

# Python für Computerlinguisten

Matthias Bethke

`bethke@linguistik.uni-erlangen.de`

Linguistische Informatik  
Universität Erlangen-Nürnberg

Wintersemester 2006/2007

# Organisatorisches

- 1 Termin
- 2 Voraussetzungen und Ablauf
- 3 Projekt?
- 4 Literatur
- 5 WWW: <http://www.linguistik.uni-erlangen.de/~msbethke/teaching/WS2006-07-Python.html>

# Warum Python?

- Weil ich einen Schein brauche...?

# Warum Python?

- „*Python is executable pseudocode.*  
*Perl is executable line noise.*“

# Warum Python?

- „*Python is executable pseudocode.*  
*Perl is executable line noise.*“

→ Unterstützt sauberes und strukturiertes Programmieren.

# Warum Python?

- „*Python is executable pseudocode.*  
*Perl is executable line noise.*“
- Unterstützt sauberes und strukturiertes Programmieren.
- Bietet für Parser u.ä. Anwendungen interessante Konstrukte.

# Warum Python?

- „*Python is executable pseudocode.*  
*Perl is executable line noise.*“
- Unterstützt sauberes und strukturiertes Programmieren.
- Bietet für Parser u.ä. Anwendungen interessante Konstrukte.
  - Mittlerweile recht beliebte Skriptsprache für (Web-)anwendungen (→ Zope/Plone, HPLIP, Systemprogramme in Gentoo Linux, ...), sollte man mal gesehen haben.

# Allgemeine Charakteristika

- Skriptsprache: kompiliert unmittelbar vor der Laufzeit
  - Kurze Entwicklungszyklen („mal eben...“)
  - Laufzeitnachteil gegenüber kompilierten Sprachen

# Allgemeine Charakteristika

- Skriptsprache: kompiliert unmittelbar vor der Laufzeit  
→ Kurze Entwicklungszyklen („mal eben...“)  
→ Laufzeitnachteil gegenüber kompilierten Sprachen
- Relativ kleiner Sprachumfang und einfache Syntax

# Allgemeine Charakteristika

- Skriptsprache: kompiliert unmittelbar vor der Laufzeit  
→ Kurze Entwicklungszyklen („mal eben...“)  
→ Laufzeitnachteil gegenüber kompilierten Sprachen
- Relativ kleiner Sprachumfang und einfache Syntax
- Unterstützt Modularisierung

# Allgemeine Charakteristika

- Skriptsprache: kompiliert unmittelbar vor der Laufzeit
  - Kurze Entwicklungszyklen („mal eben...“)
  - Laufzeitnachteil gegenüber kompilierten Sprachen
- Relativ kleiner Sprachumfang und einfache Syntax
- Unterstützt Modularisierung
- Prozedural und/oder objektorientiert zu programmieren; funktionale Elemente

# Allgemeines zu Python-Skripten

- Einfache Textdateien, üblicherweise mit der Endung „.py“ (nicht notwendig)
- Vorsicht mit verschiedenen Zeilenumbrüchen (DOS vs. Unix), kann zu rätselhaften Fehlern führen

# Allgemeines zu Python-Skripten

- Einfache Textdateien, üblicherweise mit der Endung „.py“ (nicht notwendig)
- Vorsicht mit verschiedenen Zeilenumbrüchen (DOS vs. Unix), kann zu rätselhaften Fehlern führen

## Kopfzeile zur Angabe des Interpreters

```
#!/usr/bin/python
```

Anschließend: `chmod +x meinskript.py`.

# Allgemeines zu Python-Skripten

- Einfache Textdateien, üblicherweise mit der Endung „.py“ (nicht notwendig)
- Vorsicht mit verschiedenen Zeilenumbrüchen (DOS vs. Unix), kann zu rätselhaften Fehlern führen

## Kopfzeile zur Angabe des Interpreters

```
#!/usr/bin/python
```

Anschließend: `chmod +x meinskript.py`.

## Alternativen auf der Kommandozeile

```
python [Optionen] meinskript.py  
python [Optionen] -c '<Anweisungen>'
```

# Python interaktiv

Im Unterschied z.B. zu Perl hat der Python-Interpreter einen interaktiven Modus:

## Interaktiver Modus

```
$ python
Python 2.4.3 (#1, Sep 28 2006, 15:51:33)
[GCC 3.4.6 (Gentoo 3.4.6-r1, ssp-3.4.5-1.0,
pie-8.7.9)] on linux2
Type "help", "copyright", "credits" or "license" for
more information.
>>> 1+1
2
>>> print "hallo"
hallo
>>>
```

# Hello, world!

## Das üblicherweise erste Programm

```
#!/usr/bin/python
hello="Hello, world!"
print hello
```

---

<sup>1</sup>Semikolons können verwendet werden, um mehrere Statements auf einer Zeile zu trennen.

# Hello, world!

## Das üblicherweise erste Programm

```
#!/usr/bin/python
hello="Hello, world!"
print hello
```

- Kein Abschluss von Statements mit Semikolon<sup>1</sup>

---

<sup>1</sup>Semikolons können verwendet werden, um mehrere Statements auf einer Zeile zu trennen.

# Hello, world!

## Das üblicherweise erste Programm

```
#!/usr/bin/python
hello="Hello, world!"
print hello
```

- Kein Abschluss von Statements mit Semikolon<sup>1</sup>
- Variablen „entstehen“ automatisch bei der ersten Zuweisung an sie.

---

<sup>1</sup>Semikolons können verwendet werden, um mehrere Statements auf einer Zeile zu trennen.

# Hello, world!

## Das üblicherweise erste Programm

```
#!/usr/bin/python
hello="Hello, world!"
print hello
```

- Kein Abschluss von Statements mit Semikolon<sup>1</sup>
- Variablen „entstehen“ automatisch bei der ersten Zuweisung an sie.
- Eingebaute Funktion `print` zur Textausgabe

---

<sup>1</sup>Semikolons können verwendet werden, um mehrere Statements auf einer Zeile zu trennen.

