Technische Universität Braunschweig

Python-Kurs für die pharmazeutische Analytik

Anwendungsbezogener Einstieg für Einsteiger

Jannis Wowra, 7. Mai 2025

KI-gestützte Unterlagen & Automatisierungsvorteile

- Präsentation und Codebeispiele wurden mithilfe von KI (ChatGPT & Claude) erstellt.
- Mit Grundkenntnissen in Python:
  - Routineaufgaben automatisieren
  - Große Datensätze effizient verarbeiten
  - Zeit und Ressourcen sparen
- Syntax-Fragen und Verständnislücken?
  - KI bietet schnelle Erklärungen
  - Hilft beim Debuggen und Optimieren

Warum Python?

Programmiersprache: Python

- Einfach zu erlernen und gut lesbar
- Zahlreiche fertige Module Bibliotheken verfügbar
- Ideal für vielfältige Anwendungen in der analytischen Chemie
- Intuitive Fehlersuche und Debugging-Werkzeuge

Integrierte Entwicklungsumgebung (IDE)

- Eine IDE bietet:
  - Syntax-Highlighting Autovervollständigung
  - Integriertes Debugging
  - Projekt- und Dateiverwaltung
  - Unterstützung für virtuelle Umgebungen Versionierung
- Empfohlene Tools:
  - PyCharm
  - VS Code
- **Jupyter Notebooks sind kein Ersatz für eine echte IDE**

Debugger – Was ist das und warum nutzen?

- Ein **Debugger** ist ein Werkzeug zum schrittweisen Durchlaufen von Programmen
- Ermöglicht das Setzen von **Breakpoints**, um an bestimmten Stellen anzuhalten
- Anzeige und Überwachung von **Variablenwerten** und Programmlaufzeit-Zustand
- Hilft, **Ursache von Fehlern** schnell zu identifizieren
- Unverzichtbar für effizientes **Debugging** und sauberen Code

Herausforderungen für Einsteiger

- **Umgebungs-Setup:**
  - Installation von Python und Konfiguration von virtuellen Umgebungen
  - Pfad- und Versionskonflikte können zu Importfehlern führen
- **Bibliotheken installieren:**
  - Kenntnis von `pip` bzw. `conda` notwendig
  - Unterschiedliche Versionen und Abhängigkeiten erfordern oft manuelles Troubleshooting
- **Fehler beheben:**
  - Fehlermeldungen verstehen und gezielt nach Lösungen suchen
  - Stacktraces lesen und Debugging-Strategien (z. B. `print()` oder IDE-Debugger) einsetzen
- **Strukturiertes Vorgehen:**
  - Klare Aufgabenstellung und schrittweises Testen von Teilabschnitten
  - Prompt-Formulierung: präzise Anforderungen helfen, passende Code-Vorschläge zu erhalten

Hello World in Python

Datei: `hello.py`

```
print("Hallo Welt!")
```

Ausführen im Terminal

```
$ python hello.py
Hallo Welt!
```

- Direktes Ausführen ohne Kompilierung
- `print()` schreibt den Text auf die Konsole
- Interpreter verarbeitet Skript zeilenweise

Python Basisdatentypen & Vergleichsoperatoren

Basisdatentypen

- **int** (Ganzzahlen)
- **float** (Gleitkommazahlen)
- **complex** (Komplexe Zahlen)
- **bool** (Wahrheitswerte)
- **str** (Zeichenketten, einzelnes Zeichen als Länge-1-String)

Vergleichsoperatoren & logische Operatoren

- Gleichheit/Ungleichheit: **a == b**, **a != b**
- Größer/Kleiner: **a > b**, **a < b**
- Logisch: **and**, **or**, **not**

Datentyp-Konvertierung (Casting)

- Explizite Umwandlung mit **int()**, **float()**, **str()**, **bool()**, **complex()**

Beispiel:

```
s = "123"           # Zeichenkette
i = int(s)          # "123" → 123
f = float(i)        # 123 → 123.0
t = str(f)          # 123.0 → "123.0"
b = bool(0)         # 0 → False
c = complex(i)      # 123 → (123+0j)
```

- Wichtig für **Daten-Kompatibilität**
- Unverzichtbar beim **Parsen** externer Daten

Wann benutze ich welchen iterierbaren Datentyp?

list `([a, b, c])`

- Messreihen, Datensätze hintereinander (z. B. Konzentrationswerte)
- Einfache Verarbeitung mit `append()`, Slicing

tuple `((a, b, c))`

- Feste Parametergruppen (z. B. Kalibrierungspunkte)
- Unveränderliche Konfigurationen

str `(äbc")`

- Kennzeichnungen/IDs (z. B. Probenummern)
- Parsen von Textdateien (Dateinamen, Header)

dict `({'a': 1, 'b': 2})`

- Zuordnung von Proben-ID zu Messwert (`{ID: Wert}`)
- Zugriff über Schlüssel, z. B. `values()` für alle Messungen

