Nebenläufigkeit und Parallelisierung mit Python

Threads, Prozesse und AsyncIO gezielt einsetzen

01 INTRO

Sobald Programme mehrere Aufgaben gleichzeitig bearbeiten, auf externe Systeme warten oder rechenintensive Arbeiten verteilen sollen, reicht ein rein sequenzieller Ablauf häufig nicht mehr aus. Python bietet dafür unterschiedliche Modelle – mit jeweils eigenen Stärken, Grenzen und Fehlerquellen.

Der Kurs zeigt, wie Nebenläufigkeit und Parallelisierung mit Python praktisch umgesetzt werden. Im Mittelpunkt steht die Entscheidung, welcher Ansatz für eine konkrete Aufgabe geeignet ist: Threads, Prozesse oder asynchrone Programmierung mit AsyncIO.

DAUER

3 Tage

FORMAT

Präsenz · Live Online · Inhouse

VORKENNTNISSE

Sichere Python-Kenntnisse und praktische Programmiererfahrung

02 TRAININGSPROGRAMM

Inhalte

Nebenläufigkeit und Parallelität verstehen

- sequenzielle und nebenläufige Verarbeitung
- Concurrency und Parallelism
- I/O-bound und CPU-bound
- Latenz und Durchsatz
- typische Einsatzgebiete
- Kosten und Komplexität paralleler Verarbeitung
- wann Nebenläufigkeit nicht sinnvoll ist

Python und der GIL

- Ausführung von Python-Code
- Global Interpreter Lock
- Auswirkungen auf Threads
- CPU-bound und I/O-bound Workloads
- Prozesse als Alternative
- Free-Threaded Python und aktuelle Entwicklungen
- Konsequenzen für bestehende Anwendungen

Arbeiten mit Threads

- threading
- Threads erzeugen und starten
- Lebenszyklus eines Threads
- Daten zwischen Threads
- Daemon Threads
- Exceptions in Threads
- typische Einsatzgebiete
- Grenzen von Threads

Synchronisation

- gemeinsam genutzter Zustand
- Race Conditions
- Locks
- RLock
- Semaphoren
- Events und Conditions
- kritische Abschnitte
- Deadlocks
- Synchronisation möglichst vermeiden

Kommunikation mit Queues

- Producer-Consumer-Prinzip
- queue.Queue
- Aufgaben verteilen
- Ergebnisse zurückgeben
- Backpressure
- kontrolliertes Beenden von Workern
- robuste Worker-Strukturen

Thread- und Process-Pools

- concurrent.futures
- ThreadPoolExecutor
- ProcessPoolExecutor
- Aufgaben einreichen
- Futures
- Ergebnisse einsammeln
- Exceptions behandeln
- map
- Anzahl der Worker sinnvoll bestimmen

Parallelisierung mit Prozessen

- multiprocessing
- Prozesse erzeugen
- Prozessgrenzen und Speicher
- Daten zwischen Prozessen übertragen
- Serialisierung
- Queues und Pipes
- Process Pools
- CPU-intensive Aufgaben parallelisieren
- Overhead berücksichtigen

Einstieg in AsyncIO

- asynchrone Programmierung
- Event Loop
- Coroutines
- async und await
- Tasks
- asynchrone Funktionen aufrufen
- mehrere Aufgaben gleichzeitig ausführen
- blockierenden Code erkennen

Tasks koordinieren

- Tasks erzeugen und verwalten
- mehrere Coroutines ausführen
- Ergebnisse sammeln
- Timeouts
- Cancellation
- Exceptions in asynchronen Tasks
- Task Groups
- strukturierte Nebenläufigkeit

Asynchrone Kommunikation

- asyncio.Queue
- Producer und Consumer
- Locks und Semaphoren
- Events
- konkurrierende Zugriffe begrenzen
- Ressourcen kontrolliert verwenden
- asynchrone Pipelines

Blockierenden und asynchronen Code verbinden

- synchrone Bibliotheken in AsyncIO-Anwendungen
- blockierende Funktionen erkennen
- Threads aus AsyncIO verwenden
- asyncio.to_thread
- bestehende Anwendungen schrittweise erweitern
- Grenzen gemischter Architekturen

Fehlerbehandlung und kontrolliertes Beenden

- Fehler in parallelen Aufgaben
- Exceptions propagieren
- Timeouts
- Cancellation
- Ressourcen freigeben
- laufende Aufgaben beenden
- Graceful Shutdown
- Fehler einzelner Worker behandeln

Typische Nebenläufigkeitsfehler

- Race Conditions
- Deadlocks
- Starvation
- verlorene Updates
- unkontrollierte Task-Erzeugung
- blockierende Operationen im Event Loop
- schwer reproduzierbare Fehler
- Nebenläufigkeit möglichst einfach halten

