TECHNISCHE UNIVERSITÄT DRESDEN

PYTHONKURS FÜR INGENIEUR:INNEN

Performanzmessung und -optimierung

Folien: C. Knoll, C. Statz, S. Voigt

Dresden, WiSe 2020/21

Inhalt

- Einführung
- Zeitmessung
- Allgemeine Tipps
- Beschleunigung durch C-Extensions

Was ist Performance?

- Laufzeit
- Speicherbedarf (RAM, Festplatte)
- Energie

→ Hier: nur Laufzeit (einfach messbar, oft am wichtigsten)

Fakten

- Python ist langsamer als kompilierte Sprachen (Interpreter)
- Oft aber nicht störend/wahrnehmbar (0.1s vs. 0.01s)
- Laufzeitoptimierung von Code ist selbst oft sehr zeitaufwändig

→ Zielkonflikt Ausführungs- vs. Entwicklungszeit

Was ist Performance?

- Laufzeit
- Speicherbedarf (RAM, Festplatte)
- Energie

→ Hier: nur Laufzeit (einfach messbar, oft am wichtigsten)

Fakten

- Python ist langsamer als kompilierte Sprachen (Interpreter)
- Oft aber nicht störend/wahrnehmbar (0.1s vs. 0.01s)
- Laufzeitoptimierung von Code ist selbst oft sehr zeitaufwändig

→ Zielkonflikt Ausführungs- vs. Entwicklungszeit

⇒ Allgemeine Tipps von Anfang an beachten

⇒ Spezifische Optimierung laufzeitkritischer Algorithmen-Teile

Zeitmessung (I)

- Modul `time`
- `time.time()` gibt „Epoch-Zeit“ (auch „UNIX-timestamp“)
  = die vergangenen Sekunden seit dem 01.01.1970 00:00:00.00
- Vorteil: sehr einfach
- Nachteil: zusätzlicher („boilerplate“)-Code im Programm verteilt

```
import time

s = 0
start = time.time()

for i in range(100000):
    s += i**(0.5)

print("Dauer [s]:", time.time() - start)
```

Zeitmessung (II)

- Modul `timeit`
- Laufzeitmessung eines Statements (meist Funktionsaufruf)
- Gut für Vergleich von Codesnippets für spezielles Problem
- Statement muss als string oder callable übergeben werden
- Vorteile: nicht invasiv, Mittelung mehrerer Durchläufe

Zeitmessung (II)

- Modul `timeit`
- Laufzeitmessung eines Statements (meist Funktionsaufruf)
- Gut für Vergleich von Codesnippets für spezielles Problem
- Statement muss als string oder callable übergeben werden
- Vorteile: nicht invasiv, Mittelung mehrerer Durchläufe

Listing: example-code/time-example.py

```
import timeit
import math

def wurzel_v1():
    return math.sqrt(2)

def wurzel_v2():
    return 2**0.5

print(timeit.timeit("2**0.5",  number=100000))
print(timeit.timeit(wurzel_v1, number=100000))
print(timeit.timeit(wurzel_v2, number=100000))
```

Zeitmessung (II)

- Modul `timeit`
- Laufzeitmessung eines Statements (meist Funktionsaufruf)
- Gut für Vergleich von Codesnippets für spezielles Problem
- Statement muss als string oder callable übergeben werden
- Vorteile: nicht invasiv, Mittelung mehrerer Durchläufe

Listing: example-code/time-example.py

```
import timeit
import math

def wurzel_v1():
    return math.sqrt(2)

def wurzel_v2():
    return 2**0.5

print(timeit.timeit("2**0.5",  number=100000))
print(timeit.timeit(wurzel_v1, number=100000))
print(timeit.timeit(wurzel_v2, number=100000))
```

- Siehe auch: „magische IPython-Makros“: `%timeit` und `%timeit`

Zeitmessung (III): Profiling

- Modul `cProfile`: detaillierte Laufzeitanalyse eines (ggf. sehr umfangreichen) Programms → Flaschenhals finden
- Profiling erzeugt Overhead – Programm läuft etwas langsamer ab als ohne
- Ergebnisse als print-Ausgabe oder in Datei zur Nutzung in Analysewerkzeugen
- Argument wird als String übergeben

```
import cProfile

def main():
    s = 0
    for i in range(100000):
        s += math.sqrt(i)

cProfile.run("main()")
```

Alternativ: Kommandozeilenaufruf:

`python -m cProfile -s cumtime test.py > test.txt`

  - sortiert bzgl. kummulierter Zeit
  - `... > test.txt` bewirkt Umleitung der Ausgabe in Datei: `test.txt`
  - mit Option `-o test.prfl` werden Ergebnisse im Binärformat in Datei `test.prfl` geschrieben (lässt sich dann mit `pstats` auswerten).

