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

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:

```
eingabe = 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:

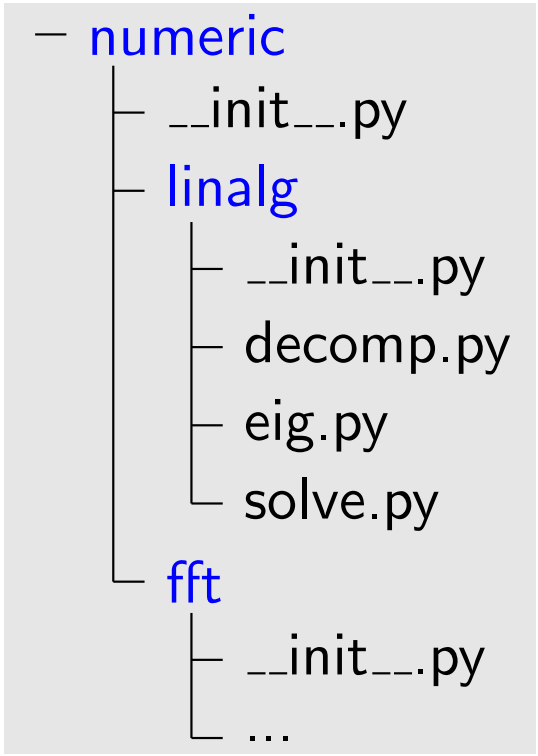
```
def add(a, b):
    return a + b

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
```

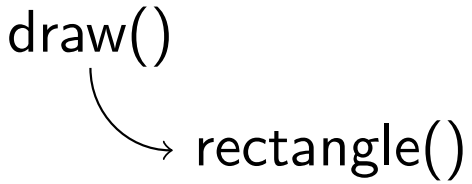
```
$ python test.py
2 No such file or directory
[Errno 2] No such file or directory: 'spam'
```

Ausnahmen in Funktionsaufrufen

draw()

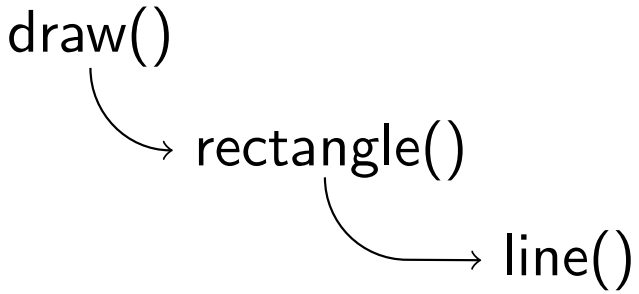
- Funktion ruft Unterfunktionen auf.
- Unterfunktion wirft Ausnahme.
- Wird Ausnahme behandelt?
- Nein: Gib Ausnahme an aufrufende Funktion weiter.

Ausnahmen in Funktionsaufrufen



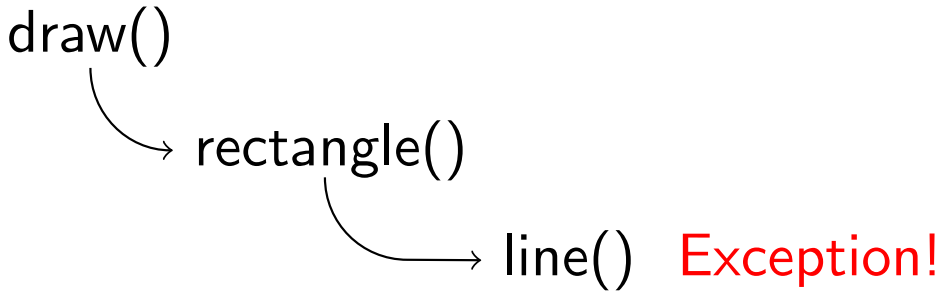
- Funktion ruft Unterfunktionen auf.
- Unterfunktion wirft Ausnahme.
- Wird Ausnahme behandelt?
- Nein: Gib Ausnahme an aufrufende Funktion weiter.

Ausnahmen in Funktionsaufrufen



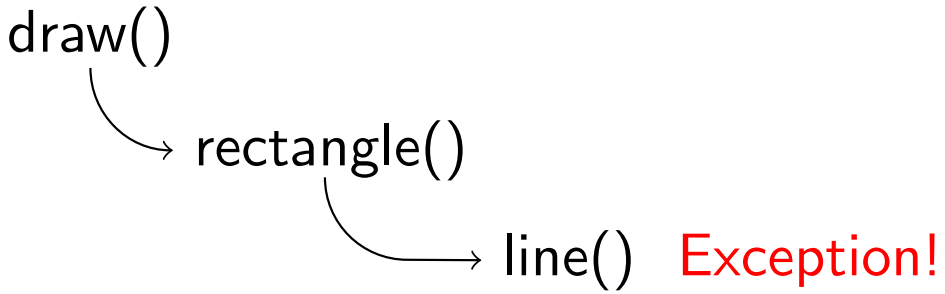
- Funktion ruft Unterfunktionen auf.
- Unterfunktion wirft Ausnahme.
- Wird Ausnahme behandelt?
- Nein: Gib Ausnahme an aufrufende Funktion weiter.

Ausnahmen in Funktionsaufrufen



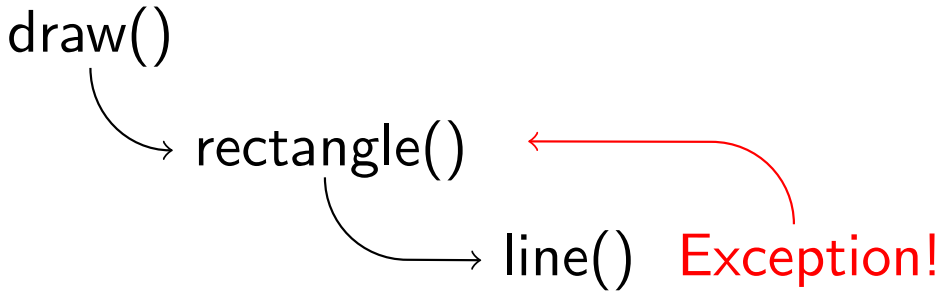
- Funktion ruft Unterfunktionen auf.
- Unterfunktion wirft Ausnahme.
- Wird Ausnahme behandelt?
- Nein: Gib Ausnahme an aufrufende Funktion weiter.

Ausnahmen in Funktionsaufrufen



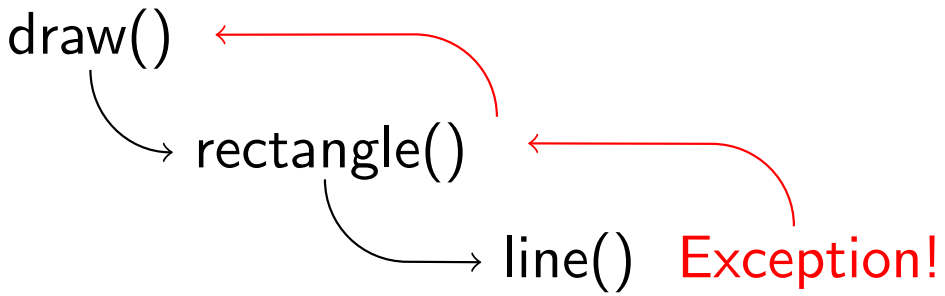
- Funktion ruft Unterfunktionen auf.
- Unterfunktion wirft Ausnahme.
- Wird Ausnahme behandelt?
- Nein: Gib Ausnahme an aufrufende Funktion weiter.

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

Einführung in Python

Rebecca Breu

Verteilte Systeme und Grid-Computing
JSC
Forschungszentrum Jülich

Oktober 2007

Inhalt — Teil 2

Datentypen II

Sets

Set (Menge): ungeordnet, doppelte Elemente werden nur einmal gespeichert

- `s = set([sequence])`
- Teilmenge: `s.issubset(t)`, `s <= t`, echte T.: `s < t`
- Obermenge: `s.issuperset(t)`, `s >= t`, echte O.: `s > t`
- Vereinigung: `s.union(t)`, `s | t`
- Schnittmenge: `s.intersection(t)`, `s & t`
- Differenz: `s.difference(t)`, `s - t`
- Symmetrische Differenz: `s.symmetric_difference(t)`, `s ^ t`
- Kopie: `s.copy()`

Wie für Sequenzen gibt es auch: `x in set`, `len(set)`, `for x in set`, `add`, `remove`

Dictionaries

Dictionary: Zuordnung Schlüssel → Wert

