ChatGPT

Pandas Self-Learning Module *(Selbstlerntool für Pandas)*

Einführung: Diese Lernmodule richten sich an Auszubildende Fachinformatiker in der Krankenkassen-Branche. Ziel ist es, die **Pandas**-Bibliothek – ein zentrales Werkzeug für die Datenanalyse in Python – interaktiv zu erlernen. Jede Lektion vermittelt Inhalte auf dem Niveau eines Fachinformatikers in Ausbildung und enthält abwechslungsreiche Übungsaufgaben (z.B. *Lückencode* zum Ausfüllen, Code-Umschreibungen, Quizfragen). Ein hoher Programmieranteil sorgt dafür, dass Sie vieles selbst praktisch ausprobieren können. Die verwendeten Beispieldaten sind **synthetisch**, aber realitätsnah für typische Aufgaben in Versicherungen (z.B. Analyse von Leistungsfällen, Vertragsdaten, Zeitreihen zu Kosten).

Hinweise: Um das Beste aus dem Kurs zu ziehen, sollten Sie grundlegende Python-Kenntnisse mitbringen (z.B. Variablen, Schleifen, Funktionen). Als Arbeitsumgebung empfehlen sich interaktive Notebooks (z.B. Jupyter) oder eine IDE wie PyCharm. Alle Übungen können dort ausgeführt werden. Nutzen Sie gerne öffentliche Datensätze oder firmeneigene Testdaten, um die Beispiele nachzuvollziehen – Pandas kann u.a. CSV-Dateien, Excel-Tabellen, SQL-Datenbanken oder JSON-Dateien einlesen [1].

Lektion 1: Grundlagen der Pandas-Bibliothek

In **Lektion 1** lernen Sie Pandas kennen: was es ist, wie man es installiert/importiert, und die Grundkonzepte *Series* und *DataFrame*. Dies bildet die Basis für alle weiteren Datenmanipulationen.

1.1 Einführung in Pandas (Überblick, Installation, Import)

Pandas ist eine **Open-Source-Bibliothek** für Datenanalyse und -manipulation in Python. Sie bietet leistungsfähige Datenstrukturen zur Arbeit mit strukturierten Daten, insbesondere die *Series* (1D) und das *DataFrame* (2D). Pandas basiert auf NumPy und ermöglicht effizientes Handling auch größerer Datenmengen [2]. In Data-Science-Projekten – etwa im Versicherungswesen – nutzt man Pandas, um z.B. Leistungsabrechnungen auszuwerten oder Kundendaten zu transformieren.

Installation: Falls Pandas noch nicht installiert ist, können Sie es via **pip** installieren: `pip install pandas`. (In manchen Umgebungen, z.B. Jupyter Notebooks, ist Pandas oft schon vorinstalliert.)

Import: Üblicherweise wird Pandas mit dem Alias **pd** importiert, um Schreibarbeit zu sparen [3]. Führen Sie z.B. Folgendes aus, um loszulegen:

```
import pandas as pd
```

Quiz – Importieren von Pandas: Füllen Sie den fehlenden Alias ein, der konventionell für Pandas verwendet wird:

`import pandas as [[pd]]`

(Erwartete Lösung: `pd` *)*

Wenn der Import funktioniert hat, können Sie die Version prüfen: `pd.__version__` (nützlich, um sicherzustellen, dass Sie mit einer aktuellen Version arbeiten – Pandas 2.x ist Stand 2025 aktuell).

1.2 Pandas Series und DataFrame (Grundlegende Strukturen)

Die **Series** und das **DataFrame** sind die Kern-Datenstrukturen von Pandas:

- **Series:** eindimensionale, eindxierte Datenstruktur (ähnlich einer Liste oder einem Array, aber mit Index). Eine Series kann z.B. eine Liste von Monatsumsätzen enthalten, indiziert nach Monat [4]. Stellen Sie sich eine Series als Spalte mit Werten vor, die jeweils einen Index (z.B. laufende Nummer oder Datum) haben.

- **DataFrame:** zweidimensionale Tabelle (ähnlich einer Excel-Tabelle oder SQL-Tabelle) mit Zeilen und Spalten [4]. Ein DataFrame kann z.B. eine Tabelle aller Kunden mit Spalten für Name, Alter, Versichertennummer etc. darstellen. Man kann es sich als Sammlung von Series (Spalten) mit gemeinsamem Index (Zeilen) vorstellen.

Beispiel: Erstellen wir ein kleines DataFrame mit einigen fiktiven Kundendaten einer Versicherung:

```
import pandas as pd

# Beispiel-Daten erstellen
data = {
    "KundenID": [1001, 1002, 1003],
    "Name": ["Alice Müller", "Bob Schmidt", "Clara Yilmaz"],
    "Alter": [29, 41, 35],
    "Versicherungstyp": ["Privat", "Gesetzlich", "Privat"],
}
df = pd.DataFrame(data)
print(df)
```

Wenn Sie den obigen Code ausführen, erhalten Sie ein DataFrame mit drei Zeilen und vier Spalten. Typischerweise werden die ersten und letzten Zeilen angezeigt, wenn das DataFrame groß ist – hier ist es klein genug, dass alles ausgegeben wird. Schauen Sie sich den **Index** (links, hier 0,1,2) und die **Spaltennamen** (oben) an.

Grundlegende Attribute: Jedes DataFrame hat nützliche Attribute wie `df.shape` (Dimensionen, z.B. 3 Zeilen x 4 Spalten) oder `df.columns` (Liste der Spaltennamen) [5]. Zudem besitzt jede Spalte einen Datentyp, abrufbar über `df.dtypes` [6] – in unserem Beispiel wären Alter z.B. integer (`int64`), Name string (`object` dtype in Pandas), etc. Eine Series hat analoge Attribute (`s.index`, `s.dtype` etc.).

Operationen und Methoden: Pandas stellt eine Fülle von Methoden bereit, um auf den Daten zu operieren. Einige grundlegende Beispiele:

- **Anzeigen von Daten:** `df.head()` zeigt die ersten 5 Zeilen, `df.tail()` die letzten 5 [7]. Beispiel: `df.head(2)` würde die ersten 2 Zeilen unseres DataFrames anzeigen.

- **Auswahl:** Dazu später mehr, aber kurz: `df["Name"]` würde die *Series* der Namen zurückgeben. `df.iloc[0]` liefert die erste Zeile (positionsbasiert), `df.loc[0]` ebenso in diesem Fall (indexbasiert).

- **Einfache Berechnungen:** Wenn `df2` eine numerische Spalte "Betrag" hätte, würde `df2["Betrag"].sum()` die Summe bilden, `df2["Betrag"].mean()` den Durchschnitt usw. Viele solche **Aggregatfunktionen** sind verfügbar.

- **Beschreibung:** `df.describe()` gibt eine schnelle Statistik-Übersicht über numerische Spalten [8] – Minimum, Maximum, Mittelwert, Quartile etc. (siehe Lektion 3).

Quiz – Verständnis Series vs. DataFrame: Welche Aussage trifft **nicht** zu? *(Single Choice)*

[( )] Eine Pandas Series ist im Grunde eine einzelne Spalte mit Index, während ein DataFrame mehrere Spalten (also mehrere Series) mit gemeinsamem Index enthält.
[( )] Ein DataFrame kann man sich wie eine Tabelle mit Zeilen und Spalten vorstellen.
[(X)] Eine Series darf nur numerische Daten enthalten, ein DataFrame nur Strings.
[( )] Sowohl Series als auch DataFrame basieren intern auf NumPy-Arrays.

(Lösung: Die dritte Aussage ist falsch – Series/DataFrames können beliebige Datentypen pro Spalte halten, nicht nur numerisch.)

Interaktive Übung: Erstellen Sie selbst eine Pandas Series `s_beiträge`, die die monatlichen Beitragszahlungen eines Kunden über 6 Monate enthält (z.B. Liste `[250, 250, 270, 300, 260, 250]` Euro). Lassen Sie Pandas einen default-Index vergeben. Prüfen Sie:
- `s_beiträge.index` (sollte RangeIndex 0–5 sein),
- `s_beiträge.mean()` (Durchschnittswert),
- `s_beiträge.head(3)` (die ersten 3 Einträge).

Fragen Sie sich: Ergibt der Durchschnitt rechnerisch Sinn? Wie könnten Ausreißer (falls vorhanden) diesen beeinflussen?

Qualität ok? Haben Sie ein grundlegendes Verständnis, was Pandas ist und wie Series/DataFrames funktionieren? Wenn ja, fahren wir mit der nächsten Lektion fort. Bei Unklarheiten klären Sie diese bitte, bevor Sie weitermachen.

Lektion 2: Datenmanipulation (Daten einlesen, transformieren, bereinigen)

In **Lektion 2** geht es um typische Datenmanipulations-Aufgaben mit Pandas. Dazu gehört das **Einlesen und Exportieren** von Daten, das Arbeiten mit **Spalten** (hinzufügen, entfernen, umbenennen), das **Auswählen** von Daten mittels Indexierung sowie die **Datenbereinigung** (fehlende Werte, Duplikate).

2.1 Daten einlesen und exportieren (CSV, Excel, SQL, JSON, ...)

Pandas bietet komfortable Funktionen, um Daten aus verschiedensten Quellen zu laden. Häufig genutzte Formate sind CSV, Excel, SQL-Datenbanken oder JSON. Alle entsprechenden Funktionen folgen

dem Schema `pd.read_<Format>()` für das Einlesen bzw. `df.to_<Format>()` für das Speichern [1] [9].

Beispiele Einlesen:
- CSV: `df = pd.read_csv("kunden.csv")` – liest eine CSV-Datei in ein DataFrame [10].
- Excel: `df = pd.read_excel("report.xlsx", sheet_name="Daten")` – benötigt eventuell `openpyxl` als Engine.
- SQL: `df = pd.read_sql("SELECT * FROM versicherte", con=conn)` – erfordert eine Datenbankverbindung `conn`.
- JSON: `df = pd.read_json("daten.json")` – für JSON-Dateien (stellen oft verschachtelte Strukturen dar).

Pandas unterstützt noch mehr Formate (HTML-Tabellen, Parquet, Pickle etc.), doch die obigen sind im Alltag gängig. Beachten Sie: Beim Einlesen immer einen kurzen Blick auf die Daten werfen (z.B. `df.head()`), um sicherzustellen, dass alles korrekt gelesen wurde (richtige Trennzeichen, Dezimalpunkte vs. Kommas etc.).

