technische universität
dortmund

Python Grundlagen
Profiling
Tests

Marcel Bargull

2. April 2014

Python Grundlagen

- Was ist Python?
- Funktionen & Variablen
- Kontrollfluss
- Datentypen und -strukturen
- Klassen
- Exceptions
- Module und Pakete
- Standardbibliothek & PyPi

Python

- So wie C++ auch imperativ und objektorientiert
- Aber dennoch sehr anders:
  - Dynamische Typisierung
  - Reduzierte Syntax
  - I.d.R. als Skriptsprache benutzt
  - Auch funktionale Aspekte
  - Nicht sehr hardwarenah
- Designziele von Python: **Lesbarkeit, Einfachheit**

Hello World!

```
print("Hello World!")
```

Erweitertes Beispiel

```
import sys

def say_hello(greeting="Hello World!"):
    print(greeting)

if __name__ == "__main__":
    for arg in sys.argv:
        say_hello(arg)
```

Funktionen & Variablen

- Funktionen

```
def func(a, b, c=3): return a + b + c
```

  - Aufruf:

```
func(1, 2)
func(1, b=2)
func(1, 2, 3)
func(b=2, a=1, c=3)
func(a=1, 2, c=3)
```

  - Immer Rückgabewert; implizit: **None** (Java: **null**)

- Variablen
  - Keine vorherige Deklaration
  - Dynamisch getypt

```
x = None
x = ""
x, y = 1, 2
```

Kontrollfluss

- Codeblöcke durch Einrückung und `:` statt `{ … }`
- Keine Klammern bei **`if`**, **`for`**, **`while`**, etc.
- **`elif`** für **`else if`** (**`switch-case`** existiert nicht)
- Natürlichsprachliche Boolesche Operatoren
- Mehrfachvergleiche möglich

```
def print_number(n):
    if not n >= 0:
        print("negative!")
    elif 0 <= n < 2 and n % 2 == 1:
        print("one!")
    else:
        print("zero or at least two")
```

Kontrollfluss: Schleifen

- **while**-Schleifen wie in C++ (aber ohne **do … while**)
- **for**-Schleifen immer mittels Iterator

```
for i in range(5):
    print(i, end=" ") # 0 1 2 3 4
for el in ["a", "b", "c"]:
    print(el, end=" ") # a b c
for i, el in enumerate(["a", "b", "c"]):
    print((i, el), end=" ") # (0, a) (1, b) (2, c)
```

- Erweiterte **else**-Syntax

```
for el in ["a", "b", "c"]:
    if el == "x":
        break
else:
    print("Could not find x")
```

Datentypen und -strukturen

- **bool**: `True`, `False`
- Zahlen: **`int`**, **`float`**, **`complex`**, fraction, decimal
- Sequenztypen
  - **`list`**
    - veränderlich
    - heterogene Elemente
  - **`tuple`**
    - unveränderlich
    - heterogene Elemente
  - **`range`**
    - Ganzzahlige Intervalle

```
list()
[]
[1]
[1, "a", None]
```

```
tuple()
(1,)
(1, "a", None)
```

```
range(3) # (0, 1, 2)
range(2, 5) # (2, 3, 4)
range(0,-3,-1) # (0,-1,-2)
range(1, 0)  # ()
```

Datentypen und -strukturen

- Sequenztypen (cont.)
  - **`str`** (Python 3)
    - unveränderlich
    - `""` und `''` gleichwertig
    - rein unicode
  - Sequenzen auf Binärdaten (Python 3):

    (z.B. für kodierte Zeichenketten)
    - **`bytes`**
      - unveränderlich
    - **`bytearray`**
      - veränderlich

```
"abc\n" # 'abc\n'
'abc\n' # 'abc\n'
r'abc\n' # 'abc\\n'
```

```
b"abc"
bytes((97,98,99))
bytearray(b"abc")
bytearray((97,98,99))
```

Datentypen und -strukturen

- Operationen auf Sequenzen
  - Existenztest:

```
1 in [0,1,2,3] # True
"ab" in "abc" # True
```

  - Verkettung:

```
[0,1] + [2,3] # [0,1,2,3]
```

  - Vervielfältigung:

```
3 * "abc" # 'abcabcabc'
```

  - Slicing:

```
L = [0,1,2,3,4,5,6,7,8]
L[1:7:2] # [1,3,5]
L[:2] = (4,5,6) # [4,5,6,2,3]
```