```
>>> a = { "spam": 1, "eggs": 17}
>>> a["eggs"]
17
>>> a["bacon"] = 42
>>> a
{'eggs': 17, 'bacon': 42, 'spam': 1}
```

Über Dictionaries iterieren:

```
for key in a:
    print key, a[key]
```

Operationen auf Dictionaries

- Eintrag löschen: `del`
- alle Einträge löschen: `a.clear()`
- Kopie: `a.copy()`
- Ist Schlüssel enthalten? `a.has_key(k)`, `k in a`
- Liste von (key, value)-Tupeln: `a.items()`
- Liste aller Schlüssel: `a.keys()`
- Liste aller Werte: `a.values()`
- Eintrag holen: `a.get(k[, x])`
- Eintrag löschen und zurückgeben: `a.pop(k[, x])`
- Eintrag löschen und zurückgeben: `a.popitem()`

Objektorientierte Programmierung

Objektorientierte Programmierung

- Bisher: prozedurale Programmierung
  - Daten
  - Funktionen, die auf den Daten operieren
- Alternative: Fasse zusammengehörige Daten und Funktionen zusammen zu eigenem Datentypen
- $\longrightarrow$ Erweiterung von Strukturen/Datenverbünden aus C/Fortran

Einfache Klassen als Structs verwenden

```
class Punkt:
    pass

p = Punkt()
p.x = 2.0
p.y = 3.3
```

- Klasse: Eigener Datentyp (hier: `Punkt`)
- Objekt: Instanz der Klasse (hier: `p`)
- Attribute (hier `x`, `y`) können dynamisch hinzugefügt werden

Klassen

```
class Punkt:
    def __init__(self, x, y):
        self.x = x
        self.y = y

p = Punkt(2.0, 3.0)
print p.x, p.y
p.x = 2.5
p.z = 42
```

- `__init__`: Wird automatisch nach Erzeugung eines Objekts aufgerufen

Methoden auf Objekten

```
class Punkt:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def norm(self):
        n = math.sqrt(self.x**2 + self.y**2)
        return n

p = Punkt(2.0, 3.0)
print p.x, p.y, p.norm()
```

- Methodenaufruf: automatisch das Objekt als erster Parameter
- $\longrightarrow$ wird üblicherweise `self` genannt
- Achtung: Kein Überladen von Methoden möglich!

Objekte in Strings konvertieren

Standard-Rückgabe von `str(...)` für eigene Objekte:

```
>>> p = Punkt(2.0, 3.0)
>>> print p  # --> print str(p)
<__main__.Punkt instance at 0x402d7a8c>
```

Das kann man anpassen:

```
def __str__(self):
    return "(%i, %i)" % (self.x, self.y)
```

```
>>> print p
(2, 3)
```

Objekte in Strings konvertieren

Standard-Rückgabe von `str(...)` für eigene Objekte:

```
>>> p = Punkt(2.0, 3.0)
>>> print p  # --> print str(p)
<__main__.Punkt instance at 0x402d7a8c>
```

Das kann man anpassen:

```
def __str__(self):
    return "(%i, %i)" % (self.x, self.y)
```

```
>>> print p
(2, 3)
```

Objekte vergleichen

Standard: == prüft Objekte eigener Klassen auf Identität.

```
>>> p1 = Punkt(2.0, 3.0)
>>> p2 = Punkt(2.0, 3.0)
>>> p1 == p2
False
```

Das kann man anpassen:

```
def __eq__(self, other):
    return (self.x == other.x) and
           (self.y == other.y)
```

```
>>> p1 == p2
True
>>> p1 is p2 # Identitaet pruefen
False
```

Objekte vergleichen

Standard: == prüft Objekte eigener Klassen auf Identität.

