Diese HTML-Datei enthält spezielle CSS2-Anweisungen, die für den Präsentationsmodus des Web-Browsers Opera erstellt wurden. Sofern sie diesen Browser benutzen, können sie mit der F11- Taste in diesen Modus (und wieder zurück) schalten. Die Präsentation wurde für eine Auflösung von 1280x1024 Bildpunkten erstellt.

die Programmiersprache

Ein Überblick

- Vortragsreihe "Chaos-Seminar"
- Veranstalter: CCC, Erfa-Kreis Ulm
  - http://www.ulm.ccc.de/
  - mail@ulm.ccc.de
  - Montagstreff: Infos und Anfahrtsskizze auf der Webseite
- Referent: Markus Schaber
  - http://www.schabi.de/
  - markus.schaber@ulm.ccc.de
- Vortrag ist im HTML-Format online
  - http://www.ulm.ccc.de/~schabi/python/

Inhaltsübersicht

Kein Einsteigerkurs, sondern ein kurzer Rundumschlag...

- Hello World
- Paradigmen
  Imperativ - Prozedural - Modular - Objektorientiert - Funktional
- Elemente und Syntax
  Einrückung und Zeilen - Strings - Stringformatierungen - String Encoding/Decoding - Docstrings - Zahlen - Tupel und Listen - Dictionaries - Exceptions - Sequenzen und Iteration - Generatoren - List Comprehension - variable Parameterlisten - Erweiterungen
- Verfügbare Module
  "Batteries Included" - GUI - und und und...
- Drumherum
  cPython - Jython - Embedding - Erweiterungen mit C - Schnittstellen, Wrapper, Wandler

Hello World

- Wohl das "klassische" Programmierbeispiel
- Quelltext:

```
#!/bin/env python
print "hello, world!"
```

- Ausgabe:

```
hello, world!
```

- Nachvollziehbar auch in der ineraktiven Shell

Imperatives Programmieren

- Hauptprinzip: Aneinandergereihte Anweisungen
- z. B. ursprüngliches Basic
- Strukturelemente: Sprünge, Schleifen, Verzweigungen
- In Python: Keine Sprünge!
- Beispiel:

```
i = 1
while i<10:
        i *= 1.5
        print i
```

- Ausgabe:

```
1.5
2.25
3.375
5.0625
7.59375
11.390625
```

Prozedurales Programmieren

- Strukturelement: Funktionen und Prozeduren
- Vorläufer: Gosub in Basic
- Auch Rekursion möglich
- Beispiel: Fakultätsfunktion fakmod.py

```
def fak(x):
  if x<=1:
    return 1
  else:
    return x*fak(x-1)
```

- Ausprobieren Interaktiv:

```
>>> fak(3)
6
>>> fak(9)
362880
>>> fak(42)
1405006117752879898543142606244511569936384000000000L
```

Modulares Programmieren

- Große Programme in Module aufteilen

- Verbinden dieser Teile zu einem ganzen Programm
- In Python: Import von Modulen zur Laufzeit (ähnlich Java-klassen)
- Beispiel: Fakultätsfunktion von vorhin

```
import fakmod
print fakmod.fak(3)
```

- Beispiel2: import für Fortgeschrittene

```
from fakmod import fak as blubber
print blubber(3)
```

- Package-System ähnlich dem von Java

```
from mathe import fakmod
print fakmod.fak(3)
```

oder

```
from mathe.fakmod import fak
print fak(3)
```

Objektorientiertes Programmieren

- Zur Zeit *das* Programmierparadigma
- Vorläufer: z. B. SmallTalk
- Vorbild: Natur, alles besteht aus Objekten, die interagieren
- Beispiel: Auto-Klasse in objekt.py

```
class Auto:
  def __init__(self):
    self.raeder = 4*["okay"]
  def nagelfahren(self, rad):
    self.raeder[rad]="platt"
  def status(self):
    print self.raeder
```

- Benutzung:

```
>>> a = Auto()
>>> a.status()
['okay', 'okay', 'okay', 'okay']
>>> a.nagelfahren(3)
>>> a.status()
['okay', 'okay', 'okay', 'platt']
```

Objektorientiertes Programmieren (2)

