合肥大學
HEFEI UNIVERSITY

Programming with Python

18. Zwischenspiel: Der Linter Ruff

Thomas Weise (汤卫思)
tweise@hfuu.edu.cn · https://thomasweise.github.io

School of Artificial Intelligence and Big Data
Hefei University
Hefei, Anhui, China

人工智能与大数据学院
合肥大学
中国安徽省合肥市

Version: 2026-11-11

Programming with Python



Dies ist ein Kurs über das Programmieren mit der Programmiersprache Python an der Universität Hefei (合肥大学).

Die Webseite mit dem Lehrmaterial dieses Kurses ist https://thomasweise.github.io/programmingWithPython (siehe auch den QR-Kode unten rechts). Dort können Sie das Kursbuch (in Englisch) und diese Slides finden. Das Repository mit den Beispielprogrammen in Python finden Sie unter https://github.com/thomasWeise/programmingWithPythonCode.

Outline



1. Einleitung
2. Mypy
3. Ruff
4. Zusammenfassung



Einleitung



Einleitung



- Vor Kurzem haben wir gelernt, dass statische Kodeanalyse helfen kann, Fehler im Kode zu finden.


```
1 my_list: list[str] = list([1, 2, 3])
2 print(my_list)
```

↓ `python3 lists_error.py` ↓

```
1 [1, 2, 3]
```

Einleitung

- Vor Kurzem haben wir gelernt, dass statische Kodeanalyse helfen kann, Fehler im Kode zu finden.
- Jetzt haben wir begonnen, mit komplexeren Datenstrukturen zu arbeiten.


```
1 my_list: list[str] = list([1, 2, 3])
2 print(my_list)
```

↓ `python3 lists_error.py` ↓

```
1 [1, 2, 3]
```

Einleitung

- Vor Kurzem haben wir gelernt, dass statische Kodeanalyse helfen kann, Fehler im Kode zu finden.
- Jetzt haben wir begonnen, mit komplexeren Datenstrukturen zu arbeiten.
- Daher gibt es nun auch neue Arten von Problemen und Fehlern, die auftauchen können.

```
1 my_list: list[str] = list([1, 2, 3])
2 print(my_list)
```

↓ `python3 lists_error.py` ↓

```
1 [1, 2, 3]
```

Einleitung



- Vor Kurzem haben wir gelernt, dass statische Kodeanalyse helfen kann, Fehler im Kode zu finden.
- Jetzt haben wir begonnen, mit komplexeren Datenstrukturen zu arbeiten.
- Daher gibt es nun auch neue Arten von Problemen und Fehlern, die auftauchen können.
- Schauen wir uns das Programm `lists_error.py` an.

```
1 my_list: list[str] = list([1, 2, 3])
2 print(my_list)
```

↓ `python3 lists_error.py` ↓

```
1 [1, 2, 3]
```

Einleitung



- Vor Kurzem haben wir gelernt, dass statische Kodeanalyse helfen kann, Fehler im Kode zu finden.
- Jetzt haben wir begonnen, mit komplexeren Datenstrukturen zu arbeiten.
- Daher gibt es nun auch neue Arten von Problemen und Fehlern, die auftauchen können.
- Schauen wir uns das Programm `lists_error.py` an.
- Es kann problemlos ausgeführt werden.

```
1 my_list: list[str] = list([1, 2, 3])
2 print(my_list)
```

↓ `python3 lists_error.py` ↓

```
1 [1, 2, 3]
```

Einleitung



- Vor Kurzem haben wir gelernt, dass statische Kodeanalyse helfen kann, Fehler im Kode zu finden.
- Jetzt haben wir begonnen, mit komplexeren Datenstrukturen zu arbeiten.
- Daher gibt es nun auch neue Arten von Problemen und Fehlern, die auftauchen können.
- Schauen wir uns das Programm `lists_error.py` an.
- Es kann problemlos ausgeführt werden.
- Und trotzdem ist es fehlerhaft. Können Sie einen Fehler finden?

```
1 my_list: list[str] = list([1, 2, 3])
2 print(my_list)
```

↓ `python3 lists_error.py` ↓

```
1 [1, 2, 3]
```

Mypy



Prüfung mit Mypy

- Das Erste, was wir immer machen, ist unseren Kode mit Mypy[14] prüfen.

Prüfung mit Mypy

```
$ mypy lists_error.py --no-strict-optional --check-untyped-defs
lists_error.py:1: error: List item 0 has incompatible type "int";
    ↪ expected "str"  [list-item]
lists_error.py:1: error: List item 1 has incompatible type "int";
    ↪ expected "str"  [list-item]
lists_error.py:1: error: List item 2 has incompatible type "int";
    ↪ expected "str"  [list-item]
Found 3 errors in 1 file (checked 1 source file)
# mypy 2.4.1 failed with exit code 1.
```

- Das Erste, was wir immer machen, ist unseren Kode mit Mypy[14] prüfen.
- Klar: Wir definieren `my_list` als Liste von Strings und speichern dann `ints` drin.

Prüfung mit Mypy

```
$ mypy lists_error.py --no-strict-optional --check-untyped-defs
lists_error.py:1: error: List item 0 has incompatible type "int";
    ↪ expected "str"  [list-item]
lists_error.py:1: error: List item 1 has incompatible type "int";
    ↪ expected "str"  [list-item]
lists_error.py:1: error: List item 2 has incompatible type "int";
    ↪ expected "str"  [list-item]
Found 3 errors in 1 file (checked 1 source file)
# mypy 2.4.1 failed with exit code 1.
```

- Das Erste, was wir immer machen, ist unseren Kode mit Mypy[14] prüfen.
- Klar: Wir definieren `my_list` als Liste von Strings und speichern dann `ints` drin.
- Das ist offensichtlich falsch und muss korrigiert werden.

Prüfung mit Mypy

```
$ mypy lists_error.py --no-strict-optional --check-untyped-defs
lists_error.py:1: error: List item 0 has incompatible type "int";
    ↪ expected "str"  [list-item]
lists_error.py:1: error: List item 1 has incompatible type "int";
    ↪ expected "str"  [list-item]
lists_error.py:1: error: List item 2 has incompatible type "int";
    ↪ expected "str"  [list-item]
Found 3 errors in 1 file (checked 1 source file)
# mypy 2.4.1 failed with exit code 1.
```

- Das Erste, was wir immer machen, ist unseren Kode mit Mypy[14] prüfen.
- Klar: Wir definieren `my_list` als Liste von Strings und speichern dann `ints` drin.
- Das ist offensichtlich falsch und muss korrigiert werden.
- Mypy sucht *nur* nach Datentyp-bezogenen Fehlern.

Prüfung mit Mypy

```
$ mypy lists_error.py --no-strict-optional --check-untyped-defs
lists_error.py:1: error: List item 0 has incompatible type "int";
    ↪ expected "str"  [list-item]
lists_error.py:1: error: List item 1 has incompatible type "int";
    ↪ expected "str"  [list-item]
lists_error.py:1: error: List item 2 has incompatible type "int";
    ↪ expected "str"  [list-item]
Found 3 errors in 1 file (checked 1 source file)
# mypy 2.4.1 failed with exit code 1.
```

- Das Erste, was wir immer machen, ist unseren Kode mit Mypy[14] prüfen.
- Klar: Wir definieren `my_list` als Liste von Strings und speichern dann `ints` drin.
- Das ist offensichtlich falsch und muss korrigiert werden.
- Mypy sucht *nur* nach Datentyp-bezogenen Fehlern.
- Aber haben wir *nur* Datentyp-bezogene Fehler gemacht?

Prüfung mit Mypy

```
$ mypy lists_error.py --no-strict-optional --check-untyped-defs
lists_error.py:1: error: List item 0 has incompatible type "int";
    ↪ expected "str"  [list-item]
lists_error.py:1: error: List item 1 has incompatible type "int";
    ↪ expected "str"  [list-item]
lists_error.py:1: error: List item 2 has incompatible type "int";
    ↪ expected "str"  [list-item]
Found 3 errors in 1 file (checked 1 source file)
# mypy 2.4.1 failed with exit code 1.
```

- Das Erste, was wir immer machen, ist unseren Kode mit Mypy[14] prüfen.
- Klar: Wir definieren `my_list` als Liste von Strings und speichern dann `ints` drin.
- Das ist offensichtlich falsch und muss korrigiert werden.
- Mypy sucht *nur* nach Datentyp-bezogenen Fehlern.
- Aber haben wir *nur* Datentyp-bezogene Fehler gemacht?
- Gibt es Werkzeuge, die auch andere Arten von Fehlern finden können?