  - Und viele Weitere:

```
len, min, max, index, count,
append, extend, insert,
remove, reverse, ...
```

Datentypen und -strukturen

- Menge: **`set`**

```
set() # set()
{1, 2, 1} # {1,2}
```

- Wörterbuch: **`dict`**

```
{} # {}
dict() # {}
d = {"a": 1, "b": 2}
d = dict(a=1, b=2)
d = dict(("a", 1), ("b", 2))
d.keys() # ['a', 'b']
d.values() # [1, 2]
d.items() # [('a', 1), ('b', 2)]
d["a"] = 0 # {'a': 0, 'b': 2}
d[0] = 3 # {'a': 0, 'b': 2, 0: 3}
```

Datentypen und -strukturen

- Weitere Datentypen: **`object`**, **`function`**, **`class`**, etc.
- Alles ist ein Objekt!
- Selbst Integer haben Attribute

```
def f():
    pass
type(f) # <class 'function'>
type({}) # <class 'dict'>
type(dict) # <class 'type'>
type(1) # <class 'int'>
dir(dict) # ...
dir(1) # ...
```

List Comprehensions

```
[x**2 for x in [3,3,4]] # [9, 9, 16]
tuple(x**2 for x in [3,3,4]) # (9, 9, 16)
{x:x**2 for x in [3,3,4]} # {3: 9, 4: 16}
{x**2 for x in [3,3,4]} # {16, 9}
```

```
list(map(len, ["a", "abc", "ab"])) # [1, 3, 2]
[len(x) for x in ["a", "abc", "ab"]] # [1, 3, 2]
```

```
[(x, x+1) for x in range(8) if x % 2 == 0]
# [(0, 1), (2, 3), (4, 5), (6, 7)]
```

```
m = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
[[row[i] for row in m] for i in range(len(m))]
# [[1, 4, 7], [2, 5, 8], [3, 6, 9]]
list(zip(*m))
# [[1, 4, 7], [2, 5, 8], [3, 6, 9]]
```

Klassen

```
class BaseA():

    def __init__(self, a):
        self.a = a

    def plus1(self, x):
        return x + 1
```

```
class Class(BaseA, BaseB):
    pass
```

```
class BaseB():

    var = "a"

    @staticmethod
    def static_pow2(x):
        return x ** 2

    @classmethod
    def get_var(cls):
        return cls.var

    def __add__(self, x):
        return self.a + x
```

Klassen

```
obj1 = Class(10)

obj1.plus1(1) # 2

Class.static_pow2(4) # 16

Class.get_var() # 'a'

obj1.get_var() # 'a'

obj1 + 1 # 11
```

```
obj1.var # 'a'

Class.var # 'a'

Class.var = "b"

obj1.var # 'b'

obj1.var = "c"

Class.var # 'b'

obj1.var # 'c'

BaseB.var # 'a'
```

Exceptions

- Ähnlich zu C++ / Java
- Häufig benutzt in der Standardbibliothek
- Exceptions sind Klassen
- Spezielle NULL- / Errorcode-Rückgaben (wie in C) mit Exceptions vermeiden

```
def forgiving_division(x, y):
  try:
    quotient = x / y
  except ZeroDivisionError:
    if x > 0:
      quotient = float("Inf")
    elif x < 0:
      quotient = -float("Inf")
    else:
      quotient = 0
  except TypeError:
    quotient = float("NaN")
  except:
    raise
  else:
    pass # no exception
  finally:
    pass # always executed
```

Module & Pakete

- Strukturierung durch Module & Pakete
- Paket: Ordner mit **__init__.py**
- Module: **.py**-Dateien

```
package/
   __init__.py
   module1.py
```

```
import package.module1
from package import module1
from package.module1 import some_func
from package import module1 as m1

package.module1.some_func()
module1.some_func()
some_func()
m1.some_func()

if __name__ == "__main__":
    print("executed as script...")
```

Standardbibliothek & PyPi

- **`dir`**`([object])` listet Attribute von Objekten auf
- **`help`**`([object])` gibt Hilfe in interaktiven Sitzungen
  - docstrings:

```
def sqr(x):
    """Return the square of x."""
    return x**2
```