- In Python sind alles Objekte!
- `a+b` ist in Wirklichkeit `a.__add__(b)`
- Auch Zahlen, Funktionen und Module sind Objekte
- Garbage-Collector sorgt für Ordnung
- Klassen-Objekte sind vollkommen dynamisch:

```
from objekt import Auto
>>> a=Auto()
>>> a.hallo
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
```

```
AttributeError: Auto instance has no attribute 'hallo'
>>> a.hallo="Gruss an alle"
>>> a.hallo
'Gruss an alle'
```

Objektorientiertes Programmieren (3)

- Sogar "Schmutzige Tricks" sind in Python möglich:
- Beispiel: Dynamische Attribute und Klassenzuweisung

```
>>> a
<objekt.Auto instance at 0x12b970>
>>> class boot:
...     def rudern(self):
...             print "plätscher..."
...
>>> a.__class__=boot
>>> a
<__main__.boot instance at 0x12b970>
>>> a.rudern
<bound method boot.rudern of <__main__.boot instance at 0x12b970>>
>>> a.rudern()
plätscher...
>>> a.nagelfahren()
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
AttributeError: boot instance has no attribute 'nagelfahren'
>>> a.__class__=Auto
>>> a..status()
['okay', 'okay', 'okay', 'okay']
```

Funktionales Programmieren

- Prinzip: Alles sind Funktionen
- In Python verschiedene Mechanismen:
  - Funktionen sind Objekte
  - apply(), map(), reduce(), filter() sind builtin-Funktionen
  - Lambda-Operator:

```
>>>f = lambda x,y: x+y
>>> f
<function <lambda> at 0xe5440>
>>> f(1,2)
3
```

  - Rufbare Objekte

```
>>> a = Auto()
>>> a()
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
AttributeError: Auto instance has no __call__ method
>>> a.__call__ = a.status
>>> a()
['okay', 'okay', 'okay', 'okay']
```

Einrückung und Zeilen

- Statement in der Regel eine Zeile lang
- Mehrere Statements durch `;`
- Kommentare von # bis physikalischem Ende
- Zeilenzusammenfügungen
  - Explizit durch `\`
  - Implizit durch Klammern
  - Mehrzeilige Strings beinhalten Zeilenumbruch
- Blockstruktur durch Einrückung
  - Keine Geschweiften Klammern `{}` oder `begin ... end`
  - Leerzeichen, Kommentare und fortgesetzte Zeilen werden ignoriert
- Beispiel indent.py

```
import string
text = """Dies ist ein
       mehrzeiliger
String"""
for zeichen in text:
  code = ord(zeichen)
  print zeichen, ord(zeichen), string.upper(
    zeichen)
```

Strings

- Unicode-Support existiert
- Aufeinanderfolgende String-Literale werden konkateniert
- Maskierung durch `", ', '''` oder `"""`
- Beispiele:

```
>>>a = "Ein Text"
>>>b = """Ein
...Text, der über
...mehrere
...Zeilen
...geht"""
>>> c = """enthält ' und ''', und """ '''sogar """ und "'''
>>> print c
enthält ' und ''', und sogar """ und "
>>> d = u"Unicode \u0042"
>>> print d
Unicode B
>>> e = r"string\0x123"
>>> print e
'string\\0x123'
```

Stringformatierungen

- Formatstrings in C sind sehr unsicher
- In Python: `%`-Operator
- Beispiele:

```
>>> count = 2
>>> language = 'Python'
>>> print '%(language)s has %(count)03d quote types.' % vars()
Python has 002 quote types.

>>> print "%+E" % 3
```

```
+3.000000E+00

>>> print "%017i" % 3
00000000000000003
```

String Encoding / Decoding

- Ursprünglich für Unicode-Codierungen
- Inzwischen auch für normale Strings
- Beispiele:

```
>>> "QNH".encode("rot13")
'DAU'

>>> "Chaos Seminar".encode("uu")
'begin 666 <data>\n-0VAA;W,@4V5M:6YA<@  \n \nend\n'

>>> "Chaos Seminar".encode("base64")
'Q2hhb3MgU2VtaW5hcg==\n'

>>> "Chaos Seminar".encode("Quoted-Printable")
'Chaos=20Seminar'

>>> u"abcäöü".encode("utf-8")
'abc\xc3\xa4\xc3\xb6\xc3\xbc'
```

Docstrings

- Ähnlicher Ansatz wie JavaDoc
- Werden implizit dem __doc__-Attribut zugewiesen
- Beispiel docstring.py:

```
""" Datei docstring.py
Demonstriert, wie man Docstrings erstellt und anwendet"""

def funktion(parameter):
  """demo-Funktion, die einfach den Parameter ausgibt"""
  print parameter
```

