HS 2014: Programmiertechniken in der Computerlinguistik I (2. Teil)

Simon Clematide
simon.clematide@uzh.ch

Hinweis: Dieses Lauftextskript wurde automatisch aus den Vorlesungsfolien generiert und ist deshalb bezüglich Layout und Formulierungen nicht für Fliesstext optimiert.

Version von 11. Dezember 2014
PDF-Skript: http://files.ifi.uzh.ch/cl/siclemat/lehre/hs14/pcl1/script/script.pdf
OLAT-Seite: https://www.olat.uzh.ch/olat/url/RepositoryEntry/8746991619

Universität Zürich
Institut für Computerlinguistik
Binzmühlestr. 14
8050 Zürich

Inhaltsverzeichnis

1 Vorspann **4**
1.1 Objekte . . . 5
1.1.1 Typen . . . 5
1.1.2 Methoden und Attribute . . . 6
1.2 Zeichen . . . 7
1.2.1 Dateikodierung . . . 7
1.2.2 Zeichenliterale . . . 9
1.2.3 Umkodieren . . . 11
1.3 Regex . . . 11
1.3.1 Ersetzen . . . 12
1.3.2 Suchen . . . 12

2 NLTK-Buch Kapitel 1 **15**
2.1 NLTK . . . 15
2.1.1 Intro . . . 15
2.1.2 Module und Packages . . . 17
2.1.3 Korpuslinguistische Demo . . . 19
2.2 Technisches . . . 19
2.2.1 Listenkomprehension . . . 19
2.2.2 Funktionen . . . 20
2.2.3 Namensräume . . . 22
2.3 Vertiefung . . . 23

3 NLTK-Buch Kapitel 2 **24**
3.1 Korpora . . . 24
3.1.1 Korpora einlesen . . . 25
3.1.2 Korpus-Typen . . . 26
3.2 Häufigkeitsverteilungen . . . 26
3.2.1 Univariat . . . 26
3.2.2 Bivariat . . . 28
3.3 Technisches . . . 29
3.3.1 Sequenzen . . . 29
3.3.2 Klassen & Objekte . . . 31
3.3.3 Statements & Expressions . . . 32
3.4 Vertiefung . . . 34

4 NLTK-Buch Kapitel 2: Lexikalische Ressourcen **35**
4.1 Lexika . . . 35
4.1.1 Wortlisten . . . 35
4.1.2 Aussprachelexika . . . 36
4.1.3 Bedeutungslexika . . . 37
4.2 Technisches . . . 38
4.2.1 Klassendefinition . . . 38
4.3 Vertiefung . . . 41

5 NLTK-Buch Kapitel 3 **42**
5.1 Konkordanzen . . . 42
5.1.1 Formatierungsausdrücke . . . 43
5.1.2 Stemmer . . . 44
5.1.3 Textindexklasse . . . 44
5.2 Technisches . . . 45
5.2.1 Generatorausdrücke . . . 45
5.2.2 xrange . . . 47
5.3 Ausnahmen . . . 47
5.4 Vertiefung . . . 50

6 NLTK-Buch Kapitel 3: Korpora **51**
6.1 NLP-Pipeline . . . 51
6.1.1 urllib . . . 51
6.1.2 HTML-Pipeline . . . 52

A Liste der verlinkten Beispielprogramme und Ressourcen **56**

Abbildungsverzeichnis

1.1 Zeichenkodetabelle von ISO-Latin-1 7

4.1 37

Kapitel 1

Vorspann

Lernziele

- Was sind Objekte? Was sind Typen?
- Was sind Funktionen? Was sind Methoden?
- Welche Rolle spielt die Kodierung des Quellkodes und der zu verarbeitenden Textdateien?
- Wie kann man Zeichenketten notieren (Literale)? Wie kann man Bytefolgen dekodieren? Wie kann man Unicode in UTF-8 enkodieren?
- Anwendung von regulären Ausdrücken in Python zum Suchen und Ersetzen in Zeichenketten mit Nicht-ASCII-Zeichen
- Verstehen der Funktionsweise eines mächtigen regex-basierten Tokenizers

Vorvorspann: Meine Binsenwahrheiten

Warum ist konzeptuelle Klarheit empfehlenswert?

- Solange beim Programmieren alles läuft, wie es soll, ist oberflächliches *Howto*-Wissen genügend.
- Sobald Schwierigkeiten auftauchen, muss man verstehen, was man eigentlich macht.
- Nur so hat man eine Chance, Fehler zu finden.
- "Traurige" Wahrheit: Debugging (Fehler lokalisieren und eliminieren) ist einiges schwieriger als Programmieren!

Dos and Don'ts der Datei- und Verzeichnisbenennung

- Benenne Python-Dateien nur mit Kleinbuchstaben, Unterstrich und Ziffern!
- Benutze NIE Umlaute! Benutze NIE Umlaute! Benutze nie Leerzeichen!

Python Dokumentation: Erläutertes Beispiel

Python-interne Hilfsfunktion → *1*

```
>>> help(re.split)
Help on function split in module re:

split(pattern, string, maxsplit=0, flags=0)
    Split the source string by the occurrences of the pattern,
    returning a list containing the resulting substrings.
```

Erklärung der Erklärung

- Mehr Info auf der Online-Dokumentation http://docs.python.org/2/.
- *Name=Wert* zeigt Standardwert optionaler Argumente.
- Ergebnis- und Argumenttypen werden im Fliesstext informell erklärt.

1.1 Objekte

1.1.1 Typen

Grundlegendes zu Objekten: Typen

Python ist eine *objektorientierte* Programmiersprache. ▶▶▶
Alle Daten (Datenstrukturen) in Python sind Objekte.

Python ist eine *dynamisch getypte* Programmiersprache. ▶▶▶
Alle Objekte haben einen *Typ*.

- Warum "dynamisch"? Der Typ einer Variablen muss nicht statisch im Quelltext bei ihrer Einführung angegeben werden, er wird dynamisch zur Laufzeit bestimmt.

Alle Objekte haben einen Typ

Eingebaute Funktion `type()`
Sie bestimmt den Typ von jedem Objekt.

Typen bestimmen

```
>>> type(1)
>>> type(1+3*42)
>>> type("ABBA")
>>> type("AB"+'BA')
>>> type(True)
>>> type(['a','b'])
>>> type(['a','b'][0])
>>> type({})
>>> type(re.search('X','aaa'))
>>> type(None)
>>> type(re.search('a','aaa'))
```

Alle Objekte haben eine kanonische String-Repräsentation

Die eingebaute Funktion `repr()`
Erzeugt eine kanonische Zeichenkette aus jedem Objekt.

```
>>> repr("a")
"'a'"
>>> repr(['a',"b"])
"['a', 'b']"
```

Read-Eval-Print-Loop (REPL) → **2**
Im interaktiven Gebrauch wird der zuletzt eingegebene Ausdruck (*expression*) evaluiert und das Resultat als kanonischer String mit `print` ausgegeben.

```
>>> ['a',"b"]
"['a', 'b']"
>>> print repr(['a',"b"])
"['a', 'b']"
```

1.1.2 Methoden und Attribute

Grundlegendes zu Objekten: Methoden und Attribute

Objekte haben *Methoden*.
Methoden sind *Funktionen*, die von einem Objekt aus aufgerufen werden.

- Methoden-Aufrufe (*invocation*) evaluieren zu einem Objekt: `"ABBA".count("B")`
- `"ABBA".count` ist eine Methode des Objekts `"ABBA"`
- Methoden ohne Argumente aufrufen: `"ABBA".lower()`

```
>>> help(str.split)
Help on method_descriptor:
split(...)
    S.split([sep [,maxsplit]]) -> list of strings

    Return a list of the words in the string S, using sep as the
    delimiter string. If maxsplit is given, at most maxsplit
    splits are done. If sep is not specified or is None, any
    whitespace string is a separator and empty strings are removed
```

Grundlegendes zu Objekten: Attribute

Objekte haben benannte *Attribute*.
Attribute sind *Objekte*.

Dokumentationsstrings von Funktionen als Attribut `__doc__`

```
>>> len.__doc__
'len(object) -> integer\n\nReturn the number of items of a sequence or collection.'
>>> help(len)
Help on built-in function len in module __builtin__:
len(...)
    len(object) -> integer
    Return the number of items of a sequence or collection.
```

Methoden = aufrufbare Attribute
Funktionen sind ebenfalls Objekte. Eine Methode ist ein aufrufbares Attribut eines Objekts.

1.2 Zeichen

1.2.1 Dateikodierung

Zeichenkodierungen und Zeichensätze

Einschränkungen von ASCII
Die weitverbreiteste Zeichenkodierung mit 128 Kodes (7-Bit) unterstützt keine nicht-englischen Buchstaben.

Verschiedene Erweiterungen mit 8-Bit Kodierungen
Zeichenkodierungen mit 256 Kodes (1 Byte) für verschiedene Alphabete oder Betriebssysteme: ISO-8859-1, ISO-8859-9, Mac-Roman, Windows-1252

Universale Lösung Unicode
Unicode weist jedem Zeichen einen eindeutigen Zahlen-Kode und eine Kategorie zu. Unicode 6.0 definiert 109'449 graphische Zeichen

Kode-Tabellen von ASCII, ISO-8859-1 und partiell Unicode

Code	...0	...1	...2	...3	...4	...5	...6	...7	...8	...9	...A	...B	...C	...D	...E	...F
0...	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
1...	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
2...	SP	!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
3...	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4...	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5...	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6...	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7...	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	DEL
8...	PAD	HOP	BPH	NBH	IND	NEL	SSA	ESA	HTS	HTJ	VTS	PLD	PLU	RI	SS2	SS3
9...	DCS	PU1	PU2	STS	CCH	MW	SPA	EPA	SOS	SGCI	SCI	CSI	ST	OSC	PM	APC
A...	NBSP	¡	¢	£	¤	¥	¦	§	¨	©	ª	«	¬	SHY	®	¯
B...	°	±	²	³	´	µ	¶	·	¸	¹	º	»	¼	½	¾	¿
C...	À	Á	Â	Ã	Ä	Å	Æ	Ç	È	É	Ê	Ë	Ì	Í	Î	Ï
D...	Ð	Ñ	Ò	Ó	Ô	Õ	Ö	×	Ø	Ù	Ú	Û	Ü	Ý	Þ	ß
E...	à	á	â	ã	ä	å	æ	ç	è	é	ê	ë	ì	í	î	ï
F...	ð	ñ	ò	ó	ô	õ	ö	÷	ø	ù	ú	û	ü	ý	þ	ÿ

Quelle: Nach http://de.wikipedia.org/wiki/ISO_8859-1

Abbildung 1.1: Zeichenkodetabelle von ISO-Latin-1

Speicher- und Transportformat UTF-8

Persistente Speicherung und Datenübertragung mit Unicode

UTF (Abk. für *Unicode Transformation Format*) beschreibt Methoden, einen Unicode-Wert auf eine Folge von Bytes abzubilden.

Beispiele für UTF-8-Kodierungen

Zeichen	Unicode	Unicode binär	UTF-8 binär	UTF-8 hexadezimal
Buchstabe *y*	U+0079	00000000 01111001	01111001	0x79
Buchstabe *ä*	U+00E4	00000000 11100100	11000011 10100100	0xC3 0xA4
Eingetragene Marke ®	U+00AE	00000000 10101110	11000010 10101110	0xC2 0xAE
Eurozeichen €	U+20AC	00100000 10101100	11100010 10000010 10101100	0xE2 0x82 0xAC

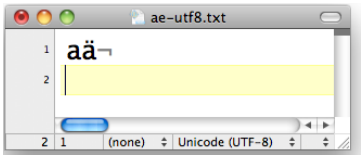
Quelle: http://de.wikipedia.org/wiki/UTF-8

Gründe für Format mit variabler Länge: Kompatibilität mit ASCII, kompakte Repräsentation, Sortierbarkeit, Erkennbarkeit von Zeichenanfängen

Textdatei als Bytefolge

Die Repräsentation der Zeichen mit Kodes > 127 sind unterschiedlich.

Datei in UTF-8-Kodierung

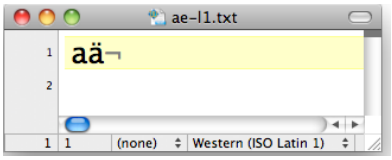


ä = 2 Bytes = C3 A4

```
$ hexdump  ae-utf8.txt
0000000 61 c3 a4 0a
0000004
```

4 Bytes

Datei in Latin-1-Kodierung



ä = 1 Byte = E4

```
$ hexdump  ae-l1.txt
0000000 61 e4 0a
0000003
```

3 Bytes

1.2.2 Zeichenliterale

Zeichen und ihre Zahlenkodes

Zeichenkode berechnen aus Zeichen

```
>>> ord("A")
65
>>> ord('a')
97
>>> ord('\n')
10
>>> ord("\t")
9
>>> ord('\x20') # Hexadezimal
32
>>> ord("\'")
39
```

Zeichen berechnen aus Zeichenkode

```
>>> chr(65)
'A'
>>> chr(97)
'a'
>>> chr(10)
'\n'
>>> chr(9)
'\t'
>>> chr(32)
' '
>>> chr(39)
"'"
```

Datentyp `str`: Folgen von Zeichen als Bytefolgen

Datentyp bestimmen und testen

```
>>> type('ABBA')
<type 'str'>

>>> type("ABBA") == str
True

>>> isinstance('ABBA',str)
True
```

Beispiele immer selber testen!