- Module:
  - **`io`**: Dateioperationen
  - **`math`**: Mathematische Funktionen
  - **`itertools`**: **`chain`**, **`combinations`**, **`permutations`**, …
  - **`re`**: Reguläre Ausdrücke
  - **`unittest`**, **`timeit`**, **`cProfile`**, …
- Python Package Index (PyPi): pypi.python.org/pypi

Profiling

- Was ist Profiling?
- Profiling in Python
- Geschwindigkeit von Python

Profiling

- Analyse des Laufzeitverhaltens
- Profiler überwacht Programmausführung
- Aufrufanzahl & (kumulierte) Laufzeiten aller Funktionen
- Gibt Auskunft über Laufzeitverteilung
- => Optimierungsmöglichkeiten eingrenzen
- Auch Profiler für Speicherausnutzung vorhanden

Profiling in Python

- Zeitmessung generell:
  - **timeit**
- Profiler in Standard Library:
  - **cProfile**, **profile**
- Weitere Profiler:
  - line_profiler

```
from time import sleep
def func():
    for x in range(1000):
        sub1()
def sub1():
    sleep(0.001)
    for x in range(100):
        sub2()
def sub2():
    sleep(0.0001)
```

```
timeit.timeit("func()", "from __main__ import func", number=1)
# 1.2006868506823012

cProfile.run("func()")
```