- Anwendung:

```
>>> import docstring
>>> docstring.__doc__
' Datei docstring.py\nDemonstriert, wie man Docstrings erstellt und anwendet'
>>> docstring.funktion.__doc__
'demo-Funktion, die einfach den Parameter ausgibt'
>>> help(docstring.funktion)
Help on function funktion in module docstring:

funktion(parameter)
    demo-Funktion, die einfach den Parameter ausgibt
```

Docstrings (2)

- Fortsetzung Anwendung docstring.py:

```
>>> help(docstring)
```

```
Help on module docstring:

NAME
    docstring

FILE
    /users/student1/s_mschab/public_html/pythonsem/examples/docstring.py

DESCRIPTION
    Datei docstring.py
Demonstriert, wie man Docstrings erstellt und anwendet

FUNCTIONS
    funktion(parameter)
        demo-Funktion, die einfach den Parameter ausgibt

DATA
    __file__ = 'docstring.py'
    __name__ = 'docstring'
```

Zahlen

- Sind in Python auch Objekte
- Vordefinierte Typen:
  - Ganze Zahlen und Lange ganze Zahlen`5`
  - Fließkommazahlen`5.0`
  - Imaginäre Zahlen`5j`
  - Komplexe Zahlen`5+5j`
- Division gerade in der Umstellung
  - "altes" `/` dividiert "klassisch"
  - "neues" `/` dividiert "real"
  - `//` dividiert "ganzzahlig"
- Erweiterung "uncertain.py" existiert

Tupel und Listen

- Tupel sind "Konglomerate" beliebiger Objekte
- Sie sind nach der Erstellung unveränderbar
- Listen sind dynamische Arrays
- Beide sind indizierbar und iterierbar
- Auch in Zuweisungen und Funktionsrückgaben verwendbar
- Beispiele:

```
>>> tupel = (2,3,4); liste=[6,7,8]
>>> erg = zip(tupel, liste)
>>> print erg
[(2, 6), (3, 7), (4, 8)]
>>> a,b,c = erg
>>> print a
(2, 6)
>>> a = divmod(2,3)
>>> print a
(0, 2)
>>> wert,rest = divmod(2,3)
>>> print wert
0
```

Dictionaries

- Hashtables, speichern Key/Value-paare
- Effizient implementiert
- intern benutzt für Scopes und Attribute
- Kann über Schlüssel, Werte und Paare iterieren
- Beispiel:

```
>>> hash  = {1:2, "hallo":7j, ((1,2),3):[7,8,9]}
>>> hash[1]
2
>>> hash["hallo"]
7j
>>> hash[1]=7
>>> hash[1]
7
```

Exceptions

- Ähnlich Java / C++
- Beispiel:

```
try:
  try:
    print "alles in Ordnung"
    raise "Ausnahme"
    print "nie erreicht"
  except "Ausnahme":
    print "Exception bekommen
finally:
  print "fertig"
```

Sequenzen und Iteration

- For-Schleife iteriert über Sequenzen
- Beispiele:

```
for i in range(23,42):
  print i
for line in iter(open("dateiname")):
  print line
for anything in meindictionary.items:
  print anything
```

Generatoren

- Einfache Möglichkeit, einen Iterator zu erzeugen
- Quasi-Nebenläufige Programmierung
- derzeit noch als Syntax-Erweiterung
- Beispiel:

```
>>> def otto(start):
...     wert = start
...     while wert>0:
```

```
...             yield start
...             yield wert
...             wert -= 1
...             yield wert
...             yield "hallo"
...
>>> for i in otto(3): print i,
...
3 3 2 hallo 3 2 1 hallo 3 1 0 hallo
```

List Comprehension

- Angelehnt an mathematische Mengenschreibweise
- Beispiel:

```
>>> vec = [2, 4, 6]
>>> [3*x for x in vec]
[6, 12, 18]
>>> [3*x for x in vec if x > 3]
[12, 18]
>>> [3*x for x in vec if x < 2]
[]
>>> [{x: x**2} for x in vec]
[{2: 4}, {4: 16}, {6: 36}]
>>> [[x,x**2] for x in vec]
[[2, 4], [4, 16], [6, 36]]
>>> vec1 = [2, 4, 6]
>>> vec2 = [4, 3, -9]
>>> [x+y for x in vec1 for y in vec2]
[6, 5, -7, 8, 7, -5, 10, 9, -3]
>>> [vec1[i]*vec2[i] for i in range(len(vec1))]
[8, 12, -54]
```