Beispiele Export: Analog können Sie DataFrames leicht speichern:
- `df.to_csv("output.csv", index=False)` – speichert als CSV, ohne den Index mit auszugeben.
- `df.to_excel("report.xlsx", sheet_name="Ergebnis")` – schreibt in eine Excel-Datei [11].
- `df.to_json("daten_out.json")` – exportiert als JSON-Text.
- `df.to_sql("Tabelle", con=conn, if_exists="replace")` – schreibt in eine SQL-Tabelle (über eine DB-Verbindung).

Übung: Angenommen, Sie haben ein DataFrame `claims_df` mit Leistungsfällen. Laden Sie es aus der Datei *claims.json*:

```
claims_df = pd.read_json("claims.json")
```

Überprüfen Sie anschließend mit `claims_df.info()`, ob alle Spalten korrekt eingelesen wurden (Datentypen, Anzahl Nicht-NA-Werte etc.). Exportieren Sie danach `claims_df` testweise wieder als CSV (z.B. *claims_backup.csv*).

Quiz – Lesen & Schreiben: Welche Pandas-Funktion würde man nutzen, um Daten aus einer SQL-Datenbanktabelle `Kunden` zu laden? *(Single Choice)*

[( )] `pd.read_excel("Kunden")`
[( )] `pd.read_sql_table("Kunden", con=...)`
[(X)] `pd.read_sql("SELECT * FROM Kunden", con=...)`
[( )] `pd.read_table("Kunden.sql")`

(Lösung: `pd.read_sql` *mit einem SQL-Query und Verbindungs-Objekt.)*

2.2 Spaltenoperationen: Hinzufügen, Entfernen, Umbenennen

In der Praxis müssen Datentabellen oft umgeformt werden – neue Spalten berechnen, unnötige Spalten loswerden oder Spaltennamen anpassen (z.B. für Einheitlichkeit). Pandas macht das einfach:

- **Spalte hinzufügen:** Weisen Sie einem neuen Spaltennamen einen Wert oder eine Berechnung zu. Beispiel: `df["Beitrag_monatlich"] = df["Jahresbeitrag"] / 12` würde aus einem jährlichen Beitrag eine monatliche Rate berechnen und als neue Spalte anfügen. Wenn Sie stattdessen allen Zeilen einen festen Wert geben wollen: `df["Land"] = "Deutschland"` fügt eine gleichbleibende Spalte hinzu.

- **Spalte entfernen:** Dafür gibt es `df.drop(columns=["SpalteA", ...])`. Beispiel: `df = df.drop(columns=["Name"])` entfernt die Spalte *Name*. Alternativ kann man auch mit `del df["Name"]` eine einzelne Spalte löschen. Achten Sie darauf: `drop()` gibt per Default ein neues DataFrame zurück – für inplace-Löschen `df.drop(..., inplace=True)` setzen oder Zuweisung wie gezeigt verwenden.

- **Spalten umbenennen:** Nutzen Sie `df.rename()`. Entweder per Dictionary: `df.rename(columns={"Versicherungstyp": "Tarifart"}, inplace=True)` um z.B. *Versicherungstyp* in *Tarifart* umzubenennen [12]. Oder Sie setzen direkt die `df.columns` neu, was sich aber nur anbietet, wenn Sie *alle* Spaltennamen ändern möchten (z.B. `df.columns = ["A","B","C", ...]` mit korrekter Länge der Liste).

Übung: Im DataFrame `df` aus 2.1 (Kundendaten) möchten wir einen jährlichen Versicherungsbeitrag simulieren. Fügen Sie eine neue Spalte `"Jahresbeitrag"` hinzu, die z.B. für *Privat*-Versicherte 1200 und für *Gesetzlich*-Versicherte 800 sein soll. *(Tipp: Verwenden Sie* `df["Versicherungstyp"]` *und eine Bedingung oder Mapping: z.B.* `df["Versicherungstyp"] == "Privat"` *ergibt eine boolesche Series, die Sie für Auswahl nutzen können.)* Entfernen Sie dann die Spalte `"Alter"` aus dem DataFrame (vielleicht braucht man sie nicht für die nächste Analyse). Benennen Sie anschließend `"Name"` um in `"Kunde"`. Überprüfen Sie zum Schluss mit `print(df.columns)`, ob die Änderungen geklappt haben.

Lückencode: Füllen Sie die Lücken aus, um im DataFrame `df` alle Spaltennamen in Kleinbuchstaben umzubenennen:

```
# Alle Spaltennamen in Kleinbuchstaben
df.columns = [col.<<lower>>() for col in df.columns]
```

(Lösung: `lower`*. Hier wird eine Listen-Komprehension genutzt, die jedes Element von* `df.columns` *(Spaltenname) nimmt und* `.lower()` *darauf anwendet.)*

2.3 Index-Management: Auswahl mit iloc und loc

Jede DataFrame-Zeile hat einen Index. Standardmäßig ist das ein fortlaufender Integer-Index von 0 an. Man kann aber auch andere Indexe haben (z.B. eine Kundennummer-Spalte als Index setzen). Für die **Daten-Auswahl** sind besonders zwei Eigenschaften wichtig:

- `.iloc` – **Integer Position** based selection. Damit wählt man nach Positionen (ähnlich wie bei Listen: 0 bis n-1) [13]. Z.B. `df.iloc[2]` gibt die dritte Zeile zurück (Index 2). `df.iloc[:5, :`

`3]` gibt einen Ausschnitt – die ersten 5 Zeilen und ersten 3 Spalten. `.iloc` erwartet also immer numerische Indizes bzw. Slices. Es ist *positionsbasiert*.

- `.loc` – **Label** based selection. Damit wählt man nach Label, also dem Indexwert (bzw. bei Spalten auch dem Spaltennamen) [14]. Z.B. `df.loc[1002]` würde die Zeile mit Index *1002* suchen (angenommen, wir haben die KundenID 1001,1002,1003 als Index gesetzt). `df.loc[df["Alter"] > 30, "Name"]` würde für alle mit Alter >30 den Namen liefern – hier nutzen wir einen **booleschen Index** (Maske). `.loc` kann also auch mit booleans arbeiten, was für **Filterung** sehr praktisch ist. Wichtig: Wenn Ihr Index numerisch ist (z.B. 0,1,2,...), dann ist `df.loc[2]` nicht die dritte Zeile per se, sondern die Zeile mit Label "2" im Index – bei Standardindex fällt das zufällig zusammen, aber wenn Sie mal einen anderen Index haben, achten Sie darauf [15].

Beispiel: Nach unserem `df` (Kundendaten) von oben:
- `df.iloc[0]` – gibt die erste Zeile (Alice Müller...).
- `df.loc[1002]` – funktioniert *nicht*, solange der Index 0,1,2 ist (KeyError). Wir könnten aber z.B. `df.set_index("KundenID", inplace=True)` machen, dann wäre `df.loc[1002]` die Zeile von Bob Schmidt.
- `df.loc[df["Alter"] > 30, ["Name", "Versicherungstyp"]]` – gibt Name und Versicherungstyp aller Kunden über 30 Jahren (hier Bob und Clara).

Übung: Sie haben ein DataFrame `claims_df` mit einem numerischen Index und Spalten `"Betrag"` (Schadenshöhe) und `"Status"` (offen/geschlossen). Führen Sie folgende Auswahlen durch:
1. Wählen Sie die ersten 10 Zeilen und Spalten Betrag+Status aus (Tipp: `claims_df.iloc[...]`).
2. Wählen Sie alle *offenen* (Status == "offen") Fälle aus und erhalten Sie nur die Beträge (Tipp: boolesche Maske mit `.loc[...]`).
3. Setzen Sie den Index von `claims_df` auf `"ClaimID"` (Angenommen, es gibt eine Spalte ClaimID). Prüfen Sie dann, dass `claims_df.loc[12345]` funktioniert (wenn 12345 eine vorhandene ClaimID ist).

Quiz – iloc vs. loc: Welche Aussage ist korrekt? *(Single Choice)*

[( )] `.loc` wählt immer nur zusammenhängende Zeilenbereiche aus, `.iloc` kann auch einzelne auswählen.
[(X)] `.loc` nutzt die Label der Indexe (bzw. Spaltennamen) zur Selektion, `.iloc` nutzt die numerische Position der Elemente [13] [16].
[( )] Mit `.iloc` kann man keine Spalten auswählen, nur Zeilen.
[( )] `.loc` erlaubt keine Verwendung von booleschen Bedingungen.

(Lösung: Die zweite Aussage ist richtig.)

2.4 Datenbereinigung: Fehlende Werte (NA/NaN) behandeln, Duplikate entfernen

Datenbereinigung ist in jedem Daten-Projekt wichtig. Echte Datensätze enthalten oft fehlende Werte (*NA/NaN* in Pandas) oder doppelte Einträge, die man bereinigen muss.

- **Fehlende Werte finden:** Pandas kennzeichnet fehlende Werte meist als `NaN` (`numpy.nan`). Mit `df.isna()` oder `df.isnull()` erhalten Sie ein DataFrame gleicher Form mit True/False-Markierungen, wo Werte fehlen. Praktischer ist oft `df.isna().sum()`, was pro Spalte zählt, wie viele NAs vorkommen.

- **Entfernen fehlender Werte:** Mit `df.dropna()` können Sie fehlende Werte ausmerzen. Standardmäßig werden alle **Zeilen** entfernt, in denen *irgendein* NA vorkommt [17]. Sie können aber Parameter angeben, z.B. `df.dropna(how="all")` würde nur Zeilen droppen, die komplett leer sind, oder `df.dropna(subset=["Spalte1","Spalte2"])` entfernt nur, wenn in bestimmten Spalten NAs sind. Auch können Sie `axis=1` setzen, um *Spalten* mit NAs zu entfernen (wenn z.B. eine ganze Spalte unbrauchbar viele NAs hat) [18].

- **Auffüllen fehlender Werte:** Mit `df.fillna()` können Sie NAs ersetzen [19]. Oft möchte man fehlende numerische Werte durch einen sinnvollen Ersatz füllen (Imputation) – z.B. 0, einen Spaltenmittelwert, oder einen Vorwert. Beispiele: `df["Einkommen"].fillna(0, inplace=True)` würde Lücken in *Einkommen* mit 0 füllen. Oder `df["Alter"].fillna(df["Alter"].median(), inplace=True)` füllt mit dem Medianalter. Für Zeitreihendaten gibt es auch Methoden wie `fillna(method="ffill")` (forward fill = mit letztem bekannten Wert auffüllen) oder `bfill` (backward fill). Wichtig: Prüfen Sie nach dem Füllen, ob noch NAs übrig sind.

