Pandas

Joern Ploennigs · AI4SC

> It isn't so much what's on the table that matters, as what's on the chairs.
>
> — W.S. Gilbert

Pandas

Pandas ist die wichtigste Python-Bibliothek für die Arbeit mit Tabellen (DataFrames). Pandas nutzt intern `Numpy` (Vektoren und Matrizen) fügt aber Tabellenkonzepte hinzu (Spaltennamen, Zeilenindizes, Aggregationsoperatoren, etc.).

Pandas eignet sich ideal zur Analyse tabellarischer Daten wie Sensorlogs von Brücken, Baufortschrittsprotokollen oder Materialprüfdaten in Laboren.

Pandas enthält zwei zentrale Datentypen: `Series` und `DataFrame`. `Series` werden oft als ergänzende Datenstruktur verwendet, die eine einzelne Variable (Spalte) im DataFrame darstellen. Sie kann aber auch als vektorisiertes Wörterbuch (`dict` in Python) verwendet werden, das Schlüssel (Indizes) mit Werten verknüpft. DataFrame ist ähnlich zu Tabellen in Programmen wie Excel oder R, wie sie in R oder Spark implementiert sind. Wenn Sie die einzelnen Spalten und Zeilen extrahieren, erhalten Sie diese normalerweise in Form von `Series`. Es ist also äußerst nützlich, die Grundlagen dieser Typen zu kennen, wenn man mit Datenrahmen arbeitet. Sowohl `DataFrame` als auch `Series` enthalten einen Index, einen speziellen Zeilenkennung (Index), der sehr nützlich ist, um Informationen auf der Grundlage von Namen zu extrahieren oder um verschiedene Variablen in einem Datenrahmen zusammenzufassen (siehe Abschnitt Verkettung von Daten mit `pd.concat`).

Wozu brauche ich das im ML? Die meisten Datensätze die man analysiert, werden als Tabellen erfasst. Wollen wir z.B. ein ML-Modell erstellen um Risse in Beton zu klassifizieren, muss man Trainingsdaten einlesen, filtern, aufbereiten. Genau dafür wird Pandas verwendet.

Pandas wird in Python meist mit der Abkürzung `pd` importiert. Da man oft auch einige Funktionen aus Numpy brauch, importiert man meist beides zusammen.

```
# Import von NumPy und Pandas
import numpy as np
import pandas as pd
```

Spalten als Series

Pandas enthält zwei zentrale Datentypen: `Series` zur Darstellung von Spalten und `DataFrame` für ganze Tabellen. Wir diskutieren zuerst `Series`, da sie die Spalten in `DataFrame` darstellen und ihnen somit zugrunde liegen.

Eine `Series` ist eine eindimensionale Spalte (oder -zeile) von Werten. In gewisser Weise ähnelt sie einer `list` in Python oder einem 1-dimensionalem `nbarray` in NumPy, worauf es abgebildet wird. Es fügt allerdings einen Index hinzu, mit dem man einzelne Werte Schlüssel zuweisen kann, womit man die Werte dann nachschlagen kann (Womit es auch einem `dict` in Python ähnelt). Es ermöglicht also nicht nur einen positionsbezogenen Zugriff durch numerische Indizes, wie in NumPy, sondern auch einen schlüsselbasierten Zugriff.

Lassen Sie uns eine einfache Serie der Einwohner in Millionen für Europa erstellen:

```
pop = pd.Series( [ 83, 67, 67, 59] )
pop
```

```
0    83
1    67
2    67
3    59
dtype: int64
```

Die Reihe wird in zwei Spalten dargestellt. Die erste ist der Index, die zweite ist der Wert. In diesem Beispiel ist der Index im Wesentlichen nur die Zeilennummer und ist nicht sehr nützlich. Das liegt daran, dass wir keinen spezifischen Index angegeben haben und Pandas daher nur die Zeilennummer ausgewählt hat. Unter den beiden Spalten sehen Sie auch den Datentyp, in diesem Fall 64-Bit-Integer, den Standard-Datentyp für Integer in Python.

