Clean Python – wartbarer und idiomatischer Code

Python-Code verständlich, wartbar und pythonisch gestalten

01 INTRO

Funktionierender Python-Code ist nicht automatisch guter Python-Code. Mit wachsendem Umfang werden Verständlichkeit, klare Strukturen und einfache Änderbarkeit zunehmend wichtiger.

Der Kurs zeigt, wie vorhandener und neuer Python-Code so gestaltet wird, dass er verständlich, testbar und wartbar bleibt. Im Mittelpunkt stehen Python-Idiome, klare Verantwortlichkeiten, sinnvolle Abstraktionen und systematisches Refactoring – ohne aus einfachen Lösungen unnötig komplizierte Architekturen zu machen.

DAUER

3 Tage

FORMAT

Präsenz · Live Online · Inhouse

VORKENNTNISSE

Sichere Python-Kenntnisse und praktische Programmiererfahrung

02 TRAININGSPROGRAMM

Inhalte

Was macht guten Python-Code aus?

- Lesbarkeit als Qualitätsmerkmal
- Code wird häufiger gelesen als geschrieben
- Verständlichkeit und Änderbarkeit
- Einfachheit statt Cleverness
- Python-Konventionen und Python-Idiome
- technische Schulden erkennen

Pythonisch programmieren

- idiomatische Schleifen und Iterationen
- Comprehensions sinnvoll einsetzen
- Packing und Unpacking
- Truth Values
- enumerate, zip und andere Built-ins
- Generator Expressions
- Context Manager
- Standardbibliothek statt eigener Hilfskonstruktionen

Namen und Verständlichkeit

- aussagekräftige Namen
- Variablen, Funktionen, Klassen und Module benennen
- Abkürzungen und Kontext
- Kommentare sinnvoll einsetzen
- Kommentare als Warnsignal
- Docstrings
- Code als Dokumentation

Funktionen gestalten

- kleine und fokussierte Funktionen
- eine klare Aufgabe pro Funktion
- Parameter begrenzen
- Rückgabewerte verständlich gestalten
- Seiteneffekte reduzieren
- Guard Clauses
- komplexe Bedingungen auflösen
- Funktionen als Bausteine

Klassen und Verantwortlichkeiten

- wann eine Klasse sinnvoll ist
- Verantwortlichkeiten klar zuordnen
- kleine Schnittstellen
- Kohäsion und Kopplung
- Komposition statt unnötiger Vererbung
- Dataclasses
- zustandslose Klassen vermeiden
- Funktionen oder Klassen?

Module und Packages strukturieren

- zusammengehörigen Code zusammenführen
- Modulgrenzen
- öffentliche und interne Schnittstellen
- Abhängigkeiten zwischen Modulen
- zyklische Abhängigkeiten vermeiden
- Packages sinnvoll schneiden
- Struktur mit wachsendem Projektumfang

Python Code Smells

- lange Funktionen
- große Klassen
- zu viele Parameter
- tiefe Verschachtelungen
- duplizierter Code
- globale Zustände
- Magic Numbers und Magic Strings
- übermäßige Vererbung
- unnötige Abstraktionen
- Clever Code

Refactoring

- Refactoring und funktionale Änderung unterscheiden
- kleine kontrollierte Änderungen
- Extract Function
- Extract Class
- Rename
- Bedingungen vereinfachen
- Duplikation beseitigen
- Verantwortlichkeiten verschieben
- bestehende Schnittstellen erhalten

Tests als Sicherheitsnetz

- Refactoring und automatisierte Tests
- Verhalten vor Änderungen absichern
- Characterization Tests bei bestehendem Code
- kleine Refactoring-Schritte
- Tests nach strukturellen Änderungen
- Testbarkeit als Hinweis auf gutes Design

Fehlerbehandlung

- Exceptions gezielt einsetzen
- Fehler dort behandeln, wo sie sinnvoll behandelt werden können
- spezifische statt allgemeiner Exceptions
- eigene Exceptions
- Fehler nicht verschlucken
- aussagekräftige Fehlermeldungen
- Logging und Exceptions unterscheiden

Typisierung als Entwicklungswerkzeug

- Type Hints gezielt einsetzen
- Typen an Schnittstellen
- komplexe Typdefinitionen vermeiden
- Protocol
- Type Aliases
- statische Typprüfung
- Typisierung und Lesbarkeit
- wann Typisierung mehr schadet als hilft

Abhängigkeiten beherrschen

- Abhängigkeiten sichtbar machen
- Dependency Injection ohne Framework
- globale Abhängigkeiten vermeiden
- externe Systeme kapseln
- testbare Schnittstellen
- lose Kopplung
- Abstraktionen nur dort einführen, wo sie gebraucht werden

Overengineering vermeiden

- YAGNI
- DRY mit Augenmaß
- Duplikation versus falsche Abstraktion
- unnötige Frameworks und Schichten
- vorzeitige Generalisierung
- vorzeitige Optimierung
- einfache Lösungen bevorzugen

Werkzeuge für Codequalität

- Formatter
- Linter
- statische Analyse
- Type Checker
- automatisierte Tests
- Code-Metriken im Überblick
- Qualitätsprüfungen automatisieren
- Werkzeuge als Unterstützung, nicht als Ersatz für Engineering

