Einführung in Python

Rebecca Breu

Verteilte Systeme und Grid-Computing
JSC
Forschungszentrum Jülich

Oktober 2007

Inhalt — Teil 1

Einführung

Was ist Python?

Python: dynamische Programmiersprache, welche verschiedene Programmierparadigmen unterstützt:

- prozedurale Programmierung
- objektorientierte Programmierung
- funktionale Programmierung

Standard: Python-Bytecode wird im Interpreter ausgeführt (ähnlich Java)
$\longrightarrow$ plattformunabhängiger Code

Warum Python?

- Syntax ist klar, leicht zu lesen & lernen (fast Pseudocode)
- intuitive Objektorientierung
- volle Modularität, hierarchische Pakete
- Fehlerbehandlung mittels Ausnahmen
- dynamische, „High Level“-Datentypen
- umfangreiche Standard-Bibliothek für viele Aufgaben
- einfache Erweiterbarkeit durch C/C++, Wrappen von C/C++-Bibliotheken

Schwerpunkt: Programmiergeschwindigkeit!

Ist Python schnell genug?

- für rechenintensive Algorithmen: evtl. besser Fortran, C, C++
- für Anwenderprogramme: Python ist schnell genug!
- Großteil der Python-Funktionen sind in C geschrieben
- Performance-kritische Teile können jederzeit in C/C++ ausgelagert werden
- erst analysieren, dann optimieren!

Hallo Welt!

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

# Dies ist ein Kommentar
print "Hallo Welt!"
```

```
$ python hallo_welt.py
Hallo Welt!
$
```

```
$ chmod 755 hallo_welt.py
$ ./hallo_welt.py
Hallo Welt!
$
```

Hallo User

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

name = raw_input("Wie heisst du?")
print "Hallo", name
```

```
$ ./hallo_user.py
Wie heisst du?
Rebecca
Hallo Rebecca
$
```

Starke und dynamische Typisierung

Starke Typisierung:

- Objekt ist genau von einem Typ! String ist immer String, `int` immer `int`
- Gegenbeispiel C: `char` kann als `short` betrachtet werden, `void *` kann alles sein

Dynamische Typisierung:

- keine Variablendeklaration
- Variablennamen können nacheinander unterschiedliche Datentypen zugewiesen werden
- Erst zur Laufzeit werden Eigenschaften eines Objekts untersucht

Starke und dynamische Typisierung

```
zahl = 3
print zahl, type(zahl)
print zahl + 42
zahl = "Rebecca"
print zahl, type(zahl)
print zahl + 42
```

```
3 <type 'int'>
45
Rebecca <type 'str'>
Traceback (most recent call last):
  File "test.py", line 6, in ?
    print zahl + 42
TypeError: cannot concatenate 'str' and
'int' objects
```

Interaktiver Modus

Der Interpreter kann im interaktiven Modus gestartet werden:

```
$ python
Python 2.4.1 (#1, Oct 13 2006, 16:58:04)
[GCC 4.0.2 20050901 (prerelease) ...
Type "help", "copyright", "credits" or ...
>>> print "hallo welt"
hallo welt
>>> a = 3 + 4
>>> print a
7
>>> 3 + 4
7
>>>
```

Dokumentation

Online-Hilfe im Interpreter:

- `help()`: allgemeine Hilfe zu Python
- `help(obj)`: Hilfe zu einem Objekt, z.B. einer Funktion oder einem Modul
- `dir()`: alle belegten Namen
- `dir(obj)`: alle Attribute eines Objekts

Offizielle Dokumentation: http://docs.python.org/

Dive into Python: http://diveintopython.org/

Dokumentation

```
>>> help(dir)
Help on built-in function dir:
...
>>> a = 3
>>> dir()
['__builtins__', '__doc__', '__file__',
'__name__', 'a']
>>> help(a)
Help on int object:
...
```

Datentypen I

Numerische Datentypen

- `int`: entspricht `long` in C
- `long`: unbegrenzter Wertebereich
- `float`: enspricht `double` in C
- `complex`: komplexe Zahlen

```
a = 1
b = 1L
c = 1.0; c = 1e0
d = 1 + 0j
```

Integers werden bei Bedarf automatisch in long umgewandelt!