Das Problem der oben dargestellten Einwohnerzahl ist, dass wir nicht wissen, welche Zahl welchem Land entspricht. Deshalb wollen wir der Serie einen informativeren Index mit den Länderkürzeln zuweisen:

```
pop = pd.Series( [ 83, 67, 67, 59], index = ['de', 'fr', 'gb', 'it'])
# population, in millions
pop
```

```
de    83
fr    67
gb    67
it    59
dtype: int64
```

Wir können Werte extrahieren und über die entsprechenden Attribute indexieren:

```
pop.values
```

```
array([83, 67, 67, 59])
```

```
pop.index
```

```
Index(['de', 'fr', 'gb', 'it'], dtype='str')
```

Reihen unterstützen auch die gewöhnliche Mathematik, z. B. können wir Operationen durchführen wie

```
2 * pop
```

```
de    166
fr    134
gb    134
it    118
dtype: int64
```

oder Filtern wie

```
pop > 70
```

```
de     True
fr    False
gb    False
it    False
dtype: bool
```

Was eine weitere Serie mit logischen Wahrheitswerten zurück gibt.

Erstellen von DataFrames

Die wichtigste Datenstruktur in Pandas ist der `DataFrame`. Er repräsentiert einfach eine Tabelle, wie ein Tabellenblatt in Excel. Intern besteht er aus vielen Serien (für jede Spalte eine), die den gleichen Index haben.

Es gibt verschiedene Möglichkeiten `DataFrame` Objekte zu erzeugen. Man kann DataFrames direkt aus Numpy Objekten erzeugen, dabei wird automatisch ein numerischer Index, wie in Numpy erzeugt.

```
A = np.array([[1, 2], [3, 4]])
A_DF = pd.DataFrame(A)
A_DF
```

	0	1
0	1	2
1	3	4

Wir können dabei aber auch neue Spaltennamen und Zeilenindexe zuweisen:

```
A = np.array([[1, 2], [3, 4]])
A_DF = pd.DataFrame(A, columns=["Spalte A", "Spalte B"], index=["Zeile 1",
"Zeile 2"])
A_DF
```

	Spalte A	Spalte B
Zeile 1	1	2
Zeile 2	3	4

Häufig ist es aber intuitiver DataFrames aus Python Dictionary Objekten zu erzeugen. Dabei erzeugt man ein Objekt, wo man den Spaltennamen die Serien mit den Werten zuweist und dann ein DataFrame mit dem entsprechendem Index erstellt.

```
df = {'de': [ 83, 82, 79, 68], 'fr': [ 67, 59, 52, 42], 'gb': [ 67, 59, 56,
50], 'it': [ 59, 58, 55, 47]}
pop = pd.DataFrame(df, index=[2010, 1998, 1973, 1950])
pop
```

	de	fr	gb	it
2010	83	67	67	59
1998	82	59	59	58
1973	79	52	56	55
1950	68	42	50	47

Wenn man Daten automatisch erstellt, wählt man meistens eine zeilenweise Darstellung, wo man eine Liste mit Objekten der einzelnen Zeilen hat. Dadurch kann man neue Zeilen in einem Skript einfach hinzufügen.

```
df = [{'de': 83, 'fr': 67, 'gb': 67, 'it': 59, 'year': 2010},
 {'de': 82, 'fr': 59, 'gb': 59, 'it': 58, 'year': 1998},
 {'de': 79, 'fr': 52, 'gb': 56, 'it': 55, 'year': 1973},
 {'de': 68, 'fr': 42, 'gb': 50, 'it': 47, 'year': 1950}]
pop = pd.DataFrame(df)
pop
```

	de	fr	gb	it	year
0	83	67	67	59	2010
1	82	59	59	58	1998
2	79	52	56	55	1973
3	68	42	50	47	1950

Hier muss man erst den Index aus der Spalte `year` zuweisen.

```
pop=pop.set_index("year")
pop
```

	de	fr	gb	it
year				
2010	83	67	67	59
1998	82	59	59	58
1973	79	52	56	55
1950	68	42	50	47

Einlesen von DataFrames aus Dateien

Für viele praktische Anwendungen liegen Daten bereits in Tabellenform vor. Ein sehr übliches Datenformat hierfür sind CSV-Dateien. Diese können einfach mit `pd.read_csv` eingelesen werden. Die Funktion nimmt den Dateinamen als erstes Argument und unterstützt viele andere Optionen weiteren Argumente. Die wichtigsten sind:

- `sep`: Trennzeichen der Spalten in der CSV-Datei (Standard: „,“ - in Deutschland oft „;“ oder “).
- `decimal`: Zeichen als Dezimaltrennzeichen erkennen (Standard: „.“ - in Deutschland oft „,“).
- `na_values`: Liste mit Werten, die als fehlende Werte interpretiert werden sollen (Standard: „NaN“).