# Datentypen: Numerische Werte

- Ganzzahlwerte (Integers): normale (natürliche Wortlänge der Architektur, mind. 32-bit) und „lange“ Werte (Suffix l; beliebig lang)
- Können als Literale dezimal, oktal (Präfix 0) oder hexadezimal (Präfix 0x) angegeben werden.

# Datentypen: Numerische Werte

- Ganzzahlwerte (Integers): normale (natürliche Wortlänge der Architektur, mind. 32-bit) und „lange“ Werte (Suffix l; beliebig lang)
- Können als Literale dezimal, oktal (Präfix 0) oder hexadezimal (Präfix 0x) angegeben werden.
- Besonderheit: imaginäre Zahlen als eigener Datentyp mit Suffix j. Können nicht direkt als Literal angegeben werden, sondern nur in Verbindung mit einem Realteil, z.B.  $1+3j$ .

# Datentypen: Numerische Werte

- Ganzzahlwerte (Integers): normale (natürliche Wortlänge der Architektur, mind. 32-bit) und „lange“ Werte (Suffix l; beliebig lang)
- Können als Literale dezimal, oktal (Präfix 0) oder hexadezimal (Präfix 0x) angegeben werden.
- Besonderheit: imaginäre Zahlen als eigener Datentyp mit Suffix j. Können nicht direkt als Literal angegeben werden, sondern nur in Verbindung mit einem Realteil, z.B.  $1+3j$ .
- Fließkommawerte (grundsätzlich doppelte Genauigkeit): Literale mit Nachkommaanteil und/oder Exponent, z.B. 3.1415,  $1e100$
- Ganzzahlwerte werden bei Berechnungen *nicht* automatisch nach Fließkomma gewandelt!

# Datentypen: Strings (1)

- Strings können in einzelne oder doppelte Anführungszeichen eingeschlossen sein:  
einstring = "Hallo\t"  
nocheiner = 'Welt\n'
- Eine Variableninterpolation wie in Perl oder entsprechende Unterscheidung der Strings nach Anführungszeichen findet nicht statt.

# Datentypen: Strings (1)

- Strings können in einzelne oder doppelte Anführungszeichen eingeschlossen sein:

```
einstring = "Hallo\t"  
nocheiner = 'Welt\n'
```

- Eine Variableninterpolation wie in Perl oder entsprechende Unterscheidung der Strings nach Anführungszeichen findet nicht statt.
- Escape-Sequenzen, eingeleitet mit Backslash, werden ähnlich wie in C interpretiert. Die Wichtigsten:

`\n` : „NewLine“, Zeilenumbruch

`\r` : „Carriage Return“, Zeilenvorschub

`\t` : Tabulator

`\\` : Backslash

`\xxx` : Byte mit numerischem Wert *xxx*

`\uxxxx` : Unicode-Zeichen mit Code *xxxx*

## Datentypen: Strings (2)

- Besonderheit in Python: *Triple-quoted Strings*
  - Eingeschlossen in Dreiergruppen von einfachen oder doppelten Anführungszeichen
  - Einfache Anführungszeichen, Newlines etc. dürfen enthalten sein:

```
langerstring = """Das Programm hat folgende Optionen:  
-l: lange Ausgabe  
-R: rekursiv arbeiten  
-a: abstuerzen"""
```

## Datentypen: Strings (2)

- Besonderheit in Python: *Triple-quoted Strings*
  - Eingeschlossen in Dreiergruppen von einfachen oder doppelten Anführungszeichen
  - Einfache Anführungszeichen, Newlines etc. dürfen enthalten sein:

```
langerstring = """Das Programm hat folgende Optionen:  
-l: lange Ausgabe  
-R: rekursiv arbeiten  
-a: abstuerzen"""
```

- *Raw Strings* (Präfix `r`)
  - Interpretieren überhaupt keine Escape-Sequenzen

```
rawstring = r"He\r\ebe\\Back\\slashes"
```
- *Unicode-Strings* (Präfix `u`): unabhängig vom Zeichensatz des Programms oder der aktuellen Locale immer im ISO-10646-Zeichensatz

```
uni = u"Auslaendisches:\uabcd\u1234"
```

## Datentypen: Strings (3)

- Im Gegensatz zu C und Perl sind alle Strings unveränderliche Objekte
- Vergleichbar mit C: die Behandlung von Strings als (konstantes!) „Array“ von Zeichen<sup>2</sup>

```
>>> print "foobar"[1]
o
>>> print "foobar"[1:3]
oo
>>> print "foobar"[0:6:2]
foa
```

---

<sup>2</sup>Vgl. Folien 17 und 21

## Datentypen: Strings (3)

- Im Gegensatz zu C und Perl sind alle Strings unveränderliche Objekte
- Vergleichbar mit C: die Behandlung von Strings als (konstantes!) „Array“ von Zeichen<sup>2</sup>

```
>>> print "foobar"[1]
o
>>> print "foobar"[1:3]
oo
>>> print "foobar"[0:6:2]
foa
```

- Ein einzelnes Zeichen hat keinen eigenen Datentyp sondern ist nur ein String der Länge 1.

---

<sup>2</sup>Vgl. Folien 17 und 21

# Kontrollstrukturen: allgemeines

Eine der charakteristischsten Eigenheiten von Python ist seine Blockstruktur:

C, Perl, Java, ...	Python
control	control ":"
<statement ";"   block>	<<statement> <";" statement>*>   <indent statement>+

- Python ist *nicht formatfrei!*
- Wo andere Sprachen nach einem **if**, **while**, etc. ein Statement oder einen geklammerten Block erwarten, nimmt Python eine Liste von Statements *oder* den gegenüber dem Kontrollstatement *eingerrückten* Block als davon abhängig.
- Es können Leerzeichen oder Tabs in beliebiger Menge zum Einrücken verwendet werden. Wichtig ist nur: *genau gleich viele* für den *gesamten Block!*

# Kontrollstrukturen: if

Die `if`-Konstruktion unterscheidet sich nicht wesentlich von anderen Programmiersprachen:

```
if expression : statement_list
if expression :
    statement
...
else :
    statement
...
```

# Kontrollstrukturen: `while`

Die `while`-Schleife bietet bis auf den **else**-Teil ebenfalls wenig aufregendes:

```
while expression : statement_list
```

```
while expression :  
    statement  
    ...
```

```
else :  
    statement  
    ...
```

- Besonderheit: der **else**-Teil wird ausgeführt, wenn die Schleife *nicht* mittels **break** verlassen wurde.
- **continue** funktioniert wie in C etc.

