Python Spezial

Dipl.-Ing. Leonard Guelmino

DI Leonard Guelmino

- HTL Hollabrunn
- BSc & MSc Informatik @TUWien
- Product Owner bei einem Finanzunternehmen
- `sofort-wohnen.at`
- Trainer @WIFI Wien
- Instruktor @Rotes Kreuz NÖ

Hallo

- whoami?
- Wie bin ich zu diesem Kurs gekommen?
- Welches Hintergrundwissen nehme ich mit?
- Was erwarte ich mir von diesem Kurs?

Pause wann?

Kursformat

Theorie	Demo	Übung	Recap
Vortrag mit Slides, um den theoretischen Hintergrund zu vermitteln	Optionale Demonstration durch den Trainer	Anwendung der Theorie anhand Coding-Beispielen im Kurs-Projekt	Gemeinsame Nachbesprechung anhand der Musterlösung

Demo

IDE Tour & venv

Übung 0

IDE Setup & “Hello World”

- Übungsordner: /Labs/00-Setup

Tag 1

Python Basics

Inhalt

- Python 13 Grundlagen
- Modularisierung & Best Practises
- File Handling & Wichtige Libraries
- Unstrukturierte Daten
- Datenbankzugriff

Python 13 Grundlagen

Python Basics

Syntax, Variablen & Datentypen

Python 13 Basics

- **Dynamische Typisierung:**
  - erkennt Datentypen automatisch zur Laufzeit
  - explizite Deklaration ist optional

- **Lesbarkeit:** Durch strikte Einrückung (Indentation) wird der Code strukturiert.

- **Namenskonventionen:** Variablen & Funktionen werden in `snake_case` geschrieben.

Syntax, Variablen & Datentypen

Python 13 Basics

```
# Basic assignment (Dynamic Typing)
species_name = "arus major"
wing_length_mm = 74.5
is_migratory = False
not_yet_set = None

# Type Hinting (Modern Standard)
ring_number: str = "AX-92831"
body_mass_g: float = 18.2
```

Listen & Tupel

Python 13 Basics

- **Liste (Mutable):** Eine veränderbare Sequenz von Elementen. Ideal für Datensätze, die während der Laufzeit wachsen oder sich ändern.
- **Tupel (Immutable):** Eine unveränderbare Sequenz. Schneller und speichereffizienter. Für konstante Daten.
- **Indexing:** Zugriff erfolgt 0-basiert. Negative Indizes erlauben Zugriff von hinten (-1 ist das letzte Element).

Listen & Tupel

Python 13 Basics

```
# List: Mutable collection of daily captures
captured_birds = ["Blaumeise", "Kohlmeise", "Rotkehlchen"]
captured_birds.append("Amsel")  # List can grow

# Tuple: Immutable capture site coordinates (Lat, Lon)
site_coords = (48.2082, 16.3738)
site_coords[0] = 49.000  # TypeError
```

Dictionaries

Python 13 Basics

- **Mapping:** Speichert Daten als Key-Value-Paare. Keys müssen unveränderbar (immutable) und einzigartig sein (oft Strings oder Zahlen).
- **Performance:** Bietet schnelle Lookups (`O(1)` Komplexität), ideal um spezifische Attribute abzurufen.
- **Anwendung:** Standardformat für strukturierte Daten in Python.

Dictionaries

Python 13 Basics

```
# Dictionary representing a single bird's biometric record
bird_record = {
    "ring_id": "H77-201",
    "species": "Erithacus rubecula",  # Rotkehlchen
    "fat_score": 3,
    "wing_length": 72.0
}
# Accessing data efficiently
print(f"Fat Score: {bird_record['fat_score']}")
```

Kontrollstrukturen

Python 13 Basics

- `if elif else`: Standardmäßige bedingte Logik. Bedingung muss True ergeben.
- `for` **Loop:** Iteriert direkt über Elemente einer Sequenz (nicht über einen Index wie in C).
- **Scope:** Variablen, die in `if`-Blöcken oder Loops definiert werden, sind oft noch außerhalb sichtbar (im Gegensatz zu vielen anderen Sprachen).

