LU07.L11 - Den echten Hotspot finden

Das Profil

```
         12004 function calls in 0.104 seconds

   Ordered by: cumulative time

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.002    0.002    0.104    0.104 pruefen
     3000    0.099    0.000    0.099    0.000 ist_gesperrt
     3000    0.002    0.000    0.003    0.000 normalisieren
     3000    0.001    0.000    0.001    0.000 {method 'strip' of 'str'
objects}
```

Hotspot: `ist_gesperrt` mit 0.099 von 0.104 Sekunden, also 95 Prozent. `normalisieren` wird **gleich oft** aufgerufen und kostet 0.003 Sekunden, also 3 Prozent.

Die Optimierung

```python
gesperrt = {f"user{i}" for i in range(5000)}    # set statt list
```

```
vorher   0.0955 s
nachher  0.00040 s     -> Faktor 240, identisches Ergebnis (1467 Treffer)
```

Eine geänderte Klammer. Die Funktion `ist_gesperrt` selbst bleibt unverändert - nur die Datenstruktur, die sie durchsucht.

Die zweite Optimierung

Wer zusätzlich die Funktionsaufrufe einspart und alles in die Comprehension zieht:

```python
def pruefen(namen):
    return [n for n in namen if n.strip().lower() in gesperrt]
```

```
mit Hilfsfunktionen  0.00040 s
inline               0.00026 s     -> nochmals 0.14 Millisekunden
```

War das gerechtfertigt? Nein. Der Gewinn beträgt 0.15 Prozent der ursprünglichen Laufzeit, und dafür verschwinden zwei benannte Funktionen, die den Code lesbar und einzeln testbar gemacht haben. Das ist ein Refactoring in die falsche Richtung, verkauft als Optimierung.

Antworten

Warum wirkt `normalisieren` teuer? Weil dort sichtbar Arbeit steht - `strip` und `lower` auf 3000 Strings. Der Blick bleibt an der Zeile hängen, die etwas *tut*. `ist_gesperrt` sieht harmlos aus: ein einziges `in`. Dass sich dahinter 5000 Vergleiche pro Aufruf verbergen, sieht man dem Code nicht an - nur der Datenstruktur.

Was sagt `ncalls`? Beide Funktionen werden genau 3000-mal aufgerufen, einmal pro Anfrage. Die Aufrufzahl allein sagt also nichts über die Kosten. Auffällig wäre eine Zahl, die man nicht erwartet - etwa 9'000'000 statt 3000: dann steckt ein Aufruf in einer Schleife, die er nicht enthalten sollte.

`tottime` gegen `cumtime` bei `pruefen`: `tottime` 0.002 s ist die Zeit in `pruefen` selbst, also im Aufbau der Liste. `cumtime` 0.104 s zählt alles mit, was `pruefen` aufruft. Die Differenz ist die Zeit der Unterfunktionen. Deshalb sucht man den Hotspot in der `tottime`-Spalte, nicht in `cumtime` - sonst steht immer die äusserste Funktion oben.

Die Lehre: Optimieren Sie nur, was die Messung zeigt - und hören Sie auf, sobald der Hotspot weg ist. Nach der ersten Änderung war das Problem gelöst; alles Weitere hat nur Lesbarkeit gekostet.

M323-LU07, M323-D1A