https://github.com/rhkraptor/sd_chap_10

Sharing Your Code: Version Control, Dependencies, and Packaging

Agenda

- Code teilen
- Versionskontrolle mit Git
- Dependencies and Virtual Environments
- Python Packaging
- Kahoot

https://github.com/rhkraptor/sd_chap_10

Code teilen

- Code teilen ist essenziell für die Zusammenarbeit in der Data Science
- Einstieg in bestehende Projekte oder eigene Projekte skalieren
- Ziel: Redundante Arbeit vermeiden, Probleme gemeinsam lösen
- Open-Source: Zugang zu riesigen Python-Bibliotheken (z. B. pandas, NumPy)
- Standardisierte Tools und Prinzipien sind entscheidend

https://github.com/rhkraptor/sd_chap_10

Versionskontrolle mit Git

Was ist Versionskontrolle? (Version Control)

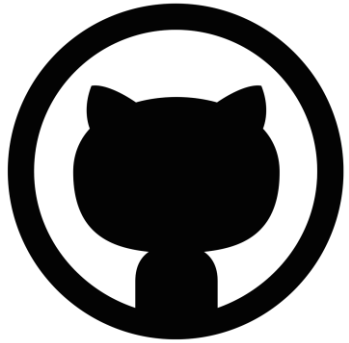
- Versionskontrolle = Änderungen am Code dokumentieren
- Erlaubt: Rückverfolgbarkeit, Zusammenarbeit, Fehlerkorrektur
- Vergleichbar mit „Speichern unter“ – aber professionell
- Beispiel: Bug gemacht? → Zur funktionierenden Version zurückkehren
- Unverzichtbar bei Teamarbeit & großen Projekten

https://github.com/rhkraptor/sd_chap_10

Versionskontrolle mit Git

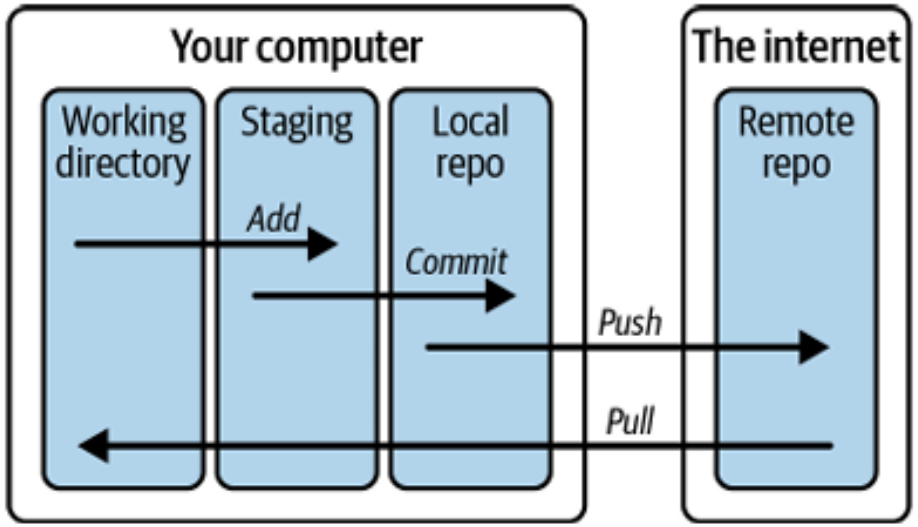
Git und GitHub – das Power-Duo

- Git = Tool zur Versionskontrolle (seit 2005, von Linus Torvalds)
- Open Source, dezentral, lokal & remote
- GitHub = Plattform zur Speicherung & Zusammenarbeit (Alternativen: GitLab, Bitbucket)
- Git ≠ GitHub – Git funktioniert auch ohne Plattform



Laut der **Stack Overflow Developer Survey 2022** verwenden **96 %** der professionellen Entwickler:innen **Git** als Versionskontrollsystem.
Andere Systeme: Subversions, Mercurial

Versionskontrolle mit Git



add → Änderungen vormerken
commit → Snapshot im lokalen Repo speichern
push / pull → Synchronisation mit Remote-Repo (z. B. GitHub)

Wie funktioniert Git? (Git Workflow verstehen)