Prüfung mit Mypy

```
$ mypy lists_error.py --no-strict-optional --check-untyped-defs
lists_error.py:1: error: List item 0 has incompatible type "int";
    ↪ expected "str"  [list-item]
lists_error.py:1: error: List item 1 has incompatible type "int";
    ↪ expected "str"  [list-item]
lists_error.py:1: error: List item 2 has incompatible type "int";
    ↪ expected "str"  [list-item]
Found 3 errors in 1 file (checked 1 source file)
# mypy 2.4.1 failed with exit code 1.
```

- Klar: Wir definieren `my_list` als Liste von Strings und speichern dann `int`s drin.
- Das ist offensichtlich falsch und muss korrigiert werden.
- Mypy sucht *nur* nach Datentyp-bezogenen Fehlern.
- Aber haben wir *nur* Datentyp-bezogene Fehler gemacht?
- Gibt es Werkzeuge, die auch andere Arten von Fehlern finden können?
- Ja: Linters sind Werkzeuge zum analysieren von Kode mit dem Ziel Bugs, Probleme, Schwachstellen, und inkonsistenten Stil zu finden[10,22].

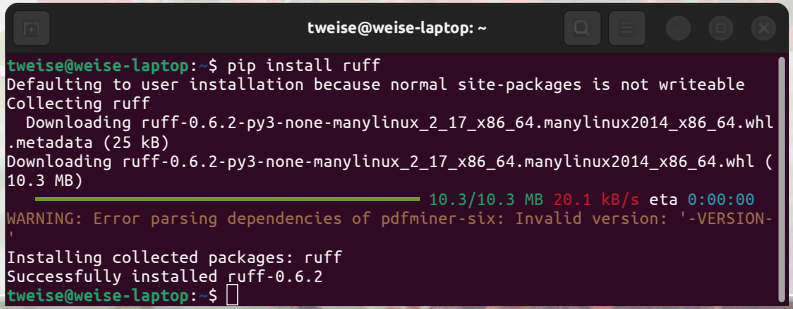

Ruff

Ruff Installieren

- Ruff ist ein sehr schneller Python Linter der Kode nach vielen verschiedenen Arten von Problem durchsuchen kann, angefangen von Formatierungsfehlern über fehlende Dokumentation und Performanz-Probleme bis hin zu potentiellen Programmierfehlern[17].

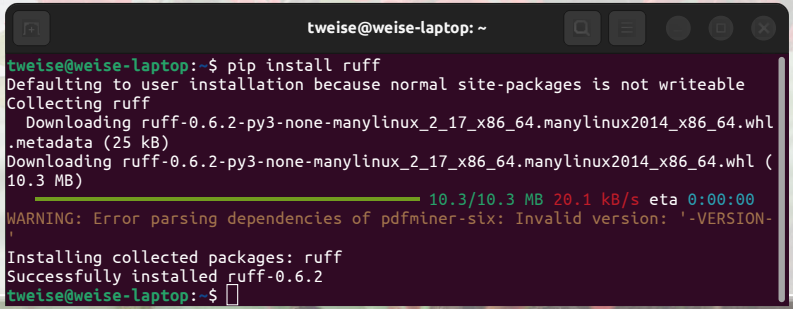
Ruff Installieren

- Ruff ist ein sehr schneller Python Linter der Kode nach vielen verschiedenen Arten von Problem durchsuchen kann, angefangen von Formatierungsfehlern über fehlende Dokumentation und Performanz-Probleme bis hin zu potentiellen Programmierfehlern[17].
- Um dieses Programm zu installieren, öffnen Sie ein Terminal, in dem Sie unter Ubuntu Ctrl + Alt + T drücken und unter Microsoft Windows mit Druck auf ⊞ + R, dann Schreiben von `cmd`, dann Druck auf ↵.



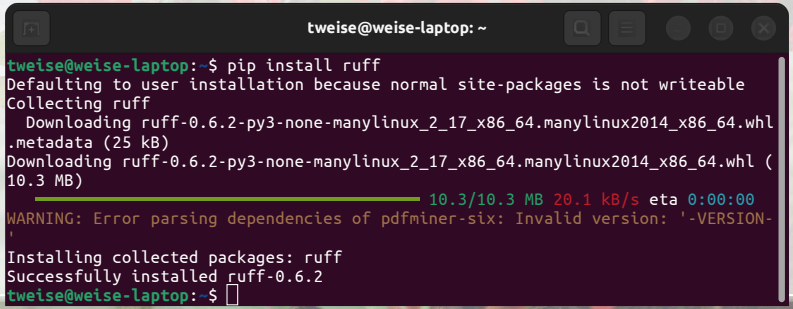
Ruff Installieren

- Um dieses Programm zu installieren, öffnen Sie ein Terminal, in dem Sie unter Ubuntu Ctrl + Alt + T drücken und unter Microsoft Windows mit Druck auf ⊞ + R, dann Schreiben von `cmd`, dann Druck auf ↵.
- Sie würden dann `pip install ruff` eintippen und ↵ drücken.



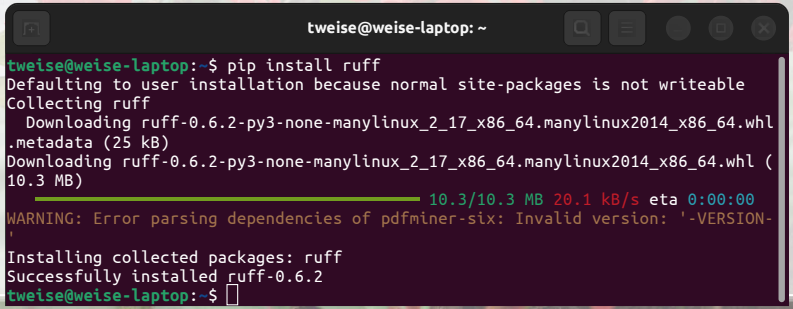
Ruff Installieren

- Um dieses Programm zu installieren, öffnen Sie ein Terminal, in dem Sie unter Ubuntu Ctrl + Alt + T drücken und unter Microsoft Windows mit Druck auf ⊞ + R, dann Schreiben von `cmd`, dann Druck auf ↵.
- Sie würden dann `pip install ruff` eintippen und ↵ drücken.
- Normalerweise machen Sie dass unter einem virtuellen Environment, was wir später diskutieren.



Ruff Installieren

- Um dieses Programm zu installieren, öffnen Sie ein Terminal, in dem Sie unter Ubuntu Ctrl + Alt + T drücken und unter Microsoft Windows mit Druck auf ⊞ + R, dann Schreiben von `cmd`, dann Druck auf ↵.
- Sie würden dann `pip install ruff` eintippen und ↵ drücken.
- Normalerweise machen Sie dass unter einem virtuellen Environment, was wir später diskutieren.
- So oder so, Ruff wird installiert.



Ruff Anwenden

- Wenden wir Ruff auf das Programm `lists_error.py` an.

Ruff Anwenden

- Wenden wir Ruff auf das Programm `lists_error.py` an.
- Das geht mit dem Kommando `ruff check myfile.py`, wobei `myfile.py` die zu prüfende Datei ist (man kann auch ein Verzeichnis angeben).

```
$ ruff check --target-version py312 --select=A,AIR,ANN,ASYNC,B,BLE,C,C4,
    ↪ COM,D,DJ,DTZ,E,ERA,EXE,F,FA,FIX,FLY,FURB,G,I,ICN,INP,ISC,INT,LOG,N
    ↪ ,NPY,PERF,PIE,PLC,PLE,PLR,PLW,PT,PYI,Q,RET,RSE,RUF,S,SIM,T,T10,TD,
    ↪ TID,TRY,UP,W,YTT --ignore=A005,ANN001,ANN002,ANN003,ANN204,ANN401,
    ↪ B008,B009,B010,C901,D203,D208,D212,D401,D407,D413,INP001,N801,
    ↪ PLC2801,PLR0904,PLR0911,PLR0912,PLR0913,PLR0914,PLR0915,PLR0916,
    ↪ PLR0917,PLR1702,PLR2004,PLR6301,PT011,PT012,PT013,PYI041,RUF100,S,
    ↪ T201,TRY003,UP035,W --line-length 79 lists_error.py
D100 Missing docstring in public module
--> lists_error.py:1:1
C410 Unnecessary list literal passed to `list()` (remove the outer call
    ↪ to `list()`)
 --> lists_error.py:1:22
1 | my_list: list[str] = list([1, 2, 3])
  |                      ^^^^^^^^^^^^^^
2 | print(my_list)
help: Remove outer `list()` call
  - my_list: list[str] = list([1, 2, 3])
1 + my_list: list[str] = [1, 2, 3]
2 | print(my_list)
note: This is an unsafe fix and may change runtime behavior
```