# Kontrollstrukturen: do-while

Eine `do-while`-Schleife existiert in Python nicht, kann aber leicht emuliert werden.

Statt:

```
do :  
    statement  
while expression
```

funktioniert folgendes:

```
while True :  
    statement  
    if (not expression) : break
```

# Kontrollstrukturen: for

- Die `for`-Schleife iteriert grundsätzlich über Listen, nicht über einen numerisch spezifizierten Wertebereich:

```
for elem in [ "foo", "bar", "baz" ]: print elem
```

- **continue**, **break** und **else** funktionieren hier genauso wie bei **while**.

# Kontrollstrukturen: for

- Die `for`-Schleife iteriert grundsätzlich über Listen, nicht über einen numerisch spezifizierten Wertebereich:

```
for elem in [ "foo", "bar", "baz" ]: print elem
```

- **continue**, **break** und **else** funktionieren hier genauso wie bei **while**.
- Die Iteration über einen Wertebereich wird mittels `range()` emuliert:

```
for i in range(0,10): print i
```
- `range(start, end)` liefert eine Liste von Werten von `start` inklusive bis `end` exklusive. Optional kann eine Schrittweite angegeben werden, z.B. `range(10, -20, -5)`.

# Datentypen: Listen (1)

Eine Liste ist eine Sequenz von Werten oder *Items* beliebigen (auch unterschiedlichen) Typs. Sie wird in eckigen Klammern geschrieben und die Werte mit Kommata getrennt:

```
liste = [ 1, 2, None, "ene", 'mene', '''muh  
und_▯raus_▯bist_▯du''' ]
```

Listenelemente werden wie üblich per Subskript ab 0 angesprochen:

```
print liste [0]
```

# Datentypen: Listen (1)

Eine Liste ist eine Sequenz von Werten oder *Items* beliebigen (auch unterschiedlichen) Typs. Sie wird in eckigen Klammern geschrieben und die Werte mit Kommata getrennt:

```
liste = [ 1, 2, None, "ene", 'mene', '''muh
und_▯raus_▯bist_▯du''' ]
```

Listenelemente werden wie üblich per Subskript ab 0 angesprochen:

```
print liste [0]
```

Teillisten können über *Slices* adressiert werden. Die Bereichsangabe gilt dabei (wie bei `range`) *exklusive* des letzten Wertes:

```
teil = liste [2:4] # [None, 'ene']
```

Auch Slices können eine Schrittweite beinhalten. Überschreitet ein Slice die Länge der Liste, ist dies im Gegensatz zur Adressierung von Einzelementen kein Fehler:

```
teil = liste [2:100:2] # [None, 'mene']
```

## Datentypen: Listen (2)

Sowohl einzelne Elemente als auch Slices sind modifizierbar:

```
liste [5] = "foo"           # [1,2, None, 'ene ', 'mene', 'foo ']  
liste [1:3] = [42]         # [1, 42, 'ene ', 'mene', 'foo ']  
liste [1:6:2] = [123,456,789] # [1, 123, None, 456, 'mene', 789]
```

Bei Slices mit Schrittweitenangabe muss die Länge der zugewiesenen Liste allerdings der Länge des Slices entsprechen!

## Datentypen: Listen (2)

Sowohl einzelne Elemente als auch Slices sind modifizierbar:

```
liste [5] = "foo"           # [1,2, None, 'ene ', 'mene', 'foo ']
liste [1:3] = [42]         # [1, 42, 'ene ', 'mene', 'foo ']
liste [1:6:2] = [123,456,789] # [1, 123, None, 456, 'mene', 789]
```

Bei Slices mit Schrittweitenangabe muss die Länge der zugewiesenen Liste allerdings der Länge des Slices entsprechen!

### Multiplikation und Addition von Listen

```
>>> liste = [ 1, 1, 2, 3, 5, 8 ]
>>> print liste * 2
[1, 1, 2, 3, 5, 8, 1, 1, 2, 3, 5, 8]
>>> a = [1,2]
>>> b = [3,4]
>>> print a + b
[1, 2, 3, 4]
```

# Datentypen: Tupel (1)

- Ein Tupel ist eine unveränderliche Liste und wird im Unterschied zu dieser in runden Klammern notiert:

```
>>> t = ("eins", 2, 3+4j)
>>> print t[2]
(3+4j)
>>> t[0] = 2
[...]
TypeError: object does not support item assignment
```

# Datentypen: Tupel (1)

- Ein Tupel ist eine unveränderliche Liste und wird im Unterschied zu dieser in runden Klammern notiert:

```
>>> t = ("eins", 2, 3+4j)
```

```
>>> print t[2]
```

```
(3+4j)
```

```
>>> t[0] = 2
```

```
[...]
```

```
TypeError: object does not support item assignment
```

- Tupel unterstützen sämtliche Sequenzoperationen<sup>3</sup>, die keine Modifikation implizieren.
- Auch Tupel können geschachtelt werden: (1, 2, (3, 4))
- Die fehlende Veränderbarkeit erlaubt eine effizientere Implementierung, ansonsten lassen sich Tupel auch durch Listen ersetzen.

---

<sup>3</sup>Vgl. Folie 21

# Datentypen: Tupel (2)

## Besonderheiten und Unterschiede zu Listen:

- Ein leeres Tupel wird durch „()“ dargestellt, eines mit einem einzelnen Element muss zur Unterscheidung von einem geklammerten Wert mit nachgestelltem Komma notiert werden: (0,)
- In Schleifenköpfen oder Zuweisungen, nicht aber Funktionsaufrufen, können die Klammern entfallen:

a, b = 1, 2

# Sequenzoperationen

Strings, Listen und Tupel unterstützen außer den erwähnten (Indizierung, Slicing, Operatoren, Zuweisung) eine Anzahl gemeinsamer Funktionen bzw. *Methoden* aufgrund ihrer Sequenzeigenschaften.

