

Einführung in Python

Rebecca Breu

Verteilte Systeme und Grid-Computing

JSC

Forschungszentrum Jülich

Juni 2010

Inhalt

Teil 1:

Einführung

Datentypen I

Statements

Funktionen

Input/Output

Module und Pakete

Fehler und Ausnahmen

Teil 2:

Datentypen II

Objektorientierte Programmierung

Pythons Standardbibliothek

Teil 3:

Fortgeschrittene Techniken

wxPython

Zusammenfassung und Ausblick

Einführung in Python

Rebecca Breu

Verteilte Systeme und Grid-Computing

JSC

Forschungszentrum Jülich

Juni 2010

Inhalt — Teil 1

Einführung

Datentypen I

Statements

Funktionen

Input/Output

Module und Pakete

Fehler und Ausnahmen

Einführung

Einführung

Datentypen I

Statements

Funktionen

Input/Output

Module und Pakete

Fehler und Ausnahmen

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)

→ 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
- Gegenbeispiele: PHP, 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 = "3"
print zahl, type(zahl)
print zahl + 42
```

```
3 <type 'int'>
45
3 <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 2.6 (r26:66714, Feb  3 2009, 20:52:03)
[GCC 4.3.2] on linux2
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/>

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

Einführung

Datentypen I

Statements

Funktionen

Input/Output

Module und Pakete

Fehler und Ausnahmen

Numerische Datentypen

- `int`: entspricht `long` in C
- `long`: unbegrenzter Wertebereich
- `float`: entspricht `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)`

```
>>> s = """hallo
... welt"""
>>> print s
hallo
welt
>>> print "sp\nam"
sp
am
>>> print r"sp\nam"    # or: print "sp\\namam"
sp\nam
```

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

Strings und Listen haben viel gemeinsam: Sie sind **Sequenzen**.

- Ist ein Element in s enthalten/nicht enthalten?
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 Anfang/bis Ende: s[: -i], s[i:], s[:]
- **Länge**: len(s)
- kleinstes/größtes Element: min(s), max(s)
- Zuweisungen: (a, b, c) = s
→ a = s[0], b = s[1], c = s[2]

Sequenzen

- Auch eine Sequenz: Datentyp `tuple`: `a = (1, 2.0, "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

Einführung

Datentypen I

Statements

Funktionen

Input/Output

Module und Pakete

Fehler und Ausnahmen

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

Einführung

Datentypen I

Statements

Funktionen

Input/Output

Module und Pakete

Fehler und Ausnahmen

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()  
(x, y) = 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:

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

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

Input/Output

Einführung

Datentypen I

Statements

Funktionen

Input/Output

Module und Pakete

Fehler und Ausnahmen

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("/tmp/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()
```

Python wandelt `\n` automatisch in den richtigen Zeilenumbruch um!

Module und Pakete

Einführung

Datentypen I

Statements

Funktionen

Input/Output

Module und Pakete

Fehler und Ausnahmen

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 (siehe `sys.path`):
 - dem Verzeichnis der aufrufenden Datei
 - Verzeichnissen aus der Umgebungsvariablen `PYTHONPATH`
 - installationsbedingten Verzeichnissen

```
>>> import sys
>>> sys.path
['', '/usr/lib/python26.zip',
 '/usr/lib/python2.6',
 '/usr/lib/python2.6/site-packages', ...]
```

Pakete importieren

Module können zu hierarchisch strukturierten Paketen zusammengefasst werden.

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

```
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, aber 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 der Modulfunktionalität.

Eigene Pakete

```
– numeric
  ├── __init__.py
  ├── linalg
  │   ├── __init__.py
  │   ├── decomp.py
  │   ├── eig.py
  │   └── solve.py
  └── fft
      ├── __init__.py
      └── ...
```

- Pakete sind Unterordner
- 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

Einführung

Datentypen I

Statements

Funktionen

Input/Output

Module und Pakete

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 2  
    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 verbleibenden 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()`

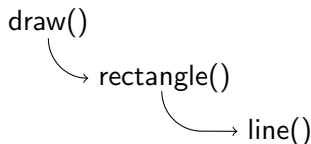
- Funktion ruft Unterfunktionen auf.
- Unterfunktion wirft Ausnahme.
- Wird Ausnahme behandelt?
- Nein: Gib Ausnahme an aufrufende Funktion weiter.

Ausnahmen in Funktionsaufrufen

draw()
 ↘
rectangle()

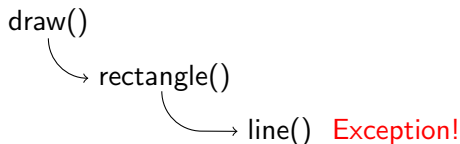
- Funktion ruft Unterfunktionen auf.
- Unterfunktion wirft Ausnahme.
- Wird Ausnahme behandelt?
- Nein: Gib Ausnahme an aufrufende Funktion weiter.

