

# Machine Learning

## Python-Crashkurs

Prof. Dr. Peter Barth (Adrian Ulges)

Hochschule RheinMain

Fachbereich Design Informatik Medien

Medieninformatik / Angewandte Informatik

18. Oktober 2016



# Outline

## 1 Basics

- Erste Schritte
- Kollektionen
- Kontrollstrukturen
- Funktionen
- List Comprehension
- Module
- Bibliotheken

## 2 Numpy

## 3 “Fortgeschrittenes”

- Objektorientierung
- Ausnahmen

# 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 (Verketteten, abbilden, slicen)
- Klassische Kontrollstrukturen (imperativ), Objektorientierung und funktionale Primitive
- Unterstützung für große Projekte (Module, Ausnahmen, ...)
- Integration (C, C++, Java, XMLRPC, REST, SOAP, ...)
- Bibliotheken (Statistik, Web, Email, reguläre Ausdrücke, XML, GUI, Threading, Unittesting, Bildverarbeitung, ...)

# 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>

Position Nov 2012	Position Nov 2011	Delta In Position	Programming Language	Ratings Nov 2012	Delta Nov 2011	Status
1	2	↑	C	19.224%	+1.90%	A
2	1	↓	Java	17.455%	-0.42%	A
3	6	↑↑↑	Objective-C	10.383%	+4.40%	A
4	3	↓	C++	9.698%	+1.61%	A
5	5	=	PHP	5.732%	-0.36%	A
6	4	↓↓	C#	5.591%	-1.73%	A
7	7	=	(Visual) Basic	5.032%	-0.01%	A
8	8	=	Python	4.062%	+0.45%	A
9	10	↑	Perl	2.182%	+0.10%	A
10	11	↑	Ruby	1.739%	+0.24%	A

# Lektüre

## Online-Lektüren (neben Buch)

- [docs.python.org/2/tutorial/](https://docs.python.org/2/tutorial/)
- [wiki.python.org/moin/BeginnersGuide](http://wiki.python.org/moin/BeginnersGuide)
- [pythonchallenge.com](http://pythonchallenge.com), [checkio.org](http://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

```

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!

# Outline

- 1 Basics
  - Erste Schritte
    - Kollektionen
    - Kontrollstrukturen
    - Funktionen
    - List Comprehension
    - Module
    - Bibliotheken
- 2 Numpy
- 3 “Fortgeschrittenes”
  - Objektorientierung
  - Ausnahmen

# 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

```
1 >>> x = 3
2 >>> x
3 3
4 >>> y
5 NameError: name 'y' is not defined
6 >>> x = 3
7 >>> x = "Hallo"
8 >>> x = 3; x + 3
9 6
10 >>> x = "3"; x + 3
11 TypeError: cannot concatenate
12     'str' and 'int' objects
```

# Erste Schritte – Zahlen und Ausdrücke

```
1 >>> print "Hallo Python Welt"
2 Hallo Python Welt
3
4 >>> # Kommentare
5
6
7 >>> 2 # Integer-Zahlen
8 2
9
10 >>> 3.14 # Fließkommazahlen
11 3.14
12
13 >>> "String" # Zeichenketten
14 'String'
15
16 >>> 0xff # Hex
17 255
18
19 >>> 0377 # Oktal
20 255
```

```
1 >>> 6+2 # Ausdrücke
2 8
3
4 >>> 6*2
5 12
6
7 >>> 6/2
8 3
9
10 >>> 6/5 # Vorgabe ganzzahlig (2.x)
11 1
12
13 >>> 6//5 # Explizit ganzzahlig ([23].x)
14 1
15
16 >>> 3**4 # Exponenten
17 81
18
19 >>> 3+1j # Komplexe Zahlen
20 (3+1j)
```

# Erste Schritte – Zuweisungen und Tests

```
1 >>> x = 1 # Variablenzuweisung
2 >>> x # hat den zugewiesenen Wert
3 1
4
5 >>> x = y = 2 # Mehrfachzuweisung
6 >>> x
7 2
8 >>> y
9 2
10
11 >>> abs(-3) # eingebaute Funktionen
12 3
13
14 >>> 0 < 1 # Tests
15 True
16 >>> 1 < 0
17 False
```

```
1 >>> True == 1 # Bool neuer Typ
2 True
3 >>> False == 0
4 True
5
6 >>> 0 < 5 < 10 # Mehrfachtest
7 True
8
9 >>> True and False
10 False
11 >>> True or False
12 True
13
14 >>> not "eins"
15 False
16 >>> not False
17 True
18
19 >>> "eins" and "zwei" # letzter Wert
20 'zwei'
```

# Erste Schritte – Division

```
1 >>> 3/2 # ganzzahlige Division (2.x)
2 1
3 >>> 3//2 # ganzzahlige Division ([23].x)
4 1
5 >>> 3.0/2 # Hochkonvertieren
6 1.5
7 >>> 3.0//2 # Abschneiden
8 1.0
9 >>> 3./2 # 0 nicht notwendig
10 1.5
11 >>> 10 % 3 # Module, Rest
12 1
13 >>> 10 // 3 # ganzzahlig
14 3
15 >>> 3*(10//3) + (10%3)
16 10
17 >>> 10.0//3 # erzwungen ganzzahlig
18 3.0 # anderer Typ
19 >>> 10.0/3
20 3.3333333333333335
```

# Große Zahlen und Konvertierung

```
1 >>> 3e10 # Exponentenschreibweise
2 30000000000.0
3
4 >>> 2**10 # Potenzieren
5 1024
6 >>> 2**20 # grosse ganze Zahl
7 1048576
8 >>> 2**1000 # sehr grosse ganze Zahl
9 107150860718626732094842504906000181
10 056140481170553360744375038837035105
11 112493612249319837881569585812759467
12 291755314682518714528569231404359845
13 775746985748039345677748242309854210
14 746050623711418779541821530464749835
15 819412673987675591655439460770629145
16 711964776865421676604298316526243868
17 37205668069376L
```