Variable Parameterlisten

- Bekannt aus C, nicht in Java
- In Python "sicher" möglich
- Beispiel:

```
>>> def funktion(a,b=1,c=2,*l,**d):
...     print a,b,c,l,d
...
>>> funktion(1)
1 1 2 () {}
>>> funktion(1,2,3,4,5,6,7)
1 2 3 (4, 5, 6, 7) {}
>>> funktion(1,e=2,c=3,d=7)
1 1 3 () {'e': 2, 'd': 7}
```

Erweiterungen

- Problem: Weicher Übergang für inkompatible Änderungen
- Lösung: virtuelles Modul `__future__`
- Muß am Anfang der Datei stehen, verändert Compiler
- Beispiele:

```
from __future__ import generators
```

```
from __future__ import division
from __future__ import nested_scopes
```

Batteries Included

- Python-Philosophie: Alles mitliefern
- Über 200 Module mitgeliefert
- Eigentlich alles Dabei:
  - Systemzugriff
  - Stringverarbeitung inkl. Regular Expressions
  - Mathematische Funktionen
  - Grafische Oberfläche (TK)
  - Netzwerk inkl. vieler Protokolle
  - Curses-Bibliothek
  - Zugriff auf Bytecode und Compiler
  - xml-Verarbeitung
  - Kompression
  - ein- und auspacken von C-Strukturen
  - Plattformspezifisches
  - ...

GUI-Bindungen

- TK (mitgeliefert)
- curses (mitgeliefert)
- Gtk+/Gnome
- QT
- wxWindows
- Carbon
- MFC
- SDL
- Swing/AWT
- Python AnyGui
- ...

Sonstiges...

- Datenbankadapter (postgres, mysql, odbc...)
- Wissenschaftliches (Matritzen, Fehlerabschätzung, ...)
- Kryptographie
- CORBA, RPC, XML-RPC...
- Stack, Queqe, B-Bäume etc.
- Zope Web Application Server (inkl. ZODB, ZServer etc.)
- Gettext / Internationalisierung
- Imaging (inklusive Scanner Access)
- Unit-Test Framework
- Debian Paket manager
- ...

cPython

- das "Original"
- Von Guido van Rossum et Al.
- Aktuelle Version: 2.2.1
- Lizenz: GPL - Kompatibel
- Jit in Entwicklung: PsyCo
- Referenzimplementierung
- Download: 1885246 Bytes, Doku 1313384 Bytes (Debian Paket)

Jython

- Komplett unabhängige Implementierung
- Java Konform, enthält nur Java und Python-Code
- Interpretiert Python Quelltext
- Compiliert Python Quelltext nach Java Source
- Compiliert Python Quelltext nach Java Bytecode
- Enthält Python-Commandozeile
- aktueller Stand: Jython 2.1
- Nahtlos integriert mit Java
- Kleinere Inkompatibilitäten dokumentiert

Embedding

- In C sehr gut einzubetten
- Eigene Dokumentation existiert
- Als shared Library verfügbar
- "Sandbox" existiert
- Eingebettet z. B. in XChat oder als Apache-Modul

Erweiterungen mit C

- Interpreter in C implementiert
- Module transparent durch shared libraries ersetzbar
- Interne Interpreter-Schnittstellen für Datenstrukturen
- Auch großer Teil der Standardbibliothek in C