<code>X in S</code>	Enthalten
<code>X not in S</code>	Nicht enthalten
<code>len(S)</code>	Anzahl der Elemente
<code>iter(S)</code>	Iterator-Objekt
<code>min(S)</code>	Kleinstes Objekt
<code>max(S)</code>	Größtes Objekt
<code>del S[i:j:k]</code>	Elemente löschen

# Funktionen

- Das Schlüsselwort **def** erzeugt eine neue benannte Funktion (später mehr zu anonymen Funktionen):

```
def name(arg, ... arg=wert, ... *arg, **arg):
```

# Funktionen

- Das Schlüsselwort **def** erzeugt eine neue benannte Funktion (später mehr zu anonymen Funktionen):

```
def name(arg, ... arg=wert, ... *arg, **arg):
```

- Argumente können auf unterschiedliche Weise angegeben werden:

*name* einfaches Argument (untypisiert), Übereinstimmung nach Position oder Name

*arg=wert* Default-Wert, falls dieses Argument fehlt

*\*arg* Liefert in der Funktion ein Tupel mit allen Positionsargumenten.

*\*\*arg* Liefert in der Funktion ein Dictionary<sup>4</sup> mit allen Schlüsselwortargumenten.

- In der ersten Zeile des Funktionskörpers kann (und sollte) ein String folgen, der die Funktion beschreibt. Er wird bei der Ausführung ignoriert.

---

<sup>4</sup>Später...

# Funktionsargumente (1)

## „Klassische“ positionale Argumente

```
def fun(foo, bar) :  
    """Diese Funktion macht nichts tolles.  
    Sie zeigt nur die Benutzung positionaler Argumente."""  
    print foo, bar/2  
  
fun("test",5)  
fun("test",2,3,4)      # TypeError: fun() takes exactly 2 arguments  
                       # (4 given)  
fun("test","xyz")     # TypeError: unsupported operand type(s)  
                       # for /: 'str' and 'int'
```

## Funktionsargumente (2)

### Defaultwerte für Argumente

```
def defaults(foo, bar=10):  
    "Diese Funktion nimmt ein oder zwei Argumente"  
    print foo, bar/2  
  
fun("test")      # "test" 5  
fun("test",2)    # "test" 1
```

Wenn Argumente mit und ohne Defaults existieren, müssen erstere in der Argumentliste weiter hinten stehen!

## Funktionsargumente (2)

### Defaultwerte für Argumente

```
def defaults(foo, bar=10):
    "Diese Funktion nimmt ein oder zwei Argumente"
    print foo, bar/2

fun("test")      # "test" 5
fun("test",2)    # "test" 1
```

Wenn Argumente mit und ohne Defaults existieren, müssen erstere in der Argumentliste weiter hinten stehen!

### Auswertung des Defaultarguments

```
a = 5
def default(foo=a): # Default fuer foo ist 5!
    print foo
a = 0                # egal was hier mit a passiert
fun()
```

## Funktionsargumente (3)

Argumente können als eine Art Attribut-Werte-Paare in beliebiger Reihenfolge angegeben werden.

### Aufruf mit explizit benannten Argumenten

```
def anyfun(foo, bar, baz="42") :  
    "Gar_nichts_tun_mit_zwei_oder_drei_Argumenten"  
    pass      # No-op  
  
fun("hallo", "test")           # Positional  
fun(bar="test", foo="hallo")  # Aequivalent
```

Werden solche *Keyword*-Argumente zusammen mit positionalen verwendet, müssen letztere weiter hinten stehen.

# Funktionsargumente (4)

Funktionsargumente mit einem bzw. zwei vorangestellten Sternchen dürfen maximal je einmal auftreten. Sie erlauben es, „überzählige“ positionale bzw. benannte Argumente anzugeben:

## Extra-Argumente

```
def fun(arg, *morepos, **morenamed) :
    """BeliebigvieleArgumente!Geil,wa?"""
    print "arg=\t", arg
    print "morepos=\t", morepos
    print "morenamed=\t", morenamed

fun("test")           # OK
fun(arg="test")       # Aequivalent
fun(foo=1)            # Fehler: arg fehlt
fun("test",foo=1)     # OK, **morenamed={"foo":1}
fun("test",2,3,4)     # OK, *morepos=(2,3,4)
fun("test",2,bar=10,3,4) # Fehler keyword b/f positional
```

## Übung<sup>6</sup>: Fibonacci-Zahlen

- Schreiben Sie eine iterative und eine rekursive Funktion, die jeweils die Fibonacci-Zahlen<sup>5</sup> bis zu einem als Parameter angebbaren Maximum ausgeben!
- Was wäre eine naive, was eine etwas intelligentere rekursive Umsetzung?
- Tip: Tupel helfen bei der iterativen Variante

---

<sup>5</sup>Reihe, für die gilt:  $i_0 = 0, i_1 = 1, i_n = i_{n-2} + i_{n-1}$

<sup>6</sup>OK, computerlinguistische Aufgaben sehen anders aus. Wenn erstmal die Regular Expressions dran waren...

## Übung<sup>6</sup>: Fibonacci-Zahlen

- Schreiben Sie eine iterative und eine rekursive Funktion, die jeweils die Fibonacci-Zahlen<sup>5</sup> bis zu einem als Parameter angebbaren Maximum ausgeben!
- Was wäre eine naive, was eine etwas intelligentere rekursive Umsetzung?
- Tip: Tupel helfen bei der iterativen Variante

---

<sup>5</sup>Reihe, für die gilt:  $i_0 = 0, i_1 = 1, i_n = i_{n-2} + i_{n-1}$

<sup>6</sup>OK, computerlinguistische Aufgaben sehen anders aus. Wenn erstmal die Regular Expressions dran waren...

