Verschiedenes: Mini-Beispiele & Weiterführendes

by Anhang

```
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import scipy.stats as stats
```

Dieses Kapitel ist eine stetig wachsende Sammlung von nützlichen Mini-Beispielen und Techniken. Jedes Beispiel ist in sich abgeschlossen und kann als Referenz für häufig auftretende Aufgaben dienen.

Schleifen mit `enumerate()`

Die `enumerate()`-Funktion in Python erlaubt es, Code zu sparen und eleganter durch Sammlungen zu iterieren - gegeben, dass man sowohl Index, als auch Wert benötigt:

```
# Eine Liste mit Früchten
fruits = ["Apfel", "Banane", "Kirsche", "Dattel"]

# Mit enumerate()
print("Beispiel mit enumerate():")
for i, fruit in enumerate(fruits):

    print(f"Index {i}: {fruit}")

# Ohne enumerate()
print("\nAlternative ohne enumerate():")
for i in range(len(fruits)):
    fruit = fruits[i]
    print(f"Index {i}: {fruit}")
```

```
Beispiel mit enumerate():
Index 0: Apfel
Index 1: Banane
Index 2: Kirsche
Index 3: Dattel

Alternative ohne enumerate():
Index 0: Apfel
Index 1: Banane
Index 2: Kirsche
Index 3: Dattel
```

Die Version mit `enumerate()` ist kürzer und lesbarer, da sie direkt den Index und den Wert in einem Schritt bereitstellt, ohne dass wir eine separate Variable für den Index verwalten müssen.

DataFrame-Erstellung mit Schleifenergebnissen

Ein häufiges Szenario in der Datenanalyse ist das schrittweise Aufbauen eines DataFrames, wenn Ergebnisse aus einer Schleife gesammelt werden müssen (z.B. bei der Berechnung von

Statistiken für verschiedene Teilmengen der Daten). Hier sind drei verschiedene Ansätze mit ihren jeweiligen Vor- und Nachteilen:

```
# Beispieldaten
datasets = ["Set1", "Set2", "Set3"]
r_values = [0.123, 0.456, 0.789]
p_values = [0.01, 0.05, 0.001]
```

Ansatz 1: Zeilenweise anhängen mit `pd.concat()`

```
# Leerer DataFrame zum Start
results1 = pd.DataFrame()

for i in range(len(datasets)):
    # Temporären DataFrame mit einer Zeile erstellen
    temp_df = pd.DataFrame({
        'Dataset': [datasets[i]],
        'Pearson r': [round(r_values[i], 4)],
        'p-value': [round(p_values[i], 4)]
    })

    # An bestehenden DataFrame anhängen
    results1 = pd.concat([results1, temp_df], ignore_index=True)

print(results1)
```

```
  Dataset  Pearson r  p-value
0    Set1      0.123    0.010
1    Set2      0.456    0.050
2    Set3      0.789    0.001
```

Ohne `ignore_index=True` würden die Indexwerte des temporären DataFrames beibehalten.Dies würde zu doppelten Indexwerten führen (da in jedem Schleifendurchlauf ein neuer DataFrame mit Index 0 erstellt wird). Mit `ignore_index=True` wird stattdessen ein neuer fortlaufender Index generiert (0, 1, 2, …)-

Ansatz 2: DataFrame mit bekannter Größe vorab erstellen und befüllen

```
# DataFrame vorab mit der bekannten Anzahl an Zeilen erstellen
results2 = pd.DataFrame(columns=['Dataset', 'Pearson r', 'p-value'],
index=range(len(datasets)))

# Befüllen der Zeilen
for i in range(len(datasets)):
    results2.loc[i] = [datasets[i], round(r_values[i], 4), round(p_values[i], 4)]

print(results2)
```

```
  Dataset Pearson r p-value
0    Set1     0.123    0.01
1    Set2     0.456    0.05
2    Set3     0.789   0.001
```

Bei diesem Ansatz wird der DataFrame bereits zu Beginn mit der korrekten Anzahl an Zeilen erstellt. Im Gegensatz zum ersten Ansatz wird keine Verkettung durchgeführt, sondern bestehende Zeilen werden direkt an ihren Positionen befüllt.