- **Duplikate finden/entfernen:** Doppelte Einträge (Zeilen) entdeckt man mit `df.duplicated()`, was eine boolesche Series zurückgibt (True für Duplikate außer dem ersten Vorkommen). Mit `df.drop_duplicates()` können Sie Duplikate entfernen. Auch hier lassen sich Parameter setzen, z.B. `subset` (worauf sollen Duplikate geprüft werden) oder `keep='last'` (um statt das erste, das letzte zu behalten). Beispiel: `kunden_df.drop_duplicates(subset=["Name", "Geburtsdatum"], keep="first")` könnte doppelte Kunden entfernen, die vielleicht versehentlich doppelt erfasst wurden.

Beispiel & Übung: Angenommen, `df` enthält eine Spalte *E-Mail*. Sie stellen fest, dass einige Kunden keine E-Mail hinterlegt haben (NaN). Außerdem tauchen einige Kunden doppelt auf. Schreiben Sie Code, der:
a) die Anzahl fehlender E-Mails ausgibt (`df["E-Mail"].isna().sum()`).
b) die Zeilen mit fehlender E-Mail entfernt **oder** alternativ die fehlenden E-Mails mit einem Platzhalter füllt (z.B. `"keine@angabe.de"`), je nachdem, was in Ihrem Kontext sinnvoller ist.
c) doppelte Kunden entfernt, wobei angenommen wird, dass Duplikate exakt gleiche Werte in allen Spalten haben. (Tipp: `df.drop_duplicates(inplace=True)` ohne Subset entfernt exakte Duplikate.)

Überprüfen Sie nach jedem Schritt die Daten mit `df.info()` oder einem geeigneten Aufruf, um sicherzugehen, dass die Bereinigung wie erwartet verlief.

Quiz – Fehlende Werte: Was bewirkt `df.dropna(axis=1, how="all")`? *(Single Choice)*

[(X)] Entfernt alle Spalten, in denen **ausschließlich** fehlende Werte stehen [17] [20].
[( )] Entfernt alle Zeilen, in denen mindestens ein Wert fehlt.
[( )] Entfernt alle Spalten, in denen mindestens ein Wert fehlt.
[( )] Entfernt alle Zeilen, in denen **alle** Werte fehlen.

(Lösung: axis=1 means operation on columns, how="all" means drop columns that are all NA.)

Qualität ok? Haben Sie verstanden, wie man Daten in Pandas ein- und auslädt, Spalten manipuliert, gezielt Datenausschnitte auswählt und typische Datenbereinigungen vornimmt? Wenn ja, geht es weiter zu den Datenanalyse-Techniken. Falls nicht, nutzen Sie die Gelegenheit, offene Fragen mit zusätzlichen Übungen oder in der Dokumentation zu klären.

Lektion 3: Datenanalyse-Techniken

Diese Lektion behandelt wichtige Techniken zur Datenanalyse mit Pandas. Dazu zählen das Gewinnen eines **Überblicks** über den Datensatz, das Arbeiten mit **Datentypen**, **Filterung und Sortierung** von Daten, **Gruppierung und Aggregation** für Auswertungen, das **Kombinieren** von Daten mittels Merge/Join sowie eine Einführung in **Zeitreihenanalyse** (Datum/Zeit-Funktionen und Resampling).

3.1 Datenüberblick: Schnelle Statistiken mit describe()

Gerade bei numerischen Daten ist es hilfreich, zunächst einige **deskriptive Statistiken** zu betrachten. Pandas bietet dafür `df.describe()` an. Diese Methode gibt für alle numerischen Spalten eines DataFrames Kennzahlen wie Mittelwert, Standardabweichung, Minimum, Maximum sowie Quartilswerte aus [8]. Für kategoriale Spalten kann man `df.describe(include="object")` aufrufen, was z.B. die Anzahl eindeutiger Werte, häufigster Wert etc. liefert.

Beispiel: Angenommen, `claims_df` enthält eine Spalte *Betrag* mit den Kosten von Leistungsfällen. `claims_df["Betrag"].describe()` könnte folgende Ausgabe zeigen (fiktive Werte): count=500, mean=250.4, std=100.7, min=20.0, 25%=180.0, 50%=240.0, 75%=300.0, max=600.0. Daraus lesen wir: Es gibt 500 Fälle, der Durchschnitt liegt bei 250€, der höchste Fall kostet 600€ usw. So gewinnt man schnell ein Gefühl für die Verteilung.

Zusätzlich nützlich: `df.info()` (zeigt Zeilenanzahl, Spaltenanzahl, Datentypen, Speicherbedarf) und `df.head()/df.tail()` zur Ansicht, hatten wir bereits erwähnt.

Übung: Laden Sie einen Datensatz (z.B. *Krankenkassen.csv* mit synthetischen Versichertendaten). Lassen Sie `df.describe()` laufen. Wie unterscheiden sich die Statistiken zwischen z.B. *Alter* der Versicherten und *Monatsbeitrag*? Nutzen Sie auch `df["Alter"].value_counts()` um zu sehen, welche Alter häufig vertreten sind, oder `df["Tarif"].value_counts(normalize=True)` um prozentuale Anteile der Tarifarten zu sehen.

3.2 Datentypen: Überblick und Konvertierung

Jede Spalte in Pandas hat einen **Datentyp** (*dtype*), der bestimmt, welche Operationen darauf möglich sind (z.B. numerisch vs. Text) und wie effizient der Speicher genutzt wird. Typische Pandas-Datentypen sind `int64`, `float64`, `object` (für Strings oder gemischte Inhalte), `bool`, `datetime64[ns]` für Zeitstempel, `category` für kategorische Daten, etc.

Mit `df.dtypes` sehen Sie die Typen jeder Spalte. Manchmal erkennt Pandas beim Einlesen einen Typ nicht optimal (z.B. Postleitzahlen könnten als int statt string gelesen werden, oder Zahlen mit NA als float). Dann möchte man konvertieren:

- **Konvertierung mit astype():** Verwenden Sie `df["Spalte"] = df["Spalte"].astype(neuer_typ)`. Beispiel: Eine Spalte *Versichertennummer* besteht aus Ziffern, aber wir wollen sie als String behandeln (weil keine Rechenoperation sinnvoll ist): `df["Versichertennummer"] = df["Versichertennummer"].astype(str)`. Oder: Eine Spalte *Kategorie* mit wenigen Ausprägungen kann man in *category* konvertieren, was Speicher spart: `df["Tarif"] = df["Tarif"].astype("category")`.

- **Datetime konvertieren:** Für Datumsangaben nutzen Sie `pd.to_datetime()`. Beispiel: `df["Geburtsdatum"] = pd.to_datetime(df["Geburtsdatum"])`. Danach ist der dtype `datetime64[ns]`. Ähnlich gibt es `pd.to_numeric`, `pd.to_timedelta` etc.

- **astype bei numerischen Werten:** z.B. float zu int: `df["Alter"] = df["Alter"].astype(int)` (vorsicht, wenn NaNs drin sind, geht das nicht ohne weiteres, da NaN floats). Oder um Speicher zu optimieren: `astype("float32")` um die Hälfte des Speicherbedarfs zu brauchen, wenn die Präzision reicht.

Übung: Im DataFrame `df` haben Sie die Spalte *Mitglied_seit* als String (z.B. "2020-05-01"). Konvertieren Sie diese mit `pd.to_datetime` in echte Datumswerte. Überprüfen Sie mit `df["Mitglied_seit"].dt.year.value_counts()`, welche Jahre vertreten sind. Konvertieren Sie außerdem die Spalte *Versichertennummer* (numerisch eingelesen) in String. Und falls eine Spalte *Tarifart* vorliegt mit wenigen Kategorien, wandeln Sie diese in den Typ *category* um. Kontrollieren Sie jeweils mit `df.dtypes`, ob die Änderung erfolgreich war.

Quiz – Datentypen: Was trifft auf Pandas-Datentypen zu? *(Multiple Choice)*

[[X]] Pandas verwendet für viele Datentypen NumPy-Typen (z.B. int64, float64) im Hintergrund.
[[X]] Die Umwandlung einer Objekt-Spalte in den Category-Typ kann Speicher sparen, wenn nur wenige verschiedene Werte in der Spalte vorkommen.
[[ ]] Ein DataFrame kann pro Spalte mehrere Datentypen haben.
[[X]] Für Datum/Uhrzeit bietet Pandas den speziellen dtype datetime64[ns] und Hilfsfunktionen wie pd.to_datetime().

(Lösung: Richtig sind 1,2,4. Eine Spalte kann immer nur einen dtype haben, nicht mehrere.)

3.3 Filterung und Sortierung: Daten filtern, sort_values(), query()

Die Fähigkeit, nur bestimmte Teilmengen der Daten anzuschauen, ist essenziell. **Filtern** hatten wir schon bei `.loc` mit booleschen Bedingungen gesehen. Recap: `df[df["Spalte"] > 5]` gibt alle Zeilen, wo die Bedingung wahr ist. Das ist syntaktisch eine Kurzform für `df.loc[df["Spalte"] > 5]`. Mehrere Bedingungen verknüpft man mit `&` (UND) oder `|` (ODER), wobei man jede Teilbedingung in Klammern setzt: z.B. `df[(df["Alter"]>30) & (df["Tarif"]=="Privat")]`.

Alternativ: .query()-Methode: Pandas bietet `df.query("Bedingung")` als etwas bequemere Syntax an, besonders wenn Spaltennamen Leerzeichen enthalten. Beispiel: `df.query("Alter > 30 and Tarif == 'Privat'")` würde dasselbe liefern wie obiges Beispiel [21]. Der Vorteil ist oft die Lesbarkeit, Nachteil kann sein, dass manche Python-Ausdrücke nicht direkt gehen (query hat eine eigene Syntax und kann z.B. keine lokalen Python-Variablen ohne Weiteres sehen, außer man nutzt @-Notation). Für einfache Filter ist es aber sehr praktisch.

Sortierung: Mit `df.sort_values(by="Spalte", ascending=True/False)` sortieren Sie das DataFrame nach einer oder mehreren Spalten. Beispiel: `claims_df.sort_values(by="Betrag", ascending=False)` sortiert Leistungsfälle absteigend nach Betrag (höchster Schaden zuerst). Mehrere Spalten: `df.sort_values(["Tarif","Alter"])` würde erst nach Tarif, dann innerhalb gleicher Tarife nach Alter sortieren.

