Python Grundlagen

Julian Huber & Matthias Panny

Pakete und Umgebungen

🎯 Lernziele

- Studierende können hilfreiche Pakete finden & installieren
- Studierende können Virtuelle Umgebungen anlegen
- Studierende können Python-Code in VS Code debuggen

Pakete

Motivation

- Bei größeren Software-Projekten gibt es eine Vielzahl an Modulen, die teilweise verwandt und teilweise sehr unabhängig voneinander sind
- Python-Pakete sind Sammlungen von wiederverwendbaren Python-Modulen, die Funktionen und Klassen bereitstellen, um Aufgaben zu erledigen.

Erklärung

- Ein Python-Paket ist eine Verzeichnisstruktur mit einer speziellen `__init__.py`-Datei
- Module in einem Paket können mithilfe von `import` in anderen Python-Dateien verwendet werden.

Pakete

- Die Syntax beim Laden wird dabei um eine Hierarchie-Ebene erweitert

Beispiel

```
# Beispiel: Importieren eines Moduls aus einem Paket
from mein_paket import mein_modul
mein_modul.meine_funktion()
```

🤓 Eigene Pakete

- Installierbare Pakete können selbst erstellt werden
- Hierzu wird ein Verzeichnis mit:
  - `pyproject.toml`-Datei
  - Quellcode unserer Module
  - `__init__.py`-Datei
- Diese Paket kann dann mit `pip install .` installiert werden
- Beispiel: my_example_package

PIP Paket Manager

Motivation

- Vorteil von Python ist Vielzahl an sehr guten Open Source Paketen für verschiedene Anwendungen → einfach zu integrieren
- PIP (`pip`) ist ein Paketmanager für Python → Verwaltet Python-Pakete und vereinfacht deren Installation

Python Package Index (PyPi) und `pip`

- pypi.org → Sammlung an Pakten die mit `pip` installiert werden können → z.B. numpy
- `pip` wurde automatisch mit Python installiert → `pip --version` auf der Kommandozeile ausführbar
- `pip` kann direkt alle Pakete aus dem Python Package Index installieren

PIP Paket Manager

Versuchtes Laden eines nicht-installierten Paketes

```
import seaborn
#>ModuleNotFoundError: No module named 'seaborn'
```

Installieren eines Paketes

- `pip` ist ein Kommandozeilen-Tool → Installation von Paketen erfolgt im Terminal, nicht im Python-Code
- Solange eine Internetverbindung besteht kann direkt mit `pip install seaborn` das Paket von pypi installiert werden

Laden eines installierten Paketes

- das `as` definiert einen Shortcut, um das Paket mit weniger tippen anzusprechen

```
import seaborn as sns
# Load an example dataset
tips = sns.load_dataset("tips")
```

PIP Paket Manager

- `pip` wir üblicherweise automatisch installiert, wenn Python installiert wird → wir müssen nichts weiter tun

Troubleshooting falls `pip` nicht gefunden wird

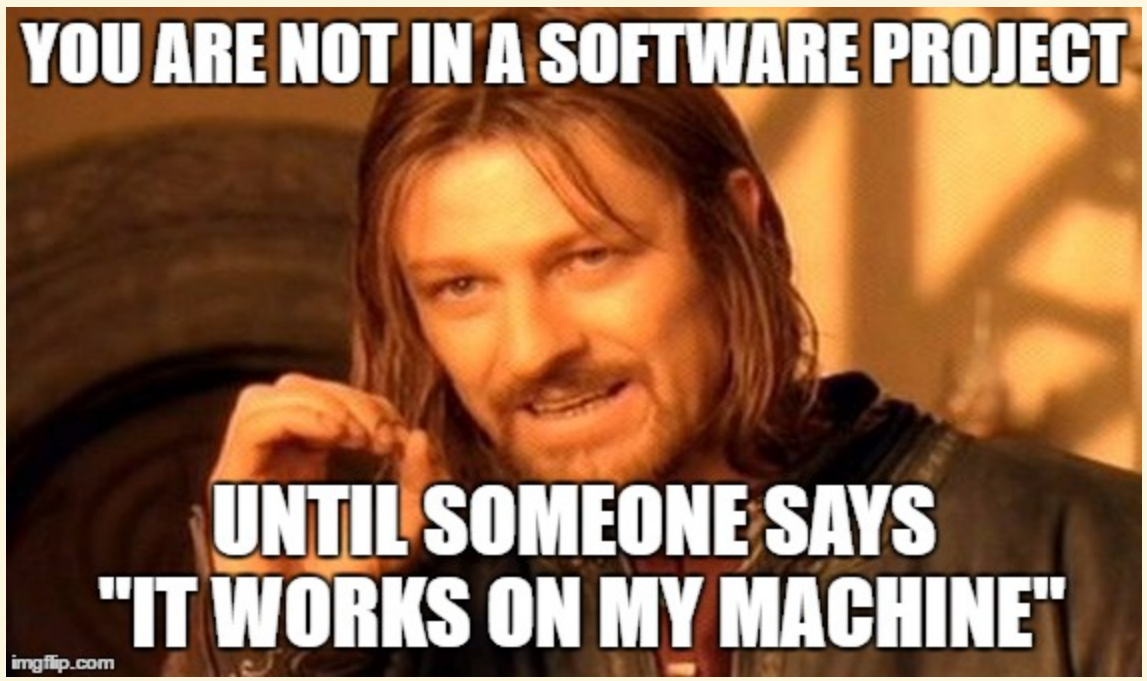
- `pip` wird im Python-Installationsverzeichnis installiert → überprüfen ob dies der Fall ist & ggf. den Pfad hinzufügen
- Nachträgliche Installation von `pip` mit `python -m ensurepip`
- Update einer veralteten Version mit `python -m ensurepip --upgrade`
- ggfs. auch `python -m pip install --upgrade pip`