```
>>> p1 = Punkt(2.0, 3.0)
>>> p2 = Punkt(2.0, 3.0)
>>> p1 == p2
False
```

Das kann man anpassen:

```
def __eq__(self, other):
    return (self.x == other.x) and
           (self.y == other.y)
```

```
>>> p1 == p2
True
>>> p1 is p2 # Identitaet pruefen
False
```

Objekte vergleichen

Weitere Vergleichsoperatoren:

- `<` : `__lt__(self, other)`
- `<=` : `__le__(self, other)`
- `!=` : `__ne__(self, other)`
- `>` : `__gt__(self, other)`
- `>=` : `__ge__(self, other)`

Alternativ: `__cmp__(self, other)`, gibt zurück:

- negativen Integer, wenn `self < other`
- null, wenn `self == other`
- positiven Integer, wenn `self > other`

Datentypen emulieren

Man kann mit Klassen vorhandene Datentypen emulieren:

- Zahlen: `int(myobj)`, `float(myobj)`, Rechenoperationen, ...
- Funktionen: `myobj(...)`
- Sequenzen: `len(myobj)`, `myobj[...]`, `x in myobj`, ...
- Iteratoren: `for i in myobj`

Siehe dazu Dokumentation:
http://docs.python.org/ref/specialnames.html

Klassenvariablen

Haben für alle Objekte einer Klasse stets den gleichen Wert:

```
class Punkt:
    anzahl = 0  #Anzahl aller Punkt-Objekte
    def __init__(self, x, y):
        self.__class__.anzahl += 1
        ...
```

```
>>> p1 = Punkt(2, 3); p2 = Punkt(3, 4)
>>> p1.anzahl
2
>>> p2.anzahl
2
>>> Punkt.anzahl
2
```

Klassenmethoden und statische Methoden

```
class Spam:
    spam = "I don't like spam."

    @classmethod
    def cmethod(cls):
        print cls.spam

    @staticmethod
    def smethod():
        print "Blah blah."
```

```
Spam.cmethod()
Spam.smethod()
s = Spam()
s.cmethod()
s.smethod()
```

Vererbung

Oft hat man verschiedene Klassen, die einander ähneln.
Vererbung erlaubt:

- Hierarchische Klassenstruktur (Ist-ein-Beziehung)
- Wiederverwenden von ähnlichem Code

Beispiel: Verschiedene Telefon-Arten

- Telefon
- Handy (ist ein Telefon mit zusätzlichen Funktionen)
- Fotohandy (ist ein Handy mit zusätzlichen Funktionen)

Vererbung

```
class Telefon:
    def telefonieren(self):
        pass

class Handy(Telefon):
    def sms_schicken(self):
        pass
```

Handy erbt jetzt Methoden und Attribute von Telefon.

```
h = Handy()
h.telefonieren() # Geerbt von Telefon
h.sms_schicken() # Eigene Methode
```

Methoden überschreiben

In der abgeleiteten Klasse können die Methoden der Elternklasse überschrieben werden:

```
class Handy(Telefon):
    def telefonieren(self):
        suche_funkverbindung()
        Telefon.telefonieren(self)
```

Mehrfachvererbung

Klassen können von mehreren Elternklassen erben. Bsp:

- Fotohandy ist ein Telefon
- Fotohandy ist eine Kamera

```
class Fotohandy(Handy, Kamera):
    pass

h = Fotohandy()
h.telefonieren()   # geerbt von Handy
h.fotografieren()  # geerbt von Kamera
```

Attribute werden in folgender Reihenfolge gesucht:
Fotohandy, Handy, Elterklasse von Handy (rekursiv), Kamera, Elternklasse von Kamera (rekursiv).

Private Attribute

- In Python gibt keine privaten Variablen oder Methoden.
- Konvention: Attribute, auf die nicht von außen zugegriffen werden sollte, beginnen mit einem Unterstrich: `_foo`.
- Um Namenskonflikte zu vermeiden: Namen der Form `__foo` werden durch `_klassenname__foo` ersetzt:

```
class Spam:
    __eggs = 3
```

```
>>> dir(Spam)
>>> ['_Spam__eggs', '__doc__', '__module__']
```

Properties

Sollen beim Zugriff auf eine Variable noch Berechnungen oder Überprüfungen durchgeführt werden: Getter und Setter

```
class Spam(object):
    def __init__(self):
        self._value = 0

    def _get_value(self):
        return self._value

    def _set_value(self, value):
        if value <= 0:  self._value = 0
        else:  self._value = value

