

# Informatik I: Einführung in die Programmierung

## 25. Effiziente Programme

Albert-Ludwigs-Universität Freiburg



Bernhard Nebel

20.01.2017

# 1 Motivation



Motivation

Das Modul  
timeit

Funktions-  
aufrufe  
zählen und  
Memoisie-  
rung

Das Modul  
cProfile

Tracing mit  
dem Modul  
trace

20.01.2017

B. Nebel – Info I

3 / 42

# Motivation



Motivation

Das Modul  
timeit

Funktions-  
aufrufe  
zählen und  
Memoisie-  
rung

Das Modul  
cProfile

Tracing mit  
dem Modul  
trace

- Wenn wir einen Algorithmus implementieren, so sind wir daran interessiert, ein allgemeines Problem zu lösen ...
- ..., d.h. es gibt viele Probleminstanzen, für die wir eine Lösung finden wollen.
- Unser Programm soll typischerweise also oftmals ausgeführt werden ...
- ... und wir wollen nicht unnötig lange auf die Lösungen warten.

In der Informatik geht es also nicht nur darum, maschinelle, automatisierte Verfahren zu entwickeln, es geht auch darum, dass diese Programme **effizient** sind.

Aber:

*We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil.*  
(Donald Knuth)

20.01.2017

B. Nebel – Info I

4 / 42

# Grundregeln



Motivation

Das Modul  
timeit

Funktions-  
aufrufe  
zählen und  
Memoisie-  
rung

Das Modul  
cProfile

Tracing mit  
dem Modul  
trace

- 1 Schreibe lesbaren Code.
- 2 Überprüfe Korrektheit (schreibe Test-Instanzen und überprüfe die Korrektheit der Implementierung nach jeder Veränderung des Codes)
- 3 Optimierte die Implementierung **dort, wo es sich lohnt!**
  - Softwarewerkzeuge können bei der Analyse hilfreich sein.
  - **Profiler**: ein Tool, mit dem sich das Laufzeitverhalten von Programm-Code analysieren lässt; erlaubt es verschiedene Implementierungen zu vergleichen
  - **Messen der Laufzeit**: Wie oft wird eine Funktion aufgerufen? Wie lange dauert das Ausführen der Funktion?
  - **Speichernutzung**: Wie viel Arbeitsspeicher wird benötigt? Wird nicht benötigter Speicher wieder freigegeben? In Python: **Garbage-Collection**

20.01.2017

B. Nebel – Info I

5 / 42

# Einige Daumenregeln für effizientes Programmieren



Es gibt eine Reihe von allgemeinen Regeln, die sich in Python-Büchern und Internet-Foren immer wieder finden:

## Daumenregeln

- Benutze Python-Tuples anstelle von Listen, sofern nur eine immutable Sequenz benötigt wird
- Benutze Iteratoren anstelle von großen Tuples oder Listen (sofern die Sequenz nicht wiederholt gebraucht wird)
- Benutze (wo möglich und sinnvoll) Python's built-in Funktionen und Datenstrukturen
- Benutze iterative anstelle rekursiver Lösungen, sofern möglich
- ...

Solche Daumenregeln sind aber im Einzelfall zu überprüfen ...

Motivation  
Das Modul  
timeit  
Funktions-  
aufrufe  
zählen und  
Memoisie-  
rung  
Das Modul  
cProfile  
Tracing mit  
dem Modul  
trace

# Rekursiv vs. iterativ



Ein beliebtes Beispiel um aufzuzeigen, dass iterative Lösungen in Python meist ein besseres Laufzeitverhalten aufweisen:

## factorial.py

```
def fac_rec(n):  
    if n > 1:  
        return n * fac_rec(n-1)  
    else:  
        return 1  
  
def fac_iter(n):  
    res = 1  
    for i in range(2, n+1):  
        res *= i  
    return res
```

Motivation  
Das Modul  
timeit  
Funktions-  
aufrufe  
zählen und  
Memoisie-  
rung  
Das Modul  
cProfile  
Tracing mit  
dem Modul  
trace