Beispiele: - Filtern: `hohe_ansprüche = claims_df[claims_df["Betrag"] > 10000]` – alle Schadensfälle über 10k€.
- Filtern mit mehreren Bedingungen: `senioren_priv = df[(df["Alter"] > 65) & (df["Versicherungstyp"]=="Privat")]` – alle Privatversicherten über 65 Jahre.
- Sortieren: `df.sort_values(by=["Versicherungstyp","Alter"], ascending=[True, False])` – sortiert erst nach Versicherungstyp (Gesetzlich vor Privat, da alphabetisch G < P), und innerhalb beider Gruppen absteigend nach Alter (d.h. älteste zuerst).

Übung: Greifen wir das vorherige Beispiel `df` (Kundendaten) auf. Filtern Sie alle Kunden, die *jünger als 30* sind **oder** einen *Privat-Tarif* haben. (Hinweis: OR-Bedingung mit `|`). Speichern Sie diese Teilmenge in `df_jung_oder_privat`. Sortieren Sie dann `df_jung_oder_privat` nach *Tarifart* (aufsteigend) und *Alter* (absteigend). Was fällt Ihnen auf? (z.B. welche Gruppe ist am stärksten vertreten?)

Testen Sie auch `df.query(...)` für dieselbe Abfrage:

```
df.query("Alter < 30 or Versicherungstyp == 'Privat'")
```

Das Ergebnis sollte mit `df_jung_oder_privat` übereinstimmen.

3.4 Gruppierung und Aggregation: groupby(), sum, mean, count, etc.

Um **Daten auszuwerten**, ist es oft nötig, nach bestimmten Kategorien zu gruppieren und Kennzahlen zu berechnen. In Pandas erledigt das `groupby()` kombiniert mit Aggregationsfunktionen. Dies folgt dem Prinzip "*Split-Apply-Combine*": Daten in Gruppen aufteilen, Funktion auf jede Gruppe anwenden, Ergebnisse zusammenführen [22] .

Beispiel-Szenario: Stellen wir uns vor, wir haben eine Tabelle aller Leistungsfälle (`claims_df`) mit Spalten: *Art* (z.B. "Zahnarzt", "Krankenhaus", "Rezept"), *Betrag*, *Erstattet_von* (Kasse X, Y, Z). Wir möchten wissen, **pro Art** die *Anzahl* der Fälle und die *Summe der Beträge*.

Mit Pandas: `claims_df.groupby("Art")["Betrag"].sum()` würde pro Art die Gesamtkosten summieren. Statt nur *Betrag* können wir auch mehrere Spalten aggregieren oder unterschiedliche Aggregationen:

```
claims_df.groupby("Art").agg(
    anzahl_faelle=("Art", "count"),
    summe_kosten=("Betrag", "sum"),
    durchschnitt_kosten=("Betrag", "mean")
)
```

Der obige Code liefert ein DataFrame mit index=Art und drei neuen Spalten: anzahl_faelle, summe_kosten, durchschnitt_kosten. `("Art","count")` zählt wie oft jede *Art* vorkommt (auch `count()` auf *Betrag* würde gehen, außer bei NaNs Unterschied – count auf Art zählt die Zeilen direkt). `("Betrag","sum")` summiert die Beträge je Gruppe, `("Betrag","mean")` mittelt sie. So erhält man schnell aggregierte Statistiken je Kategorie.

Weitere Aggregationsfunktionen: sum, mean, count, median, min, max, std (Standardabweichung), var (Varianz), first, last, etc. Man kann eigene Funktionen per `.agg(func)` oder `.apply(func)` ebenfalls anwenden.

Mehrere Gruppierungsstufen: `groupby(["Art","Erstattet_von"])` würde z.B. erst nach Art und dann innerhalb der Art nach Kasse gruppieren – das Ergebnis hat dann einen **MultiIndex** (dazu mehr in Lektion 4), z.B. Indexstufe1=Art, Stufe2=Kasse. Man kann so verschachtelte Gruppierungen vornehmen.

Übung: Nehmen wir unser `df` (Kundendaten) erweitert um eine Spalte "Region" (z.B. Bundesland oder PLZ-Gebiet). Finden Sie heraus, wie viele Kunden pro Region vorhanden sind *und* wie das durchschnittliche Alter pro Region ist. (Tipp: groupby("Region").agg(anzahl=("Kunde","count"), mittel_alter=("Alter","mean"))). Welche Region hat die meisten Kunden?

Erweitern Sie die Analyse: gruppieren Sie zusätzlich nach *Versicherungstyp*: also `groupby(["Region","Versicherungstyp"])` und zählen Sie die Kunden. Dies liefert pro Region getrennt für Gesetzlich/Privat die Zahlen – gibt es Regionen, in denen z.B. besonders viele Privatversicherte sind?

Quiz – groupby: Welche Aussage beschreibt das `groupby()`-Konzept korrekt? *(Single Choice)*

[(X)] *Split-Apply-Combine*: Daten nach einem Kriterium aufteilen, je Gruppe Funktion berechnen, Ergebnisse zusammenführen [22].
[( )] Alle Daten werden gefiltert, bevor eine Funktion angewandt wird.
[( )] `groupby` kann nur mit einer einzigen Aggregatfunktion verwendet werden.
[( )] Man muss nach `groupby` immer `.reset_index()` aufrufen, sonst bekommt man keinen DataFrame.

(Lösung: Die Split-Apply-Combine-Beschreibung ist richtig. Man kann natürlich mehrere Aggregatfunktionen verwenden und reset_index ist optional – nur nötig, wenn man das Gruppierungsergebnis wieder als flaches DataFrame haben möchte.)

3.5 Merge/Join-Operationen: Daten kombinieren mit concat() und merge()

In der Praxis liegen Daten nicht immer in **einer** Tabelle vor. Man hat z.B. separate Tabellen für Kundenstammdaten und Leistungsdaten. Pandas erlaubt, Daten ähnlich wie in SQL-Joins zu kombinieren.

- **concat():** Zum **Aneinanderhängen** von Daten (vertikal oder horizontal). Vertikal (Zeilen anhängen): z.B. `df_gesamt = pd.concat([df_2024, df_2025])` um zwei Jahres-Datensätze untereinander zu stellen. Dabei sollten die Spalten übereinstimmen. Horizontale Verkettung (Spalten nebeneinander) geht mit `axis=1`, wird aber seltener genutzt (dafür eher merge). `concat` kann auch eine Liste von DataFrames aufnehmen und hat Parameter wie `ignore_index=True` (um Index neu zu vergeben) usw.

- **merge():** Für **Joins** nach Schlüsselspalten (wie Datenbank-Joins). Standardaufruf: `pd.merge(left_df, right_df, on="gemeinsame_Spalte", how="inner")`. Der `how`-Parameter steuert den Jointyp: `"inner"` (nur übereinstimmende Werte, Schnittmenge), `"left"` (alle Zeilen aus left_df, falls kein Match bei right -> NaN), `"right"`, `"outer"` (Vereinigungsmenge aller). Man kann auch unterschiedliche Schlüsselnamen angeben mit `left_on` / `right_on`. Beispiel: `kunden_df.merge(vertraege_df, on="KundenID",`

`how="left")` würde jedem Kunden seine Vertragsdaten anheften (wenn `vertraege_df` z.B. Spalten KundeID, VertragsID, Beginn, etc. hat). Kunden ohne Vertrag hätten dann NaNs in den Vertrags-Spalten bei left-join.

- **join():** DataFrame-Methode `df1.join(df2)` ist ähnlich wie merge, verwendet standardmäßig die Indexe zum Joinen. Das kann bequem sein, wenn man z.B. einen DataFrame nach Index sortiert hat und einen anderen mit identischen Index-Werten hat.

Beispiel: `policen_df` (Vertragsinformationen) und `kunden_df` (Kundeninfos) sollen kombiniert werden. Beide haben die Spalte *KundenID*. Mit `ges_df = pd.merge(kunden_df, policen_df, on="KundenID", how="inner")` erhält man ein DataFrame, das alle Kunden enthält, die einen Vertrag haben (inner join), mit allen Informationen aus beiden Tabellen. Hat ein Kunde mehrere Verträge, würde der Kunde mehrfach auftauchen (one-to-many Beziehung). Umgekehrt, ein left-join von kunden auf policen würde alle Kunden behalten, auch jene ohne Vertrag, aber mit NaN in Vertragsfeldern.

Übung: Sie haben `kunden_df` (KundenID, Name, Geburtsdatum) und `leistungen_df` (LeistungID, KundenID, Betrag, Datum). Verbinden Sie die beiden, um ein kombiniertes DataFrame zu erhalten, das pro Leistung auch den Kundennamen und das Geburtsdatum enthält. Verwenden Sie einen inner join, da Sie nur Leistungen mit gültiger KundenID betrachten wollen:

```
kombiniert_df = pd.merge(leistungen_df, kunden_df, on="KundenID",
how="inner")
```

Überprüfen Sie, wie viele Zeilen `kombiniert_df` hat im Vergleich zu `leistungen_df` – stimmen diese überein? (Sollten gleich sein, wenn jede Leistung einen existierenden Kunden hat. Falls nicht, gab es Leistungen mit unbekannter KundenID.)

Testen Sie auch einen `how="outer"`-Join zwischen beiden DataFrames: Wieviele Zeilen ergeben sich da? (Dies würde auch Kunden ohne Leistungen und Leistungen ohne gültigen Kunden zeigen, sofern solche existieren.) Nutzen Sie zur Kontrolle vielleicht `indikator = True` in merge (`pd.merge(..., indicator=True)`) – dann entsteht eine Spalte, die zeigt, aus welcher Tabelle der Datensatz stammt (left_only, right_only, both).

3.6 Zeitreihenanalyse: Datums-/Zeitmanipulation und Resampling

Gerade im Versicherungsbereich sind **Zeitreihen** wichtig – z.B. monatliche Beitragszahlungen, jährliche Kostenentwicklung, etc. Pandas bietet spezielle Funktionen für das Arbeiten mit Zeitdaten.

Datumsfunktionen: Sobald eine Spalte als datetime64 dtype vorliegt (siehe 3.2), können Sie bequem damit arbeiten. `df["Datum"].dt.year` gibt z.B. das Jahr jedes Datums, `.dt.month` den Monat. Man kann filtern wie `df[df["Datum"] >= "2023-01-01"]` (string wird zu Datum geparst). Außerdem: `pd.date_range()` erstellt Datumsreihen (z.B. alle Monate eines Jahres). Für Zeitabstände gibt es `pd.Timedelta` oder `pd.to_timedelta` (z.B. um Differenzen zu berechnen).