Ruff Anwenden

- Das geht mit dem Kommando `ruff check myfile.py`, wobei `myfile.py` die zu prüfende Datei ist (man kann auch ein Verzeichnis angeben).
- Ruff hat viele zusätzliche Parameter[16,17].

```
$ ruff check --target-version py312 --select=A,AIR,ANN,ASYNC,B,BLE,C,C4,
    ↪ COM,D,DJ,DTZ,E,ERA,EXE,F,FA,FIX,FLY,FURB,G,I,ICN,INP,ISC,INT,LOG,N
    ↪ ,NPY,PERF,PIE,PLC,PLE,PLR,PLW,PT,PYI,Q,RET,RSE,RUF,S,SIM,T,T10,TD,
    ↪ TID,TRY,UP,W,YTT --ignore=A005,ANN001,ANN002,ANN003,ANN204,ANN401,
    ↪ B008,B009,B010,C901,D203,D208,D212,D401,D407,D413,INP001,N801,
    ↪ PLC2801,PLR0904,PLR0911,PLR0912,PLR0913,PLR0914,PLR0915,PLR0916,
    ↪ PLR0917,PLR1702,PLR2004,PLR6301,PT011,PT012,PT013,PYI041,RUF100,S,
    ↪ T201,TRY003,UP035,W --line-length 79 lists_error.py
D100 Missing docstring in public module
--> lists_error.py:1:1
C410 Unnecessary list literal passed to `list()` (remove the outer call
    ↪ to `list()`)
 --> lists_error.py:1:22
1 | my_list: list[str] = list([1, 2, 3])
  |                      ^^^^^^^^^^^^^^
2 | print(my_list)
help: Remove outer `list()` call
  - my_list: list[str] = list([1, 2, 3])
1 + my_list: list[str] = [1, 2, 3]
2 | print(my_list)
note: This is an unsafe fix and may change runtime behavior
```

Ruff Anwenden

- Ruff hat viele zusätzliche Parameter[16,17].
- Wollen wir zum Beispiel unseren Kode für Python Version 3.12 analysieren, dann geben wir `--target-version py312` an.

```
$ ruff check --target-version py312 --select=A,AIR,ANN,ASYNC,B,BLE,C,C4,
   ↪ COM,D,DJ,DTZ,E,ERA,EXE,F,FA,FIX,FLY,FURB,G,I,ICN,INP,ISC,INT,LOG,N
   ↪ ,NPY,PERF,PIE,PLC,PLE,PLR,PLW,PT,PYI,Q,RET,RSE,RUF,S,SIM,T,T10,TD,
   ↪ TID,TRY,UP,W,YTT --ignore=A005,ANN001,ANN002,ANN003,ANN204,ANN401,
   ↪ B008,B009,B010,C901,D203,D208,D212,D401,D407,D413,INP001,N801,
   ↪ PLC2801,PLR0904,PLR0911,PLR0912,PLR0913,PLR0914,PLR0915,PLR0916,
   ↪ PLR0917,PLR1702,PLR2004,PLR6301,PT011,PT012,PT013,PYI041,RUF100,S,
   ↪ T201,TRY003,UP035,W --line-length 79 lists_error.py
D100 Missing docstring in public module
--> lists_error.py:1:1
C410 Unnecessary list literal passed to `list()` (remove the outer call
   ↪ to `list()`)
 --> lists_error.py:1:22
1 | my_list: list[str] = list([1, 2, 3])
  |                      ^^^^^^^^^^^^^^
2 | print(my_list)
help: Remove outer `list()` call
  - my_list: list[str] = list([1, 2, 3])
1 + my_list: list[str] = [1, 2, 3]
2 | print(my_list)
note: This is an unsafe fix and may change runtime behavior
```

Ruff Anwenden

- Mit dem optionalen Parameter `--select=...` können wir zusätzliche Regelsätze angeben, die Ruff zum überprüfen des Programms anwenden soll, siehe https://docs.astral.sh/ruff unter *Rules*.

```
$ ruff check --target-version py312 --select=A,AIR,ANN,ASYNC,B,BLE,C,C4,
   ↪ COM,D,DJ,DTZ,E,ERA,EXE,F,FA,FIX,FLY,FURB,G,I,ICN,INP,ISC,INT,LOG,N
   ↪ ,NPY,PERF,PIE,PLC,PLE,PLR,PLW,PT,PYI,Q,RET,RSE,RUF,S,SIM,T,T10,TD,
   ↪ TID,TRY,UP,W,YTT --ignore=A005,ANN001,ANN002,ANN003,ANN204,ANN401,
   ↪ B008,B009,B010,C901,D203,D208,D212,D401,D407,D413,INP001,N801,
   ↪ PLC2801,PLR0904,PLR0911,PLR0912,PLR0913,PLR0914,PLR0915,PLR0916,
   ↪ PLR0917,PLR1702,PLR2004,PLR6301,PT011,PT012,PT013,PYI041,RUF100,S,
   ↪ T201,TRY003,UP035,W --line-length 79 lists_error.py
D100 Missing docstring in public module
--> lists_error.py:1:1
C410 Unnecessary list literal passed to `list()` (remove the outer call
   ↪ to `list()`)
 --> lists_error.py:1:22
1 | my_list: list[str] = list([1, 2, 3])
  |                      ^^^^^^^^^^^^^^
2 | print(my_list)
help: Remove outer `list()` call
  - my_list: list[str] = list([1, 2, 3])
1 + my_list: list[str] = [1, 2, 3]
2 | print(my_list)
note: This is an unsafe fix and may change runtime behavior
```

Ruff Anwenden

- Manchmal wollen wir bestimmte Regeln ausschalten, was mit `--ignore=...` geht.

```
$ ruff check --target-version py312 --select=A,AIR,ANN,ASYNC,B,BLE,C,C4,
    ↪ COM,D,DJ,DTZ,E,ERA,EXE,F,FA,FIX,FLY,FURB,G,I,ICN,INP,ISC,INT,LOG,N
    ↪ ,NPY,PERF,PIE,PLC,PLE,PLR,PLW,PT,PYI,Q,RET,RSE,RUF,S,SIM,T,T10,TD,
    ↪ TID,TRY,UP,W,YTT --ignore=A005,ANN001,ANN002,ANN003,ANN204,ANN401,
    ↪ B008,B009,B010,C901,D203,D208,D212,D401,D407,D413,INP001,N801,
    ↪ PLC2801,PLR0904,PLR0911,PLR0912,PLR0913,PLR0914,PLR0915,PLR0916,
    ↪ PLR0917,PLR1702,PLR2004,PLR6301,PT011,PT012,PT013,PYI041,RUF100,S,
    ↪ T201,TRY003,UP035,W --line-length 79 lists_error.py
D100 Missing docstring in public module
--> lists_error.py:1:1
C410 Unnecessary list literal passed to `list()` (remove the outer call
    ↪ to `list()`)
 --> lists_error.py:1:22
1 | my_list: list[str] = list([1, 2, 3])
  |                      ^^^^^^^^^^^^^^
2 | print(my_list)
help: Remove outer `list()` call
  - my_list: list[str] = list([1, 2, 3])
1 + my_list: list[str] = [1, 2, 3]
2 | print(my_list)
note: This is an unsafe fix and may change runtime behavior
```

Ruff Anwenden

- Manchmal wollen wir bestimmte Regeln ausschalten, was mit `--ignore=...` geht.
- Das ganze Kommando steht da ... vielleicht wollen Sie sich ein Shell-Skript machen, damit Sie das nicht immer abtippen müssen.

```
$ ruff check --target-version py312 --select=A,AIR,ANN,ASYNC,B,BLE,C,C4,
    ↪ COM,D,DJ,DTZ,E,ERA,EXE,F,FA,FIX,FLY,FURB,G,I,ICN,INP,ISC,INT,LOG,N
    ↪ ,NPY,PERF,PIE,PLC,PLE,PLR,PLW,PT,PYI,Q,RET,RSE,RUF,S,SIM,T,T10,TD,
    ↪ TID,TRY,UP,W,YTT --ignore=A005,ANN001,ANN002,ANN003,ANN204,ANN401,
    ↪ B008,B009,B010,C901,D203,D208,D212,D401,D407,D413,INP001,N801,
    ↪ PLC2801,PLR0904,PLR0911,PLR0912,PLR0913,PLR0914,PLR0915,PLR0916,
    ↪ PLR0917,PLR1702,PLR2004,PLR6301,PT011,PT012,PT013,PYI041,RUF100,S,
    ↪ T201,TRY003,UP035,W --line-length 79 lists_error.py
D100 Missing docstring in public module
--> lists_error.py:1:1
C410 Unnecessary list literal passed to `list()` (remove the outer call
    ↪ to `list()`)
 --> lists_error.py:1:22
1 | my_list: list[str] = list([1, 2, 3])
  |                      ^^^^^^^^^^^^^^
2 | print(my_list)
help: Remove outer `list()` call
  - my_list: list[str] = list([1, 2, 3])
1 + my_list: list[str] = [1, 2, 3]
2 | print(my_list)
note: This is an unsafe fix and may change runtime behavior
```