```
1 >>> 3L # explizit grosse Zahlen
2 3L
3 >>> float(3) # explizite Konvertierung
4 3.0 # in double
5 >>> int(3.0) # und zurueck
6 3
7 >>> long(3.0) # explizite Konvertierung
8 3L
9 >>> int("12") # auch fuer Strings
10 12
11 >>> int("11", 16) # explizite Basis
12 17
13
14 >>> int("zwoelf") # nur wenn moeglich
15 # ansonsten Ausnahme
16 Traceback (most recent call last):
17   File "<stdin>", line 1, in <module>
18   ValueError: invalid literal for int()
19     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

```

1 >>> lis = [3,1,2]
2 >>> len(lis) # Funktionsaufruf
3 3
4 >>> lis.append(8) # Methode
5 >>> lis # Ausgabe
6 [3, 1, 2, 8]
7 >>> len(lis) < 6 and len(lis)
8 # Zusammengesetzt
9 4
10 >>> 1 < len(lis) < 8 # Bereich
11 True

```

```

1 >> lis
2 [3, 1, 2]
3 >>> print lis # 2.x
4 [3, 1, 2]
5 >>> print(lis) # 2.x, 3.x
6 [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

```
1 >>> lis = [1,2,3]
2 >>> print lis, "zwei", 4
3 [1, 2, 3] zwei 4
4 >>> print lis,"zwei",4; print "weiter"
5 [1, 2, 3] zwei 4
6 weiter
7 >>> print lis,"zwei",4,; print "weiter"
8 [1, 2, 3] zwei 4 weiter
9 >>> print "%s %s" % ("Hallo", "Welt")
10 Hallo Welt
11 >>> print(lis)
12 [1, 2, 3]
```

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

# Zeichenkette, Strings – Darstellung

## Syntax

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

```
1 >>> "hallo"
2 'hallo'
3 >>> 'hallo'
4 'hallo'
```

```
1 >>> "spam's"
2 "spam's"
3 >>> 'spam"s'
4 'spam"s'
5 >>> 'spam\'s'
6 "spam's"
```

## Mehrzeilige Strings

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

```
1 >>> """hallo
2 ... du "tolle"
3 ... 'perfekte' Welt """
4 '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)

```

1 >>> "Hallo" + " Welt"
2 'Hallo Welt'
3 >>> "Hallo"*3
4 'HalloHalloHallo'
5 >>> "Hallo"[0]
6 'H'
7 >>> "Hallo"[-1]
8 'o'

```

```

1 >>> "Hallo"[1:3]
2 'al'
3 >>> "Hallo"[1:]
4 'allo'
5 >>> "Hallo"[:-1]
6 'Hall'

```

# Slicing

"SLICEOFSPAM"[von:bis]

- von, einschließlich
- bis, ausschließend

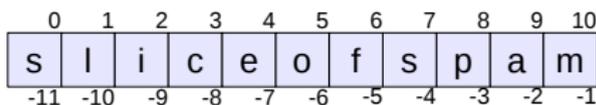
Vorzeichen

- 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)



```

1 >>> "sliceofspam"[2:-1]
2 'iceofspa'
3 >>> "sliceofspam"[5:]
4 'ofspam'
5 >>> "sliceofspam"[:5]
6 'slice'
7 >>> "sliceofspam"[2:5]
8 'ice'

```

```

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

```
1 >>> "ein %s Papagei" % "toter"
2 'ein toter Papagei'
3 >>> "%d %s Papageie" % (2, "tote")
4 '2 tote Papageie'
5 >>> "%x" % 42
6 '2a'
7 >>> "%X" % 42
8 '2A'
9 >>> "%e %f %g" % (1.1, 1.2, 1.3)
10 '1.100000e+00 1.200000 1.3'
11 >>> "%%%s%" % "hallo"
12 '%hallo%'
```

# Strings – Builtins

## Eingebaute Funktionen, Builtins

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

```
1 >>> len("Hallo")
2 5
3 >>> "hallo" < "wallo"
4 True
5 >>> "al" in "Hallo"
6 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']
```

```
1 >>> import string
2 >>> "ab".upper(), string.upper("ab")
3 ('AB', 'AB')
4 >>> string.find("spammify", "mm")
5 3
6 >>> string.split("spammify", "mm")
7 ['spa', 'ify']
8 >>> 'spam no\nspam'.split()
9 ['spam', 'no', 'spam']
10 >>> "XX".join(["spa", "ify"])
11 'spaXXify'
12 >>> string.strip(" Meaning of Life \n")
13 'Meaning of Life'
```

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

# Python – Do It Yourself!

- Öffnen Sie den Python-Interpreter (“python” in der Shell)
- Berechnen Sie die Zahl  $x = 2^{1000} - 1$ .
- 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?



# Outline

## 1 Basics

- Erste Schritte
- **Kollektionen**
- Kontrollstrukturen
- Funktionen
- List Comprehension
- Module
- Bibliotheken

## 2 Numpy

## 3 "Fortgeschrittenes"

- Objektorientierung
- Ausnahmen

# Kollektionen

## Eingebaute Kollektionen

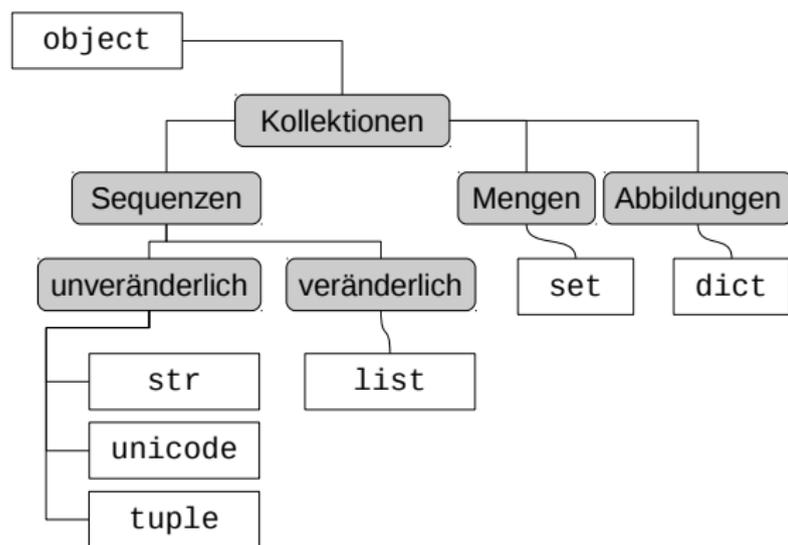
- Eingebaute Typen zum Verwalten von mehreren Objekten
- Teil der Sprache, nicht nur der Bibliothek

## 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**