    value = property(_get_value, _set_value)
```

Properties

Auf Properties wird wie auf gewöhnliche Attribute zugegriffen:

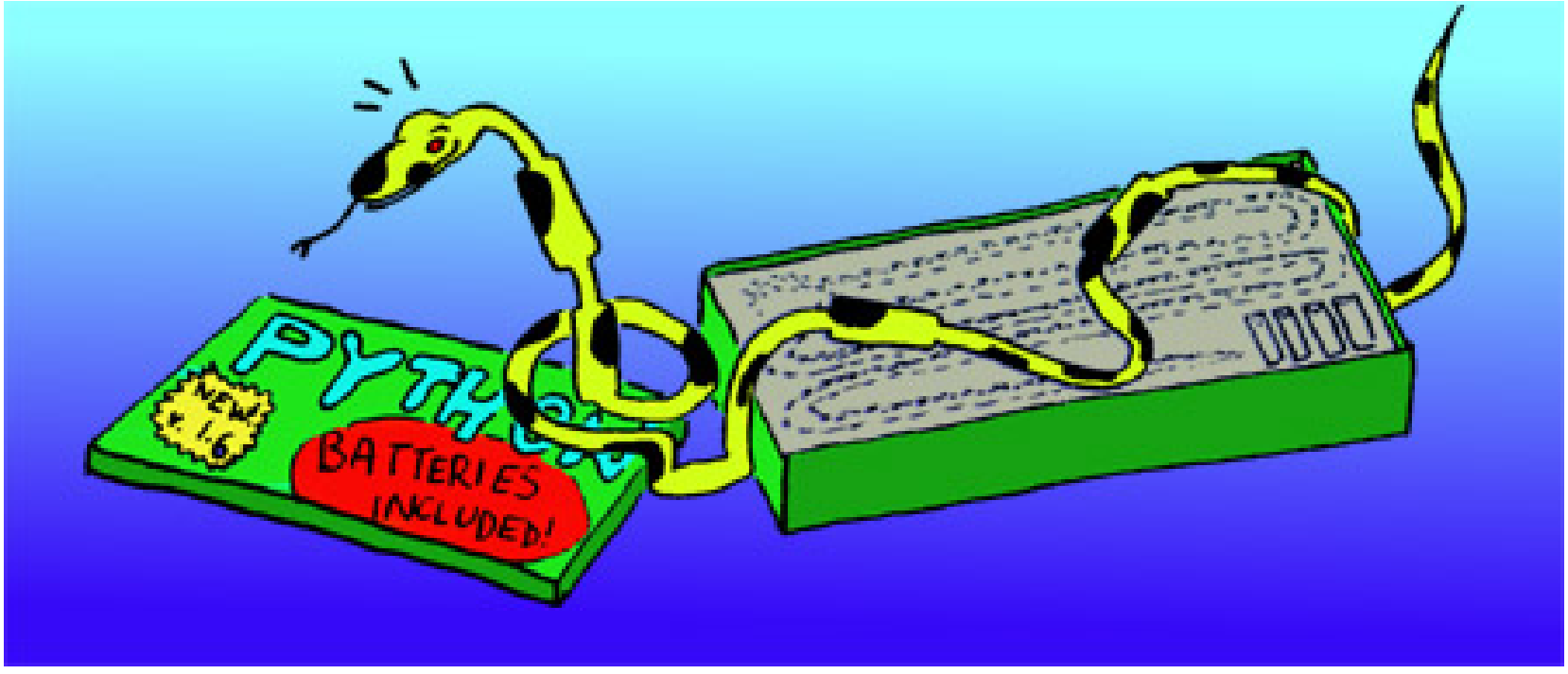
```
>>> s = Spam()
>>> s.value = 6    # set_value(6)
>>> s.value        # get_value()
>>> 6
>>> s.value = -6   # set_value(-6)
>>> s.value        # get_value()
>>> 0
```

- Getter und Setter können nachträglich hinzugefügt werden, ohne die API zu verändern.
- Zugriff auf `_value` immer noch möglich

Pythons Standardbibliothek

Pythons Standardbibliothek

„Batteries included“: umfassende Standardbibliothek für die verschiedensten Aufgaben



Mathematik: `math`

- Konstanten: `e`, `pi`
- Auf- und Abrunden: `floor(x)`, `ceil(x)`
- Exponentialfunktion: `exp(x)`
- Logarithmus: `log(x[, base])`, `log10(x)`
- Potenz und Quadratwurzel: `pow(x, y)`, `sqrt(x)`
- Trigonometrische Funktionen: `sin(x)`, `cos(x)`, `tan(x)`
- Kovertierung Winkel $\leftrightarrow$ Radiant: `degrees(x)`, `radians(x)`

```
>>> import math
>>> math.sin(math.pi)
1.2246063538223773e-16
>>> math.cos(math.radians(30))
0.8660254037844387
```

Zufall: `random`

- Zufällige Integers:
  `randint(a, b)`, `randrange([start,] stop[, step])`
- Zufällige Floats (Gleichverteilg.): `random()`, `uniform(a, b)`
- Andere Verteilungen: `expovariate(lambd)`,
  `gammavariate(alpha, beta)`, `gauss(mu, sigma)`, ...
- Zufälliges Element einer Sequenz: `choice(seq)`
- Mehrere eindeutige, zufällige Elemente einer Sequenz:
  `sample(population, k)`
- Sequenz mischen: `shuffle(seq[, random])`

```
>>> s = [1, 2, 3, 4, 5]
>>> random.shuffle(s)
>>> s
[2, 5, 4, 3, 1]
>>> random.choice("Hallo Welt!")
'e'
```

Operationen auf Verzeichnisnamen: os.path

- Pfade: `abspath(path)`, `basename(path)`, `normpath(path)`, `realpath(path)`
- Pfad zusammensetzen: `join(path1[, path2[, ...]])`
- Pfade aufspalten: `split(path)`, `splitext(path)`
- Datei-Informationen: `isfile(path)`, `isdir(path)`, `islink(path)`, `getsize(path)`, ...
- Home-Verzeichnis vervollständigen: `expanduser(path)`
- Umgebungsvariablen vervollständigen: `expandvars(path)`

```
>>> os.path.join("spam", "eggs", "ham.txt")
'spam/eggs/ham.txt'
>>> os.path.splitext("spam/eggs.py")
('spam/eggs', '.py')
>>> os.path.expanduser("~/spam")
'/home/rbreu/spam'
>>> os.path.expandvars("/bla/$TEST")
'/bla/test.py'
```

Dateien und Verzeichnisse: os

- Working directory: `getcwd()`, `chdir(path)`
- Dateirechte ändern: `chmod(path, mode)`
- Benutzer ändern: `chown(path, uid, gid)`
- Verzeichnis erstellen: `mkdir(path[, mode])`, `makedirs(path[, mode])`
- Dateien löschen: `remove(path)`, `removedirs(path)`
- Dateien umbenennen: `rename(src, dst)`, `renames(old, new)`
- Liste von Dateien in Verzeichnis: `listdir(path)`

```
for datei in os.listdir("mydir"):
    os.chmod(os.path.join("mydir", datei),
             stat.S_IRUSR|stat.S_IWUSR)
```

Verzeichnislisting: `glob`

Liste von Dateien in Verzeichnis, mit Unix-artiger Wildcard-Vervollständigung: `glob(path)`