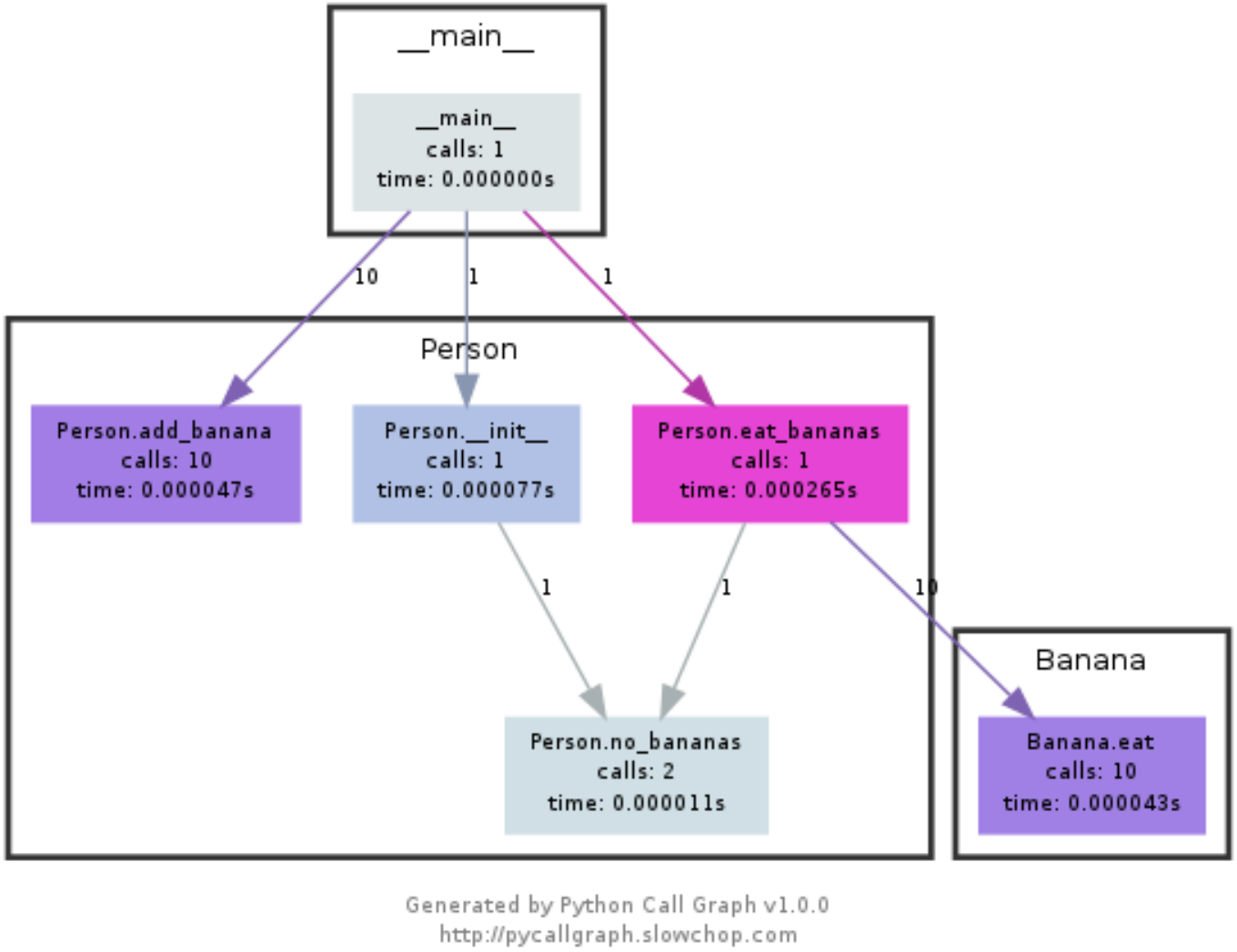
Profiling in Python

cProfile.run Ausgabe:

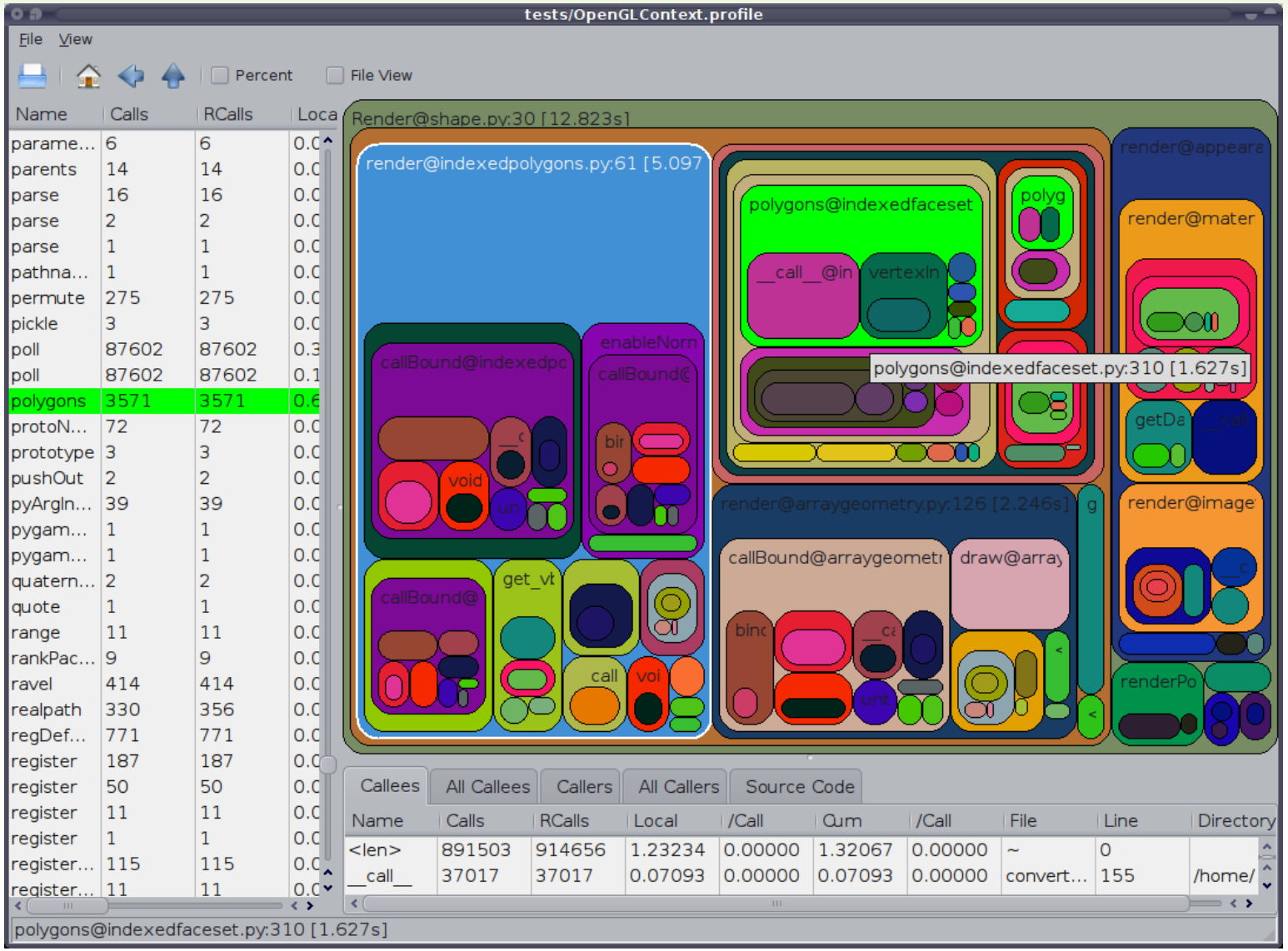
```
Ordered by: standard name

ncalls tottime percall cumtime percall filename:lineno(function)
     1   0.003   0.003   1.295   1.295 testing.py:5(func)
  1000   0.311   0.000   1.292   0.001 testing.py:8(sub1)
100000   0.358   0.000   0.591   0.000 testing.py:12(sub2)
101000   0.623   0.000   0.623   0.000 {built-in method sleep}
```

```
def func():
    for x in range(1000):
        sub1()
def sub1():
    sleep(0.001)
    for x in range(100):
        sub2()
def sub2():
    sleep(0.0001)
```