String-Literale notieren → *3*

```
# Einzeilige (!) Zeichenkette
# mit Escape-Sequenzen
s1 = "a\n\x61"
# Rohe Sequenz r"..."
# ohne Escapes
s2 = r"a\n\x61"

print "Canonical s2: ",repr(s2)
```

```
s3 = """a
a"""

print "Canonical s3: ",repr(s3)
a"""

print "Canonical s4: ",repr(s4)
print "Printed s4: ",s4
```

Datentyp `unicode`: Folgen von Unicodes

Unicode Zeichenkodes

```
>>> ord(u'€')
8364
>>> unichr(8364)
u'\u20ac'
```

Datentyp bestimmen und testen

```
>>> type(u'A')
<type 'unicode'>
>>> type(u'ab') == unicode
True
>>> isinstance('ab',unicode)
False
```

String-Literale notieren → *4*

```
# Einzeilige (!) Zeichenkette
# mit Escape-Sequenzen
s1 = u"\u20aca\n\xe4"

# Rohe Sequenz ur"..."
# \uNNNN werden aufgelöst!
s2 = ur'\u20aca\n\xe4'

# Longstring
s3 = u"""\u20aca

# Roher Longstring
s4 = ur"""€a
\u00e4"""
```

Kodierung der Python-Quellkodes deklarieren

Kodierungskommentar für UTF-8-kodierte Quelltexte

Deklariere Kodierung immer mit Kodierungskommentar, wenn Nicht-ASCII-Zeichen vorkommen!

Datei in UTF-8-Kodierung → *5* → *6*

```
#!/usr/bin/python
# -*- coding: utf-8 -*-

print "Length of 'a':", len('a'), "Canonical:", repr('a')
print "Length of 'ä':", len('ä'), "Canonical:", repr('ä')
print "Length of u'a':", len(u'a'), "Canonical:", repr(u'a')
print "Length of u'ä':", len(u'ä'), "Canonical:", repr(u'ä')
```

Für Latin-1: `# -*- coding: iso-8859-1 -*-`

iso-8859-1 ist in Python 2 Standard.

1.2.3 Umkodieren

Enkodieren und Dekodieren von Zeichenketten

Explizites Dekodieren von UTF-8-Repräsentation

```
>>> text = 'B\xc3\xa4h' # UTF-8-Repräsentation von Bäh
>>> unicodetext = text.decode('utf-8')
```

Es entsteht ein Unicode-Objekt!

Explizites Enkodieren von Unicode-Zeichen als UTF-8-Bytefolge

```
>>> unicode_text = u'Bäh'
>>> utf8_text = unicode_text.encode('utf-8')
```

Es entsteht eine Byte-String!

Das Modul `codecs`

`codecs`: Kodieren und Dekodieren

Funktionen für Lesen und Schreiben von Unicode-Strings

Einlesen von Latin-1 und schreiben von UTF-8 → 7

```
import codecs

# Decode from l1 encoded file into unicode strings
f = codecs.open("./ae-l1.txt", "r", "l1")

# Encode unicode strings into UTF-8 encoded file
g = codecs.open("./AE-l1-encoded-as-utf8.txt", "w", "utf-8")
for line in f:
    g.write(line.upper())
```

Beim Einlesen entstehen Zeichenketten vom Typ `unicode`.

1.3 Regex in Python

Warum *raw strings* für Reguläre Ausdrücke in Python?

- Generell empfohlen in http://docs.python.org/2/library/re.html.
- Für viele Escape-Sequenzen macht es zwar keinen Unterschied, weil Python-Strings und Reguläre Ausdrücke letztlich dieselben Zeichen bedeuten: `\a,\f,\n,\r,\t,\v`
- Andere Escape-Sequenzen existieren nur in der Regulären Notation und Python lässt den Backslash stehen: `\A,\B,\w,\W,\d,\D,\s,\S`
- *Aber* andere Reguläre Notationen würden beim Einlesen von Nicht-Raw-Strings missinterpretiert: Um einen einzelnen Backslash zu matchen, müssten wir schreiben: `re.match("\\\\", "\\")`
- `\b`: Bell-Zeichen (ASCII-Code 8) im String; aber Grenze zwischen Wortzeichen und Nicht-Wortzeichen in Regex. `re.sub("\\bthe\\b","THE", "Other uses of the")`
- Numerische Rückreferenzen `\1` innerhalb eines Regex

1.3.1 Ersetzen

Funktion `re.sub()`: Ersetzen mit regulären Ausdrücken

Globales Ersetzen mit Rückreferenz → *8*

```
import re

text = u"Hässliche Köche verdürben das Gebräu"

pattern = ur"([aeioäöü]+)"

# Im Ersetzungstext können gematchte Gruppen eingefügt werden.
# \N (N ist die N-te gruppierende Klammer im Pattern)
replacement = ur"[\1]"

print re.sub(pattern, replacement, text)
```

`replacement` ist eine Zeichenkette, kein regulärer Ausdruck! Falls nichts gematcht wird, bleibt die Zeichenkette unverändert!

Gruppierung mit/ohne Rückreferenzen

Runde Klammern ergeben referenzierbare Gruppierungen

```
>>> text = 'Blick-Leser, A-Post-Fans, andere Bindestrich-Komposita'
>>> re.sub(r'(\w+-)+(\w+)', r'\2', text)
```

Nicht alle Gruppen müssen referenzierbar sein! (Effizienzgründe!)

Nichtreferenzierbare Gruppierung: `(?:REGEX)`

```
>>> text = 'Blick-Leser, A-Post-Fans, andere Bindestrich-Komposita'
>>> re.sub(r'(?:\w+-)+(\w+)', r'\1', text)
```

1.3.2 Suchen

Funktion `re.findall()`: Globale Suche

Alle nicht-überlappenden Matches extrahieren → *9*

```
import re

text = u"Viele Köche verderben den Brei."
pattern = ur"(\w+)"

# Alle Matches finden
m = re.findall(pattern, text)

for g in m:
    print g
```

Pattern und Text müssen immer vom gleichen String-Typ sein!

Regex-Alternative ist nicht kommutativ!

Die Reihenfolge in einer Regex-Alternative ist nicht beliebig!

```
import re
print re.findall(r'a|aa',"Saal")
print re.findall(r'aa|a',"Saal")
```

Gruppierung mit/ohne Rückreferenzen: `re.findall()`

`re.findall()` und gruppierende Klammern

Unterschiedliche Funktionalität, falls gruppierende Klammern im regulären Ausdruck sind oder nicht!

- **Ohne**: Liste der Matches
- **Mit**: Liste von Tupeln, wobei *jedes* Tupel-Element den gematchten Inhalt der entsprechenden gruppierenden Klammer enthält.

```
>>> re.findall(r'a(h)|a(a)', "kahler Saal") [('h', ''), ('', 'a')]
```

Unicode Flag: Was ist ein Wortzeichen?

Unicode-Kategorien aktivieren → *10*

```
import re

text = u"Viele Köche verderben den Brei."
pattern = ur"(\w+)"

# Das Flag (?u) aktiviert Unicode-Kategorien fuer \w und \b
pattern = ur"(?u)(\w+)"

# Resultat ist eine Liste
m = re.findall(pattern, text)

for s in m:
    print s
```

Das Unicode-Flag `(?u)` zählt nicht als Gruppe wie alle `(?...)`.

Lesbare und kommentierte reguläre Ausdrücke

Was matcht dieser Ausdruck?
`(?:[A-Z]\.)+|\w+(?:-\w+)*|\$?\d+[.\d]*%?|\.\.\.|[.,;?]+|\S+`

Lesbare und kommentierbare Ausdrücke dank Flag `(?x)` → *11*

```
import re
text = "That U.S.A. poster-print costs $12.40..."
pattern = r'''(?x)    # set flag (?x) to allow verbose regexps
     (?:[A-Z]\.)+     # abbreviations, e.g. U.S.A.
   | \w+(?:-\w+)*     # words with optional internal hyphens
   | \$?\d+[.,\d]*%?   # currency and percentages, $12.40, 82%
   | \.\.\.           # ellipsis
   | [.,;?]+          # interpunctuation
   | \S+              # catch-all for non-layout characters
   '''
m = re.findall(pattern,text)
print m
```

Kontrollfragen

- Was passiert und was entsteht, wenn ein Ausdruck wie `type(2 == 1*1+1)` evaluiert wird? ▶▶▶
- Können alle Ausdrücke evaluiert werden?

- Was ist der Unterschied zwischen der Notation von String-Literalen mit `'...'` oder `"..."`?
- Worin unterscheiden sich `r'...'` und `'...'`?
- Worin unterscheiden sich `"""..."""` und `"..."`?
- Wie kann man Reguläre Ausdrücke lesbar schreiben?
- Wie weitet man Reguläre Notationen wie `\w` von reinem ASCII auf alle alphanumerischen Zeichen von Unicode aus?
- Wie kann man in einer Zeichenkette alle mit einem Regulären Ausdruck gematchten Teilzeichenketten ersetzen?

Kapitel 2

NLTK-Buch Kapitel 1

Lernziele

NLTK

- NLTK installieren, kennenlernen und selbst anwenden
- Korpuslinguistische Funktionen aufrufen

Technisches

- Was sind Module und Packages? Wie kann man sie importieren?
- Was ist Listenkomprehension? Wie funktioniert sie?
- Was ist bei der Definition von Funktionen zu beachten? Was bewirkt das Statement `return`?
- Was sind globale und lokale Namensräume?

2.1 NLTK

2.1.1 Intro

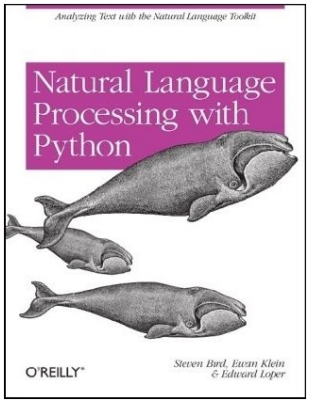
NLTK (Natural Language Toolkit)

NLTK-Frameworkhttp://www.nltk.org

- Sammlung von Open-Source-Python-Modulen für die Sprachverarbeitung (Natural Language Processing, NLP)
- Ressourcen: frei verfügbare Korpora, Treebanks, Lexika ...
- Applikationen: Tokenizer, Stemmer, Tagger, Chunker, Parser, Semantikberechnung ... (teilweise eher Toy-Implementationen für Lehrzwecke)
- Module für Evaluation, Classifier, Clusterer, Maschinelles Lernen (Schnittstellen zu State-of-the-Art-Bibliotheken)
- API (Application Programming Interface) für WordNet und Lexika

Installationsanleitungen für Win, Mac, Linux
http://www.nltk.org/install.html

Bird et. al (2009): Natural Language Processing in Python



- Praktische Einführung in NLP mit Hilfe von NLTK
- Anwendungsorientiert, keine vertiefte Einführung in Python-Konzepte
- Lehrbuch für PCL I (und teilweise II)
- 1. Edition online verfügbar: http://www.nltk.org/book_1ed/
- 2. Edition http://www.nltk.org/book ist kompatibel mit Python 3
- Weiteres NLTK-basiertes Buch mit vielen NLP-Rezepten: [Perkins 2010]

Modulübersicht

Language processing task	NLTK modules	Functionality
Accessing corpora	nltk.corpus	standardized interfaces to corpora and lexicons
String processing	nltk.tokenize, nltk.stem	tokenizers, sentence tokenizers, stemmers
Collocation discovery	nltk.collocations	t-test, chi-squared, point-wise mutual information
Part-of-speech tagging	nltk.tag	n-gram, backoff, Brill, HMM, TnT
Classification	nltk.classify, nltk.cluster	decision tree, maximum entropy, naive Bayes, EM, k-means
Chunking	nltk.chunk	regular expression, n-gram, named-entity
Parsing	nltk.parse	chart, feature-based, unification, probabilistic, dependency
Semantic interpretation	nltk.sem, nltk.inference	lambda calculus, first-order logic, model checking
Evaluation metrics	nltk.metrics	precision, recall, agreement coefficients
Probability and estimation	nltk.probability	frequency distributions, smoothed probability distributions
Applications	nltk.app, nltk.chat	graphical concordancer, parsers, WordNet browser, chatbots
Linguistic fieldwork	nltk.toolbox	manipulate data in SIL Toolbox format

2.1.2 Module und Packages

Verzeichnisstruktur vom NLTK 3

```
$ tree -F /Library/Python/2.7/site-packages/nltk/
/Library/Python/2.7/site-packages/nltk
|-- VERSION
|-- __init__.py
|-- __init__.pyc
|-- align/
|   |-- __init__.py
|   |-- __init__.pyc
|   |-- api.py
|   |-- ibm3.py
|   |-- ibm3.pyc
|-- app/
|   |-- __init__.py
|   |-- __init__.pyc
|   |-- chartparser_app.py
...
|-- util.py
...
```

```
21 directories, 545 files
```