Ruff Anwenden

- Das ganze Kommando steht da ... vielleicht wollen Sie sich ein Shell-Skript machen, damit Sie das nicht immer abtippen müssen.
- Ruff prüft aktuell keine Datentyp-bezogenen Fehler.

```
$ ruff check --target-version py312 --select=A,AIR,ANN,ASYNC,B,BLE,C,C4,
    ↪ COM,D,DJ,DTZ,E,ERA,EXE,F,FA,FIX,FLY,FURB,G,I,ICN,INP,ISC,INT,LOG,N
    ↪ ,NPY,PERF,PIE,PLC,PLE,PLR,PLW,PT,PYI,Q,RET,RSE,RUF,S,SIM,T,T10,TD,
    ↪ TID,TRY,UP,W,YTT --ignore=A005,ANN001,ANN002,ANN003,ANN204,ANN401,
    ↪ B008,B009,B010,C901,D203,D208,D212,D401,D407,D413,INP001,N801,
    ↪ PLC2801,PLR0904,PLR0911,PLR0912,PLR0913,PLR0914,PLR0915,PLR0916,
    ↪ PLR0917,PLR1702,PLR2004,PLR6301,PT011,PT012,PT013,PYI041,RUF100,S,
    ↪ T201,TRY003,UP035,W --line-length 79 lists_error.py
D100 Missing docstring in public module
--> lists_error.py:1:1
C410 Unnecessary list literal passed to `list()` (remove the outer call
    ↪ to `list()`)
 --> lists_error.py:1:22
1 | my_list: list[str] = list([1, 2, 3])
  |                      ^^^^^^^^^^^^^^
2 | print(my_list)
help: Remove outer `list()` call
  - my_list: list[str] = list([1, 2, 3])
1 + my_list: list[str] = [1, 2, 3]
2 | print(my_list)
note: This is an unsafe fix and may change runtime behavior
```

Ruff Anwenden

- Ruff prüft aktuell keine Datentyp-bezogenen Fehler.
- Aber es findet zwei andere Probleme.

```
$ ruff check --target-version py312 --select=A,AIR,ANN,ASYNC,B,BLE,C,C4,
   ↪ COM,D,DJ,DTZ,E,ERA,EXE,F,FA,FIX,FLY,FURB,G,I,ICN,INP,ISC,INT,LOG,N
   ↪ ,NPY,PERF,PIE,PLC,PLE,PLR,PLW,PT,PYI,Q,RET,RSE,RUF,S,SIM,T,T10,TD,
   ↪ TID,TRY,UP,W,YTT --ignore=A005,ANN001,ANN002,ANN003,ANN204,ANN401,
   ↪ B008,B009,B010,C901,D203,D208,D212,D401,D407,D413,INP001,N801,
   ↪ PLC2801,PLR0904,PLR0911,PLR0912,PLR0913,PLR0914,PLR0915,PLR0916,
   ↪ PLR0917,PLR1702,PLR2004,PLR6301,PT011,PT012,PT013,PYI041,RUF100,S,
   ↪ T201,TRY003,UP035,W --line-length 79 lists_error.py
D100 Missing docstring in public module
--> lists_error.py:1:1
C410 Unnecessary list literal passed to `list()` (remove the outer call
   ↪ to `list()`)
 --> lists_error.py:1:22
1 | my_list: list[str] = list([1, 2, 3])
  |                      ^^^^^^^^^^^^^^
2 | print(my_list)
help: Remove outer `list()` call
  - my_list: list[str] = list([1, 2, 3])
1 + my_list: list[str] = [1, 2, 3]
2 | print(my_list)
note: This is an unsafe fix and may change runtime behavior
```

Fehlende Dokumentation

- Der erste Fehler ist, dass die Datei nicht mit einer Zeichenkette anfängt, die den Sinn und Zweck der Datei erklärt.

```
$ ruff check --target-version py312 --select=A,AIR,ANN,ASYNC,B,BLE,C,C4,
    ↪ COM,D,DJ,DTZ,E,ERA,EXE,F,FA,FIX,FLY,FURB,G,I,ICN,INP,ISC,INT,LOG,N
    ↪ ,NPY,PERF,PIE,PLC,PLE,PLR,PLW,PT,PYI,Q,RET,RSE,RUF,S,SIM,T,T10,TD,
    ↪ TID,TRY,UP,W,YTT --ignore=A005,ANN001,ANN002,ANN003,ANN204,ANN401,
    ↪ B008,B009,B010,C901,D203,D208,D212,D401,D407,D413,INP001,N801,
    ↪ PLC2801,PLR0904,PLR0911,PLR0912,PLR0913,PLR0914,PLR0915,PLR0916,
    ↪ PLR0917,PLR1702,PLR2004,PLR6301,PT011,PT012,PT013,PYI041,RUF100,S,
    ↪ T201,TRY003,UP035,W --line-length 79 lists_error.py
D100 Missing docstring in public module
--> lists_error.py:1:1
C410 Unnecessary list literal passed to `list()` (remove the outer call
    ↪ to `list()`)
 --> lists_error.py:1:22
1 | my_list: list[str] = list([1, 2, 3])
  |                      ^^^^^^^^^^^^^^
2 | print(my_list)
help: Remove outer `list()` call
  - my_list: list[str] = list([1, 2, 3])
1 + my_list: list[str] = [1, 2, 3]
2 | print(my_list)
note: This is an unsafe fix and may change runtime behavior
```

Fehlende Dokumentation



- Der erste Fehler ist, dass die Datei nicht mit einer Zeichenkette anfängt, die den Sinn und Zweck der Datei erklärt.
- Solche so genannten Docstrings machen es für andere Programmierer leichter zu verstehen, was welche Datei eines Projekts macht.

Fehlende Dokumentation



- Der erste Fehler ist, dass die Datei nicht mit einer Zeichenkette anfängt, die den Sinn und Zweck der Datei erklärt.
- Solche so genannten Docstrings machen es für andere Programmierer leichter zu verstehen, was welche Datei eines Projekts macht.

Gute Praxis

Jede Python-Datei muss mit einem String anfangen, der ihren Zweck erklärt[6].

Fehlende Dokumentation



- Der erste Fehler ist, dass die Datei nicht mit einer Zeichenkette anfängt, die den Sinn und Zweck der Datei erklärt.
- Solche so genannten Docstrings machen es für andere Programmierer leichter zu verstehen, was welche Datei eines Projekts macht.

Gute Praxis

Jede Python-Datei muss mit einem String anfangen, der ihren Zweck erklärt[6]. Das kann eine einzele Zeile sein, wie eine Überschrift, oder auch ein längerer Text.

Fehlende Dokumentation



- Der erste Fehler ist, dass die Datei nicht mit einer Zeichenkette anfängt, die den Sinn und Zweck der Datei erklärt.
- Solche so genannten Docstrings machen es für andere Programmierer leichter zu verstehen, was welche Datei eines Projekts macht.

Gute Praxis

Jede Python-Datei muss mit einem String anfangen, der ihren Zweck erklärt[6]. Das kann eine einzele Zeile sein, wie eine Überschrift, oder auch ein längerer Text. Im zweiten Fall muss die erste Zeile eine Überschrift gefolgt von einer Leerzeile sein, auf die dann der restliche Text folgt.

Fehlende Dokumentation



- Der erste Fehler ist, dass die Datei nicht mit einer Zeichenkette anfängt, die den Sinn und Zweck der Datei erklärt.
- Solche so genannten Docstrings machen es für andere Programmierer leichter zu verstehen, was welche Datei eines Projekts macht.

Gute Praxis

Jede Python-Datei muss mit einem String anfangen, der ihren Zweck erklärt[6]. Das kann eine einzele Zeile sein, wie eine Überschrift, oder auch ein längerer Text. Im zweiten Fall muss die erste Zeile eine Überschrift gefolgt von einer Leerzeile sein, auf die dann der restliche Text folgt. So oder so, ein Docstring muss mit `"""..."""` begrenzt sein[6,29].