# Zeitmessung mit dem Modul time

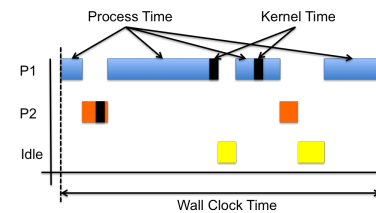


## time

```
if __name__ == '__main__':  
    import time  
    t0 = time.time()  
    for _ in range(1000000):  
        fac_iter(20)  
    delta = time.time() - t0  
    print("fac_iter: %s sec." % delta)  
    t0 = time.time()  
    for _ in range(1000000):  
        fac_rec(20)  
    delta = time.time() - t0  
    print("fac_rec: %s sec." % delta)  
  
■ Unterschied ist Faktor 2!  
■ Aber Unterschiede im µSekunden-Bereich.
```

Motivation  
Das Modul  
timeit  
Funktions-  
aufrufe  
zählen und  
Memoisie-  
rung  
Das Modul  
cProfile  
Tracing mit  
dem Modul  
trace

# Welche Zeiten kann man messen?



- **Process time:** Die Zeit, in der die CPU für den Prozess aktiv war.
- **Kernel time:** Die Zeit der *Process time*, die in *Kernel*-(System-)Routinen verbracht wurde.
- **User space time = Process time - Kernel time**
- **Wall clock time:** Die tatsächlich vergangene Zeit vom Start des Prozesses bis zum Ende (inkl. Warten auf I/O und Schlafen).

Motivation  
Das Modul  
timeit  
Funktions-  
aufrufe  
zählen und  
Memoisie-  
rung  
Das Modul  
cProfile  
Tracing mit  
dem Modul  
trace

## Welche Zeiten sind relevant?



- Will man einen CPU-intensiven Algorithmus evaluieren, dann misst man normalerweise die Prozesszeit.  
(→ `time.process_time()`)
- Das Herausrechnen der *Kernelzeit* geht in Python nicht!
- Ist man an *Antwortzeiten* interessiert, dann ist die *Wanduhr-Zeit* entscheidend (→ `time.time()`). Auf einer Einbenutzermaschine mit einem CPU-intensiven Prozess sollte es praktisch keinen Unterschied zur Prozesszeit geben.
- **Wichtig:** Die eingebaute Uhr misst meist mit geringer Auflösung (z.B. nur 1 Sekunde oder Miko-Sekunde). Außerdem kann diese Uhr verstellt werden (automatisch oder manuell).
- Deshalb gibt es meist auch einen Zeitzähler mit hoher Auflösung (nsec), der nicht verstellt werden kann.  
(→ `time.perf_counter()`)

Motivation  
Das Modul  
`timeit`  
Funktions-  
aufrufe  
zählen und  
Memoisie-  
rung  
Das Modul  
`cProfile`  
Tracing mit  
dem Modul  
`trace`

## 2 Das Modul `timeit`



Motivation  
Das Modul  
`timeit`  
Funktions-  
aufrufe  
zählen und  
Memoisie-  
rung  
Das Modul  
`cProfile`  
Tracing mit  
dem Modul  
`trace`

## `timeit`: „Measure execution time of small code snippets“



Das Modul `timeit` erlaubt es kleine (und auch größere) Programm-Teile auf Ihre Laufzeit zu untersuchen:

`timeit`

```
>>> from timeit import timeit
>>> timeit('"-".join(str(n) for n in range(100))',
...       number=10000)
0.4888567019952461
>>> timeit('"-".join([str(n) for n in range(100)])',
...       number=10000)
0.4697451650281437
>>> timeit('"-".join(map(str, range(100)))',
...       number=10000)
0.38516487198648974
```

Motivation  
Das Modul  
`timeit`  
Funktions-  
aufrufe  
zählen und  
Memoisie-  
rung  
Das Modul  
`cProfile`  
Tracing mit  
dem Modul  
`trace`

## `timeit`: „Measure execution time ...“



Wie das `Doctest`-Modul hat das Modul `timeit` auch ein Command-Line Interface:

### `timeit` in der Konsole

```
$ python3 -m timeit '"-".join(str(n) for n in range(100))'
10000 loops, best of 3: 41.5 usec per loop
$ python3 -m timeit '"-".join([str(n) for n in range(100)])'
10000 loops, best of 3: 38.7 usec per loop
$ python3 -m timeit '"-".join(map(str, range(100)))'
10000 loops, best of 3: 32.4 usec per loop
```

Für weitere mögliche Optionen beim Aufruf aus der Konsole siehe die *Documentation* (<http://docs.python.org/3.3/library/timeit.html>).