Operatoren auf Zahlen

- Grundrechenarten: `+`, `-`, `*`, `/`
- Div- und Modulo-Operator: `//`, `%`, `divmod(x, y)`
- Betrag: `abs(x)`
- Runden: `round(x)`
- Konvertierung: `int(x)`, `long(x)`, `float(x)`, `complex(re [, im=0])`
- Konjugierte einer komplexen Zahl: `x.conjugate()`
- Potenzen: `x ** y`, `pow(x, y)`

Ergebnis einer Verknüpfung unterschiedlicher Datentypen ist vom Typ des „größeren“ Datentyps.

Strings

Datentyp: `str`

- `s = 'spam'`, `s = "spam"`
- Mehrzeilige Strings: `s = """spam"""`
- keine Interpretation von Escape-Sequenzen: `s = r"spam"`
- Strings aus anderen Datentypen erzeugen: `str(1.0)`

```
>>> print "sp\nam"
sp
am
>>> print r"sp\nam"
sp\nam
>>> s = """hallo
... welt"""
>>> print s
hallo
welt
```

String-Methoden

- Vorkommen von Substrings zählen:
  `s.count(sub [, start[, end]])`
- beginnt/endet s mit einem Substring?
  `s.startswith(sub[, start[, end]])`,
  `s.endswith(sub[, start[, end]])`
- s in Groß-/Kleinbuchstaben: `s.upper()`, `s.lower()`
- Leerraum entfernen: `s.strip([chars])`
- an Substrings trennen: `s.split([sub [,maxsplit]])`
- Position eines Substrings finden:
  `s.index(sub[, start[, end]])`
- einen Substring ersetzen: `s.replace(old, new[, count])`

Weitere Methoden: `help(str)`, `dir(str)`

Listen

Datentyp: `list`

- `s = [1, "spam", 9.0, 42]`, `s = []`
- Element anhängen: `s.append(x)`
- um zweite Liste erweitern: `s.extend(s2)`
- Vorkommen eines Elements zählen: `s.count(x)`
- Position eines Elements: `s.index(x[, min[, max]])`
- Element an Position einfügen: `s.insert(i, x)`
- Element an Position löschen und zurückgeben: `s.pop([i])`
- Element löschen: `s.remove(x)`
- Liste umkehren: `s.reverse()`
- Sortieren: `s.sort([cmp[, key[, reverse]]])`
- Summe der Elemente: `sum(s)`

Operationen auf Sequenzen

Stings und Listen haben viel gemeinsam: Sie sind Sequenzen.

- Ist ein Element in `s` enhalten/nicht enthalen?
  `x in s`, `x not in s`
- Sequenzen aneinanderhängen: `s + t`
- Sequenzen vervielfältigen: `n * s`, `s * n`
- i-tes Element: `s[i]`, von hinten: `s[-i]`
- Subsequenz: `s[i:j]`, mit Schrittweite k: `s[i:j:k]`
- Subsequenz von Anfgang/bis Ende: `s[:-2]`, `s[2:]`, `s[:]`
- Länge: `len(s)`
- kleinstes/größtes Element: `min(s)`, `max(s)`
- Zuweisungen: `(a, b, c) = s`
  $\longrightarrow$ `a = s[0]`, `b = s[1]`, `c = s[2]`

Sequenzen

- Auch eine Sequenz: Datentyp `tuple`: a = (1, 2, 3)
- Listen sind veränderbar
- Strings und Tupel sind nicht veränderbar
  - Keine Zuweisung `s[i] = ...`
  - Kein Anhängen und Löschen von Elementen
  - Funktionen wie `upper` liefern einen neuen String zurück!

```
>>> s1 = "spam"
>>> s2 = s1.upper()
>>> s1
'spam'
>>> s2
'SPAM'
```

Referenzen

- In Python ist alles eine Referenz auf ein Objekt!
- Vorsicht bei Zuweisungen:

```
>>> s1 = [1, 2, 3, 4]
>>> s2 = s1
>>> s2[1] = 17
>>> s1
[1, 17, 3, 4]
>>> s2
[1, 17, 3, 4]
```

Flache Kopie einer Liste: `s2 = s1[:]` oder `s2 = list(s1)`

Wahrheitswerte

Datentyp bool: `True`, `False`
Werte, die zu `False` ausgewertet werden:

- `None`
- `False`
- `0` (in jedem numerischen Datentyp)
- leere Strings, Listen und Tupel: `''`, `()`, `[]`
- leere Dictionaries: {}
- leere Sets

Andere Objekte von eingebauten Datentypen werden stets zu `True` ausgewertet!

```
>>> bool([1, 2, 3])
True
>>> bool("")
False
```