# Datentypen: Dictionaries

- Dictionaries ordnen unveränderlichen Schlüsseln beliebige Werte zu (entsprechend Hashes in Perl)

- Syntax:

```
dict = 'key': "value", 123: 456, (1,"foo"): []
```

# Datentypen: Dictionaries

- Dictionaries ordnen unveränderlichen Schlüsseln beliebige Werte zu (entsprechend Hashes in Perl)
- Syntax:  

```
dict = 'key': "value", 123: 456, (1, "foo"): []
```
- Listen, Dictionaries und andere veränderliche Typen können nicht als Schlüssel dienen; Tupel nur, wenn sie keine veränderlichen Typen enthalten.

# Dictionary-Methoden/Operatoren

<code>a[k] = v</code>	Wert <code>v</code> unter Schlüssel <code>k</code> speichern
<code>a.keys()</code>	Liste der Schlüssel
<code>a.items()</code>	Liste der Werte
<code>del a[k]</code>	Schlüssel und Wert löschen
<code>a.has_key(k)</code>	Schlüssel enthalten <sup>7</sup>
<code>a.get(k[, x])</code>	<code>a[k]</code> if <code>k in a</code> , else <code>x</code>
<code>a.setdefault(k[, x])</code>	<code>a[k]</code> if <code>k in a</code> , else <code>x</code> (und setzt <code>a[k]</code> )
<code>a.pop(k[, x])</code>	<code>a[k]</code> if <code>k in a</code> , else <code>x</code> (und löscht <code>a[k]</code> )
<code>a.popitem()</code>	Bel. A/V-Paar zurückgeben und löschen
<code>a.clear()</code>	Alle Zuordnungen löschen
<code>a.iteritems()</code>	Iterator über alle A/V-Paare

<sup>7</sup>`k in a` bzw. `k not in a` werden unterstützt, aber die Methodenschreibweise wird empfohlen.

# Datentypen: Sets

- Sets modellieren mathematische Mengen und werden aus Sequenzen erzeugt.

Syntax: `s = set([1, 2, 3, 2, 2, 3])`    *# set([1, 2, 3])*

- Unterschiede zur Liste:
  - Sets sind ungeordnet und keine Sequenzen
  - Jedes Element kommt garantiert nur einmal vor

# Datentypen: Sets

- Sets modellieren mathematische Mengen und werden aus Sequenzen erzeugt.

Syntax: `s = set([1, 2, 3, 2, 2, 3])`    `# set([1, 2, 3])`

- Unterschiede zur Liste:
  - Sets sind ungeordnet und keine Sequenzen
  - Jedes Element kommt garantiert nur einmal vor
- Set-Operatoren:
  - Differenzmenge:  $a - b$
  - Vereinigungsmenge:  $a | b$
  - Schnittmenge:  $a \& b$
  - Symmetrische Differenz:  $a \hat{=} b$

# Module

- Das Schlüsselwort `import` bindet ein Modul in ein Python-Programm ein.
- Es können entweder ganze Module oder einzelne Bezeichner daraus importiert werden.

# Module

- Das Schlüsselwort `import` bindet ein Modul in ein Python-Programm ein.
- Es können entweder ganze Module oder einzelne Bezeichner daraus importiert werden.

## import

```
import sys           # Ganzes Modul "sys" importieren
from sys import stderr # stderr in den lokalen Namensraum importieren
from sys import *     # Alles aus sys in den lokalen Namensraum imp.
import sys as foo    # Modul sys mit Namensraum foo importieren
from sys import stderr as mecker # stderr als "mecker" importieren
```

# Module

- Das Schlüsselwort `import` bindet ein Modul in ein Python-Programm ein.
- Es können entweder ganze Module oder einzelne Bezeichner daraus importiert werden.

## import

```
import sys           # Ganzes Modul "sys" importieren
from sys import stderr # stderr in den lokalen Namensraum importieren
from sys import *     # Alles aus sys in den lokalen Namensraum imp.
import sys as foo    # Modul sys mit Namensraum foo importieren
from sys import stderr as mecker # stderr als "mecker" importieren
```

- Hilfreiche Funktionen zum Umgang mit Modulen:
  - `dir(Module)`: Übersicht über die definierten Attribute
  - `help(Attr)`: Inline-Doku zum spezifizierten Attribut anzeigen

# Dateien

Dateien sind in Python echte Objekte, nicht wie in Perl etwas „spezielle“ Typen. Grundlegendes:

- Dateien öffnen: `f = open(Name, [mode], [buffering])`
  - Schließen: `f.close()`
- Mit `dir()` und `help()` nachschauen, welche Methoden noch unterstützt werden.
- Wozu taugt die `__iter__`-Methode?

# Übung: Frequenzliste

Das beliebte Beispiel der Korpuslinguisten!

- Schreiben Sie ein Programm, das aus einer Wortliste wie `/korpora/Limas/limas.pp` eine Frequenzliste erzeugt.
- Zeilen lesen geht mit `readline()` oder einfach mit einem Iterator.
- Wer „chomp“ vermisst, will `rstrip()` haben.
- Dictionaries klingen nicht nur so linguistisch, sie sind hier auch dringend notwendig.
- Sortiert muss die Ausgabe *noch nicht* sein.

# Praktisches zum Umgang mit Zeichensätzen

## Zeichensatz-Codecs

```
import codecs
import sys
# newin= codecs.EncodedFile(file, inputenc [, outputenc [, errors ]])
newin= codecs.EncodedFile(sys.stdin, "utf8")
# codecs.open(filename, mode[, encoding[, errors [, buffering ]]])
corpus = codecs.open("/korpora/Limas/limas.txt", "r", "latin1")
s = corpus.readline ()
# u'Als es zu schneien aufgeh\xxf6rt hatte , verlie\xxdf Johanna von\n'
print s
# Als es zu schneien aufgeh\u00f6rt hatte , verlie\u00df Johanna von
```

- Codecs wandeln Kodierungen transparent beim lesen und schreiben
- Ausgabe sind meist Unicode-Strings

# Klassen (1)

## Klassen

```
class Example:
    def __init__ ( self ,foo):
        self . attr = foo
    def myMethod(self):
        print self . attr
xmp = Example("hallo")
xmp.myMethod()
```

- Klassen werden mit dem Schlüsselwort **class** eingeleitet und enthalten normale Funktionsdefinitionen als Methoden.