```
1 >>> tup = (1, "zwei", (3, 4))
2 >>> len(tup)
3 3
4 >>> tup + tup
5 (1, 'zwei', (3, 4), 1, 'zwei', (3, 4))
6 >>> tup*2
7 (1, 'zwei', (3, 4), 1, 'zwei', (3, 4))
8 >>> (3, 4) in tup
9 True
```

```
1 >>> 1, 2
2 (1, 2)
3 >>> (1, 2)
4 (1, 2)
5 >>> (a, b) = (1, 2)
6 >>> print a, b
7 1 2
8 >>> b, a = a, b
9 >>> a, b
10 (2, 1)
```

# Tupel – Unveränderliche Sequenz

Tupel sind unveränderlich

- Operationen mit Tupel erzeugen neue Tupel
- Achtung! Tupel kann veränderliche Werte als Tupelemente haben

```
1 >>> tup = (1,2,3)
2 >>> tup[1] = "zwei"
3 Traceback (most recent call last):
4   File "<stdin>", line 1, in <module>
5   TypeError: 'tuple' object does not
6       support item assignment
7 >>> tup = (1, [], 3)
8 >>> tup[1].append("zwei")
9 >>> tup
10 (1, ['zwei'], 3)
```

# Listen – Veränderliche 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 >>> [1, 2, 3]
2 [1, 2, 3]
3 >>> [1], []
4 [1], []
5 >>> [1, [2, "zwei"], (3, "drei")]
6 [1, [2, 'zwei'], (3, 'drei')]
7 >>> lis = [1, [2, "zwei"], (3, "drei")]
8 >>> lis[0]
9 1
10 >>> lis[1:]
11 [[2, 'zwei'], (3, 'drei')]
12 >>> lis[2][0], len(lis)
13 (3, 3)
14 >>> lis + [4,5]
15 [1, [2, 'zwei'], (3, 'drei'), 4, 5]
16 >>> 1 in lis
17 True
18 >>> (lis*2)[1:-2]
19 [[2, 'zwei'], (3, 'drei'), 1]
```

# 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

```

1 >>> lis = [4, 3, 2.5, 2, 1, 0, -1]
2 >>> sorted(lis)
3 [-1, 0, 1, 2, 2.5, 3, 4]
4 >>> lis.sort(); lis
5 [-1, 0, 1, 2, 2.5, 3, 4]
```

```

1 >>> range(5)
2 [0, 1, 2, 3, 4]
3 >>> range(1, 5)
4 [1, 2, 3, 4]
5 >>> range(0, 50, 10)
6 [0, 10, 20, 30, 40]
7 >>> lis = range(5)
8 >>> lis.append(5); lis
9 [0, 1, 2, 3, 4, 5]
10 >>> lis.pop()
11 5
12 >>> lis
13 [0, 1, 2, 3, 4]
14 >>> lis.insert(0, -1); lis
15 [-1, 0, 1, 2, 3, 4]
16 >>> lis.insert(4, 2.5); lis
17 [-1, 0, 1, 2, 2.5, 3, 4]
18 >>> lis.reverse(); lis
19 [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

```

# Dictionaries – Builtins und Schlüsseleigenschaften

## Eingebaute Funktionen

- `len` für Anzahl Schlüssel
- `del` zum Löschen
- Keine Slicing-Operationen  
Dictionary ist keine Sequenz

## Schlüssel unveränderlich

- Intern mit Hashing realisiert
- Zahlen, Strings, etc.
- Tupel (auch tiefe Strukturen) sind erlaubt

## Ausnahme bei nicht vorhandenem Schlüssel

```
1 >>> dic = {'eggs': 42, 'spam': 3}
2 >>> len(dic)
3 2
4 >>> del dic['eggs']; dic
5 {'spam': 3}
6 >>> dic[0:]
7 Traceback (most recent call last):
8   File "<stdin>", line 1, in <module>
9   TypeError: unhashable type
10 >>> dic[(1,2)] = 'tupel'
11 >>> dic
12 {(1, 2): 'tupel', 'spam': 3}
13 >>> dic[[1,2]] = 'liste'
14 Traceback (most recent call last):
15   File "<stdin>", line 1, in <module>
16   TypeError: unhashable type: 'list'
17 >>> dic['cheese']
18 Traceback (most recent call last):
19   File "<stdin>", line 1, in <module>
20   KeyError: 'cheese'
```

# 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 `&`

```
1 >>> s = set()
2 >>> s.add(1)
3 >>> s
4 set([1])
5 >>> s.add(2); s.add(3)
6 >>> s
7 set([1, 2, 3])
8 >>> s.add(2)
9 >>> s
10 set([1, 2, 3])
11 >>> 2 in s
12 True
13 >>> 4 in s
14 False
15 >>> len(s)
16 3
17 >>> s | set([2,3,4])
18 set([1, 2, 3, 4])
19 >>> s & set([2,3,4])
20 set([2, 3])
```

# Outline

## 1 Basics

- Erste Schritte
- Kollektionen
- **Kontrollstrukturen**
- Funktionen
- List Comprehension
- Module
- Bibliotheken

## 2 Numpy

## 3 “Fortgeschrittenes”

- Objektorientierung
- Ausnahmen

# 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 <bedingung>:  
    <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 inklusiv, bis exklusiv
- 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
```

# Kontrollstrukturen – for-Schleife, typischer Einsatz

## Klassische for-Schleife

- Durchlaufen von Index-Werten
- Sequenz von Index-Werten, mit **range** erzeugt
- Selten benötigt
- Java/C Idiom, weniger Python

## Feldzugriff vermeiden

- Keine expliziten Indexvariablen
- Eleganter
- Kein Indexfehler möglich!

