Profiling in Python

Seminar: Effiziente Programmierung

Jan Pohlmann

November 2017

Gliederung

1. Grundlagen Profiling
   - Was ist das
   - Ziele und Herausforderungen
   - Klassifizierung
   - Zeitpunkt
2. Python
3. Profiling
   - Python Built-ins
   - externe Bibliotheken

Profiling - Was ist das?

- *"[...] verallgemeinert die Erstellung, Aktualisierung und Verwendung von Profilen aus der Sammlung, Analyse, Auswertung und Rückkopplung heutzutage hauptsächlich im Internet gewonnener Daten, beispielsweise zum Zweck der Identifikation, Optimierung, Überwachung [...]"* *https://de.wikipedia.org/wiki/Profiling*
- Laufzeit-Analyse von Progammen oder Programmteilen
- Messeinheiten:
  - Zeit
  - CPU-Zeit
  - Speicherverbrauch

Profiling - Ziele

- Identifizieren von Engpässen
- Vergleich der Performance auf unterschiedlichen Umgebungen
- Effizienzanalyse von Parallelisierung

Profiling - Herausforderungen

- Profiler ist selbst ein Programm
  - Mehr oder minder effizient
- Einfluss auf die Laufzeit des Zielprogrammes (Overhead)
  - Unterbrechen des Zielprogramms zum Sammeln von Daten
  - Speichern der erfassten Daten

Profiling - Klassifizierung

- Event-basiert
  - Sammelt Informationen bei Events, wie
    - Start/Ende einer Funktion
    - Exception
    - ...
- Sampling-basiert
  - Speichert Informationen in definiertem Zeitabstand

Profiling - Zeitpunkt

- Grundsätzlich in allen Projektphasen möglich
- Prioritäten in der Entwicklung
  - Hoher Detaillierungsgrad
- Prioritäten in Produktion
  - Geringer Overhead

Python

- 1990 von Guido van Rossum entwickelt
- Dynamisch typisiert
- In Bytecode compiliert, dann interpretiert
- Unterstützung aller gängigen Progammierparadigmen
- Ziele
  - Einfachheit
  - Produktivität
- Vorinstalliert in Linux
- Steigende Verbreitung in der Wissenschaft

Python - wissenschaftliches Arbeiten

- Erweiterungen
  - PyPy
    - Alternative Python-Implementierung
    - Interpretation des Codes mit Just-In-Time Compiler (JIT)
  - NumPy
    - Bietet Strukturen und Algorithmen auf Basis von mehrdimensionalen Arrays (ndarray)
  - Cython
    - Erweiterung von Python um optionale, statische Typisierung
    - Wird in C übersetzt und in Python compiliert

Python - wissenschaftliches Arbeiten

- Builts-ins
  - Umfangreiche Standardbibliothek
    - Datenstrukturen
    - Algorithmen
  - Modul: multiprocessing
  - Verschiedene Profiling-Bibliotheken

Profiling - Python Built-ins

- Modul 'time':

```
1: start = time.time()
2:
3: [...CODE HERE ...]
4:
5: end = time.time()
6: total = end - start
```

Profiling - Python Built-ins

- Modul 'timeit':
  aus der Shell:
  $python -m timeit "1+2"
  10000000 loops, best of 3: 0.0196 usec per loop

  im Skript:

```
1: import timeit
2: timeit.timeit(*METHODNAME*)
```

Profiling - Python Built-ins

- time/timeit
  - Vorteil
    - Schnell
    - Einfach
  - Nachteil
    - Wenig Informationen

Profiling - Python Built-ins

- cProfile

```
>>> import cProfile
>>> import sleep
>>> cProfile.run('sleep.main()')
slept 1 secs
slept 2 secs
slept 3 secs
         12 function calls in 6.007 seconds

   Ordered by: standard name

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.000    0.000    6.007    6.007 <string>:1(<module>)
        1    0.000    0.000    6.007    6.007 sleep.py:12(main)
        1    0.000    0.000    1.001    1.001 sleep.py:4(fast)
        1    0.000    0.000    2.002    2.002 sleep.py:6(medium)
        1    0.000    0.000    3.003    3.003 sleep.py:8(slow)
        1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' objects}
        3    0.000    0.000    0.000    0.000 {method 'format' of 'str' objects}
        3    6.007    2.002    6.007    2.002 {time.sleep}
```

Figure 1: cProfile aus der Python-Konsole

Profiling - Python Built-ins

- cProfile

```
1: import cProfile
2: p = cProfile.Profile()
3: p.enable()
4:
5: [...CODE HERE ...]
6:
7: p.disable()
8: p.print_stats()
```

Profiling - Python Built-ins

- cProfile
  - Visualisierung mit pyprof2calltree und kcachgrind

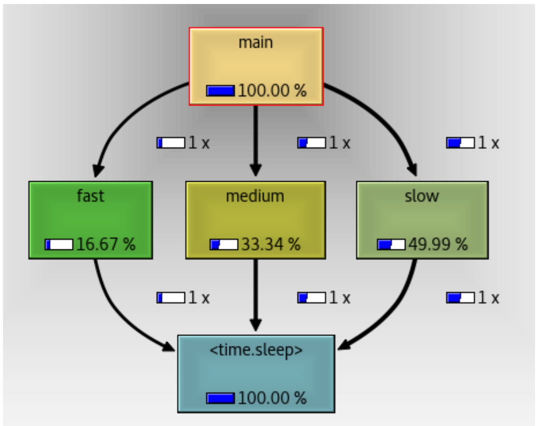

Figure 2: call-graph auf Basis der Daten von cProfile

Profiling - Python Built-ins

- cProfile : DEMO

Profiling - Python Built-ins

- cProfile
  - Klassifizierung:
    - verfolgt Start und Ende jeder ausgeführten Methode
      $\rightarrow$ event-basiert
  - Overhead
    - Besonders hoch bei vielen kleinen Methoden
    - Beispiel: $\sim$Faktor 2,5