Performanzproblem

- Der zweite Fehler ist ein Performanz-Problem.

```
$ ruff check --target-version py312 --select=A,AIR,ANN,ASYNC,B,BLE,C,C4,
    ↪ COM,D,DJ,DTZ,E,ERA,EXE,F,FA,FIX,FLY,FURB,G,I,ICN,INP,ISC,INT,LOG,N
    ↪ ,NPY,PERF,PIE,PLC,PLE,PLR,PLW,PT,PYI,Q,RET,RSE,RUF,S,SIM,T,T10,TD,
    ↪ TID,TRY,UP,W,YTT --ignore=A005,ANN001,ANN002,ANN003,ANN204,ANN401,
    ↪ B008,B009,B010,C901,D203,D208,D212,D401,D407,D413,INP001,N801,
    ↪ PLC2801,PLR0904,PLR0911,PLR0912,PLR0913,PLR0914,PLR0915,PLR0916,
    ↪ PLR0917,PLR1702,PLR2004,PLR6301,PT011,PT012,PT013,PYI041,RUF100,S,
    ↪ T201,TRY003,UP035,W --line-length 79 lists_error.py
D100 Missing docstring in public module
--> lists_error.py:1:1
C410 Unnecessary list literal passed to `list()` (remove the outer call
    ↪ to `list()`)
 --> lists_error.py:1:22
1 | my_list: list[str] = list([1, 2, 3])
  |                      ^^^^^^^^^^^^^^
2 | print(my_list)
help: Remove outer `list()` call
  - my_list: list[str] = list([1, 2, 3])
1 + my_list: list[str] = [1, 2, 3]
2 | print(my_list)
note: This is an unsafe fix and may change runtime behavior
```

Performanzproblem



- Der zweite Fehler ist ein Performanz-Problem.
- `list([1, 2, 3])` zu schreiben ist eine sinnlose Verschwendung von Speicher und Zeit.

Performanzproblem



- Der zweite Fehler ist ein Performanz-Problem.
- `list([1, 2, 3])` zu schreiben ist eine sinnlose Verschwendung von Speicher und Zeit.
- Es erstellt zuerst ein Listen-Literal via `[1, 2, 3]`.

Performanzproblem



- Der zweite Fehler ist ein Performanz-Problem.
- `list([1, 2, 3])` zu schreiben ist eine sinnlose Verschwendung von Speicher und Zeit.
- Es erstellt zuerst ein Listen-Literal via `[1, 2, 3]`.
- Dann macht es sofort eine Kopie dieser Liste mit der `list(...)`-Funktion.

Performanzproblem



- Der zweite Fehler ist ein Performanz-Problem.
- `list([1, 2, 3])` zu schreiben ist eine sinnlose Verschwendung von Speicher und Zeit.
- Es erstellt zuerst ein Listen-Literal via `[1, 2, 3]`.
- Dann macht es sofort eine Kopie dieser Liste mit der `list(...)`-Funktion.
- Wir könnten also das `list(...)` weglassen.

Probleme Gelöst



- Lösen wir also alle drei entdeckten Probleme und erstellen ein Programm `lists_fixed.py`.

Probleme Gelöst

- Lösen wir also alle drei entdeckten Probleme und erstellen ein Programm `lists_fixed.py`:

```
"""
A fixed version of the original, erroneous program.

The original program was only two lines, namely:

> my_list: list[str] = list([1, 2, 3])
> print(my_list)

There were three errors:

1. mypy will detect that we store integers in a list of str.
2. ruff finds the missing docstring at the program head.
3. ruff finds that writing [1, 2, 3] is better than list([1, 2, 3]).

We now fix it here.
"""
my_list: list[int] = [1, 2, 3]
print(my_list)
```

↓ `python3 lists_fixed.py` ↓

```
[1, 2, 3]
```

Probleme Gelöst



- Lösen wir also alle drei entdeckten Probleme und erstellen ein Programm `lists_fixed.py`.
- Mypy läuft nun problemlos durch.

```
$ mypy lists_fixed.py --no-strict-optional --check-untyped-defs
Success: no issues found in 1 source file
# mypy 2.4.1 succeeded with exit code 0.
```

Probleme Gelöst

- Lösen wir also alle drei entdeckten Probleme und erstellen ein Programm `lists_fixed.py`.
- Mypy läuft nun problemlos durch.
- Ruff auch.

```
$ ruff check --target-version py312 --select=A,AIR,ANN,ASYNC,B,BLE,C,C4,
    ↪ COM,D,DJ,DTZ,E,ERA,EXE,F,FA,FIX,FLY,FURB,G,I,ICN,INP,ISC,INT,LOG,N
    ↪ ,NPY,PERF,PIE,PLC,PLE,PLR,PLW,PT,PYI,Q,RET,RSE,RUF,S,SIM,T,T10,TD,
    ↪ TID,TRY,UP,W,YTT --ignore=A005,ANN001,ANN002,ANN003,ANN204,ANN401,
    ↪ B008,B009,B010,C901,D203,D208,D212,D401,D407,D413,INP001,N801,
    ↪ PLC2801,PLR0904,PLR0911,PLR0912,PLR0913,PLR0914,PLR0915,PLR0916,
    ↪ PLR0917,PLR1702,PLR2004,PLR6301,PT011,PT012,PT013,PYI041,RUF100,S,
    ↪ T201,TRY003,UP035,W --line-length 79 lists_fixed.py
All checks passed!
# ruff 0.17.4 succeeded with exit code 0.
```

Was wir daraus lernen

- OK, das war ein Programm aus *zwei* Zeilen.



Was wir daraus lernen

- OK, das war ein Programm aus *zwei* Zeilen.
- Fragen Sie sich selbst mal...



Was wir daraus lernen



- OK, das war ein Programm aus *zwei* Zeilen.
- Fragen Sie sich selbst mal
  - Haben Sie den Fehler mit dem Type Hint gesehen, als Sie das Programm zum ersten Mal gelesen haben?

Was wir daraus lernen



- OK, das war ein Programm aus *zwei* Zeilen.
- Fragen Sie sich selbst mal
  - Haben Sie den Fehler mit dem Type Hint gesehen, als Sie das Programm zum ersten Mal gelesen haben? Naja, vielleicht...



Was wir daraus lernen



- OK, das war ein Programm aus *zwei* Zeilen.
- Fragen Sie sich selbst mal
  - Haben Sie den Fehler mit dem Type Hint gesehen, als Sie das Programm zum ersten Mal gelesen haben? Naja, vielleicht...
  - Aber ist Ihnen auch aufgefallen, dass wir eine Liste erstellt und dann sofort sinnlos kopiert haben, anstatt sie direkt zu verwenden?

Was wir daraus lernen



- OK, das war ein Programm aus *zwei* Zeilen.
- Fragen Sie sich selbst mal
  - Haben Sie den Fehler mit dem Type Hint gesehen, als Sie das Programm zum ersten Mal gelesen haben? Naja, vielleicht...
  - Aber ist Ihnen auch aufgefallen, dass wir eine Liste erstellt und dann sofort sinnlos kopiert haben, anstatt sie direkt zu verwenden?
  - Das mit dem Docstring sei Ihnen geschenkt, da haben wir ja noch nicht drüber geredet.

Was wir daraus lernen



- OK, das war ein Programm aus *zwei* Zeilen.
- Fragen Sie sich selbst mal
  - Haben Sie den Fehler mit dem Type Hint gesehen, als Sie das Programm zum ersten Mal gelesen haben? Naja, vielleicht...
  - Aber ist Ihnen auch aufgefallen, dass wir eine Liste erstellt und dann sofort sinnlos kopiert haben, anstatt sie direkt zu verwenden?
  - Das mit dem Docstring sei Ihnen geschenkt, da haben wir ja noch nicht drüber geredet.
- Jetzt stellen Sie sich mal vor, dass wären nicht 2, sondern 1000 Zeilen gewesen.

Was wir daraus lernen



- OK, das war ein Programm aus *zwei* Zeilen.
- Fragen Sie sich selbst mal
  - Haben Sie den Fehler mit dem Type Hint gesehen, als Sie das Programm zum ersten Mal gelesen haben? Naja, vielleicht...
  - Aber ist Ihnen auch aufgefallen, dass wir eine Liste erstellt und dann sofort sinnlos kopiert haben, anstatt sie direkt zu verwenden?
  - Das mit dem Docstring sei Ihnen geschenkt, da haben wir ja noch nicht drüber geredet.
- Jetzt stellen Sie sich mal vor, dass wären nicht 2, sondern 1000 Zeilen gewesen.
- Und bedenken Sie, dass das Programm trotz aller Fehler ja problemlos ausgeführt werden konnte ... die Fehler wären halt irgendwann später mal zu Problemen geworden...