Bedingte Anweisungen: if, elif, else

Struktur

- **if** *Bedingung*:
  - Auszuführender Block, wenn wahr
- **elif** *Bedingung*:
  - Zusätzlicher Block, wenn vorherige Bedingung falsch
- **else**:
  - Ausführungsblock, wenn keine Bedingung zutrifft

Kurzbeispiel

- `if x > 0: print("positiv")|`
- `elif x == 0: print("null")|`
- `else: print("negativ")|`

Die for-Schleife

Zweck

- Wiederholung über alle Elemente einer Sequenz (z. B. Messwerte, Proben-IDs)
- Automatisierte Schritt-für-Schritt-Verarbeitung

Syntax

- `for element in iterable:`
- `  Auszuführender Block`

Anwendungsbeispiel

- `for wert in konzentrationen:`
- `  print(f"Konzentration: wert mg/L")`

Funktionen – Syntax

```
def funktionsname(param1, param2, ...) -> Rückgabetyp:
    """
    Docstring (optional)
    """
    # Anweisungsblock
    return wert
```

- Definition beginnt mit **def** und endet mit **:**
- Rückgabewert(en) mit **return** zurückliefern

Funktionen – Beispiele

Beispiel 1: Zwei Rückgabewerte, kein Parameter

```
def gib_zwei_werte() -> (int, str):
    zahl = 42
    text = "Antwort"
    return zahl, text
```

Beispiel 2: Zwei Parameter, Summe ausgeben

```
def berechne_summe(a, b):
    print(a + b)
```

Was ist eine Klasse?

- Eine **Klasse** ist ein Bauplan für Objekte
- Bündelt und kapselt zusammengehörige **Daten** (Attribute) und **Funktionen** (Methoden)
- Methoden sind Funktionen, die das Verhalten der Klasse definieren
- Erlaubt, komplexe Strukturen übersichtlich und wiederverwendbar abzubilden

Klassen – Einfaches Beispiel

```python
class Human:
    def __init__(self, size, weight):
        self.size = size        # Instanz-Variable
        self.weight = weight    # Instanz-Variable

    def get_bmi(self):
        return self.weight / (self.size/100)**2  # Methode

# Instanziierung und Methodenaufruf
jannis = Human(180, 75)              # Objekt erstellen
jannis_bmi = jannis.get_bmi()       # get_bmi() aufrufen
print(jannis_bmi)                    # BMI ausgeben
```

Module & Bibliotheken

- **Module**: Einzelne **Python**-Dateien (.py) mit zusammengehörigen Funktionen und Klassen
- **Bibliotheken**: Sammlungen von Modulen für spezifische Aufgaben (z. B. NumPy, pandas)
- Spart Entwicklungsaufwand: bewährte, getestete Funktionen sofort nutzen
- Fördert Lesbarkeit und Wartbarkeit durch klare Struktur
- Einbinden mit **import modulename** bzw. Installation via **pip install paketname**

pip – Paketmanager für Python

- pip ist der Standard-Paketmanager für Python-Bibliotheken
- Wird im Terminal (nicht in der Python-Shell) ausgeführt
- Wichtige Befehle: `pip install`, `pip uninstall`, `pip list`

Beispiel: Installation mehrerer Pakete

```
$ pip install numpy pandas
```

Wichtige Bibliotheken für analytische Chemie

- **NumPy**: Matrix- und Vektor-Operationen für numerische Berechnungen
- **pandas**: Laden und Verarbeiten von CSV-/Tabellendaten
- **Matplotlib**: Erstellung von Grafiken und Diagrammen
- **SciPy**: Statistik, Regressionsrechnung und Signalverarbeitung
- **scikit-learn**: Einfache Anwendung von Machine-Learning-Modellen
- **PyTorch**: Eigenständige Entwicklung und Training von Deep-Learning-Modellen

Virtuelle Umgebungen – Warum nutzen?

- Ohne virtuelle Umgebungen führt globales Installieren oft zu Versionskonflikten
- `.venv` isoliert Projekt-abhängigkeiten und garantiert reproduzierbare Ergebnisse
- Anfänger haben häufig Probleme beim Installieren und Aktivieren – besser gleich von Anfang an einsetzen

Allgemeines Vorgehen

- **Projekt erstellen**
- **Environment einrichten**
- **Bibliotheken installieren**
- **Eigenen Code schreiben**