- `header`: Zeilennummer des Headers (Standard: 0 für erste Zeile). Kann `false` gesetzt werden um kein Header einzulesen, wenn dieser z.B. nicht vorhanden ist.
- `skiprows`: Anzahl der zu überspringenden Zeilen am Anfang der Datei. Sollte verwendet werden, wenn am Anfang der Datei Meta-Information stehen, die zu ignorieren sind.
- `names`: Optionale Liste mit Spaltennamen (überschreibt die Header-Zeile).
- `nrows`: Anzahl der zu lesenden Zeilen (Standard: alle).
- `usecols`: Liste mit Spaltennamen, die gelesen werden sollen.

Es empfiehlt sich CSV-Dateien zuerst in einem Texteditor oder in Excel zu öffnen, um die Struktur zu verstehen. Im folgenden Beispiel lesen wir historische Daten des Deutschen Wetterdienstes ein. Die ersten Zeilen in der Datei mit den Wetterdaten sehen wie folgt aus:

Wir sehen also, dass die Datei in der ersten Zeile die Spaltennamen hat. Ab der zweiten Spalte stehen Werte. Für alle Werte sind die Spalten in der Datei durch Semikolon separiert. Pandas geht aber von der Trennung durch Komma aus, es heißt ja CSV - Comma Separated Values. Zudem ist bei deutschen CSV auch darauf zu achten, dass als Dezimaltrennzeichen, das `,` verwendet wird, im Englischen jedoch standartmäßig der `.`, was auch Pandas annimmt.

Für unser Beispiel wollen wir nur die Spalten „MESS_DATUM“, „PM“, und „TMK“ auswählen und nur 200 Zeilen einlesen:

```
wetter = pd.read_csv("../data/Wetter/warnemuende_1960.csv", sep=';',
usecols=["MESS_DATUM", "PM", "TMK"], nrows=200)
wetter
```

	MESS_DATUM	PM	TMK
0	19470101	1019.3	−2.1
1	19470103	1032.4	−4.8
2	19470105	1031.9	−12.9
3	19470108	1023.1	−12.9
4	19470109	1017.5	−9.9
…	…	…	…
195	19470719	1009.5	17.2
196	19470720	1008.2	17.4
197	19470721	1013.5	16.8
198	19470722	1019.8	18.6
199	19470723	1019.6	19.8

Den Umfang der eingelesenen Daten können wir uns mit `shape` Anzeigen lassen.

```
wetter.shape
```

```
(200, 3)
```

Mit `columns` können wir die Spaltennamen anzeigen

```
wetter.columns
```

```
Index(['MESS_DATUM', 'PM', 'TMK'], dtype='str')
```

und sie auch durch Zuweisung umbenennen

```
wetter.columns = ["Datum", "Druck", "Temperatur"]
```

Daten auswählen

Indizierung bezieht sich auf die Auswahl von Daten aus Datenrahmen und Serien basierend auf Variablennamen, logischen Bedingungen und Position. Es handelt sich um eine komplexe Aufgabe mit verschiedenen Varianten für unterschiedliche Anwendungsfälle.