Motivation  
Das Modul  
`timeit`  
Funktions-  
aufrufe  
zählen und  
Memoisie-  
rung  
Das Modul  
`cProfile`  
Tracing mit  
dem Modul  
`trace`

## timeit: „Measure execution time ...“



Das Modul stellt neben einer Klasse `Timer`, mit dem sich spezielle Timer-Objekte erzeugen lassen, die folgenden beiden Funktionen zur Verfügung:

- `timeit.timeit(stmt='pass', setup='pass', timer=<default timer>, number=1000000)`: Erzeugt eine `Timer`-Instanz mit dem gegebenen Python-Snippet `stmt` (quotiert) und einem Python-Snippet `setup`, der initial ausgeführt wird. `timer` ist per Default `time.perf_counter()`. Anschließend wird die `timeit()`-Methode des Timers `number`-oft ausgeführt.
- `timeit.repeat(stmt='pass', setup='pass', timer=<default timer>, repeat=3, number=1000000)`: wie die letzte Funktion mit dem Unterschied, dass die `repeat` Methode des Timers aufgerufen wird (und zwar `repeat`-mal).

Motivation  
Das Modul `timeit`  
Funktionsaufrufe zählen und Memoisierung  
Das Modul `cProfile`  
Tracing mit dem Modul `trace`

## timeit: „Measure execution time ...“



Wie kann man das nun benutzen, um verschiedene Funktionen zu testen?

### Eine `timeit`-Testfunktion

```
def func_a(): ...

def func_b(): ...

def func_c(): ...

repeat = 5
number = 1000
for fct in ("func_a", "func_b", "func_c"):
    t = timeit.repeat(
        "%s()" % fct,
        setup="from __main__ import %s" % fct,
        repeat=repeat, number=number)
    print("%s:" % fct, str(t))
```

Motivation  
Das Modul `timeit`  
Funktionsaufrufe zählen und Memoisierung  
Das Modul `cProfile`  
Tracing mit dem Modul `trace`

## 3 Funktionsaufrufe zählen und Memoisierung



Motivation  
Das Modul `timeit`  
Funktionsaufrufe zählen und Memoisierung  
Das Modul `cProfile`  
Tracing mit dem Modul `trace`

## Funktionsaufrufe zählen (1)



Oftmals resultiert eine ineffiziente Implementierung daher, dass ein und dasselbe Ergebnis iteriert berechnet wird.

Ein typisches Beispiel ist die Fibonacci-Funktion, die uns schon früher begegnet ist:

### `fibonacci.py`

```
def fib(n):
    if n <= 1:
        return n
    else:
        return fib(n-1) + fib(n-2)
```

Die folgende kleine Variante zählt die Aufrufe der Funktion `fib`

...

Motivation  
Das Modul `timeit`  
Funktionsaufrufe zählen und Memoisierung  
Das Modul `cProfile`  
Tracing mit dem Modul `trace`

## Funktionsaufrufe zählen (2)



```
fibonacci.py
def fib(n):
    global _calls
    _calls += 1
    if n <= 1:
        return n
    else:
        return fib(n-1) + fib(n-2)

if __name__ == "__main__":
    for n in range(2, 11):
        _calls = 0
        fib(n)
        print("fib(%s): fib has been called %s times" %
              (n, _calls))
```

Motivation  
Das Modul  
timeit  
Funktions-  
aufrufe  
zählen und  
Memoisie-  
rung  
Das Modul  
cProfile  
Tracing mit  
dem Modul  
trace

## Funktionsaufrufe zählen (3)



Wenn wir dies auf der Konsole ausführen, erhalten wir:

```
fib(2): fib has been called 3 times
fib(3): fib has been called 5 times
fib(4): fib has been called 9 times
fib(5): fib has been called 15 times
fib(6): fib has been called 25 times
fib(7): fib has been called 41 times
fib(8): fib has been called 67 times
fib(9): fib has been called 109 times
fib(10): fib has been called 177 times
```

Hoppla! Um fib(10) zu berechnen brauchen wir doch eigentlich nur die Ergebnisse von fib(9) bis fib(2)? Wieso also 177 Aufrufe?