Ausnahmen in Funktionsaufrufen



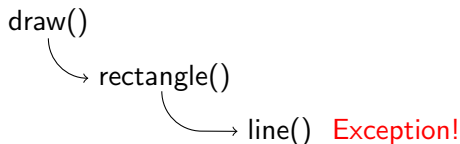
- Funktion ruft Unterfunktionen auf.
- Unterfunktion wirft Ausnahme.
- Wird Ausnahme behandelt?
- Nein: Gib Ausnahme an aufrufende Funktion weiter.

Ausnahmen in Funktionsaufrufen



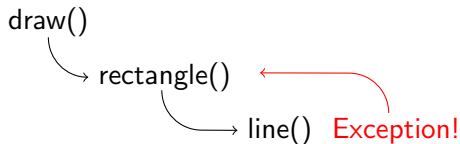
- Funktion ruft Unterfunktionen auf.
- Unterfunktion wirft Ausnahme.
- Wird Ausnahme behandelt?
- Nein: Gib Ausnahme an aufrufende Funktion weiter.

Ausnahmen in Funktionsaufrufen



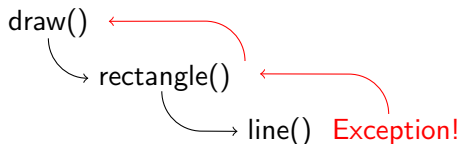
- Funktion ruft Unterfunktionen auf.
- Unterfunktion wirft Ausnahme.
- Wird Ausnahme behandelt?
- Nein: Gib Ausnahme an aufrufende Funktion weiter.

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")
```


Ausnahmen vs. if-Abfragen von Werten

Ausnahmen bevorzugen!

```
def quadrat(x):  
    if type(x) == int or type(x) == float:  
        return x ** 2  
    else:  
        return None
```

Schlecht!

- Was ist mit anderen numerischen Datentypen (komplexe Zahlen, eigene Typen)? Besser: Versuchen zu potenzieren und eventuelle Ausnahmen abfangen! → [Duck-Typing](#)
- Aufrufer der Funktion kann vergessen, den Rückgabewert zu prüfen (und weiterzureichen). Besser: Ausnahme auslösen!

Ausnahmen vs. if-Abfragen von Werten

Ausnahmen bevorzugen!

```
def quadrat(x):  
    if type(x) == int or type(x) == float:  
        return x ** 2  
    else:  
        return None
```

Schlecht!

- Was ist mit anderen numerischen Datentypen (komplexe Zahlen, eigene Typen)? Besser: Versuchen zu potenzieren und eventuelle Ausnahmen abfangen! → [Duck-Typing](#)
- Aufrufer der Funktion kann vergessen, den Rückgabewert zu prüfen (und weiterzureichen). Besser: Ausnahme auslösen!

Das with-Statement

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

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

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

In Python 2.5 ist folgender Import nötig:

```
from __future__ import with_statement
```

Viel Spaß mit



Einführung in Python

Rebecca Breu

Verteilte Systeme und Grid-Computing

JSC

Forschungszentrum Jülich

Juni 2010

Inhalt — Teil 2

Datentypen II

Objektorientierte Programmierung

Pythons Standardbibliothek

Datentypen II

Datentypen II

Objektorientierte Programmierung

Pythons Standardbibliothek

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

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

Über Dictionaries iterieren:

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

Operationen auf Dictionaries

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

Objektorientierte Programmierung

Datentypen II

Objektorientierte Programmierung

Pythons Standardbibliothek

Objektorientierte Programmierung

- Bisher: **prozedurale Programmierung**
 - Daten
 - Funktionen, die Daten als Parameter entgegennehmen und Ergebnis zurückliefern
- Alternative: Fasse zusammengehörige Daten und Funktionen zusammen zu **eigenen Datentypen**
- → 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
- → 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: `,` Rechenoperationen, `int(myobj)`, `float(myobj)`,
...
- 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, Elternklasse von Handy (rekursiv), Kamera,
Elternklasse von Kamera (rekursiv).

Private Attribute

- In Python gibt es 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

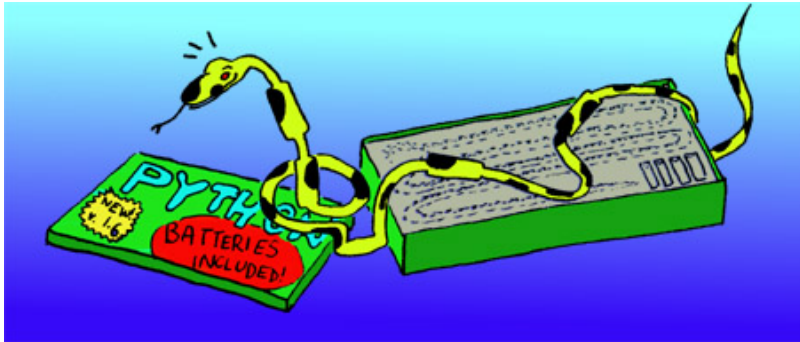
Datentypen II

Objektorientierte Programmierung

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))
0.86602540378443871
```