```
1 for idx in range(100): # 0-99 in Python
2     print idx
```

```
1 // 0-99 in Java
2 for(int i=0; i < 100; i+=1) {
3     System.out.println(idx);
4 }
```

```
1 for ele in arr: # Feldzugriff in Python
2     print ele
```

```
1 // Feldzugriff in Java
2 for(int i=0; i < arr.length; i+=1) {
3     System.out.println(arr[i]);
4 }
```

# Python – Do It Yourself!

- Lesen Sie den Text von Bram Stoker's Roman "Dracula" aus der Datei "dracula.txt" ein.
- Splitten Sie den Text in einzelne Wörter (*tokens*) 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"?



# Outline

## 1 Basics

- Erste Schritte
- Kollektionen
- Kontrollstrukturen
- **Funktionen**
- List Comprehension
- Module
- Bibliotheken

## 2 Numpy

## 3 “Fortgeschrittenes”

- Objektorientierung
- Ausnahmen

# 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

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

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

## Auch interaktiv eingebbar

- Leerzeile zum Beenden

The screenshot shows a Python Shell window with the following content:

```
Python 2.7.3 (default, Aug 1 2012, 05:14:39)
[GCC 4.6.3] on linux2
Type "copyright", "credits" or "license()" for more information.
>>> def mult(x, y):
        "Multipliziert zwei Objekte x,y"
        return x*y

>>> mult(
(x, y)
Multipliziert zwei Objekte x,y
```

The status bar at the bottom right of the window indicates "Ln: 8 Col: 9".

# 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
- Stellungparameter, 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
```

# Parameterübergabe mit Schlüsselwort

## Ziel

- Viele Vorgabewerte verwenden, aber einige Parameter anders
- Zwang Angabe aller früheren Parameter auflösen

## Schlüsselwortparameter

- Parameter über Namen zuweisen
- Name des Parameters als linke Seite einer Zuweisung im Aufruf
- Reihenfolge der Parameter beliebig
- Für alle Argumente außer den Schlüsselwortparametern Vorgabe
- Parameter ohne Vorgabewert müssen als Positions- oder Schlüsselwortparameter übergeben werden

```
1 def suche(seq, ele, start=0, ende=None):
2     if ende is None:
3         ende = len(seq)
4     for pos in range(start, ende):
5         if seq[pos] == ele:
6             return pos
```

```
1 >>> lis = [1, 2, 3, 4, 5, 6]
2 >>> suche(lis, 2, start=0, ende=len(lis))
3 1
4 >>> suche(lis, 2, 0, len(lis))
5 1
6 >>> suche(lis, 2, ende=len(lis))
7 1
8 >>> suche(lis, 2, start=1, ende=len(lis))
9 1
10 >>> suche(lis, ende=len(lis), ele=2)
11 1
12 >>> suche(lis, 2, ende=1)
13 >>>
```

# Outline

- 1 Basics
  - Erste Schritte
  - Kollektionen
  - Kontrollstrukturen
  - Funktionen
  - **List Comprehension**
  - Module
  - Bibliotheken
- 2 Numpy
- 3 "Fortgeschrittenes"
  - Objektorientierung
  - Ausnahmen

# List Comprehension

## Listentransformation

- Anwendungsgebiet wie funktionales Programmieren (siehe “Fortgeschrittenes”) **map** und **filter**

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

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

## Ausdrucksweisen

- Statt verschachtelte **for**-Schleifen
- Ähneln math. Mengenschreibweise

```

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

```

```

1 >>> [x**2 for x in range(1, 11)
2         if x % 2 == 0]
3 [4, 16, 36, 64, 100]

```

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

# Outline

## 1 Basics

- Erste Schritte
- Kollektionen
- Kontrollstrukturen
- Funktionen
- List Comprehension
- **Module**
- Bibliotheken

## 2 Numpy

## 3 “Fortgeschrittenes”

- Objektorientierung
- Ausnahmen

# Module

## Warum Module?

- Wiederverwendung von Code
- Bereitstellung von allgemein 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`

```
1 >>> import math
2 >>> math.sqrt(9)
3 3.0
4 >>> from math import sqrt
5 >>> sqrt(9)
6 3.0
```

# 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 eines 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!

- Wir repräsentieren einen Kniffel-Wurf als ein 5-Tupel von Zahlen zwischen 1 und 6 (z.B. `roll = (3,1,5,4,4)`).
- Schreiben Sie ein Modul `kniffel.py` mit einer Funktion `is_small_straight(roll)`, die – gegeben einen Kniffel-Wurf – entscheidet ob eine kleine Straße geworfen wurde.
- Zählen Sie mittels eines List-Comprehension-Ausdrucks (a la `[... for roll1 in range(1,7) for roll2 in range(1,7) ...]`), wieviele mögliche Kombinationen eine kleine Straße ergeben.
- Finden Sie heraus, ob es mehr Kombinationen für eine kleine Straße gibt als für ein Full House (indem Sie eine entsprechende Funktion `is_full` schreiben und verwenden).



# Outline

## 1 Basics

- Erste Schritte
- Kollektionen
- Kontrollstrukturen
- Funktionen
- List Comprehension
- Module
- **Bibliotheken**

## 2 Numpy

## 3 "Fortgeschrittenes"

- Objektorientierung
- Ausnahmen

# 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

```
1 >>> import os
2 >>> os.getcwd()
3 '/home/mi/ssinn001'
4 >>> os.chdir("/usr/share/games/fortunes")
5 >>> os.getcwd()
6 '/usr/share/games/fortunes'
7 >>> [x for x in os.listdir(os.getcwd())
8     if x.endswith(".dat")]
9 ['fortunes.dat', 'riddles.dat',
10  'literature.dat']
11 >>> import sys
12 >>> sys.argv
13 ['']
14 >>> import re
15 >>> re.findall(r'ab+c', "abbcdsabcb")
16 ['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'>
```

# Outline

- 1 Basics
  - Erste Schritte
  - Kollektionen
  - Kontrollstrukturen
  - Funktionen
  - List Comprehension
  - Module
  - Bibliotheken
- 2 Numpy
- 3 “Fortgeschrittenes”
  - Objektorientierung
  - Ausnahmen

# 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', ...

```

1 >>> import numpy as np
2 >>> a = np.array([1,2,3])
3 >>> b = np.zeros(2)
4 >>> print a,b
5 [1 2 3] [ 0.  0.]
6 >>> print type(a)
7 <type 'numpy.ndarray'>

```

```

1 >>> a = array([1,2,3])
2 >>> a.dtype
3 dtype('int64')
4 >>> b = array([1., 2., 3.])
5 >>> b.dtype
6 dtype('float64')
7 >>> a = array([1,2,3], dtype='float64')
8 >>> a.dtype
9 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)`.