Variablen auswählen

Wir haben bereits festgestellt, dass der Hauptunterschied zwischen Numpy und Pandas ist, dass die Tabellen in Form von DataFrames Spaltennamen haben. Wir können eine einzelne Variable entweder mit `["Spaltenname"]` oder einer Abkürzung als Attribut `.Spaltenname` extrahieren (Hinweis: Ersetzen Sie *Spaltenname* durch den Namen der relevanten Variable):

```
wetter["Temperatur"]  # Spalte Temperatur als Serie
wetter.Temperatur  # Die gleiche Spalte Temperatur als Serie
```

```
0      -2.1
1      -4.8
2     -12.9
3     -12.9
4      -9.9
       ...
195    17.2
196    17.4
197    16.8
198    18.6
199    19.8
Name: Temperatur, Length: 200, dtype: float64
```

Diese Konstrukte geben die Spalte als Serie zurück. Wenn wir lieber ein DataFrame mit einer einzelnen Spalte erhalten möchten, können wir den Variablennamen in eine Liste einpacken:

```
wetter[["Temperatur","Druck"]]  # Spalte Temperatur als DataFrame
```

	Temperatur	Druck
0	−2.1	1019.3
1	−4.8	1032.4
2	−12.9	1031.9
3	−12.9	1023.1
4	−9.9	1017.5
…	…	…
195	17.2	1009.5
196	17.4	1008.2
197	16.8	1013.5
198	18.6	1019.8
199	19.8	1019.6

Die Attributschreibweise ist normalerweise der einfachere Weg, funktioniert jedoch nicht, wenn Sie einen indirekten Variablennamen verwenden müssen (Variablenname, der in einer anderen Variablen gespeichert ist) oder wenn der Variablenname Leerzeichen oder andere Sonderzeichen enthält. Es funktioniert auch nicht, um neue Variablen im Datenrahmen zu erstellen. Weitere Informationen finden Sie im Abschnitt Daten modifizieren.

Das vorherige Beispiel, bei dem wir eine einzelne Spalte als Datenrahmen extrahiert haben, anstelle von *Series*, deutet auch darauf hin, wie man mehr als eine Variable extrahiert: Verpacken Sie einfach alle erforderlichen Variablennamen in eine Liste:

```
vars = ["Datum", "Temperatur"]
wetter[vars]
```

	Datum	Temperatur
0	19470101	−2.1
1	19470103	−4.8
2	19470105	−12.9
3	19470108	−12.9
4	19470109	−9.9
…	…	…
195	19470719	17.2

	Datum	Temperatur
196	19470720	17.4
197	19470721	16.8
198	19470722	18.6
199	19470723	19.8

Daten filtern

Filtern bezieht sich darauf, nur eine Teilmenge von Zeilen aus dem DataFrame basierend auf bestimmten Bedingungen zu extrahieren. Die Bedingungen sind logische Operationen, die je nach den Werten in jeder Zeile entweder wahr oder falsch sein können. Durch Filtern wird ein Unter-DataFrame erzeugt, in dem nur diejenigen Beobachtungen vorhanden sind, die den Auswahlkriterien entsprechen. Hier ist ein Beispiel:

```
wetter[wetter.Temperatur > 20]
```

	Datum	Druck	Temperatur
129	19470514	1011.8	20.6
146	19470531	1018.8	21.6
147	19470601	1014.0	20.8
148	19470602	1011.9	20.7
171	19470625	1016.6	20.8
174	19470628	1020.8	24.7
175	19470629	1019.0	27.8
176	19470630	1016.2	22.0
180	19470704	1013.4	21.7

Bitte beachten Sie, dass wir auf Datenvariablen in der logischen Bedingung als `wetter.Temperatur` verweisen. Dabei können wir komplexere Auswahlbedingungen verwenden, zum Beispiel können wir nach sehr niedrigen oder sehr hohen Temperaturen suchen, wie folgt:

```
wetter[wetter.Temperatur < 0 | (wetter.Temperatur > 30)]
```

	Datum	Druck	Temperatur
0	19470101	1019.3	−2.1
1	19470103	1032.4	−4.8
2	19470105	1031.9	−12.9

	Datum	Druck	Temperatur
3	19470108	1023.1	−12.9
4	19470109	1017.5	−9.9
…	…	…	…
65	19470311	1003.2	−5.8
66	19470312	1015.8	−7.4
67	19470313	1004.4	−4.7
68	19470314	988.6	−1.6
69	19470315	1016.6	−3.2

Bitte beachten Sie, dass wir den vektorisierten „oder“ Operator `|` verwenden, nicht das Basis-Python `or`. Wir müssen auch sowohl den „kleiner als“ als auch den „größer als“ Teil in Klammern setzen. Weitere Informationen finden Sie in Abschnitt Numpy.

