TECHNISCHE UNIVERSITÄT DRESDEN

python™

Carsten Knoll

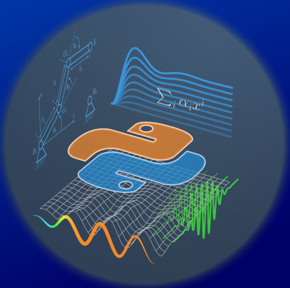
Chair of Fundamentals of Electrical Engineering

Python for Engineers
Pythonkurs für Ingenieur:innen

Performance Optimization
Performanzoptimierung

Dresden (Online), 2023-12-05

`https://tu-dresden.de/pythonkurs`
`https://python-fuer-ingenieure.de`



Outline

- Introduction
- Timing
- General Tips
- Compiled Code

Introduction

What is performance?

- runtime
- memory requirements (RAM, hard disk)
- power consumption

→ In this lecture: only **execution time** considered
(most important in most cases, easy to measure, correlated with power consumption)

Facts

- Python: slower than compiled languages (interpreters)
- in many applications: difference not even perceptible (0.1s vs. 0.01s)
- runtime optimization of code itself often very time consuming

→ conflict of goals: execution vs. development time

Introduction

What is performance?

- runtime
- memory requirements (RAM, hard disk)
- power consumption

$\rightarrow$ In this lecture: only **execution time** considered
(most important in most cases, easy to measure, correlated with power consumption)

Facts

- Python: slower than compiled languages (interpreters)
- in many applications: difference not even perceptible (0.1s vs. 0.01s)
- runtime optimization of code itself often very time consuming

$\rightarrow$ conflict of goals: execution vs. development time

$\Rightarrow$ general tips to follow from the start

$\Rightarrow$ specific optimization of runtime for **critical algorithm parts**

Time Measurement (I)

- module `time`
- `time.time()` returns “epoch-time” (also called “UNIX-timestamp”)
  `time` ≙ seconds since 01/01/1970 00:00:00.00
- advantage: very simple
- disadvantage: additional (“boilerplate”) code distributed in the program

```python
import time

s = 0
start = time.time()

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

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

Time Measurement (II)

- module `timeit`, see docs
- runtime measurement of a statement (mostly function call)
- good for comparison of code snippets for special problem
- statement must be passed as *string* or *callable*
- advantages: non-invasive, averaging of multiple runs

Time Measurement (II)

- module `timeit`, see docs
- runtime measurement of a statement (mostly function call)
- good for comparison of code snippets for special problem
- statement must be passed as *string* or *callable*
- advantages: non-invasive, averaging of multiple runs

Listing: example-code/time-example.py

```
import math; from timeit import timeit

def root1(x=2): return x**0.5

def root2(): return math.sqrt(2)

N = int(1e6)
print("2**0.5:    ", timeit("2**0.5", number=N), "sqrt(2):", timeit("math.sqrt(2)", setup="import math", number=N))

# func calls without argument
print("root1():   ", timeit(root1, number=N), "root2():", timeit(root2, number=N))

# func calls with argument (timeit(root2(x=2), number=N))) would "see" only the return value
# Thus, wee need to pass the function call as string. Then the `globals`- keyword argument is also needed.
print("root1(x=2):", timeit("root1(x=2)", number=N, globals=globals()))
```

Time Measurement (II)

- module `timeit`, see docs
- runtime measurement of a statement (mostly function call)
- good for comparison of code snippets for special problem
- statement must be passed as *string* or *callable*
- advantages: non-invasive, averaging of multiple runs

Listing: example-code/time-example.py

```
import math; from timeit import timeit

def root1(x=2): return x**0.5

def root2(): return math.sqrt(2)

N = int(1e6)
print("2**0.5:    ", timeit("2**0.5", number=N), "sqrt(2):", timeit("math.sqrt(2)", setup="import math", number=N))