Kontrollstrukturen

Python 13 Basics

```python
daily_weights = [18.5, 19.2, 17.8, 21.0]


for weight in daily_weights:
    if weight > 20.0:
        status = "High reserves"
    elif weight < 18.0:
        status = "Underweight"
    else:
        continue
    print(f"Weight {weight} is {status}")
```

Erweiterte Syntax

Python 13 Basics

- **List Comprehensions:** Eine prägnante Syntax, um Listen basierend auf existierenden Listen zu erstellen. Pythonischer und oft schneller als klassische Loops.

- **Type Hinting & DocStrings:** Essenziell für Wartbarkeit in Teams.

Erweiterte Syntax

Python 13 Basics

```
raw_data = [18.5, 19.2, 17.8, 21.0]

# List Comprehension: Convert all to integers (rounding down)
int_weights = [int(w) for w in raw_data]

def process_bird(ring: str) -> bool:
    """Validates ring format."""
    return len(ring) == 6
```

Demo

Python Grundlagen

Übung 1

Python Grundlagen & Datenstrukturen

- Übungsordner: /Labs/01-Python13Grundlagen

Modularisierung & Best Practises

Python Basics

Funktionen & Kapselung

Modularisierung & Best Practises

- **DRY-Prinzip:** "Don't Repeat Yourself". Wenn Sie Code mehrfach kopieren, gehört er in eine Function.
- **Parameter & Return:** Funktionen akzeptieren Inputs (Argumente) und geben Ergebnisse zurück. Vermeiden Sie globale Variablen innerhalb von Funktionen (Side Effects).
- **Docstrings:** Dokumentieren Sie was die Funktion tut direkt unter der Definition (`"""..."""`).
- **Default Arguments:** Parameter können Standardwerte haben, was die Flexibilität erhöht.

Funktionen & Kapselung

Modularisierung & Best Practises

```python
def calculate_condition_index(weight_g: float, wing_len_mm: float) -> float:
  """
  Calculates bird body condition index using scaled mass index concept.
  """
  if wing_len_mm == 0:
    # Liefere 0 um ZeroDivisionError zu vermeiden
    return 0.0
  return weight_g / wing_len_mm


# Usage
index = calculate_condition_index(18.5, 74.0)
```

Module & Imports

Modularisierung & Best Practises

- **Standard Library:** Python kommt "batteries included". Module wie `math`, `datetime` oder `os` sind sofort verfügbar.
- **Import-Strategien:**
  - `import module`: Importiert das ganze Modul (Namespace bleibt sauber).
  - `from module import function`: Importiert spezifische Teile (kürzerer Aufruf, aber Vorsicht vor Namenskonflikten).
- **Eigene Module:** Jede `.py` Datei ist ein Modul und kann von anderen Dateien importiert werden.

Module & Imports

Modularisierung & Best Practises

```
import math
from datetime import datetime


def get_banding_timestamp():
    # Returns current time in ISO format
    return datetime.now().isoformat()


# Using standard library math
wing_area = math.pi * (5.2 ** 2)
```

pip & Virtual Environments

Modularisierung & Best Practises

- **PyPI (Python Package Index)**: Das Repository für externe Libraries (z.B. `pandas` für Datenanalyse, `requests` für API-Calls).
- **Virtual Environments** (venv): Erstellt isolierte Umgebungen für Projekte.
  - *Problem*: Projekt A braucht Library Version 1.0, Projekt B braucht Version 2.0.
  - *Lösung*: Jedes Projekt hat sein eigenes Environment.
- **Best Practice:** Niemals global installieren, immer in ein Environment.

pip & Virtual Environments

Modularisierung & Best Practises

```
# 1. Create environment
python -m venv .venv

# 2. Activate environment (Windows)
.venv\Scripts\activate

# 3. Install package
pip install pandas
```

Skript-Struktur & Entry Points

Modularisierung & Best Practises

- **PEP 8:** Der Style Guide für Python Code (4 Leerzeichen Einrückung, Leerzeilen zwischen Funktionen, Importe ganz oben).
- **Main Guard:** `if __name__ == "__main__":`
  - Dieser Block wird nur ausgeführt, wenn das Skript direkt gestartet wird.
  - Er wird nicht ausgeführt, wenn das Skript als Modul importiert wird.
- **Struktur:** Imports -> Konstanten -> Klassen/Funktionen -> Main Block.