Ansatz 3: Daten sammeln und am Ende einen DataFrame erstellen

```
# Liste für Datensätze vorbereiten
data_list = []

for i in range(len(datasets)):
    data_list.append({
        'Dataset': datasets[i],
        'Pearson r': round(r_values[i], 4),
        'p-value': round(p_values[i], 4)
    })

# Nach der Schleife einen DataFrame erstellen
results3 = pd.DataFrame(data_list)
print(results3)
```

```
  Dataset  Pearson r  p-value
0    Set1      0.123    0.010
1    Set2      0.456    0.050
2    Set3      0.789    0.001
```

Bewertung der Ansätze

Alle drei Methoden erzeugen das gleiche Ergebnis, unterscheiden sich aber in Bezug auf Performance und Anwendungsfall:

1. **Ansatz 1 (concat)** ist einfach zu verstehen, kann aber bei vielen Iterationen ineffizient sein, da bei jedem Schritt ein neuer DataFrame erstellt und mit dem existierenden verkettet wird, was bei großen Datenmengen zu Performanceproblemen führen kann.
2. **Ansatz 2 (vorab erstellen)** ist effizient, da die Größe vorab bekannt ist und kein wiederholtes Verketten nötig ist. Er setzt allerdings voraus, dass man die endgültige Größe des DataFrames schon vor der Schleife kennt.
3. **Ansatz 3 (Sammeln und am Ende erstellen)** ist in der Regel am effizientesten, besonders bei großen Datensätzen, da nur einmal ein DataFrame erstellt wird. Diese Methode ist besonders empfehlenswert, wenn die Anzahl der Einträge vorab nicht bekannt ist oder wenn die Daten komplex strukturiert sind.

Für die meisten Anwendungsfälle wird **Ansatz 3** empfohlen, da er eine gute Balance zwischen Lesbarkeit und Performance bietet.

Lambda-Funktionen für Inline-Operationen

Lambda-Funktionen sind kleine, anonyme Funktionen, die sich direkt dort definieren lassen, wo sie benötigt werden. Sie sind besonders nützlich, wenn man eine einfache Operation nur einmalig braucht und keine vollständige Funktionsdefinition rechtfertigt.

Ein typisches Anwendungsgebiet sind `groupby`-Operationen, bei denen verschiedene Aggregationsfunktionen unterschiedliche Nachbearbeitungen benötigen. Betrachten wir ein Experiment mit Ertragsdaten:

```
# Beispieldaten erstellen
experiment_data = {
    'duenger': ['Toad', 'Toad', 'Yoshi', 'Yoshi'] * 6,
    'sorte': ['Simba', 'Nala', 'Timon', 'Pumba'] * 6,
    'ertrag': [6192, 6269, 7470, 7862, 5522, 4504, 7260, 1594,
               7146, 6872, 7578, 6324, 5970, 5126, 6392, 1690,
               6860, 6444, 7642, 6666, 6550, 4218, 6410, 2856]
}

df_experiment = pd.DataFrame(experiment_data)
```

Ein Beispiel-Problem: Unterschiedliche Rundungsanforderungen

Wenn wir Statistiken pro Gruppe berechnen, möchten wir z.B. Mittelwert und Standardabweichung auf eine Nachkommastelle runden, die Anzahl der Beobachtungen aber als ganze Zahl belassen:

```
# Ansatz 1: Erst groupby, dann nachträglich runden (umständlich)
stats_raw = df_experiment.groupby(['duenger'])['ertrag'].agg(['mean', 'std', 'count'])
stats_rounded = stats_raw.copy()
stats_rounded['mean'] = stats_raw['mean'].round(1)
stats_rounded['std'] = stats_raw['std'].round(1)
# 'count' bleibt unverändert

stats_rounded
```

```
           mean     std  count
duenger
Toad     5972.8   944.8     12
Yoshi    5812.0  2349.5     12
```