- Git speichert **Snapshots** deines Codes (nicht nur Unterschiede)
- Projekt wird in einem **Repository** (Verzeichnis) organisiert
- .git Ordner enthält die Versionshistorie
- Drei zentrale Bereiche:
    - **Working Directory** (lokale Arbeitskopie)
    - **Staging Area** (Zwischenspeicher vor dem Speichern)
    - **Local Repository** (lokales Archiv)

https://github.com/rhkraptor/sd_chap_10

Versionskontrolle mit Git

Änderungen nachverfolgen & Committen (Tracking Changes and Committing)

- Projekt starten**:$ git init** → Lokales Git-Repository initialisieren
- Änderungen hinzufügen**:$ git add** README.md → in Staging Area
- Status prüfen**:$ git status** → Zeigt was vorgemerkt ist
- Commit erstellen (Snapshot speichern)**:$ git commit -m** "Initial commit„

Best Practices:

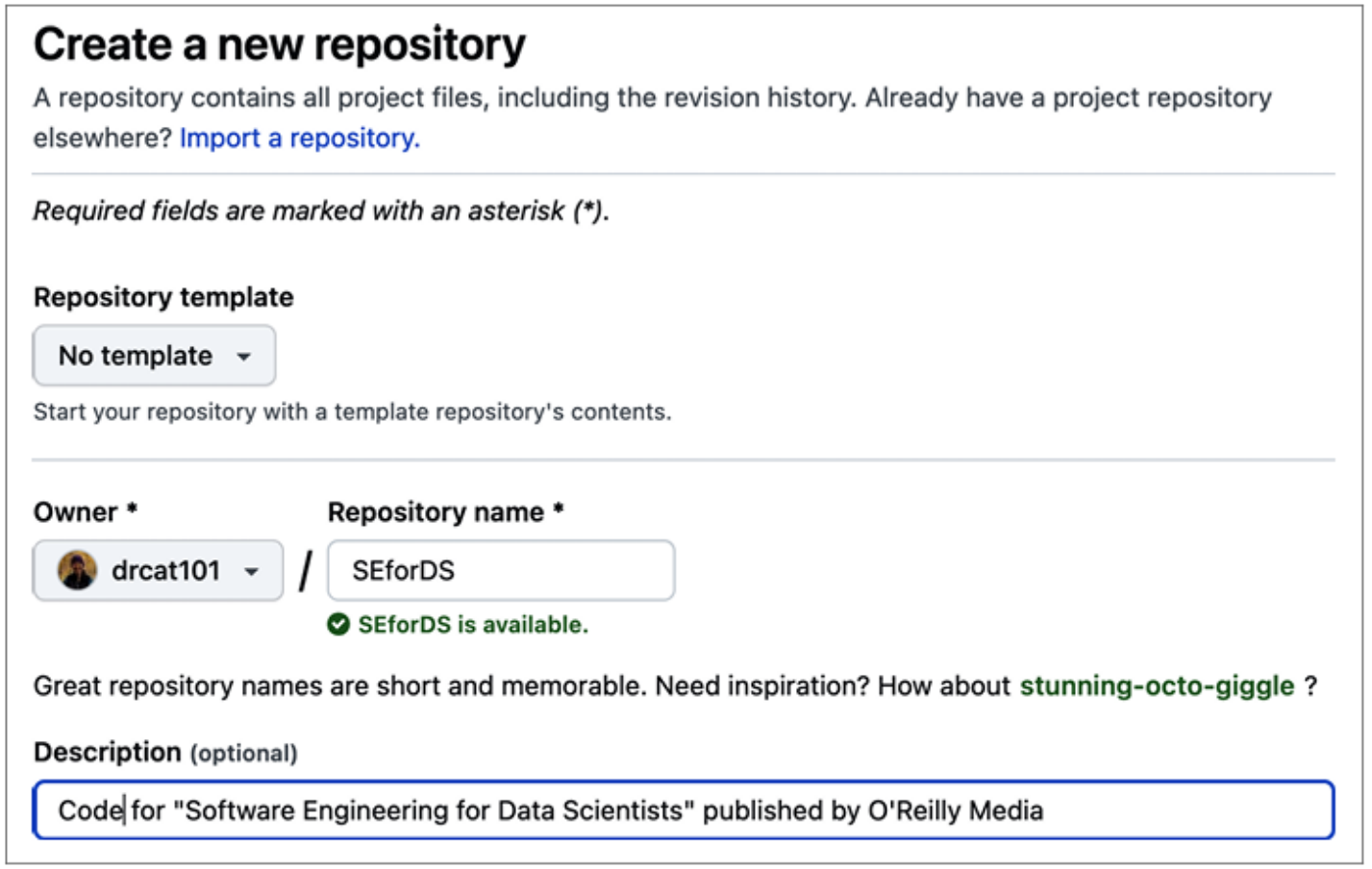
- Ein Commit = eine Änderung / Funktion
- Aussagekräftige Commit-Messages
- Vor jedem Commit: Tests laufen lassen