Skript-Struktur & Entry Points

Modularisierung & Best Practises

```
import csv


DEFAULT_SPECIES = "Unknown"


def main():
    print("Starting ringing session...")
    # Main logic calls here


if __name__ == "__main__":
    # Entry point
    main()
```

Exception Handling

Modularisierung & Best Practises

- **Robustheit:** Code darf bei fehlerhaften Daten nicht abstürzen.
- **Try / Except:** Fängt Fehler ab und definiert das Verhalten im Ausnahmefall.
- **Philosophie: EAFP** "It's easier to ask for forgiveness than permission". In Python probiert man es oft einfach (`try/except`) und fängt den Fehler, statt vorher alles zu prüfen (LBYL).

Exception Handling

Modularisierung & Best Practises

```
raw_weight = "18.5g" # Malformed data


try:
    # Try to convert to float
    weight = float(raw_weight)
except ValueError as e:
    # Handle the specific error
    print(f"Error reading scale: {e}")
    weight = None
```

Demo

Module & Exception handling

Übung 2

Modularisierung & Best Practices

- Übungsordner: /Labs/02-ModularisierungBestPractices

File Handling & Wichtige Libraries

Python Basics

File I/O & Context Managers

File Handling & Wichtige Libraries

- **Der** `open()` **Befehl**: Der klassische Weg, um Dateien zu öffnen. Modi: `'r'` (read), `'w'` (write - überschreibt!), `'a'` (append).
- **Der Context Manager** (`with`): **Best Practice.** Er garantiert, dass die Datei automatisch geschlossen wird, selbst wenn Fehler auftreten. Das verhindert Resource Leaks und korrupte Dateien.
- **Encoding:** Geben Sie immer `encoding='utf-8'` an, um Probleme mit Umlauten oder Sonderzeichen zu vermeiden.

File I/O & Context Managers

File Handling & Wichtige Libraries

```
# Reading raw field notes
log_file = "field_notes_2024.txt"

# The 'with' statement ensures the file closes safely
with open(log_file, mode='r', encoding='utf-8') as f:
    content = f.read()
    # File is open here

# File is automatically closed here
print(f"Read {len(content)} characters.")
```

CSV Verarbeitung

File Handling & Wichtige Libraries

- **Standard:** CSV (Comma Separated Values) ist das Austauschformat Nr. 1 für Tabellendaten.
- `csv` **Modul:** Bietet robustere Parser als einfaches String-Splitting (behandelt z.B. Kommas innerhalb von Textfeldern korrekt).
- **DictReader:** Liest Zeilen direkt in Dictionaries ein, wobei die Header-Zeile als Keys dient. Das macht den Code lesbarer und robuster gegen Spaltenverschiebungen.

CSV Verarbeitung

File Handling & Wichtige Libraries

```
import csv


# Reading ringing data
with open('capture_data.csv', mode='r', newline='') as csvfile:
    reader = csv.DictReader(csvfile)
    for row in reader:
        # Access by column name, not index
        print(f"Bird ID: {row['Ring_ID']} - Weight: {row['Weight']}")
```

JSON & Serialisierung

File Handling & Wichtige Libraries

- **JSON (JavaScript Object Notation):** Das Standardformat für Web-APIs und NoSQL-Datenbanken. Es bildet Python-Datentypen (Lists, Dicts, Strings, Numbers) fast 1:1 ab.

- **Workflow:**
  - `json.dump()`: Speichert Python-Objekte in eine Datei.
  - `json.load()`: Lädt Daten aus einer Datei in Python-Objekte.