```
>>> glob.glob("python/[a-c]*.py")
['python/confitest.py',
 'python/basics.py',
 'python/curses_test2.py',
 'python/curses_keys.py',
 'python/cmp.py',
 'python/button_test.py',
 'python/argument.py',
 'python/curses_test.py']
```

Dateien und Verzeichnisse: `shutil`

Higher Level-Operationen auf Dateien und Verzeichnissen.

- Datei kopieren: `copyfile(src, dst)`, `copy(src, dst)`
- Rekursiv kopieren; `copytree(src, dst[, symlinks])`
- Rekursiv löschen:
  `rmtree(path[, ignore_errors[, onerror]])`
- Rekursiv verschieben: `move(src, dst)`

```
shutil.copytree("spam/eggs", "../beans",
                symlinks=True)
```

Andere Prozesse starten: subprocess

Einfaches Ausführen eines Programmes:

```
p = subprocess.Popen(["ls", "~"], shell=True)
returncode = p.wait() # Auf Ende warten
```

Zugriff auf die Ausgabe eines Programmes:

```
p = Popen(["ls"], stdout=PIPE, stderr=STDOUT,
          close_fds=True)
p.wait()
output = p.stdout.read()
```

Pipes zwischen Prozessen (`ls -l | grep txt`)

```
p1 = Popen(["ls", "-l"], stdout=PIPE)
p2 = Popen(["grep", "txt"], stdin=p1.stdout)
```

Threads: threading

Programmteile gleichzeitig ablaufen lassen mit Thread-Objekten:

```
class Counter(threading.Thread):
    def __init__(self):
        threading.Thread.__init__(self)
        self.counter = 0

    def run(self):  # Hauptteil
        while self.counter < 10:
            self.counter += 1
            print self.counter

counter = Counter()
counter.start() # Thread starten
# hier etwas gleichzeitig tun...
counter.join() # Warte auf Ende des Threads
```

Threads: `threading`

- Problem, wenn zwei Threads gleichzeitig auf das gleiche Objekt schreibend zugreifen wollen!
- $\longrightarrow$ Verhindern, dass Programmteile gleichzeitig ausgeführt werden mit Lock-Objekten
- Locks haben genau zwei Zustände: `locked` und `unlocked`

```
lock = threading.Lock()

lock.aquire() # Warte bis Lock frei ist
              # und locke es dann
#... wichtiger Code
lock.release() # Lock freigeben fuer andere
```

Threads: `threading`

- Kommunikation zwischen Threads: Z.B. mittels Event-Objekten
- Events haben zwei Zustände: gesetzt und nicht gesetzt
- ähnlich Locks, aber ohne gegenseitigen Ausschluss

Bsp: Event, um Threads mitzuteilen, dass sie sich beenden sollen.
Methoden auf Event-Objekten:

- Status des Events abfragen: `isSet()`
- Setzen des Events: `set()`
- Zurücksetzten des Events: `clear()`
- Warten, dass Event gesetzt wird: `wait([timeout])`

Zugriff auf Kommandozeilenparameter: optparse

- Einfach: Liste mit Parametern → `sys.argv`
- Komfortabler für mehrere Optionen: `OptionParser`

```
parser = optparse.OptionParser()
parser.add_option("-f", "--file",
                  dest="filename",
                  default="out.txt",
                  help="Output file")
parser.add_option("-q", "--quiet",
                  action="store_false",
                  dest="verbose",
                  default=True,
                  help="don't print info")

(options, args) = parser.parse_args()
print options.filename, options.verbose
print args
```

Zugriff auf Kommandozeilenparameter: optparse

So wird ein optparse-Programm verwendet:

```
$ ./test.py -h
usage: test.py [options]

options:
  -h, --help            show this help message and exit
  -f FILENAME, --file=FILENAME
                        Output file
  -q, --quiet           don't print status messages
```

```
$ ./test.py -f aa bb cc
aa True
['bb', 'cc']
```

Konfigurationsdateien: `ConfigParser`

Einfaches Format zum Speichern von Konfigurationen u.Ä.:
Windows INI-Format

```
[font]
font = Times New Roman
#this is a comment
size = 16

[colors]
font = black
pointer = %(font)s
background = white
```

Konfigurationsdateien: ConfigParser

Config-Datei lesen:

```
parser = ConfigParser.SafeConfigParser()
parser.readfp(open("config.ini", "r"))
print parser.get("colors", "font")
```

Weitere Parser-Methoden:

- Liste aller Sections: `sections()`
- Liste aller Optionen: `options(section)`
- Liste aller Optionen und Werte: `items(section)`
- Werte lesen: `get(sect, opt)`, `getint(sect, opt)`, `getfloat(sect, opt)`, `getboolean(sect, opt)`

Konfigurationsdateien: ConfigParser

Config-Datei schreiben:

```
parser = ConfigParser.SafeConfigParser()
parser.add_section("colors")
parser.set("colors", "font", "black")
parser.write(open("config.ini", "w"))
```

Weitere Parser-Methoden:

- Section hinzufügen: `add_section(section)`
- Section löschen: `remove_section(section)`
- Option hinzufügen: `set(section, option, value)`
- Option entfernen: `remove_option(section, option)`

CSV-Dateien: csv

CSV: Comma-seperated values

- Tabellendaten im ASCII-Format
- Spalten durch ein festgelegtes Zeichen (meist Komma) getrennt

```
reader = csv.reader(open("test.csv", "rb"))
for row in reader:
    for item in row:
        print item
```

```
writer = csv.writer(open(outfile, "wb"))
writer.writerow([1, 2, 3, 4])
```

CSV-Dateien: csv

Mit verschiedenen Formaten (Dialekten) umgehen:

```
reader(csvfile, dialect='excel') # Default
writer(csvfile, dialect='excel_tab')
```

Einzelne Formatparameter angeben:

```
reader(csvfile, delimiter=";")
```

Weitere Formatparameter: `lineterminator`, `quotechar`, `skipinitialspace`, ...

Objekte serialisieren: `pickle`

Beliebige Objekte in Dateien speichern:

```
obj = {"hallo": "welt", "spam":1}
pickle.dump(obj, open("bla.txt", "wb"))
# ...
obj = pickle.load(open("bla.txt", "rb"))
```

Objekt in String unwandeln (z.B. zum Verschicken über Streams):

```
s = pickle.dumps(obj)
# ...
obj = pickle.loads(s)
```

Persistente Dictionaries: `shelve`

Ein Shelve benutzt man wie ein Dictionary, es speichert seinen Inhalt in eine Datei.