Positionales Indizieren

Abgesehen von der Auswahl von Variablen und der Filterung nach logischen Bedingungen müssen wir gelegentlich auf Elemente nach Index oder Position zugreifen. Zum Beispiel wenn wir durch Zeilen oder Spalten iterieren wollen, wobei performantere vektorisierte Funktionen vorzuziehen sind.

Achtung: Filter erzeugen nur Ansichten

Das gefilterte Objekt ist kein neuer Datenrahmen, sondern eine *Ansicht* des Originaldatenrahmens. Dadurch wird Speicher gespart, weil insbesondere das Kopieren großer DataFrames viel Speicher und Rechenzeit verbraucht. Allerdings können dadurch später Warnungen und Fehler entstehen, wenn Sie versuchen, die gefilterten Daten zu ändern. Wenn Sie das beabsichtigen, führen Sie eine *tiefe Kopie* der Daten mit der Methode `.copy()` durch. Weitere Informationen finden Sie in Abschnitt Daten modifizieren.

Wir können auf die Werte der Serie auf zwei Arten zugreifen: nach Position und nach Index. Um auf Elemente nach Position zuzugreifen, müssen wir das Attribut `.iloc[]` verwenden, wobei das *i* für „integer“ steht. Im Gegensatz zu den meisten anderen Methoden erwartet `.iloc` Argumente in eckigen Klammern. Eine einzelne Zahl in eckigen Klammern gibt das Element als Element zurück. Z. B. eine einzelne Zeile aus unserem DataFrame

```
wetter.iloc[0]  # Gibt 1. Zeile zurück
```

```
Datum         19470101.0
Druck             1019.3
Temperatur          -2.1
Name: 0, dtype: float64
```

Wenn die Klammern eine Liste enthalten (das sieht aus wie doppelte Klammern), gibt es eine Serie zurück, die möglicherweise nur ein einzelnes Element enthält. Um also das 1. und 3. Element zu extrahieren, können wir schreiben:

```
wetter.iloc[[0,3]]   # Gibt 1. und 3. Zeile zurück
```

	Datum	Druck	Temperatur
0	19470101	1019.3	−2.1
3	19470108	1023.1	−12.9

Wir können auch einzelne Werte extrahieren indem wir die Zeile und Spaltennummer angeben.

```
wetter.iloc[0, 0]  # Gibt den Druck aus der 1. Zeile zurück
```

```
np.int64(19470101)
```

Alternativ können wir die Elemente auch nach Index extrahieren. Das ist insbesondere, dann sinnvoll, wenn man aussagekräftige Indexe verwendet. Bei unserem Beispiel empfiehlt es sich zum Beispiel die Spalte „Datum“ als Index zuzuweisen. Das geht mit der Funktion `set_index`

```
wetter.set_index("Datum", inplace=True)
wetter.head()
```

Datum	Druck	Temperatur
19470101	1019.3	−2.1
19470103	1032.4	−4.8
19470105	1031.9	−12.9
19470108	1023.1	−12.9
19470109	1017.5	−9.9

Jetzt können wir diesen Index nutzen, um auf Zeilen zuzugreifen. Dies funktioniert ähnlich, außer dass wir `.loc[]` anstelle von `.iloc[]` verwenden müssen. Die Regeln für einzelne und doppelte Klammern gelten ähnlich wie beim Zugriff über die Position.

```
wetter.loc[19470101]  # Extrahiert die Zeile für das Datum "01.01.1947"
```

```
Druck         1019.3
Temperatur      -2.1
Name: 19470101, dtype: float64
```

```
wetter.loc[[19470101, 19470103]]  # Extrahiert die Zeile für das Datum
"01.01.1947" und "03.01.1947"
```

Datum	Druck	Temperatur
19470101	1019.3	−2.1
19470103	1032.4	−4.8

```
wetter.loc[[19470101, 19470103],["Druck"]]
```

Datum	Druck
19470101	1019.3
19470103	1032.4

Die Tatsache, dass es mehrere Möglichkeiten gibt, Positionsdaten abzurufen, führt bei Anfängern oft zu Verwirrung. Dies wird durch die häufige Gewohnheit, keine Indizes zu verwenden und sich einfach auf die automatischen Zeilennummern zu verlassen, nicht erleichtert. In diesem Fall liefert der positionelle Zugriff über `.iloc[]` genau die gleichen Ergebnisse wie der Indexzugriff über `.loc[]`, und man kann bequem den Index vergessen und das verwenden, was einfacher erscheint. Manchmal ändert sich jedoch der Index als Folge bestimmter Operationen, was zu Fehlern oder unerwarteten Ergebnissen führen kann.