Statements

Das if-Statement

```
if a == 3:
    print "Aha!"
```

- Blöcke werden durch Einrückung festgelegt!
- Standard: Einrückung mit vier Leerzeichen

```
if a == 3:
    print "spam"
elif a == 10:
    print "eggs"
elif a == -3:
    print "bacon"
else:
    print "something else"
```

Vergleichsoperatoren

- Vergleich des Inhalts: ==, <, >, <=, >=, !=
- Vergleich der Objektidentität: `a is b`, `a is not b`
- Und/Oder-Verknüpfung: `a and b`, `a or b`
- Negation: `not a`

```
if not (a==b) and (c<3):
    pass
```

for-Schleifen

```
for i in range(10):
    print i    # 0, 1, 2, 3, ..., 9

for i in range(3, 10):
    print i    # 3, 4, 5, ..., 9

for i in range(0, 10, 2):
    print i    # 0, 2, 4, ..., 8
else:
    print "Schleife komplett durchlaufen."
```

- Schleife vorzeitig beenden: `break`
- nächster Durchlauf: `continue`
- `else` wird ausgeführt, wenn die Schleife nicht vorzeitig verlassen wurde

Über Sequenzen kann man direkt (ohne Index) iterieren:

```
for item in ["spam", "eggs", "bacon"]:
    print item
```

Auch die range-Funktion liefert eine Liste:

```
>>> range(0, 10, 2)
[0, 2, 4, 6, 8]
```

Benötigt man doch Indices:

```
for (i, char) in enumerate("hallo welt"):
    print i, char
```

while-Schleifen

```
while i < 10:
    i += 1
```

Auch hier können `break` und `continue` verwendet werden.

Ersatz für do-while-Schleife:

```
while True:
    # wichtiger Code
    if bedingung:
        break
```

Funktionen I

Funktionen

```
def addiere(a, b):
    """Gibt die Summe von a und b zurueck."""

    summe = a + b
    return summe
```

```
>>> ergebnis = addiere(3, 5)
>>> print ergebnis
8
>>> help(addiere)
Help on function addiere in module __main__:

addiere(a, b)
    Gibt die Summe von a und b zurueck.
```

Rückgabewerte und Parameter

- Funktionen können beliebige Objekte als Parameter und Rückgabewerte haben
- Typen der Rückgabewerte und Parameter sind nicht festgelegt
- Funktionen ohne expliziten Rückgabewert geben `None` zurück

```
def hallo_welt():
    print "Hallo Welt!"

a = hallo_welt()
print a
```

```
$ mein_programm.py
Hallo Welt
None
```

Mehrere Rückgabewerte

Mehrere Rückgabewerte werden mittels Tupel oder Listen realisiert:

```
def foo():
    a = 17
    b = 42
    return (a, b)

ret = foo()
(z1, z2) = foo()
```

Keywords und Defaultwerte

Man kann Parameter auch in anderer Reihenfolge als definiert angeben:

```
def foo(a, b, c):
    print a, b, c

foo(b=3, c=1, a="hallo")
```

Defaultwerte festlegen:

```
def foo(a, b, c=1.3):
    print a, b, c

foo(1, 2)
foo(1, 17, 42)
```

Funktionen sind Objekte

Funktionen sind Objekte und können wie solche zugewiesen und übergeben werden:

```
>>> def foo(f):
...     print f(33)
...
>>> foo(float)
33.0
```

```
>>> a = float
>>> a(22)
22.0
```

Input/Output

String-Formatierung

Stringformatierung ähnlich C:

```
print "Die Antwort ist %i." % 42
s = "%s: %3.4f" % ("spam", 3.14)
```

- Integer dezimal: d, i
- Integer oktal: o
- Integer hexadezimal: x, X
- Float: f, F
- Float in Exponentialdarstellung: e, E, g, G
- Einzelnes Zeichen: c
- String: s

Ein %-Zeichen gibt man als %% aus.