```

1 >>> a = np.array( [ [1,2,3], [4,5,6] ] )
2 array([[1, 2, 3],
3        [4, 5, 6]])
4 >>> a.shape
5 (2, 3)
6 >>> b = np.zeros( (2,3) ); b.shape
7 (2, 3)
8 >>> a = np.array([1,2,3])
9 >>> a.shape
10 (3,)
11 >>> c = np.ones( (2,2,2) )
12 >>> c
13 array([[[ 1., 1.],
14         [ 1., 1.]],
15
16        [[ 1., 1.],
17         [ 1., 1.]])
18 >>> c.shape
19 (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.

```

1 >>> r = np.arange(10000)
2 >>> r
3 array([ 0,  1,  2, ..., 9997, 9998, 9999])
4 >>> r.reshape(100,100)
5 array([[ 0,  1,  2, ..., 97, 98, 99],
6        [100, 101, 102, ..., 197, 198, 199],
7        [200, 201, 202, ..., 297, 298, 299],
8        ...,
9        [9700, 9701, 9702, ..., 9797, 9798, 9799],
10       [9800, 9801, 9802, ..., 9897, 9898, 9899],
11       [9900, 9901, 9902, ..., 9997, 9998, 9999]])

```

```

1 >>> a
2 array([[1, 2, 3],
3        [4, 5, 6]])
4 >>> a.reshape(3, 2)
5 array([[1, 2],
6        [3, 4],
7        [5, 6]])
8 >>> a
9 array([[1, 2, 3],
10       [4, 5, 6]])
11 >>> a.reshape(2, 4)
12 Traceback (most recent call last):
13   File "<stdin>", line 1, in <module>
14 ValueError: total size of new array
15 must be unchanged
16 >>> b = a.reshape( (3,2) )
17 >>> a[0,0] = 0
18 >>> a
19 array([[0, 2, 3],
20        [4, 5, 6]])
21 >>> b
22 array([[0, 2],
23        [3, 4],
24        [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*.

```

1 >>> np.arange( 1, 5 )
2 array([1, 2, 3, 4])
3 >>> pi = np.pi
4 >>> x = np.arange( 0, 2*pi, pi/4 )
5 >>> x
6 array([ 0.      ,  0.78539816,
7         1.57079633,  2.35619449,
8         3.14159265,  3.92699082,
9         4.71238898,  5.49778714])
10 >>> np.sin(x)
11 array([ 0.000e+00,  7.071e-01,
12         1.000e+00,  7.071e-01,
13         1.224e-16, -7.071e-01,
14        -1.000e+00, -7.071e-01])

```

```

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

```

1 >>> b = np.array([1,2,3])
2 >>> np.dot(b, b)
3 14
4 >>> A = np.array([ [1,2,3], [4,5,6] ])
5 array([[1, 2, 3],
6        [4, 5, 6]])
7 >>> np.dot(A, b)
8 array([14, 32])
9 >>> B = A.transpose()
10 array([[1, 4],
11         [2, 5],
12         [3, 6]])
13 >>> np.dot(A,B)
14 array([[14, 32],
15         [32, 77]])
16 >>> np.dot(B,A)
17 array([[17, 22, 27],
18         [22, 29, 36],
19         [27, 36, 45]])
20 >>> np.dot(A,A)
21 Traceback (most recent call last):
22   File "<console>", line 1, in <module>
23 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!)

```

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

```

```

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

```

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

### Weitere Ressourcen:

[http://wiki.scipy.org/Tentative\\_NumPy\\_Tutorial](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.
- Können Sie sich eine Strategie überlegen, mit deren Hilfe Sie die Figur im Bild vom Hintergrund trennen können?



# Outline

- 1 Basics
  - Erste Schritte
  - Kollektionen
  - Kontrollstrukturen
  - Funktionen
  - List Comprehension
  - Module
  - Bibliotheken
- 2 Numpy
- 3 "Fortgeschrittenes"
  - Objektorientierung
  - Ausnahmen

# Eigenschaften von Strings

Strings sind unveränderlich

- Operation mit Strings (+, [], etc.) erzeugen immer neue Strings
- Vorhandene Strings werden niemals verändert

Unicode Strings

- u vor dem String
- Typ **unicode**, nicht **str**

```

1 >>> s = "hallo"
2 >>> s[0] = "H"
3 Traceback (most recent call last):
4   File "<stdin>", line 1, in <module>
5   TypeError: 'str' object does not
6     support item assignment

```

```

1 >>> s = u"hallo"
2 >>> s
3 u'hallo'
4 >>> s == "hallo"
5 True
6 >>> type("hallo"), type(u"hallo")
7 (<type 'str'>, <type 'unicode'>)
8 >>> u"hallo" == "hallo"
9 True

```

```

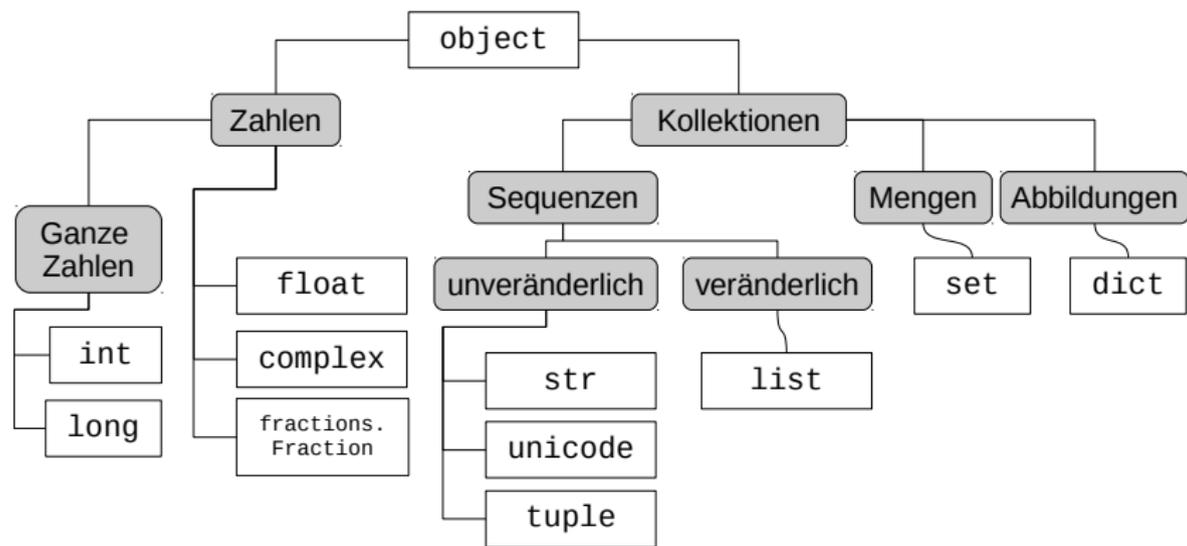
>>> "hällo" == u"hällo"
__main__:1: UnicodeWarning: Unicode
equal comparison failed...

```

# Typhierarchie eingebauter Typen

object als Wurzel für alle Typen  
 isinstance zum Testen

