Anwendungen der Künstlichen Intelligenz

Python-Crashkurs

Prof. Dr. Peter Barth (Adrian Ulges)

Hochschule RheinMain

Fachbereich Design Informatik Medien

Medieninformatik / Angewandte Informatik / Wirtschaftsinformatik / ITS

14. April 2018

Python

Pseudocode that runs

Was ist Python?

- Objektorientierte Skriptsprache (“Perl by a sane person”)
- Einfach erlernbar (Einsteigersprache), interaktiv, Open Source
- Um Größenordnung schnellere Entwicklung, auch für Prototypen geeignet
- Plattformunabhängig, interpretiert, Bytecode (JIT mit pypy)

Features

- Dynamische Typisierung, Garbage Collection
- Eingebaute Datenstrukturen (Listen, Tupel, Dictionaries, String, . . .)
- Mächtige Ausdrucksweise und Werkzeuge (Verketten, abbilden, slicen)
- Klassische Kontrollstrukturen (imperativ), Objektorientierung und funktionale Primitive
- Unterstützung für große Projekte (Module, Ausnahmen, . . .)
- Integration (C, C++, Java, XMLRPC, REST, SOAP, . . .)
- Bibliotheken (Web, Email, reguläre Ausdrücke, XML, GUI, Threading, Unittesting, Bildbearbeitung, . . .)

Verbreitung

Plattform für Anwendungen

- Hunderte Skripte in `/usr/bin`
- Blender, Maya: 3D-Graphik
- Mercurial: Versionsmanagement
- Trac: Projekt/Software Management
- Google App Engine, Django, Raspberry Pi

Interner Einsatz

- Google, NASA, Yahoo, ILM

Entwickler

- Sehr weit verbreitet
- Die meist genutzte “general purpose” Skriptsprache
- Top Ten in TIOBE
  `http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html`

Apr 2017	Apr 2016	Change	Programming Language	Ratings	Change
1	1		Java	15.568%	-5.28%
2	2		C	6.966%	-6.94%
3	3		C++	4.554%	-1.36%
4	4		C#	3.579%	-0.22%
5	5		Python	3.457%	+0.13%
6	6		PHP	3.376%	+0.38%
7	10	^	Visual Basic .NET	3.251%	+0.98%
8	7	v	JavaScript	2.851%	+0.28%
9	11	^	Delphi/Object Pascal	2.816%	+0.60%
10	8	v	Perl	2.413%	-0.11%

Umgebung

Interpreter und Tools

- `www.python.org/2.7.3`
- Integrierte Entwicklungsumgebung (IDLE), und interaktiver Interpreter

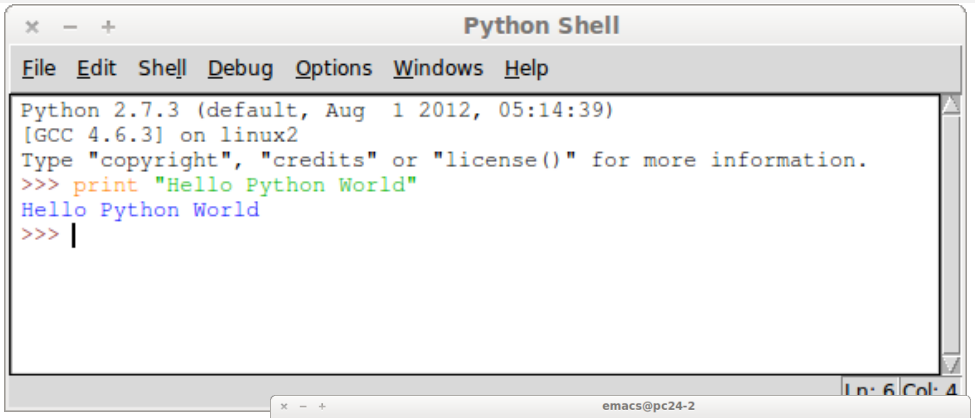
Interaktive “Shell” in IDLE

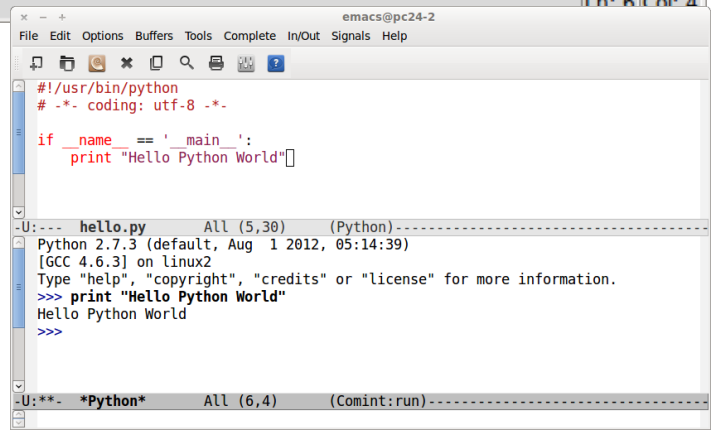
- Lernen der Sprache
- Experimentieren mit der Bibliothek
- Testen der eigenen Programme
- Auswerten von allen Ausdrücken

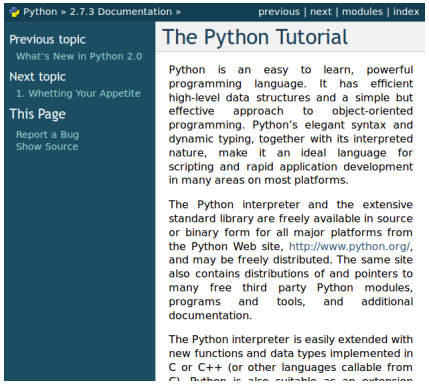
Alternativen: Emacs, eclipse pydev

Online-Lektüren (neben Buch)

- `docs.python.org/2/tutorial/`
- `wiki.python.org/moin/BeginnersGuide`
- `pythonchallenge.com`, `checkio.org`







Encoding bei Python Skripten – Unicode einstellen

Angabe der Zeichenkodierung

- Erste Zeile im Skript
  `# -*- coding: <encoding name> -*-`
  oder zweite Zeile, falls erste
  `#!/usr/bin/python`
- Spezieller Kommentar
- Teilt python (und IDLE) das Encoding des Quelltexts mit

Verfügbare Zeichenkodierungen

- `utf-8`, empfohlen
  - Unicode: alle Sprachen
  - Ab Python 3.0 Standard
  - Sehr stark empfohlen
- `iso-8859-1`, vermeiden
  - Standard Westeuropa, `latin1`
  - Böse: `Mac-Roman`, `cp1285`

BÃ¶ses Encoding

```
1 #!/usr/bin/python
2 # -*- coding: utf-8 -*-
3
4 print "Hello Python World"
```

1. Shebang: Starte den Python-Interpreter mit dem folgenden Zeilen als interaktive Eingabe
2. Encoding: Der ganze Quelltext, inklusive Kommentare und den Texten in Strings, ist in dem angegebenen Encoding

Alle ausführbaren Programme mit Zeilen 1 und 2 beginnen

Erste Schritte – Zahlen und Ausdrücke

```
>>> print "Hallo Python Welt"
Hallo Python Welt

>>> # Kommentare


>>> 2 # Integer-Zahlen
2

>>> 3.14 # Fliesskommazahlen
3.14

>>> "String" # Zeichenketten
'String'

>>> 0xff # Hex
255

>>> 0377 # Oktal
255
```

```
>>> 6+2 # Ausdruecke
8

>>> 6*2
12

>>> 6/2
3

>>> 6/5 # Vorgabe ganzzahlig (2.x)
1

>>> 6//5 # Explizit ganzzahlig ([23].x)
1

>>> 3**4 # Exponenten
81

>>> 3+1j # Komplexe Zahlen
(3+1j)
```

Erste Schritte – Zuweisungen und Tests

```
>>> x = 1 # Variablenzuweisung
>>> x # hat den zugewiesenen Wert
1

>>> x = y = 2 # Mehrfachzuweisung
>>> x
2
>>> y
2

>>> abs(-3) # eingebaute Funktionen
3

>>> 0 < 1 # Tests
True
>>> 1 < 0
False
```

```
>>> True == 1 # Bool neuer Typ
True
>>> False == 0
True

>>> 0 < 5 < 10 # Mehrfachtest
True

>>> True and False
False
>>> True or False
True

>>> not "eins"
False
>>> not False
True

>>> "eins" and "zwei" # letzter Wert
'zwei'
```

Große Zahlen und Konvertierung

```
>>> 3e10 # Exponentenschreibweise
30000000000.0

>>> 2**10 # Potenzieren
1024
>>> 2**20 # grosse ganze Zahl
1048576
>>> 2**1000 # sehr grosse ganze Zahl
10715086071862673209484250490600018105614048117055336074437503883703510511249361224931983788156958581275946729175531468251871452856923140435984577574698574803934567774824230985421074605062371141877954182153046474983581941267398767559165543946077062914571196477686542167660429831652624386837205668069376L
```

```
>>> 3L # explizit grosse Zahlen
3L
>>> float(3) # explizite Konvertierung
3.0 # in double
>>> int(3.0) # und zurueck
3
>>> long(3.0) # explizite Konvertierung
3L
>>> int("12") # auch fuer Strings
12
>>> int("11", 16) # explizite Basis
17

>>> int("zwoelf") # nur wenn moeglich
# ansonsten Ausnahme
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int()
          with base 10: 'zwoelf'
```

Zahlen und Operationen

Zahlentypen

- ganze Zahlen, **`int`** und **`long`** (beliebig lang)
- Gleitkommazahlen, **`float`** mit `double` Genauigkeit, komplexe Zahlen
- Intuitive Darstellung und Eingabe

Alle üblichen arithmetischen Operationen

- Standardarithmetik: `+`, `-`, `*`, `/`, `**`, `%`, `//`
- Logische Arithmetik: `~`, `^`, `|`, `&`, `<<`, `>>`
- Vorrangregeln und Klammern wie gewohnt

Boolesche Werte: `False`, `True`

- Entspricht 0 und 1 (Historie)
- Oder leerer String, nichtleerer String, ...