Was wir daraus lernen



- OK, das war ein Programm aus *zwei* Zeilen.
- Fragen Sie sich selbst mal
  - Haben Sie den Fehler mit dem Type Hint gesehen, als Sie das Programm zum ersten Mal gelesen haben? Naja, vielleicht...
  - Aber ist Ihnen auch aufgefallen, dass wir eine Liste erstellt und dann sofort sinnlos kopiert haben, anstatt sie direkt zu verwenden?
  - Das mit dem Docstring sei Ihnen geschenkt, da haben wir ja noch nicht drüber geredet.
- Jetzt stellen Sie sich mal vor, dass wären nicht 2, sondern 1000 Zeilen gewesen.
- Und bedenken Sie, dass das Programm trotz aller Fehler ja problemlos ausgeführt werden konnte ... die Fehler wären halt irgendwann später mal zu Problemen geworden...
- Wäre es nicht toll, wenn der Kollege das 1000-Zeilen-Programm gründlich dokumentiert, type-hinted, und geprüft hätte?

Was wir daraus lernen



- OK, das war ein Programm aus *zwei* Zeilen.
- Fragen Sie sich selbst mal
  - Haben Sie den Fehler mit dem Type Hint gesehen, als Sie das Programm zum ersten Mal gelesen haben? Naja, vielleicht...
  - Aber ist Ihnen auch aufgefallen, dass wir eine Liste erstellt und dann sofort sinnlos kopiert haben, anstatt sie direkt zu verwenden?
  - Das mit dem Docstring sei Ihnen geschenkt, da haben wir ja noch nicht drüber geredet.
- Jetzt stellen Sie sich mal vor, dass wären nicht 2, sondern 1000 Zeilen gewesen.
- Und bedenken Sie, dass das Programm trotz aller Fehler ja problemlos ausgeführt werden konnte ... die Fehler wären halt irgendwann später mal zu Problemen geworden...
- Wäre es nicht toll, wenn der Kollege das 1000-Zeilen-Programm gründlich dokumentiert, type-hinted, und geprüft hätte?
- Werden Sie so ein Kollege!

Was wir daraus lernen



- Fragen Sie sich selbst mal
  - Haben Sie den Fehler mit dem Type Hint gesehen, als Sie das Programm zum ersten Mal gelesen haben? Naja, vielleicht...
  - Aber ist Ihnen auch aufgefallen, dass wir eine Liste erstellt und dann sofort sinnlos kopiert haben, anstatt sie direkt zu verwenden?
- Jetzt stellen Sie sich mal vor, dass wären nicht 2, sondern 1000 Zeilen gewesen.
- Und bedenken Sie, dass das Programm trotz aller Fehler ja problemlos ausgeführt werden konnte ... die Fehler wären halt irgendwann später mal zu Problemen geworden...

Gute Praxis

Benutzen Sie immer und viele statische Kode-Analyse Werkzeuge.

Was wir daraus lernen



- Fragen Sie sich selbst mal
  - Haben Sie den Fehler mit dem Type Hint gesehen, als Sie das Programm zum ersten Mal gelesen haben? Naja, vielleicht...
  - Aber ist Ihnen auch aufgefallen, dass wir eine Liste erstellt und dann sofort sinnlos kopiert haben, anstatt sie direkt zu verwenden?
- Jetzt stellen Sie sich mal vor, dass wären nicht 2, sondern 1000 Zeilen gewesen.
- Und bedenken Sie, dass das Programm trotz aller Fehler ja problemlos ausgeführt werden konnte ... die Fehler wären halt irgendwann später mal zu Problemen geworden...

Gute Praxis

Benutzen Sie immer und viele statische Kode-Analyse Werkzeuge. Sie können helfen, viele verschiedene Arten von Problem oder Verbesserungsmöglichkeiten zu finden.

Was wir daraus lernen



- Fragen Sie sich selbst mal
  - Haben Sie den Fehler mit dem Type Hint gesehen, als Sie das Programm zum ersten Mal gelesen haben? Naja, vielleicht...
  - Aber ist Ihnen auch aufgefallen, dass wir eine Liste erstellt und dann sofort sinnlos kopiert haben, anstatt sie direkt zu verwenden?
- Jetzt stellen Sie sich mal vor, dass wären nicht 2, sondern 1000 Zeilen gewesen.
- Und bedenken Sie, dass das Programm trotz aller Fehler ja problemlos ausgeführt werden konnte ... die Fehler wären halt irgendwann später mal zu Problemen geworden...

Gute Praxis

Benutzen Sie immer und viele statische Kode-Analyse Werkzeuge. Sie können helfen, viele verschiedene Arten von Problem oder Verbesserungsmöglichkeiten zu finden. Sie können helfen, Ihren Kode sauber zu halten und erzwingen einen konsistenten Programmierstil.

Was wir daraus lernen



- Fragen Sie sich selbst mal
  - Haben Sie den Fehler mit dem Type Hint gesehen, als Sie das Programm zum ersten Mal gelesen haben? Naja, vielleicht...
  - Aber ist Ihnen auch aufgefallen, dass wir eine Liste erstellt und dann sofort sinnlos kopiert haben, anstatt sie direkt zu verwenden?
- Jetzt stellen Sie sich mal vor, dass wären nicht 2, sondern 1000 Zeilen gewesen.
- Und bedenken Sie, dass das Programm trotz aller Fehler ja problemlos ausgeführt werden konnte ... die Fehler wären halt irgendwann später mal zu Problemen geworden...

Gute Praxis

Benutzen Sie immer und viele statische Kode-Analyse Werkzeuge. Sie können helfen, viele verschiedene Arten von Problem oder Verbesserungsmöglichkeiten zu finden. Sie können helfen, Ihren Kode sauber zu halten und erzwingen einen konsistenten Programmierstil. Und benutzen Sie sie nicht nur – implementieren Sie alle gefunden Hinweise!

Zusammenfassung



Zusammenfassung

- Mit Ruff und Mypy kennen wir nun zwei Werkzeuge zur statischen Kode-Analyse.

Zusammenfassung

- Mit Ruff und Mypy kennen wir nun zwei Werkzeuge zur statischen Kode-Analyse.
- Mypy sucht nach Datentyp-Problemen. Wenn wir einen `int` in einer `str`-Variable speichern wollen, dann wird Mypy meckern.

Zusammenfassung

- Mit Ruff und Mypy kennen wir nun zwei Werkzeuge zur statischen Kode-Analyse.
- Mypy sucht nach Datentyp-Problemen. Wenn wir einen `int` in einer `str`-Variable speichern wollen, dann wird Mypy meckern.
- Ruff ist ein so genannter Linter und sucht nach allen möglichen (anderen) Problem. Es findet fehlende Dokumentation, Performanz-Probleme, und verschiedene Programmierfehler.

Zusammenfassung

- Mit Ruff und Mypy kennen wir nun zwei Werkzeuge zur statischen Kode-Analyse.
- Mypy sucht nach Datentyp-Problemen. Wenn wir einen `int` in einer `str`-Variable speichern wollen, dann wird Mypy meckern.
- Ruff ist ein so genannter Linter und sucht nach allen möglichen (anderen) Problem. Es findet fehlende Dokumentation, Performanz-Probleme, und verschiedene Programmierfehler.
- Es gibt noch viel mehr solche Werkzeuge.

Zusammenfassung

- Mit Ruff und Mypy kennen wir nun zwei Werkzeuge zur statischen Kode-Analyse.
- Mypy sucht nach Datentyp-Problemen. Wenn wir einen `int` in einer `str`-Variable speichern wollen, dann wird Mypy meckern.
- Ruff ist ein so genannter Linter und sucht nach allen möglichen (anderen) Problem. Es findet fehlende Dokumentation, Performanz-Probleme, und verschiedene Programmierfehler.
- Es gibt noch viel mehr solche Werkzeuge.
- Je mehr wir benutzen, desto größer unsere Chance, Fehler in unserem Kode zu entdecken.