Python Call Graph



RunSnakeRun



http://www.vrplumber.com/programming/runsnakerun/screenshot-2.0.png

Geschwindigkeit von Python

- Referenzimplementierung: CPython
  - Keine weitreichende Optimierung
  - Zu langsam für unsere Zwecke
- Abhilfen:
  - PyPy: Interpreter mit JIT-Compiler
  - NumPy: Bibliothek mit C-Code für Arrays etc.
  - ctypes: Foreign Function Interface für C-Bibliotheken
  - Cython: Python nach C Übersetzer
  - ShedSkin: Python nach C++ Übersetzer

Tests

- Warum testen?
- Unit-Tests
- Weiterführende Tests
- Testbarer Code

Warum testen?

- Korrektheit des Programms sicherstellen
- Debugging ist aufwendig
- Änderungen am Code (bes. Refactoring) können Nebeneffekte verursachen

Unit-Tests

- Unit-Tests für kleine **Einheiten**
- Möglichst erschöpfend testen
- Erst den Test, dann den Code schreiben
  - Test-Driven Development (TDD)
- Unit-Test sind schnell
  - Vor Commit Tests laufen lassen
- Python-Module für das Testen:
  - `unittest`
  - `nosetest`
  - `pytest`
  - `coverage`
  - `mock`

Unit-Tests

Code:

```
def abs_incr(x):
  if x < 0:
    return x - 1
  else:
    return x + 1
```

Testfälle:

```
import pytest:

def test_zero(x):
  assert abs_incr(0) == 1

def test_positive(x):
  assert abs_incr(1) == 2

def test_negative(x):
  assert abs_incr(-1) == -2

def test_invalid(x):
  with pytest.raises(TypeError):
    abs_incr("")
```

Weiterführende Tests

- Unit-Tests sollen schnell und isoliert ablaufen
- Auch Tests für Zusammenspiel von Komponenten nötig
- Integrations-Tests
- System-Tests
- Weniger erschöpfend und langsamer

Testbarer Code

- Wann funktionieren Unit-Tests?
  - Codeteile (Funktionen etc.) aus kleinen Einheiten
  - Eine Aktion ↔ eine Funktion
  - Wenige, überschaubare oder keine Nebeneffekte
- Weitere Vorteile:

  Im Zusammenhang mit anderen Best Practices wie
  - Sinnvolle, klare Bezeichner
  - Docstrings und andere hilfreiche Kommentare
  - Konventionen:
    - Python: PEP 8, PEP 257, PEP 20 (`import this`) unter http://python.org/dev/peps/
    - C++: http://gcc.gnu.org/wiki/CppConventions

  wird Code strukturierter und lesbarer

Integrierte Entwicklungsumgebungen

- Test und Profiling sowie Debugging in IDEs möglich
- Auch grafische Aufbereitung gegeben
- Gute Python-Unterstützung in verbreiteten IDEs mit
  - PyDev in Eclipse
  - Python Tools for Visual Studio (PTVS)