Logische Ausdrücke: **`and`**, **`or`**, **`not`**

- Erstes bestimmendes Element wird zurück gegeben
- Die andere Argumente werden nicht ausgewertet

```
1 Python 2.7.3
2 >>> 3/2
3 1
4 >>> 3//2
5 1
```

```
1 Python 3.2.3 ...
2 >>> 3/2
3 1.5
4 >>> 3//2
5 1
```

```
1 >>> 0 or ""
2     or "Hallo"
3     or "Welt"
4 'Hallo'
```

Ausdrücke

Aufrufe von Funktionen und Methoden

- Funktionsaufrufe
- Methodenaufrufe
- Interaktive Ausgabe des Wertes
- Zusammengesetzte Ausdrücke
- Bereichstests

Werte ausgeben in der interaktiven Kommandozeile

- Zuweisungen haben *keinen* Wert
- Wert von Ausdrücken wird ausgegeben
- **`print`** für explizite Ausgabe

```
>>> lis = [3,1,2]
>>> len(lis) # Funktionsaufruf
3
>>> lis.append(8) # Methode
>>> lis # Ausgabe
[3, 1, 2, 8]
>>> len(lis) < 6 and len(lis)
# Zusammengesetzt
4
>>> 1 < len(lis) < 8 # Bereich
True
```

```
>> lis
[3, 1, 2]
>>> print lis # 2.x
[3, 1, 2]
>>> print(lis) # 2.x, 3.x
[3, 1, 2]
```

Ausgabe mit `print`

Ausgabe von Objekten mit **`print`**

- Lesbare Ausgabe von Objekten
- Ausgabe aller Objekte durch Leerzeichen getrennt
- Mit Komma am Ende wird Zeilenumbruch unterdrückt
- Ausgabe mit Formatstring, wie in C, %-Operator
- Python 3.x, nur noch in Klammern (jetzt schon möglich)

Alternative

- Ausgabe wie in Java

```
>>> lis = [1,2,3]
>>> print lis, "zwei", 4
[1, 2, 3] zwei 4
>>> print lis,"zwei",4; print "weiter"
[1, 2, 3] zwei 4
weiter
>>> print lis,"zwei",4,; print "weiter"
[1, 2, 3] zwei 4 weiter
>>> print "%s %s" % ("Hallo", "Welt")
Hallo Welt
>>> print(lis)
[1, 2, 3]
```

```
>>> import sys
>>> sys.stdout.write("Hallo Welt\n")
Hallo Welt
```

Zeichenkette, Strings – Darstellung

Syntax

- Einfache oder doppelte Anführungszeichen
- Nicht limitierendes Anführungszeichen nutzbar
- Alternativ Escape-Zeichen verwenden Backslash \

```
>>> "hallo"
'hallo'
>>> 'hallo'
'hallo'
```

```
>>> "spam's"
"spam's"
>>> 'spam"s'
'spam"s'
>>> 'spam\'s'
"spam's"
```

Mehrzeilige Strings

- Drei Anführungszeichen am Anfang und am Ende
- Zeilenendezeichen als \n verfügbar
- Alle Anführungszeichen verwendbar

```
>>> """hallo
... du "tolle"
...    'perfekte' Welt """
'hallo\n du "tolle"\n
\'perfekte\' Welt '
```

Strings – Operationen

Alle üblichen Operationen vorhanden

- `+` Konkatenation
- `*` Wiederholung

`[+i]` Indizierung
positive Zahl, 0 ist erstes Zeichen

`[-i]` Indizierung von hinten
negative Zahl, -1 ist letztes Zeichen

- Slicing

`[von:bis]` Ausschnitt von inklusive, bis exklusive
`[von:]` Ausschnitt bis Ende
`[:bis]` Ausschnitt ab Anfang
`[:]` Gesamt (Kopie)

```
>>> "Hallo" + " Welt"
'Hallo Welt'
>>> "Hallo"*3
'HalloHalloHallo'
>>> "Hallo"[0]
'H'
>>> "Hallo"[-1]
'o'
```

```
>>> "Hallo"[1:3]
'al'
>>> "Hallo"[1:]
'allo'
>>> "Hallo"[:-1]
'Hall'
```

Slicing

`"SLICEOFSPAM"[von:bis]`

- von, einschließend
- bis, ausschließend

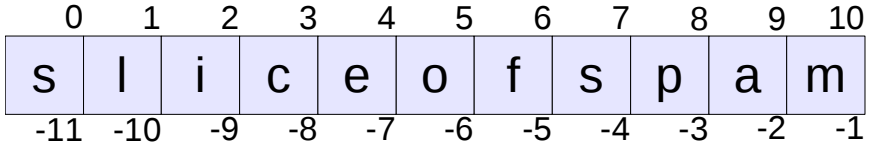
Vorzeichen

- Positive Zahlen, von links, ab 0
- Negative Zahlen, von rechts, ab -1

Dritter Parameter Schrittweite

- `"SLICEOFSPAM"[von:bis:step]`
- Schrittweite, normalerweise 1

Geht auch mit Listen (später)




```
>>> "sliceofspam"[2:-1]
'iceofspa'
>>> "sliceofspam"[5:]
'ofspam'
>>> "sliceofspam"[:5]
'slice'
>>> "sliceofspam"[2:5]
'ice'
```

```
>>> "sliceofspam"[2:-1:3]
'iop'
>>> "sliceofspam"[::4]
'sep'
>>> lis = [0,1,2,3,4,5,6,7]
>>> lis[::2], lis[1::2]
([0, 2, 4, 6], [1, 3, 5, 7])
```

Strings – Formatierung und Builtins

Formatierung

- Formatstring und Argumente
- Ähnlich zu C, `printf`
- `%s` String
- `%d` Zahl
- `%x` Hexadezimal
- `%e, %f, %g` Gleitkommaformate
- `%%` Prozentzeichen

```
>>> "ein %s Papagei" % "toter"
'ein toter Papagei'
>>> "%d %s Papageie" % (2, "tote")
'2 tote Papageie'
>>> "%x" % 42
'2a'
>>> "%X" % 42
'2A'
>>> "%e %f %g" % (1.1, 1.2, 1.3)
'1.100000e+00 1.200000 1.3'
>>> "%%%s%%" % "hallo"
'%hallo%'
```

Strings – Builtins

Eingebaute Funktionen, Builtins

- Länge mit **`len`**
- Standardoperationen für Vergleich
- Test auf Enthaltensein mit **`in`**

```
>>> len("Hallo")
5
>>> "hallo" < "wallo"
True
>>> "al" in "Hallo"
True
```

Strings – weitere Features

`string`-Modul

- als Modul “importieren”
- Funktionen oder String-Methoden
- Suchen, finden, ersetzen
- Konvertieren, splitten, kleben
- Leerzeichen etc. entfernen

Kommandozeilenargumente

- Modul `sys`
- Liste von Argumente `sys.argv`

```
$ python argv.py a b c d
['argv.py', 'a', 'b', 'c', 'd']
```

```
>>> import string
>>> "ab".upper(), string.upper("ab")
('AB', 'AB')
>>> string.find("spammify", "mm")
3
>>> string.split("spammify"), "mm")
['spa', 'ify']
>>> 'spam no\nspam'.split()
['spam', 'no', 'spam']
>>> "XX".join(["spa", "ify"])
'spaXXify'
>>> string.strip(" Meaning of Life \n")
'Meaning of Life'
```

```
#!/usr/bin/python
import sys
print sys.argv
```

Python – Do It Yourself!

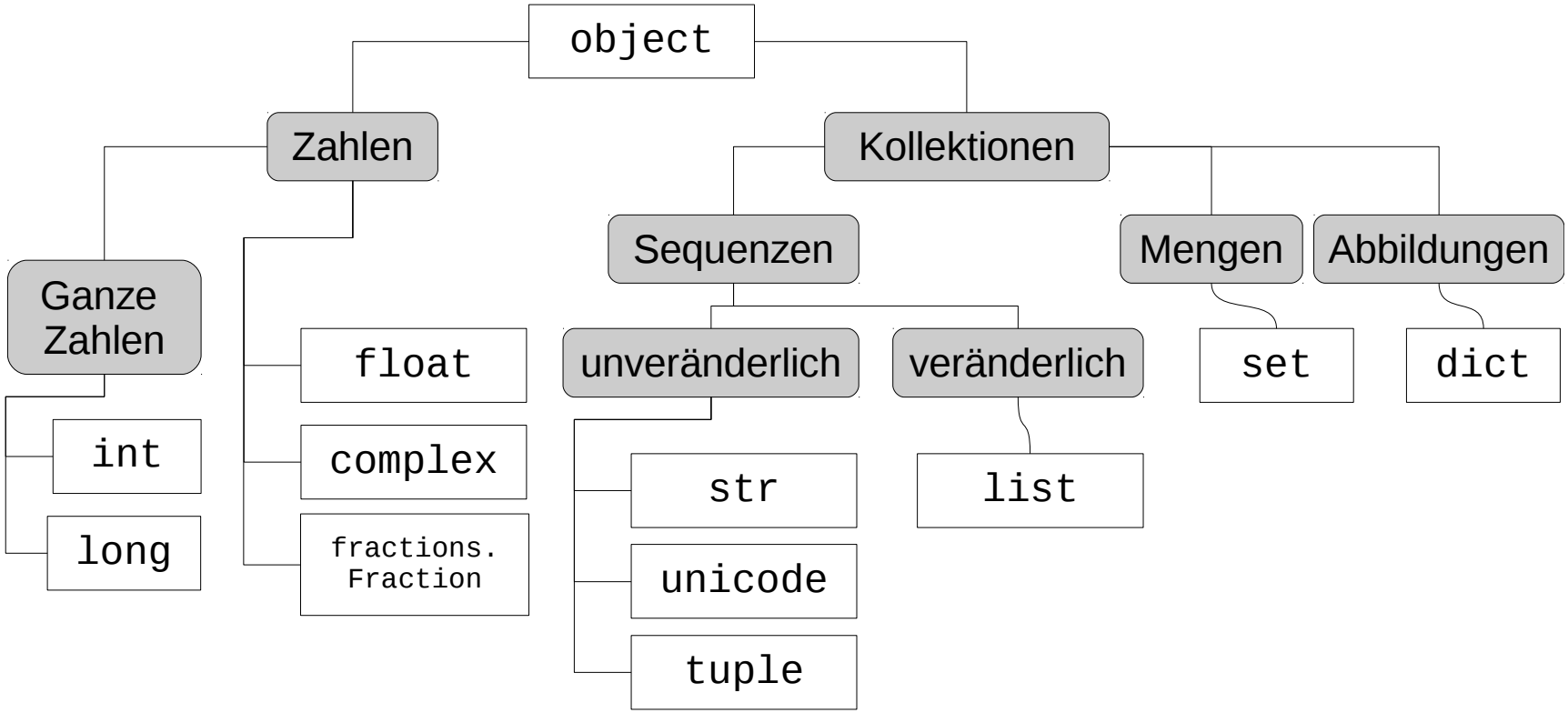
- Öffnen Sie den Python-Interpreter (“`python`” in der Shell)
- Berechnen Sie die Zahl $x = 2^{100} - 1$.
- Konvertieren Sie x in einen String (als Dezimalzahl ohne Nachkommastellen).
- Wieviele Stellen hat x in Dezimaldarstellung?
- Geben Sie den formatierten String aus: “Last four digits: XXXX.” (wobei “XXXX” die letzten vier Dezimalstellen von x sind).
- Wieviele Nullen kommen in x vor?