# Klassen (1)

## Klassen

```
class Example:
    def __init__( self ,foo):
        self . attr = foo
    def myMethod(self):
        print self . attr
xmp = Example("hallo")
xmp.myMethod()
```

- Klassen werden mit dem Schlüsselwort **class** eingeleitet und enthalten normale Funktionsdefinitionen als Methoden.
- Jede Methode bekommt das aktuelle Objekt („this“ in C++/Java) als impliziten ersten Parameter, der konventionell `self` genannt wird.

## Klassen (3): Spezialmethoden

Einige Methodennamen sind vordefiniert und werden in speziellen Kontexten aufgerufen. Alle beginnen und enden mit doppeltem Unterstrich. Die wichtigsten:

`__new__(class[,...])` Eigentlicher Konstruktor, aber selten gebraucht.

`__init__(self[,...])` Objektinitialisierung; entspricht eher dem Konstruktor in Java.

`__del__(self)` Destruktor; wird aufgerufen, wenn der *Garbage Collector* das Objekt löscht.

`__repr__(self)` Erzeugt die „offizielle“ String-Repräsentation des Objekts, möglichst als Python-Ausdruck.

`__str__(self)` Erzeugt die informelle String-Repräsentation des Objekt.

## Klassen (4): Container-Klassen

Für Container-Klassen<sup>8</sup> gibt es wiederum einige Spezialmethoden, die automatisch aufgerufen werden, wenn entsprechende Operatoren auf ein Objekt angewendet werden:

`__len__(self)` Gibt die Länge des Objekts zurück.

`__getitem__(self, key)` Behandelt lesende Array- bzw. Dictionary-Indizierung.,

`__setitem__(self, key, value)` Dito, schreibend

`__delitem__(self, key)` Löscht ein Element aus dem Container.

`__iter__(self)` Gibt einen Iterator über alle Elemente im Container zurück.

---

<sup>8</sup>Klassen, die Sammlungen anderen Objekte darstellen. 

# Generatoren (1)

- Einfach zu benutzende Umsetzung eines Konzepts aus der funktionalen Programmierung
- Erlauben viele interessante Tricks wie unendliche Sequenzen
- Vereinfachen oft erheblich das Programmieren von Parsern

# Generatoren (1)

- Einfach zu benutzende Umsetzung eines Konzepts aus der funktionalen Programmierung
- Erlauben viele interessante Tricks wie unendliche Sequenzen
- Vereinfachen oft erheblich das Programmieren von Parsern

## Generator

```
def generator():  
    i = 0  
    while True:  
        i += 2  
        yield i  
  
for j in generator(): print j
```

## Generatoren (2)

- Eine Funktion, die `yield` benutzt, gibt automatisch einen Iterator zurück.
- `yield` funktioniert wie **`return`**, speichert aber den gesamten Zustand der Funktion
- Beim Aufruf des Iterators wird die Funktion jeweils hinter dem `yield` fortgesetzt.
- Besonders praktisch zum programmieren von `__iter__`-Methoden in Container-Objekten

# Abgeleitete Klassen

- Um eine Unterklasse zu einer existierenden zu deklarieren, wird ihr Name in Klammern hinter dem Klassennamen angegeben:  
`class MyException( Exception ):`
- Mehrfachvererbung ist ebenso möglich, wenn auch nicht besonders empfehlenswert:  
`class Unterklasse( Klasse1 , Klasse2 , Klasse3 ):`

# Abgeleitete Klassen

- Um eine Unterklasse zu einer existierenden zu deklarieren, wird ihr Name in Klammern hinter dem Klassennamen angegeben:  
`class MyException( Exception ):`
- Mehrfachvererbung ist ebenso möglich, wenn auch nicht besonders empfehlenswert:  
`class Unterklasse( Klasse1 , Klasse2 , Klasse3 ):`
- Beim Aufruf werden Methoden und Attribute zuerst im Objekt der abgeleiteten Klasse gesucht; schlägt dies fehl, wird in der nächsthöheren Klasse weiter gesucht.

# Abgeleitete Klassen

- Um eine Unterklasse zu einer existierenden zu deklarieren, wird ihr Name in Klammern hinter dem Klassennamen angegeben:  
`class MyException( Exception ):`
- Mehrfachvererbung ist ebenso möglich, wenn auch nicht besonders empfehlenswert:  
`class Unterklasse( Klasse1 , Klasse2 , Klasse3 ):`
- Beim Aufruf werden Methoden und Attribute zuerst im Objekt der abgeleiteten Klasse gesucht; schlägt dies fehl, wird in der nächsthöheren Klasse weiter gesucht.
- Methoden sind, in C++-Terminologie, immer „virtual“, d.h. eine Unterklasse kann sie überschreiben und auch ein Aufruf *aus der Oberklasse heraus* ruft den Code der Unterklasse auf.
- Um eine Methode der Oberklasse explizit aufzurufen (z.B. um sie nicht einfach zu überschreiben, sondern zu erweitern), kann sie wie eine „static“-Methode in Java aufgerufen werden:  
`Oberklasse.methode( self , args )`

# Exceptions: Erzeugen

- Wie in Java werden Exceptions zur Fehlerbehandlung benutzt.
- Exceptions sind Klassen und können auch erweitert werden.
- Erzeugen einer Exception:  
`raise` Exceptionklasse [args] oder  
`raise` Exceptioninstanz
- „args“ ist i.d.R. ein skalarer Datentyp oder ein Tupel und wird dem Konstruktor der Exceptionklasse übergeben.