Profiling - Python Built-ins

- cProfile
  - Vorteile
    - Schnelle und einfache Nutzung
    - Liefert Daten kompatibel für verschiedene Visualisierungswerkzeuge
  - Nachteile
    - Hoher Overhead bei vielen Funktionsaufrufen

Profiling - externe Bibliotheken

- statProf

```
1: import statprof
2: import profileMe
3:
4: statprof.start()
5: try:
6:     profileMe.main()
7: finally:
8:     statprof.stop()
9:     statprof.display()
```

Profiling - externe Bibliotheken

- statProf
  - installierbar mit pip ("pip installs packages")

```
./statProfMe.py
  %   cumulative      self
 time     seconds   seconds  name
 25.00       0.06      0.06  profileMe.py:13:slow
 25.00       0.11      0.06  profileMe.py:19:main
 25.00       0.06      0.06  profileMe.py:9:medium
 25.00       0.11      0.06  profileMe.py:17:main
  0.00       0.22      0.00  statProfMe.py:7:<module>
---
Sample count: 4
Total time: 0.220000 seconds
```

Profiling - externe Bibliotheken

- statProf
  - Klassifizierung
    - Messeinheit: CPU-Zeit
    - Sampling-basiert (default-Intervall:1ms)
  - Overhead
    - Grundsätzlich geringer
    - Variable: Abhängig der Samplingrate

Profiling - externe Bibliotheken

- statProf
  - Vorteile:
    - Geringer Overhead
  - Nachteile:
    - Wenig Details
    - Funktioniert nur für Main-Thread
    - Keine direkte Visualisierung

Profiling - externe Bibliotheken

- memory_profiler
  - Installierbar mit pip
  - Abhängigkeiten(optional): matplotlib, psutil
  - @profile-Dekorator an Methoden

```
Filename: profileMe.py

Line #    Mem usage    Increment   Line Contents
================================================
 14   13.941 MiB    0.000 MiB   @profile
 15                             def slow():
 16  502.234 MiB  488.293 MiB       b = bytearray(512000000)
 17  502.234 MiB    0.000 MiB       time.sleep(3)
```

Profiling - externe Bibliotheken

- Visualisierung mit mprof

```
$ mprof run myScript.py
$ mprof plot
```

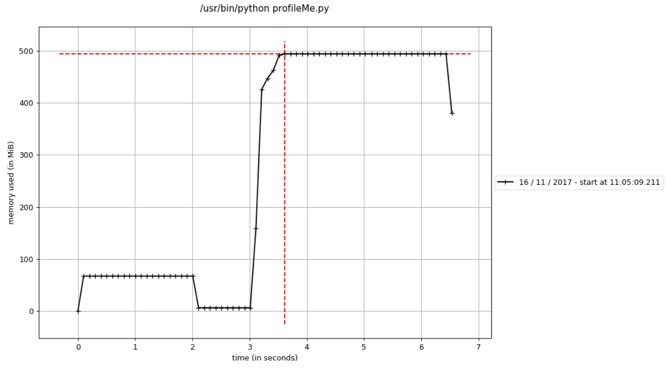

Figure 3: Visualisierung mit mprof

Profiling - externe Bibliotheken

- memory_profiler: DEMO

Profiling - externe Bibliotheken

- memory_profiler
  - Klassifizierung:
    - Event-basiert
    - Messeinheit: Speicher
  - Overhead:
    - Hoch (variabel mit Anzahl der Zeilen)

Profiling - externe Bibliotheken

- memory_profiler
  - Vorteile:
    - Enthält mprof zur Visualisierung
    - Profilen einzelner Methoden möglich
  - Nachteile:
    - Ungeeignet für sehr langen Code
    - Codeanpassung (Dekorator) notwendig

Zusammenfassung

- Umfangreiche Auswahl an Profiler
  - Hauptunterschiede
    - Event-basiert/Sampling-basiert
    - Messeinheit
- Jedes Modul hat Vor- und Nachteile
- Nutzung mehrerer Profiler sinnvoll

Quellen I

memory_profiler.
Eingesehen am 14.11.2017.

profile, cprofile, and pstats – performance analysis of python programs.
https://pymotw.com/2/profile/.
Eingesehen am 31.10.2017.

Profiler types and their overhead.
https://pymotw.com/2/profile/.
Eingesehen am 31.10.2017.

Profiling python in production.
https://www.nylas.com/blog/performance/.
Eingesehen am 31.10.2017.

Quellen II

Pypy.
https://pypy.org/.
Eingesehen am 05.11.2017.

Python 102: How to profile your code.
https://www.blog.pythonlibrary.org/2014/03/20/python-102-how-to-profile-your-code/.
Eingesehen am 31.10.2017.

Scipy-package.
https://docs.scipy.org/doc/numpy-1.13.0/index.html.
Eingesehen am 05.11.2017.

statprof.
Eingesehen am 17.11.2017.

Quellen III

M. F. SANNER.
Python: A programming language for software integration and development.
https://pdfs.semanticscholar.org/409d/3f740518eafcfaadb054d9239009f3f34600.pdf.
Eingesehen am 31.10.2017.