```
d = shelve.open("bla")
d["spam"] = "eggs"
d["bla"] = 1
del d["foo"]
d.close()
```

Tar-Archive: `tarfile`

Ein tgz entpacken:

```
tar = tarfile.open("spam.tgz")
tar.extractall()
tar.close()
```

Ein tgz erstellen:

```
tar = tarfile.open("spam.tgz", "w:gz")
tar.add("/home/rbreu/test")
tar.close()
```

Log-Ausgaben: `logging`

Flexible Ausgabe von Informationen, kann schnell angepasst werden.

```
import logging
logging.debug("Very special information.")
logging.info("I am doing this and that.")
logging.warning("You should know this.")
```

```
WARNING:root:You should know this.
```

- Messages bekommen einen Rang (Dringlichkeit): `CRITICAL`, `ERROR`, `WARNING`, `INFO`, `DEBUG`
- Default: Nur Messages mit Rang `WARNING` oder höher werden ausgegeben

Log-Ausgaben: `logging`

Beispiel: Ausgabe in Datei, benutzerdefiniertes Format, größeres Log-Level:

```
logging.basicConfig(level=logging.DEBUG,
  format="%(asctime)s %(levelname)-8s %(message)s",
  datefmt="%Y-%m-%d %H:%M:%S",
  filename='/tmp/spam.log', filemode='w')
```

```
$ cat /tmp/spam.log
2007-05-07 16:25:14 DEBUG    Very special information.
2007-05-07 16:25:14 INFO     I am doing this and that.
2007-05-07 16:25:14 WARNING  You should know this.
```

Es können auch verschiedene Loginstanzen gleichzeitig benutzt werden, siehe Python-Dokumentation.

Reguläre Ausdrücke: `re`

Einfaches Suchen nach Mustern:

```
>>> re.findall(r"\[.*?\]", "a[bc]g[hal]def")
['[bc]', '[hal]']
```

Ersetzen von Mustern:

```
>>> re.sub(r"\[.*?\]", "!", "a[bc]g[hal]def")
'a!g!def'
```

Wird ein Regex-Muster mehrfach verwendet, sollte es aus Geschwindigkeitsgründen compiliert werden:

```
>>> pattern = re.compile(r"\[.*?\]")
>>> pattern.findall("a[bc]g[hal]def")
['[bc]', '[hal]']
```

Reguläre Ausdrücke: `re`

Umgang mit Gruppen:

```
>>> re.findall("(\[.*?\])|(<.*?>)",
                "[hi]s<b>sdd<hal>")
[('[hi]', ''), ('', '<b>'), ('', '<hal>')]
```

Flags, die das Verhalten des Matching beeinflussen:

```
>>> re.findall("^a", "abc\nAbc", re.I|re.M)
>>> ['a', 'A']
```

- `re.I`: Groß-/Kleinschreibung ingnorieren
- `re.M`: `^` matcht am Anfang jeder Zeile (nicht nur am Anfang des Strings)
- `re.S`: `.` matcht auch Zeilenumbruch

Sockets: socket

Client-Socket erstellen und mit Server verbinden:

```
sock = socket.socket(socket.AF_INET,
                     socket.SOCK_STREAM)
sock.connect(("whois.denic.de", 43))
```

Mit dem Server kommunizieren:

```
sock.send("fz-juelich.de" + "\n")
print sock.recv(4096) # Antwort lesen
sock.close()
```

Sockets: socket

Server-Socket erstellen:

```
server_socket = socket.socket(socket.AF_INET)
server_socket.bind(("localhost", 6666))
```

Auf Client-Verbindungen warten und sie akzeptieren:

```
server_socket.listen(1)
(sock, address) = server_socket.accept()
```

Mit dem Client kommunizieren:

```
sock.send("Willkommen!\n")
# ...
```

Zusammenfassung und Ausblick

Zusammenfassung

Wir haben kennengelernt:

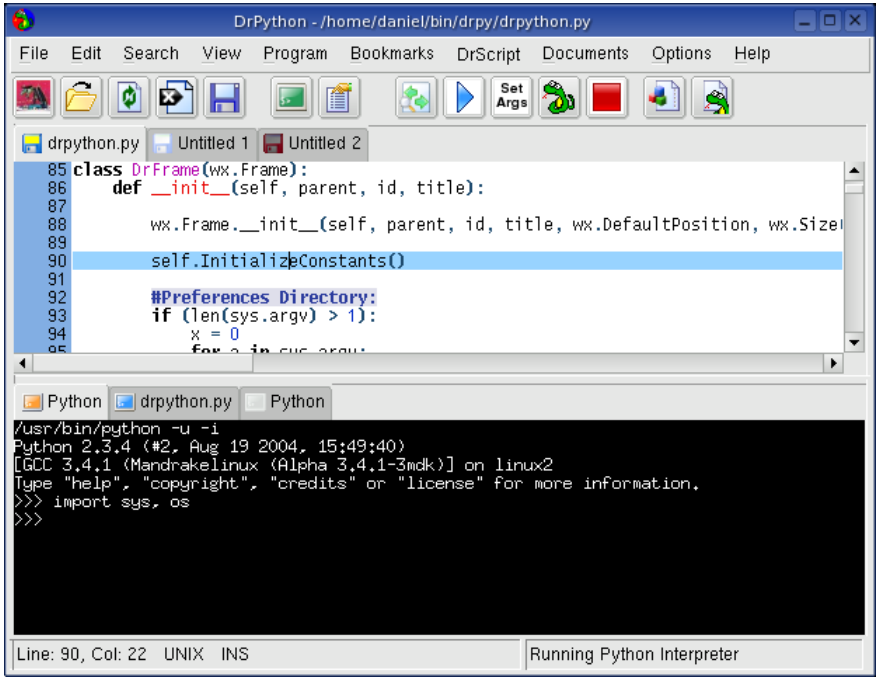
- verschiedene Datentypen (tw. „High Level“)
- die wichtigsten Statements
- Funktionsdeklaration und -Benutzung
- Module und Pakete
- Fehler und Ausnahmen, Behandlung selbiger
- objektorientierte Programmierung
- einige häufig verwendete Standardmodule

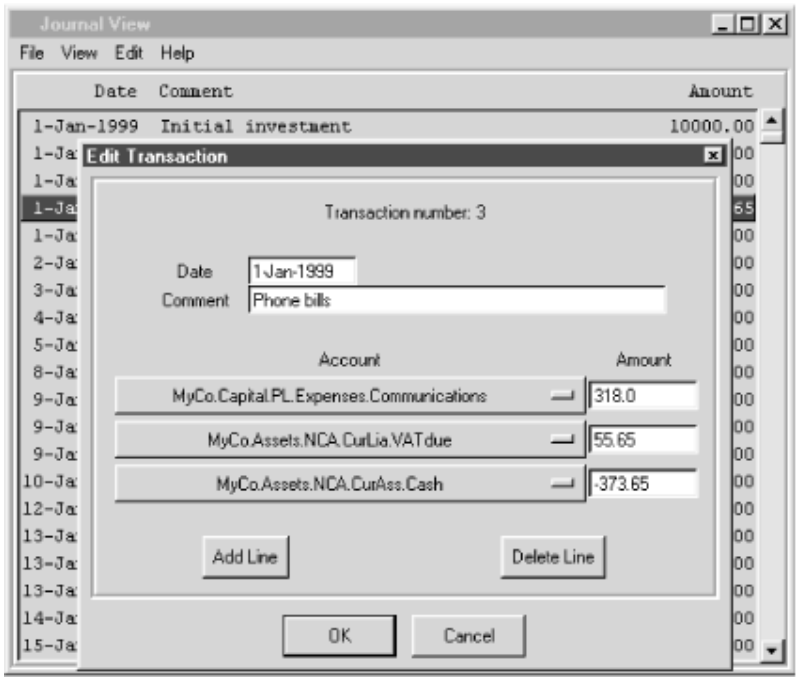
Offene Punkte

Nicht behandelte, tw. fortgeschrittene Themen:

- funktionale Techniken mit Listen: List Comprehensions, `filter()`, `map()`, `reduce()`
- Lambda-Funktionen, Funktionen mit variablen Parametern
- Iteratoren, Generatoren
- Closures, Dekoratoren (Funktionswrapper)
- Ausnutzung von Pythons Dynamik:
  `getattr`, `setattr`, Metaklassen, ...
- Weitere Standardmodule: Mail, HTML, XML, Zeit&Datum, Profiling, Debugging, Unittesting, ...
- Third Party-Module:
  GUI, Grafik, Webprogrammierung, Datenbanken, ...

Grafische Benutzeroberflächen





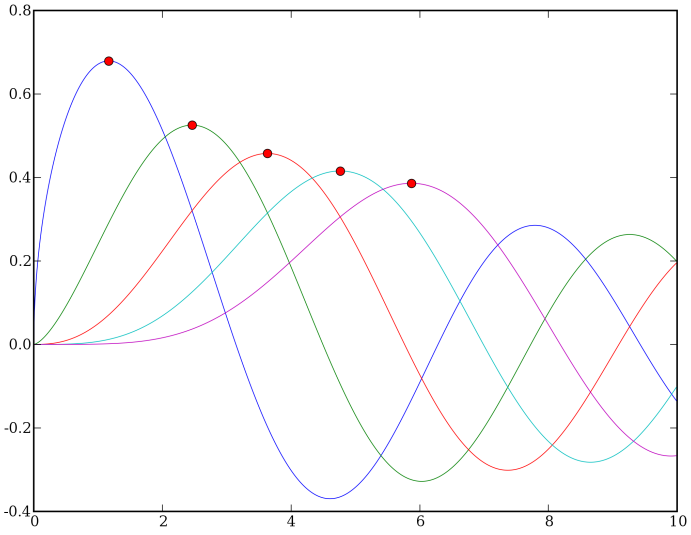
- Tk (aus Standardbibliothek)
- wxWidgets (GUI-Toolkit je nach Betriebssystem)
- GTK
- QT

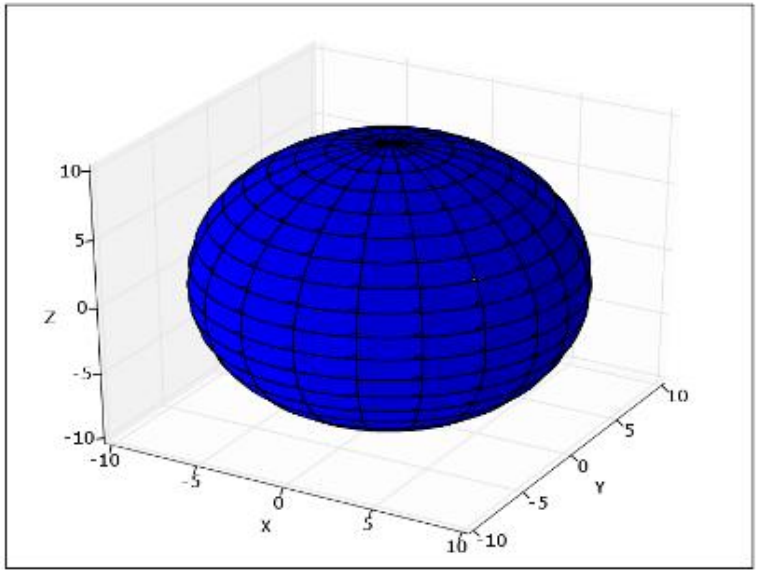
Web-Programmierung

- CGI-Scripte: Modul `cgi` aus Standardbibliothek
- Webframeworks: Django, TurboGears, Pylons, ...
- Templatesysteme: Cheetah, Genshi, Jinja, ...
- Content Management Systeme (CMS): Zope, Plone, Skeletonz, ...
- Wikis: MoinMoin, ...

NumPy + SciPy + Matplotlib = Pylab

Ein Ersatz für MatLab: Matritzenrechnung, numerische Funktionen, Plotten, ...