# Exceptions: Erzeugen

- Wie in Java werden Exceptions zur Fehlerbehandlung benutzt.
- Exceptions sind Klassen und können auch erweitert werden.
- Erzeugen einer Exception:  
`raise` Exceptionklasse [args] oder  
`raise` Exceptioninstanz
- „args“ ist i.d.R. ein skalarer Datentyp oder ein Tupel und wird dem Konstruktor der Exceptionklasse übergeben.
- Die Varianten des `raise`-Statements sind vielfältig und kompliziert, die häufigsten Fälle sind aber mit den obigen abgedeckt. Details s. Reference Manual

# Exceptions: Abfangen

- Das Abfangen von Exceptions wird häufiger gebraucht, da sie auch von eingebauten Funktionen und der Standardbibliothek erzeugt werden.
- Allgemeines Schema: **try...except**-Block, optional mit **else** (wird ausgeführt, falls *keine* Exception auftrat) und **finally** (wird *auf jeden Fall* ausgeführt)

## Zwei Exception-Typen

```
try:
    file = open(filename)
    s = f.readline()
    i = int(s.strip())
except IOError, (errno, strerror):
    print "I/O error(%s): %s" % (errno, strerror)
except ValueError:
    print "Could not convert data to an integer."
```

# Funktionale Programmierung: lambda

- Lambda-Ausdrücke sind eine Methode, einfache anonyme Funktionen darzustellen.
- Sie können *nur einen einzelnen Ausdruck* enthalten, sind insofern also weniger mächtig als gewöhnliche Funktionsdefinitionen, allerdings auch kürzer hinschreiben

## Lambda

```
f = lambda a,b: a*b      # gibt das Produkt der beiden Argumente zurück
```

```
# Äquivalent:
```

```
f = multargs
```

```
def multargs(a,b): return a*b
```

- Lambdas sind nützlich z.B. für „callbacks“ in Sortier- und Filterfunktionen, für die man nicht jedesmal eine öffentliche, benannte Funktion definieren will.

# Funktionale Programmierung: `filter()` und `reduce()`

- `filter(f, sequence)` liefert eine Sequenz derjenigen Elemente aus `sequence`, für die `f(element)` True ist<sup>9</sup>.

## Großbuchstabenfilter

```
print filter(lambda c: c.isupper(), "Nobody expects the Spanish Inquisition!")
```

- `reduce(f, sequence)` reduziert eine Sequenz auf einen einzelnen Wert, indem `f(a, b)` zuerst mit den ersten zwei Werten aufgerufen wird, dann mit dem Ergebnis von `f` und dem zweiten Wert, usw.

## Reduce

```
reduce(lambda x,y: x+y, range(1, 11))
```

<sup>9</sup>Perl: `grep()`.

# Funktionale Programmierung: `map()`

- `map(f, sequence [, ...])` ruft *f* mit jedem Element der *sequence* auf und gibt eine Sequenz der Rückgabewerte zurück<sup>10</sup>.
- Wird mehr als eine Sequence angegeben, muss *f* entsprechend viele Parameter haben und wird dann mit den einander entsprechenden Elementen jeder Sequence aufgerufen

## SwapCase

```
print map(lambda c: c.swapcase(),"Kreuzigung_ist_doch_Firlefanzt.")
```

---

<sup>10</sup>Perl: ebenso.

# Funktionale Programmierung: `map()`

- `map(f, sequence [, ...])` ruft `f` mit jedem Element der `sequence` auf und gibt eine Sequenz der Rückgabewerte zurück<sup>10</sup>.
- Wird mehr als eine Sequence angegeben, muss `f` entsprechend viele Parameter haben und wird dann mit den einander entsprechenden Elementen jeder Sequence aufgerufen

## SwapCase

```
print map(lambda c: c.swapcase(),"Kreuzigung_ist_doch_Firlefanzt.")
```

- Obiges nicht benutzen, geht mit `string.swapcase()` schneller und einfacher!

---

<sup>10</sup>Perl: ebenso.

# Reguläre Ausdrücke

Reguläre Ausdrücke sind nicht wie in Perl fester Sprachbestandteil, sondern werden von dem Modul `re`<sup>11</sup> bereitgestellt.

- Allgemeine Vorgehensweise
  - Regex-Objekt mit `re.compile(pattern [, flags])` erzeugen
  - Abpassen/ersetzen durch die Methoden `match(string[, pos[, endpos]])`, `search`, `split`, `sub`, `finditer` etc. des Regex-Objekts
- `search()` entspricht der Perl-Funktionalität, `match()` dem Default in Java (`search("^foo$") ~ match("foo")`)
- `sub(replacement, string[, count])` ersetzt Muster.
- Viele Methoden des Regex-Objekts haben quasi äquivalente Funktionen in `re`, die als zusätzlichen Parameter den regulären Ausdruck nehmen und das vorkompilieren sparen (vorsicht: sind manchmal weniger mächtig!).

---

<sup>11</sup>Die eigentliche Implementation (und Doku!) steckt derzeit in „`re`“.

# Übung

- Schreiben Sie eine Wrapper-Klasse für `file`, die den Inhalt einer Textdatei *wortweise* über einen Iterator bereitstellt.
  - „Zu Fuß“ mittels `__init__()`, `__iterator__()` und `next()`, oder:
  - Per Generator mit nur `__init__()` und `__iterator__()`
  - Tip: die String-Methode `split()` wird gebraucht.
- Schreiben Sie mit Hilfe dieser Klasse eine weitere, die KWIC-Listen generiert. Dateiname, Suchbegriff (optional als regulärer Ausdruck) und Kontext (Wortformen vor und hinter dem Suchbegriff) sollen dem Konstruktor übergeben werden können.
  - Tip: Array als Queue benutzen mit `append()` und `pop(0)`
  - Arrays kopieren mit Slice-Notation: `array[:]`

# Übung

- Erweitern Sie das KWIC-Programm aus der vorigen Übung um eine eigene Exception-Klasse, die den Parameter der `__init__`-Methode im Attribut „value“ speichert.
- Probieren Sie auf [clue45](#) das Modul `chardet` aus. Lassen Sie den Konstruktor der KWIC-Klasse den Zeichensatz der Eingabedatei bestimmen (Textmenge begrenzen!).
- Bei einer Sicherheit von  $<70\%$  soll die eigene Exception geworfen werden.
- Fangen Sie die eigene Exception sowie `IOError` im Hauptprogramm ab.