Motivation  
Das Modul  
timeit  
Funktions-  
aufrufe  
zählen und  
Memoisie-  
rung  
Das Modul  
cProfile  
Tracing mit  
dem Modul  
trace

## Memoisierung (1)



Offensichtlich berechnet der Aufruf fib(10) fib(9) 1-mal, fib(8) 2-mal, fib(7) 3-mal, fib(6) 5-mal, etc.  
Lösung: speichere Resultate in einem Dictionary ...

```
fibonacci.py
_fibs = {}
def fib_memo(n):
    if n <= 1:
        return n
    elif n in _fibs:
        # Check fib(n) already calculated
        return _fibs[n]
    else:
        res = fib_memo(n-1) + fib_memo(n-2)
        _fibs[n] = res
        return res
```

Motivation  
Das Modul  
timeit  
Funktions-  
aufrufe  
zählen und  
Memoisie-  
rung  
Das Modul  
cProfile  
Tracing mit  
dem Modul  
trace

## Memoisierung (2)



Wenn wir jetzt die Funktionsaufrufe und die Lookups im Dictionary zählen wollen:

```
fibonacci.py
_fibs = {}
def fib_memo(n):
    global _calls, _lookups
    _calls += 1
    if n <= 1:
        return n
    elif n in _fibs:
        _lookups += 1
        return _fibs[n]
    else:
        res = fib_memo(n-1) + fib_memo(n-2)
        _fibs[n] = res
        return res
```

Motivation  
Das Modul  
timeit  
Funktions-  
aufrufe  
zählen und  
Memoisie-  
rung  
Das Modul  
cProfile  
Tracing mit  
dem Modul  
trace

## Memoisierung (3)



Ein Aufruf von

`fibonacci.py`

```
if __name__=="__main__":
    ...
    for n in range(2, 11):
        _calls = 0
        _lookups = 0
        _fibs = {}
        fib_memo(n)
        print("fib_memo(%s): fib_memo called %s "
              "times (with %s lookups)" %
              (n, _calls, _lookups))
```

liefert dann:

Motivation  
Das Modul  
timeit  
Funktions-  
aufrufe  
zählen und  
Memoisie-  
rung  
Das Modul  
cProfile  
Tracing mit  
dem Modul  
trace

## Memoisierung (4)



```
fib_memo(2): fib_memo called 3 times (with 0 lookups)
fib_memo(3): fib_memo called 5 times (with 0 lookups)
fib_memo(4): fib_memo called 7 times (with 1 lookups)
fib_memo(5): fib_memo called 9 times (with 2 lookups)
fib_memo(6): fib_memo called 11 times (with 3 lookups)
fib_memo(7): fib_memo called 13 times (with 4 lookups)
fib_memo(8): fib_memo called 15 times (with 5 lookups)
fib_memo(9): fib_memo called 17 times (with 6 lookups)
fib_memo(10): fib_memo called 19 times (with 7 lookups)
```

Hieraus wird klar, dass die Implementierung `fib_memo` bei weitem effizienter ist als die ursprüngliche.

Motivation  
Das Modul  
timeit  
Funktions-  
aufrufe  
zählen und  
Memoisie-  
rung  
Das Modul  
cProfile  
Tracing mit  
dem Modul  
trace

## 4 Das Modul cProfile



Motivation  
Das Modul  
timeit  
Funktions-  
aufrufe  
zählen und  
Memoisie-  
rung  
Das Modul  
cProfile  
Tracing mit  
dem Modul  
trace

## Profiling



- Bei größeren Programmen weiß man oft nicht, **wo** denn tatsächlich die Laufzeit „verbraten“ wird.
  - „Blindes“ Optimieren ist eher **kontraproduktiv**:
    - Man steckt viel Arbeit in das Optimieren von Programmstellen, die aber gar keinen signifikanten Anteil an der Gesamtlaufzeit haben;
    - insgesamt hat dadurch diese Arbeit keinen sichtbaren positiven Effekt;
    - und womöglich führt das auch zu schlechter lesbarem und/oder wartbarem Code.
  - Zuerst feststellen, wo sich Arbeit lohnt!
- Für diesen Zweck gibt es **Profiler**.

