
London     8615246
Paris      2273305
Berlin     3562166
Hamburg    1760433
dtype: int64
```

18.3.1 Die Methoden isnull() und notnull()

Wir sehen, dass die Städte, die nicht im Dictionary existieren, den Wert NaN zugewiesen bekommen. `NaN` steht für „not a number“. Es kann in unserem Beispiel auch als „fehlt“ verstanden werden.

Wir können mit den Methoden `isnull` und `notnull` fehlende Werte prüfen:

```
my_cities = ["London", "Paris", "Zurich", "Berlin",
             "Stuttgart", "Hamburg"]
my_city_series = pd.Series(cities, index=my_cities)
print(my_city_series.isnull())
```

Ausgabe:

```
London       False
Paris        False
Zurich        True
Berlin       False
Stuttgart     True
Hamburg      False
dtype: bool
```

```
print(my_city_series.notnull())
```

Ausgabe:

```
London        True
Paris         True
Zurich       False
```

```
Berlin        True
Stuttgart    False
Hamburg       True
dtype: bool
```

18.3.2 Zusammenhang zwischen NaN und None

Wir erhalten ebenfalls `NaN`, wenn ein Wert in dem Dictionary `None` ist:

```
d = {"a":23, "b":45, "c":None, "d":0}
S = pd.Series(d)
print(S)
```

Ausgabe:

```
a    23.0
b    45.0
c     NaN
d     0.0
dtype: float64
```

```
pd.isnull(S)
```

Ausgabe:

```
a    False
b    False
c     True
d    False
dtype: bool
```

```
pd.notnull(S)
```

Ausgabe:

```
a     True
b     True
c    False
d     True
dtype: bool
```

18.3.3 Fehlende Daten filtern

Es ist möglich, die fehlenden Daten mit der Methode `dropna` aus einem Series-Objekt herauszufiltern. Die Methode liefert ein neues Series-Objekt zurück, welches keine `NaN`-Werte enthält:

```
print("Vorher:\n")
print(my_city_series)
print("\nNachher:\n")
print(my_city_series.dropna())
```

Ausgabe:

```
Vorher:

London       8615246.0
Paris        2273305.0
Zurich             NaN
```

```
Berlin       3562166.0
Stuttgart          NaN
Hamburg      1760433.0
dtype: float64

Nachher:

London     8615246.0
Paris      2273305.0
Berlin     3562166.0
Hamburg    1760433.0
dtype: float64
```

18.3.4 Fehlende Daten auffüllen

In vielen Fällen will man die fehlenden Daten gar nicht filtern. Stattdessen möchten man diese mit passenden Werten auffüllen. Eine gute Methode ist `fillna`:

```
print(my_city_series.fillna(0))
```

Ausgabe:

```
London       8615246.0
Paris        2273305.0
Zurich             0.0
Berlin       3562166.0
Stuttgart          0.0
Hamburg      1760433.0
dtype: float64
```

Okay, das sind nicht wirklich passende Werte für die Bevölkerung von Zürich und Stuttgart. Wenn wir der Methode `fillna` ein Dictionary mitgeben, können wir so die passenden Daten bereitstellen, z.B. die Bevölkerungswerte für Zürich und Stuttgart. Wir setzen den Parameter `inplace` auf `True`, damit die Änderungen auch in dem Objekt geändert werden. Bei `True` wird ein neues Objekt mit den Einsetzungen erzeugt und zurückgeliefert, und das alte bleibt dabei unverändert:

```
missing_cities = {"Stuttgart":597939, "Zurich":378884}
my_city_series.fillna(missing_cities, inplace=True)
my_city_series
```

Ausgabe:

```
London       8615246.0
Paris        2273305.0
Zurich        378884.0
Berlin       3562166.0
Stuttgart     597939.0
Hamburg      1760433.0
dtype: float64
```

Dabei haben wir aber immer noch das Problem mit Integer-Werten. Die Werte, die Integer sein sollten wie die Anzahl der Menschen, werden nach wie vor in Float-Werte gewandelt. Mit der Methode `astype` können wir die Daten in Integer wandeln:

```
my_city_series = my_city_series.astype(int)
print(my_city_series)
```

Ausgabe:

```
London       8615246
Paris        2273305
Zurich        378884
Berlin       3562166
Stuttgart     597939
Hamburg      1760433
dtype: int64
```