Exakte Dezimalzahlen: fractions

```
>>> from fractions import Fraction
>>> a = Fraction(2, 3)
>>> b = Fraction(2, 5)
>>> print float(a), float(b)
0.6666666666666666 0.40000000000000002
>>> a+b
Fraction(16, 15)
>>> a/b
Fraction(5, 3)
```

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", "-l", "mydir"])  
returncode = p.wait() # Auf Ende warten
```

Zugriff auf die Ausgabe eines Programmes:

```
p = Popen(["ls"], stdout=PIPE, stderr=STDOUT)  
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!
- → Verhindern, dass Programmteile gleichzeitig ausgeführt werden mit **Lock-Objekten**
- Locks haben genau zwei Zustände: locked und unlocked

```
lock = threading.Lock()

lock.acquire() # 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()`
- Zurücksetzen 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("-v", "--verbose",
                  action="store_true",
                  dest="verbose",
                  default=False,
                  help="verbose output")

(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
  -v, --verbose         verbose output
```

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

Konfigurationsdateien: ConfigParser

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

```
[font]
font = Times New Roman
# Kommentar (oder: ! als Kommentarzeichen)
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.bin", "wb"))
# ...
obj = pickle.load(open("bla.bin", "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()
```

Leichtgewichtige Datenbank: sqlite3

Datenbank in Datei oder im Memory, ab Python 2.5 in der stdlib.

```
conn = sqlite3.connect("bla.db")
c = conn.cursor()

c.execute("""CREATE TABLE Friends
            (vorname TEXT, nachname TEXT)""")
c.execute("""INSERT INTO Friends
            VALUES("Max", "Mueller")""")
conn.commit()
```

```
c.execute("""SELECT * FROM Friends""")
for row in c: print row

c.close(); conn.close()
```

Leichtgewichtige Datenbank: sqlite3

String-Formatter sind unsicher, da beliebiger SQL-Code eingeschleust werden kann!

```
# Never do this!  
symbol = "Max"  
c.execute("... WHERE name = '%s'" % symbol)
```

Stattdessen die Platzhalter der Datenbank-API benutzen:

```
c.execute("... WHERE name = ?", symbol)
```

```
f = (("Alice", "Meier"), ("Bob", "Schmidt"))  
for item in friends:  
    c.execute("""INSERT INTO Friends  
                VALUES (?,?)""", item)
```

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, feineres 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 ignorieren
- re.M: ^ bzw. \$ matchen am Anfang/Ende 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("-T dn 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")  
# ...
```

XML-RPC-Client: xmlrpclib

- XML-RPC: [Remote Procedure Call](#) via XML und HTTP
- unabhängig von Plattform und Programmiersprache

```
import xmlrpclib

s = xmlrpclib.Server("http://localhost:8000")
print s.add(2,3)
print s.sub(5,2)
```

Konvertierungen für die gängigen Datentypen geschehen automatisch: Booleans, Integer, Floats, Strings, Tupel, Listen, Dictionaries mit Strings als Keys, ...

XML-RPC-Server: SimpleXMLRPCServer

```
from SimpleXMLRPCServer import SimpleXMLRPCServer

# Methoden, die der Server zur Verfuegung
# stellen soll:
class MyFuncs:
    def add(self, x, y):
        return x + y
    def sub(self, x, y):
        return x - y

# Erstelle und starte Server:
server = SimpleXMLRPCServer(("localhost", 8000))
server.register_instance(MyFuncs())
server.serve_forever()
```

Viel Spaß mit



Einführung in Python

Rebecca Breu

Verteilte Systeme und Grid-Computing

JSC

Forschungszentrum Jülich

Juni 2010

Inhalt — Teil 3

Fortgeschrittene Techniken

wxPython

Zusammenfassung und Ausblick

Fortgeschrittene Techniken

Fortgeschrittene Techniken

wxPython

Zusammenfassung und Ausblick

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"
```