Bestehenden Code verbessern

- fremden Code systematisch erschließen
- problematische Stellen identifizieren
- Risiken einer Änderung abschätzen
- Verhalten zunächst absichern
- schrittweise refaktorieren
- Verbesserungen überprüfen
- wann man Code besser in Ruhe lässt

03 DURCHFÜHRUNG

Rahmen und Organisation

Dauer	3 Tage
Durchführung	Präsenz oder Live Online, vorzugsweise als Inhouse-Schulung
Schwerpunkt	Inhouse-Schulungen für Unternehmen und Teams
Sprache	Deutsch, englische Fachbegriffe und Dokumentation
Unterlagen	Kursunterlagen, Beispiele und Übungsaufgaben
Technik	Eigener Rechner mit aktueller Python-Version und Entwicklungsumgebung

> **Inhouse-Schulungen**
> Die Schwerpunkte können an vorhandene Kenntnisse, eingesetzte Technologien und konkrete Anforderungen des Teams angepasst werden.

04 ZIELGRUPPE

Für wen ist der Kurs gedacht?

Der Kurs richtet sich an Python-Entwicklerinnen und -Entwickler, die bereits praktisch mit Python arbeiten und die Qualität ihrer Programme systematisch verbessern möchten.

Er eignet sich insbesondere für Teilnehmer, die vorhandenen oder gewachsenen Python-Code warten und weiterentwickeln, Code Reviews durchführen oder gemeinsame Entwicklungsstandards im Team etablieren möchten.

Die Teilnehmer sollten über praktische Python-Erfahrung verfügen und mit folgenden Themen sicher umgehen können:

- Funktionen
- Module und Packages
- Klassen und Objekte
- Exceptions
- Datenstrukturen
- grundlegende automatisierte Tests

> **Kein Einstiegskurs**
> Der Kurs ist nicht als Python-Einstieg gedacht.

05 LERNZIELE

Was Sie nach dem Kurs können

Nach dem Kurs können die Teilnehmer:

- Python-Code hinsichtlich Verständlichkeit und Wartbarkeit beurteilen
- typische Python-Idiome gezielt einsetzen
- Funktionen, Klassen und Module mit klaren Verantwortlichkeiten gestalten
- häufige Code Smells erkennen
- bestehenden Code schrittweise und kontrolliert refaktorieren
- Tests als Sicherheitsnetz für Änderungen einsetzen
- Abhängigkeiten reduzieren und Schnittstellen klarer gestalten
- Type Hints sinnvoll zur Unterstützung der Entwicklung einsetzen
- unnötige Komplexität und Overengineering erkennen
- Werkzeuge zur automatisierten Qualitätsprüfung einsetzen
- bei Code Reviews konkrete und nachvollziehbare Verbesserungen vorschlagen
- zwischen sinnvoller Verbesserung und unnötigem Refactoring unterscheiden

06 FAQ

Häufige Fragen

Wie unterscheidet sich der Kurs von PY-002 „Python für Fortgeschrittene“?

PY-002 vertieft die Sprache und ihre fortgeschrittenen Möglichkeiten. PY-009 konzentriert sich darauf, wie Python-Code strukturiert, beurteilt, gewartet und verbessert wird.

Geht es hauptsächlich um PEP 8?

Nein. Formatierungs- und Namenskonventionen gehören dazu, sind aber nur ein kleiner Teil von Codequalität. Der Schwerpunkt liegt auf Verständlichkeit, Struktur, Verantwortlichkeiten, Abhängigkeiten und Änderbarkeit.

Wird Clean Code nach Robert C. Martin behandelt?

Clean-Code-Prinzipien werden dort aufgegriffen, wo sie für Python sinnvoll sind. Sie werden aber nicht schematisch übernommen. Python hat eigene Idiome und ermöglicht häufig einfachere Lösungen.

Werden SOLID und Design Patterns behandelt?

Einzelne Prinzipien können bei konkreten Designproblemen eine Rolle spielen. Der Kurs ist jedoch weder ein SOLID- noch ein Design-Patterns-Kurs. Im Vordergrund steht die praktische Verbesserung von Python-Code.

Wird mit bestehendem Code gearbeitet?

Ja. Das Analysieren und Refaktorieren bestehenden Codes ist ein wesentlicher Bestandteil des Themas. Bei Inhouse-Schulungen können geeignete Beispiele aus der vorhandenen Codebasis einbezogen werden.

Werden Tools wie Ruff, Black oder mypy eingesetzt?

Geeignete Formatter, Linter, Type Checker und Analysewerkzeuge werden praktisch eingesetzt. Die konkrete Toolauswahl kann an die Entwicklungsumgebung des Unternehmens angepasst werden.

Ist der Kurs auch für Teams mit gewachsenen Python-Anwendungen geeignet?

Ja. Gerade bei länger bestehenden Anwendungen können Schwerpunkte auf Wartbarkeit, technische Schulden, Refactoring und gemeinsame Entwicklungsstandards gelegt werden.

UC IT Service · Ulrich Cuber · kontakt@uc-it.de
https://www.uc-it.de/coaching/clean-python/