Zusammenfassung

- Mit Ruff und Mypy kennen wir nun zwei Werkzeuge zur statischen Kode-Analyse.
- Mypy sucht nach Datentyp-Problemen. Wenn wir einen `int` in einer `str`-Variable speichern wollen, dann wird Mypy meckern.
- Ruff ist ein so genannter Linter und sucht nach allen möglichen (anderen) Problem. Es findet fehlende Dokumentation, Performanz-Probleme, und verschiedene Programmierfehler.
- Es gibt noch viel mehr solche Werkzeuge.
- Je mehr wir benutzen, desto größer unsere Chance, Fehler in unserem Kode zu entdecken.
- Ein interessanter Aspekt ist, dass wir von den Hinweisen dieser Werkzeuge auch lernen können, bessere Programmierer zu werden.

Zusammenfassung



- Mit Ruff und Mypy kennen wir nun zwei Werkzeuge zur statischen Kode-Analyse.
- Mypy sucht nach Datentyp-Problemen. Wenn wir einen `int` in einer `str`-Variable speichern wollen, dann wird Mypy meckern.
- Ruff ist ein so genannter Linter und sucht nach allen möglichen (anderen) Problem. Es findet fehlende Dokumentation, Performanz-Probleme, und verschiedene Programmierfehler.
- Es gibt noch viel mehr solche Werkzeuge.
- Je mehr wir benutzen, desto größer unsere Chance, Fehler in unserem Kode zu entdecken.
- Ein interessanter Aspekt ist, dass wir von den Hinweisen dieser Werkzeuge auch lernen können, bessere Programmierer zu werden.
- Denn natürlich ist es immer besser, gut zu programmieren und von Anfang an wenige Fehler zu machen.

Zusammenfassung

- Mit Ruff und Mypy kennen wir nun zwei Werkzeuge zur statischen Kode-Analyse.
- Mypy sucht nach Datentyp-Problemen. Wenn wir einen `int` in einer `str`-Variable speichern wollen, dann wird Mypy meckern.
- Ruff ist ein so genannter Linter und sucht nach allen möglichen (anderen) Problem. Es findet fehlende Dokumentation, Performanz-Probleme, und verschiedene Programmierfehler.
- Es gibt noch viel mehr solche Werkzeuge.
- Je mehr wir benutzen, desto größer unsere Chance, Fehler in unserem Kode zu entdecken.
- Ein interessanter Aspekt ist, dass wir von den Hinweisen dieser Werkzeuge auch lernen können, bessere Programmierer zu werden.
- Denn natürlich ist es immer besser, gut zu programmieren und von Anfang an wenige Fehler zu machen.
- Aber da selbst der beste Programmier Fehler macht, sollte man niemals zu arrogant sein, solche Werkzeuge zu verwenden.

谢谢您们！

Thank you!

Vielen Dank!



References I

[1] Daniel J. Barrett. *Efficient Linux at the Command Line*. Sebastopol, CA, USA: O'Reilly Media, Inc., Feb. 2022. ISBN: 978-1-0981-1340-7 (siehe S. 77, 78).

[2] Ed Bott. *Windows 11 Inside Out*. Hoboken, NJ, USA: Microsoft Press, Pearson Education, Inc., Feb. 2023. ISBN: 978-0-13-769132-6 (siehe S. 78).

[3] Ron Brash und Ganesh Naik. *Bash Cookbook*. Birmingham, England, UK: Packt Publishing Ltd, Juli 2018. ISBN: 978-1-78862-936-2 (siehe S. 77).

[4] David Clinton und Christopher Negus. *Ubuntu Linux Bible*. 10. Aufl. Bible Series. Chichester, West Sussex, England, UK: John Wiley and Sons Ltd., 10. Nov. 2020. ISBN: 978-1-119-72233-5 (siehe S. 78).

[5] Slobodan Dmitrović. *Modern C for Absolute Beginners: A Friendly Introduction to the C Programming Language*. New York, NY, USA: Apress Media, LLC, März 2024. ISBN: 979-8-8688-0224-9 (siehe S. 77).

[6] David Goodger und Guido van Rossum. *Docstring Conventions*. Python Enhancement Proposal (PEP) 257. Beaverton, OR, USA: Python Software Foundation (PSF), 29. Mai–13. Juni 2001. URL: https://peps.python.org/pep-0257 (besucht am 2024-07-27) (siehe S. 34–39, 77).

[7] Michael Hausenblas. *Learning Modern Linux*. Sebastopol, CA, USA: O'Reilly Media, Inc., Apr. 2022. ISBN: 978-1-0981-0894-6 (siehe S. 77).

[8] Matthew Helmke. *Ubuntu Linux Unleashed 2021 Edition*. 14. Aufl. Reading, MA, USA: Addison-Wesley Professional, Aug. 2020. ISBN: 978-0-13-668539-5 (siehe S. 78).

[9] John Hunt. *A Beginners Guide to Python 3 Programming*. 2. Aufl. Undergraduate Topics in Computer Science (UTICS). Cham, Switzerland: Springer, 2023. ISSN: 1863-7310. ISBN: 978-3-031-35121-1. doi:10.1007/978-3-031-35122-8 (siehe S. 78).

[10] Stephen Curtis Johnson. *Lint, a C Program Checker*. Computing Science Technical Report 78-1273. New York, NY, USA: Bell Telephone Laboratories, Incorporated, 25. Okt. 1978. URL: https://wolfram.schneider.org/bsd/7thEdManVol2/lint/lint.pdf (besucht am 2024-08-23) (siehe S. 18, 77).

References II

[11] Łukasz Langa. *Literature Overview for Type Hints*. Python Enhancement Proposal (PEP) 482. Beaverton, OR, USA: Python Software Foundation (PSF), 8. Jan. 2015. URL: https://peps.python.org/pep-0482 (besucht am 2024-10-09) (siehe S. 78).

[12] Kent D. Lee und Steve Hubbard. *Data Structures and Algorithms with Python*. Undergraduate Topics in Computer Science (UTICS). Cham, Switzerland: Springer, 2015. ISSN: 1863-7310. ISBN: 978-3-319-13071-2. doi:10.1007/978-3-319-13072-9 (siehe S. 78).

[13] Michael Lee, Ivan Levkivskyi und Jukka Lehtosalo. *Literal Types*. Python Enhancement Proposal (PEP) 586. Beaverton, OR, USA: Python Software Foundation (PSF), 14. März 2019. URL: https://peps.python.org/pep-0586 (besucht am 2024-12-17) (siehe S. 77).

[14] Jukka Lehtosalo, Ivan Levkivskyi, Jared Hance, Ethan Smith, Guido van Rossum, Jelle „JelleZijlstra" Zijlstra, Michael J. Sullivan, Shantanu Jain, Xuanda Yang, Jingchen Ye, Nikita Sobolev und Mypy Contributors. *Mypy – Static Typing for Python*. San Francisco, CA, USA: GitHub Inc, 2024. URL: https://github.com/python/mypy (besucht am 2024-08-17) (siehe S. 12–17, 78).

[15] Mark Lutz. *Learning Python*. 6. Aufl. Sebastopol, CA, USA: O'Reilly Media, Inc., März 2025. ISBN: 978-1-0981-7130-8 (siehe S. 78).

[16] Charlie Marsh. "Ruff". In: URL: https://pypi.org/project/ruff (besucht am 2025-08-29) (siehe S. 25–28, 78).

[17] Charlie Marsh. *ruff: An Extremely Fast Python Linter and Code Formatter, Written in Rust*. New York, NY, USA: Astral Software Inc., 28. Aug. 2022. URL: https://docs.astral.sh/ruff (besucht am 2024-08-23) (siehe S. 20, 21, 25–29, 78).

[18] Carl Meyer. *Python Virtual Environments*. Python Enhancement Proposal (PEP) 405. Beaverton, OR, USA: Python Software Foundation (PSF), 13. Juni 2011–24. Mai 2012. URL: https://peps.python.org/pep-0405 (besucht am 2024-12-25) (siehe S. 79).

[19] Cameron Newham und Bill Rosenblatt. *Learning the Bash Shell – Unix Shell Programming: Covers Bash 3.0*. 3. Aufl. Sebastopol, CA, USA: O'Reilly Media, Inc., 2005. ISBN: 978-0-596-00965-6 (siehe S. 77).

[20] Yasset Pérez-Riverol, Laurent Gatto, Rui Wang, Timo Sachsenberg, Julian Uszkoreit, Felipe da Veiga Leprevost, Christian Fufezan, Tobias Ternent, Stephen J. Eglen, Daniel S. Katz, Tom J. Pollard, Alexander Konovalov, Robert M. Flight, Kai Blin und Juan Antonio Vizcaíno. "Ten Simple Rules for Taking Advantage of Git and GitHub". *PLOS Computational Biology* 12(7), 14. Juli 2016. San Francisco, CA, USA: Public Library of Science (PLOS). ISSN: 1553-7358. doi:10.1371/JOURNAL.PCBI.1004947 (siehe S. 77).