Resampling: Sehr mächtig ist `df.resample()` für Zeitreihendaten mit Datetime-Index. Damit kann man die **Frequenz ändern**, z.B. tägliche Daten auf Monats- oder Jahresdaten aggregieren [23]. Voraussetzung: Der DataFrame ist nach Datum indiziert (oder man gibt `on="Datumsspalte"` an). Beispiel: Wenn `claims_ts` einen DatetimeIndex hat und im Wert die täglichen Ausgaben, dann liefert

`claims_ts.resample('M').sum()` die Summen pro Monat. Resample-Code `'M'` steht für Monatsende, `'MS'` für Monatsstart; analog `'D'` Tag, `'Q'` Quartal, `'A'` Jahr etc. Man kann auch andere Aggregationen anwenden (ähnlich groupby).

Beispiel: Wir haben einen DataFrame `zahlungen_df` mit Spalten *Datum* (Beitrags-Einzugstag) und *Betrag* (eingezogener Betrag) pro Einzug. Wir möchten monatliche Gesamtbeträge. Vorgehen: `zahlungen_df["Datum"] = pd.to_datetime(zahlungen_df["Datum"]); zahlungen_df.set_index("Datum", inplace=True)`. Dann: `monats_summen = zahlungen_df["Betrag"].resample('M').sum()`. Das Ergebnis ist eine **Zeitreihe** (Series) der monatlichen Summe. Genauso könnte man `mean()` oder `count()` nutzen. Für gleitende Zeitfenster gibt es `.rolling()` (z.B. 7-Tage-Durchschnitt etc.), aber das führt hier zu weit.

Übung: Aus dem `leistungen_df` haben wir das Datum jeder Leistung. Finden Sie heraus, wie viele Leistungsfälle *pro Quartal* angefallen sind. (Tipp: Erzeugen Sie zuerst ggf. einen DatetimeIndex oder sortieren nach Datum. Nutzen Sie dann `resample('Q').count()` auf der Spalte LeistungID oder einer anderen Spalte.) Plotten Sie das Ergebnis mit `plot()`, um einen Trend über die Quartale zu sehen.

Erstellen Sie eine neue Series `neue_mitglieder` mit fiktiven Eintritten pro Monat: z.B. Jan:100, Feb: 120, Mar:90, Apr:130, Mai:110 (nehmen wir an, 2024). Nutzen Sie `pd.date_range` mit Frequenz 'M' für die Indexerstellung. Resampeln Sie diese Zeitreihe auf Quartalsbasis mit `resample('Q').sum()` – entspricht das der Summe der Eintritte pro Quartal (Jan-Mar, Apr-Jun,...)?

Quiz – Resampling: Was ist Voraussetzung, damit `df.resample('M')` sinnvoll funktioniert? *(Single Choice)*

[( )] Das DataFrame muss genau eine Spalte haben.
[(X)] Das DataFrame (oder die Ziel-Series) muss einen DatetimeIndex besitzen (oder ein `on`-Parameter mit Datetime-Spalte muss angegeben sein) [24].
[( )] Man muss vorher `df.sort_values(by="Datum")` aufrufen, sonst schlägt resample fehl.
[( )] Resampling funktioniert nur mit *datetime64[ns]*, nicht mit *datetime.date*.

(Lösung: Der DatetimeIndex ist entscheidend. Sortierung ist zwar empfehlenswert, aber Pandas resample kommt i.d.R. auch mit unsortierten Indizes zurecht, sortiert intern. Resample auf datetime.date könnte tatsächlich konvertieren, aber Hauptpunkt ist Index.)

Qualität ok? Fühlen Sie sich sicher im Umgang mit beschreibenden Statistiken, Datentypen-Konvertierung, Filter/Sortierung, Gruppierung/Aggregation, dem Mergen von Daten sowie grundlegender Zeitreihenmanipulation? Diese Fähigkeiten erlauben es Ihnen, aus Rohdaten sinnvolle Informationen zu gewinnen. Im nächsten Abschnitt gehen wir auf fortgeschrittenere Manipulationen ein.

Lektion 4: Erweiterte Datenmanipulation (Fortgeschrittene Techniken)

In Lektion 4 behandeln wir einige weitere nützliche Techniken: **Textdatenverarbeitung** mit Pandas, Arbeiten mit **hierarchischen Indizes (MultiIndex)** sowie den Einsatz von **Lambda-Funktionen** mit `apply` und `map` für flexible Berechnungen.

4.1 Verarbeitung von Textdaten: String-Methoden mit .str

Viele Daten (z.B. Namen, Adressen, Codes) sind **Text**. Pandas Series vom Typ *object* (oder dem neueren *string*-Dtype) besitzen einen `.str`**-Accessor**, der eine Vielzahl von Vektorisierten String-Operationen zur Verfügung stellt [25]. Das heißt, man kann auf einen ganzen Spaltenvektor an Strings gleichzeitig eine Operation anwenden, anstatt über einzelne Werte zu loopen.

Häufige `.str`**-Methoden:** - `str.lower(), str.upper()`: alles klein/groß schreiben.
- `str.strip()`: Leerzeichen am Anfang/Ende entfernen (gut für Bereinigung).
- `str.contains("xyz")`: Boolesche Series, ob Substring vorkommt (Regex möglich).
- `str.len()`: Länge jedes Strings.
- `str.replace("alt", "neu")`: Teilstrings ersetzen (kann Regex nutzen).
- `str.split(separator)`: Strings aufteilen (ergibt Liste je Eintrag, oft mit `.str[0]` oder `.str.get(n)` kombiniert, um z.B. erste Komponente zu bekommen).
- `str.slice(start, end)`: Zeichenkettenausschnitt wie bei Python Slices.
Und viele mehr: z.B. `str.findall`, `str.isdigit`, `str.pad`, `str.zfill` etc., um typische string Transformationen bequem vektorisiert durchzuführen [25].

Beispiel: In unserer Kundenliste `df` haben wir "Name" als "Vorname Nachname". Möchten wir z.B. den Nachnamen in einer eigenen Spalte: `df["Nachname"] = df["Name"].str.split().str[-1]`. Das nimmt den Namen, teilt an Whitespace in Liste [Vorname, Nachname] und `.str[-1]` greift jeweils das letzte Element (Nachname) heraus. Ebenso könnte `df["Vorname"] = df["Name"].str.split().str[0]` den ersten Teil geben.

Ein anderes Beispiel: Versicherungsnummern liegen als String "KV-12345/2023" vor, und man möchte den Teil vor dem '-' und den Jahrgang extrahieren. Angenommen `df["VersNr"]` enthält solche Strings:
- `df["VersArt"] = df["VersNr"].str.split("-").str[0]` (gibt "KV")
- `df["VersJahr"] = df["VersNr"].str.split("/").str[1]` (gibt "2023")
Man kann auch mit Regex arbeiten: `df["VersNr"].str.extract(r'(\w+)-\d+\/(\d+)', expand=True)` würde z.B. zwei Gruppen extrahieren (alles Buchstabige vor dem '-', und Ziffern nach dem '/'). Das führt aber schon tiefer in Regex-Thematik.

Übung: In `df` gibt es eine Spalte "Adresse" mit Einträgen wie "Musterstr. 12, 12345 Stadt". Extrahieren Sie die Postleitzahl in eine neue Spalte "PLZ". (Tipp: `str.split(",")` und dann den zweiten Teil, oder regex r'\b(\d{5})\b'). Erzeugen Sie außerdem eine Spalte "Stadt" ähnlich. Testen Sie `df["Stadt"].value_counts()`: Welche Stadt kommt am häufigsten vor?

Reinigen Sie danach die "Adresse"-Spalte, indem Sie z.B. `df["Adresse"] = df["Adresse"].str.strip()` aufrufen, um etwaige führende/trailende Leerzeichen loszuwerden.

Quiz – String-Operationen: Gegeben `df["Name"] = "Max Mustermann" (bei allen Zeilen unterschiedlich)`. Was tut `df["Name"].str.contains("Must")`? *(Single Choice)*

[(X)] Gibt eine boolesche Series zurück, die anzeigt, ob "Must" im Namen vorkommt (z.B. True für "Max Mustermann") [25].
[( )] Teilt den Namen an "Must" und behält den ersten Teil.
[( )] Entfernt "Must" aus dem Namen.
[( )] Wandelt den String in Großbuchstaben um.

(Lösung: `.str.contains` *prüft das Vorkommen eines Substrings und gibt True/False pro Element zurück.)*

4.2 MultiIndex DataFrames: Arbeiten mit hierarchischen Indizes

Ein **MultiIndex** ist ein *hierarchischer Index* mit mehreren Ebenen. Sowohl Zeilen als auch Spalten können einen MultiIndex haben. Dies erlaubt z.B. zweistufige Gruppierungen oder Kreuztabellen in einem DataFrame darzustellen. Wie zuvor in Lektion 3.4 erwähnt, entsteht ein MultiIndex häufig automatisch, z.B. bei `groupby(["Region","Versicherungstyp"])`. Man kann aber auch manuell einen MultiIndex setzen.

Vorteile: MultiIndex kann komplexe Daten übersichtlich repräsentieren. Man kann damit z.B. Daten mit drei Dimensionen (z.B. Zeit, Region, Kategorie) in einem 2D-DataFrame halten, indem zwei Dimensionen in den Index gepackt werden [26]. Es ermöglicht fortgeschrittene Pivot-Tabellen, etc. Allerdings ist es zunächst etwas ungewohnt in der Handhabung.

Zugriff: Wenn `df` einen MultiIndex auf den Zeilen hat (z.B. Level0=Region, Level1=Tarif), kann man `df.loc[("Nord","Privat")]` verwenden, um die entsprechenden Zeilen auszuwählen. Man kann auch nur eine Ebene angeben: `df.loc["Nord"]` würde alle Zeilen der Region Nord (egal welcher Tarif) geben. Für MultiIndex-Spalten ähnlich: z.B. ein DataFrame mit Spalten MultiIndex (Level0 könnte Kennzahl wie "Summe", "Anzahl"; Level1 könnten Kategorien) – da greift man mit `df["Summe", "Privat"]` etwa auf die entsprechende Spalte.

Erstellung: Man kann MultiIndex selbst erzeugen, z.B. `df.set_index(["Region","Tarif"], inplace=True)` würde einen MultiIndex aus zwei vorhandenen Spalten machen. Oder via `pd.MultiIndex.from_product()` etc., um Kombinationen zu erzeugen [27]. Oftmals genügt aber die automatische Erstellung durch Hierarchical Grouping oder Stack/Unstack Operationen.