Lösungsansatz mit eigenen Funktionen

Wir könnten auch separate Funktionen definieren:

```
# Ansatz 2: Eigene Funktionen definieren (für diesen Fall übertrieben)
def mean_rounded(x):
    return round(x.mean(), 1)

def std_rounded(x):
    return round(x.std(), 1)

stats_functions = (
    df_experiment.groupby(['duenger'])['ertrag']
    .agg(mean_rounded=mean_rounded, std_rounded=std_rounded, count='count')
)

stats_functions
```

```
         mean_rounded  std_rounded  count
duenger
Toad           5972.8        944.8     12
Yoshi          5812.0       2349.5     12
```

Die elegante Lösung: Lambda-Funktionen

Lambda-Funktionen erlauben es, die Funktion direkt an Ort und Stelle zu definieren:

```
# Ansatz 3: Lambda-Funktionen (elegant und kompakt)
stats_lambda = (
    df_experiment.groupby(['duenger'])['ertrag']
    .agg(
        mean=lambda x: round(x.mean(), 1),
        std=lambda x: round(x.std(), 1),
        count='count'
    )
)

stats_lambda
```

```
           mean     std  count
duenger
Toad     5972.8   944.8     12
Yoshi    5812.0  2349.5     12
```

Syntax und Funktionsweise

Die Syntax einer Lambda-Funktion ist:

```
lambda parameter: operation
```

In unserem Beispiel bedeutet `lambda x: round(x.mean(), 1)`:

- `x` ist der Parameter (die Gruppe von Werten)
- `round(x.mean(), 1)` ist die Operation, die ausgeführt wird
- Das Ergebnis wird automatisch zurückgegeben

Lambda-Funktionen sind besonders praktisch, wenn:

- Die Operation einfach ist
- Sie nur an einer Stelle verwendet wird
- Eine vollständige Funktionsdefinition übertrieben wäre

Für komplexere Operationen oder wiederverwendbare Logik sind reguläre Funktionen nach wie vor die bessere Wahl.

Filtern mit Masken (True/False-Arrays)

Beim Arbeiten mit NumPy-Arrays oder Pandas-DataFrames taucht oft der Begriff **Maske** auf. Eine Maske ist nichts anderes als ein Array von True/False-Werten, das festlegt, welche Elemente ausgewählt werden.

```
# Beispiel-Daten
numbers = np.array([5, 12, 18, 7, 3, 25])

# Eine Maske erstellen: Welche Zahlen sind größer als 10?
mask = numbers > 10
print("Maske:", mask)

# Mit der Maske filtern
print("Gefilterte Werte:", numbers[mask])
```

```
# Dasselbe mit einem DataFrame
df = pd.DataFrame({"Name": ["Anna", "Ben", "Clara", "David"],
                   "Alter": [23, 35, 29, 18]})

mask_df = df["Alter"] >= 30
print("\nMaske für df:\n", mask_df)

print("\nGefilterte Zeilen:\n", df[mask_df])
```

```
Maske: [False  True  True False False  True]
Gefilterte Werte: [12 18 25]

Maske für df:
 0    False
1     True
2    False
3    False
Name: Alter, dtype: bool

Gefilterte Zeilen:
   Name  Alter
1  Ben     35
```

Erklärung:

- Die Maske ist ein Array der gleichen Länge wie die Daten (True/False)
- Beim Anwenden (`numbers[mask]` oder `df[mask_df]`) werden nur die Zeilen/Elemente behalten, bei denen die Maske `True` ist

Warum nützlich?

- Sehr flexibel: Jede Bedingung kann zur Maske werden (==, >, <, Kombinationen mit & oder |)
- Lesbarer als komplizierte Schleifen
- Lässt sich auch in Plots einsetzen, um nur bestimmte Datenpunkte darzustellen