```
1 >>> isinstance(3, int)
2 True
3 >>> isinstance(3, object)
4 True
```



# Dateien

## Modul `os` für Systemoperationen

- `chdir` um Verzeichnis zu wechseln
- `getcwd` um Verzeichnis anzuzeigen
- Unix und Windows Notation

## Dateien

- `file` ist eigener Datentyp
- Konstruktor mit Dateiname und Bearbeitungsart (r lesen, w schreiben, a erweitern)
- `read` zum Lesen der gesamten! Datei
- `readlines` für String-Liste der Zeilen
- `close` Schließen optional, über Garbage Collector
- Schreiben funktioniert ähnlich

```

1 >>> import os
2 >>> os.chdir('/usr/share/dict')
3 >>> os.getcwd()
4 '/usr/share/dict'
5 >>> eingabe = file("words", "r")
6 >>> inhalt = eingabe.read()
7 >>> len(inhalt)
8 938848
9 >>> inhalt[700539:700600]
10 "python\npython's\npythons\npyx\npyx's
11 \npyxes\nq\nqua\nquack\nquack's\nq"
12 >>> eingabe.close()
13 >>> os.chdir('/tmp')
14 >>> ausgabe = file('wordskopie', 'w')
15 >>> ausgabe.write(inhalt)
16 >>> ausgabe.close()

```

# Zuweisung und Ausdrücke

## Gleichheit und Identität

- Alles ist ein Objekt
- Gleichheit `==`, Objektidentität `is`

## Zuweisung

- Variablen sind Namen, die an Objekte gebunden sind
- Rechte Seite einer Zuweisung gibt Objekt (vorhandenes oder neues) zurück
- Linke Seite einer Zuweisungen ist ein Name an den Objekt gebunden wird

```

1 >>> lis = [1, 2, 3]
2 >>> kat = [1, 2, 3]
3 >>> lis == kat, lis is kat
4 (True, False)
5 >>> lis.append(4)
6 >>> lis
7 [1, 2, 3, 4]
8 >>> kat
9 [1, 2, 3]
10 >>> lis = [1, 2, 3]
11 >>> kat = lis[:] # Kopie
12 >>> lis == kat, lis is kat
13 (True, False)

```

# Referenzsemantik

## Referenzsemantik

- Variablen sind Namen, die an Objekte gebunden sind
- Rechte Seite einer Zuweisung gibt Objekt (vorhandenes oder neues) zurück
- Linke Seite einer Zuweisungen ist eine Name an den Objekt gebunden wird

```
1 >>> lis = [1, 2, 3]
2 >>> kat = lis
3 >>> lis == kat, lis is kat
4 (True, True)
5 >>> lis.append(4)
6 >>> lis
7 [1, 2, 3, 4]
8 >>> kat
9 [1, 2, 3, 4]
```

# Gültigkeitsbereiche in Funktionen

Funktionsaufrufe sind Namensraum

- Jeder Funktionsaufruf bietet einen neuen Gültigkeitsbereich (Namensraum)
- Die Definition auch

Beispiel f, g

- Variable g wird nicht verändert
  - Weder bei der Definition
  - Noch bei dem Aufruf
- g in der Funktion ist eine neue lokale Variable

```

1 >>> g = 1
2 >>> def f(x, y):
3     g=2
4     return x+y
5
6 >>> g
7 1
8 >>> f(5, 6)
9 11
10 >>> g
11 1

```

# Parameterübergabe

Parameterübergabe ist Namensbindung  
Verwendung

- Call by Reference
- Das Ändern eines änderbaren Objekts hat Auswirkungen (Seiteneffekte)
- Bei unveränderlichen Objekten kann es keine Auswirkungen haben, wie call by value

```
1 def aendern(x, y):
2     x = 2
3     y[1] = 1
```

```
1 >>> x, y = ([1,2], [3,4])
2 >>> aendern(x, y)
3 >>> x, y
4 ([1, 2], [3, 1])
```

## 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()

```

# Funktionale Primitive – filter

Nur Elemente für die Funktion ok ist

- Für alle Objekte einer Sequenz eine Funktion anwenden, nur die durchlassen, für die der Funktionswert äquivalent zu wahr ist
- Schlüsselwort **filter**
- Erster Parameter die Funktion
- Zweiter Parameter die Sequenz
- Ergebnis ist Liste der ursprünglichen Elemente, deren Funktionswert äquivalent zu wahr ist

Beispiele

- Test ob Zahl größer 0

```

1 >>> def groesser0(x):
2       return x>0
3 >>> def gerade(x):
4       return x%2==0
5 >>> filter(groesser0, [1, -1, 2, -2, 3, -3, 4])
6 [1, 2, 3, 4]
7 >>> filter(gerade, [1, 2, 3, 4, 5])
8 [2, 4]

```

# Einsatzbeispiel für filter

## Aufgabe

- Berechne die Liste der Primzahlen zwischen 2 und 100

## Lösung

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

## Funktionale Lösung

- Einzeiler mit **filter**
- Liste der Zahlen mit **range**
- Funktion zum Testen ob Primzahl wie bei **for**-Schleife
- Keine Schleifen

```

1 def is_prim(zahl):
2     for teiler in range(2, zahl):
3         if zahl%teiler == 0:
4             return False
5     return True
6 def prims():
7     primzahlen = []
8     for zahl in range(2, 100):
9         if is_prim(zahl):
10            primzahlen.append(zahl)
11    return primzahlen
12 print prims()

```

```

[2, 3, 5, 7, 11, 13, 17, 19, 23, 29,
31, 37, 41, 43, 47, 53, 59, 61, 67,
71, 73, 79, 83, 89, 97]

```

```

1 def prims():
2     return filter(is_prim, range(2, 100))