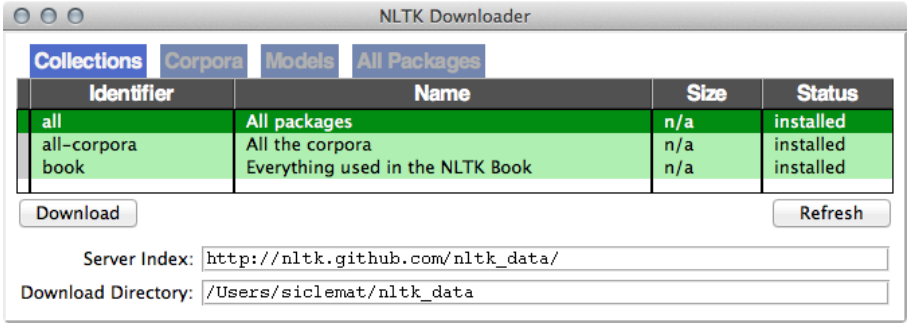
- *Module*: Dateien mit Python-Quellkode: `util.py`
- Maschinenunabhängig kompilierter *Bytekode*: `align.pyc`
- *Packages*: Verzeichnisse wie `nltk` oder `app` mit `__init__.py` bzw. `__init__.pyc`

NLTK-Data: Korpora und weitere Ressourcen

Installation der Ressourcen-Sammlung `nltk_data`

Um die Beispiele im Buch ausführen zu können, muss das Verzeichnis `nltk_data` runtergeladen werden. Nur die Kollektion `book` ist notwendig für PCL I.

```
>>> import nltk
>>> nltk.download()
```



Module importieren

Anweisung: `import Module` → *12*

```
# Importiere Modul book aus Package nltk
import nltk.book

# Objekte und Funktionen aus nltk.book können nur in
# vollqualifizierter Punktnotation bezeichnet werden.
print "Zweites Wort aus text1:", nltk.book.text1[1]

# Objekte und Funktionen können nicht direkt bezeichnet werden:
print text1[1]
```

Alle Objekte und Funktionen aus Modulen importieren

Anweisung: `from Module import *` → *13*

```
# Lade Modul book aus Package nltk und
# importiere alle Objekte und Funktionen ins aktuelle Modul
from nltk.book import *

# Objekte und Funktionen aus nltk.book können ohne
# Modulpräfixe bezeichnet werden.
print "Zweites Token aus text1:", text1[1]

# Die vollqualifizierter Punktnotation geht dann nicht
print "Zweites Wort aus text1:", nltk.book.text1[1]
```

2.1.3 Korpuslinguistische Demo

Eine Tour durch Kapitel 1

- Repräsentation von Text-Korpora als Objekt vom Typ Text (im Wesentlichen als Liste von String-Token)
- KWIC (*Keyword in context*): Konkordanzen erstellen und anzeigen
- Vorkommensähnlichkeit (*similarity*): Welche unterschiedlichen Wörter erscheinen häufig in ähnlichen Kontexten?
- Häufigkeitsverteilungen (*frequency distribution*) berechnen: Wie oft kommt welche Wortform vor?
- Statistische Kollokationen (*collocations*): Welche Wortpaare kommen viel häufiger zusammen vor als zufällig zu erwarten wäre?
- Dispersion-Plot (Korpuslinguistik): An welchen Stellen in einem Korpus kommt ein Wort vor? [Baker et al. 2006]

Visualisierungen mit Plotting benötigen separate Diagramm-Bibliothek matplotlib (Download via matplotlib.org).

2.2 Technisches

2.2.1 Listenkomprehension

Listenkomprehension (list comprehension)

Mathematische Mengenkomprehension ▶▶▶

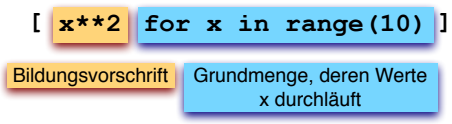
Die Menge aller Quadratzahlen aus der Grundmenge der Zahlen von 0 bis 9.

$$\{x^2 \mid x \in \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}\}$$

In Python mit Listen als Mengen:

```
>>> [x**2 for x in range(10)]
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Syntax



Listen ohne Listenkomprehension

Auftrag: Baue eine zu Liste `a` äquivalente Liste `b` auf, ohne Listenkomprehension zu benützen, dafür aber mit einer normalen For-Schleife?

```
a = [x**2 for x in range(10)]
```

Listenkomprehension mit Bedingungen

Syntaxschema (aus [Bird et al. 2009, 19])

$$\{w \mid w \in V \,\&\, P(w)\}$$

`[w for w in V if p(w)]`

Die Liste aller Elemente w aus V, für die die Eigenschaft $P(w)$ wahr ist.

Filtern von Vokabularlisten → *14*

```
from nltk.book import *

words = set(text1)

longwords = [w for w in words if len(w)>15]
```

`set(text1)` erzeugt Menge aller Listenelemente aus `text1`.

2.2.2 Funktionen

Funktionen definieren und aufrufen

Definition der einstelligen Funktion `foo()` → *15*

```
def foo(a):
    b = 0
    for item in a:
    b += item
    return b
```

Statement `return` bestimmt den *Rückgabewert*, d.h. den Funktionswert. Die *Parameter* der Funktion (Platzhalter für Argumente) stehen in Klammern (komma-getrennt).

Funktionsaufruf (call)

```
c = foo([5,10,23])
```

Mehrere `return`-Anweisungen

Effekt der return-Anweisung → *16*

```
def describe_number(n):
    if n > 1000000:
        return "LARGE"
    elif n > 1000:
        return "Medium"
    else:
        return "small"
    print "Never printed!"
```

- Verarbeitung der `return`-Anweisung *beendet* die Ausführung der Funktion.
- Beliebige Objekte können als Funktionswert zurückgegeben werden, auch Listen.

Wann und wozu sind Funktionen gut?

```
def foo(a):
    b = 0
    for item in a:
    b += item
    return b
```

Heilmittel gegen Spaghettikode

- *Abstraktion*: Eine Funktion kann einige Zeilen Kode bezeichnen, welche oft gebraucht werden.
- *Schnittstelle*: Die Parameter einer Funktion machen den Kode an ganz bestimmten Stellen variabel (= Parametrisierung).
- *Klarheit*: Eine gute Funktion hat eine klar beschreibbare (=spezifizierbare) Funktionalität.

Funktionen und eingerückte Code-Blöcke I

Counting words

```
def wc1(textfile):
    c = 0
    for line in textfile:
        for word in line.split():
            c += 1
    return c
```

Fragen

1. Wann stoppt die Funktion `wc1()`?
2. Was berechnet sie?

Funktionen und eingerückte Code-Blöcke II

Counting words

```
def wc2(textfile):
    c = 0
    for line in textfile:
        for word in line.split():
            c += 1
        return c
```

Fragen

1. Wann stoppt die Funktion `wc2()`?
2. Was berechnet sie?

Funktionen und eingerückte Code-Blöcke III

Counting words → *17*

```
def wc3(textfile):
    c = 0
    for line in textfile:
        for word in line.split():
            c += 1
            return c
```

Fragen

1. Wann stoppt die Funktion `wc3()`?
2. Was berechnet sie?

2.2.3 Namensräume

Skopus (Erreichbarkeit) von Variablennamen

Modulweit erreichbare globale Variablen (*globals*)

- (Variablen-)Namen, die in einem Modul zugewiesen werden, sind danach im ganzen Modul erreichbar.
- Modul `foo` ist Python-Quellkode aus Datei `foo.py`.

Funktionsweit erreichbare lokale Variablen (*locals*)

- Parameter `a` und `b` einer Funktion `foo(a,b)`, sind nur innerhalb der Funktion `foo()` erreichbar.
- (Variablen)-Namen, die in einer Funktion definiert werden, sind nur innerhalb der Funktion erreichbar.

Introspektion

Die eingebauten Funktionen `globals()` und `locals()` geben die zum Aufrufzeitpunkt definierten Namen aus.

Globale und lokale Variablennamen

Derselbe Name kann global und lokal unterschiedliche Werte haben.

Auszug aus `globals_and_locals.py` ▶▶▶ → *18*

```
a = "Globale Variable"

def foo(a):
    print "In Funktion: a =", a
    return a

c = foo("Lokale Variable")

print "In Modul: a =", a
```

Lokale Variablennamen nur lokal!

Erreichbarkeit aus Funktionsdefinitionen ►►► → *19*

```
g = "Globale Variable"

def foo():
    a = g
    return a

def bar():
    b = a
    return b

foo()
bar()
```

Was passiert?

Wie und wo entstehen überall Variablennamen?
Sind die entstehenden Namen lokal oder global?

2.3 Vertiefung

- Pflichtlektüre: Kapitel 1.1. bis und mit 1.4 aus [Bird et al. 2009]
- Enthält nochmals anschauliche Repetition zu vielen bisher behandelten Themen (Listen, Zeichenketten, Bedingungen)

Kapitel 3

NLTK-Buch Kapitel 2

Lernziele

NLTK

- Zugriff auf Textkorpora und POS-annotierte Korpora
- Univariate und bivariate Häufigkeitsverteilungen von Ereignissen: Eine Datenstruktur für einfache deskriptive Statistiken

Technisches

- Listenkomprehension mit Tupeln
- Klassen/Typen und Objekte
- Attribute und Methoden
- Anweisungen und Ausdrücke
- Lambda-Ausdrücke und Listenkomprehensionsausdrücke und ihre verwandten Statements

3.1 Korpora

Gutenberg-Projekte: Elektronische Edition älterer Texte

Definition 3.1.1 (Korpus (sächlich, im Plural Korpora))**.** Ein Korpus ist eine Sammlung von Texten.

Sammlung vorwiegend englischsprachiger Bücher

Sammlung von über 46'000 frei verfügbaren Büchern, deren Copyright abgelaufen ist in den USA. http://www.gutenberg.org

Sammlung deutschsprachiger Bücher

Sammlung von über 7'000 frei verfügbaren Büchern, deren Copyright abgelaufen ist in Deutschland. D.h. 70 Jahre nach dem Tod des Autors oder Übersetzers. http://gutenberg.spiegel.de

3.1.1 Korpora einlesen

Zugriff auf Korpora[1]

Das Modul `nltk.corpus`
Enthält Korpus-Objekte, über die Korpora in verschiedensten Formaten eingelesen werden können.

Das Objekt `nltk.corpus.gutenberg`
Stellt eine Auswahl von 18 englischsprachigen Gutenberg-Texten als Teil der NLTK-Korpusdaten zum Einlesen zur Verfügung.

Download von Korpora
Mit Hilfe von `nltk.download()` lassen sich weitere Korpora einfach downloaden und installieren.

Funktionen des Objekts `nltk.corpus.gutenberg`

Repräsentationen für reine Text-Korpora → *20*

```
from nltk.corpus import gutenberg

filename = 'austen-emma.txt' # oder absoluter Pfad einer Textdatei

# Korpus als eine lange Zeichenkette
emma_chars = gutenberg.raw(filename)

# Korpus als Liste von Wörtern (Wort ist Zeichenkette)
emma_words = gutenberg.words(filename)

# Korpus als Liste von Sätzen (Satz ist Liste von Wörten)
emma_sents = gutenberg.sents(filename)

# Korpus als Liste von Paragraphen (Paragraph ist Liste von Sätzen)
emma_paras = gutenberg.paras(filename)
```

Methoden des Objekts `nltk.corpus.brown`[2]
Zusätzlich zu den Funktionen von Textkorpora, gibts Listen mit *Paaren* aus einem *Token* und *seinem POS-Tag.*

Repräsentationen für getaggte Korpora → *21*

```
from nltk.corpus import brown

# Korpus als Liste von 2-Tupeln (Wort, POS-Tag)
brown_tagged_words = brown.tagged_words()
```

Eigenheiten des Brownkorpus: Unterschiedliche Textsorten