Virtuelle Python Umgebungen

(engl. virtual environments)

Virtuelle Python Umgebungen

- moderne Softwareprojekte greifen meinst auf eine Vielzahl von bestehenden Open Source Paketen zurück → Pakete haben wiederum Abhängigkeiten → Konflikte sind vorprogrammiert



Entwicklungs-System != Production-System

- Unterschiedliche Python Versionen
- Unterschiedliche Versionen von Python-Paketen
- Unterschiedliche Betriebssysteme (Windows, Linux, etc.)

Virtuelle Python Umgebungen

- Wollen unsere Projekte abkapseln und voneinander isolieren → vermeidet Konflikte zwischen Abhängigkeiten
- Wollen sicherstellen, dass unsere Software auf verschiedenen Systemen funktioniert → Übertragbarkeit
- Wollen "Rezept" um unsere Umgebung zu reproduzieren → wechsel zwischen verschiedenen Projekten

Lösung: Virtuelle Umgebungen

- Jedes Projekt hat seine eigene Umgebung → projektspezifische Installation/Version von Python
- Jede Umgebung hat ihre eigenen Pakete
- Dokumentation der installierten Pakete

Virtuelle Python Umgebungen

Anlegen

- erfolgt im Terminal mittels Modul `venv`:
  `python -m venv <umgebungsname>`
  - `<umgebungsname>` ist ein Platzhalter für den Namen der Umgebung hier z.B. `.venv` eingeben
- ⚠ Vorsicht: hierdurch wird ein neuer Ordner mit über 1000 Dateien in Projektverzeichnis angelegt!

Aktivieren

- Windows: `<umgebungsname>\Scripts\activate`
- unix / Mac: `source <umgebungsname>/bin/activate`
- Sobald aktiviert sollte der Umgebungsname in der Kommandozeile erscheinen

Deaktivieren

- `deactivate`

Virtuelle Python Umgebungen

Probleme beim Aktivieren

- Es kann sein, dass das `activate` Skript auf Ihrem Computer aus Sicherheitsgründen nicht ausgeführt werden darf
- Schritte zur Behebung:
  - `PowerShell` als Administrator ausführen
  - Folgenden Befehl ausführen:

```
Set-ExecutionPolicy -ExecutionPolicy Unrestricted
```

Virtuelle Python Umgebungen

Beispiel

- `python -m venv .venv`
  → (`.` ist unix-konvention für verstecken Ordner)
- `(.venv)` wird in der Kommandozeile angezeigt → Umgebung `.venv` ist aktiviert
- `pip list` zeigt alle in der Umgebung installierten

```
Package    Version
---------- -------
pip        23.0.1
setuptools 65.5.0
```

- Standard-Pakete wie `pip` und `setuptools` sind bereits installiert
  - `setuptools` erleichtert die Installation von (eigenen) Paketen