https://github.com/rhkraptor/sd_chap_10

Versionskontrolle mit Git

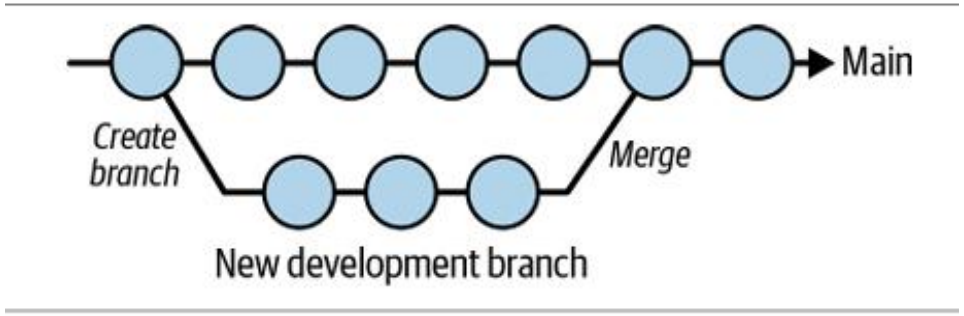
Lokales vs. Remote-Repository

- Lokales Repository = dein Code auf deinem Rechner
- Remote Repository = zentraler Ort z. B. auf GitHub
- Verbindung per**:$ git remote add origin** <URL>
- Hochladen mit**:$ git push -u origin main**
- Projekt herunterladen mit**:$ git clone** <URL>
- Clone via HTTPS (einfach) oder SSH (sicherer)



https://github.com/rhkraptor/sd_chap_10

Versionskontrolle mit Git



Branches – Neues testen ohne Risiko

- Branch = Entwicklungszweig, isoliert vom Hauptcode
- Keine Kopie → nur Änderungshistorie wird verfolgt
- Hauptbranch: meist ***main*** (nicht mehr ***master***)

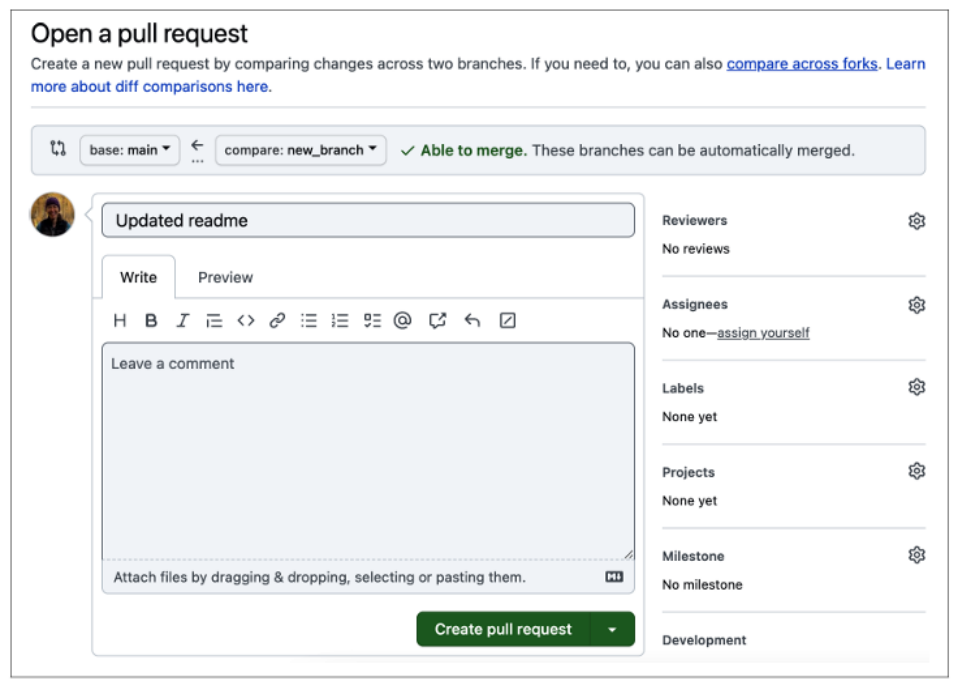
Wechsel zwischen Branches:

- **$ git branch** new_branch
- **$ git checkout** new_branch oder **git checkout -b** new_branch
- **$ git push origin** new_branch
- **$ git checkout** main
- **$ git merge** new_branch, änderungen später via **merge** zurück in **main**

https://github.com/rhkraptor/sd_chap_10

Aufgabe 1 & 2

https://github.com/rhkraptor/sd_chap_10



Versionskontrolle mit Git

Pull Requests & Code Reviews

- Änderungen im Branch sollen überprüft werden? → Pull Request
- Erstellung meist über GitHub-UI
- Vorteile:
  - Kollaboration & Diskussion vor dem Merge
  - Feedback & Fehlererkennung
  - Dokumentation von Entscheidungswegen

- **git push origin** branchname → Code hochladen

- Gute Pull Requests: klar, kommentiert, nachvollziehbar

https://github.com/rhkraptor/sd_chap_10

Versionskontrolle mit Git

Konflikte

Merge-Konflikte: zwei Branches ändern dieselbe Zeile

Besonders problematisch bei Jupyter Notebooks

Lösungen:

- Outputs vor Merge leeren
- Tools wie **nbdime** oder **jupytext** verwenden

https://github.com/rhkraptor/sd_chap_10

Dependencies and Virtual Environments

Was sind Abhängigkeiten & warum sind sie wichtig?

- Abhängigkeit (Dependency) = externe Bibliothek wie pandas, NumPy
- Andere brauchen dieselben Versionen → Reproduzierbarkeit
- Versionierung: z. B. NumPy==1.24.3
- Versionskonflikte vermeiden durch genaues Festlegen
- Tools helfen, Versionen automatisch zu verwalten

https://github.com/rhkraptor/sd_chap_10

Dependencies and Virtual Environments

Versionierung verstehen – SemVer

- Versionen geben Änderungen strukturiert an: SemVer (Semantic Versioning)
- Aufbau: MAJOR.MINOR.PATCH (z. B. 1.2.3)
- MAJOR → Breaking Changes
- MINOR → Neue Features ohne Brüche
- PATCH → Kleine Fixes & Bugbehebungen
- Auch verbreitet: CalVer (Kalenderbasiert)

Dependencies and Virtual Environments

Virtuelle Umgebungen – Isolierte Welten

- Virtuelle Umgebung = isolierter Raum für Libraries
- Tools: venv, virtualenv, conda, pyenv, poetry, pdm, hatch
- Vorteil: mehrere Versionen auf einem System möglich
- Aktivieren: **$ source myenv/bin/activate**
- Deaktivieren: **(myenv)$ deactivate**

Dependencies and Virtual Environments

Abhängigkeiten verwalten mit pip & requirements.txt

- Abhängigkeiten speichern:**$ python -m pip freeze > requirements.txt**
- Installation in neuer Umgebung:**$ pip install -r requirements.txt**

Nachteile:

- Subdependencies nicht automatisch bereinigt
- Keine Info über Python-Version
- Manuelle Pflege nötig

Dependencies and Virtual Environments

Modern & mächtig – Abhängigkeiten mit Poetry

- Poetry = Dependency- und Packaging-Manager in einem **pyproject.toml** zentrale Konfigurationsdatei
- Bibliothek hinzufügen:**$ poetry add pandas**
- Aktivieren der Umgebung:**$ poetry shell**
- Installieren in neuer Umgebung:**$ poetry install**

poetry.lock speichert exakte Paket-Versionen & Hashes

https://github.com/rhkraptor/sd_chap_10

Augabe 3 & 4

https://github.com/rhkraptor/sd_chap_10

Python Packaging

Warum Python Packaging?

- Packaging = eigenen Code wie pandas oder NumPy installierbar machen
- Vorteile:
  - Wiederverwendbarkeit
  - Reproduzierbarkeit
  - Zusammenarbeit im Team oder öffentlich via PyPI