# func calls without argument
print("root1():   ", timeit(root1, number=N), "root2():", timeit(root2, number=N))

# func calls with argument (timeit(root2(x=2), number=N))) would "see" only the return value
# Thus, wee need to pass the function call as string. Then the `globals`- keyword argument is also needed.
print("root1(x=2):", timeit("root1(x=2)", number=N, globals=globals()))
```

- See also: “magic macros” for both IPython and Jupyter: `%time` and `%timeit`

Professional Timing: Profiling (I)

- module `cProfile`: detailed runtime analysis of a (possibly very large) program $\rightarrow$ Find bottleneck.
- profiling creates overhead, i. e. program runs slightly slower than without it
- results as print output or to file for use in analysis tools
- argument is passed as string

Listing: example-code/profile-example.py

```python
import cProfile
import math

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

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

alternative (command line call):

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

- sorted by cumulative time
- `... > test.txt` redirects output to file: `test.txt`
- with option `-o test.prfl` will redirect results in binary format to file `test.prfl` (can then be evaluated with `pstats`, see docs).

Profiling (II)

Output of the example

```
100004 function calls in 0.021 seconds

Ordered by: standard name

ncalls  tottime  percall  cumtime  percall filename:lineno(function)
     1    0.000    0.000    0.021    0.021 <string>:1(<module>)
     1    0.014    0.014    0.021    0.021 profile-example.py:4(main)
     1    0.000    0.000    0.021    0.021 {built-in method builtins.exec}
100000    0.006    0.000    0.006    0.000 {built-in method math.sqrt}
     1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' objects}
```

- shows which function was called how often and how much time it needed

$\rightarrow$ find starting points for optimization

- interesting here: only $\approx \frac{1}{3}$ of the runtime for `sqrt` needed
- rest: overhead (function call, loop)

Profiling (II)

Output of the example

```
100004 function calls in 0.021 seconds

Ordered by: standard name

ncalls  tottime  percall  cumtime  percall filename:lineno(function)
     1    0.000    0.000    0.021    0.021 <string>:1(<module>)
     1    0.014    0.014    0.021    0.021 profile-example.py:4(main)
     1    0.000    0.000    0.021    0.021 {built-in method builtins.exec}
100000    0.006    0.000    0.006    0.000 {built-in method math.sqrt}
     1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' objects}
```

- shows which function was called how often and how much time it needed

$\rightarrow$ find starting points for optimization

- interesting here: only $\approx \frac{1}{3}$ of the runtime for `sqrt` needed
- rest: overhead (function call, loop)
- Further analysis: `pstats`, see docs

General Tips (I)

- optimize code only when there is an actual need
  (“Premature optimisation is the root of all evil.”)

$\rightarrow$ use profiling and identify only the worthwhile jobs.

- optimize only correct code
- use unit tests to ensure correctness of the code during/after rework
  - order: „Make it run. Make it right. Make it fast.“

- use appropriate libraries for respective problem
- e.g. `numpy` for numerics
  - is written in C/Fortran $\rightarrow$ much faster than pure Python
- Python scripts usually faster than Jupyter Notebooks (rendering overhead)

General Tips (II)

- use appropriate data types: `tuple` or `dict` instead of `list`.
  example: “element Lookup”

```
res = 3 in {1: True, 2: True, 3: True}   # effort: O(1)  (= const)
res = 3 in [1, 2, 3] # effort O(n)
```

- in (nested) loops: move functionality “from inside to outside”.
  - initializations of variables
  - calculations $\rightarrow$ intermediate results save/cache

  $\rightarrow$ **execute statements only as often as necessary, but as rarely as possible**
  - “loops in functions” are faster than “functions in loops”
    (every function-call costs time)
- create auxiliary **local variables** to avoid “points” (e. g. from object orientation):
  - each point (`obj.attr`) means attributes/member lookup,
  - local caching is worthwhile especially in loops

```
root = math.sqrt
# ...
root(2)  # inside a loop avoid name-lookup
```

Outdated Tips (III)

Often recommended but not so effective (anymore) w.r.t speedup

- use iterators (e.g. `range(4)` instead of `[0, 1, 2, 3]`)
  - background: iterators generate function to calculate next element,
  - more memory-efficient than generating whole sequence in advance

- use list comprehension instead of `for`-loops

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

- vectorize functions for array operations (`numpy.vectorize`), see docs

```
def f(x):
    if x > 2: return x*100
    else:     return x

xx = np.arange(5)
# f(xx)   # -> ValueError
f_vect = np.vectorize(f)
f_vect(xx)   # -> array([  0,    1,    2, 300, 400])
```

Compiled Code (Overview)

Python source code:

- Compiled into “bytecode” and executed by the Python *interpreter* at runtime.