Performance messen

- Laufzeit messen
- sequenzielle und parallele Varianten vergleichen
- Durchsatz und Latenz
- Overhead von Threads und Prozessen
- Skalierung untersuchen
- Profiling im Überblick
- messen statt vermuten

Nebenläufigen Code testen und debuggen

- deterministische Tests anstreben
- Timeouts in Tests
- Race Conditions reproduzieren
- Synchronisationspunkte kontrollieren
- Fehler in Threads und Tasks sichtbar machen
- Logging mit mehreren Ausführungseinheiten
- typische Probleme bei automatisierten Tests

Den richtigen Ansatz wählen

- sequenzielle Verarbeitung
- Threads
- Prozesse
- AsyncIO
- hybride Ansätze
- Entscheidung anhand der konkreten Aufgabe
- Komplexität gegen möglichen Nutzen abwägen
- einfache Architektur bevorzugen

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 Anwendungen mit parallelen oder nebenläufigen Abläufen entwickeln oder bestehende Programme hinsichtlich Durchsatz und Reaktionsfähigkeit verbessern möchten.

Er eignet sich insbesondere für Entwickler, die Threads, Multiprocessing oder AsyncIO bereits punktuell eingesetzt haben und die unterschiedlichen Modelle systematisch verstehen und gezielt auswählen möchten.

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

- Funktionen
- Module und Packages
- Klassen und Objekte
- Exceptions
- Context Manager
- grundlegende Datenstrukturen

> **Erfahrung mit Nebenläufigkeit**
> Erfahrungen mit Threads, Prozessen oder asynchroner Programmierung sind nicht erforderlich.

05 LERNZIELE

Was Sie nach dem Kurs können

Nach dem Kurs können die Teilnehmer:

- Nebenläufigkeit und Parallelität voneinander unterscheiden
- I/O-bound und CPU-bound Aufgaben erkennen
- Auswirkungen des Python-Ausführungsmodells auf parallele Programme beurteilen
- Threads für geeignete Aufgaben einsetzen
- gemeinsam genutzte Ressourcen sicher synchronisieren
- Thread- und Process-Pools verwenden
- CPU-intensive Aufgaben auf mehrere Prozesse verteilen
- asynchrone Anwendungen mit AsyncIO entwickeln
- Tasks koordinieren, abbrechen und auf Fehler reagieren
- synchrone und asynchrone Komponenten miteinander verbinden
- typische Race Conditions und Deadlocks erkennen und vermeiden
- Performance-Gewinne durch Messungen überprüfen
- nebenläufigen Code testbarer und diagnostizierbarer gestalten
- für eine konkrete Aufgabe zwischen Threads, Prozessen, AsyncIO und sequenzieller Verarbeitung entscheiden

06 FAQ

Häufige Fragen

Was ist der Schwerpunkt des Kurses: Threads, Multiprocessing oder AsyncIO?

Alle drei Ansätze werden behandelt und miteinander verglichen. Ein wesentliches Ziel des Kurses ist zu verstehen, welcher Ansatz für welche Art von Problem geeignet ist.

Muss ich async und await bereits kennen?

Nein. AsyncIO wird systematisch eingeführt. Sichere allgemeine Python-Kenntnisse werden jedoch vorausgesetzt.

Wird der Global Interpreter Lock behandelt?

Ja. Der GIL und seine Auswirkungen gehören zum Verständnis klassischer Python-Nebenläufigkeit. Dabei werden auch die Entwicklungen rund um Free-Threaded Python berücksichtigt.

Geht es hauptsächlich um Performance?

Nein. Performance ist ein wichtiger Anwendungsfall, aber nicht der einzige. Nebenläufigkeit wird beispielsweise auch benötigt, um auf mehrere externe Systeme zu warten oder Anwendungen reaktionsfähig zu halten.

Wird FastAPI behandelt?

Nein. Asynchrone Programmierung mit async und await ist zwar auch für FastAPI relevant, der Kurs behandelt das Thema jedoch unabhängig von einem Webframework. FastAPI ist Gegenstand von PY-006.

Werden verteilte Systeme behandelt?

Nein. Der Schwerpunkt liegt auf Nebenläufigkeit und Parallelisierung innerhalb einer Python-Anwendung beziehungsweise eines Systems. Verteilte Architekturen und Message Broker sind nicht Gegenstand des Kurses.

Kann der Kurs an unsere konkreten Anwendungen angepasst werden?

Ja. Bei Inhouse-Schulungen können typische Workloads des Unternehmens einbezogen werden, beispielsweise parallele Datenverarbeitung, API-Aufrufe, Batch-Prozesse oder bestehende AsyncIO-Anwendungen.

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