Kommandozeilen-Eingaben

Benutzer-Eingaben:

```
input = raw_input("Gib was ein:")
```

Kommandozeilen-Parameter:

```
import sys
print sys.argv
```

```
$ ./params.py spam
['params.py', 'spam']
```

Dateien

```
datei1 = open("spam", "r")
datei2 = open("eggs", "wb")
```

- Lesemodus: r
- Schreibmodus: w
- Binärdateien behandeln: b
- Schreibmodus, an Daten am Ende anhängen: a
- Lesen und schreiben: r+

```
for line in datei1:
    print line
```

Operationen auf Dateien

- lesen: `f.read([size])`
- Zeile lesen: `f.readline()`
- mehrere Zeilen lesen: `f.readlines([sizehint])`
- schreiben: `f.write(str)`
- mehrere Zeilen schreiben: `f.writelines(sequence)`
- Datei schließen: `f.close()`

```
datei = open("test", "w")
lines = ["spam\n", "eggs\n", "ham\n"]
datei.writelines(lines)
datei.close()
```

Module und Pakete

Module importieren

Funktionen, Klassen und Objekte, die thematisch zusammengehören, werden in Modulen gebündelt.

```
import math
s = math.sin(math.pi)
```

```
import math as m
s = m.sin(m.pi)
```

```
from math import pi as PI, sin
s = sin(PI)
```

```
from math import *
s = sin(pi)
```

Module

- Hilfe: `dir(math)`, `help(math)`
- Module werden gesucht in:
  - dem Verzeichnis der aufrufenden Datei
  - Verzeichnissen aus der Umgebungsvariablen `PYTHONPATH`
  - installationsbedingten Verzeichnissen

```
>>> import sys
>>> sys.path
['', '/usr/lib/python24.zip',
 '/usr/lib/python2.4',
 '/usr/lib/python2.4/site-packages', ...]
```

Pakete importieren

Module können zu hierarchisch strukturierten Paketen zusammengefasst werden.

```
from email.mime import text as mtext
msg = mtext.MIMEText("Hallo Welt!")
```

```
from email.mime.text import MIMEText
msg = MIMEText("Hallo Welt!")
```

Eigene Module

Jedes Python-Programm kann als Modul importiert werden.

```
"""Mein erstes Modul: mein_modul.py"""

def add(a, b):
    """Addiere a und b."""
    return a + b

print add(2, 3)
```

```
>>> import mein_modul
5
>>> mein_modul.add(17, 42)
59
```

Top-Level-Anweisungen werden beim Import ausgeführt!

Eigene Module

Sollen Anweisungen nur beim direkten Ausführen, nicht beim Importieren ausgeführt werden:

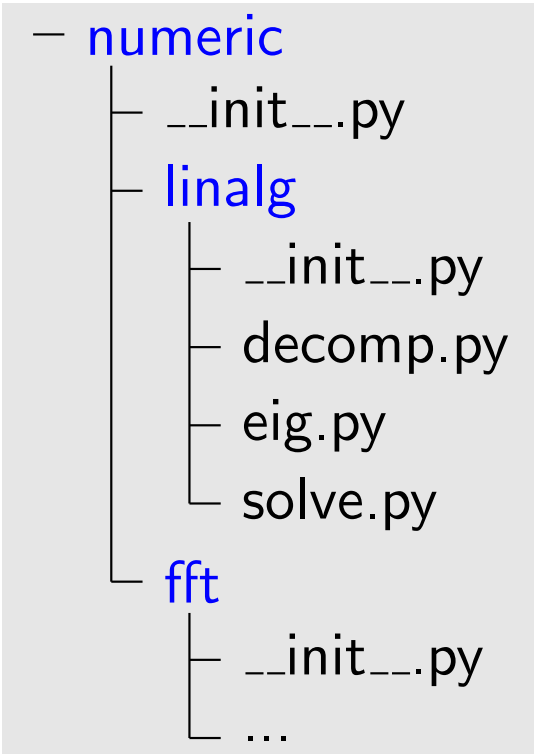
```
def add(a, b):
    return a + b

def main():
    print add(2, 3)

if __name__ == "__main__":
    main()
```

Sinnvoll z.B. für Tests.

Eigene Pakete



In jedem Paket-Ordner: `__init__.py`
(kann leer sein)

```
import numeric
numeric.foo() #Aus __init__.py
numeric.linalg.eig.foo()
```

```
from numeric.linalg import eig
eig.foo()
```

Fehler und Ausnahmen

Syntax Errors, Indentation Errors

Fehler beim Parsen: Programm wird nicht ausgeführt. Z.B.:

- Klammerungsfehler
- Falsche oder fehlende Semikolons, Doppelpunkte, Kommas
- Einrückungsfehler

```
print "Ich laufe..."
def add(a, b)
    return a + b
```

```
$ ./addiere.py
  File "addiere.py", line 1
    def add(a, b)
                 ^
SyntaxError: invalid syntax
```