🎨 Aufgabe

- Führen Sie `pip list` in Ihrer aktuellen (globalen) Umgebung aus
- Legen Sie eine neue virtuelle Umgebung mit beliebigem Namen an
- Aktivieren Sie die Umgebung und geben Sie `pip list` erneut ein
- Verlassen Sie die Umgebung

Virtuelle Python Umgebungen

- Wir wollen nicht alle Pakete händisch installieren müssen wenn wir unser Projekt auf einem anderen System ausführen wollen
- `pip` ermöglicht das Speichern der installierten Pakete & derer Version in einer Datei → `requirements.txt`
- Voraussetzung ist eine aktivierte Umgebung → erkennbar an dem vorangestellten Umgebungsname in der Kommandozeile

Installierte Pakete in `requirements.txt` speichern

```
(<umgebungsname>)
$ pip freeze > requirements.txt
```

Pakete aus `requirement.txt` in Umgebung installieren

```
(<umgebungsname>)
$ pip install -r requirements.txt
```

✍️ Aufgabe

- Setzten sie einen geeigneten Standard-Interpreter für ihr VS Code.
- Öffnen Sie dazu das Menü mittels `Shift` + `Strg` + `P` und suchen Sie `Preferences: Open User Settings`. Nun Können Sie im Bereich Python den Pfad zum `python.defaultInterpreterPath` setzen.

Debugging

Debugging

 Was ist Debugging?

- Debugging ist der Prozess des Findens und Behebens von Fehlern (Bugs) in Ihrem Programmcode
- Dazu wird der Code ausgeführt und dabei ein tiefere Blick *unter die Haube* geworfen
- z.B. indem Variablen-Werte überwacht werden

Warum ist Debugging wichtig?

- Fehlerbehebung: Debugging hilft dabei, Fehler in Ihrem Code zu finden und zu beheben, um sicherzustellen, dass Ihr Programm wie erwartet funktioniert

Debugging

Wie funktioniert Debugging in VS Code?

- Setzen von Breakpoints: Klicken Sie auf die linke Seite Ihres Codes, um Breakpoints zu setzen, an denen Ihr Programm anhalten soll
- Starten des Debuggers: Klicken Sie auf das "Run and Debug" Symbol oder drücken Sie F5, um den Debugger zu starten.
- Auch jetzt wird der Code interpretiert und ausgeführt, allerdings können wir einige Einblicke mehr nehmen als sonst

Debugging

Debugging-Schritte in VS Code

- ● Breakpoints sind Punkte zu denen die Codeausführung in der Laufzeit gestoppt wird, damit wir uns den Systemzustand näher anschauen können
- Klicken Sie dazu links in die Zeile bei der Sie einen Breakpoint setzen wollen

Überwachung von Variablen und Ausdrücken

- Links werden die aktuellen Variablenwerte angezeigt
- zudem können die spezielle Ausdrücke (die Variablen enthalten können) automatisch ausgewertet werden

Steuerung des Debuggers

- grundsätzlich wird der Code mit wem `weiter`-Befehl immer nur bis zum nächsten Breakpoint ausgeführt
- von dort kann man die Ausführung des Codes durch erneutes Wählen des `weiter`-Befehls bis zum Ende oder nächsten Breakpoint durchführen
- Schrittweise Ausführung: Verwenden Sie `Prozedurschritt`, um Zeile für Zeile durch Ihren Code zu gehen, ohne in Funktionen einzutreten
- Schritt in Funktionen: Mit `Einzelschritt` [sic!] können Sie in Funktionen eintreten und deren Ausführung verfolgen
- Schritt aus Funktionen: Mittels `Ausführen bis Rücksprung` verlassen Sie eine Funktion und begeben sich an die Stelle, wo die Funktion aufgerufen wurde

Fehlerbehebung und Verbesserung

- Wenn Sie eine andere Lösung testen wollen können Sie:
  - entweder den Ausdruck direkt im `Überwachen`-Fenster testen
  - oder den Ausdruck direkt im Code ändern und den Code erneut ausführen

✍️ Aufgabe

- Legen Sie die folgende Datei `debug.py` an
- Setzen Sie einen Breakpoint vor der dritten Zeile
- Führen die das Debugging aus
- Nutzen Sie den Einzelschritt, um in den Fehler zu laufen
- Legen Sie eine Überwachung an
- Beheben Sie den Fehler indem Sie die Zeile auskommentieren
- Verlassen Sie den Debug-Modus

Ausgangslage - `debug.py`

```
print("Hello, World!")
x = 1
x = x / 0
print(x)
x = 2
```