Typhierarchie eingebauter Typen

`object` als Wurzel für alle Typen

`isinstance` zum Testen

```
>>> isinstance(3, int)
True
>>> isinstance(3, object)
True
```



Kollektionen

Eingebaute Kollektionen

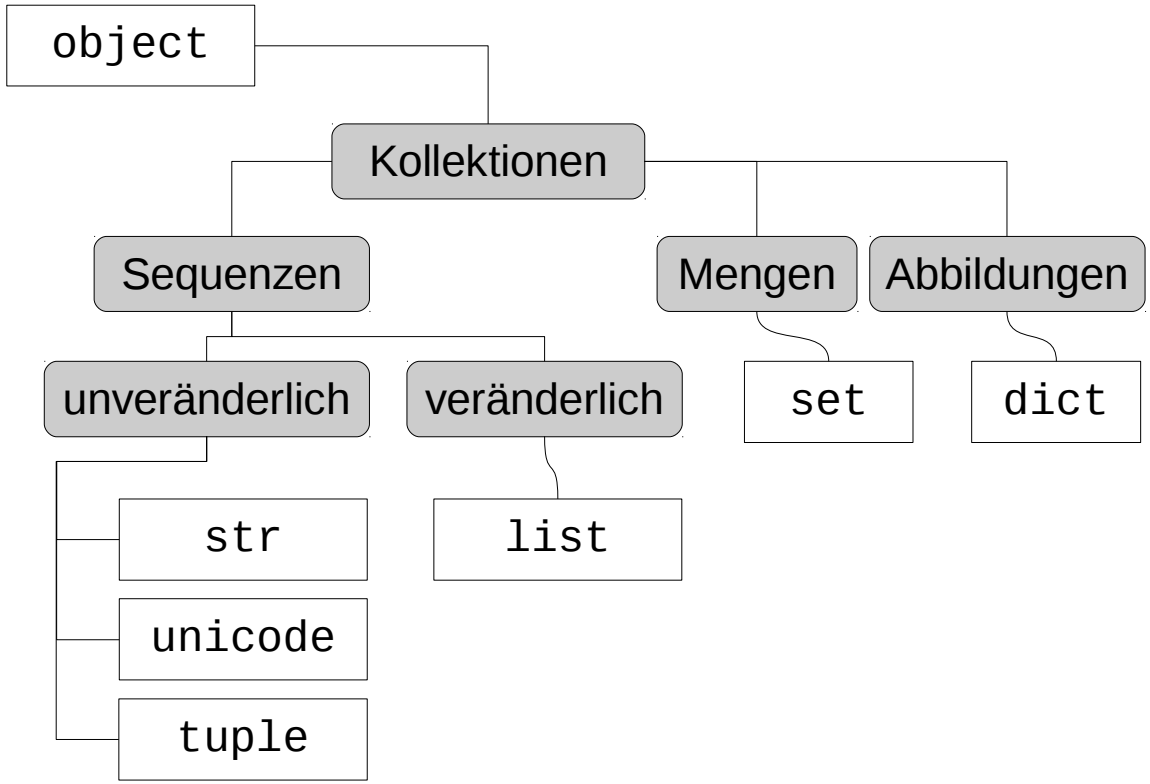
- Eingebaute Typen zum Verwalten von mehreren Objekten
- Teil der Sprache, nicht nur der Bibliothek

Vorteile

- Effiziente Implementierung
- Einfache Formulierung
- Konzentration aufs Problem nicht auf technische Implementierungsdetails

Umfangreich

- Alles was man braucht
- Ungefähr `java.util.*`
- Strings, Listen, Tupel, Mengen, Wörterbücher



Tupel – Sequenz

Tupel sind eine Sequenz

- Syntax n-Tupel, n$\geq$ 2 wie erwartet
- 1-Tupel hat spezielle Syntax
  `(1)` wäre nur ein arithmetischer Ausdruck mit Klammerung
- Leeres Tupel wie erwartet

Beliebig kombinierbar

- Beliebige Typen, schachtelbar

Indizierung

- mit `[]`
- Slicing (wie mit allen Sequenzen)
  Ergebnis ist wieder Tupel

```
1  >>> (1, 2)
2  (1, 2)
3  >>> (1, 1, 1)
4  (1, 1, 1)
5  >>> (1, )
6  (1,)
7  >>> (1)
8  1
9  >>> ()
10 ()
11 >>> (1, "eins", 2.0, (34, 4))
12 (1, 'eins', 2.0, (34, 4))
13 >>> tup = (1, "zwei", (3, 4))
14 >>> tup[0]
15 1
16 >>> tup[2][1]
17 4
18 >>> tup[1:]
19 ('zwei', (3, 4))
```

Tupel – Builtins

Builtins von Sequenzen

- Länge mit **len**
- Konkatenation mit +
- Wiederholung mit ∗
- Test auf Enthaltensein mit **in**

Automatisch Tupel

- Auch ohne Klammern

Tupelzuweisung

- Nette Syntax
- Geht auch ohne Klammern
- **Gut bei Rückgabewerten**

```
>>> tup = (1, "zwei", (3, 4))
>>> len(tup)
3
>>> tup + tup
(1, 'zwei', (3, 4), 1, 'zwei', (3, 4))
>>> tup*2
(1, 'zwei', (3, 4), 1, 'zwei', (3, 4))
>>> (3, 4) in tup
True
```

```
>>> 1, 2
(1, 2)
>>> (1, 2)
(1, 2)
>>> (a, b) = (1, 2)
>>> print a, b
1 2
>>> b, a = a, b
>>> a, b
(2, 1)
```

Listen – Sequenz

Syntax wie Tupel

- Umschlossen mit `[` und `]`
- Keine spezielle Syntax mehr für einelementige Listen
- Leere Liste mit `[]`

Beliebig kombinierbar und schachtelbar

- Auch mit Tupel

Sonst wie Tupel

```
>>> [1, 2, 3]
[1, 2, 3]
>>> [1], []
[1], []
>>> [1, [2, "zwei"], (3, "drei")]
[1, [2, 'zwei'], (3, 'drei')]
>>> lis = [1, [2, "zwei"], (3, "drei")]
>>> lis[0]
1
>>> lis[1:]
[[2, 'zwei'], (3, 'drei')]
>>> lis[2][0], len(lis)
(3, 3)
>>> lis + [4,5]
[1, [2, 'zwei'], (3, 'drei'), 4, 5]
>>> 1 in lis
True
>>> (lis*2)[1:-2]
[[2, 'zwei'], (3, 'drei'), 1]
```

Listen – Veränderliche Sequenz

Ändern

- Zuweisung von Teilen
- Löschen

Kopien und Zuweisung

- Operationen erzeugen Kopien
- Nachträgliches Verändern der Parameter ohne Einfluss

Flache Kopien

- Kopien nur auf oberster Ebene
- Erzeugen neuer Listenstruktur
- Die Elemente werden nicht kopiert, sondern nur Referenzen
- Ändern der Elemente ist sichtbar

```
>>> lis = [1, 2, 3]
>>> lis[0] = "eins"
>>> lis
['eins', 2, 3]
>>> del lis[0]
>>> lis
[2, 3]
```

```
>>> lis, kat = ["a", "b"], ["c", "d"]
>>> lk = lis + kat
>>> lk
['a', 'b', 'c', 'd']
>>> lis[0] = "X"
>>> lis, lk
(['X', 'b'], ['a', 'b', 'c', 'd'])
>>> tup = (lis, kat)
>>> tup
(['X', 'b'], ['c', 'd'])
>>> lis[1] = "Y"
>>> tup
(['X', 'Y'], ['c', 'd'])
```

Listen – Zuweisungen

Zuweisungskompatibilität

- Sequenzen sind untereinander zuweisungskompatibel

Slice-Zuweisung möglich

- Der Bereich wird durch Sequenz als Liste ersetzt

Destruktive Methoden

- `append` und `extend` ändern gegebene Liste, kein Rückgabewert
- **`del`** ändert Liste
- Zuweisung an Index ändert Liste

```
1  >>> lis = [1, 2, 3]
2  >>> lis[1:2] = ("zwei", "two")
3  >>> lis
4  [1, 'zwei', 'two', 3]
5  >>> lis[1:2] = ["zwei", "two"]
6  >>> lis
7  [1, 'zwei', 'two', 'two', 3]
8  >>> lis.append(4)
9  >>> lis
10 [1, 'zwei', 'two', 'two', 3, 4]
11 >>> lis.extend([5, 6])
12 >>> lis
13 [1, 'zwei', 'two', 'two', 3, 4, 5, 6]
14 >>> del lis[1:3]
15 >>> lis
16 [1, 'two', 3, 4, 5, 6]
17 >>> lis[1] = 2
18 >>> lis
19 [1, 2, 3, 4, 5, 6]
```