Profiling (II)

Ausgabe des Beispiels:

```
100004 function calls in 0.036 seconds

   Ordered by: standard name

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.022    0.022    0.036    0.036 <module1>:34(main)
        1    0.000    0.000    0.036    0.036 <string>:1(<module>)
   100000    0.013    0.000    0.013    0.000 {math.sqrt}
        1    0.000    0.000    0.000    0.000 {method [...]}
        1    0.001    0.001    0.001    0.001 {range}
```

- Zeigt: welche Funktion wie oft aufgerufen wurde und wieviel Zeit dafür benötigt wurde

→ Ansatzpunkte für Optimierung finden

- Hier interessant: nur ca. 1/3 der Laufzeit für `sqrt` benötigt
- Rest ist Overhead (Funktion, Schleife)

Profiling (II)

Ausgabe des Beispiels:

```
100004 function calls in 0.036 seconds

   Ordered by: standard name

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.022    0.022    0.036    0.036 <module1>:34(main)
        1    0.000    0.000    0.036    0.036 <string>:1(<module>)
   100000    0.013    0.000    0.013    0.000 {math.sqrt}
        1    0.000    0.000    0.000    0.000 {method [...]}
        1    0.001    0.001    0.001    0.001 {range}
```

- Zeigt: welche Funktion wie oft aufgerufen wurde und wieviel Zeit dafür benötigt wurde

→ Ansatzpunkte für Optimierung finden

- Hier interessant: nur ca. 1/3 der Laufzeit für `sqrt` benötigt
- Rest ist Overhead (Funktion, Schleife)
- Weitergehende Analyse: Modul `pstats`

Allgemeine Tipps (I)

- Code erst optimieren, wenn tatsächlich Bedarf besteht:
  - „Premature optimisation is the root of all evil.“

→ Profiling nutzen und nur die lohnenswerten Stellen zu identifizieren

- Nur korrekten Code optimieren
- Unit-Tests nutzen, um Richtigkeit des Codes während der Überarbeitung sicherzustellen
  - Reihenfolge: „Make it run. Make it right. Make it fast.“

- Spezielle Bibliotheken für jeweiliges Problem verwenden
- Z. B. `numpy` für Numerik
  - Ist in C/Fortran geschrieben → viel schneller als reines Python

Allgemeine Tipps (II)

- Angemessene Datentypen verwenden: `tuple` oder `dict` statt `list`
- Beispiel: „Element Lookup“

```
3 in {1:True, 2:True, 3:True} # Aufwand O(1)
3 in [1, 2, 3] # Aufwand O(n)
```

- „Punkte“ (Objektorientierung) vermeiden:
  - jeder Punkt bedeutet Attribute/Member Lookup,
  - lokales Zwischenspeichern lohnt sich besonders in Schleifen

```
wurzel = math.sqrt
wurzel(2)
```

- Sog. list comprehension statt `for` -Schleife nutzen

```
r = [ str(k) for k in [1, 2, 3] ]
#  statt
r = []
for k in [1, 2, 3]:
    r.append(str(k))
```

Allgemeine Tipps (III)

- In (verschachtelten) Schleifen: Funktionalität „von innen nach außen“ verlagern
  - Initialisierungen von Variablen
  - Berechnungen → Zwischenergebnisse Speichern/Cachen
  - Allgemein:
    Anweisungen nur so oft wie nötig ausführen, aber so selten wie möglich

- Iteratoren nutzen (z.B. `range(4)` statt `[0, 1, 2, 3]`)
  - Hintergrund: Iteratoren erzeugen Funktion, um nächstes Element zu berechnen,
  - Oft effizienter als komplette Datenstruktur für Iteration im Vorfeld zu generieren