Daten modifizieren

Das Modifizieren von DataFrames kann auf ähnliche Weise wie das Extrahieren von Elementen erfolgen. Es gibt jedoch einige Ausnahmen und Einschränkungen. Lassen Sie uns dies anhand der Modifizierung des DataFrames für das Wetter demonstrieren.

Neue Spalten erstellen

Wir können eine einzelne Serie als `wetter.Temperatur` extrahieren, aber wenn wir eine neue Spalte (Variable) erstellen, müssen wir sie in Klammern angeben. Zum Beispiel:

```
wetter["Nullpunkt"] = 0
wetter
```

Datum	Druck	Temperatur	Nullpunkt
19470101	1019.3	−2.1	0
19470103	1032.4	−4.8	0
19470105	1031.9	−12.9	0
19470108	1023.1	−12.9	0
19470109	1017.5	−9.9	0
…	…	…	…
19470719	1009.5	17.2	0
19470720	1008.2	17.4	0
19470721	1013.5	16.8	0
19470722	1019.8	18.6	0
19470723	1019.6	19.8	0

Danach können wir auf die neue Variable als `wetter.Nullpunkt` zugreifen.

Erstellen expliziter Kopien beim Arbeiten mit gefilterten Daten

Ein typischer Datenwissenschafts-Workflow besteht aus a) Filtern von Daten auf relevante Fälle und b) Modifizieren des resultierenden Teilsatzes. Der erste Schritt beinhaltet oft das Entfernen von fehlenden Werten oder die Begrenzung der Analyse auf einen bestimmten interessanten Teilsatz. Es ist wichtig zu erkennen, dass das Filtern mit *Pandas* die interessanten Fälle nicht im Speicher kopiert, sondern möglicherweise nur eine *Ansicht* erstellt, d.h. denselben Speicherort im Computer-Speicher wiederverwendet, aber nur den Zugriff auf einen bestimmten Teil davon beschränkt. *Pandas* entscheidet in jedem Fall separat, ob eine Kopie oder eine Ansicht erstellt wird, abhängig davon, was der effizientere Ansatz ist. Dies ist eine sehr gute Idee in Bezug auf Speicherkonservierung und Vermeidung unnötiger Kopiervorgänge. Dies kann jedoch Warnungen und Fehler verursachen, wenn die gefilterten Daten später modifiziert werden. Wir demonstrieren dies am selben Datensatz.

Wählen Sie nur große Länder aus (Bevölkerung über 20 Mio):

```
warm = wetter[wetter.Temperatur > 22]
warm
```

Datum	Druck	Temperatur	Nullpunkt
19470628	1020.8	24.7	0
19470629	1019.0	27.8	0

Wir habe einen Teilmenge von Wetter erstellt, die nur die warmen Tage enthält. Versuchen wir jetzt eine neue Spalte zu dieser gefilterten Teilelemente zu erstellen, so gibt das eine Fehlermeldung

```
warm["status"] = "Warm"
```

Beachte die Warnung: *Ein Wert wird versucht, auf einer Kopie eines Slices gesetzt zu werden....* Dies sagt dir, dass das Filtern von `wetter[wetter.Temperatur > 22]` kein neues Datenrahmen erstellt hat, sondern eine Ansicht des bestehenden im Speicher, und *Pandas* ist unzufrieden mit dem Code, der nur einen Teil des ursprünglichen Datenrahmens modifiziert.