```

# List Comprehension – Beispiele mit Zahlen

## Quadratzahlen

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

```

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

```

## Komplexe Definition

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

```

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

```

## Kartesisches Produkt

- Sequenzen kombinieren
- Kartesisches Produkt, Vollkombination

```

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

```

## zip

- Builtin, generiert Tupel

# Funktionale Programmierung – Zusammenfassung

## Vorteile

- Verminderung der Fehleranfälligkeit, zum Beispiel sind viele Indexfehler gar nicht mehr formulierbar
- Kompakter, ausdrucksstarker Code nahe an der Spezifikation
- Gut zu lesen und zu verstehen, wenn nicht zu viel in einer Zeile passiert
- Seiteneffektfrei (falls richtig implementiert)

## Nachteile

- Syntax/Ausdrucksweise etwas gewöhnungsbedürftig
- Funktionales Denken erfordert etwas Einarbeitungszeit

## Online-Referenzen

- <http://www-106.ibm.com/developerworks/linux/library/l-prog.html>
- <http://www.freenetpages.co.uk/hp/alan.gauld/tutfctnl.htm>
- ...

# 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

```

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

```

## 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

```

1 >>> d = "zahlen.dat"
2 >>> file(d).readlines()
3 ['1 2 3\n', '42 343 12\n',
4  '8 77 223\n', '23 43\n',
5  '45 3 3 66 7\n']
6 >>> [tuple(map(int, line.split()))
7       for line in file(d)]
8 [(1, 2, 3), (42, 343, 12), (8, 77, 223),
9  (23, 43), (45, 3, 3, 66, 7)]

```

# Outline

- 1 Basics
  - Erste Schritte
  - Kollektionen
  - Kontrollstrukturen
  - Funktionen
  - List Comprehension
  - Module
  - Bibliotheken
- 2 Numpy
- 3 "Fortgeschrittenes"
  - Objektorientierung
  - Ausnahmen

# 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 Klassennamen 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
```

# Vererbung und Mehrfachvererbung

## Vererbung

- Superklassen in Klammern dahinter
- **object** nur dann nicht mehr notwendig
  - Ganz ohne Superklassen werden es Old-Style Klassen, deprecated
- Mehrfachvererbung möglich

## Mehrfachvererbung

- Suche nach Attribut oder Methodenname
  - Erst in aktueller Klasse
  - Dann in Superklassen wie definiert
  - Erst alle Superklassen der Superklassen
- Kurz: von links nach rechts, Tiefensuche\*

\*In der Realität etwas komplizierter

<http://www.python.org/download/releases/2.3/mro/>

```
class <name>(<superkl> [, <superkl> ]*):
```

```
...
fancystack.py
```

```
1 from stack import Stack
2
3 class FancyStack(Stack):
4     def peek(self, i):
5         return self.liste[i]
```

```
1 >>> from fancystack import FancyStack
2 >>> s = FancyStack()
3 >>> s
4 <fancystack.FancyStack object at 0x1c31f9
5 >>> s.push(1); s.push(2); s.push(3)
6 >>> s.peek(2)
7 3
8 >>> s.peek(0)
9 1
```

# 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
```

# Statische Methoden

## Statische Methoden

- Klassenmethode, bezieht sich nicht auf Instanz, in Java `static`
- Syntax: Dekorator `@staticmethod`
- Kein erstes Argument `self` mehr vorhanden

```
1 class K(object):
2     zaehle = 1
3     def __init__(self):
4         self.zahl = 17
5     @staticmethod
6     def statisch(x):
7         K.zaehle += x
8     @classmethod
9     def klassenmethode(cls, x):
10        cls.zaehle += x
```

```
1 >>> k = K()
2 >>> k.zaehle, K.zaehle
3 (1, 1)
4 >>> k.statisch(3)
5 >>> K.zaehle
6 4
7 >>> k.klassenmethode(3)
8 >>> K.zaehle
9 7
```

# 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.verbose

```

```

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__', '__weakref__',
14  'toggle', 'verbose']

```

# Von eingebauten Klassen ableiten

## Beispiel eigenes Wörterbuch

- Verhalten wie `dict`, aber nur Strings als Schlüssel
- Alles andere gleich

## Umsetzung

- Von `dict` ableiten, Vererben alternativ zu Komposition
- Überladen von `__setitem__`
- Nutzen von `__setitem__` der Superklasse

```
1 class strdict(dict):
2     def __setitem__(self, key, value):
3         if type(key) != str:
4             msg = "nur String als Key"
5             raise AttributeError(msg)
6         dict.__setitem__(self, key, value)
```

```
1 >>> dic = dict()
2 >>> dic[1] = 2
3 >>> dic["eins"] = 2
4 >>> dic
5 {1: 2, 'eins': 2}
6 >>> strdic = strdict()
7 >>> strdic[1] = 2
8 AttributeError: nur String als Key
9 >>> strdic["eins"] = 2
10 >>> strdic
11 {'eins': 2}
```

# Outline

- 1 Basics
  - Erste Schritte
  - Kollektionen
  - Kontrollstrukturen
  - Funktionen
  - List Comprehension
  - Module
  - Bibliotheken
- 2 Numpy
- 3 "Fortgeschrittenes"
  - Objektorientierung
  - **Ausnahmen**

# 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 `except` fangen
- Eine oder mehrere Ausnahmen als Tupel möglich
- Mehrere **except** möglich
- Mit optionalem `an` Ausnahme-Instanz `<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

```

1 >>> 3/0
2 ZeroDivisionError: integer div.. by zero
3 >>> try:
4     3/0
5     except ZeroDivisionError:
6         print "nicht durch Null"
7 nicht durch Null
8 >>> try:
9     3/0
10    except ZeroDivisionError as e:
11        print type(e), e
12 <type 'exceptions.ZeroDivisionError'>
13 integer division or modulo by zero
14 >>> try:
15     3/0
16    except IndexError:
17        print "IE"
18    except ZeroDivisionError:
19        print "ZDE"
20 ZDE

```

```

1 >>> try:
2     3/0
3     except ZeroDivisionError:
4         print "ZDE"
5     else:
6         print "OK"
7 ZDE
8 >>> try:
9     3/1
10    except ZeroDivisionError:
11        print "ZDE"
12    else:
13        print "OK"
14 3
15 OK

```

ZeroDivisionError, builtin Ausnahmetyp  
 Nicht gefangene Ausnahme propagiert  
 zum TopLevel

## Ausnahmen fangen – Beispiele, finally