- Lokale Variablen verwenden
  - der Zugriff ist hier schneller, als auf Variable außerhalb des aktuellen Namensraums
  - Funktionen vektorisieren für schnelle Array-Operationen (`numpy.vectorize`)

Kompilierter Code (Übersicht)

Python-Quelltext

- Wird in sog. Bytecode übersetzt und zur Laufzeit vom Interpreter ausgeführt

→ hohe Flexibilität, aber vergleichsweise geringe Ausführungsgeschwindigkeit

Kompilierter Code (Übersicht)

Python-Quelltext

- Wird in sog. Bytecode übersetzt und zur Laufzeit vom Interpreter ausgeführt

→ hohe Flexibilität, aber vergleichsweise geringe Ausführungsgeschwindigkeit

Kompilierter Code

- Wird in Maschinensprache übersetzt und direkt vom Prozessor verarbeitet

→ hohe Ausführungsgeschwindigkeit, geringe Flexibilität (z.B. statische Datentypen, Speichermanagement)

Kompilierter Code (Übersicht)

Python-Quelltext

- Wird in sog. Bytecode übersetzt und zur Laufzeit vom Interpreter ausgeführt

→ hohe Flexibilität, aber vergleichsweise geringe Ausführungsgeschwindigkeit

Kompilierter Code

- Wird in Maschinensprache übersetzt und direkt vom Prozessor verarbeitet

→ hohe Ausführungsgeschwindigkeit, geringe Flexibilität (z.B. statische Datentypen, Speichermanagement)

Kombinationsmöglichkeiten (Einbettung von komp. Code in Python):

- `ctypes`
  - Kann externe Bibliotheken (z.B. *.dll unter Windows) in Python laden

  → sehr mächtig und flexibel
  - Hier nicht näher betrachtet, siehe ggf. `https://github.com/cknoll/python-c-code-example`

Kompilierter Code (Übersicht)

Python-Quelltext

- Wird in sog. Bytecode übersetzt und zur Laufzeit vom Interpreter ausgeführt

→ hohe Flexibilität, aber vergleichsweise geringe Ausführungsgeschwindigkeit

Kompilierter Code

- Wird in Maschinensprache übersetzt und direkt vom Prozessor verarbeitet

→ hohe Ausführungsgeschwindigkeit, geringe Flexibilität (z.B. statische Datentypen, Speichermanagement)

Kombinationsmöglichkeiten (Einbettung von komp. Code in Python):

- `ctypes`
  - Kann externe Bibliotheken (z.B. *.dll unter Windows) in Python laden

  → sehr mächtig und flexibel
  - Hier nicht näher betrachtet, siehe ggf. `https://github.com/cknoll/python-c-code-example`
- „Just in Time“-Kompilierung von bestimmten Code-Abschnitten (z.B. Modul `numba`)

- Übersetzen des Python Codes in `cython`
  - Sehr ähnlich zu Python aber statisch typisiert und kompiliert

Just-in-time-Kompilierung mit `numba`

- Erhebliches Beschleunigungspotential bei mathematischen Operationen
- Notwendig: `pip install numba`
- Beispiel: „Mandelbort Menge“
  - (einfache Mathematik, hoher numerischer Aufwand, visuelles Ergebnis)

Listing: example-code/numba1.py (14-29)

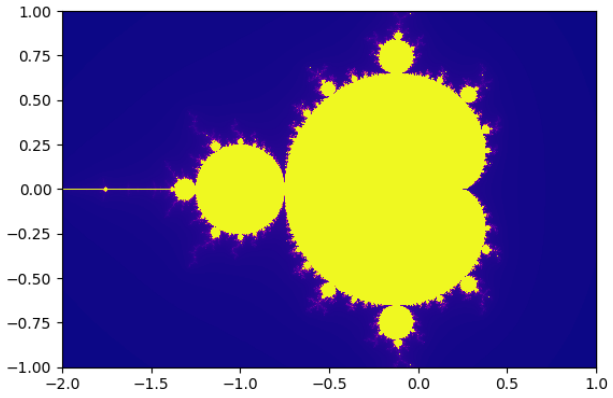
```
@jit  # Decorator für just-in-time Kompilierung
def mandel(x, y, max_iters):
    """
    Given a complex number x + y*j, determine
    if it is part of the Mandelbrot set given
    a fixed number of iterations.
    """
    i = 0
    c = complex(x, y)
    z = 0.0j
    for i in range(max_iters):
        z = z*z + c
        if (z.real*z.real + z.imag*z.imag) >= 1e3:
            return i

    return 255
```