- **Nesting:** JSON erlaubt tiefe Verschachtelungen (z.B. Liste von Messwerten innerhalb eines Vogel-Objekts).

JSON & Serialisierung

File Handling & Wichtige Libraries

```python
import json
bird_data = {
    "species": "Ciconia ciconia", # Weißstorch
    "rings": ["H8812", "GPS-Tracker-09"],
    "measurements": {"bill": 180, "wing": 590}
}

# Serialization (Writing to disk)
with open('stork_data.json', 'w') as f:
    json.dump(bird_data, f, indent=4)
```

Datetime Handling

File Handling & Wichtige Libraries

- **ISO 8601** (`YYYY-MM-DD`): Das einzig vernünftige Format für Datenaustausch. Vermeiden Sie lokale Formate wie `DD.MM.YYYY` in Rohdaten.
- **Parsing:** `datetime.strptime()` wandelt Strings in Zeit-Objekte um.
- **Rechnen:** Mit `datetime` Objekten können Sie rechnen (z.B. `capture_time - release_time = duration`).

Datetime Handling

File Handling & Wichtige Libraries

```
from datetime import datetime, timedelta


raw_date = "2024-05-12 14:30:00"
# Parse string to object
capture_time = datetime.strptime(raw_date, "%Y-%m-%d %H:%M:%S")


# Calculate release time (20 mins later)
release_time = capture_time + timedelta(minutes=20)


print(f"Release ISO: {release_time.isoformat()}")
```

Regex

File Handling & Wichtige Libraries

- **Regular Expressions** (`re`): Mächtiges Tool zur Mustererkennung in Strings.
- **Use Case:** Extrahieren von Ring-IDs aus unstrukturierten Kommentaren oder Validierung von Eingabeformaten.
- **Wichtige Funktionen:**
  - `re.search()`: Sucht das erste Vorkommen.
  - `re.findall()`: Findet alle Vorkommen.
  - `re.sub()`: Ersetzt Muster (Suchen & Ersetzen).

Regex

File Handling & Wichtige Libraries

```
import re

notes = "Bird spotted with ring AX-9921 near the lake, maybe AX-9922 too."

# Pattern: 2 Letters, hyphen, 4 Digits
pattern = r"[A-Z]{2}-\d{4}"

found_rings = re.findall(pattern, notes)
# Result: ['AX-9921', 'AX-9922']
```

Demo

File Handling & Regex

Übung 3

File Handling & Libraries

- Übungsordner:
  /Labs/03-FileHandlingImportantLibraries

Unstrukturierte Daten

Python Basics

Word-Automatisierung

Unstrukturierte Daten

- **Struktur:** `.docx` Dateien sind technisch gesehen gezippte XML-Dateien. Die Library `python-docx` abstrahiert dies in `Document`, `Paragraph` und `Run` Objekte.
- **Reading:** Zugriff erfolgt meist iterativ über alle Paragraphen. Formatierungen (Fett, Kursiv) sind in sogenannten "Runs" gespeichert.
- **Anwendung:** Ideal zum Auslesen von standardisierten Protokollen, die von Freiwilligen als Word-Datei eingereicht wurden.

Word-Automatisierung

Unstrukturierte Daten

```
from docx import Document
# Load the field report
doc = Document('ringing_protocol_2024.docx')

full_text = []
for para in doc.paragraphs:
    # Extract only text, ignore style for now
    if "Species:" in para.text:
        full_text.append(para.text)

print(f"Extracted {len(full_text)} lines.")
```

Word-Reports Generieren

Unstrukturierte Daten

- **Writing:** `python-docx` kann neue Dokumente erstellen, Bilder einfügen und Tabellen generieren.
- **Templating:** Für komplexe Layouts ist es effizienter, ein "Template-Dokument" zu laden (mit Kopfzeilen/Logos) und nur den Inhalt dynamisch zu ergänzen.
- **Business Value:** Automatisierte Erstellung von behördlichen Beringungsberichten ("Ring Fund Report") auf Knopfdruck.