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-Parameter 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
```

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)]
```

Mit Bedingung:

```
a = [i**2 for i in range(10) if i != 4]
```

Anonyme Funktionen: Lambda

```
>>> f = lambda x, y: x + y
>>> f(2, 3)
5
>>> (lambda x: x**2)(3)
9
```

Nützlich, wenn einfache Funktionen als Parameter übergeben werden sollen.

```
l = ["alice", "Bob"]
l.sort()
l.sort(lambda a,b: cmp(a.upper(), b.upper()))
```


Map

Anwenden einer Funktion auf alle Elemente einer Liste:

```
>>> li = [1, 4, 81, 9]
>>> map(math.sqrt, li)
[1.0, 2.0, 9.0, 3.0]
>>> map(lambda x: x * 2, li)
[2, 8, 162, 18]
```

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

```
>>> map(math.pow, li, [1, 2, 3, 4])
[1.0, 16.0, 531441.0, 6561.0]
```

Filter

Wie Map, jedoch enthält die Ergebnisliste nur die Elemente, welche wahr sind:

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

Zip

Zusammenfügen mehrerer Sequenzen zu einer Liste von Tupeln:

```
>>> zip("ABC", "123")  
[('A', '1'), ('B', '2'), ('C', '3')]  
>>> zip([1, 2, 3], "ABC", "XYZ")  
[(1, 'A', 'X'), (2, 'B', 'Y'), (3, 'C', 'Z')]
```

defaultdict

Ein Dictionary, welches Defaultwerte für nicht vorhandene Schlüsselwörter erzeugen kann:

```
from collections import defaultdict
haeufigkeiten = defaultdict(lambda: 0)

for c in "Hallo Welt!":
    haeufigkeiten[c] += 1
```

Parameter (optional): Funktion zum Erzeugen der Defaultwerte

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 element in data[::-1]:  
        yield element
```

```
>>> 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 (elem for elem in data[::-1]):
...     print c,
...
m a p s
```


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.

wxPython

Fortgeschrittene Techniken

wxPython

Zusammenfassung und Ausblick

Grafische Benutzeroberflächen (GUIs)

Verbreitete GUI-Toolkits mit Bindings für (u.A.) Python:

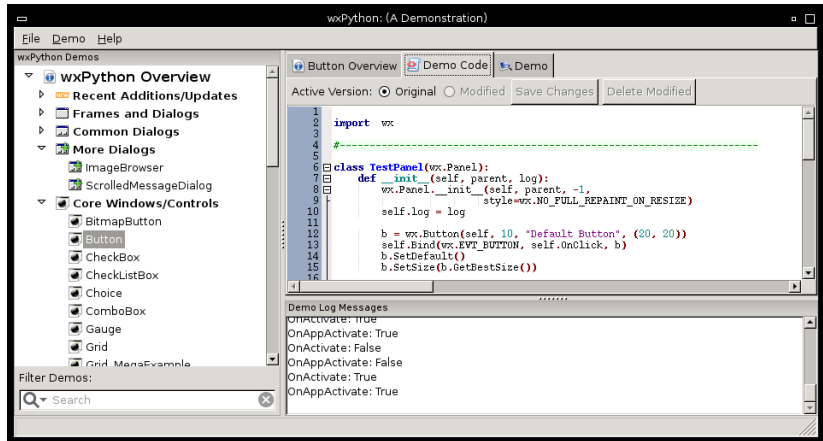
- **Tk**: In Pythons Standardbibliothek, simpel (ungeeignet für komplexe Anwendungen), veraltetes Aussehen
- **GTK**: z.B. Gnome Desktop, GIMP, Eclipse, ...
- **QT**: KDE Desktop, Skype, Scribus, ...

Alle werden auf den gängigen Betriebssystemen unterstützt.

- **wxWidgets**: Benutzt Windows-, Mac OS-Bibliotheken oder GTK → Look and Feel des jeweiligen Betriebssystems

Die wxPython-Demo

/usr/share/doc/wx2.8-examples/examples/wxPython/demo.py



Kurze Beschreibung aller Features mit Live-Demo und
Beispiel-Code

Hello World

```
import wx

class MainFrame(wx.Frame):
    def __init__(self):
        wx.Frame.__init__(self, parent=None,
                           title="Hello World")
        self.Show(True)

app = wx.PySimpleApp()
frame = MainFrame()
app.MainLoop()
```

Erzeugt ein leeres Fenster mit Titel „Hello World“.

Die Basis: Application und Top Level Windows

Application:

- Kern eines wx-Programms, betreibt die **Hauptschleife**
- Hauptschleife verarbeitet alle **Events** (Mausbewegung, Tastaturanschlag, ...)
- `PySimpleApp`: Für einfache Anwendungen mit nur einem Top Level Window

Zur Application gehört mindestens ein **Top Level Window**:

- Präsentiert dem Anwender die wichtigsten Daten und Kontrollelemente
- Wird das letzte Top Level Window geschlossen, beendet sich die Application (die Hauptschleife wird verlassen)

Das allgemeinste Widget: wx.Frame

