

NLTK-Buch Kapitel 2

Simon Clematide

`simon.clematide@uzh.ch`

Institut für Computerlinguistik
Universität Zürich

Programmiertechniken in die Computerlinguistik I

Übersicht

Korpora

- Korpora einlesen

- Korpus-Typen

Häufigkeitsverteilungen

- Univariat

- Bivariat

Technisches

- Sequenzen

- Klassen & Objekte

- Statements und Expressions

Lernziele

NLTK

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

Technisches

- ▶ Listenkomprehension mit Tupeln
- ▶ Klassen/Typen und Objekte
- ▶ Konstruktoren
- ▶ Anweisungen und Ausdrücke:
Lambda-Ausdrücke und Komprehensions-Ausdrücke und ihre analogen Anweisungen

Gutenberg-Projekte: Elektronische Edition älterer Texte

Definition (Korpus (sächlich, im Plural Korpora))

Ein Korpus ist eine Sammlung von Texten.

Sammlung vorwiegend englischsprachiger Bücher



Sammlung von über 50'000 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>



Zugriff auf Korpora¹

Das Package `nltk.corpus`

Enthält Packages und Module, mit denen Korpora in verschiedensten Formaten eingelesen werden können.

Das Objekt `nltk.corpus.gutenberg`

Stellt eine Auswahl von 18 englischsprachigen Gutenberg-Texten (ASCII) als Teil der NLTK-Korpusdaten zum Einlesen zur Verfügung.

¹<http://www.nltk.org/api/nltk.corpus.html>

Funktionen des Objekts `nltk.corpus.gutenberg`

Repräsentationen für reine Text-Korpora (ASCII oder iso-latin-1) ▶1

```
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 Wörtern)
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



```
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()
```

²http://en.wikipedia.org/wiki/Brown_Corpus

Arten von Korpora: Korpus-Typologie

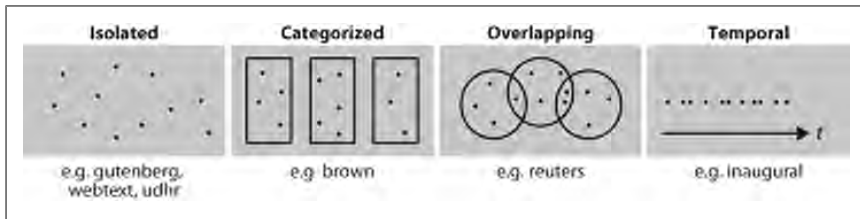


Figure 2-3. Common structures for text corpora: The simplest kind of corpus is a collection of isolated texts with no particular organization; some corpora are structured into categories, such as genre (Brown Corpus); some categorizations overlap, such as topic categories (Reuters Corpus); other corpora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

Erbsenzählerei: Häufigkeiten ermitteln

Word Tally

the	
been	
message	
persevere	
nation	

- ▶ 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 eine 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>)

Methoden der Objekte der Klasse `nltk.FreqDist`

<http://www.nltk.org/api/nltk.html#nltk.probability.FreqDist>

Anwendung der Klasse nltk.FreqDist

Berechnen der häufigsten längsten Wörter



```
import nltk
from nltk.corpus import gutenbergl

emma_words = gutenbergl.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

w1 = sorted([w for w in emma_fd.keys()
              if len(w)>10 and emma_fd[w]> 7])
```

Bivariate (bedingte) Häufigkeitsverteilungen

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

►4

```
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)
```

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

Sequenz-Datentypen: list, str, unicode, tuple

Definition (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

Type	Mutable	Länge		
		0	1	2
list	Ja	<code>[]</code>	<code>[1]</code>	<code>[1, 'n']</code>
str	Nein	<code>''</code>	<code>'1'</code>	<code>'ab'</code>
tuple	Nein	<code>()</code>	<code>(1,)</code>	<code>(1, 'n')</code>



Einertupel braucht Komma! Die runden Klammern sind meist weglassbar.

Listen: Veränderliche (*mutable*) Sequenzen ►

Typische Modifikationen für Listen

```
l = []                # Zuweisung ist keine Listenmodifikation!  
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 Listen, d.h. veränderlichen Sequenzen, können Elemente (oder Abschnitte (*slices*)) gelöscht, ersetzt oder ergänzt werden.
- 💰 : Listen-Methoden, welche in-place-Modifikationen durchführen, liefern als Rückgabewert *None* zurück und nicht die Liste!

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 unveränderliche Sequenz ausreicht.

Wozu braucht's Listen?