Beispiel: Nach `groupby(["Region","Versicherungstyp"]).agg(...)` erhält man einen MultiIndex (Region als oberste Indexebene, Tarif als zweite). Das Ergebnis sieht z.B. so aus:

```
                    anzahl   mittel_alter
Region    Versicherungstyp
Nord      Gesetzlich       120        45.3
          Privat            30        50.1
Sued      Gesetzlich       150        40.2
          Privat            20        47.8
```

Hier ist ein zweistufiger Zeilenindex. Um z.B. alle Daten für Region "Nord" zu bekommen, kann man `result.loc["Nord"]` nutzen (gibt beide Zeilen Nord/Gesetzlich und Nord/Privat). Einzelner Wert: `result.loc[("Nord","Privat"), "anzahl"]` wäre 30 in dem Beispiel.

Unstack: Mit `df.unstack(level)` kann man eine Index-Ebene in Spalten überführen. Z.B. `result.unstack("Versicherungstyp")` würde den obigen Output in ein DataFrame mit Index=Region und zwei Spalten (anzahl_priv, anzahl_gesetzlich – je nach Aggregat evtl. MultiIndex in Spalten). Umgekehrt macht `df.stack()` aus Spalten wieder eine Index-Ebene.

Übung: Nehmen wir an, `sales_df` enthält Spalten: *Jahr*, *Quartal*, *Umsatz*. Erzeugen Sie einen MultiIndex, indem Sie `sales_df.set_index(["Jahr","Quartal"], inplace=True)`. Nun haben Sie einen zweistufigen Index nach Jahr und Quartal. Versuchen Sie, mittels `sales_df.loc[2019]` alle

Quartale von 2019 auszuwählen. Wählen Sie mittels `sales_df.loc[(2020, 2)]` explizit das 2. Quartal 2020 aus.

Machen Sie dann ein `sales_df.unstack(level=0)` – was beobachten Sie? (Vermutlich werden die Jahre zu Spalten, Quartale bleiben Index – man erhält eine Quartal x Jahr Tabelle der Umsätze.)

Quiz – MultiIndex: Was ist **kein** typisches Merkmal eines Pandas MultiIndex? *(Single Choice)*

[( )] Er erlaubt hierarchische Gliederung von Daten, z.B. zweistufige Indexierung nach Region und Tarif.
[( )] Man kann sowohl Zeilen als auch Spalten als MultiIndex haben.
[( )] MultiIndex ist im Grunde eine Ebene von Tupel-Werten statt einfachen Werten.
[(X)] Er erzwingt, dass alle Ebenen sortiert sein müssen, sonst kommt es zu Fehlern.

(Lösung: Sortierung ist keine zwingende Voraussetzung – Pandas kann unsortierte MultiIndex handhaben, auch wenn manche Ops warnen können. Die anderen Aussagen beschreiben MultiIndex korrekt [27] *.)*

4.3 Lambda-Funktionen und DataFrame-Methoden: map(), apply(), applymap()

Manchmal reichen die eingebauten Methoden nicht aus und man möchte eine eigene Funktion auf Daten anwenden. Dafür gibt es in Pandas mehrere Wege:

- **map():** Wendet eine Funktion oder ein Mapping (Dict, Series) auf jede **Einzelwerte** einer Series an. Beispiel: `df["Name"].map(len)` würde die Länge jedes Namens als Zahl zurückgeben (len-Funktion auf jeden String). Oder `df["Region"].map({"Nord": "N", "Sued": "S"})` könnte Nord/Sued zu Abkürzungen mappen (Werte ohne Mapping resultieren in NaN). `map` ist nur für Series.

- **apply() auf Series:** Ähnlich wie map, aber flexibler – `df["Alter"].apply(lambda x: x*2)` würde alle Alter verdoppeln (sinnlos, aber als Demo).

- **apply() auf DataFrame:** Hier kann man `df.apply(func, axis=0 or 1)` nutzen. axis=0 (Standard) heißt, *die Funktion wird pro Spalte* angewandt (bekommt jeweils eine Series = Spalte). axis=1 heißt, *die Funktion wird pro Zeile* angewandt (bekommt jeweils eine Series = Zeile). Beispiel: `df.apply(lambda col: col.isna().sum())` mit axis=0 würde eine Series zurückgeben mit der Anzahl NAs pro Spalte. Oder `df.apply(lambda row: row["Betrag"] if row["Status"]=="offen" else 0, axis=1)` würde pro Zeile entscheiden: wenn Status offen, nehme Betrag, sonst 0 – Ergebnis wäre z.B. eine Series offener Betrag pro Zeile.

- **applymap():** Wendet eine Funktion elementweise auf *jede Zelle* eines DataFrames an (selten gebraucht, da meist Column/Row-wise oder vectorized einfacher geht). z.B. `df[["A","B"]].applymap(lambda x: x*2)` verdoppelt alle Werte in Spalten A und B. Oft kann man aber direkt `df*2` machen, daher seltener.

Performance-Hinweis: Vectorisierte Methoden (wie die .str-Methoden oder rechnerische Operationen auf Series) sind in Pandas intern in C implementiert und deutlich schneller als apply mit Python-Lambda. Apply/Map sind mächtig, aber wenn möglich, versuchen Sie Pandas/NumPy-Vektorlogik zu nutzen – das ist i.d.R. **performanter** (siehe nächstes Kapitel zur Performance) [28] .

Beispiele: - `df["BMI"] = df.apply(lambda row: row["Gewicht"] / (row["Groesse"]/100)**2, axis=1)` – berechnet Body-Mass-Index pro Zeile (Gewicht(kg)/Größe(m)^2), falls Gewicht in

kg und Größe in cm vorliegen. - `df["Name"].apply(lambda x: x.split()[-1])` – gibt den Nachname wie oben, aber via apply (während `.str.split().str[-1]` effizienter wäre, aber zur Demonstration). - `df_num = df[["Col1","Col2"]].applymap(float)` – konvertiert zwei Spalten (sofern vorher z.B. int) in float durch Element-Umwandlung.

Übung: In `kunden_df` gibt es Spalten *Vorname* und *Nachname* oder alternativ eine Spalte *Name*. Nutzen Sie `apply` oder `map`, um z.B. die Initialen jedes Kunden zu erzeugen. (Tipp: `df.apply(lambda row: row["Vorname"][0] + row["Nachname"][0], axis=1)` wenn getrennte Spalten, oder `df["Name"].map(lambda x: "".join([n[0] for n in x.split()]))` wenn ein Feld.) Speichern Sie das Ergebnis in `df["Initialen"]`.

Ein weiteres Beispiel: Angenommen, es gibt eine Spalte *Einkommen* und *Alter*. Sie wollen eine einfache Risikokategorie berechnen: z.B. wenn Einkommen < 30000 und Alter > 50 => "hoch", sonst "normal" (nur als willkürliches Beispiel). Schreiben Sie eine Funktion `risk_level(row)` die eine Zeile entgegennimmt und so eine Kategorie zurückgibt, und nutzen Sie `df.apply(risk_level, axis=1)`, um eine neue Spalte *Risiko* zu füllen.

Quiz – apply vs. applymap: Was ist der Unterschied? *(Single Choice)*

[( )] `apply` gibt immer einen DataFrame zurück, `applymap` immer eine Series.
[( )] `apply` kann keine Lambda-Funktionen verwenden, `applymap` schon.
[(X)] `apply` kann spalten- oder zeilenweise auf ein DataFrame angewandt werden (oder auf eine einzelne Series), während `applymap` stets auf jedes einzelne Element eines DataFrames wirkt [29].
[( )] Es gibt keinen Unterschied, `applymap` ist nur ein Alias für `apply`.

(Lösung: `apply` *ist flexibler und kann auf Zeilen/Spaltenebene wirken,* `applymap` *ist elementweise für DataFrames.)*

Qualität ok? Beherrschen Sie nun auch fortgeschrittene Manipulationen wie das Verarbeiten von Strings, das Nutzen von MultiIndex-Strukturen und den gezielten Einsatz von eigenen Funktionen via apply/map? Wenn ja, sind Sie bereit für den letzten inhaltlichen Abschnitt: Datenvisualisierung und Performance.

Lektion 5: Datenvisualisierung und Performance-Optimierung

Im abschließenden inhaltlichen Kapitel betrachten wir die **Visualisierung** von Daten mit Pandas (bzw. in Verbindung mit Bibliotheken wie Matplotlib oder Plotly) sowie einige Hinweise zur **Performance-Optimierung**, damit selbst größere Daten effizient verarbeitet werden können.

5.1 Visualisierung mit Pandas (Matplotlib/Plotly Integration)

Warum Visualisierung? Ein Diagramm sagt oft mehr als tausend Zahlenreihen. Für Zusammenhänge und Präsentationen sind Grafiken enorm wertvoll. Pandas bietet eine einfache Schnittstelle zu Matplotlib, der Standard-Plotbibliothek für Python. Zusätzlich kann Pandas mit der interaktiven Bibliothek Plotly verwendet werden (z.B. über den Plotly-Express Interface oder als Backend).

Pandas .plot()-API: Jedes DataFrame und jede Series hat eine `.plot()` Methode. Intern ruft sie Matplotlib auf [30]. Standardmäßig erzeugt `df.plot()` je nach Daten einen Liniendiagramm für

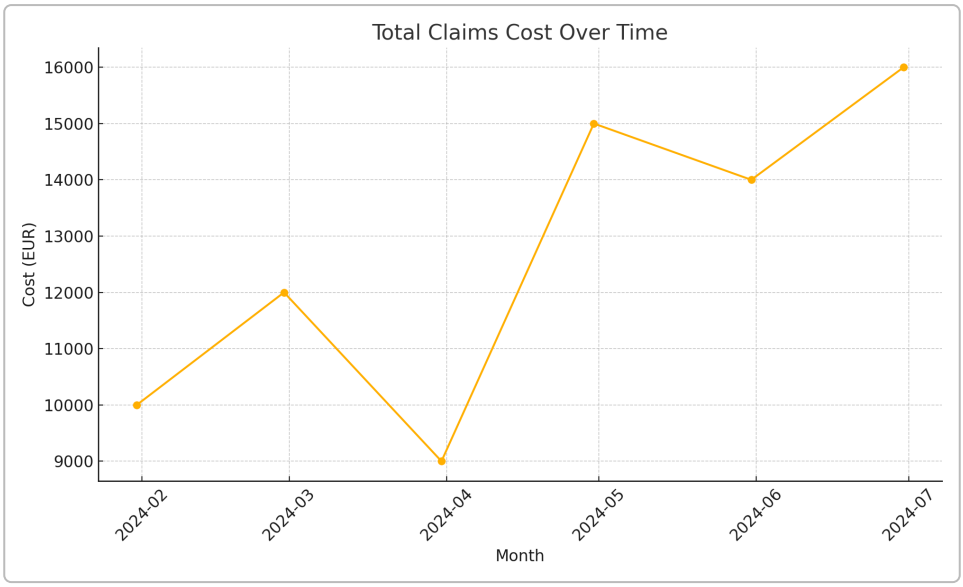
numerische Spalten (Index auf X-Achse). Man kann aber spezifizieren: `df.plot(kind="bar")` für Balkendiagramm, `df.plot(kind="hist")` für Histogramm, `df.plot(kind="box")` usw. Es gibt auch spezielle Methoden: `df.plot.bar()`, `df.plot.hist()` etc., oder für eine Series z.B. `s.plot.pie()`.