```
wx.Frame(parent, id=-1, title="",  
         pos=(-1,-1), size=(-1,-1), ...)
```

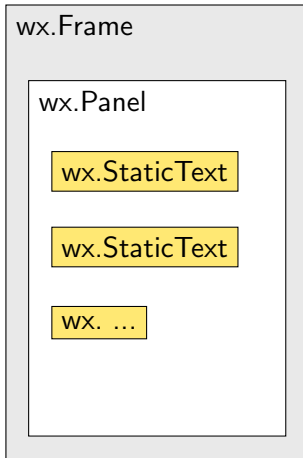
- **parent**: Ist None für Top Level Windows
- **id**: Integer; Automatische Generierung mit -1 (zu bevorzugen)
- **title**: Fenstertitel, wird in Titelleiste angezeigt
- **pos**: Integer-Tupel (x, y); (-1, -1) lässt das unterliegende System entscheiden
- **size**: Integer-Tupel (width, height); (-1, -1) lässt das unterliegende System entscheiden

Widgets in ein Frame einfügen

Etwas Text in unserem Fenster:

```
class MainFrame(wx.Frame):  
    def __init__(self):  
        wx.Frame.__init__(self, parent=None,  
                           title="Hello World")  
        panel = wx.Panel(parent=self)  
        wx.StaticText(parent=panel,  
                       label="How are you?")  
        self.Show(True)
```

Widgets in ein Frame einfügen



- **Panel:** Container, welcher beliebig viele weitere Widgets enthalten kann.
- Parent-Beziehungen legen fest, welches Widget in welchem Widget dargestellt wird

Nicht-editierbarer Text: StaticText

```
wx.StaticText(parent, id=-1, label="",  
              pos=(-1,-1), size=(-1,-1),  
              style=0, ...)
```

- **label**: Der darzustellende Text
- **pos** bezieht sich auf die Position innerhalb des Parent-Widgets
- **style**: `wx.ALIGN_CENTER`, `wx.ALIGN_LEFT`, `wx.ALIGN_RIGHT`
- Auch mehrzeiliger Text möglich
- Einige Methoden:
 - `SetLabel`: Text nachträglich ändern
 - `SetForegroundColour`, `SetBackgroundColour`
 - `SetFont`

Auf Benutzeraktionen reagieren

```
class MainFrame(wx.Frame):  
    def __init__(self):  
        wx.Frame.__init__(parent=None)  
        panel = wx.Panel(parent=self)  
        button = wx.Button(parent=panel,  
                             label("&Click me"))  
        self.Bind(wx.EVT_BUTTON, self.on_button,  
                   button)  
        self.Show(True)  
  
    def on_button(self, evt):  
        print "You pressed the button!"
```

Der Button kann mit Alt+C „geklickt“ werden (wg. &C...)

Ereignisgesteuerte Programmierung

- Herkömmliche Programme laufen linear ab
- GUI-Programme: Anwender kann Bedienelemente zu beliebiger Zeit in beliebiger Reihenfolge bedienen
- GUI-Programm *reagiert* auf den Anwender
- → **Hauptschleife** wartet auf **Events** und leitet diese an passende **Event-Handler** weiter

```
self.Bind(wx.EVT_BUTTON, self.on_button,  
          button)
```

MainFrame soll alle Button-Events vom Widget button mit der Methode onButton behandeln.

Events und die Widget-Hierarchie

```
class MainFrame(wx.Frame):  
    def __init__(self):  
        ...  
        self.Bind(wx.EVT_BUTTON ,  
                   self.on_buttonF, button)  
        button.Bind(wx.EVT_BUTTON ,  
                    self.on_buttonB, button)  
  
    def on_buttonF(self, evt):  
        print "You pressed the button!"  
  
    def on_buttonB(self, evt):  
        print "You pressed me!"
```

Events und die Widget-Hierarchie

- Widget generiert Event
- Hat das Widget passenden Event-Handler?
 - ja: behandle Event
 - nein: Sende Event an das Parent-Widget
- Hat das Parent-Event passenden Event-Handler? ...

→ Nur `onButtonB` wird ausgeführt!

Behandeltes Event weiter propagieren mit `Skip`:

```
def on_buttonB(self, evt):  
    print "You pressed me!"  
    evt.Skip()
```

→ `onButtonB` und `onButtonF` werden ausgeführt

Widgets anordnen mit Sizern

Widgets per Hand anordnen hat Nachteile:

- Unpraktikabel für viele Widgets
- Widgets haben für unterschiedliche Standard-Schriften unterschiedliche Ausmaße
- Muss angepasst werden, wenn die Fenstergröße verändert wird

→ Sizer

- Nehmen mehrere Widgets auf
- Ordnen sie nach vorgegebenen Regeln in einem Panel an
- Ordnen sie automatisch neu

Widgets anordnen mit Sizer

```
# Sizer erstellen:
panel = wx.Panel(parent=self)
box = wx.BoxSizer(wx.HORIZONTAL)
panel.SetSizer(box)

# Widgets einfüegen:
button = wx.Button(parent=panel, label="Spam")
box.Add(button, proportion=1, flag=wx.CENTER)
```

Es können beliebig viele Widgets mit `Add` in den Sizer eingefügt werden, jedes mit eigenen Platzierungs-Regeln.

Widgets anordnen mit Sizern

```
Add(widget, proportion=0, flag=0, border=0)
```

- **proportion**: Verhältnis, in dem der freie Platz zwischen Widgets aufgeteilt wird (nur bei BoxSizern.)
- **flag**: Bestimmt Ausrichtung dieses Widgets und seines Rahmens:
 - `wx.ALIGN_TOP`, `wx.ALIGN_BOTTOM`, `wx.ALIGN_LEFT`, `wx.ALIGN_RIGHT`, `wx.ALIGN_CENTER`: Ausrichtung des Widgets
 - `wx.ALIGN_EXPAND`: Widget wird gestreckt
 - `wx.ALL`, `wx.TOP`, `wx.BOTTOM`, `wx.LEFT`, `wx.RIGHT`: An welchen Seiten ein Rahmen eingefügt werden soll
 - Flags können mit `|` kombiniert werden:
`flag=wx.ALIGN_CENTER|wx.ALL`
- **border**: Rahmen (Freiraum) um das Widget in Pixeln

BoxSizer und GridSizer

```
BoxSizer(wx.HORIZONTAL) # oder wx.VERTICAL
```

BoxSizer: Widgets werden in einer horizontalen oder vertikalen Reihe angeordnet.

```
GridSizer(rows, cols, hgap, vgap)
```

GridSizer: Widgets werden in einem regelmäßigen Gitter angeordnet.

- **rows, cols:** Anzahl der Zeilen und Spalten an Widgets
- **hgap, vgap:** Horizontaler/Vertikaler Abstand zwischen Widgets in Pixeln

FlexGridSizer und GridBagSizer

```
grid = wx.FlexGridSizer(3, 3, 5, 5)
grid.AddGrowableRow(idx=2, proportion=1)
grid.AddGrowableCol(idx=2, proportion=1)
```

FlexGridSizer: Wie GridSizer, aber:

- Zeilen/Spalten mit unterschiedlichen Höhen/Breiten möglich
- Zeilen/Spalten können flexibel in Höhen/Breiten wachsen, ähnlich BoxSizer

GridBagSizer:

- Bei **Add** kann die Zelle angegeben werden, in welche das Widget eingefügt wird
- Widgets können über mehrere Zellen gehen

Texteingaben mit TextCtrl

```
TextCtrl(parent, id=-1, value="", pos=(-1,-1),  
          size=(-1,-1), style=0, ...)
```

- automatisch Standard-Tastaturkürzel: Ctrl-x, Ctrl-v, ...
- **value**: Der anfängliche Inhalt des Textfeldes
- **style**:
 - wx.TE_CENTER, wx.TE_LEFT, wx.TE_RIGHT: Ausrichtung des Textes
 - wx.TE_MULTILINE: Mehrzeilige Texteingabe zulassen
 - wx.TE_PASSWORD: Text wird durch Sternchen verborgen
 - ...

Texteingaben mit TextCtrl

Einige Methoden von TextCtrl:

- `GetValue`, `SetValue`: Textinhalt lesen/setzen
- `GetStringSelection`: Den markierten Textbereich lesen
- `Clear`: Textinhalt löschen

```
txt = wx.TextCtrl(panel, value="Default",  
                  style=wx.TE_MULTILINE|wx.TE_CENTER)  
  
txt.SetValue("Neuer Default")  
print txt.GetStringSelection()
```

Auswahl mit Checkboxes

```
check = wx.CheckBox(parent=panel ,  
                    label="Check &me")  
self.Bind(wx.EVT_CHECKBOX , self.on_checkbox ,  
          check)  
print check.IsChecked()
```

- Statusabfrage mit der Methode IsChecked
- Betätigung der Checkbox löst wx.EVT_CHECKBOX aus
- Liste von Checkboxes: Voneinander unabhängige Checkboxes, es können beliebig viele Boxen ausgewählt werden

Einzel-Auswahl mit RadioBox

Aus einer Liste von Optionen kann nur eine ausgewählt werden.