```
# Das balancierte Korpus umfasst Texte aus 15 Kategorien
brown.categories()
```

[1] http://www.nltk.org/api/nltk.corpus.html
[2] http://en.wikipedia.org/wiki/Brown_Corpus

Word Tally

the	𝍸 𝍸 𝍸 ||||
been	𝍸 𝍸 |
message	||||
persevere	|
nation	𝍸 |||

3.1.2 Korpus-Typen

Arten von Korpora: Korpus-Typologie

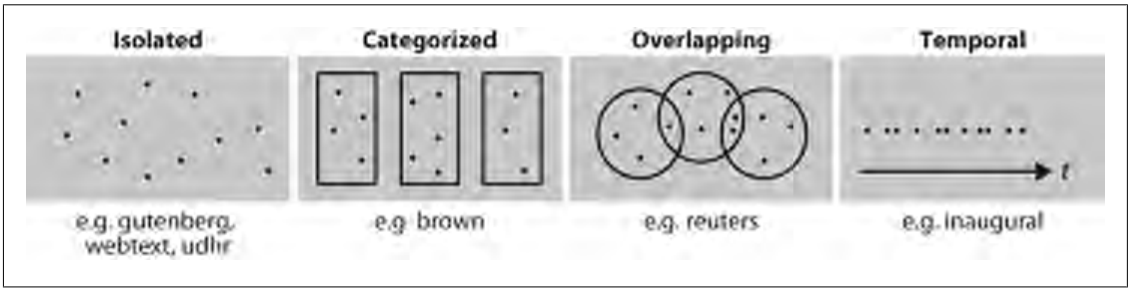


Figure 2-3. Common structures for text corpora: The simplest kind of corpus is a collection of isolated texts with no particular organization; some corpora are structured into categories, such as genre (Brown Corpus); some categorizations overlap, such as topic categories (Reuters Corpus); other corpora represent language use over time (Inaugural Address Corpus).

Quelle: [Bird et al. 2009, 50]

- Die Texte in einem Korpus (Textsammlung, d.h. mehrere Texte) können in unterschiedlicher Ordnung zueinander stehen.
- Ein Korpus kann *balanciert* (repräsentativ zusammengestellt) oder *opportunistisch* (nimm, was du kannst!) sein.

3.2 Häufigkeitsverteilungen

3.2.1 Univariat

Erbsenzählerei: Häufigkeiten ermitteln

- Deskriptive Statistiken: *Fundamentale Funktionalität* in korpuslinguistischen Auswertungen
- Buchstaben, Wörter, Eigennamen, Sätze, Paragraphen *zählen*
- Minima, Maxima und Mittelwerte ermitteln
- Verteilung der Häufigkeiten *darstellen* (Tabelle, Plots)
- Letztlich: Verteilungen *vergleichen*

Häufigkeitsverteilungen als allgemeine Datenstruktur

- Allgemein: *Häufigkeit der Items* einer variierenden Grösse (eine statistische Variable) auszählen
- NLTK-Klasse `nltk.FreqDist` ist Datenstruktur zum einfachen Erstellen von Häufigkeitsverteilungen (*frequency distribution*)

(Abstrakte) Datenstrukturen

"In der Informatik und Softwaretechnik ist eine *Datenstruktur ein Objekt zur Speicherung und Organisation von Daten.* Es handelt sich um eine Struktur, weil die Daten in einer bestimmten Art und Weise angeordnet und verknüpft werden, um den Zugriff auf sie und ihre Verwaltung effizient zu ermöglichen. Datenstrukturen sind nicht nur durch die enthaltenen Daten charakterisiert, sondern vor allem durch die *Operationen auf diesen Daten, die Zugriff und Verwaltung ermöglichen und realisieren.*" (http://de.wikipedia.org/wiki/Datenstruktur)

Funktionen der Klasse `nltk.FreqDist`

Table 1-2. Functions defined for NLTK's frequency distributions

Example	Description
`fdist = FreqDist(samples)`	Create a frequency distribution containing the given samples
`fdist.inc(sample)`	Increment the count for this sample
`fdist['monstrous']`	Count of the number of times a given sample occurred
`fdist.freq('monstrous')`	Frequency of a given sample
`fdist.N()`	Total number of samples
`fdist.keys()`	The samples sorted in order of decreasing frequency
`for sample in fdist:`	Iterate over the samples, in order of decreasing frequency
`fdist.max()`	Sample with the greatest count
`fdist.tabulate()`	Tabulate the frequency distribution
`fdist.plot()`	Graphical plot of the frequency distribution
`fdist.plot(cumulative=True)`	Cumulative plot of the frequency distribution
`fdist1 < fdist2`	Test if samples in `fdist1` occur less frequently than in `fdist2`

Anwendung der Klasse `nltk.FreqDist`

Berechnen der häufigsten längsten Wörter → 22

```
import nltk
from nltk.corpus import gutenberg

emma_words = gutenberg.words('austen-emma.txt')

emma_fd = nltk.FreqDist(emma_words)
```

```
# Finde alle Wörter für die gilt:
# - mehr als 10 Buchstaben und
# - kommen mindestens 10 mal vor

wl = sorted([w for w in emma_fd.keys()
           if len(w)>10 and emma_fd[w]> 7])
```

3.2.2 Bivariat

Bivariate (bedingte) Häufigkeitsverteilungen

Condition: News		Condition: Romance	
the	𝍸 𝍸 𝍸 ||||	the	𝍸 𝍸 |||
cute		cute	|||
Monday	𝍸 ||||	Monday	|
could	|	could	𝍸 𝍸 𝍸
will	𝍸 |||	will	||||

Gemeinsame Häufigkeit der Items von *2 variierenden Grössen* (zweier statistischer Variable) auszählen

- Sprechweise: Die eine Variable heisst in NLTK Bedingung (*condition*), die andere Ereignis (*event*, *sample*)
- Eine bedingte Häufigkeitsverteilung besteht pro Bedingung aus *einer einfachen Häufigkeitsverteilung.*
- NLTK-Klasse `nltk.ConditionalFreqDist` umfasst geeignete Methoden für Frequenzdistributionen von *Paaren (=2er-Tupel): (condition,sample)*
- Beispiel: Mit den 15 Kategorien im Brownkorpus ergeben sich 15 Bedingungen mit insgesamt 1'161'192 Events (Wörtern).

Bedingte Häufigkeiten berechnen

Modalverben in Abhängigkeit von Textkategorien → *23*

```
import nltk
from nltk.corpus import brown

cfd = nltk.ConditionalFreqDist([
    (genre, word)
    for genre in brown.categories()
    for word in brown.words(categories=genre)])

genres = ['news', 'religion', 'hobbies',
        'science_fiction', 'romance', 'humor']

modals = ['can', 'could', 'may', 'might', 'must', 'will']

cfd.tabulate(conditions=genres, samples=modals)
```

Type	*Mutable*	0	1	2
list	Ja	`[]`	`[1]`	`[1,'n']`
str	Nein	`''`	`'1'`	`'ab'`
tuple	Nein	`()`	`(1,)`	`(1,'n')`

Funktionen der Klasse `nltk.ConditionalFreqDist`

Table 2-4. NLTK's conditional frequency distributions: Commonly used methods and idioms for defining, accessing, and visualizing a conditional frequency distribution of counters

Example	Description
`cfdist = ConditionalFreqDist(pairs)`	Create a conditional frequency distribution from a list of pairs
`cfdist.conditions()`	Alphabetically sorted list of conditions
`cfdist[condition]`	The frequency distribution for this condition
`cfdist[condition][sample]`	Frequency for the given sample for this condition
`cfdist.tabulate()`	Tabulate the conditional frequency distribution
`cfdist.tabulate(samples, conditions)`	Tabulation limited to the specified samples and conditions
`cfdist.plot()`	Graphical plot of the conditional frequency distribution
`cfdist.plot(samples, conditions)`	Graphical plot limited to the specified samples and conditions
`cfdist1 < cfdist2`	Test if samples in `cfdist1` occur less frequently than in `cfdist2`

3.3 Technisches

3.3.1 Sequenzen

Sequenz-Datentypen: list, str, unicode, tuple

Definition 3.3.1 (Sequenz = Endliche Folge von Objekten).

- Zugriff auf *Elemente einer Sequenz* mittels ganzzahligem Index: `s[i]`
- Zugriff auf *Abschnitte* (slice) mittels Angabe von Start- und exklusiver Endposition: `s[start:end]`
- Bestimmen der Anzahl Element mittels `len(s)`

Typen von Sequenzen und ihre Notation

Einertupel braucht Komma! Die runden Klammern sind meist weglassbar.

Listen: Veränderliche (*mutable*) Sequenzen

▶▶▶

Typische Modifikationen für Listen

```
l = []
l.append(1)         # ein Element anhängen
l.extend((4,'x',5)) # eine ganze Sequenz anhängen
del l[3]            # ein Element löschen
```

```
l[2] = 3            # eine Element austauschen
l.sort(reverse=True) # in-place rückwärts sortieren
print l
```

- *Nur bei veränderlichen (mutable) Sequenzen*, d.h. Listen, können Elemente (oder Abschnitte (*slices*)) gelöscht, ersetzt oder ergänzt werden.

- : Methoden, welche in-place-Modifikationen durchführen, liefern als Rückgabewert `None` zurück!

Wozu braucht's Tupel und Listen?

Wozu braucht's Tupel?

- `dict` können *nur unveränderliche Keys* haben. Also keine Listen!
- Der Mengentyp `set` kann nur unveränderliche Elemente haben.
- Typischerweise dort, wo eine Sequenz erzeugt wird, deren Elemente nicht modifiziert werden (sollen).

Wozu braucht's Listen?

- Speicher-effiziente Modifikation von *Elementen* der Sequenz (Löschen, Ersetzen, Einfügen). `sorted(Liste` ist nicht speichereffizient!
- Für *In-Place*-Sortieren via `my_list.sort()`. Im Gegensatz zur Funktion `sorted()`, welche eine frisch erzeugte, sortierte Liste als Funktionswert zurück liefert.

Syntaktischer Zucker für Methoden von Sequenzen
Python bietet für wichtige Methoden von Sequenzen *Spezialnotation* an. Ob die Spezialnotation funktioniert, hängt nur davon ab, ob mein Objekt die entsprechende Methode kann!

Enthalten (Membership)
```
>>> 3 in [1,2,3]
True
>>> [1,2,3].__contains__(3)
True
```

Abschnitt (Slicing)
```
>>> "ABBA"[1:3]
'BB'
>>> "ABBA".__getslice__(1,3)
'BB'
```

i-tes Element
```
>>> ('a','c')[1]
'c'
>>> ('a','c').__getitem__(1)
'c'
```

```
 >>> help(str.__getitem__)
Help on wrapper_descriptor:
__getitem__(...)
    x.__getitem__(y) <==> x[y]
```

Gattungen und Individuen in der Welt

Gattung	Individuum
Mensch	Elvis Presley
Hauptstadt	Paris

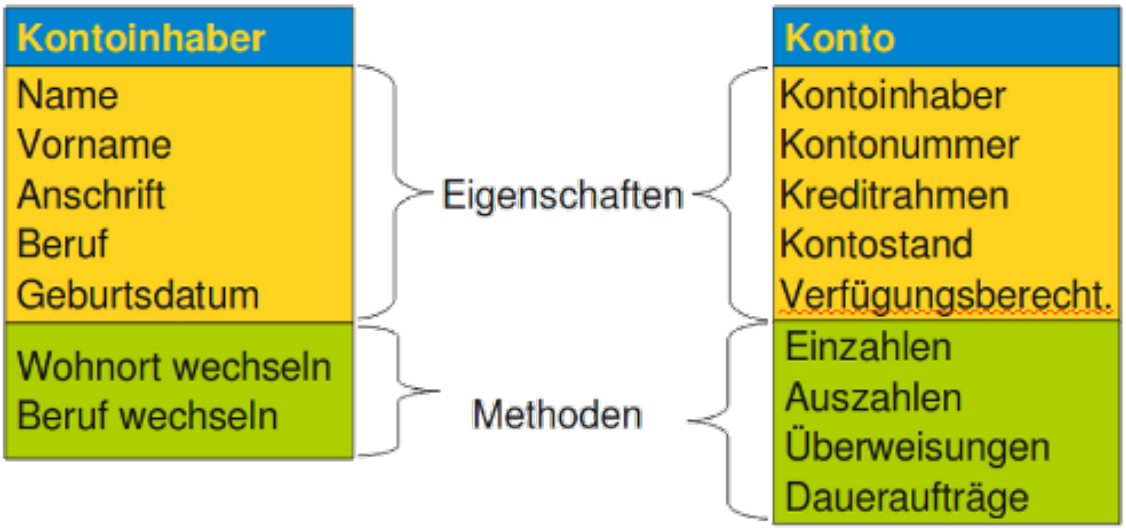
Typen/Klassen und Objekte/Klassen-Instanzen in Python

Typ/Klasse	Objekt/Instanz
int	`3`
str	`'abc'`
unicode	`u'abc'`
nltk.probability.FreqDist	`nltk.FreqDist([1,1,2])`
nltk.text.Text	`nltk.Text(["a","text"])`

3.3.2 Klassen & Objekte

Klassen und Objekte

Objektorientierte Modellierung



Quelle: http://www.python-kurs.eu/klassen.php Mit Hilfe von *Klassendefinitionen* können *eigene (Daten-)Typen* geschaffen werden.

Typen/Klassen und Objekte
Wichtig: Objekte sind Instanzen eines Typs oder einer Klasse.

Typ-Aufrufe als Objekt-Konstruktoren
Konstruiere Objekte von einem bestimmten Typ, indem du den Typ wie eine Funktion aufrufst!

Default-Objekte

```
>>> str()
''
>>> int()
0
>>> list()
```

```
[]
>>> dict()
{}
>>> set()
set([])
```

Viele Konstruktor-Funktionen erlauben *Argumente*. Erklärungen gibt `help(type)`.

Konstruktoren mit Parametern