Matplotlib direkt: Oft will man Feintuning, dann kann man auch Matplotlib-Aufrufe kombinieren. Z.B. `ax = df["Betrag"].plot(kind="hist"); ax.set_title("Histogramm der Beträge")`. Für komplexere Plots (z.B. mehrere DataFrames in einem Chart) nutzt man eher Matplotlib oder eine spezialisierte Bibliothek.

Plotly: Durch Setzen von `pd.options.plotting.backend = "plotly"` kann man die `.plot()`-Aufrufe über Plotly laufen lassen, was interaktive Grafiken ergibt (z.B. zoombar, tooltips). Alternativ direkt Plotly Express: z.B. `import plotly.express as px; px.line(df, x="Datum", y="Betrag", color="Art")`. Plotly erfordert etwas Einarbeitung, liefert aber schöne interaktive Ergebnisse. In diesem Kurs beschränken wir uns auf Matplotlib/Standard-Plots.

Beispiele: - Liniendiagramm: `monats_summen.plot(title="Monatliche Summen")` – wenn `monats_summen` eine Series mit DatetimeIndex ist, ergibt das eine Zeitreihengrafik über die Monate.
- Balkendiagramm: `df["Tarifart"].value_counts().plot(kind="bar")` – zeigt die Häufigkeiten je Tarifart als Balken.
- Streudiagramm: `df.plot(kind="scatter", x="Alter", y="Betrag")` – z.B. um Zusammenhang Alter vs. Schadensbetrag zu sehen (gibt es einen Trend?).
- Mehrere Linien: `df[["2019","2020","2021"]].plot()` – falls df z.B. Umsätze pro Monat hat als Index und Spalten=Jahr, werden drei Linien geplottet.

Übung: Visualisieren Sie einige der Ergebnisse früherer Lektionen: 1. Nehmen Sie die Gruppierung pro Region aus Lektion 3.4 – erzeugen Sie ein Balkendiagramm der Kundenanzahl pro Region (`plot.bar()`). Beschriften Sie ggf. x- und y-Achse via Matplotlib: `plt.xlabel("Region"); plt.ylabel("Anzahl Kunden")`.
2. Plotten Sie die Zeitreihe der Quartalsfälle aus Lektion 3.6. Nutzen Sie `title="Fälle pro Quartal"` in `plot()`. Wenn Sie mögen, formatieren Sie das Datum auf der X-Achse lesbar (in Jupyter z.B. `plt.xticks(rotation=45)`).
3. Bonus: Erstellen Sie ein **Kuchendiagramm** (Pie Chart) der Vertragsanteile: z.B. `df["Versicherungstyp"].value_counts().plot.pie(autopct="%1.1f%%")` – das zeigt prozentual, wie viel Anteil Privat vs. Gesetzlich haben. (Beachten: Kuchendiagramme sind oft verpönt bei >2 Kategorien, aber zur Demo okay.)



Beispiel: Das folgende Liniendiagramm zeigt eine fiktive **Zeitreihe der Gesamtkosten von Leistungsfällen pro Monat**. Solche Visualisierungen helfen, Trends zu erkennen – z.B. hier ein starker Anstieg im Mai. *(Dieses Diagramm wurde mit Pandas/Matplotlib erzeugt.)*

5.2 Performance-Optimierung: Speicher und Geschwindigkeit (Vectorization)

Pandas ist bequem, aber man sollte Performance im Blick behalten, gerade mit großen Daten (100k+ Zeilen). Einige Tipps:

- **Vermeide Python-Schleifen:** Pandas/NumPy arbeiten intern in C/NumPy-Vektoren viel schneller. Also statt `for`-Schleife über jede Zeile lieber vektorisiert rechnen. Beispiel: `df["x+y"] = df["x"] + df["y"]` **statt** Zeile für Zeile zu addieren. Das ist *vectorization*: Operationen auf ganze Arrays gleichzeitig [28]. Eine Loop kann um Größenordnungen langsamer sein.

- **Use built-in functions:** z.B. statt eine eigene Summe zu loop-en, `df["col"].sum()` nutzen. Statt custom string Ops ggf. .str-Methoden. Statt row-wise apply, schauen ob man mit einfachen Pandas-Methoden hinkommt (z.B. viele `np.where` oder boolean masks statt apply mit if/else).

- **Categoricals nutzen:** Wenn Sie viele wiederholte Strings haben (z.B. 100k Zeilen mit nur 5 verschiedenen Kategorien), wandeln Sie sie in `category` um. Das spart Speicher und macht viele Operationen schneller (Vergleiche, Grouping).

- **Dtypes anpassen:** Ein DataFrame mit 1 Million integer-Zahlen braucht in int64 ~8 MB pro Spalte; in int32 nur 4 MB – falls Werte eh < 2e9 sind, könnte man downcasten. `pd.to_numeric(df["col"], downcast="integer")` versucht z.B., einen kleineren Integer-Typ zu wählen. Gleiches für float.

- **Chunking beim Einlesen:** Bei sehr großen CSVs kann `pd.read_csv(..., chunksize=100000)` helfen, häppchenweise zu verarbeiten statt alles auf einmal (Memory Trade-off).

- **bewusst mit Kopien umgehen:** Pandas warnt manchmal vor *chained assignment* (df[df.x>0]["y"]=5), was ineffizient sein kann. Besser in Schritten oder mit `.loc` assignen.

- **Alternative Technologien:** Wenn Pandas an Grenzen stößt, gibt es Alternativen: **Dask** (verteiltes Pandas), **Polars** (DataFrame-Library in Rust, sehr schnell), **SQL-Datenbanken** oder **PySpark**. Für viele tägliche Aufgaben reichen aber Pandas auf einem modernen Rechner aus. Polars z.B. bietet teils 5-10x Geschwindigkeit bei bestimmten Operationen und geringeren Speicherverbrauch [31] [32], ist aber (Stand 2025) noch nicht ganz so breit unterstützt wie Pandas.

Beispiel: Ein konkreter Fall – Sie müssen millionen Kunden-Datensätze nach bestimmten Kriterien filtern und berechnen. Versuchen Sie zuerst, alles mit Pandas Bordmitteln zu lösen (boolesches Masking, vectorized Ops). Falls die Performance dennoch hapert, prüfen Sie: sind Datentypen optimal (vielleicht `category` für wenige Kategorien)? Sind unnötige Kopien im Code? Notfalls überlegen, ob eine Aggregation schon in der Datenbank erfolgen kann (um weniger Daten zu ziehen).

Übung (Experiment): Schreiben Sie zwei Funktionen, die dasselbe tun: z.B. eine Spalte `"x*2"` berechnen für 1 Million Zufallszahlen. Erste Funktion `loop_double(df)` iteriert mit `for i in df.index: df.loc[i,"z"] = df.loc[i,"x"]*2`. Zweite Funktion `vectorized_double(df)` macht `df["z"] = df["x"]*2`. Messen Sie die Zeit (z.B. mit `%%timeit` in Jupyter oder `time`-Modul). Sie werden sehen, dass die vektorisierte Variante um ein Vielfaches schneller ist. Ebenso testen: `df.apply(lambda x: x*2)` auf Spalte vs direkt `*2`.

Hinweis: In Pandas 2.x werden manche Operationen unter der Haube schon optimierter (z.B. via PyArrow-Backed data). Dennoch gelten obige Prinzipien. Bei sehr großen Daten kann ein Wechsel zu Polars erwogen werden – Polars bietet hohe Performance und ein ähnliches API-Konzept (DataFrame), erfordert aber etwas Einarbeitung. Ein Vorteil: **Polars** nutzt Rust im Backend und hat keine Global Interpreter Lock Probleme, wodurch es mehrere Threads nutzen kann. Benchmarks zeigen Polars oft deutlich schneller als Pandas bei Gruppierungen, Joins etc., während Pandas eine lange Entwicklung und breitere Funktionalität hat [32]. In Summe: Wählen Sie das Werkzeug, das am besten zu Ihrer Problemgröße passt.

Quiz – Performance: Warum sind *vectorized operations* in Pandas in der Regel schneller als Python-Schleifen? *(Single Choice)*

[( )] Weil Python-Schleifen in Pandas intern parallelisiert werden.
[( )] Weil der Pandas-Interpreter optimierten Bytecode generiert.
[(X)] Weil Pandas/NumPy Operationen in niedrigere Sprachen (C/NumPy) ausgelagert sind und auf ganze Arrays angewandt werden, statt Element-für-Element in Python [28].
[( )] Das stimmt nicht, Schleifen sind genauso schnell, nur der Code ist länger.

(Lösung: Pandas nutzt optimierte, in C implementierte Vectorized Operations.)

Qualität ok? Haben Sie einen Eindruck, wie man mit Pandas Diagramme erstellt und worauf man bei Performance achten sollte? Damit haben wir die inhaltlichen Module abgeschlossen. Als nächstes folgt noch ein Bonus-Thema sowie eine Zusammenfassung mit Projektideen und Übungen zur Lernfortschrittskontrolle.

Bonus-Thema: Geopandas & Polars (Ausblick)

Abschließend ein kurzer Ausblick auf zwei verwandte Themen, die über den Kern von Pandas hinausgehen, aber in speziellen Kontexten interessant sein können.

- **GeoPandas:** Die Bibliothek GeoPandas erweitert Pandas um die Fähigkeit, **räumliche Daten** (Geo-Daten) zu verarbeiten. Sie stellt GeoDataFrame und GeoSeries bereit (Unterklassen von Pandas DataFrame/Series) [33], mit denen man z.B. Shapefiles (Geodatenformate) lesen kann, geometrische Operationen (Schnittmengen, Puffer, Distanzberechnungen) durchführen kann, und Geodaten plotten kann. Für eine Krankenkasse könnten Geodaten z.B. eingesetzt werden, um Mitglieder oder Leistungsfälle auf Regionen oder Karten zu beziehen (etwa: Heatmap der Kosten nach Bundesland). GeoPandas integriert sich gut, erfordert aber Kenntnis von GIS-Konzepten. Wenn Ihre Aufgaben Geodaten beinhalten, ist GeoPandas das Mittel der Wahl.