Word-Reports Generieren

Unstrukturierte Daten

```
from docx import Document
doc = Document() # Creates new blank doc
doc.add_heading('Annual Ringing Report', 0)

# Add a table with data
data = [('A12', 'Blaumeise'), ('B99', 'Rotkehlchen')]
table = doc.add_table(rows=1, cols=2)

# Set header
hdr_cells = table.rows[0].cells
hdr_cells[0].text = 'Ring ID'
hdr_cells[1].text = 'Species'
```

Word-Reports Generieren

Unstrukturierte Daten

```
# ...


# Fill rows
for ring_id, species in data:
    row_cells = table.add_row().cells
    row_cells[0].text = ring_id
    row_cells[1].text = species


doc.save('report_output.docx')
```

PDF Text Extraktion

Unstrukturierte Daten

- **Herausforderung:** PDF ist ein Layout-Format, kein Daten-Format. Es gibt keine logische Struktur wie "Absätze" oder "Tabellen", nur Buchstaben an Koordinaten.
- **pypdf:** Robuste Library zum Lesen, Splitten und Mergen von PDFs.
- **Text Extraction:** `extract_text()` versucht, den visuellen Textfluss in einen String zu rekonstruieren. Das Ergebnis ist oft unstrukturiert (Header vermischen sich mit Content).

PDF Text Extraktion

Unstrukturierte Daten

```python
from pypdf import PdfReader

reader = PdfReader("scientific_paper_v1.pdf")
page = reader.pages[0]

# Extract text content
raw_text = page.extract_text()

# Basic filtering
if "Parus major" in raw_text:
   print("Found mention of Kohlmeise on page 1")
```

Tabellen aus PDFs

Unstrukturierte Daten

- **Problem:** pypdf scheitert oft an Tabellenlayouts.
- **Lösung (`pdfplumber`):** Eine spezialisierte Library, die Linien und Abstände im PDF analysiert, um Tabellenstrukturen visuell zu erkennen.
- **Visual Debugging:** `pdfplumber` erlaubt es, die erkannten Bounding-Boxes grafisch darzustellen, um die Extraktionslogik zu verfeinern.

Tabellen aus PDFs

Unstrukturierte Daten

```
import pdfplumber

# Opening a PDF containing morphometric tables
with pdfplumber.open("data_sheet.pdf") as pdf:
 first_page = pdf.pages[0]

 # Returns a list of lists (rows/columns)
 table_data = first_page.extract_table()

 for row in table_data:
   # Filter out None values or empty rows
   if row and row[0] != "Date":
     print(f"Date: {row[0]}, Count: {row[1]}")
```

Demo

Word & PDF

Übung 4

Unstrukturierte Daten

- Übungsordner: /Labs/04-UnstrukturierteDaten

Recap Tag 1

Python Basics

Fundamente & Datenfluss

Recap Tag 1

- **Datenstrukturen:** Wir nutzen `Lists` für Sequenzen und `Dictionaries` für strukturierte Datensätze.
- **Sauberer Code:**
  - **Modularisierung:** Logik gehört in Funktionen (DRY-Prinzip) und Module.
  - **Type Hinting:** Erhöht die Lesbarkeit und IDE-Support
    ```
    def func(x: int) -> str:
    ```
- **File I/O:** Der Context Manager (`with open(...)`) garantiert Datensicherheit beim Lesen/Schreiben von CSV und JSON.

Fundamente & Datenfluss

Recap Tag 1

```python
import json
def save_capture(bird_data: dict, filename: str) -> None:
 """Appends a bird record to a JSON log safely."""

 # Context Manager handles opening/closing
 with open(filename, 'a', encoding='utf-8') as f:
   json.dump(bird_data, f)
   f.write('\n')  # Newline for JSONL format

record = {"species": "Parus major", "ring": "AX-99", "weight": 18.5}
save_capture(record, "daily_log.json")
```