Ausnahmen

Ausnahmen (Exceptions) treten zur Laufzeit auf:

```
import math
print "Ich laufe..."
math.foo()
```

```
$ ./test.py
Ich laufe...
Traceback (most recent call last):
  File "test.py", line 3, in ?
    math.foo()
AttributeError: 'module' object has no
attribute 'foo'
```

Ausnahmen behandeln

```
try:
    s = raw_input("Gib eine Zahl ein: ")
    zahl = float(s)
except ValueError:
    print "Das ist keine Zahl!"
```

- except-Block wird ausgeführt, wenn Code im try-Block eine passende Ausnahme wirft
- danach läuft Programm normal weiter
- nicht behandelte Ausnahmen führen zum Programmabbruch

Verschiedene Ausnahmen abfangen:

```
except (ValueError, TypeError, NameError):
```

Ausnahmen behandeln

```
try:
    s = raw_input("Gib eine Zahl ein: ")
    zahl = 1/float(s)
except ValueError:
    print "Das ist keine Zahl!"
except ZeroDivisionError:
    print "Man kann nicht durch Null teilen!"
except:
    print "Was ist hier passiert?"
```

- Mehrere except-Statements für verschiedene Ausnahmen
- Letztes except kann ohne Ausnahme-Typ verwendet werden: Fängt alle verbleibenen Ausnahmen ab
  - Vorsicht: Kann ungewollte Programmierfehler verdecken!

Ausnahmen behandeln

- `else` wird ausgeführt, wenn keine Ausnahme auftrat
- `finally` wird in jedem Fall ausgeführt

```
try:
    f = open("spam")
except IOError:
    print "Cannot open file"
else:
    print f.read()
    f.close()
finally:
    print "Ende von try."
```

Ausnahme-Objekte

Auf das Ausnahme-Objekt zugreifen:

```
try:
    f = open("spam")
except IOError, e:
    print e.errno, e.strerror
    print e
```

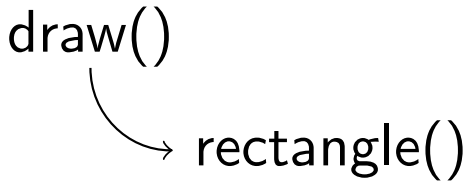
```
$ python test.py
2 No such file or directory
[Errno 2] No such file or directory: 'spam'
```

Ausnahmen in Funktionsaufrufen

draw()

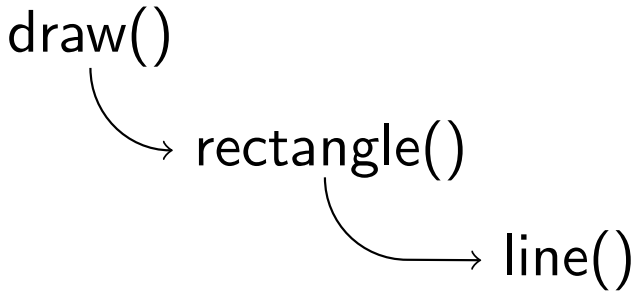
- Funktion ruft Unterfunktionen auf.
- Unterfunktion wirft Ausnahme.
- Wird Ausnahme behandelt?
- Nein: Gib Ausnahme an aufrufende Funktion weiter.

Ausnahmen in Funktionsaufrufen



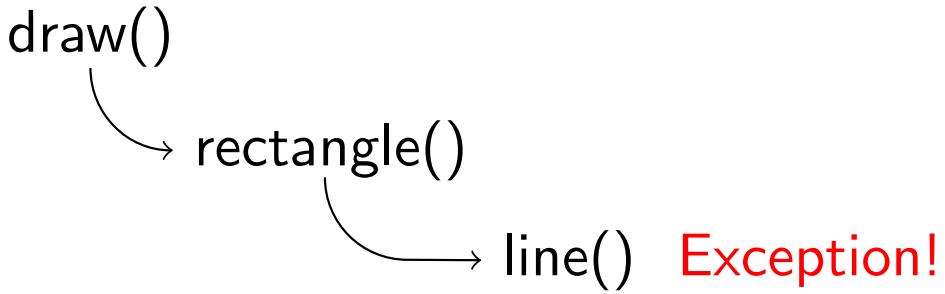
- Funktion ruft Unterfunktionen auf.
- Unterfunktion wirft Ausnahme.
- Wird Ausnahme behandelt?
- Nein: Gib Ausnahme an aufrufende Funktion weiter.