```
A = matrix([[1, 2], [2, 1]]); b = array([1, -1])
matshow(A)
(eigvals, eigvecs) = eig(A)
x = linalg.solve(A, b)
```

Viel Spaß mit

Einführung in Python

Rebecca Breu

Verteilte Systeme und Grid-Computing
JSC
Forschungszentrum Jülich

Oktober 2007

Inhalt — Anhang

Neues in Python 2.5

Conditional Expressions

Kurze Schreibweise für bedingte Zuweisung. Statt:

```
if zahl<0:
    s = "Negativ"
else:
    s = "Positiv"
```

kann man schreiben:

```
s = "Negativ" if zahl<0 else "Positiv"
```

Das with-Statement

Einige Objekte bieten Kontext-Management an. Damit können try... finally-Blöcke einfacher geschrieben werden:

```
from __future__ import with_statement

with open("test.txt") as f:
    for line in f:
        print line
```

Nach dem with-Block ist das Dateiobjekt stets wieder geschlossen, auch wenn im Block eine Exception auftrat.

Partielle Funktionsanwendung

```
import functools

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

add_ten = functools.partial(add, b=10)
add_ten(42)
```

Flexiblere Funktionen

Funktionsparameter aus Listen und Dictionaries

```
def spam(a, b, c, d):
    print a, b, c, d
```

Man kann positionale Parameter aus Listen erzeugen:

```
>>> args = [3, 6, 2, 3]
>>> spam(*args)
3 6 2 3
```

Man kann Keyword-Paramter aus Dictionaries erzeugen:

```
>>> kwargs = {"c": 5, "a": 2, "b": 4, "d":1}
>>> spam(**kwargs)
2 4 5 1
```

Funktionen mit beliebigen Parametern

```
def spam(*args, **kwargs):
    for i in args:
        print i
    for i in kwargs:
        print i, kwargs[i]
```

```
>>> spam(1, 2, c=3, d=4)
1
2
c 3
d 4
```

Anonyme Funktionen

```
def make_incrementor(n):
    return lambda x: x + n
```

```
>>> f = make_incrementor(42)
>>> f(0)
42
>>> f(1)
43
```

Sinnvoll, wenn einfache Funktionen als Parameter übergeben werden sollen.

Funktionale Techniken mit Listen

List Comprehension

Abkürzende Schreibweise zum Erstellen von Listen aus for-Schleifen. Statt:

```
a = []
for i in range(10):
    a.append(i**2)
```

kann man schreiben:

```
a = [i**2 for i in range(10)]
```

Map

Anwenden einer Funktion auf alle Elemente einer Liste:

```
>>> a = [1.6, 4.0, 81.0, 9.0]
>>> map(math.sqrt, a)
[1.2649110640673518, 2.0, 9.0, 3.0]
>>> map(lambda x: x * 2, a)
[3.2000000000000002, 8.0, 162.0, 18.0]
```

Wenn die Funktion mehr als einen Parameter nimmt, kann je zusätzlichem Parameter eine weitere Liste übergeben werden:

```
>>> map(math.pow, a, [2 for i in a])
[2.5600000000000005, 16.0, 6561.0, 81.0]
```

Filter

Gibt Elemente einer Liste zurück, die nach Anwendung einer Funktion wahr sind:

```
>>> a = [1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> filter(lambda x: x % 2, a)
[1, 3, 5, 7, 9]
```

Reduce

Wendet Funktion auf die ersten beiden Elemente an, dann auf das Ergebnis und das nächste Element etc.

```
>>> def add(x, y):
...     return x + y
...
>>> a = [1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> reduce(add, a)
45
```

Optionaler Startwert, der vor die Liste gesetzt wird:

```
>>> reduce(add, a, 10)
55
```

Iteratoren, Generatoren, Generator-Audrücke

Iteratoren

Was passiert, wenn `for` auf einem Objekt aufgerufen wird?

```
for i in obj:
    pass
```

- Auf `obj` wird die `__iter__`-Methode aufgerufen, welche einen Iterator zurückgibt
- Auf dem Iterator wird bei jedem Durchlauf `next()` aufgerufen
- Eine `StopIteration`-Ausnahme beendet die for-Schleife

Iteratoren

```
class Reverse:
    def __init__(self, data):
        self.data = data
        self.index = len(data)
    def __iter__(self):
        return self
    def next(self):
        if self.index == 0:
            raise StopIteration
        self.index = self.index - 1
        return self.data[self.index]
```

```
>>> for char in Reverse("spam"):
...     print char,
...
m a p s
```

Generatoren

Einfache Weise, Iteratoren zu erzeugen:

- Werden wie Funktionen definiert
- `yield`-Statement, um Daten zurückzugeben und beim nächsten `next`-Aufruf dort weiterzumachen

```
def reverse(data):
    for index in range(len(data)-1, -1, -1):
        yield data[index]
```

```
>>> for char in reverse("spam"):
...     print char,
...
m a p s
```

Generator-Audrücke

Ähnlich zu List Comprehensions kann man anonyme Iteratoren erzeugen:

```
>>> data = "spam"
>>> for c in (data[i] for i in
...               range(len(data)-1, -1, -1)):
...     print c,
...
m a p s
```

Etwas Dynamik

Dynamische Attribute

Erinnerung: Man kann Attribute von Objekten zur Laufzeit hinzufügen:

```
class Empty:
    pass

a = Empty()
a.spam = 42
a.eggs = 17
```

Und entfernen:

```
del a.spam
```

getattr, setattr

Man kann Attribute von Objekten als Strings ansprechen:

```
import math
f = getattr(math, "sin")
print f(x) # sin(x)
```

```
a = Empty()
setattr(a, "spam", 42)
print a.spam
```

Nützlich, wenn man z.B. Attributnamen aus User-Input oder Dateien liest.