References III

[21] *Programming Languages – C, Working Document of SC22/WG14*. International Standard ISO/IEC9899:2017 C17 Ballot N2176. Geneva, Switzerland: International Organization for Standardization (ISO) und International Electrotechnical Commission (IEC), Nov. 2017. URL: https://files.lhmouse.com/standards/ISO%20C%20N2176.pdf (besucht am 2024-06-29) (siehe S. 77).

[22] Yeonhee Ryou, Sangwoo Joh, Joonmo Yang, Sujin Kim und Youil Kim. "Code Understanding Linter to Detect Variable Misuse". In: *37th IEEE/ACM International Conference on Automated Software Engineering (ASE'2022)*. 10.–14. Okt. 2022, Rochester, MI, USA. New York, NY, USA: Association for Computing Machinery (ACM), 2022, 133:1–133:5. ISBN: 978-1-4503-9475-8. doi:10.1145/3551349.3559497 (siehe S. 18, 77).

[23] Ellen Siever, Stephen Figgins, Robert Love und Arnold Robbins. *Linux in a Nutshell*. 6. Aufl. Sebastopol, CA, USA: O'Reilly Media, Inc., Sep. 2009. ISBN: 978-0-596-15448-6 (siehe S. 77).

[24] Anna Skoulikari. *Learning Git*. Sebastopol, CA, USA: O'Reilly Media, Inc., Mai 2023. ISBN: 978-1-0981-3391-7 (siehe S. 77).

[25] ."Literals". In: *Static Typing with Python*. Hrsg. von The Python Typing Team. Beaverton, OR, USA: Python Software Foundation (PSF), 2021. URL: https://typing.python.org/en/latest/spec/literal.html (besucht am 2025-08-29) (siehe S. 77).

[26] Linus Torvalds. "The Linux Edge". *Communications of the ACM (CACM)* 42(4):38–39, Apr. 1999. New York, NY, USA: Association for Computing Machinery (ACM). ISSN: 0001-0782. doi:10.1145/299157.299165 (siehe S. 77).

[27] Mariot Tsitoara. *Beginning Git and GitHub: Version Control, Project Management and Teamwork for the New Developer*. New York, NY, USA: Apress Media, LLC, März 2024. ISBN: 979-8-8688-0215-7 (siehe S. 77, 79).

[28] Guido van Rossum und Łukasz Langa. *Type Hints*. Python Enhancement Proposal (PEP) 484. Beaverton, OR, USA: Python Software Foundation (PSF), 29. Sep. 2014. URL: https://peps.python.org/pep-0484 (besucht am 2024-08-22) (siehe S. 78).

[29] Guido van Rossum, Barry Warsaw und Alyssa Coghlan. *Style Guide for Python Code*. Python Enhancement Proposal (PEP) 8. Beaverton, OR, USA: Python Software Foundation (PSF), 5. Juli 2001. URL: https://peps.python.org/pep-0008 (besucht am 2024-07-27) (siehe S. 34–39, 77).

References IV

[30] Sander van Vugt. *Linux Fundamentals*. 2. Aufl. Hoboken, NJ, USA: Pearson IT Certification, Juni 2022. ISBN: 978-0-13-792931-3 (siehe S. 77).

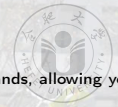
[31] "Virtual Environments and Packages". In: *Python 3 Documentation*. *The Python Tutorial*. Beaverton, OR, USA: Python Software Foundation (PSF), 2001–2025. Kap. 12. URL: https://docs.python.org/3/tutorial/venv.html (besucht am 2024-12-24) (siehe S. 79).

[32] Thomas Weise (汤卫思). *Programming with C*. Hefei, Anhui, China (中国安徽省合肥市): Hefei University (合肥大学), School of Artificial Intelligence and Big Data (人工智能与大数据学院), 2026. URL: https://thomasweise.github.io/programmingWithC (besucht am 2026-05-10) (siehe S. 77).

[33] Thomas Weise (汤卫思). *Programming with Python*. Hefei, Anhui, China (中国安徽省合肥市): Hefei University (合肥大学), School of Artificial Intelligence and Big Data (人工智能与大数据学院), 2024–2026. URL: https://thomasweise.github.io/programmingWithPython (besucht am 2025-01-05) (siehe S. 78).

[34] Giorgio Zarrelli. *Mastering Bash*. Birmingham, England, UK: Packt Publishing Ltd, Juni 2017. ISBN: 978-1-78439-687-9 (siehe S. 77).

Glossary (in English) I



Bash is a shell used under Ubuntu Linux, i.e., the program that „runs“ in the terminal and interprets your commands, allowing you to start and interact with other programs[3,19,34]. Learn more at https://www.gnu.org/software/bash.

C is a programming language, which is very successful in system programming situations[5,21,32].

docstring Docstrings are special string constants in Python that contain documentation for modules or functions[6]. They must be delimited by `"""..."""`[6,29].

Git is a distributed Version Control System (VCS) which allows multiple users to work on the same code while preserving the history of the code changes[24,27]. Learn more at https://git-scm.com.

GitHub is a website where software projects can be hosted and managed via the Git VCS[20,27]. Learn more at https://github.com.

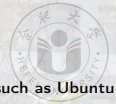
IT information technology

linter A linter is a tool for analyzing program code to identify bugs, problems, vulnerabilities, and inconsistent code styles[10,22]. Ruff is an example of a linter used in the Python world.

Linux is the leading open source operating system, i.e., a free alternative to Microsoft Windows[1,7,23,26,30]. We recommend using it for this course, for software development, and for research. Learn more at https://www.linux.org. Its variant Ubuntu is particularly easy to use and install.

literal A literal is a specific concrete value, something that is written down as-is[13,25]. In Python, for example, `"abc"` is a string literal, `5` is an integer literal, and `23.3` is a `float` literal. In contrast, `sin(3)` is not a literal. Also, while `5` is an integer literal, if we create a variable `a = 5` then `a` is not a literal either (it is a variable). Hence, literals are values that the Python interpreter reads directly from the source code and creates as objects in memory. They are not something that is the result from a computation or the result of a variable lookup. Python supports some type hints for literals, including the type `LiteralString` for string literals and the type `Literal[xyz]` for arbitrary literals `xyz`.

Glossary (in English) II



Microsoft Windows is a commercial proprietary operating system[2]. It is widely spread, but we recommend using a Linux variant such as Ubuntu for software development and for our course. Learn more at https://www.microsoft.com/windows.

Mypy is a static type checking tool for Python[14] that makes use of type hints. Learn more at https://github.com/python/mypy and in [33].

Python The Python programming language[9,12,15,33], i.e., what you will learn about in our book[33]. Learn more at https://python.org.

Ruff is a linter and code formatting tool for Python[16,17]. Learn more at https://docs.astral.sh/ruff or in [33].

terminal A terminal is a text-based window where you can enter commands and execute them[1,4]. Knowing what a terminal is and how to use it is very essential in any programming- or system administration-related task. If you want to open a terminal under Microsoft Windows, you can Druck auf ⊞+R, dann Schreiben von `cmd`, dann Druck auf ↵. Under Ubuntu Linux, Ctrl+Alt+T opens a terminal, which then runs a Bash shell inside.

type hint are annotations that help programmers and static code analysis tools such as Mypy to better understand what type a variable or function parameter is supposed to be[11,28]. Python is a dynamically typed programming language where you do not need to specify the type of, e.g., a variable. This creates problems for code analysis, both automated as well as manual: For example, it may not always be clear whether a variable or function parameter should be an integer or floating point number. The annotations allow us to explicitly state which type is expected. They are *ignored* during the program execution. They are a basically a piece of documentation.

Ubuntu is a variant of the open source operating system Linux[4,8]. We recommend that you use this operating system to follow this class, for software development, and for research. Learn more at https://ubuntu.com. If you are in China, you can download it from https://mirrors.ustc.edu.cn/ubuntu-releases.

Glossary (in English) III



VCS
: A *Version Control System* is a software which allows you to manage and preserve the historical development of your program code[27]. A distributed VCS allows multiple users to work on the same code and upload their changes to the server, which then preserves the change history. The most popular distributed VCS is Git.

virtual environment
: A virtual environment is a directory that contains a local Python installation[18,31]. It comes with its own package installation directory. Multiple different virtual environments can be installed on a system. This allows different applications to use different versions of the same packages without conflict, because we can simply install these applications into different virtual environments.