- ▶ Speicher-effiziente Modifikation von **Elementen** der Sequenz: Einfügen, Löschen, Ersetzen.
- ▶ Für *In-Place*-Sortieren via `my_list.sort()`. Im Gegensatz zur Funktion `sorted(Liste)`, 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*)

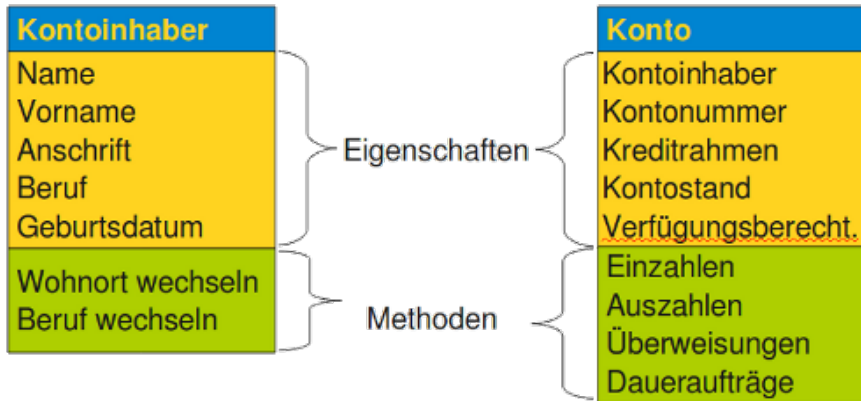
```
>>> "ABBA"[1:3]
'BB'
>>> "ABBA".__getslice__(1,3)
'BB'
```

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]
```

Objektorientierte Modellierung



Quelle: <http://www.python-kurs.eu/klassen.php>

Mittels **Klassen** können **eigene (Daten-)Typen** geschaffen werden.

Typen/Klassen und Objekte

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'
list	[1,2,3]

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

```
>>> unicode()  
u''  
>>> 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' }
```

Objekte konstruieren mittels Klassen/Typ-Konstruktor

Konstrukturen (*constructors*) vs Literale (*literals*)

- ▶  Nur die Objekte der wichtigsten eingebauten Datentypen können als Literale mit Spezialnotation konstruiert werden.
- ▶ Konstrukturen der Form `TYPE()` sind immer möglich!

Typ/Klasse

`nltk.probability.FreqDist`

`collections.Counter`

Objekt/Instanz herstellen

`nltk.FreqDist("abrakadabra")`

`collections.Counter('abrakadabra')`

Hinweis: Die Klasse `nltk.probability.FreqDist` erweitert die eingebaute Python-Klasse `collections.Counter` ▶!

Klassenbezeichner sind wichtig! Keine Objekte ohne Typen/Klassen!

Ein Blick hinter die Kulisse: Methodenaufrufe

Schein: Objekt ruft seine Methode auf

`OBJECT.METHOD(ARG1)`

Sein: Klasse/Type ruft Methode mit Objekt als 1. Argument auf

`CLASS.METHOD(OBJECT, ARG1)`

Schein

Sein

`"A Test".lower()`

`str.lower("A Test")`

`"ABBA".count('A')`

`str.count("ABBA", 'A')`

Das macht VIEL Sinn!

Die Addition definiert man auch auf der Klasse der Ganzzahlen, nicht für jede Zahl einzeln!

Objekte rufen Methoden auf, welche auf Klassenebene definiert sind!

Unterschied zwischen *Statements* und *Expressions*

Anweisungen (*statements*)

►5

werden vom Python-Interpreter ausgeführt und evaluieren zu **keinem Wert**.

```
print Statement
```

►6

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

Ausdrücke (*expressions*)

►7

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: Listenkompensation

```
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

►8

```
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

►9

```
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?

Vertiefung

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

Verständnisfragen

- ▶ Wie kann man Textkorpora als Datenstruktur repräsentieren?
- ▶ Welche Arten von Sequenztypen sind in Python eingebaut?
- ▶ Welche Methoden muss ein Objekt können, damit die typischen Spezialnotationen für Sequenzen verwendet werden kann?
- ▶ Was unterscheidet univariate und bivariate Häufigkeitsverteilungen?
- ▶ Inwiefern hängen Typen/Klassen und Konstruktor-Funktionen zusammen?
- ▶ Was passiert eigentlich, wenn ein Objekt eine seiner Methoden aufruft?
- ▶ Können Ausdrücke Statements enthalten?

Liste der verlinkten Programme und Ressourcen I

Folie

►1 Programm: http://www.cl.uzh.ch/siclemat/lehre/hs15/pcl1/lst/nltk2/nltk_corpus_gutenberg_austen.py	6
►2 Programm: http://www.cl.uzh.ch/siclemat/lehre/hs15/pcl1/lst/nltk2/nltk_corpus_gutenberg_brown.py	7
►3 Programm: http://www.cl.uzh.ch/siclemat/lehre/hs15/pcl1/lst/nltk2/freqdist_emma.py	12
►4 Programm: http://www.cl.uzh.ch/siclemat/lehre/hs15/pcl1/lst/nltk2/CondFreqDist_brown.py	14
►5 Dokumentation zu Statements: http://docs.python.org/reference/simple_stmts.html	25
►6 Programm: http://www.cl.uzh.ch/siclemat/lehre/hs15/pcl1/lst/nltk2/statement_vs_expression.py	25
►7 Dokumentation zu Expressions: http://docs.python.org/reference/simple_stmts.html	25
►8 Programm: http://www.cl.uzh.ch/siclemat/lehre/hs15/pcl1/lst/nltk2/functions_statement_vs_expression.py	28
►9 Programm: http://www.cl.uzh.ch/siclemat/lehre/hs15/pcl1/lst/nltk2/other_comprehensions.py	29

Literaturangaben I

- BIRD, STEVEN, E. KLEIN und E. LOPER (2009).
Natural Language Processing with Python. O'Reilly.