Listen – Weitere Methoden

range generiert Zahlenliste

- Anfang, erstes Argument, optional (default 0)
- Ende, zweites Argument
- Schrittweite, drittes Argument, optional (default 1)

Weitere destruktive Methoden

- `pop`, `insert`
  Kellerverhalten und mehr
- `reverse`, `sort`, **`sorted`**
  klassische Listenoperationen

```
>>> lis = [4, 3, 2.5, 2, 1, 0, -1]
>>> sorted(lis)
[-1, 0, 1, 2, 2.5, 3, 4]
>>> lis.sort(); lis
[-1, 0, 1, 2, 2.5, 3, 4]
```

```
>>> range(5)
[0, 1, 2, 3, 4]
>>> range(1, 5)
[1, 2, 3, 4]
>>> range(0, 50, 10)
[0, 10, 20, 30, 40]
>>> lis = range(5)
>>> lis.append(5); lis
[0, 1, 2, 3, 4, 5]
>>> lis.pop()
5
>>> lis
[0, 1, 2, 3, 4]
>>> lis.insert(0, -1); lis
[-1, 0, 1, 2, 3, 4]
>>> lis.insert(4, 2.5); lis
[-1, 0, 1, 2, 2.5, 3, 4]
>>> lis.reverse(); lis
[4, 3, 2.5, 2, 1, 0, -1]
```

Dictionaries, Wörterbücher

Dictionary, Wörterbuch

- Abbildung unveränderlicher Werte (Schlüssel) auf beliebigen Werten
- Syntax für Zugriff und Zuweisung wie beim Feld
- Realisiert über Hashing

Vorteile

- Verwendung einfach und intuitiv
- Schlüssel nicht beschränkt auf Zahlen
- Keine Feldgröße notwendig

Dictionaries und Sequenzen

- Schlüsselmenge (Sequenz), Wertemenge und Tupelmenge Schlüssel/Wert erzeugbar
- Test auf Enthaltensein gegen Schlüsselmenge

```
1  >>> dic = {}
2  >>> dic = { 'spam' : 2 }
3  >>> dic['spam']
4  2
5  >>> dic['spam'] = 3
6  >>> dic
7  {'spam': 3}
8  >>> dic['eggs'] = 42
9  >>> dic
10 {'eggs': 42, 'spam': 3}
11 >>> dic.keys()
12 ['eggs', 'spam']
13 >>> dic.values()
14 [42, 3]
15 >>> dic.items()
16 [('eggs', 42), ('spam', 3)]
17 >>> 'eggs' in dic
18 True
19 >>> dic.has_key('spam')
20 True
```

Mengen – Veränderlich, keine Reihenfolge

Mengen

- Keine doppelten Elemente
- Keine Reihenfolge, keine Sequenz
- Realisiert wie Dictionaries, nur ist der Wert egal

Eingebaute Funktionen und Operationen

- **`len`** für Anzahl Elemente
- **`in`** für Test auf Enthaltensein
- Mengenoperationen mit `|` und `&`

```
>>> s = set()
>>> s.add(1)
>>> s
set([1])
>>> s.add(2); s.add(3)
>>> s
set([1, 2, 3])
>>> s.add(2)
>>> s
set([1, 2, 3])
>>> 2 in s
True
>>> 4 in s
False
>>> len(s)
3
>>> s | set([2,3,4])
set([1, 2, 3, 4])
>>> s & set([2,3,4])
set([2, 3])
```

Variablen

Verwendung

- Müssen nicht deklariert werden
- Müssen vor erster Verwendung initialisiert werden

Typ, dynamisch und streng

- Dynamische Typisierung, Variable an alles bindbar
- Strenge Typisierung, Gebundenes Objekt hat Typ

Gültige Variablenbezeichner wie in Java, keine Schlüsselwörter

```
>>> x = 3
>>> x
3
>>> y
NameError: name 'y' is not defined
>>> x = 3
>>> x = "Hallo"
>>> x = 3; x + 3
6
>>> x = "3"; x + 3
TypeError: cannot concatenate
    'str' and 'int' objects
```

Kontrollstrukturen – Verzweigung

Verzweigung

- Schlüsselwörter: **`if`**, **`elif`**, **`else`**
- `<bedingung>`: Ausdruck, der als Wahrheitswert interpretiert wird
- `<anweisungen>`: Beliebige Folge von Anweisungen

Syntax

- Bedingung endet mit Doppelpunkt, Keine Klammern notwendig
- **ACHTUNG! Einrückung ist signifikant und Teil der Syntax!**
  - Block durch Einrücktiefe bestimmt
  - Einrückung Tabulator oder gleiche Anzahl an Leerzeichen
  - Keine (geschweiften) Klammern!

Einzeiler möglich, vermeiden

```
if <bedingung>:
   <anweisungen>
[elif <bedigung>:
   <anweisungen>]*
[else:
   <anweisungen>]
```

```
1  if 0 == 0:
2     print "ja"
3  if "nichtleerer String":
4     print "ja"
5  else:
6     print "nein"
7  if 0:
8     print "eins"
9     print "zwei"
10 print "drei"
11 if 1: print "eins"
```

`if/else` Ausdruck statt ?-Operator

Syntax ungewöhnlich

- Erst Ergebniswert im “normalen” Fall
- Dann Bedingung
- Dann “alternativer” Fall

Alternative mit booleschem Ausdruck

```
<wert> if <bedingung> else <alternative>
```

```
// in Java, C
<bedingung> ? <wert> : <alternative>
```

```
1 >>> x, y = 1, 2
2 >>> e = "kleiner" if x<y else "groessergl"
3 >>> e
4 'kleiner'
```

Kontrollstrukturen – `while`-Schleife

`while`-Schleife

- Schlüsselwörter: **`while`**, **`else`**, **`break`**, **`continue`**, **`pass`**
- **`else`**-Zweig nur, wenn Schleife ohne **`break`** verlassen wurde
- **`pass`**, leere Anweisung, macht nichts, Ersatz für `{}`

Syntax

- Wieder Einrücken als integraler Bestandteil der Schleifensyntax

```
while <bedingung>:
    <anweisungen>
[else:
    <anweisungen>]
```

```
1 lis = [1, 2, 3, 4]
2 while lis:
3     print lis.pop()
4 else:
5     print "Ende"
```

```
4
3
2
1
Ende
```

Kontrollstrukturen – `for`-Schleife

`for`-Schleife

- Schlüsselwörter: **`for`**, **`in`**, **`break`**, **`continue`**, **`pass`**
- Iterieren über Elemente in Sequenzen
- **`in`** wählt Elemente aus Sequenz sukzessive aus und weist sie je Durchlauf der Variable zu
- Klassische Schleife über Indexe nur nach als Nachbau mit **`range`**

`range`

- **`range`**`(von, bis, step)`
- Erzeugt eine Liste von Zahlen `von` inklusive, `bis` exklusive
- `von` optional, default 0
- `step` optional, default 1

```
for <variable> in <sequenz>:
   <anweisungen>
[else:
   <anweisungen>]
```

```
1 for ele in ["Hallo", "Welt"]:
2    print ele
3 for idx in range(1, 4):
4    print idx
5 for ch in "Hi":
6    print ch
```

```
Hallo
Welt
1
2
3
H
i
```

Kontrollstrukturen – `for`-Schleife

`for`-Schleife in Kombination mit Listenkonstrukten

- `enumerate`: gleichzeitiger Zugriff auf Element und Index
- `zip`: simultaner Durchlauf mehrerer Listen

```
>>> nachnamen = ['Shannon', 'Minsky',
                 'Rosenblatt']
>>> vornamen = ['Claude', 'Marvin',
                'Frank']
>>> for vn,nn in zip(vornamen,nachnamen):
        print vn,nn
Claude Shannon
Marvin Minsky
Frank Rosenblatt
>>> for i,name in enumerate(vornamen):
        print i,name
0 Claude
1 Marvin
2 Rosenblatt
```

Kontrollstrukturen – `for`-Schleife, Dictionaries

`for`-Schleife und Dictionaries

- Fragt Objekte, ob sie auch eine Sequenz sein können
- Das Dictionary sagt, ja, dann bin ich eine Sequenz der Schlüssel
- Dann wird über die Schlüssel iteriert

Auch nett mit `items` und Tupelzuweisung

```
1 dic = { 1: 'eins', 2: 'zwei' }
2 for key in dic: # wie dic.keys()
3    print key, dic[key]
```

```
1 eins
2 zwei
```

```
1 dic = { 1: 'eins', 2: 'zwei' }
2 print dic.items()
3 for key, value in dic.items():
4    print key, value
```

```
[(1, 'eins'), (2, 'zwei')]
1 eins
2 zwei
```

Python – Do It Yourself!

- Lesen Sie den Text von Bram Stoker's Roman "Dracula" aus der Datei "dracula.txt" ein (z.B. mit Hilfe der `readlines()`-Methode, siehe `hier`)
- Splitten Sie den Text in einzelne Wörter und packen Sie diese in eine Liste. (*Anmerkung: Wir vernachlässigen hier Satzzeichen wie Kommata)*
- Konvertieren Sie sämtliche Großbuchstaben in Kleinbuchstaben.
- Durchlaufen Sie die Tokens und zählen Sie in einem Python-Dictionary, wie oft jedes Token im Text vorkommt.
- Was ist das häufigste Wort in "Dracula"?