Schauen wir uns das Ergebnis in der Variable `warm` an:

```
warm.head()
```

Datum	Druck	Temperatur	Nullpunkt	status
19470628	1020.8	24.7	0	Warm
19470629	1019.0	27.8	0	Warm

Ok, dieser wurde korrekt erweitert. Der original DataFrame `wetter` hingegen nicht

```
wetter.head()
```

Datum	Druck	Temperatur	Nullpunkt
19470101	1019.3	−2.1	0
19470103	1032.4	−4.8	0
19470105	1031.9	−12.9	0
19470108	1023.1	−12.9	0
19470109	1017.5	−9.9	0

Achtung: Daten ersetzen auf Slices

Obwohl das Ergebnis hier korrekt erscheint, sollten Sie sich nicht auf diesen Ansatz verlassen! Es könnte funktionieren oder auch nicht, abhängig von der genauen Speicherstruktur des Datensatzes!

Glücklicherweise ist die Lösung sehr einfach. Wir müssen eine explizite Kopie mit der Methode `.copy` erstellen, bevor wir mit Änderungen beginnen:

```
warm = wetter[wetter.Temperatur > 22].copy()  # Neue Kopie
warm["status"] = "Warm"
warm.head()
```

Datum	Druck	Temperatur	Nullpunkt	status
19470628	1020.8	24.7	0	Warm
19470629	1019.0	27.8	0	Warm

```
wetter['status']='undefined'
wetter
```

Datum	Druck	Temperatur	Nullpunkt	status
19470101	1019.3	−2.1	0	undefined
19470103	1032.4	−4.8	0	undefined
19470105	1031.9	−12.9	0	undefined
19470108	1023.1	−12.9	0	undefined
19470109	1017.5	−9.9	0	undefined
…	…	…	…	…
19470719	1009.5	17.2	0	undefined
19470720	1008.2	17.4	0	undefined
19470721	1013.5	16.8	0	undefined
19470722	1019.8	18.6	0	undefined
19470723	1019.6	19.8	0	undefined

```
wetter.loc[wetter.Temperatur > 22, "status"] = "warm"
```

```
wetter[wetter.Temperatur > 22]
```

Datum	Druck	Temperatur	Nullpunkt	status
19470628	1020.8	24.7	0	warm
19470629	1019.0	27.8	0	warm

Jetzt funktioniert die Modifikation ohne Warnung. Sie müssen eine explizite Kopie nicht immer durchführen. Viele verschiedene Filterungsschritte lassen sich ohne `.copy` durchführen. Allerdings sollten sie auf Warnungen achten und vor Modifikationen neue Kopien erstellen.

Index anpassen

Der Index, der an Serien und Datenrahmen angehängt ist, ist potenziell ein nützliches und informatives Werkzeug. Manchmal ist er jedoch nicht sehr nützlich. Zum Beispiel, wenn Sie Daten von der Festplatte laden, wird der Index standardmäßig auf die Zeilennummer gesetzt, die selten repräsentativ für die Zeile ist. In solchen Fällen möchte man möglicherweise den Index ändern. Wenn Sie einen neuen Index erstellen möchten, können Sie ihn einfach `df.index` zuweisen. Zum Beispiel können wir einfach Ländernamen als Index unserem Datenrahmen großer Länder zuweisen:

```
warm.index = ["28.06.1947", "29.06.1947"]
warm
```

	Druck	Temperatur	Nullpunkt	status
28.06.1947	1020.8	24.7	0	Warm
29.06.1947	1019.0	27.8	0	Warm

Mit der Funktion `.reset_index()` kann ein Index entfernt werden. Dadurch wird eine neue Spalte mit den Indexwerten hinzugefügt und ein numerischer Index anhand der Zeilennummern erzeugt. Beachten Sie, dass `.reset_index()` standardmäßig einen neuen Datenrahmen zurückgibt, anstatt ihn direkt zu ändern. Wenn Sie ihn also beibehalten möchten, müssen Sie ihn entweder neu zuweisen oder `inplace=True` nutzen.

```
warm = warm.reset_index()
warm
```

	index	Druck	Temperatur	Nullpunkt	status
0	28.06.1947	1020.8	24.7	0	Warm
1	29.06.1947	1019.0	27.8	0	Warm

Die Methode `.set_index()` haben wir schon kennen gelernt um eine Spalte in einen Index umzuwandeln. Dies entfernt die Spalte „index“ aus dem Datenrahmen, da ihre Werte stattdessen im Index sein werden.

```
warm.set_index("index", inplace=True)
```

Bibliographie