```
items = ["One", "Two", "Three"]
radio = wx.RadioButton(parent=panel,
                        choices=items,
                        label="Your choice")
```

- Statusabfrage mit der Methode GetStringSelection
- Betätigung der Checkbox löst wx.EVT_RADIOBOX aus
- Mit zusätzlichen Parametern des Konstruktors kann Anzahl Zeilen/Spalten bestimmt werden:
 - **majorDimension**: Anzahl Zeilen oder Spalten
 - **style**: wx.RA_SPECIFY_COLS oder wx.RA_SPECIFY_ROWS

Auswahl mit ListBox

```
items = ["One", "Two", "Three"]  
list = wx.ListBox(parent=panel,  
                  choices=items,  
                  style=wx.SINGLE)
```

- Statusabfrage mit der Methode GetStringSelection oder GetSelections
- Betätigung der Listbox löst wx.EVT_LISTBOX aus
- Verschiedene Styles:
 - wx.LB_SINGLE: Anwender kann nur eine Option auf einmal auswählen
 - wx.EXTENDED: Anwender kann einen Bereich auswählen
 - wx.MULTIPLE: Anwender kann beliebig viele Optionen auswählen

Modale Dialoge

Modaler Dialog: Kleines Popup-Fenster, welches die restliche Anwendung blockiert.

```
msg = wx.MessageDialog(parent=panel ,
                        message="Are you ok?",
                        caption="Question",
                        style=wx.YES_NO|wx.ICON_QUESTION)

value = msg.ShowModal()
if value == wx.ID_YES:
    print "That's fine!"
else:
    print "I'm sorry."
```

MessageDialog

Stellt ein (optionales) Icon, einen Text und Buttons dar.

```
wx.MessageDialog(parent, message,  
                 caption="Message box",  
                 style=wx.OK | wx.CANCEL, pos=(-1, -1))
```

Style-Optionen:

- `wx.YES_NO`, `wx.OK`, `wx.CANCEL`: Dargestellte Buttons
- `wx.ICON_ERROR`, `wx.ICON_INFORMATION`,
`wx.ICON_QUESTION`: Dargestelltes Icon

TextEntryDialog

Für kurze Eingaben vom Anwender.

```
dlg = wx.TextEntryDialog(parent=panel,
                          message="Your name?",
                          caption="Question", style=wx.OK)

value = dlg.ShowModal()
if value == wx.ID_OK:
    print dlg.GetValue()
```

Weitere Dialoge:

- `wx.PasswordEntryDialog`
- `wx.SingleChoiceDialog` (Stellt eine `ListBox` dar)

FileDialog

```
dlg = wx.FileDialog(parent=panel,
                    message="Choose a file",
                    wildcard="Python|*.py|All|*",
                    style=wx.OPEN)

value = dlg.ShowModal()
if value == wx.ID_OK:
    print dlg.GetPath()
```

- Wichtigste Style-Optionen: `wx.OPEN` oder `wx.SAVE`
- Ähnlich: `DirDialog` für Verzeichnisse

Menüs und Menüleiste: MenuBar

Vorgehensweise für eine vollständige Menüleiste:

- **MenuBar** erstellen und dem Frame zuordnen
- Einzelne **Menüs** erstellen und der MenuBar hinzufügen
- **Items** zu den einzelnen Menüs hinzufügen
- **Event Handler** erstellen und den Items zuordnen

```
class MainFrame(wx.Frame):  
    def __init__(self):  
        wx.Frame.__init__(self, parent=None)  
        menubar = wx.MenuBar()  
        self.SetMenuBar(menubar)
```

Menüs in die Menüleiste einfügen

```
menu = wx.Menu()
menubar.Append(menu, "&Menu")

blah = menu.Append(-1, "&Blah\tCtrl+B",
                    "Some help text")
self.Bind(wx.EVT_MENU, self.on_blah, blah)
```

- **Mnemonic Shortcuts** mit & im Item-Namen
- **Accelerator Shortcuts** mit \t im Item-Namen
- **Hilfetext** wird in der Statuszeile angezeigt
- **AppendSeparator()** zum Unterteilen der Items mit einer Linie

Statuszeile: StatusBar