Funktionen

Definition mit **`def`**

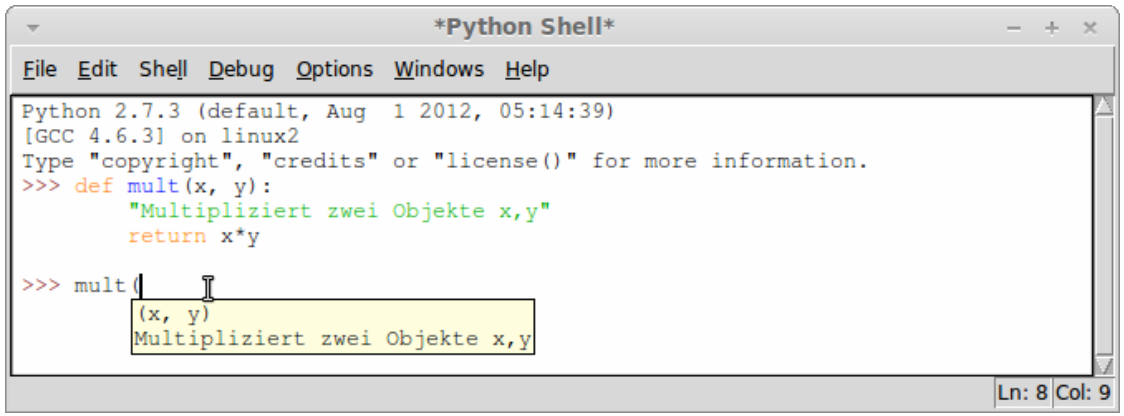
- Erstellt Funktionsobjekt und bindet Namen `<fname>` daran
- **`return`** für Rückgabe, ansonsten automatisch `None`
- Eins oder mehrere Argumente/Parameter `<arg>` werden per Namensbindung übergeben
- Dokumentationsstring `<docstring>` eingebaut, optionale erste Zeile

Auch interaktiv eingebbar

- Leerzeile zum Beenden

```
def <fname>([<arg> [, <arg> ...]]):
    [<docstring>]
    <anweisungen>
    [return <name>]
```

```
1 def mult(x, y):
2     "Multipliziere zwei Objekte x, y"
3     return x*y
```



Parameterübergabe – Vorgabewerte

Vorgabewerte (Default)

- Vorgabewert kann optional bei jedem Parameter angegeben werden
- Zusätzlich = und Ausdruck, der zu Vorgabewert evaluiert
- Parameter mit Vorgabewerten hinter die Parameter ohne Vorgabewerte
- Stellungsparameter, Positionsparameter

Aufruf

- Wert für Parameter mit Vorgabewert kann weggelassen werden
- Falls Wert weggelassen, wird Vorgabewert eingesetzt

```
1 def ink(x, a=1):
2   return x+a
```

```
1 >>> ink(3)
2 4
3 >>> ink(3, 1)
4 4
5 >>> ink(3, 17)
6 20
```

Funktionale Primitive – `map`

Funktion auf allen Elementen

- Für alle Objekte von Sequenzen eine Funktion anwenden
- Schlüsselwort **`map`**
- Erster Parameter die Funktion
- Ab zweiten Parameter die Sequenzen, so viele Sequenzen wie die Funktion Parameter hat
- Ergebnis ist Liste Funktionsergebnis je Element der Eingabesequenz
- Kürzeste Sequenz bestimmt Länge des Ergebnisses

Beispiele

- Erhöhe alle Werte um 1
- Addiere die Inhalte einer Liste paarweise

```
1 >>> def ink(x):
2       return x+1
3 >>> map(ink, [1, 2, 3])
4 [2, 3, 4]
5 >>> def add(x, y):
6      return x+y
7 >>> map(add, [1, 2, 3], [4, 5, 6])
8 [5, 7, 9]
```

Einsatzbeispiel für `map`

Aufgabe

- Berechne die Liste der Quadratzahlen von 1 bis 10

Lösung

- **`for`**-Schleife
- Liste zusammenbauen
- Zurückgeben

Funktionale Lösung

- Einzeiler mit **`map`**
- Liste der Zahlen mit **`range`**
- Keine Schleifen

```
1 def quadratzahlen(von=1, bis=10):
2   lis = []
3   for i in range(von, bis+1):
4     lis.append(i*i)
5   return lis
6
7 print quadratzahlen()
```

```
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

```
1 def quadratzahlen(von=1, bis=10):
2    def sqr(x):
3       return x*x
4    return map(sqr, range(von, bis+1))
5
6 print quadratzahlen()
```

List Comprehension

Listentransformation

- Anwendungsgebiet wie funktionales Programmieren
- Einfachere Verwendung der Möglichkeiten von **`map`** und **`filter`**

Syntax `[<ausdruck> for <var> in <seq>]`

- Durch **`for`** wird `<var>` sukkzesive ein Wert aus `<seq>` zugewiesen
- Zusätzlich **`if`** `<bedingung>`, als Filter
- Beliebig kombinierbar

Ausdrucksweisen

- Statt verschachtelte **`for`**-Schleifen
- Ähnelt math. Mengenschreibweise

```
>>> def ink(x): return x+1
>>> map(ink, [1,2,3,4])
[2, 3, 4, 5]
>>> [x+1 for x in [1,2,3,4]]
[2, 3, 4, 5]
>>> [x for x in [1, 5, 2, 6, 7] if x>3]
[5, 6, 7]
```

```
>>> [x**2 for x in range(1, 11)
        if x % 2 == 0]
[4, 16, 36, 64, 100]
```

$$\{x^2 | 1 \leq x \leq 10, x \mod 2 = 0\} = \{4, 16, 36, 64, 100\}$$

List Comprehension – Beispiele mit Zahlen

Quadratzahlen

- Zahlen von 0 bis 9; 1 bis 10
- Ausdruck statt Funktion

Komplexe Definition

- Mischen von Filtern und Funktionsapplikation
- Aufzählen durch **`for`**

Kartesisches Produkt

- Sequenzen kombinieren
- Kartesisches Produkt, Vollkombination

`zip`

- Builtin, generiert Tupel

```
>>> [x for x in range(10)]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> [x*x for x in range(1, 11)]
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

```
>>> [(x,y) for x in range(4)
        for y in range(0,5,2)
        if x < y]
[(0, 2), (0, 4), (1, 2), (1, 4),
 (2, 4), (3, 4)]
```

```
>>> [x+y for (x,y) in zip(range(5),range(5))]
[0, 2, 4, 6, 8]
>>> zip(range(5), range(5))
[(0, 0), (1, 1), (2, 2), (3, 3), (4, 4)]
```

List Comprehension – Beispiele mit Text

Wörter ersetzen

- Kleine Zahlen durch Wörter ersetzen
- Für alle Worte, falls das Wort 1,2 oder 3 ist,
- dann ersetze durch Zahlwort und setze neuen String zusammen

Datei von Zahlen zeilenweise lesen

- Zeilenweise einlesen und aufteilen (splitten)
- Alle Strings aus Ziffern je Zeile in Liste von Zahlen umwandeln
- Tupel aus Liste machen

```
>>> trans = {'2':'zwei', '3': 'drei',
          '4': 'vier'}
>>> s = """2 Sachen und 3 Sachen sind
nicht 4 Sachen"""
>>> print " ".join([trans[w]
                if w in trans else w
                for w in s.split()])
zwei Sachen und drei Sachen sind
nicht vier Sachen
```

```
>>> d = "zahlen.dat"
>>> file(d).readlines()
['1 2 3\n', '42 343 12\n',
 '8  77 223\n', '23 43\n',
 '45 3 3 66 7\n']
>>> [tuple(map(int, line.split()))
            for line in file(d)]
[(1, 2, 3), (42, 343, 12), (8, 77, 223),
 (23, 43), (45, 3, 3, 66, 7)]
```

Module

Warum Module?

- Wiederverwendung von Code
- Bereitstellung von allgmein nutzbaren Diensten und Bibliotheken
- Unterteilung des Namensraums bei großen Projekten

Python Module

- Jede Python-Datei (Endung mit `.py`) ist ein Modul
- C-Erweiterungen sind Modul
- Nutzen mit **`import`**, **`from`**

Suche nach Dateien/Module

- Umgebungsvariable `PYTHONPATH`
- Abhängig von Wert in `sys.path`

```
>>> import sys
>>> sys.path
['', # sucht auch im akt. Verzeichnis
 '/usr/lib/python2.7',
 '/usr/lib/python2.7/plat-linux2',
 '/usr/lib/python2.7/lib-tk',
 '/usr/lib/python2.7/lib-old',
 '/usr/lib/python2.7/lib-dynload',
 '/usr/local/lib/python2.7/dist-packages',
 '/usr/lib/python2.7/dist-packages',
 '/usr/lib/python2.7/dist-packages/PIL',
 '/usr/lib/python2.7/dist-packages/gst-0.10',
 '/usr/lib/python2.7/dist-packages/gtk-2.0',
 '/usr/lib/pymodules/python2.7']
```

Definieren von Modulen

Ein Modul ist eine Datei

- Mit der Endung `.py`, oder vorkompiliert `.pyc`
- In einem Verzeichnis
- Beliebiger Python-Code
- Keine Python-Schlüsselwörter als Dateiname

Im Suchpfad erreichbar

- Vorgabe ist aktuelles Verzeichnis und Standard-Bibliothek von Python
- `sys.path`

meinmodul.py

```
1  # Modul meinmodul, meinmodul.py
2  # Variablen und Funktionen
3
4  deutsch = "deutsch"
5  englisch = "englisch"
6
7  sprache = deutsch
8
9  def drucke(x):
10     print x
11
12 def gruss():
13     if sprache == deutsch:
14         drucke("Hallo Modul Welt")
15     else:
16         drucke("Hello module world")
17
18 print "meinmodul geladen"
```

Importieren von Modulen

Modul importieren

- Im aktuellen Verzeichnis
  - Wechseln mit `os.chdir()`
- Importiert Modul mit dem Namen `<modul>`
- Objekte nur über den Modul-Präfix erreichbar

Namen von Modulen importieren

- Modul selbst nicht sichtbar
- Alle Namen `<name>` im aktuellen Namensraum verfügbar
  - Aber an Objekt in Modul gebunden
- $*$ für alle Namen außer Namen, die mit einem Unterstrich _ beginnen

import <modul> [, <modul>]*

```
1  $ ls meinmodul.py
2  meinmodul.py
3  $ python
4  >>> import meinmodul
5  meinmodul geladen
6  >>> meinmodul.gruss()
7  Hallo Modul Welt
8  >>> meinmodul.sprache=meinmodul.englisch
9  >>> meinmodul.gruss()
10 Hello module world
11 >>> meinmodul.sprache = englisch
12 NameError: name 'englisch' is not defined
```

from <modul> **import** <name> [, <name>]*

```
1  >>> from meinmodul import gruss
2  >>> gruss()
3  Hallo Modul Welt
```

Hauptprogramm mit `main`

Ziel

- Ausführen eins Code-Blocks nur, wenn es als Hauptprogramm gestartet wurde; nicht wenn es importiert wurde
- Simulieren der `main`-Funktion/Methode von C/Java
- Häufig zum Testen eines Moduls verwendet

Umsetzung, Idiom

- Der Name des interaktiven oder Haupt-Moduls ist `'__main__'`
- Test auf Name und bedingt ausführen

main.py

```
1 import sys
2
3 def drucke(x):
4    print "|%s|" % x
5
6 if __name__ == '__main__':
7    print "drucker"
8    for ele in sys.argv[1:]:
9       drucke(ele)
```

```
1 >>> import main
2 >>> main.drucke("hallo")
3 |hallo|
4 >>> main.drucke("hello")
5 |hello|
```

```
$ python main.py hallo hello
drucker
|hallo|
|hello|
```

Python – Do It Yourself!