Ausnahmen in Funktionsaufrufen



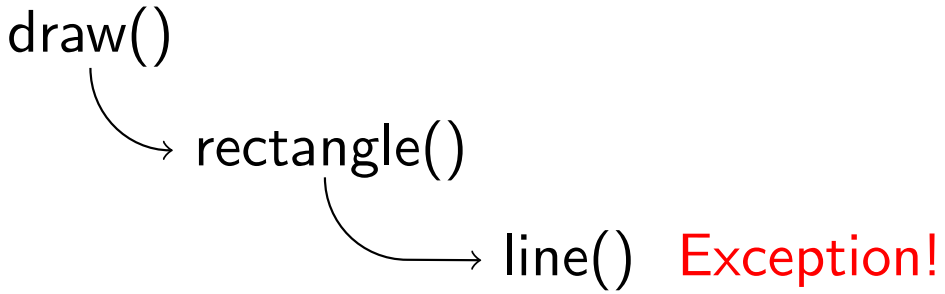
- Funktion ruft Unterfunktionen auf.
- Unterfunktion wirft Ausnahme.
- Wird Ausnahme behandelt?
- Nein: Gib Ausnahme an aufrufende Funktion weiter.

Ausnahmen in Funktionsaufrufen



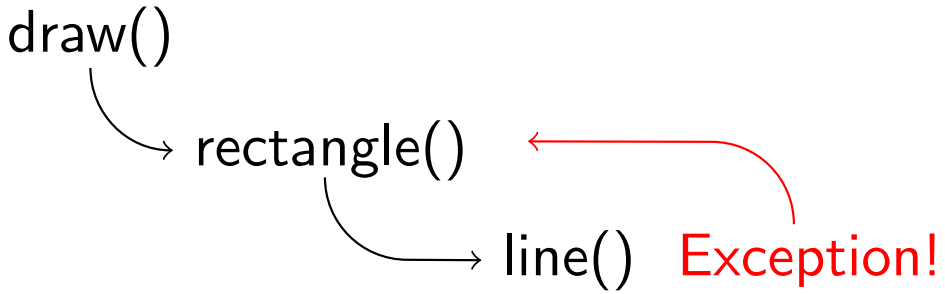
- Funktion ruft Unterfunktionen auf.
- Unterfunktion wirft Ausnahme.
- Wird Ausnahme behandelt?
- Nein: Gib Ausnahme an aufrufende Funktion weiter.

Ausnahmen in Funktionsaufrufen



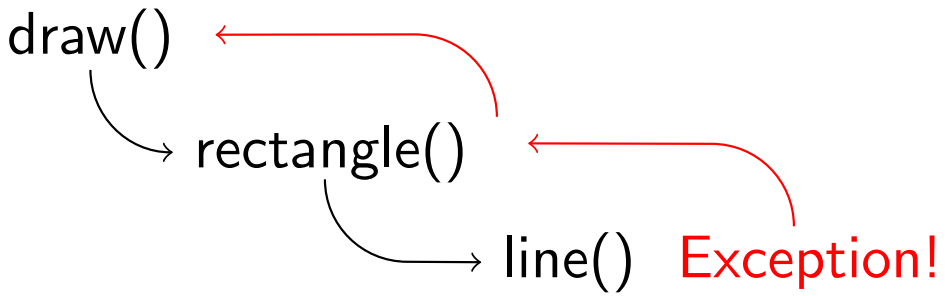
- Funktion ruft Unterfunktionen auf.
- Unterfunktion wirft Ausnahme.
- Wird Ausnahme behandelt?
- Nein: Gib Ausnahme an aufrufende Funktion weiter.

Ausnahmen in Funktionsaufrufen



- Funktion ruft Unterfunktionen auf.
- Unterfunktion wirft Ausnahme.
- Wird Ausnahme behandelt?
- Nein: Gib Ausnahme an aufrufende Funktion weiter.

Ausnahmen in Funktionsaufrufen



- Funktion ruft Unterfunktionen auf.
- Unterfunktion wirft Ausnahme.
- Wird Ausnahme behandelt?
- Nein: Gib Ausnahme an aufrufende Funktion weiter.

Ausnahmen auslösen

Ausnahmen weiterreichen:

```
try:
    f = open("spam")
except IOError:
    print "Fehler beim Oeffnen!"
    raise
```

Ausnahmen auslösen:

```
def gauss_solver(matrix):
    # Wichtiger Code
    raise ValueError("Matrix singulaer")
```

Viel Spaß mit