```
>>> str(123)
'123'
>>> int('10110',2)
22
>>> set([3,3,2,2,'a',1.1,'a'])
set(['a', 2, 3, 1.1])
>>> list(set([2,1,'a']))
['a', 1, 2]
>>> dict(a='DET',do='VB')
{'a': 'DET', 'do': 'VB'}
```

Methodenaufrufe

Methoden werden normalerweise auf der Ebene der Klasse/Typs *definiert*, aber normalerweise direkt von den Objekten *aufgerufen*.

Syntaktischer Zucker für Typen

`type.function(object)` ≡ `object.function()`

`str.lower("A Test")` ≡ `"A Test".lower()`

Syntaktischer Zucker für Klassen

`class.function(object)` ≡ `object.function()`

`nltk.Text.vocab(nltk.Text(["a","b","c"]))` ≡ `nltk.Text(["a","b","c"]).vocab()`

Jeder Methodenaufruf von einem Objekt aus hat als 1. Argument implizit das Objekt, das die Methode aufruft: `vocab(self)`. → 24

3.3.3 Statements und Expressions

Unterschied zwischen *Statements* und *Expressions*

Anweisungen (*statements*) → **25**
werden vom Python-Interpreter ausgeführt und evaluieren zu *keinem Wert*.

`print` Statement → *26*

```
print "Something to print"
```

Ausdrücke (*expressions*) → **27**
werden zu einem Wert (Objekt) evaluiert und enthalten *keine Statements*.

Boole'sche und andere Ausdrücke innerhalb von Statements

```
# If-Statement mit komplexen Ausdrücken drin
if len("A "+"String") > 5:
    print "A "+"String".lower()
```

Listenbildung via Anweisungen und Ausdruck

Listenbildung mit iterativen Statements

```
sl = list()
for c in "St. Moritz-Str. 23":
    if c.isalnum():
        sl.append(c.lower())
```

Listenbildung mit einem Ausdruck: Listenkomprehension

```
el = [c.lower() for c in "St. Moritz-Str. 23" if c.isalnum()]
```

If-then-else als Anweisung und If-Else als Ausdruck

Listenbildung mit iterativen Statements

```
sl = []
for c in "St. Moritz-Str. 23":
    if c.isalnum():
        sl.append(c)
    else:
        sl.append(' ')
```

Default-if-else Ausdruck

```
el = [ c if c.isalnum() else ' ' for c in "St. Moritz-Str. 23" ]
```

: Abweichende Reihenfolge von if-then-else-Bestandteilen, da typischerweise der Then-Ausdruck der Standardwert ist.

Funktionsdefinition via Anweisungen und Ausdruck

Funktionsdefinition mit iterativem Statement → *28*

```
def sf(s):
    return re.sub(r'\s+','',s)
```

Funktionsdefinition via Lambda-Ausdruck

```
ef = lambda s: re.sub(r'\s+','',s)
```

Lambda-Ausdrücke (Lambda expression))

Mathematische Notation zur Definition von anonymen Funktionen:

- Funktionsdefinition (Rechenvorschrift): $(\lambda x : x + 1)$
- Funktionsevaluation: $(\lambda x : x + 1)(3) = 4$
- Lambda bindet/abstrahiert die Funktionsparameter im Funktionsrumpf
- Kurz: Parametrisierte Ausdrücke

Komprehension von Mengen und Dictionaries

Mengenkomprehension und iterative Lösung → *29*

```
mc = {x.lower() for x in "Das Alphabet" if x.isalnum()}

ms = set()
for x in "Das Alphabet":
    if x.isalnum():
        ms.add(x.lower())
```

Komprehension von Dictionaries

```
text = "abrakadabra"
dc = {c:text.count(c) for c in set(text)}
```

Wie würde man das iterativ programmieren?

3.4 Vertiefung

- Pflichtlektüre: Kapitel 2.1. bis und mit 2.4 aus [Bird et al. 2009]

Kapitel 4

NLTK-Buch Kapitel 2: Lexikalische Ressourcen

Lernziele I

NLTK

- Zugriff auf lexikalische Ressourcen
- Effizientes Filtern von Stoppwörtern auf Textkorpora
- Lexika mit komplexer Datenstruktur

Technisches

- Ober- und Unterklassen verstehen
- Eigene Klassen definieren
- Konstruktorfunktion `__init__()` von Klassen verstehen
- Definieren von eigenen Methoden und Attributen

4.1 Lexika

4.1.1 Wortlisten

Wortlisten als Lexika

Definition 4.1.1 (Wortlisten)**.** Die einfachste Form von Lexika sind *Wortlisten*. Als Rohtext-Datei typischerweise 1 Wort pro Zeile und sortiert.

Stoppwortlisten (stopwords) in NLTK → *30*

```
stopwords_en = nltk.corpus.stopwords.words('english')

print len(stopwords_en), stopwords_en[::20]
# >>> 127 ['i', 'herself', 'was', 'because', 'from', 'any', 't']
```

Hinweis: Spezialsyntax `[::20]` gibt jedes 20. Element zurück.

Rechnen mit Stoppwortlisten

Was berechnet `foo()`? Was wäre ein guter Funktionsname? → *31*

```
import nltk
stopwords_en = nltk.corpus.stopwords.words('english')

# Was berechnet foo()?
def foo(text):
    """ Hier fehlt Dokumentation... """
    bar = [w for w in text if w.lower() not in stopwords_en]
    return len(bar)/len(text)*100.
```

- Wie kann man besser dokumentieren?
- Wie kann man effizienter berechnen?

Anteil von echten Inhaltswörtern bestimmen

Wie kann man die Interpunktionstoken eliminieren? → *32*

```
import re

def delete_punctuation(s):
    """ Return string with all punctuation symbols of iso-latin 1 deleted. """
    p = r'[!"#%&\x27`()*,-./:;?@[\]_{}\xa1\xab\xb7\xbb\xbf]'
    return re.sub(p,'',s)

def content_word_percentage(text):
    """ Return the percentage of content words in a list of English tokens. """
    content_words = [w for w in text
                     if delete_punctuation(w) != ''
                     and w.lower() not in stopwords_en_set]
```

4.1.2 Aussprachelexika

CMU (Carnegie Mellon University) Pronouncing Dictionary

```
File Format: Each line consists of an uppercased word,
a counter (for alternative pronunciations), and a transcription.
Vowels are marked for stress (1=primary, 2=secondary, 0=no stress).
E.g.: NATURAL 1 N AE1 CH ER0 AH0 L

The dictionary contains 127069 entries.  Of these, 119400 words are assigned
a unique pronunciation, 6830 words have two pronunciations, and 839 words have
three or more pronunciations.  Many of these are fast-speech variants.

Phonemes: There are 39 phonemes, as shown below:

    Phoneme Example Translation    Phoneme Example Translation
    ------- ------- -----------    ------- ------- -----------
    AA      odd     AA D           AE      at      AE T
    AH      hut     HH AH T        AO      ought   AO T
...
$ grep -w  RESEARCH  /Users/siclemat/nltk_data/corpora/cmudict/cmudict
RESEARCH 1 R IY0 S ER1 CH
RESEARCH 2 R IY1 S ER0 CH
```

Wie soll man solche Information in Python als *Daten* repräsentieren?

CMU (Carnegie Mellon University) Pronouncing Dictionary

Strukturierte Lexikoneinträge
CMU besteht aus Paaren von Lemma und Listen von phonetischen Kodes.

Filtern von Lexikoneinträgen → *33*

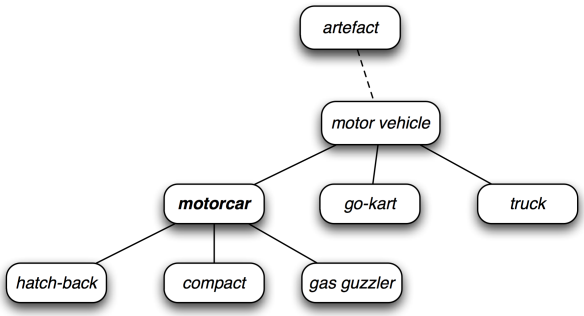
```
import nltk

entries = nltk.corpus.cmudict.entries()

print entries[71607]
# ('love', ['L', 'AH1', 'V'])]

# Finde alle Wörter auf -n, welche als -M ausgeprochen werden.
print [ word for (word,pron) in entries
          if  pron[-1] == 'M'
          and word[-1] == 'n' ]
```

4.1.3 Bedeutungslexika

WordNet: Ein Netz von Bedeutungsbeziehungen[1]

Wie lässt sich die Bedeutung eines Worts angeben?

- Klassische Charakterisierung: Umschreibung, Definition
- Relationale lexikalische Semantik = Bedeutungsbeziehungen
- Angabe von Synonymen, Hypernymen, Hyponymen, Antonymen usw., welche ein Netz (Hierarchie) von verknüpften Bedeutungen ergeben



Quelle: http://www.nltk.org/images/wordnet-hierarchy.png

Abbildung 4.1:

WordNet: Komplexe lexikalische Datenstruktur[2]

Speziell zugeschnittene Datenstruktur benötigt

- Zugriff auf Bedeutungen (*synsets*) und Wörter (*lemmas*)

[1] http://wordnetweb.princeton.edu/perl/webwn
[2] Siehe http://www.nltk.org/book/ch02.html#fig-wn-hierarchy

- Navigation im Wortnetz entlang der semantischen Relationen (Oberbegriffe, Unterbegriffe, Gegenbegriffe)
- Berechnen von semantischer Verwandtschaft (Nähe, Bezüge) im Netz

WordNet in NLTK → *34*

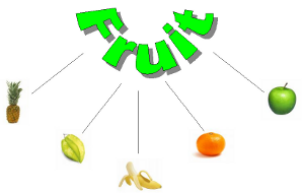
```
import nltk
from nltk.corpus import wordnet as wn
# Welche Bedeutungen hat das Wort "car"?
print wn.synsets('car')
# Definition einer Bedeutung
print wn.synset('car.n.01').definition()
# Alle hyponymen Bedeutungen eines Lemmas berechnen
print wn.synset('car.n.01').hyponyms()
```

4.2 Technisches

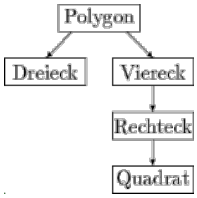
Klassen

Gemeinsame Eigenschaften und Fähigkeiten von Objekten
Was verbindet oder unterscheidet die Objekte der verschiedenen Klassen?

Essbare Objekte



Geometrische Objekte



4.2.1 Klassendefinition

Klassen

- Klassen spezifizieren und implementieren die *Eigenschaften* (Attribute) und *Funktionalitäten* (Methoden) von Objekten.
- Klassen *abstrahieren* gemeinsame Eigenschaften und Funktionalitäten.
- Klassen sind in Unter-/Oberklassen (*superclass/subclass*) organisiert (*Vererbung*).
- Vererbung heisst, dass Eigenschaften/Methoden einer Oberklasse *defaultmässig* auch in der Unterklasse zur Verfügung stehen.

- Die Methoden können in der Unterklasse aber auch *umdefiniert* werden (*Flexibilität*).
- Die allgemeinste Klasse ist in Python heisst `object`.
- Jede selbstdefinierte Klasse hat *mindestens eine Oberklasse*, von der sie eine Unterklasse ist.
- *Mehrfachvererbung* ist möglich, d.h. eine Klasse kann Unterklasse von unterschiedlichen Klassen sein.

Objektorientierte Programmierung (OOP)

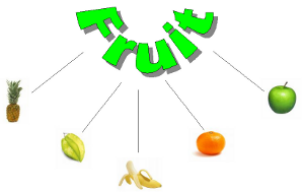
Kernkonzepte nach http://en.wikipedia.org/wiki/Object-oriented_programming

- *Datenkapselung I*: Bündeln von Datenstrukturen und zugehöriger Funktionalität unter einer Adresse (=Objekt)
- *Datenkapselung (Abstraktion) II*: Klare Schnittstelle, welche Attribute und Methoden für öffentliche und welche für private (objektinterne) Zwecke nutzbar sind
- *Klassenzugehörigkeit*: Objekte sind Instanzen einer Klasse
- *Vererbung*: Unterklassen können Attribute/Methoden von ihren Oberklassen erben
- *Dynamische Bindung*: Welche Methode (d.h. Methode von welcher (Ober-)klasse) ein Objekt benutzt, wird erst beim Aufruf der Methode festgelegt anhand der *method resolution order*.
- *Selbst-Parameter* (*self*): Platzhalter für das Instanzobjekt in der Definition einer Klasse

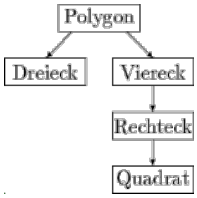
Typen von Klassenhierarchien

ISA-Relation: Beziehung zwischen einer Klasse und ihrer Oberklasse
Ein Apfel ist eine Frucht. Jedes Quadrat ist ein Rechteck.

Flache Hierarchie → **35**



Verschachtelte Hierarchie



Klassenhierachie in NLTK → **36**
Ist die NLTK-Klasse `nltk.FreqDist` ist eine Unterklasse von `dict`?

```
>>> issubclass(nltk.FreqDist, dict)
True
```

Die Eigentümlichkeiten der Oberklasse `object`

Die Klasse `object` ist trotz ihres Namens eine Klasse!