$\rightarrow$ high flexibility, but comparatively low execution speed

Compiled Code (Overview)

Python source code:

- Compiled into “bytecode” and executed by the Python *interpreter* at runtime.

$\rightarrow$ high flexibility, but comparatively low execution speed

Compiled code (C, C++, Rust, ...):

- Compiled into machine language and processed directly by the processor

$\rightarrow$ high execution speed, low flexibility (e.g. static data types, memory management)

Compiled Code (Overview)

Python source code:

- Compiled into “bytecode” and executed by the Python *interpreter* at runtime.

$\rightarrow$ high flexibility, but comparatively low execution speed

Compiled code (C, C++, Rust, ...):

- Compiled into machine language and processed directly by the processor

$\rightarrow$ high execution speed, low flexibility (e.g. static data types, memory management)

Possible combinations (embedding compiled code in Python):

- “Just in Time” compilation of certain code sections (e.g. module `numba`)

Compiled Code (Overview)

Python source code:

- Compiled into "bytecode" and executed by the Python *interpreter* at runtime.

$\rightarrow$ high flexibility, but comparatively low execution speed

Compiled code (C, C++, Rust, ...):

- Compiled into machine language and processed directly by the processor

$\rightarrow$ high execution speed, low flexibility (e.g. static data types, memory management)

Possible combinations (embedding compiled code in Python):

- "Just in Time" compilation of certain code sections (e.g. module `numba`)
- compile Python code into `cython`
  - very similar to Python but statically typed and compiled $\rightarrow$ fast

Compiled Code (Overview)

Python source code:

- Compiled into "bytecode" and executed by the Python *interpreter* at runtime.

→ high flexibility, but comparatively low execution speed

Compiled code (C, C++, Rust, ...):

- Compiled into machine language and processed directly by the processor

→ high execution speed, low flexibility (e.g. static data types, memory management)

Possible combinations (embedding compiled code in Python):

- "Just in Time" compilation of certain code sections (e.g. module `numba`)
- compile Python code into `cython`
  – very similar to Python but statically typed and compiled → fast
- `ctypes`
  – Can load external libraries into Python (e.g. *.dll on Windows, *.so on Unix)
  → very powerful and flexible
  – Not considered here; if necessary see python-c-code-example (github)
  – mostly useful if C-library already exists (or is needed anyway, e.g. for target hardware)

Just-in-time-Compilation with `numba`

- Significant acceleration potential for mathematical operations.
- Necessary: `pip install numba`
- Example: "Mandelbrot set"
  - (simple math, high numerical effort, visual result).

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

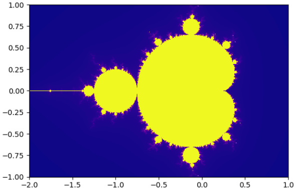
```
# Decorator for just-in-time comp. (-> 30x speedup)
@jit
def mandel(x, y, max_iters):
    """
    Given a complex number x + y*j, determine
    if it is part of the Mandelbrot set given
    a fixed number of iterations.
    """
    i = 0
    c = complex(x, y)
    z = 0.0j
    for i in range(max_iters):
        z = z*z + c
        if (z.real*z.real + z.imag*z.imag) >= 1e3:
            return i
```

Just-in-time-Compilation with `numba`

- Significant acceleration potential for mathematical operations.
- Necessary: `pip install numba`
- Example: "'Mandelbrot set'"
  - (simple math, high numerical effort, visual result).