Gegeben seien die folgenden Mitglieder von zwei Fernsehfamilien mit ihrem Alter:

```
surnames = ["homer", "marge", "bart", "walter", “skyler", "flynn"]
familynames = 3*[“simpson"] + 3*["white"]
ages = ages = [45, 40, 10, 55, 39, 19]
```

Schreiben Sie ein Modul `names.py` und implementieren Sie folgende Methoden (schreiben Sie in einem Modul `main_names.py` jeweils eine kleine Testroutine). Halten Sie Ihren Code mittels **funktionaler Konstrukte** möglichst kompakt.

- Eine Methode `zip_names()`, die (gegeben zwei Listen von Vor- und Nachnamen) eine Liste im Vormat “Nachname,Vorname” (z.B. [“simpson, homer”, ...]) zurückgibt.
- Eine Methode `list_by_age()`, die die Charaktere nach ihrem (absteigenden) Alter nach sortiert und eine Rangliste ausgibt (Hinweis: `sorted()` ist nützlich!):

```
1. White, Walter 55
2. Simpson, Homer 45 ...
```

- ...

Python – Do It Yourself!

- Eine Methode `simpsons()`, die (gegeben zwei Listen von Vor- und Nachnamen) nur die Vornamen der Simpsons-Mitglieder zurückgibt.
- Eine Methode, die ein Dictionary zurückgibt, das jeder Familie die Summe ihres Alters zuordnet.

Objektorientierung in Python

Ähnlich zu Java

- Schlüsselworte, Referenzsemantik
- Attribute, Vererbung, Polymorphismus
- Einfach, Sprache mit OO konzipiert

Unterschiede, neue Features

- Dynamische Attribute, Attributsuche
- Mehrfachvererbung, keine Schnittstellen
- Kapselung (Komposition) nur per Konvention, aber Properties
- Operatorenüberladung, Duck-Typing (reagieren auf Index-Zugriff, Ausgabe, Slicen)
- Selbst erstellte Klassen verhalten sich wie eingebaute Typen
- Alles! ist ein Objekt (Zahlen, Module, . . .)

```
1  >>> i=3
2  >>> type(i)
3  <type 'int'>
4  >>> isinstance(i, int)
5  True
6  >>> i.__add__
7  <method-wrapper '__add__'
8   of int object at 0x2152788>
9  >>> i.__add__(4)
10 7
11 >>> i.__mul__(4)
12 12
13 >>> object
14 <type 'object'>
15 >>> issubclass(int, object)
16 True
```

Klassen

Klassendefinition

- **`class`**: Schlüsselwort
- `<klasse>`: Name für die Klasse
- **`object`**: Wurzelklasse, immer angeben
- `<docstring>`: Dokumentationsstring
- `<definitionen>`: Funktionsdefinitionen für Methoden

Besonderheiten

- Erstes Methodenargument ist **immer** Instanzobjekt
  - Per Konvention der Name `self`, in Java wäre es `this`
- Spezielle Methode `__init__` statt Konstruktor
- Instanzattribute dynamisch (`__init__`) in Namensraum `self` schreiben

```
class <klasse>(object):
    [<docstring>]
    <definitionen>
```

klasse.py

```
1 class A(object):
2   "Eine Klasse A"
3   def __init__(self):
4     self.a = 17
5   def inca(self):
6     self.a += 1
7 print A
```

```
$ python klasse.py
<class '__main__.A'>
```

Ausführung erzeugt Klassenobjekt (Klassen sind Objekte!), binden Klassenname an Klassenobjekt

Klassendefinition – Beispiel Stack

Stack auf Basis von Listen (Komposition)

- Initialisierung der Instanzvariable `liste` mit leere Liste
- `self` ist immer das erste Argument bei Methoden
- `self` muss explizit aufgeführt werden bei der Definition
- `self` ist Namensraum einer Instanz
- Weitere Argumente bei Methoden erlaubt
- Manipulation der Instanzattribute in `self` sollte nur durch Methoden erfolgen
  - Keine `get/set` Methoden schreiben!
  - Später mit Properties

```
1  class Stack(object):
2    "Stack-Klasse mit Python Listen"
3
4    def __init__(self):
5      self.liste = []
6
7    def push(self, elem):
8      self.liste.append(elem)
9
10   def pop(self):
11     del self.liste[-1]
12
13   def top(self):
14     return self.liste[-1]
15
16   def clear(self):
17     self.liste = []
18
19   def empty(self):
20     return self.liste == []
```

Generieren und Nutzen von Instanzen

Klassendefinition

- Klassenname ist Name gebunden an Klassenobjekt
- Im Beispiel `Stack` in `stack.py`

Instanziierung

- Neue Instanz durch Aufruf Klassennamen (kein `new`)
  - Implizit wird `__init__` aufgerufen
- Zuweisung der Instanz an Namen
  - Name referenziert dann Instanzobjekt

Nutzen wie gewohnt

- Kapselung nicht erzwungen aber Konvention

```
>>> from stack import Stack
>>> Stack
<class 'stack.Stack'>
>>> s = Stack()
<stack.Stack object at 0x157fad0>
>>> s.empty()
True
>>> s.push(1); s.push(2); s.push(3)
>>> s.liste # Boese, nicht machen
[1, 2, 3]
>>> s.empty()
False
>>> s.top()
3
>>> s.pop()
>>> s.top()
2
```

Klassen- und Instanzvariablen

Instanzvariablen

- Zuweisung an durch `self` aufgespannten Namensraum
- Unabhängig von allen anderen Namen
- Entspricht Instanzvariablen in Java

Klassenvariablen

- Initialisierung als Attribute
- In durch Klassennamen aufgespannten Namensraum
- Entspricht `static` Variablen in Java
- Achtung: Syntax wie Instanzvariablen in Java

```
1  class K(object):
2    verbose = 1
3    def __init__(self, v):
4      self.verbose = v
5    def show_verbose(self):
6      print "Klasse", K.verbose
7      print "Instanz", self.verbose
```

```
1  >>> K.verbose
2  1
3  >>> i = K(2)
4  >>> i.verbose
5  2
6  >>> K.verbose
7  1
8  >>> i.show_verbose()
9  Klasse 1
10 Instanz 2
```

Klassen, Selbstbeobachtung

Umsetzung, Interna

- Attribute mit doppelten Unterstrichen
- Direkter Zugriff nur wenn notwendig

Klassen

- Name, Superklassen

Instanzen

- Klasse der Instanz
- Namensraum mit `__dict__`
- Realisiert mit Dictionaries

Selbst ausprobieren!

```
1  class K(object):
2    verbose = 1
3    def __init__(self): self.verbose = 1
4    def toggle(self): self.verbose = 1-self
```

```
1  >>> from klasse import K
2  >>> K.__name__ # der Name
3  'K'
4  >>> K.__bases__ # die Superklassen
5  (<type 'object'>,)
6  >>> k = K()
7  >>> k.__class__
8  <class 'klasse.K'>
9  >>> dir(k)
10 ['__class__', '__delattr__', '__dict__',
11  '__doc__', '__format__', '__getattribute
12  '__hash__', '__init__', '__module__', ..
13  '__str__', '__subclasshook__', '__weakre
14  'toggle', 'verbose']
```

Ausnahmen

Ausnahmen – spezieller Kontrollfluss

- Fokus auf Fehlerbehandlung
- Erlaubt saubere Trennung von funktionalen Bestandteilen und “Rest” wie Fehlerbehandlung und Spezialfällen
- Sollte nicht für allgemeinen Kontrollfluss verwendet werden

Ausnahmen in Python

- In Sprache integriert, einfach zu verwenden
- Ähnlich zu Java

Syntax

- **`try`**, **`else`**, **`except`**, **`finally`**: Rahmen für Code-Strecken mit potentiellen Ausnahmen, Fangen und Verarbeiten der Ausnahmen
- **`raise`**: Werfen von Ausnahmen
- **`assert`**: Wie in C

Ausnahmen fangen – Schema

- `<anweisungen>` im **`try`**-Block ausführen
- Falls Ausführung Ausnahme wirft mit `expect` fangen
- Eine oder mehrere Ausnahmen als Tupel möglich
- Mehrere **`except`** möglich
- Mit optionalem `as` Ausnahme-Instanz an `<name>` binden
- Falls Ausnahme gefangen entsprechenden **`except`**-Block ausführen
- **`else`**-Block (optional) falls keine Ausnahme geworfen wurde
- **`finally`**-Block (optional) immer, ob Ausnahme geworfen wurde (und gefangen wurde oder nicht) oder nicht

```
try:
    <anweisungen>
except <ausnahme>:
  <anweisungen>
except (<ausnahme>[, <ausnahme>]*):
  <anweisungen>
except <ausnahme> as <name>:
  <anweisungen>
else:
    <anweisungen>
finally:
    <anweisungen>
```