```
>>> help(object)
Help on class object in module __builtin__:
class object
 |  The most base type
```

Objekte (Instanzen) der Klasse `object`

```
>>> o = object()
>>> type(o)
<type 'object'>
```

Gibt es eine Oberklasse der Klasse `object`?

```
>>> issubclass(object, object)
True
```

Klassen definieren: Case-insensitive Strings

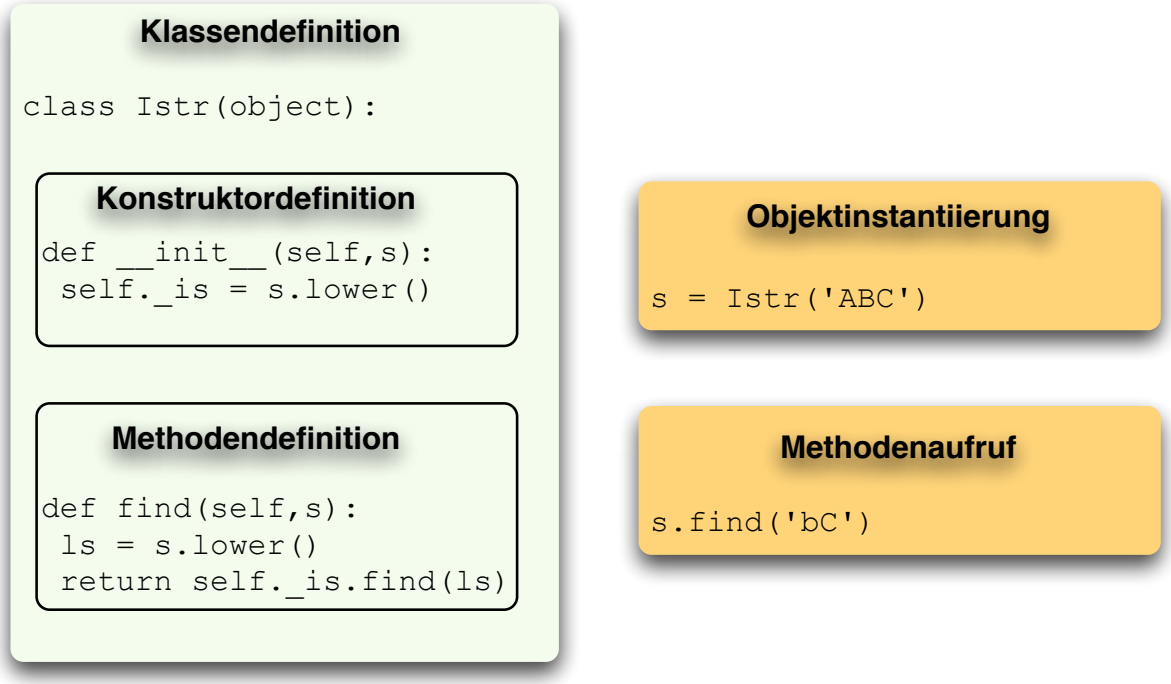
Definition der Klasse, der Konstruktorfunktion und einer Methode → 37

```
class Istr(object):                # Unterklasse von object
    """Case-insensitive string class"""
        self._is = s.lower()       # self ist Instanzparameter
                                   # self._is ist Objektattribut

    def endswith(self, s):         # Methode endswith(s)
        return self._is.endswith(s.lower())
```

Instantiierung eines Objekts und Methodenaufruf

```
s = Istr('ABC') # Konstruktion eines Objekt der Klasse Istr
s.endswith('c') # Methoden-Aufruf
```

Zusammenhang von Definition und Verwendung



4.3 Vertiefung

- Pflichtlektüre: Kapitel 2.1. bis und mit 2.5 aus [Bird et al. 2009]

- Falls ihr mit NLTK 2 (virtuelle Maschine) arbeitet ist die PDF-Version geeignet. Falls ihr NLTK 3 benutzt, ist die Online-Version des Buches geeigneter, um die Beispiele nachzuvollziehen.

- Gutes deutschsprachiges Tutorat http://www.python-kurs.eu/klassen.php oder http://www5.in.tum.de/~/ferstlc/python_kapitel_12_002.htm

Kapitel 5

NLTK-Buch Kapitel 3

Lernziele

NLTK

- KWIC mit eigenen Klassen definieren
- Erstellung eines Index
- Porter-Stemmer für Englisch

Technisches

- Formatierungsausdrücke
- Generatoren als Ausdrücke
- Generatoren mit `yield` in Funktionsdefinitionen
- Ausnahmen (exceptions) behandeln
- Dateien öffnen mit dem with-Konstrukt

5.1 Konkordanzen

Motivation

Ziel: KWIC in Python
Eine Klasse programmieren, welche eine Konkordanz über einem gestemmten Index anzeigt.

Beispiel-Output

```
>>> text.concordance('die', width=30)

 BLACK KNIGHT : Then you shall die . ARTHUR : I command you
Camelot . He was not afraid to die , O brave Sir Robin .
 Concorde , you shall not have died in vain ! CONCORDE : Uh
               2 : Oh , he ' s died ! FATHER : And I want
 that ? MAYNARD : He must have died while carving it . LAUNCE
     ARTHUR : Look , if he was dying , he wouldn ' t bother
```

Beispiel: Konkordanzprogramm über gestemmten Wörtern

KWIC als Klasse: Datenstrukturen und Funktionalitäten

- *Text*: Folge von Wörtern
- *Index*: Abbildung von (gestemmtem) Wort zu allen Vorkommenspositionen im Text
- *Stemmer*: Stemming von Wortformen
- *KWIC-Anzeige*: Formatierung der Treffer im KWIC-Stil

Benötigte Kompetenzen

- Wie lassen sich (einfache) Klassen definieren?
- Definition eigener Regex-Stemmer oder Benutzung von NLTK-Stemmern
- Formatierung von zentriertem textuellem Output mit Format-Ausdrücken

5.1.1 Formatierungsausdrücke

Formatierung mit Hilfe von Format-Ausdrücken

- Flexiblere Kontrolle für Ausgabe von Zahlen und Strings ist erwünscht
- Formatierungsausdruck: `'STRINGTEMPLATE WITH FORMATS' % TUPLE`
- Ein *Formatierungsausdruck* (*string formatting expression*) trennt Layout (Platzhalter `%d`, `%f` für Zahlen, `%s` für Strings) von den variablen Daten (Tupel)
- Anzahl Nachkommastellen (`'%.2f'`), Padding mit Leerzeichen (`'% 4.2f'`), linksbündig (`'%-7s'`), rechtsbündig(`'%7s'`)

```
>>> 'a string:%s and an integer:% 4d' % ('abc',3)
'a string:abc and an integer:   3'
>>> 'Padding a string:%-6s and a float:% 8.2f' % ('abc',3.175)
'Padding a string:abc    and a float:    3.17'
```

Formatierungsausdrücke

Überraschung → *38*

```
>>> '%.1f' % 0.05
'0.1'
>>> '%.1f' % 0.15
'0.1'
>>> round(0.05,1)
0.1
>>> round(0.15,1)
0.1
```

Schulregel mit aufzurundendem .5 verzerrt systematisch (*bias*) → 39

Prozentzeichen schützen mit %

```
>>> '%.1f%%' % 0.15
'0.1%'
```

*Variables Padding mit **

```
>>> width = 8
>>> '%*s' % (width, 'abc')
'     abc'
>>> '%-*s' % (width, 'abc')
'abc     '
```

Regex-Stemmer-Klasse definieren

Klasse mit Default-Argument → *40*

```
class RegexStemmer(object):
    def __init__(self, r=r'^(.*?)(ing|ly|ed|ious|ies|ive|es|s|ment)?$'):
        self._r = r

    def stem(self,word):
        m = re.match(self._r, word)
        return m.group(1)
```

Initialisierung und Verwendung

```
regex_stemmer = RegexStemmer()
regex_stemmer.stem('seeming')
```

5.1.2 Stemmer

Textindex als Klasse `IndexedText` definieren

```
class IndexedText(object):
    def __init__(self, stemmer, text):
        self._text = text
        self._stemmer = stemmer
        self._index = nltk.Index((self._stem(word), i)
                                 for (i, word) in enumerate(text))
```

`enumerate(s)` generiert Paare `(Position,Element)` aus Sequenz

```
l = ['wenn', 'fliegen', 'hinter', 'fliegen', 'fliegen']
>>> list(enumerate(l))
[(0, 'wenn'), (1, 'fliegen'), (2, 'hinter'), (3, 'fliegen'), (4, 'fliegen')]
```

`nltk.Index(Pairs)` erzeugt invertierten Index aus `(Element,Position)`

```
>>> index = nltk.Index((w,i) for (i,w) in enumerate(l))
>>> index['fliegen']
[1, 3, 4]
```

5.1.3 Textindexklasse

Private und öffentliche Methode von `IndexedText`

*Unterstrich markiert Privatheit: Nur für Benutzung **in** der Klasse*

```
    def _stem(self, word):
        return self._stemmer.stem(word).lower()
```

Öffentliche Methode für Formatierung

```
def concordance(self, word, width=40):
    key = self._stem(word)    # stemmed keyword
    wc = width/4              # words of context
    for i in self._index[key]:
        lcontext = ' '.join(self._text[i-wc:i])
        rcontext = ' '.join(self._text[i:i+wc])
        ldisplay = '%*s' % (width, lcontext[-width:])
        rdisplay = '%-*s' % (width, rcontext[:width])
        print ldisplay, rdisplay
```

Noch privater sind Attribute der Form `o.__NAME`.

5.2 Technisches

5.2.1 Generatorausdrücke

Generatorausdrücke (*generator expressions*)

Listenkomprehension: Prinzip "Liste aller Dinge, die ..."
Baue die *Liste* aller kleingeschriebenen Wörter aus dem Brown-Korpus und erzeuge danach aus der Liste eine Menge! `set([w.lower() for w in nltk.corpus.brown.words()])`

Generatorausdrücke: Prinzip "Der Nächste, bitte!"
Nimm ein kleingeschriebenes Wort nach dem andern und mache es zum Element der Menge! `set(w.lower() for w in nltk.corpus.brown.words())`

Listenkomprehension vs. Generatorausdrücke

Generatorausdrücke statt Listenkomprehension
Im NLTK-Buch wird aus Effizienzgründen `set(w.lower() for w in text)` statt `set([w.lower() for w in text]` notiert.

- *Listenkomprehension* erzeugt im Arbeitsspeicher immer eine *Liste aller Elemente*.
- *Generatorausdrücke* sind *speichereffizient*. Sie übergeben ihre Element auf Verlangen einzeln der auswertenden Funktion (intern `g.next()`).
- Generatorausdrücke unterstützten darum *Längenmethode* `len()` nicht.
- Generatorausdrücke unterstützten *kein Slicing*: `l[:10]`.
- Mit `list(generator)` wird jeder Generator zur Liste.
- Speichereffizienz ist bei allen Funktionen optimiert, welche Daten vom Typ *`iterable`* verarbeiten: `max()`, `sum()`, `set()` usw.
- Generatoren sind nach 1 Durchgang erschöpft, d.h. aufgebraucht!

Generatorausdrücke und die Iteratorfunktion `next()` → 41

```
>>> quadrat = (i*i for i in [10,11])
>>> quadrat
<generator object <genexpr> at 0x16a6f80>
>>> type(quadrat)
```

```
<type 'generator'>
>>> quadrat.next()
100
>>> quadrat.next()
121
>>> quadrat.next()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

Die Ausnahme (*exception*) `StopIteration` erscheint, wenn der Generator erschöpft ist. Konsumenten von Generatoren müssen Ausnahme korrekt behandeln.

Generatorfunktionen mit `yield` → 42

```
def quadriere(iterierbar):
    for i in iterierbar:
        yield i*i

quadrat = quadriere([10,11])
>>> quadrat
<generator object quadriere at 0x103945500>
>>> type(quadrat)
<type 'generator'>
>>> quadrat.next()
100
>>> sum(quadrat)
121
>>> sum(quadriere([10,11]))
221
```

Rechenzeit und Speicherverbrauch messen

Programm mit Generatorausdrücken → *43*

```
import nltk, timeit, time, os
words = nltk.corpus.brown.words()

def test_generator():
    return set(w.lower() for w in words)

# Initialisiere Timer-Objekt
tg = timeit.Timer(test_generator)

# Timing von Generatorausdruck
print 'Timed generator (seconds):', tg.timeit(1)
```

Der Speicherverbrauch muss extern gemessen werden.

Rechenzeit und Speicherverbrauch messen

Programm mit Listencomprehension → *44*

```
import nltk, timeit, time, os
words = nltk.corpus.brown.words()

def test_listcomprehension():
```

```
    return set([w.lower() for w in words])

# Initialisiere Timer-Objekt
tl = timeit.Timer(test_listcomprehension)

# Timing von Listenkomprehension
print 'Timed list comprehension (seconds):', tl.timeit(1)
```

Der Speicherverbrauch muss extern gemessen werden.