# Übung

- Das Limas-Korpus in `/korpora/Limas/limas.t` ist mit dem *Standard Tübingen Tag Set* annotiert. Erweitern Sie das Frequenzlistenprogramm so, dass die Liste auf ein bestimmtes Tag bzw. ein Tag-Muster<sup>12</sup> eingeschränkt werden kann.
- Erweitern Sie auch das KWIC-Programm so, dass z. B. zwischen der Funktion von „das“ als Artikel und als Personalpronomen unterschieden werden kann.

---

<sup>12</sup>Es kann sinnvoll sein, nur einen Präfix des Tags, z.B. „N“ zu spezifizieren.

# XML-Verarbeitung

Zwei konkurrierende Modelle: DOM und SAX

**DOM** : *Document Object Model*; das gesamte Dokument wird an einem Stück geparkt und als Baum von Objekten im Speicher gehalten.

- Vorteil: wahlfreier Zugriff auf jedes beliebige Element
- Nachteil: evtl. extremer Speicherverbrauch;  
Verarbeitung kann erst beginnen, wenn das komplette Dokument eingelesen ist.

**SAX** : *Simple API for XML*; der Parser geht das Dokument elementweise durch und ruft dabei ggf. benutzerdefinierte Methoden auf, die die Verarbeitung erledigen.

- Vorteil: wesentlich geringerer Speicherverbrauch;  
Verarbeitung beginnt mit dem Lesen des ersten Elements.
- Nachteil: kompliziertere Verarbeitungslogik, benötigt fehlerträchtige Automaten.

# XML-Verarbeitung: Python-Module

## DOM

```
import sys
from xml.dom.ext.reader import Sax2
reader = Sax2.Reader() # Reader-Objekt erzeugen
doc = reader.fromStream(sys.stdin) # Parser anwerfen
doc = reader.fromUri("file://mydoc.xml") # Daten aus URI
```

## SAX

```
import xml.sax
from xml.sax import saxutils
class MyHandler(saxutils.DefaultHandler):
    def characters(self, ch): [...]
    def startElement(self, name, attrs): [...]

parser = xml.sax.make_parser()
parser.setContentHandler(MyHandler())
parser.parse("mydoc.xml")
```

# XML-Verarbeitung: DOM

## DOM

```
import sys
from xml.dom.ext.reader import Sax2
reader = Sax2.Reader() # Reader-Objekt erzeugen
doc = reader.fromStream(sys.stdin) # Parser anwerfen
doc = reader.fromUri("file://mydoc.xml") # Daten aus URI
```

# XML-Verarbeitung: SAX

- Von `saxutils.DefaultHandler` einen eigenen Handler ableiten und die vordefinierten NOP-Methoden überschreiben:
  - `characters` Für jegliche Daten zwischen Tags ("" CDATA").
  - `startElement` Öffnendes Element gefunden; Parameter: Tagname und Attribute als Dictionary.
  - `endElement` Für schließende Elemente, bekommt nur den Tagnamen.
  - `error` Bekommt Fehlerbeschreibung als Unterklasse von `SAXException`
- `xml.sax.make_parser()` erzeugt ein neues Parser-Objekt.
- Eigenes Handler-Objekt mit `setContentHandler()`-Methode setzen
- Zwecks-Effizienz: Namespaces abschalten, falls nicht benötigt (`setFeature(xml.sax.handler.feature_namespaces, 0)`)
- `parse()` nimmt einen URI und parst das darunter abgelegte Dokument.

# Modularisierung eigener Programme (1)

- Mit **import** können nicht nur Module der Standardbibliothek importiert werden, sondern auch eigene.
- Das aktuelle Verzeichnis befindet sich standardmäßig im Modul-Suchpfad, so dass dort abgelegte Python-Dateien direkt importiert werden können.
- Außerhalb von Klassen oder Funktionen notierter Python-Code in einem Modul wird beim Importieren direkt ausgeführt.
- Definitionen aus einem Modul bleiben normalerweise in ihrem eigenen Namensraum!

## Modularisierung eigener Programme (2)

### meinmodul.py

```
print "Importiere meinmodul"  
def hello ():  
    print "Hello, world!"
```

### main.py

```
import meinmodul  
mainmodul.hello()
```

# Kommandozeilenparsing mit *getopt*

Statt `sys.argv` per hand zu parsen, können Optionen bequemer mit dem Modul *getopt* verarbeitet werden.

- Die Funktion `getopt(args, options [, long_options])` und ihr Gegenstück `gnu_getopt` mit der selben Parameterliste verarbeiten ganze Kommandozeilen.
- Als *options* wird ein String aus Zeichen angegeben, die als Optionen erkannt werden sollen. Benötigt eine Option ein Argument, muss dem Zeichen ein Doppelpunkt folgen.
- Für *long\_options* muss eine Liste von Strings angegeben werden, die als Optionen erkannt werden sollen, optional mit „=" am Ende, falls ein Argument verlangt wird.
- Der Rückgabewert ist ein Tupel aus
  - 1 Einer Liste von Tupeln aus Option und Leerstring bzw. Argument, falls eins angegeben wurde.
  - 2 Einer Liste der übrigen Argumente

# Übung

- Unter `~msbethke/Courses/Python/Code/tiger.xml` finden Sie ein Fragment des Tiger-Korpus.
- Versuchen Sie, die Datei mittels DOM-Parser zu parsen; schmeißen Sie den Code im Angesicht der Laufzeit angewidert weg und benutzen Sie SAX.
- Schreiben Sie eine Handler-Klasse, die den Text des Korpus extrahiert.
- Wie könnte man einen allgemeinen Handler schreiben, der nur in bestimmten (konfigurierbaren) Tags enthaltenen Text berücksichtigt?

# Übung

Die Aufgabe aus der EMSV-Übung, heute in Python. . .  
Erstellen Sie ein Skript,

- welches die Sätze des Limaskorpus zeilenweise ausgibt.
- welches zu jeder Wortform des Limaskorpus notiert
  - in welchen Sätzen (Satznummer)
  - an wievielter Stelle in diesen Sätzen (Wortnummer)diese Wortform auftritt.
- Geben Sie die Ergebnisse einmal lesbar und einmal als SQL-Kommandos zum Einfügen des Index in ein passendes Datenbankschema aus.