Masken im Plot

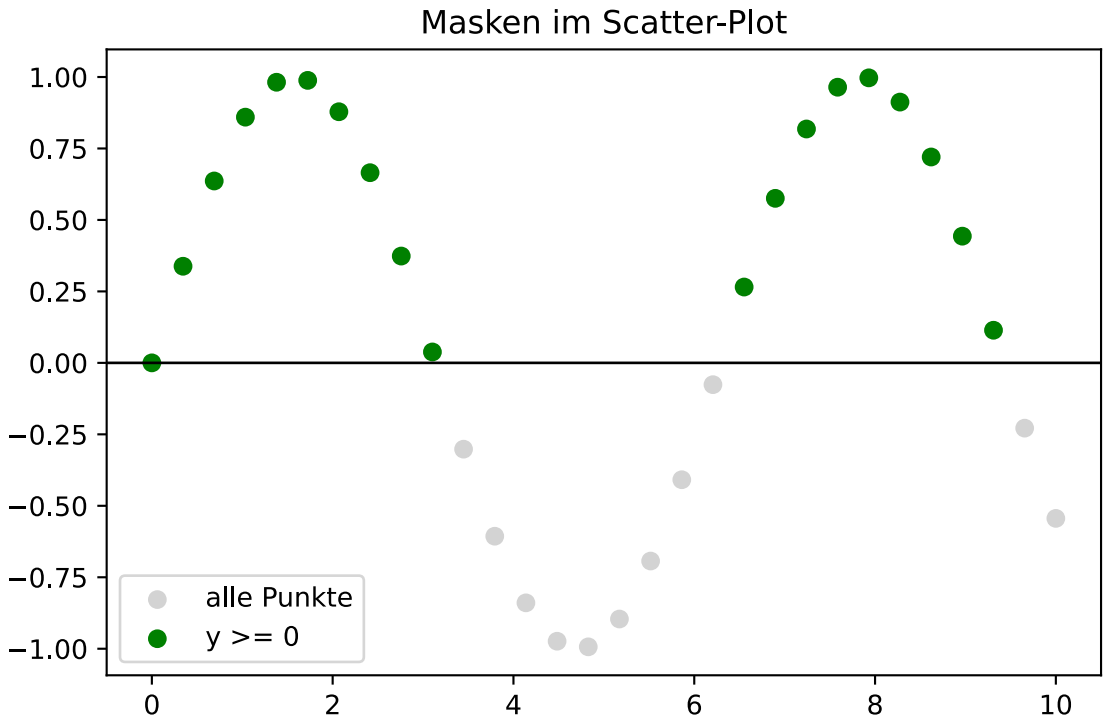
```
# Beispiel-Daten für einen Plot
x = np.linspace(0, 10, 30)
y = np.sin(x)

# Maske: Werte oberhalb der x-Achse
mask = y >= 0

# Plot: erst alle Punkte grau, dann gefilterte Punkte grün (darüber gezeichnet)
fig, ax = plt.subplots(layout='tight')
ax.scatter(x, y, color="lightgray", label="alle Punkte")
ax.scatter(x[mask], y[mask], color="green", label="y >= 0")
ax.axhline(0, color="black", linewidth=1)
ax.legend()
ax.set_title("Masken im Scatter-Plot")
plt.show()

print("=== y ===")
print(y)
print("=== mask ===")
print(mask)
```

```
print("=== y[mask] ===")
print(y[mask])
```




```
=== y ===
[ 0.          0.33803442  0.6362712   0.85959818  0.98172251  0.9882662
  0.87845883  0.6652283   0.37367879  0.03813513 -0.30189827 -0.60638843
 -0.83948697 -0.9737506  -0.99337213 -0.89604148 -0.69321762 -0.40877952
 -0.07621478  0.26532292  0.57562349  0.81815446  0.96436206  0.9970329
  0.91232056  0.72019844  0.44328555  0.11418355 -0.22836157 -0.54402111]
=== mask ===
[ True  True  True  True  True  True  True  True  True  True False False
 False False False False False False  True  True  True  True  True
  True  True  True  True False False]
=== y[mask] ===
[0.         0.33803442 0.6362712  0.85959818 0.98172251 0.9882662
 0.87845883 0.6652283  0.37367879 0.03813513 0.26532292 0.57562349
 0.81815446 0.96436206 0.9970329  0.91232056 0.72019844 0.44328555
 0.11418355]
```

In diesem Beispiel werden alle Punkte hellgrau dargestellt, die Punkte oberhalb der x-Achse (Maske `y >= 0`) zusätzlich grün hervorgehoben.

So sieht man: Masken sind nicht nur zum Filtern von Tabellen da, sondern auch ein elegantes Werkzeug für bedingte Visualisierungen.