Motivation  
Das Modul  
timeit  
Funktions-  
aufrufe  
zählen und  
Memoisie-  
rung  
Das Modul  
cProfile  
Tracing mit  
dem Modul  
trace

## Das Modul: cProfile



- `cProfile.run(command, sort=-1)`:  
command ist das Python-Kommando (als String), das aufgerufen werden soll, sort spezifiziert, nach welcher Spalte sortiert werden soll.
- Es wird dann eine Tabelle ausgegeben, in der in jeder Zeile eine Funktion beschrieben wird. Es gibt folgende Spalten:
  - `ncalls`: Anzahl der Aufrufe. Im Falle von zwei Zahlen, beschreibt die erste Zahl die totale Anzahl von Aufrufen, die zweite die Anzahl der primitiven (nicht-rekursiven) Aufrufe.
  - `tottime`: CPU Sekunden, die die Funktion verbraucht hat ohne Zeit für aufgerufene Funktionen zu berücksichtigen.
  - `percall = tottime / ncalls`.
  - `cumtime`: CPU Sekunden, die die Funktion inklusive der Zeit für aufgerufene Funktionen verbraucht hat.
  - `percall = cumtime / ncalls`.

Motivation  
Das Modul  
timeit  
Funktions-  
aufrufe  
zählen und  
Memoisie-  
rung  
Das Modul  
cProfile  
Tracing mit  
dem Modul  
trace

20.01.2017

B. Nebel – Info I

29 / 42

## cProfile bei der Arbeit: fib und fib\_memo



### Python-Interpreter

```
>>> import cProfile
>>> cProfile.run('fib(35)')
```

```
29860706 function calls (4 primitive calls) in 10.555 seconds

Ordered by: standard name

ncalls  tottime  percall  cumtime  percall filename:lineno(function)
1        0.000    0.000    10.555    10.555 <string>:1(<module>)
29860703/1 10.555    0.000    10.555    10.555 fibonacci_cprofile.py:9(fib)
1        0.000    0.000    10.555    10.555 {built-in method exec}
1        0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' objects}
```

```
>>> cProfile.run('fib_memo(35)')
```

```
72 function calls (4 primitive calls) in 0.000 seconds

Ordered by: standard name

ncalls  tottime  percall  cumtime  percall filename:lineno(function)
1        0.000    0.000    0.000    0.000 <string>:1(<module>)
69/1     0.000    0.000    0.000    0.000 fibonacci_cprofile.py:18(fib_memo)
1        0.000    0.000    0.000    0.000 {built-in method exec}
1        0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' objects}
```

20.01.2017

B. Nebel – Info I

30 / 42

## Ein „richtiges“ Programm: Der BF-Interpreter



Wir erinnern uns: Das Programm `bf.py` ist ein brainfuck-Interpreter. Es gibt:

- `bf(sourcefn, infn, outf)`:  
Die Hauptfunktion
- `open_files(sfn, infn, outf)`:  
Hilfsfunktion zum Öffnen der Dateien
- `bfinterpret(srctext, fin=sys.stdin, fout=sys.stdout)`: Die Interpreterschleife
- `noop(pc, ptr, src, data, fin, fout)`: Das erste der BF-Kommandos: Keine Operation, d.h. nur Inkrementieren des Programmzählers `pc`
- `left(pc, ptr, src, data, fin, fout)`: Das zweite der BF-Kommandos: Bewege den Datenzeiger `ptr` nach links
- ... und die weiteren Kommandos

Motivation  
Das Modul  
timeit  
Funktions-  
aufrufe  
zählen und  
Memoisie-  
rung  
Das Modul  
cProfile  
Tracing mit  
dem Modul  
trace