```
class MainFrame(wx.Frame):  
    def __init__(self):  
        wx.Frame.__init__(self, parent=None)  
        self.CreateStatusBar()  
        self.SetStatusText("Hallo Welt")
```

- Hilfetext der Menü-Items wird automatisch angezeigt
- Setzen des angezeigten Textes mit `SetStatusText`

Weitere Möglichkeiten

- Toolbars mit `wx.ToolBar`
- Tabs und gesplittete Fenster: `wx.NoteBook`, `wx.SplitterWindow`
- Flexible Listen und Tabellen: `wx.ListCtrl`, `wx.grid.Grid`
- Baumdarstellungen: `TreeCtrl`
- Schriften und Schrift-Auswahldialoge: `wx.Font`, `wx.FontDialog`
- Farben und Farb-Auswahldialoge: `wx.Colour`, `wx.ColourDialog`
- Umgang mit Bildern und Grafik; Zeichnen
- ... → wxPython-Demo

Dokumentation:

- <http://www.wxpython.org/onlinedocs.php>
- Buch: *wxPython in Action*

Bekannte wxPython-Anwendungen

- [wxGlade](#): GUI-Designer für wxWidgets
- [Boa Constructor](#): Python-IDE und GUI-Designer für wxWidgets
- [SPE](#): Python-IDE und GUI-Designer für wxWidgets
- [DrPython](#): Python-IDE
- [BitTorrent](#): Bittorrent-Client
- [wxRemind](#): Graphisches Frontend für den Linux-Kalender Remind

Zusammenfassung und Ausblick

Fortgeschrittene Techniken

wxPython

Zusammenfassung und Ausblick

Zusammenfassung

Wir haben kennengelernt:

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

Offene Punkte

Nicht behandelte, tw. fortgeschrittene Themen:

- Closures, Dekoratoren (Funktionswrapper)
- Metaklassen
- Weitere Standardmodule: Mail, WWW, XML, Zeit&Datum, ... → <http://docs.python.org/lib/modindex.html>
- Profiling, Debugging, Unittesting
- Extending und Embedding: Python & C/C++
→ <http://docs.python.org/ext/ext.html>
- Third Party-Module: Grafik, Webprogrammierung, Datenbanken, ... → <http://pypi.python.org/pypi>

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, ...



The MoinMoin Wiki Engine

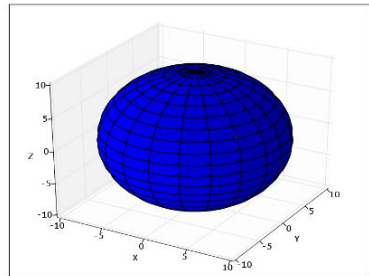
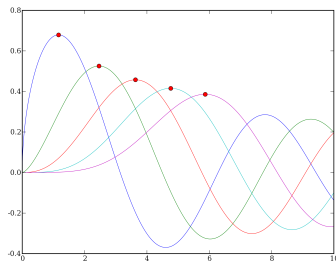
Overview

[MoinMoin](#) is an advanced, easy to use and extensible [WikiEngine](#) with a large community of users. Said in a few words, it is about collaboration on easily editable web pages. MoinMoin is Free Software licensed under the [GPL](#).

- If you want to learn more about wiki in general, first read about [WikiWikiWeb](#), then about [WhyWikiWorks](#) and the [WikiNature](#).
- If you want to play with it, please use the [WikiSandBox](#).
- [MoinMoinFeatures](#) documents why you really want to use MoinMoin rather than another wiki engine.
- [MoinMoinScreenShots](#) shows how it looks like. You can also browse *this* wiki or visit some other [MoinMoinWikis](#).

NumPy + SciPy + Matplotlib = PyLab

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



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

Und mehr...

- **ipython**: Eine komfortablere Python-Shell
- **Scons**: Ein Build-System ähnlich Make/autoconf/automake oder Ant
 - basiert auf Python, nutzt Python-Syntax
 - Built-in-support für viele Programmiersprachen, LaTeX, ...
 - plattformunabhängig
- Python und andere Programmiersprachen:
 - Jython: Python-Code in der Java VM ausführen
 - Ctypes: C-Libraries mit Python ansprechen (ab 2.5 in der stdlib)
 - SWIG: C- und C++ -Libraries mit Python ansprechen
- **PIL**: Python Imaging Library für Bildmanipulation

PyCologne



PyCologne: Python User Group Köln

- Trifft sich jeden zweiten Mittwoch im Monat am Rechenzentrum der Uni Köln
- URL:
<http://wiki.python.de/pyCologne>

Viel Spaß mit