Schnittstellen, Wrapper, Wandler

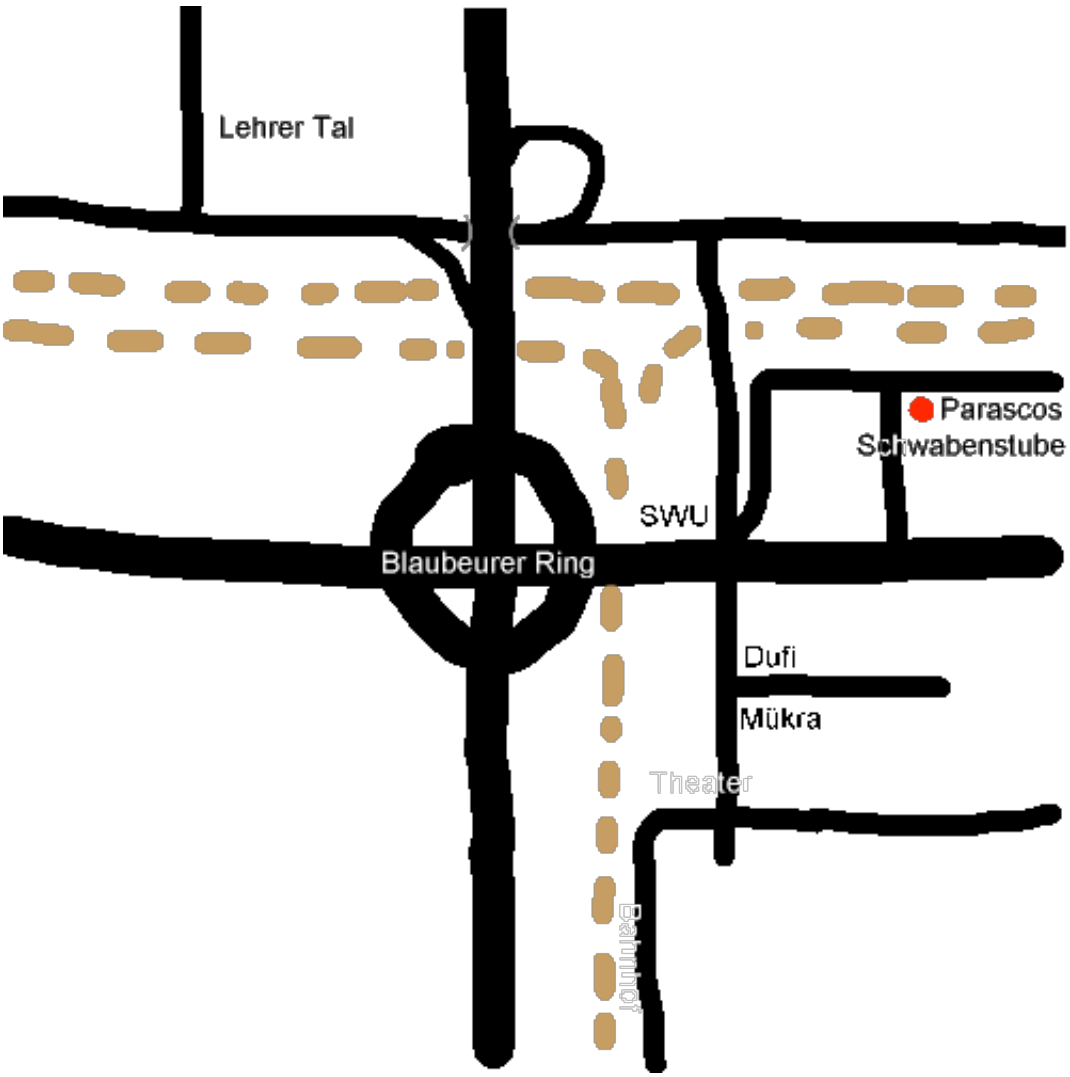
- PyJava: JNI-Schnittstelle für cPython
- SWIG: Generiert nahezu automatisch Wrapper für C Libraries
- Minotaur: Schnittstelle zu Perl und TCL
- Bridgekeeper: Perl -> Python-Konverter
- Visual Python: Integriert in MS Visual Studio .NET
- CXX Objects: C++-Objekte in Python benutzen
- SCXX: Python-Objekte in C++ benutzen
- PyFort: Fortran Schnittstelle

- Python / Erlang: Python Erlang Schnittstelle
- Python for Delphi: Python-DLL in Delphi integrieren

Literatur (das wichtigste)

- http://www.python.org/ - die Python Homepage
  Enthält auch die offizielle Python-Dokumentation zum Download und Linklisten.
- Python 2 - Einführung und Referenz der objektorientierten Skriptsprache
  Martin von Löwis und Nils Fischbeck,
  2. Auflage, Addison-Wesley Verlag,
  ISBN3-8273-1691-X
- http://py.vaults.ca/parnassus/ - Python Ressources
- http://www.google.com/ - die Suchmaschine meiner Wahl

Ende



- http://www.ulm.ccc.de/
- mail@ulm.ccc.de
- zweiter Montag: 20:00 Uhr Vortrag hier
- sonst: ab ca 19:30, Cafe Einstein (Uni)
- Verwendete Software:
  - Opera - Amaya - Mozilla
  - Debian Woody Linux

    - nedit - Gimp
    - Python
- Ab zu Parasco?