20.01.2017

B. Nebel – Info I

31 / 42

## cProfile bei der Arbeit: BF



### Python-Interpreter

```
>>> cProfile.run("bf('prime.b')", sort=1)
```

```
Primes up to: 40
2 3 5 7 11 13 17 19 23 29 31 37
34489714 function calls in 30.598 seconds
```

```
Ordered by: internal time

ncalls  tottime  percall  cumtime  percall filename:lineno(function)
192     11.698    0.061    11.698    0.061 {method 'acquire' of '_thread.lock' objects}
1        8.417    8.417    30.598    30.598 bf.py:54(bfinterpret)
705038   4.117    0.000    4.117    0.000 bf.py:108(endloop)
982715   2.086    0.000    2.554    0.000 bf.py:94(beginloop)
11879063 1.329    0.000    1.329    0.000 {method 'get' of 'dict' objects}
13162690 1.244    0.000    1.244    0.000 {built-in method len}
3086856   0.389    0.000    0.389    0.000 bf.py:77(noop)
726183   0.361    0.000    0.447    0.000 bf.py:86(incr)
726253   0.353    0.000    0.438    0.000 bf.py:90(decr)
1608314   0.299    0.000    0.299    0.000 bf.py:83(right)
1608314   0.288    0.000    0.288    0.000 bf.py:80(left)
96        0.002    0.000    11.702    0.122 rpc.py:303(_getresponse)
...
```

20.01.2017

B. Nebel – Info I

32 / 42



- Wieso wird die Methode `acquire` der `_thread.lock` aufgerufen und verbraucht dann so viel Zeit?
- Kommt durch Konsolen-I/O. Kann ignoriert werden!
- `bfinterpret` verbraucht den Großteil der Zeit! Können wir die Interpreterschleife beschleunigen?
- `beginloop` und `endloop` könnten auch eine Beschleunigung gebrauchen.
- Die interne Methode `len` wird offensichtlich sehr oft aufgerufen.
- Alles andere verbraucht nicht genug Zeit, als dass man sich hier Gedanken machen sollte.

Motivation  
Das Modul `timeit`  
Funktionsaufrufe zählen und Memoisierung  
Das Modul `cProfile`  
Tracing mit dem Modul `trace`

`bf.py`

```
def bfinterpret(srctext, fin=sys.stdin, fout=sys.stdout):  
    pc = 0  
    ptr = 0  
    data = dict();  
    while (pc < len(srctext)):  
        (pc, ptr) = instr.get(srctext[pc],noop)(pc,  
                                ptr, srctext, data, fin, fout)  
  
    pc += 1
```

- Beschleunigungsmöglichkeiten:
  - `len` vorziehen und in lokaler Variable speichern;
  - statt `dict` eine Liste (mit direkter Indizierung) oder `if ...elif ...` einsetzen (kann maximal 1,5 Sekunden bringen!);
  - keine Funktionsaufrufe, sondern direkt den Code hinschreiben (unschön!!)

Motivation  
Das Modul `timeit`  
Funktionsaufrufe zählen und Memoisierung  
Das Modul `cProfile`  
Tracing mit dem Modul `trace`

`bf.py`

```
def beginloop(pc, ptr, src, data, fin, fout):  
    if data.get(ptr,0): return (pc, ptr)  
    loop = 1;  
    while loop > 0:  
        pc += 1  
        if pc >= len(src): raise BFEError()  
        if src[pc] == ']': loop -= 1  
        elif src[pc] == '[': loop += 1  
        return(pc, ptr)  
def endloop(pc, ptr, src, data, fin, fout):  
    ...
```

- Beschleunigungsmöglichkeiten:
  - `len` vorziehen und in lokaler Variable speichern;
  - Loop-Startadresse in einem Stack speichern und für Rücksprung nutzen

Motivation  
Das Modul `timeit`  
Funktionsaufrufe zählen und Memoisierung  
Das Modul `cProfile`  
Tracing mit dem Modul `trace`