5.2.2 xrange

Effizienz in Rechenzeit

Zufällige Auswahl von Elementen aus einem Bereich → *45*

```
# Modul zur Zeitmessung von Python-Statements
import timeit
setup = 'import random'

# Konstruiere 2 Timer-Objekte
tr = timeit.Timer('random.sample( range(1000000),100)', setup)
tx = timeit.Timer('random.sample(xrange(1000000),100)', setup)

# Führe Timings je einmal durch und speichere Anzahl Sekunden
trsecs = tr.timeit(1)
txsecs = tx.timeit(1)

print "Aufgabe: Sample 100 Zahlen aus dem Bereich 0 bis 999999."
print "Zeit mit xrange:", txsecs, "Sekunden"
print "Zeit mit range:", trsecs, "Sekunden"
print "xrange ist etwa", trsecs/txsecs, "Mal schneller!"
```

Zufälliges Auswählen von Wörtern

Das Ziehen einer zufälligen Stichprobe (*sample*) aus einem Korpus. → 46

```
import nltk, random

corpus = nltk.corpus.nps_chat.words()

# for demonstration
for i in random.sample(xrange(len(corpus)),20):
    print corpus[i]
# as a reusable function with a generator return value
def sample_corpus1(text,size):
    return (text[i] for i in random.sample(xrange(len(text)),size))

# as a reusable function with a list return value
def sample_corpus2(text,size):
    return [text[i] for i in random.sample(xrange(len(text)),size)]
```

5.3 Ausnahmen

Häufige Exceptions

Ausnahmen können in jeder Stufe der Programmausführung auftreten!

SyntaxError
```
print 1 2
```

NameError
```
print a
```

ZeroDivisionError
```
1 / 0
```

IndexError
```
a = [1, 2, 3]
a[3]
```

KeyError
```
a = {}
a["test"]
```

RuntimeError
```
def x(): return x()
x()
```

TypeError
```
sum(["1", "2", "3"])
```

Wie gehe ich mit Fehlern um?

```
x = raw_input()
```

Robuste Programmierung

- Wir wollen `x` in eine Zahl umwandeln, bei ungültiger Eingabe eine neue Eingabe verlangen.
- `float(x)` führt zu Programmterminierung
- `x.isdigit()` akzeptiert nur Teilmenge aller Zahlen
- Verkettung von Regeln möglich, aber umständlich

Ausnahmen (Exceptions)

- Ausnahmen können im Programm abgefangen werden, anstatt dass sie zur Terminierung führen.
- Oft eleganter, als Ausnahmen zu vermeiden.

Ausnahmen auffangen: *try*-Konstrukt

Syntax-Schema
```
try:
   block1
except E:
   block2
```

Syntax-Schema mit finally

```
try:
   block1
except E:
   block2
finally:
   block3
```

Erklärung

- Führe `block1` aus.
- Wenn währenddessen eine Ausnahme vom Typ `E` auftritt, führe `block2` aus
- Führe `block3` auf jeden Fall am Schluss aus.

Ausnahmen sind ebenfalls Objekte und haben infolgedessen einen Typ!

Ausnahmen ignorieren → **47**

```
while True:
  x = raw_input('Please type in a number: ')
  try:
    float(x)
    break
  except ValueError:
    pass
```

Leere Blöcke

- Blöcke müssen immer mindestens eine Anweisung enthalten
- `pass` für leere Blöcke (*no operation*, no(o)p)

Philosophien der Fehlerbehandlung: LBYL vs EAFP

LBYL

```
if w in freqs:
  freqs[w] += 1
else:
  freqs[w] = 1
```

Look **b**efore **y**ou **l**eap.

EAFP

```
try:
  freqs[w] += 1
except KeyError:
  freqs[w] = 1
```

It's **e**asier to **a**sk for **f**orgiveness than for **p**ermission.

Fehler auffangen: Wie spezifisch?

```
try:
  ...
  (ganz viel Code)
  ...
except:
  pass
```

Welche Exceptions soll man abfangen?

- Zu allgemeine except-Klauseln erschweren das Bemerken und Finden von Programmierfehlern.
- Setze try/except-Klauseln gezielt ein.
- Bestimme den Ausnahmentyp, der abgefangen werden soll.

with-Konstrukt für Datei-Handling → 48

- Das Betriebssystem erlaubt nicht, dass Hunderte von Dateien von einem Prozess geöffnet sind.
- Bei Prozessen, welche vielen Dateien lesen/schreiben, müssen die Dateien geschlossen werden.
- Das with-Konstrukt mit Datei-Objekten macht dies automatisch (was auch immer für Ausnahmesituationen beim Dateiverarbeiten entstehen).

```
filename = "with_open.py"

with open(filename,'r') as f:
  for l in f:
    if l.rstrip() != '':
      sys.stdout.write(l)
```

5.4 Vertiefung

- Pflichtlektüre: Kapitel 3.1. bis und mit 3.6 aus dem NLTK-Buch

Kapitel 6

NLTK-Buch Kapitel 3: Korpora

Lernziele

NLTK

- Typische NLP-Pipeline zur Vorverarbeitung programmieren
- Laden von lokalen Rohtext-Korpora im UTF-8-Format
- Laden von Dateien von Web-Adressen
- Säubern von HTML
- Tokenisierung als Satz- und Wortsegmentierung
- Vokabulardifferenzen berechnen

6.1 NLP-Pipeline

6.1.1 urllib

Download und Verarbeitung von Web-Texten

Zuerst Download, dann lokale Verarbeitung

- Download und lokale Speicherung *interaktiv* mit Browser
- Programmierter Download in der Shell mit (`wget` oder `curl`)
- Einfache Optionen ergeben komplexe Funktionalität:
- Download von Verzeichnissen oder mit URL-Wildcards
- Konfigurierbarer Ressourcenverbrauch: Pause zwischen Anfragen, Anzahl Versuche bei Fehlschlägen, Netzwerkauslastung (Daten pro Sekunde)
- Achtung: Nicht alle Sites wollen/erlauben automatisierten Download (`http://SITE/robots.txt` gibt Auskunft).

Download auf der Kommandozeile: `wget`

```
$ wget http://www.gutenberg.org/files/2554/2554.txt
$ head 2554.txt
```

Rohtext-Korpora im UTF-8-Format einlesen

Einlesen von Rohtext-Korpora als `unicode`*-Strings* → *49*

```
import nltk
from nltk.corpus import PlaintextCorpusReader

corp_dir = ''
filename = '2554.txt'
crp = PlaintextCorpusReader(corp_dir, filename, encoding='utf-8')
words = crp.words()

print words[9000:10000]

# Wo ist das Korpus gespeichert?
print crp.abspath(filename)
```

6.1.2 HTML-Pipeline

Download und Verarbeitung von Web-Texten

Download und Verarbeitung in Python

Die Funktion `urlopen()` vom Modul `urllib` erlaubt entfernte Dateien (*remote files*) wie lokale Dateien zu öffnen.

Einlesen einer entfernten Textdatei mit `urlopen()` → *50*

```
import nltk
from urllib import urlopen

url = "http://www.gutenberg.org/files/2554/2554.txt"

# Lese URL-Inhalt ein und dekodiere das UTF-8
raw = urlopen(url).read().decode('utf-8')
# Textverarbeitung
tokens = nltk.wordpunct_tokenize(raw)
text = nltk.Text(tokens)
text.collocations(num=40)
```

Spoofing des Browsers[1] mit `urllib2`

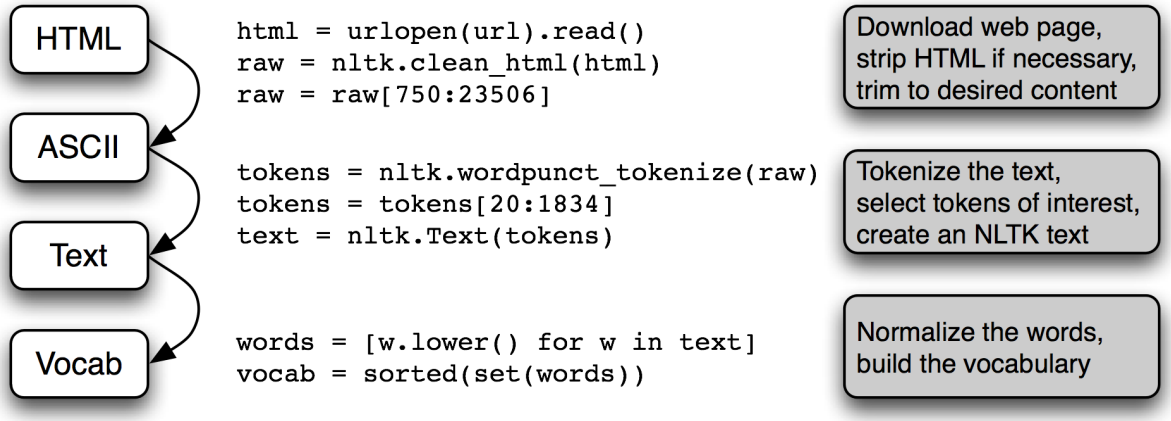
: Manchmal lassen Websites keinen automatisierten Download zu!

Einlesen einer entfernten Textdatei mit `urllib2.urlopen()` → *51*

```
import urllib2

user_agent = 'Mozilla/5.0 (Macintosh; Intel Mac OS X 10.7; rv:12.0) Gecko/20100101 Firefox/12.0'
url = "http://www.gutenberg.org/files/2554/2554.txt"

# Request Objekt erzeugen und konfigurieren
request = urllib2.Request(url)
request.add_header('User-Agent',user_agent)

# URL einlesen
raw = urllib2.urlopen(request).read().decode('utf-8')
```

[1] http://whatsmyuseragent.com

Typische Pipeline der Vorverarbeitung von HTML-Dateien



Schritt 1: Download von HTML als Python-String

Web-Page downloaden (Link) → 52

```
import nltk, re
from urllib import urlopen
### (1) Download Html-Datei
url = "http://news.bbc.co.uk/2/hi/health/2284783.stm"
html = urlopen(url).read().decode('utf-8')


# Was ist da drin?
html[:60]
# print html
```

HTML-Markup entfernen: Quick-And-Dirty-Methode

Hilfsfunktion `nltk.clean_html()` in NLTK 2 (in NLTK 3 entfernt)
Robustes Entfernen von allem, was nach HTML-Struktur und Kommentaren aussieht mit Hilfe von `re.sub()`.

Definition von `clean_html()`

```
def nltk_clean_html(html):
    # First we remove inline JavaScript/CSS:
    cleaned = re.sub(r"(?is)<(script|style).*?>.*?(</\1>)", "", html.strip())
    # Then we remove html comments. This has to be done before removing regular
    # tags since comments can contain '>' characters.
    cleaned = re.sub(r"(?s)<!--(.*?)-->[\n]?", "", cleaned)
    # Next we can remove the remaining tags:
    cleaned = re.sub(r"(?s)<.*?>", " ", cleaned)
    # Finally, we deal with whitespace
    cleaned = re.sub(r" ", " ", cleaned)
    cleaned = re.sub(r" ", " ", cleaned)
    cleaned = re.sub(r" ", " ", cleaned)
    return cleaned.strip()
```

Quick-And-Dirty-Regex-Techniken für (X)HTML

Typische Regex-Techniken beim Quick-And-Dirty-Säubern von HTML

```
re.sub(r"(?is)<(script|style).*?>.*?(</\1>)", "", string) re.sub(r"(?s)<!--.*?-->", "", string)
re.sub(r"(?s)<.*?>", "", string)
```

- Flag `(?i)` für case-insensitive Matching
- Flag `(?s)`, damit Metazeichen `.` auch Zeilenwechsel (`[\n\r]`) matcht
- Nicht-gieriges Matching `.*?`: Lese so wenig Zeichen wie möglich!
- Rückreferenz im regulären Ausdruck: `\1`: Paarige Konstrukte finden!