```
1 >> try:
2 ... 3/0
3 ... except ZeroDivisionError:
4 ...     print "ZDE"
5 ... finally:
6 ...     print "immer"
7 ...
8 ZDE
9 immer
10 >>> try:
11 ... 3/1
12 ... except ZeroDivisionError:
13 ...     print "ZDE"
14 ... finally:
15 ...     print "immer"
16 ...
17 3
18 immer
```

```
1 >>> try:
2 ... 3/0
3 ... except IndexError:
4 ...     print "IE"
5 ... finally:
6 ...     print "immer"
7 ...
8 immer
9 ZeroDivisionError: integer division
10 ... 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

```

1 BaseException
2 +-- SystemExit
3 +-- KeyboardInterrupt
4 +-- GeneratorExit
5 +-- Exception
6     +-- StopIteration
7     +-- StandardError
8         | +-- BufferError
9         | +-- ArithmeticError
10        | | +-- FloatingPointError
11        | | +-- OverflowError
12        | | +-- ZeroDivisionError
13        | +-- AssertionError
14        | +-- AttributeError
15        | +-- EnvironmentError
16        | | +-- IOError
17        | | +-- OSError
18        | |     +-- WindowsError (Windows)
19        | |     +-- VMSError (VMS)
20        | +-- EOFError
21        | +-- ImportError
22        | +-- LookupError
23        | | +-- IndexError
24        | | +-- KeyError
25        | +-- MemoryError
26        | +-- NameError
27        | +-- UnboundLocalError

```

```

1 | +-- ReferenceError
2 | +-- RuntimeError
3 | | +-- NotImplementedError
4 | +-- SyntaxError
5 | | +-- IndentationError
6 | |     +-- TabError
7 | +-- SystemError
8 | +-- TypeError
9 | +-- ValueError
10 |     +-- UnicodeError
11 |         +-- UnicodeDecodeError
12 |         +-- UnicodeEncodeError
13 |         +-- UnicodeTranslateError
14 +-- Warning
15     +-- DeprecationWarning
16     +-- PendingDeprecationWarning
17     +-- RuntimeWarning
18     +-- SyntaxWarning
19     +-- UserWarning
20     +-- FutureWarning
21 +-- ImportWarning
22     +-- UnicodeWarning
23 +-- 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

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

```
1 >>> class MeineAusnahme(object):
2     pass
3 >>> raise MeineAusnahme
4 TypeError: exceptions must be old-style
5     classes or derived from
6     BaseException, not type
7
8 >>> class MeineAusnahme(Exception):
9     pass
10 >>> raise MeineAusnahme
11 __main__.MeineAusnahme
12
13 >>> try:
14     msg="Weil es nicht geht"
15     raise MeineAusnahme(msg)
16 except MeineAusnahme as inst:
17     print "Warum?", inst
18 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.



# Unit Testing

## Ziel

- Funktionalitäten feingranular testen
- Automatisierung, einfache Verwaltung von Testfällen

## Umsetzung – unittest, Verwendung

- Anlehnung an JUnit
- Von Klasse TestCase erben
- Testfunktionen mit test\_ am Anfang des Namens
- Zusicherung, self.<name>: Erwartet
  - assert\_: True, wie **assert**
  - assertEquals: Zwei gleich Parameter
  - assertRaises: Passende Ausnahme
- setUp, tearDown: Vor/nach jedem Test, Test Fixture
- Aufrufen mit unittest.main()

```
import unittest
class <testfall>(unittest.TestCase):
    [def setUp(self):
        ...
    ]
    [def tearDown(self):
        ...
    ]
    [def test<einzeltest>(self):
        ...
        [self.assert_(<testausdruck>) |
        [self.assertEqual(<o1>, <o2>) |
        [self.assertRaises(<ausnahme>)]*
        ...
    ]*
if __name__ == '__main__':
    unittest.main()
```

Anpassbar, flexibel, weitere Optionen

# Unit Testing – Beispiel

## Beispieltest "Ist Primzahl?"

- Importieren Modul
- Zu testende Funktion `is_prim`, meist in anderem Modul
- Testfunktionen
- Start der Tests

## Ausgabe

- Zwei Tests ok durchgelaufen

..

-----  
Ran 2 tests in 0.000s

OK

```

1 import unittest
2
3 def is_prim(x):
4     if x < 2: return False
5     ts = [t for t in range(2, x) if x%t==0]
6     return not ts
7
8 class PrimTest(unittest.TestCase):
9     def setUp(self):
10        self.r = range(1, 101) # 1..100
11        self.ps = 25 # 25 Primzahlen
12    def testPrim1(self):
13        self.assert_(is_prim(2))
14        self.assert_(not is_prim(4))
15        self.assertEqual(is_prim(7), True)
16    def testPrim2(self):
17        primes = filter(is_prim, self.r)
18        self.assertEqual(len(primes), self.ps)
19 if __name__ == '__main__':
20    unittest.main()

```

# Internetdaten verarbeiten

- rfc822, mimetools, mimetypes, mailcap, imaplib, mailbox, smtplib, nntplib: Email- und News-spezifische Bibliotheken
- uu, binascii: Konvertierungen
- SocketServer, SimpleHTTPServer, ftplib: Protokollimplementierungen
- urllib, urllib2, urlparse: URL-Manipulation, Anfragen

```
1 >>> import urllib2, re
2 >>> s = urllib2.urlopen("http://www.heise.de/newsticker/").read()
3 >>> s[:100]
4 '<!DOCTYPE html>\n<html lang="de">\n\n<head>\n \n
5 \n\n <title>7-Tage-News | heise online</title>\n\n '
6 >>> hl = re.findall(r'<a href="/newsticker/meldung/.*">(.*?)</a>', s)
7 >>> for t in [t for t in hl if "Android" in t]: print t
8 Unzuverlaessige Trojaner-Warnungen durch Android 4.2
9 LTE-Smartphone mit Android 4.0 von HTC
10 Java-IDE IDEA 12: neue Oberflaeche, neuer Compiler-Modus, neuer Android-Designer
11 Gingerbread bleibt die am meisten benutzte Android-Version
12 Aktualisierte Maps-API fuer Android-Entwickler
```