- `bfa.py`: `len` aus Schleifen nehmen und Rücksprungadressen bei Schleifen merken: Statt 18 Sekunden, 12 Sekunden!
  - `bfb.py`: Statt `dict` indizierbares Tuple: 11 Sekunden
  - `bfc.py`: Statt Funktionsaufrufe über Tuple eine `elif`-Struktur und den Code direkt eingesetzt: 6 Sekunden!
  - **Fazit**: Programm ist Faktor 3 schneller, ist jetzt aber sehr un schön. Speziell die letzte Modifikation macht das Programm schwerer les- und wartbar!
  - Wenn es *wirklich* um Geschwindigkeit geht, sollte man andere Programmiersprachen, wie C oder C++, einsetzen.
  - Welchen Faktor kann man denn da gewinnen?
- <http://benchmarksgame.alioth.debian.org/>

Motivation  
Das Modul `timeit`  
Funktionsaufrufe zählen und Memoisierung  
Das Modul `cProfile`  
Tracing mit dem Modul `trace`



## 5 Tracing mit dem Modul trace



Motivation  
Das Modul  
timeit  
Funktions-  
aufrufe  
zählen und  
Memoisie-  
rung  
Das Modul  
cProfile  
Tracing mit  
dem Modul  
trace

## Überdeckungsanalyse



Motivation  
Das Modul  
timeit  
Funktions-  
aufrufe  
zählen und  
Memoisie-  
rung  
Das Modul  
cProfile  
Tracing mit  
dem Modul  
trace

- Der Profiler arbeitet nur auf der Ebene von Funktionen, nicht auf der Ebene von Zeilen.
  - Manchmal möchte man wissen, wo denn die meiste Zeit **innerhalb** einer Funktion verbraucht wird.
  - Manchmal möchte man auch wissen, welche Zeilen **nicht** ausgeführt wurden. Das ist wichtig, wenn man alle Zeilen mindestens einmal getestet haben möchte.
- Überdeckungsanalyse mit dem Modul trace
- Hier wird gezählt, **wie oft** eine Zeile ausgeführt wird.

## trace bei der Arbeit



Motivation  
Das Modul  
timeit  
Funktions-  
aufrufe  
zählen und  
Memoisie-  
rung  
Das Modul  
cProfile  
Tracing mit  
dem Modul  
trace

### bftrace.py

```
import sys
import trace
```

```
tracer = trace.Trace(
    ignoredirs=[sys.prefix, sys.exec_prefix],
    trace=0)
```

```
tracer.run("bf('prime.b', 'in10.txt', 'out.txt')")
r = tracer.results()
r.write_results()
```

Schreibt die Datei *modulename.cover* in den Ordner, in dem auch das Modul liegt – für alle Module, die benutzt wurden, außer den ignorierten.

## Das Ergebnis am Beispiel beginloop



Motivation  
Das Modul  
timeit  
Funktions-  
aufrufe  
zählen und  
Memoisie-  
rung  
Das Modul  
cProfile  
Tracing mit  
dem Modul  
trace

### bftrace.cover

```
>>>>> def beginloop(pc, ptr, src, data, fin, fout):
10019:     if data.get(ptr,0): return (pc, ptr)
2956:         loop = 1;
91293:     while loop > 0:
88337:         pc += 1
88337:         if pc >= len(src):
>>>>>             raise BLError("Kein ']'")
88337:         if src[pc] == ']':
4831:             loop -= 1
83506:         elif src[pc] == '[':
1875:             loop += 1
2956:     return(pc, ptr)
```

- In Python kann man die Ausführungszeiten mit den folgenden Funktionen messen:
  - `time.process_time()`: Verbrauchte CPU/Prozess-Zeit inklusive der Zeit im Kernel (vernachlässigbar).
  - `time.perf_counter()`: Wallclock-Zeit in hoher Auflösung, nicht verstellbar!
- Das Modul `timeit` stellt Funktionen und eine Aufrufchnittstelle zur Verfügung, um den Gesamtzeitbedarf zu messen.
- Das Modul `cProfile` hilft, ein Programm auf Funktionsebene zu messen und speziell die Flaschenhälse zu identifizieren.
- Das Modul `trace` misst, wie oft jede Zeile ausgeführt wird.

Motivation

Das Modul  
`timeit`

Funktions-  
aufrufe  
zählen und  
Memoisie-  
rung

Das Modul  
`cProfile`

Tracing mit  
dem Modul  
`trace`