- **Polars:** Bereits erwähnt in Performance: Polars ist eine relativ neue **DataFrame-Bibliothek** (Open-Source, in Rust geschrieben, mit Python-API). Sie ist ein **sehr performanter** Ersatz für Pandas in vielen Anwendungsfällen. Polars arbeitet lazy (ähnlich SQL-Abfrageplanung) und nutzt effizient Multi-Core. Für uns wichtig: Das API von Polars ist nicht identisch zu Pandas, aber ähnlich. Es verwendet z.B. Ausdrücke statt direct apply, aber Konzepte wie DataFrame, GroupBy, Join gibt es auch. Wenn in Ihrem Unternehmen sehr große Daten verarbeitet werden müssen und Pandas an Grenzen stößt, lohnt ein Blick auf Polars. Oft kann man vorhandene Pandas-Kenntnisse übertragen. **Vergleich:** *"Polars adds performance, efficiency, and a modern design to the DataFrame world, whereas Pandas offers flexibility and a deep history."* [32] – d.h. Pandas ist etabliert und vielseitig, Polars punktet mit Geschwindigkeit und niedrigerem Speicherverbrauch. Vielleicht werden künftige Selbstlernmodule Polars tiefer behandeln – für den Moment genügt es zu wissen, dass es Alternativen gibt.

Ressourcen zu Bonus: Schauen Sie bei Interesse auf die offizielle GeoPandas Doku oder ein Polars Tutorial. Beide Communities sind aktiv und bieten gute Einstiegsbeispiele.

Empfohlene Ressourcen und Abschluss

Zum Weiterlernen und Vertiefen der Pandas-Kenntnisse hier einige **Ressourcen**:

- **Offizielle Pandas-Dokumentation:** Umfangreich aber sehr informativ, mit einem einsteigerfreundlichen Tutorial ("10 Minutes to pandas" [34] und den "Getting Started" Guides). Bei speziellen Fragen hilft die API-Reference schnell weiter.

- **Bücher/Kurse:** *Python for Data Analysis* (Wes McKinney) – vom Pandas-Entwickler geschrieben, sehr praxisnah. Online-Kurse auf Datacamp, Kaggle Learn [35] etc. bieten interaktive Übungen.

- **Community und Hilfe:** StackOverflow hat tausende Pandas-Fragen bereits beantwortet. Die Pandas-Tag-Suche dort ist oft Gold wert. Auch die Pandas Github-Discussions oder Reddit /r/pandas können helfen.

- **Tools:** Wie erwähnt, nutzen Sie Jupyter Notebook/Lab für interaktives Arbeiten – dort lassen sich DataFrames gut anzeigen, man kann schrittweise vorgehen. Für längere Projekte ist eine IDE wie

PyCharm oder VS Code mit Python-Extension hilfreich, gerade wenn man Scripts modular aufbaut.

- **Datensätze zum Üben:** Es gibt zahlreiche öffentliche Datensätze, um Pandas-Fähigkeiten zu schulen. Z.B. auf Kaggle Datasets findet man alles von Gesundheitsdaten bis zu Finanzdaten. Das **Bundesgesundheitsministerium** oder andere Behörden veröffentlichen Open Data (CSV/JSON) – ideal, um echte Datenanalyse zu proben. Nutzen Sie auch firmeneigene anonymisierte Daten, falls verfügbar, um den direkten Praxisbezug herzustellen.

Projektphasen und Lernkontrolle

Abschließend soll ein **Praxisprojekt** den gelernten Stoff verankern. Orientieren Sie sich dabei an typischen **Projektphasen** der Datenanalyse:

1. **Ziele und Fragen definieren:** Überlegen Sie sich ein Szenario – z.B. *"Analyse der Krankenkassenleistungen im Jahr 2024"* – und definieren Sie konkrete Fragen (z.B. *Welche Leistungsart verursacht die höchsten Kosten? Wie entwickeln sich die Kosten quartalsweise? Gibt es Unterschiede zwischen Regionen?*).

2. **Daten identifizieren/beschaffen:** Stellen Sie (synthetische) Daten zusammen, die Sie zur Beantwortung brauchen. Im Beispiel könnten Sie Datensätze zu Leistungsfällen, zu Versichertenstammdaten und evtl. regionale Zuordnung haben. Laden Sie diese Daten (CSV/Excel/SQL) mit Pandas ein.

3. **Datenbereinigung:** Säubern Sie die Daten: Fehlende Einträge behandeln, Datentypen korrekt setzen (z.B. Datumsfelder mit `to_datetime`), Duplikate entfernen, evtl. Ausreißer identifizieren. Dokumentieren Sie die Schritte.

4. **Datenanalyse:** Wenden Sie die Techniken aus Lektion 3 und 4 an, um Ihre Fragen zu beantworten. Erstellen Sie Gruppierungen, Pivot-Tabellen, berechnen Sie Kennzahlen. Nutzen Sie `describe()` und andere Methoden, um Einblicke zu bekommen. Schreiben Sie vielleicht Funktionen, um bestimmte Berechnungen zu automatisieren (z.B. Kostensumme pro Versichertem).

5. **Datenauswertung & -visualisierung:** Bereiten Sie die Ergebnisse in verständlicher Form auf. Erstellen Sie Diagramme für die wichtigsten Erkenntnisse (z.B. Balkendiagramm Top-5 Leistungsarten nach Kosten, Liniendiagramm Kostenverlauf über das Jahr, etc.). Stellen Sie sicher, dass jedes Diagramm klar beschriftet ist.

6. **Data Storytelling:** Präsentieren Sie die Ergebnisse, als würden Sie sie einem Entscheider erläutern. Schreiben Sie ein kleines Fazit oder einen Bericht: Welche Trends oder Auffälligkeiten haben Sie gefunden? Was könnte die Krankenkasse daraus lernen oder als Maßnahme ableiten? Storytelling bedeutet, aus den Daten eine verständliche Geschichte zu formen.

Während dieses Projekts wenden Sie *alle* Module an – von Import über Bereinigung bis Visualisierung. Das konsolidiert Ihr Wissen.

Lernfortschrittskontrolle: Zur Überprüfung Ihres Lernfortschritts können Sie folgende Schritte durchlaufen:

- **Quizfragen beantworten:** Die im Kurs gestellten Quiz (und gern weitere ausgedachte) ohne Hilfsmittel beantworten. Wenn Sie hier Lücken merken, gehen Sie die entsprechende Lektion nochmals durch.

- **Programmieraufgaben lösen:** Nehmen Sie sich 2–3 Aufgaben, z.B. *"Lese Datei X ein und ermittle Y"*, *"Transformiere Datensatz Z auf folgende Weise…"* und versuchen Sie, den Code ohne Spickzettel zu schreiben. Übung macht den Meister.

- **Abschlussprojekt umsetzen:** Führen Sie das oben skizzierte Projekt eigenständig durch. Das ist der beste Beweis, dass Sie Pandas in der Praxis anwenden können. Scheuen Sie nicht, zwischendrin die Dokumentation zu Rate zu ziehen – auch Profis googeln Syntax oder Details regelmäßig.

Am Ende sollten Sie in der Lage sein, eine komplette **Daten-Analyse-Pipeline** mit Pandas zu bewältigen – von den Rohdaten bis zur visualisierten Erkenntnis. Viel Erfolg bei Ihrem Abschlussprojekt und den weiteren Daten-Abenteuern!

(Dieser Lehrplan kombinierte theoretische Grundlagen mit praktischen Übungen. Wenn Sie bis hierhin alles nachvollzogen haben, verfügen Sie nun über ein solides Pandas-Werkzeugset, das Ihnen im Ausbildungsalltag – speziell in einer Krankenkasse mit vielen Daten – wertvolle Dienste leisten wird.)

---

1 9 10 11 How do I read and write tabular data? — pandas 2.3.0 documentation
https://pandas.pydata.org/docs/getting_started/intro_tutorials/02_read_write.html

2 3 4 5 6 7 8 34 10 minutes to pandas — pandas 2.3.0 documentation
https://pandas.pydata.org/docs/user_guide/10min.html

12 How to rename columns in Pandas DataFrame - GeeksforGeeks
https://www.geeksforgeeks.org/how-to-rename-columns-in-pandas-dataframe/

13 pandas.DataFrame.iloc — pandas 2.3.0 documentation
https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.iloc.html

14 15 16 pandas.DataFrame.loc — pandas 2.1.0 documentation
https://pandas.pydata.org/pandas-docs/version/2.1.0/reference/api/pandas.DataFrame.loc.html

17 18 19 20 Working with missing data — pandas 2.3.0 documentation
https://pandas.pydata.org/docs/user_guide/missing_data.html

21 Pandas DataFrame query() Method - W3Schools
https://www.w3schools.com/python/pandas/ref_df_query.asp

22 Group by: split-apply-combine — pandas 2.3.0 documentation
https://pandas.pydata.org/docs/user_guide/groupby.html

23 24 pandas.Series.resample — pandas 2.3.0 documentation
https://pandas.pydata.org/docs/reference/api/pandas.Series.resample.html

25 Working with text data — pandas 2.3.0 documentation
https://pandas.pydata.org/docs/user_guide/text.html

26 27 MultiIndex / advanced indexing — pandas 2.3.0 documentation
https://pandas.pydata.org/docs/user_guide/advanced.html

28 What is Pandas Vectorization? - Medium
https://medium.com/@whyamit101/what-is-pandas-vectorization-459ed4d93806

29 pandas.DataFrame.apply — pandas 2.3.0 documentation - PyData |
https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.apply.html

30 pandas.DataFrame.plot — pandas 2.3.0 documentation - PyData |
https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.plot.html

31 Pandas vs. Polars: Benchmarking Dataframe Libraries with Real ...
https://pipeline2insights.substack.com/p/pandas-vs-polars-benchmarking-dataframe

32 Choosing Polars Over Pandas for High-Performance Data Analysis | Factspan
https://www.factspan.com/blogs/choosing-polars-over-pandas-for-high-performance-data-analysis/

33 Introduction to GeoPandas — GeoPandas 1.1.0+0.gc36eba0.dirty documentation
https://geopandas.org/en/stable/getting_started/introduction.html

35 Learn Pandas - Kaggle
https://www.kaggle.com/learn/pandas