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

```
# Decorator for just-in-time comp. (-> 30x speedup)
@jit
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
```



`Cython` (I)

- Cython is a separate programming language, installation: `pip install cython`
- Very close to Python but with explicit **static type** information

$\rightarrow$ can be compiled to C automatically $\rightarrow$ compilable $\rightarrow$ faster

- details: see docs

Cython (I)

- Cython is a separate programming language, installation:
  `pip install cython`
- Very close to Python but with explicit **static type** information

$\rightarrow$ can be compiled to C automatically $\rightarrow$ compilable $\rightarrow$ faster

- details: see docs
- procedure:
  - Develop algorithm in pure Python (“Make it run” + “Make it right”)
  - Translate Python to Cython manually
  - Translate Cython code to C
  - Compile C code
  - Import / use module created this way “as usual” ($\rightarrow$ “Make it fast”)

Cython (I)

- Cython is a separate programming language, installation:
  `pip install cython`
- Very close to Python but with explicit **static type** information

$\rightarrow$ can be compiled to C automatically $\rightarrow$ compilable $\rightarrow$ faster

- details: see docs
- procedure:
  - Develop algorithm in pure Python (“Make it run” + “Make it right”)
  - Translate Python to Cython manually
  - Translate Cython code to C
  - Compile C code
  - Import / use module created this way “as usual” ($\rightarrow$ “Make it fast”)

Typically 3 files, e. g.

- `mandel-cython.pyx`: cython source code
- `mandel-cython-setup.py`: for compiling
- `mandel-cython-main.py`: to import and call compiled code

Listing: example-code/mandel-cython.pyx

```
# Cython source code
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)

- script for conversion Cython $\rightarrow$ C:

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

```
"script for conversion of cython-code to c-code"

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(),],
)
```

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

$\rightarrow$ C code is compiled and an importable library is created

Cython (IV)

- Calling the compiled code and visualization of Mandelbrot set:

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

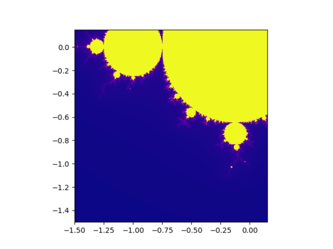
```
import numpy as np
import matplotlib.pyplot as plt
import mandelcy  # our Cython module (for the real work)

import time

# define section of the Gaussian number plane
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

t1 = time.time()
dataarray = np.zeros((500, 500), dtype=np.uint8)
t2 = time.time()
print("Time needed", t2 - t1)

# execution of the compiled code
mandelcy.create_fractal(min_x, max_x, min_y, nb_iterations, dataarray)

dataarray = dataarray.T[::-1, :]  # Transpose and reverse order along first axis

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)

- Calling the compiled code and visualization of Mandelbrot set:

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

```
import numpy as np
import matplotlib.pyplot as plt
import mandelcy  # our Cython module (for the real work)

import time

# define section of the Gaussian number plane
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

t1 = time.time()
dataarray = np.zeros((500, 500), dtype=np.uint8)
t2 = time.time()
print("Time needed", t2 - t1)

# execution of the compiled code
mandelcy.create_fractal(min_x, max_x, min_y, nb_iterations, dataarray)

dataarray = dataarray.T[::-1, :]  # Transpose and reverse order along first axis

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



(⟶ 500x speedup (Py vs Cy))

Summary

- $\exists$ many ways to tweak Python code to make it faster
- If that is not enough:
  - identify bottle necks using profiling
- replace critical program parts with compiled code
  - just-in-time compilation `numba` (effort: low)
  - manual port to `cython` (effort: moderate)
  - custom C code using `ctypes` (effort might be considerable)

Summary

- ∃ many ways to tweak Python code to make it faster
- If that is not enough:
  - identify bottle necks using profiling
- replace critical program parts with compiled code
  - just-in-time compilation `numba` (effort: low)
  - manual port to `cython` (effort: moderate)
  - custom C code using `ctypes` (effort might be considerable)

  - (∃ more possibilities, e.g. PyPy)

- not covered here:
  - `threading/multiprocessing`, parallelization via asyncio
  - pyjion (“drop-in JIT Compiler for Python 3.10”)