Ausnahmen fangen – Beispiele, `except`, `else`

```
>>> 3/0
ZeroDivisionError: integer div.. by zero
>>> try:
      3/0
   except ZeroDivisionError:
      print "nicht durch Null"
nicht durch Null
>>> try:
      3/0
   except ZeroDivisionError as e:
      print type(e), e
<type 'exceptions.ZeroDivisionError'>
   integer division or modulo by zero
>>> try:
      3/0
   except IndexError:
      print "IE"
   except ZeroDivisionError:
      print "ZDE"
ZDE
```

```
>>> try:
      3/0
   except ZeroDivisionError:
      print "ZDE"
   else:
      print "OK"
ZDE
>>> try:
      3/1
   except ZeroDivisionError:
      print "ZDE"
   else:
      print "OK"
3
OK
```

`ZeroDivisionError`, builtin Ausnahmetyp
Nicht gefangene Ausnahme propagiert zum TopLevel

Ausnahmen fangen – Beispiele, `finally`

```
>> try:
...   3/0
... except ZeroDivisionError:
...   print "ZDE"
... finally:
...   print "immer"
...
ZDE
immer
>>> try:
...   3/1
... except ZeroDivisionError:
...   print "ZDE"
... finally:
...   print "immer"
...
3
immer
```

```
>>> try:
...   3/0
... except IndexError:
...   print "IE"
... finally:
...   print "immer"
...
immer
ZeroDivisionError: integer division
              or modulo by zero
```

`finally` wird immer ausgeführt
Beinhaltet meist Aufräumaktionen

Ausnahmen fangen – Datei-Beispiel

Mit Ausnahmen arbeiten

- Code unter der Annahme schreiben, dass alles funktioniert
- Ausnahme und Behandlung bei Fehler

Mit Dateien arbeiten

- Zähle Zeilen in Datei
- 0, wenn Datei nicht vorhanden

```
1 def countlinesfile(filename):
2     try:
3         count = 0
4         for _ in file(filename):
5             count += 1
6     except IOError:
7         count = 0
8     return count
```

```
1 >>> countlinesfile("/etc/passwd")
2 44
3 >>> countlinesfile("/GIBTESNICHT")
4 0
```

Ausnahmeklassen – `Exception`, Hierarchie

```
BaseException
 +-- SystemExit
 +-- KeyboardInterrupt
 +-- GeneratorExit
 +-- Exception
     +-- StopIteration
     +-- StandardError
     |   +-- BufferError
     |   +-- ArithmeticError
     |   |   +-- FloatingPointError
     |   |   +-- OverflowError
     |   |   +-- ZeroDivisionError
     |   +-- AssertionError
     |   +-- AttributeError
     |   +-- EnvironmentError
     |   |   +-- IOError
     |   |   +-- OSError
     |   |       +-- WindowsError (Windows)
     |   |       +-- VMSError (VMS)
     |   +-- EOFError
     |   +-- ImportError
     |   +-- LookupError
     |   |   +-- IndexError
     |   |   +-- KeyError
     |   +-- MemoryError
     |   +-- NameError
     |   |   +-- UnboundLocalError
     |   +-- ReferenceError
     |   +-- RuntimeError
     |   |   +-- NotImplementedError
     |   +-- SyntaxError
     |   |   +-- IndentationError
     |   |       +-- TabError
     |   +-- SystemError
     |   +-- TypeError
     |   +-- ValueError
     |       +-- UnicodeError
     |           +-- UnicodeDecodeError
     |           +-- UnicodeEncodeError
     |           +-- UnicodeTranslateError
     +-- Warning
         +-- DeprecationWarning
         +-- PendingDeprecationWarning
         +-- RuntimeWarning
         +-- SyntaxWarning
         +-- UserWarning
         +-- FutureWarning
       +-- ImportWarning
         +-- UnicodeWarning
        +-- BytesWarning
```

Eigene Ausnahmeklassen passend ableiten

Ausnahmen werfen

Ausnahmen werfen mit **raise**

- **raise** <klasse>: Wirft Ausnahme der Klasse <klasse>, muss von BaseException erben
- **raise** <instanz>: Wirft Ausnahme der Instanz <instanz>, muss Instanz von BaseException ein

Instanzen oder Klassen werfen

- Instanz um Daten mitzugeben
- Häufig sind Daten ein String
  Vordefinierte Ausnahmen verwendbar

```
>>> raise Exception("nix geht")
Exception: nix geht
```

```
>>> class MeineAusnahme(object):
      pass
>>> raise MeineAusnahme
TypeError: exceptions must be old-style
        classes or derived from
        BaseException, not type

>>> class MeineAusnahme(Exception):
      pass
>>> raise MeineAusnahme
__main__.MeineAusnahme

>>> try:
      msg="Weil es nicht geht"
      raise MeineAusnahme(msg)
   except MeineAusnahme as inst:
      print "Warum?", inst
Warum? Weil es nicht geht
```

Python – Do It Yourself!

- Schreiben Sie eine Klasse `WordCounter`, die die Vorkommen eines Wortes in einer Text-Datei zählt.
  - Bei der Erstellung einer WordCounter-Instanz soll der Dateiname übergeben werden.
  - Die Klasse soll eine Methode `count(token)` anbieten, die – gegeben ein Wort – die Anzahl der Vorkommen des Wortes angibt. Kommt das Wort in der Datei nicht vor, soll 0 zurückgegeben werden.
  - Es sollen die Fehlermeldung “Datei nicht gefunden!” ausgegeben werden, falls die angegebene Datei nicht existiert.
- Schreiben Sie eine Subklasse `WordCounterEnhanced`, die in einem zusätzlichen Verarbeitungsschritt Satzzeichen entfernt.

Bibliotheken – Module in der Standardbibliothek

Python-Bibliotheken – viel im Standard

- Als Module/Pakete verfügbar
- Breites Angebot, “Batteries included”, ausführlich dokumentiert

Beispiele:

- `os`: Systemnah, Verzeichnisse/Dateien
  `sys`: Python-Umgebung
- `unittest`: Unit Testing
  `pickle`: Objektpersistenz
  `urllib, cgi, imap, ftp, http`: Web, Internet Protokolle
  `PIL`: Bildbearbeitung (nicht standard)
  `re`: Reguläre Ausdrücke

```
>>> import os
>>> os.getcwd()
'/home/mi/ssinn001'
>>> os.chdir("/usr/share/games/fortunes")
>>> os.getcwd()
'/usr/share/games/fortunes'
>>> [x for x in os.listdir(os.getcwd())
     if x.endswith(".dat")]
['fortunes.dat', 'riddles.dat',
 'literature.dat']
>>> import sys
>>> sys.argv
['']
>>> import re
>>> re.findall(r'ab+c', "abbcdsabcab")
['abbc', 'abc']
```

Pickle – Einfache Objektpersistenz

Ziel

- Serialisierung von Objektinstanzen
- Plattform- und versionsunabhängig
- Kein Persistenzframework

Umsetzung – `pickle`

- Für eingebaute Standardtypen
- Hierarchien und Referenzen
- Klassen, aber nur Instanzen

Beispiel

- Schreiben mit `dump`
- Lesen mit `load`

Aclass.py

```
1  class A(object):
2     def __init__(self, end=3):
3        self.lis = range(1, end)
4     def __str__(self):
5        return "A["+str(self.lis)+"]"
6     __repr__ = __str__
```

```
1  import pickle
2  import Aclass
3
4  tup=(1, 1.0, 1+1j, (1,), "1", {1:2})
5  lis=[1,2,3]
6  lis.append(lis)
7  alis=[Aclass.A(i) for i in range(1, 6)]
8  pickle.dump((tup, lis),
9           file("tuplis.pickle", "wb"))
10 pickle.dump(alis,
11          file("alis.pickle", "wb"))
```

Pickle – Einlesen

Beispiel

- Referenzen, auch zirkuläre, werden richtig wiederhergestellt
- Instanzen von Objekten werden richtig wieder instanziiert
  - Klasse/Modul muss verfügbar sein, wird nicht gepickelt
  - Einladen über **`import`** muss klappen

Erweiterungen

- Pickeln von Instanzen über Methoden beeinflussbar
  - Pickeln von Klassen oder Funktionen nicht möglich
- `cPickle`, schneller
- Unterschiedliche Protokollversionen

```
1  >>> import pickle
2  >>> f = file("tuplis.pickle", "rb")
3  >>> t, l = pickle.load(f)
4  >>> t
5  (1, 1.0, (1+1j), (1,), '1', {'1': 'eins'})
6  >>> l
7  [1, 2, 3, [...]]
8  >>> al = pickle.load(file("alis.pickle","rb"))
9  >>> al
10 [A[[]], A[[1]], A[[1, 2]], A[[1, 2, 3]],
11  A[[1, 2, 3, 4]]]
12 >>> al[0].__class__
13 <class 'Aclass.A'>
```

Numpy – Was ist das?

Weitverbreitetes Python-Modul für Arrays und Matrizen

- bietet zahlreiche mathematische Operationen der linearen Algebra:
  - Vektor-/Matrizenmultiplikation
  - Gleichungssysteme lösen
  - Operationen auf Bildern
  - Zahlreiche wissenschaftliche und mathematische Python-Pakete nutzen Numpy.
  - ...
- Komplexe Operationen als Einzeiler verfügbar

- Zusammen mit scipy:
  - Plotting
  - Optimierung
  - Data Mining
  - Spektralanalyse, Clustering, Statistische Kennwerte, schnelle räumliche Suche, ...