Just-in-time-Kompilierung mit `numba`

- Erhebliches Beschleunigungspotential bei mathematischen Operationen
- Notwendig: `pip install numba`
- Beispiel: „Mandelbort Menge“
  - (einfache Mathematik, hoher numerischer Aufwand, visuelles Ergebnis)

Listing: example-code/numba1.py (14-29)

```
@jit  # Decorator für just-in-time Kompilierung
def mandel(x, y, max_iters):
    """
    Given a complex number x + y*j, determine
    if it is part of the Mandelbrot set given
    a fixed number of iterations.
    """
    i = 0
    c = complex(x, y)
    z = 0.0j
    for i in range(max_iters):
        z = z*z + c
        if (z.real*z.real + z.imag*z.imag) >= 1e3:
            return i

    return 255
```



cython (I)

- Cython ist eigene Programmiersprache, Installation: `pip install cython`
- Sehr eng an Python angelehnt aber mit expliziten statischen Typ-Informationen

→ Kann automatisch nach C übersetzt werden → kompilierbar → schneller

- Details: siehe Doku

cython (I)

- Cython ist eigene Programmiersprache, Installation: `pip install cython`
- Sehr eng an Python angelehnt aber mit expliziten statischen Typ-Informationen

→ Kann automatisch nach C übersetzt werden → kompilierbar → schneller

- Details: siehe Doku

- Vorgehen:
  - Algorithmus in reinem Python entwickeln („Make it run“ + „Make it right“)
  - Python manuell nach Cython übersetzen
  - Cython-Code nach C übersetzen lassen
  - C-Code kompilieren
  - So erstelltes Modul „ganz normal“ importieren / benutzen (→ „Make it fast“)

cython (I)

- Cython ist eigene Programmiersprache, Installation: `pip install cython`
- Sehr eng an Python angelehnt aber mit expliziten statischen Typ-Informationen

→ Kann automatisch nach C übersetzt werden → kompilierbar → schneller

- Details: siehe Doku

- Vorgehen:
  - Algorithmus in reinem Python entwickeln („Make it run“ + „Make it right“)
  - Python manuell nach Cython übersetzen
  - Cython-Code nach C übersetzen lassen
  - C-Code kompilieren
  - So erstelltes Modul „ganz normal“ importieren / benutzen (→ „Make it fast“)

  Typischerweise 3 Dateien, z.B.
  - `mandel-cython.pyx` : Cython-Quelltext
  - `mandel-cython-setup.py` : Zum Kompilieren
  - `mandel-cython-main.py` : Zum Importieren u. Aufrufen

Listing: example-code/mandel-cython.pyx

```
# Cython-Quelltext
cimport numpy as np # for the special numpy stuff

cdef inline int mandel(double real, double imag, int max_iterations=20):
    """Given a complex number x + y*j, determine if it is part of the
    Mandelbrot set given a fixed number of iterations. """

    cdef double z_real = 0., z_imag = 0.
    cdef int i

    for i in range(0, max_iterations):
        z_real, z_imag = ( z_real*z_real - z_imag*z_imag + real,
                           2*z_real*z_imag + imag )
        if (z_real*z_real + z_imag*z_imag) >= 1000:
            return i
    # return -1
    return 255

def create_fractal( double min_x, double max_x, double min_y, int nb_iterations,
                            np.ndarray[np.uint8_t, ndim=2, mode="c"] image not None):

    cdef int width, height, x, y, start_y, end_y
    cdef double real, imag, pixel_size

    width = image.shape[0]
    height = image.shape[1]

    pixel_size = (max_x - min_x) / width

    for x in range(width):
        real = min_x + x*pixel_size
        for y in range(height):
            imag = min_y + y*pixel_size
            image[x, y] = mandel(real, imag, nb_iterations)
```

cython (III)

- Skript zur Konvertierung in C-Code:

Listing: example-code/mandel-cython-setup.py

```
#!/usr/bin/env python

"""
setup.py  to build mandelbot code with cython
"""
from distutils.core import setup
from distutils.extension import Extension
from Cython.Distutils import build_ext
import numpy # to get includes


setup(
    cmdclass = {'build_ext': build_ext},
    ext_modules = [Extension("mandelcy", ["mandel-cython.pyx"], )],
    include_dirs = [numpy.get_include(),],
)
```

- Aufruf: `python mandel-cython-setup.py build_ext --inplace`

→ C-Code wird kompiliert und eine importierbare Bibliothek erstellt

cython (IV)

- Aufruf des Kompilierten Codes und Visulaisierung:

Listing: example-code/mandel-cython-setup.py

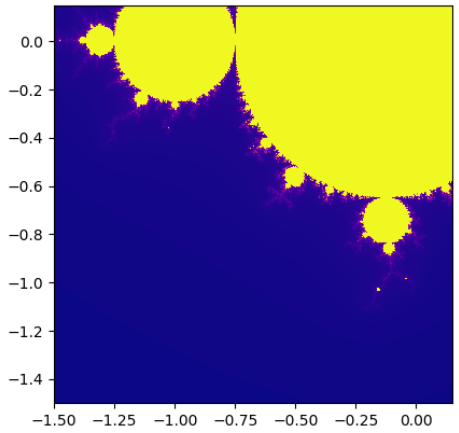
```
import numpy as np
import matplotlib.pyplot as plt
import mandelcy # unser Cython Modul (macht die eigentliche Arbeit)


# Ausschnitt der Gausschen Zahlenebene festlegen
min_x = -1.5
max_x =  0.15
min_y = -1.5
max_y = min_y + max_x - min_x

# to have same section like numba script
# min_x = -2; max_x =  1; min_y = -1.5

nb_iterations = 255

dataarray = np.zeros((500, 500), dtype=np.uint8)

# Ausführung des Kompilierten Codes
mandelcy.create_fractal(min_x, max_x, min_y, nb_iterations, dataarray)

# Transponieren und erste Achse spiegeln
dataarray = dataarray.T[::-1, :]

plt.imshow(dataarray, extent=(min_x, max_x, min_y, max_x), cmap=plt.cm.plasma)
plt.savefig("mandel-cython.png")
plt.show()
```

cython (IV)

- Aufruf des Kompilierten Codes und Visulaisierung:

Listing: example-code/mandel-cython-setup.py

```
import numpy as np
import matplotlib.pyplot as plt
import mandelcy # unser Cython Modul (macht die eigentliche Arbeit)


# Ausschnitt der Gausschen Zahlenebene festlegen
min_x = -1.5
max_x =  0.15
min_y = -1.5
max_y = min_y + max_x - min_x

# to have same section like numba script
# min_x = -2; max_x =  1; min_y = -1.5

nb_iterations = 255

dataarray = np.zeros((500, 500), dtype=np.uint8)

# Ausführung des Kompilierten Codes
mandelcy.create_fractal(min_x, max_x, min_y, nb_iterations, dataarray)

# Transponieren und erste Achse spiegeln
dataarray = dataarray.T[::-1, :]

plt.imshow(dataarray, extent=(min_x, max_x, min_y, max_x), cmap=plt.cm.plasma)
plt.savefig("mandel-cython.png")
plt.show()
```



Zusammenfassung

- Viele Möglichkeiten, Python selber schneller zu machen
- Wenn das nicht reicht:
  - Flaschenhälse mittels Profiling identifizieren
- Kritische Programmteile durch kompilierten Code ersetzen
  - Just-in-Time-Kompilierung `numba` (Aufwand: gering)
  - Manuelle Portierung nach `cython` (Aufwand: mäßig)
  - Eigener C-Code mit `ctypes` (Aufwand ggf. erheblich)

Zusammenfassung

- Viele Möglichkeiten, Python selber schneller zu machen
- Wenn das nicht reicht:
  - Flaschenhälse mittels Profiling identifizieren
- Kritische Programmteile durch kompilierten Code ersetzen
  - Just-in-Time-Kompilierung `numba` (Aufwand: gering)
  - Manuelle Portierung nach `cython` (Aufwand: mäßig)
  - Eigener C-Code mit `ctypes` (Aufwand ggf. erheblich)
  - ($\exists$ weitere Möglichkeiten)
- Hier nicht behandelt: `Threading/Multiprocessing`