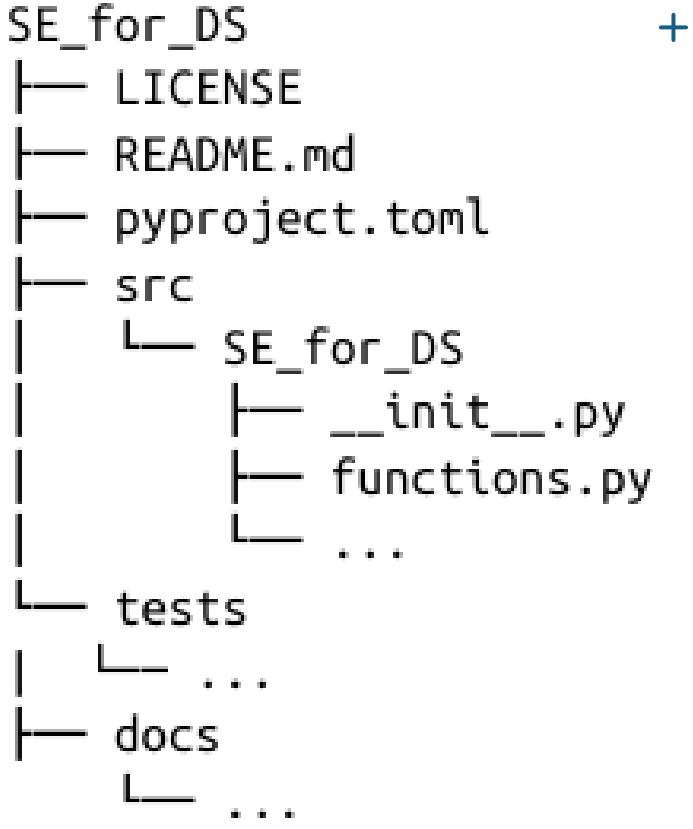
- PyPI = zentrales Archiv (über 470.000 Pakete!)
- Private oder interne Distribution ebenfalls möglich
- Verantwortung als Maintainer bei öffentlichen Paketen

https://github.com/rhkraptor/sd_chap_10

Python Packaging

Aufbau eines Python-Pakets

- __init__.py signalisiert: Das ist ein Package
- Trennung von Code, Tests und Doku
- pyproject.toml: zentrale Konfigurationsdatei
- Vor dem Packaging: testen, dokumentieren, sauberer Code!



https://github.com/rhkraptor/sd_chap_10

Python Packaging

pyproject.toml – Das Herzstück

- .toml = Konfigurationsformat für Python-Projekte
- Enthält: Metadaten, Abhängigkeiten, Build-Tool

Poetry schreibt auch pyproject.toml, aber mit eigenem Block:

```
[tool.poetry]
name = "se-for-ds"
version = "0.1.0"
description = ""
authors = ["Catherine Nelson <email_address>"]
readme = "README.md"

[tool.poetry.dependencies]
python = "^3.10"
pandas = "^2.1.0"

[build-system]
requires = ["poetry-core"]
build-backend = "poetry.core.masonry.api"
```

https://github.com/rhkraptor/sd_chap_10

```
[build-system] ❶
requires = ["setuptools>=61.0"]
build-backend = "setuptools.build_meta"

[project] ❷
name = "se_for_ds"
version = "0.0.1"
authors = [
  { name="Catherine Nelson", email="email_address" },
]
description = "An example package for Software Engineering
for Data Scientists"
readme = "README.md"
requires-python = ">=3.9"
classifiers = [
    "Programming Language :: Python :: 3",
    "License :: OSI Approved :: MIT License",
    "Operating System :: OS Independent",
]

[project.urls]
"Homepage" = "https://github.com/pypa/sampleproject"
```

Python Packaging

- Build-Tools: setuptools, build, poetry, hatch
- Build starten:
  - pip install build
  - python3 -m build
- Ergebnisse:
  - .tar.gz = Source Distribution
  - .whl = installierbare „Wheel“-Datei
- Hochladen:
  - pip install twine
  - Test: twine upload -r testpypi dist/*
  - Produktion: twine upload dist/*

Poetry-Alternative:

  - poetry build
  - poetry publish

https://github.com/rhkraptor/sd_chap_10

Python Packaging

Dein Code als Paket

- Packaging = professioneller Weg, Code zu teilen
- Struktur, Metadaten & Dokumentation sind Pflicht
- Tools:
  - pyproject.toml = zentrales Setup
  - build, setuptools, twine, poetry
- Veröffentlichung möglich über:
  - PyPI (öffentlich)
  - TestPyPI (vorher testen!)
  - Internes Repo (z. B. im Unternehmen)

Aufgabe 5 & 6

https://github.com/rhkraptor/sd_chap_10

Kahoot Quiz

https://github.com/rhkraptor/sd_chap_10