Numpy – Arrays

Wichtigster Datentyp: `array`

- homogenes, multidimensionales Array
- Konstruktion aus Liste/Tupel
- Spezielle Konstruktoren (`zeros`, `arange`, ...)
- Vektor/Matrix von Zahlen eines bestimmten Grundtyps (`dtype`)
  - z.B. `'float64'`, `'int64'`, ...

```
>>> import numpy as np
>>> a = np.array([1,2,3])
>>> b = np.zeros(2)
>>> print a,b
[1 2 3] [ 0. 0.]
>>> print type(a)
<type 'numpy.ndarray'>
```

```
>>> a = array([1,2,3])
>>> a.dtype
dtype('int64')
>>> b = array([1., 2., 3.])
>>> b.dtype
dtype('float64')
>>> a = array([1,2,3], dtype='float64')
>>> a.dtype
dtype('float64')
```

Numpy – Mehrdimensionale Arrays

Wichtigster Datentyp: `array`

- homogenes, **multidimensionales** Array
  - 1D-Array = Vektor
  - 2D-Array = Matrix, Grauwertbild
  - 3D-Array = Tensor, Farbbild
  - 4D-Array = Farbvideo
  - ...
- Form des Arrays: `a.shape` (gibt ein Tupel zurück)
- $n \times m$-Matrix $\Rightarrow$ `shape = (n,m)`.

```
>>> a = np.array( [ [1,2,3], [4,5,6] ] )
array([[1, 2, 3],
       [4, 5, 6]])
>>> a.shape
(2, 3)
>>> b = np.zeros( (2,3) ); b.shape
(2, 3)
>>> a = np.array([1,2,3])
>>> a.shape
(3,)
>>> c = np.ones( (2,2,2) )
>>> c
array([[[ 1., 1.],
        [ 1., 1.]],

       [[ 1., 1.],
        [ 1., 1.]]])
>>> c.shape
(2, 2, 2)
```

Numpy – Form von Arrays ändern

Die Form eines Arrays kann mit `reshape` geändert werden

- Gibt eine neue **View** auf die Daten zurück
  - (keine Kopie – das geht mit `a.copy()`!)
  - Keine Manipulation des Originals – das geht mit `a.resize()`!
- Anzahl der Einträge darf sich nicht ändern.

```
>>> r = np.arange(10000)
>>> r
array([ 0,  1,  2, ..., 9997, 9998, 9999])
>>> r.reshape(100,100)
array([[ 0,  1,  2, ..., 97, 98, 99],
       [ 100, 101, 102, ..., 197, 198, 199],
       [ 200, 201, 202, ..., 297, 298, 299],
       ...,
       [9700, 9701, 9702, ..., 9797, 9798, 9799],
       [9800, 9801, 9802, ..., 9897, 9898, 9899],
       [9900, 9901, 9902, ..., 9997, 9998, 9999]])
```

```
>>> a
array([[1, 2, 3],
       [4, 5, 6]])
>>> a.reshape(3, 2)
array([[1, 2],
       [3, 4],
       [5, 6]])
>>> a
array([[1, 2, 3],
       [4, 5, 6]])
>>> a.reshape(2, 4)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: total size of new array
must be unchanged
>>> b = a.reshape( (3,2) )
>>> a[0,0] = 0
>>> a
array([[0, 2, 3],
       [4, 5, 6]])
>>> b
array([[0, 2],
       [3, 4],
       [5, 6]])
```

Numpy – arange und elementweise Operationen

- `arange` – ähnlich zur Standard-Funktion `range`.
- Gibt allerdings ein array (keine Liste) zurück.
- Numpy bietet Funktionen, die **elementweise** auf die einzelnen Einträge des Arrays angewandt werden.
- Beispiele: `sin(), exp(), sqrt(), ...`
- Äquivalent für Listen: *list comprehension*.

```
>>> np.arange( 1, 5 )
array([1, 2, 3, 4])
>>> pi = np.pi
>>> x = np.arange( 0, 2*pi, pi/4 )
>>> x
array([ 0.   , 0.78539816,
       1.57079633, 2.35619449,
       3.14159265, 3.92699082,
       4.71238898, 5.49778714])
>>> np.sin(x)
array([ 0.000e+00, 7.071e-01,
        1.000e+00, 7.071e-01,
        1.224e-16, -7.071e-01,
       -1.000e+00, -7.071e-01])
```

```
>>> b = [1,2,3]
>>> [np.sin(e) for e in b]
[0.84147, 0.9092, 0.14112]
```

Numpy – Operatoren

- Die bekannten Standardoperatoren (-, +, * /, **) können auf arrays angewandt werden und wirken elementweise.
- Dasselbe gilt für boolesche Operatoren (==, <, <=, >, >=, !=). Das Ergebnis sind jeweils Arrays boolescher Werte.
- Die obigen Operatoren erzeugen jeweils neue Arrays. Hingegen haben die Varianten +=, *=, ... einen Seiteneffekt auf den linksseitigen Operanden.
- Achtung: “*” bezeichnet **nicht** das Skalarprodukt! Stattdessen: `numpy.dot()`.

```
1  >>> a = np.array([1,2,3,4])
2  >>> b = np.array([0,-1,1,2])
3  >>> a-b
4  array([1, 3, 2, 2])
5  >>> 10 * a
6  array([10, 20, 30, 40])
7  >>> a**2
8  array([ 1, 4, 9, 16])
9  >>> a < 3
10 array([ True,
   True, False, False], dtype=bool)
11 >>> a * b
12 array([ 0, -2, 3, 8])
13 >>> np.dot(a, b)
14 9
15 >>> a += b
16 >>> a
17 array([1, 1, 4, 6])
```

Numpy – Konversion des `dtype`

Achtung, wenn wir Arrays unterschiedlichen `dtypes` kombinieren:

- 'a+b': widening conversion.
- 'a+=b': b wird vor Addition auf `a.dtype` gecastet.

```
1 >>> a = np.array([1,2,3])
2 >>> b = np.random.random(3)
3 array([ 0.50797156, 0.3569057 ,
  0.42207076])
4 >>> a + b
5 array([ 1.50797156, 2.3569057 ,
  3.42207076])
6 >>> a += b; a
7 array([1, 2, 3])
```

Numpy – Vektor-/Matrix-Multiplikation

Die Funktion `numpy.dot()`

- Vektoren: Skalarprodukt
- Vektor, Matrix $\Rightarrow$ Vektor
- Matrix, Matrix $\Rightarrow$ Matrix
- Achtung: Dimensionen müssen passen! (`a.shape[1]==b.shape[0]`)

```
>>> b = np.array([1,2,3])
>>> np.dot(b, b)
14
>>> A = np.array([ [1,2,3], [4,5,6] ])
array([[1, 2, 3],
       [4, 5, 6]])
>>> np.dot(A, b)
array([14, 32])
>>> B = A.transpose()
array([[1, 4],
       [2, 5],
       [3, 6]])
>>> np.dot(A,B)
array([[14, 32],
       [32, 77]])
>>> np.dot(B,A)
array([[17, 22, 27],
       [22, 29, 36],
       [27, 36, 45]])
>>> np.dot(A,A)
Traceback (most recent call last):
  File "<console>", line 1, in <module>
ValueError: objects are not aligned
```

Summe, Minimum, Maximum, und Looping

- Funktionen zur Bestimmung von Minimum, Maximum, Summe.
- Zusatzparameter `axis`: Berechnung über Spalten/Zeilen.
- Wir können wie für Listen über die Elemente des Arrays loopen (Achtung: Gibt uns für 2D-Arrays die einzelnen Zeilen, nicht Werte!)

```
>>> a
array([[1, 2, 3],
       [4, 5, 6]])
>>> np.sum(a); np.min(a); np.max(a)
21, 1, 6
>>> np.sum(a, axis=0)
array([5, 7, 9])
>>> np.sum(a, axis=1)
array([ 6, 15])
```

```
>>> a = np.arange(16)
>>> for val in a: print val
0
1
2
...
15
>>> a.resize(4,4)
>>> for line in a: print line
[0 1 2 3]
[4 5 6 7]
[ 8 9 10 11]
[12 13 14 15]
```

Slicing

... funktioniert genau wie für Listen

- Syntax: `a[von:bis:step]`
- Kanonische Erweiterung für mehrdimensionale Arrays
- Weglassen einer Dimension: Alle Werte werden ausgeliefert.
- Beispiel: Wähle jede zweite Zeile/Spalte (entspricht Skalierung eines Bildes um Faktor $\frac{1}{2}$)

```
>>> a
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> a[1:20:2] # out-of-bounds: Kein Fehler!
array([1, 3, 5, 7, 9])
>>> a[-3:]
array([7, 8, 9])
>>> a[::-1]  # = Umkehrung der Reihenfolge
array([9, 8, 7, 6, 5, 4, 3, 2, 1, 0])
>>> A
array([[ 0, 1, 2, 3, 4],
       [ 5, 6, 7, 8, 9],
       [10, 11, 12, 13, 14],
       [15, 16, 17, 18, 19],
       [20, 21, 22, 23, 24]])
>>> A[1:3,]  # Zeilen 2-3
array([[ 5, 6, 7, 8, 9],
       [10, 11, 12, 13, 14]])
>>> A[1:3,2:4] # Zeilen 2-3, Spalten 3-4
array([[ 7, 8],
       [12, 13]])
>>> A[::2,::2] # Jede zweite Zeile/Spalte
               # (= Herunterskalieren eines Bildes)
array([[ 0, 2, 4],
       [10, 12, 14],
       [20, 22, 24]])
```

... und mehr

Zahlreiche weitere Funktionen für `arrays`:

- Konkatenieren, stacken (`vstack(A,B)`)
- Check von Bedingungen (`indices = where(A > 3)`)
- Quantoren-Check (`all(A>3)`)
- Selektion (`a.argmin()`)
- Sortierung (`sorted(a)`)
- Statistische Kennwerte (`var(a)`)
- ...

Weitere Ressourcen:
http://wiki.scipy.org/Tentative_NumPy_Tutorial
Bressert, Eli (2012). Scipy and Numpy: An Overview for Developers.

NumPy – Do It Yourself!

- Lesen Sie das Bild ’`test.jpg`’ mit Hilfe des folgenden Codes als numpy-array ein...

```
1 import cv2
2 import numpy as np
3
4 image = cv2.imread('test.jpg')
```

- Das vorliegende Array ist ein dreidimensionales Farbbild. Konvertieren Sie es in ein Grauwertbild, indem Sie die Werte der drei Farbkanäle mitteln.
- Schneiden Sie das rechte untere Viertel des Bildes aus und speichern Sie es in ein neues Bild (z.B. `cv2.imwrite('out.jpg', img_result)`)