Sauberes HTML-Säubern, d.h. Verwandlung in Rohtext

- Obige Methode versagt bei HTML-Entitäten wie `&`.
- Besser: Bibliothek wie lxml.de oder *Beautiful Soup* → 53 (siehe Beispiel im Online-NLTK-Version): `$ sudo pip install BeautifulSoup`

Textzoning: Artikeltext extrahieren

- Manuell den *Rohtextstring* trimmen: `raw = raw[750:2306]` Allenfalls mit Unterstützung von `str.find()` bzw. `str.rfind()` zum Bestimmen der Offsets.
- Manuell die *Tokens* trimmen: `tokens = tokens[96:399]`
- Problem: Muss händisch für jedes Dokument gemacht werden.
- Alternativ: *Vor* dem Säubern des HTMLs den für eine Website typische Begrenzer des Artikels suchen. Nur den Text des Artikels verwenden:

```
art_html = \
   re.sub(r'(?s).*class="bodytext">(.+?)<div class="rhshead".*',
          r'\1',html)
```

- Vorteil: Funktioniert automatisch pro analysierter Website

Satz- und Wortsegmentierung für Englisch

Ziel: Saubere Satz- und Wortsegmentierung im Penn-Treebank-Stil

1. Sätze segmentieren: Liste von Satzstrings
2. Wörter segmentieren: Liste von Wortstrings

Listenkomprehension mit 2 Schlaufen

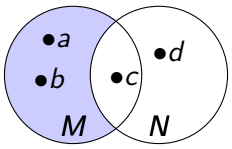
```
art_sents = nltk.sent_tokenize(art_raw)
art_tokens = [w
          for s in art_sents
          for w in nltk.word_tokenize(s)]
```

`nltk.word_tokenize()` ist für die Verarbeitung eines einzelnen Satzes gedacht! `nltk.wordpunct_tokenize()` trennt stärker auf bei Symbolzeichen als die Penn-Treebank-Tokenisierung! [2]

[2] Vgl. http://text-processing.com/demo/tokenize

Vocabular-Differenzen berechnen

Um die Tokenisierungsqualität zu beurteilen, ist die Mengen-Differenz geeignet. In Python `Diff = set(M).difference(N)`



$$Diff = M - N$$

```
# Welche Tokens aus t_word_tokenize sind nicht in art_tokens?
t_word_tokenize = nltk.word_tokenize(art_raw)
set(t_word_tokenize).difference(art_tokens)

# Welche Tokens aus r_wordpunct_tokenize sind nicht in art_tokens?
r_wordpunct_tokenize = nltk.wordpunct_tokenize(art_raw)
set(r_wordpunct_tokenize).difference(art_tokens)
```

Tokenisat weiterverarbeiten

Z.B. Tokenliste zu nltk.Text machen und damit experimentieren

```
art_text = nltk.Text(art_tokens)
art_text.findall('<blond.*><.*>')
art_text.collocations()
```

Warnung vor `nltk.Text.findall`

- : `re.findall` und `nltk.Text.findall` sind *nicht* dasselbe
- Nur für interaktive Benutzung gedacht, Resultate werden nur rausgeschrieben und stehen nicht als Datenstruktur zur Verfügung.

Typische weiterführende NLP-Pipeline-Verarbeitungsstufen

- POS-Tagging, Lemmatisierung
- Named-Entity-Recognition
- Chunking, Parsing

Anhang A

Liste der verlinkten Beispielprogramme und Ressourcen

→ 1 Online-Dokumentation: http://docs.python.org/library/re.html?#re.split 5
→ 2 http://en.wikipedia.org/wiki/Read-eval-print_loop 6
→ 3 Programm: http://www.cl.uzh.ch/siclemat/lehre/hs14/pcl1/lst/vorspann/str_literals.py 9
→ 4 Programm: http://www.cl.uzh.ch/siclemat/lehre/hs14/pcl1/lst/vorspann/unicode_literals.py 10
→ 5 Programm: http://www.cl.uzh.ch/siclemat/lehre/hs14/pcl1/lst/vorspann/str_representation_utf8.py . 10
→ 6 Programm: http://www.cl.uzh.ch/siclemat/lehre/hs14/pcl1/lst/vorspann/str_representation_l1.py 10
→ 7 Programm: http://www.cl.uzh.ch/siclemat/lehre/hs14/pcl1/lst/vorspann/codecs_open_files.py 11
→ 8 Programm: http://www.cl.uzh.ch/siclemat/lehre/hs14/pcl1/lst/vorspann/re_sub.py 12
→ 9 Programm: http://www.cl.uzh.ch/siclemat/lehre/hs14/pcl1/lst/vorspann/re_findall_flag_u.py 12
→ 10 Programm: http://www.cl.uzh.ch/siclemat/lehre/hs14/pcl1/lst/vorspann/re_findall_flag_u.py 13
→ 11 Programm: http://www.cl.uzh.ch/siclemat/lehre/hs14/pcl1/lst/vorspann/re_findall_tokenizer.py 13
→ 12 Programm: http://www.cl.uzh.ch/siclemat/lehre/hs14/pcl1/lst/nltk1/import_nltk_book.py 18
→ 13 Programm: http://www.cl.uzh.ch/siclemat/lehre/hs14/pcl1/lst/nltk1/import_from_nltk_book.py . . . 18
→ 14 Programm: http://www.cl.uzh.ch/siclemat/lehre/hs14/pcl1/lst/nltk1/list_comprehension_if.py 20
→ 15 Programm: http://www.cl.uzh.ch/siclemat/lehre/hs14/pcl1/lst/nltk1/def_foo.py 20
→ 16 Programm: http://www.cl.uzh.ch/siclemat/lehre/hs14/pcl1/lst/nltk1/return_statement.py 20
→ 17 Programm: http://www.cl.uzh.ch/siclemat/lehre/hs14/pcl1/lst/nltk1/def_block.py 22
→ 18 Programm: http://www.cl.uzh.ch/siclemat/lehre/hs14/pcl1/lst/nltk1/globals_and_locals.py 22
→ 19 Programm: http://www.cl.uzh.ch/siclemat/lehre/hs14/pcl1/lst/nltk1/locals.py 23
→ 20 Programm: http://www.cl.uzh.ch/siclemat/lehre/hs14/pcl1/lst/nltk2/nltk_corpus_gutenberg_austen.py 25
→ 21 Programm: http://www.cl.uzh.ch/siclemat/lehre/hs14/pcl1/lst/nltk2/nltk_corpus_gutenberg_brown.py 25
→ 22 Programm: http://www.cl.uzh.ch/siclemat/lehre/hs14/pcl1/lst/nltk2/freqdist_emma.py 27
→ 23 Programm: http://www.cl.uzh.ch/siclemat/lehre/hs14/pcl1/lst/nltk2/CondFreqDist_brown.py 28
→ 24 Doku: http://nltk.org/api/nltk.html#nltk.text.Text 32
→ 25 Dokumentation zu Statements: http://docs.python.org/reference/simple_stmts.html 32
→ 26 Programm: http://www.cl.uzh.ch/siclemat/lehre/hs14/pcl1/lst/nltk2/statement_vs_expression.py . . . 32
→ 27 Dokumentation zu Expressions: http://docs.python.org/reference/simple_stmts.html 32
→ 28 Programm: http://www.cl.uzh.ch/siclemat/lehre/hs14/pcl1/lst/nltk2/functions_statement_vs_expression.py 33
→ 29 Programm: http://www.cl.uzh.ch/siclemat/lehre/hs14/pcl1/lst/nltk2/other_comprehensions.py 33
→ 30 Programm: http://www.cl.uzh.ch/siclemat/lehre/hs14/pcl1/lst/nltk2lex/nltk_corpus_stopwords_english.py 35

→ 31 Programm: http://www.cl.uzh.ch/siclemat/lehre/hs14/pcl1/lst/nltk2lex/foo_fraction_en.py 36
→ 32 Programm: http://www.cl.uzh.ch/siclemat/lehre/hs14/pcl1/lst/nltk2lex/foo_fraction_en.py 36
→ 33 Programm: http://www.cl.uzh.ch/siclemat/lehre/hs14/pcl1/lst/nltk2lex/nltk_corpus_cmudict.py 37
→ 34 Programm: http://www.cl.uzh.ch/siclemat/lehre/hs14/pcl1/lst/nltk2lex/nltk_wordnet.py 38
→ 35 Gutes Tutorat: http://www.python-kurs.eu/klassen.php 39
→ 36 FreqDist-Definition: http://nltk.org/_modules/nltk/probability.html#FreqDist 39
→ 37 Programm: http://tinyurl.com/pcl-1-hs14-classdef 40
→ 38 Programm: http://www.cl.uzh.ch/siclemat/lehre/hs14/pcl1/lst/nltk3/round_floats.py 43
→ 39 Runden von floats: http://en.wikipedia.org/wiki/Rounding#Round_half_away_from_zero 43
→ 40 Programm: http://www.cl.uzh.ch/siclemat/lehre/hs14/pcl1/lst/nltk3/stemmed_kwic.py 44
→ 41 Programm: http://www.cl.uzh.ch/siclemat/lehre/hs14/pcl1/lst/generator/generators_next.py 45
→ 42 Programm: http://tinyurl.com/pcl1-hs14-generator-yield 46
→ 43 Programm: http://www.cl.uzh.ch/siclemat/lehre/hs14/pcl1/lst/generator/timeit_generator.py 46
→ 44 Programm: http://www.cl.uzh.ch/siclemat/lehre/hs14/pcl1/lst/generator/timeit_listcomprehension.py 46
→ 45 Programm: http://www.cl.uzh.ch/siclemat/lehre/hs14/pcl1/lst/generator/random_sample_xrange_timeit.py 47
→ 46 Programm: http://www.cl.uzh.ch/siclemat/lehre/hs14/pcl1/lst/generator/random_sample_xrange.py 47
→ 47 Programm: http://www.cl.uzh.ch/siclemat/lehre/hs14/pcl1/lst/nltk3/float_raw_input.py 49
→ 48 Programm: http://www.cl.uzh.ch/siclemat/lehre/hs14/pcl1/lst/nltk3/with_open.py 50
→ 49 Programm: http://www.cl.uzh.ch/siclemat/lehre/hs14/pcl1/lst/nltk3raw/nltk_plaintextcorpusreader.py 52
→ 50 Programm: http://www.cl.uzh.ch/siclemat/lehre/hs14/pcl1/lst/lst/nltk3raw/urllib_gutenberg.py 52
→ 51 Programm: http://www.cl.uzh.ch/siclemat/lehre/hs14/pcl1/lst/lst/nltk3raw/urllib2_gutenberg.py ... 52
→ 52 Programm: http://www.cl.uzh.ch/siclemat/lehre/hs14/pcl1/lst/nltk3raw/nltk_clean_html.py 53
→ 53 Paket für HTML-Verarbeiten: http://www.crummy.com/software/BeautifulSoup 54

Literaturverzeichnis

[Baker et al. 2006] Baker, Paul, A. Hardie und T. McEnery (2006). *A glossary of corpus linguistics*. Edinburgh University Press, Edinburgh.

[Bird et al. 2009] Bird, Steven, E. Klein und E. Loper (2009). *Natural Language Processing with Python*. O'Reilly.

[Perkins 2010] Perkins, Jacob (2010). *Python Text Processing with NLTK 2.0 Cookbook*. Packt Publishing.

Index

`(?:)`, 12
`# -*- coding: iso-8859-1 -*-`, 11
`# -*- coding: utf-8 -*-`, 10
`__contains__()`, 30
`__getitem__()`, 30
`__getslice__()`, 30
`__init__`, 40

Anweisung, 32
ASCII, 7
Attribut, 6
Ausdruck, 56
Ausnahmen, 47

Bytekode, 18

`chr`, 9
`codecs.open()`, 11

Datenkapselung, 39
Datentyp, 5
Download-Techniken, 51
Dynamische Bindung, 39

Expression, 56
    If-Else-Ausdruck, 33
    Lambda-Ausdruck, 33

Formatierungsausdruck, 43
    *, 44
    %%, 43
    %d, 43
    %f, 43
    %s, 43
Funktion, 21
    Definition, 20
    Parameter, 20
    Rückgabewert, 20
    Return-Statement, 20

Generatorausdruck, 45
`globals()`, 22

Häufigkeitsverteilung, bedingt, 28
Häufigkeitsverteilung, bivariat, 28
Häufigkeitsverteilung, univariat, 27

`import`, 18
Instanz, 31
`isinstance`, 9
ISO-8859-1, 7
`issubclass()`, 39
Iteration, 21

Klassen, 38
Konstruktor, 31
Korpus, 24
    Python-Repräsentation, 25
Korpus, balanciert, 26
Korpus, opportunistisch, 26

Lambda-Ausdruck, 33
Latin-1, 7
Listenkomprehension, 45
    Bedingungen, 20
    einfach, 19
`locals()`, 22

Methode, 6
    öffentlich, 44
    privat, 44
Methodenaufruf, 32
Modul, 18

Namensraum, 22
`next()`, 46
NLP, 15
`nltk.ConditionalFreqDist`, 29
`nltk.corpus.brown`, 25
`nltk.corpus.gutenberg`, 25
`nltk.corpus.stopwords`, 35
`nltk.download()`, 25
`nltk.FreqDist`, 27
`nltk.Text.findall()`, 55
`NoneType`, 5

`object`, 38

Objekt, 5
Objektinstantiierung, 40
Objektkonstruktor-Funktion, 40
Objektorientierung, 5, 39
`ord`, 9

Package, 18
`PlaintextCorpusReader`, 52
Punktnotation, 18

`r"`, 9
`random.sample()`, 47
`re.findall()`, 12, 13
`re.sub()`, 12
Rechenzeit, 47
Regex-Flag
    `(?u)`, 13
    `(?x)`, 13
`return`, 20
`round()`, 43

Selbst-Parameter, 39
Sequenz, 29
`set`, 20
`set().difference()`, 55
Skopus, 22
Statement, 32
`str`, 9
`str.find`, 54
`str.rfind`, 54
String-Literal, 9, 10

`timeit`, 47
Tokenisierung
    `nltk.word_tokenize()`, 54
    `nltk.wordpunct_tokenize()`, 54
    Satzsegmentierung, 54
    Wortsegmentierung, 54
try-Konstrukt, 48
Tupel, 29
`type`, 5

`u"`, 10
`unichr`, 10
`unicode`, 10
`ur"`, 10
`urllib`, 52
`urllib2`, 52
`urlopen()`, 52
UTF, 8

Vererbung, 39

with-Konstrukt, 50
Wortlisten, 35