Seminarprojekt *Tagging*

Teil 1: Taggen einzelner Tokens

Grundlagen für diesen Teil: Kapitel 2 und 3 im NLTK-Book, die Texte über Tokenisierung, *List Comprehension* und Reguläre Ausdrücke sowie Übung 1.

Beschreibung der Aufgabe:

In dieser Aufgabe geht es darum, einen vorgegebenen englischen Text mit verschiedenen Taggern zu taggen, die mit Ausnahme des Regex-Taggers auf der Basis eines Ausschnitts des Brown Corpus trainiert wurden. Es soll dabei folgendes ermittelt / erkundet werden:

- wie verhalten sich die Tagger (1) einzeln und (2) in Kombination bei bestimmten Problemfällen, z.B. ambig kategorisierten Tokens und Tokens, die nicht im Trainingscorpus vorgekommen sind?
- wie wird ein Regular Expression Tagger (Regex-Tagger, in Python allerdings RegexpTagger) aufgebaut?

Besonderheit der Aufgabe: wir wollen für diese Aufgabe nicht das ursprüngliche Tagset des Brown Copus verwenden (dessen Kategorisierung für das menschliche Auge und unseren Bedarf zu umständlich ist), sondern stattdessen mit einem Tagset aus nur 36 Kategorien arbeiten, die den uns bekannten Klassen *det*, *noun* und *verb* etc. entsprechen.

Vorarbeit

Legen Sie eine Datei namens **`tagging.py`** an. Alle weiteren Schritte werden in dieser Datei formuliert.

Schritt (A): Erzeugen der Datengrundlage

Importieren Sie das **`nltk`**.
Importieren Sie das Modul **`brown`** aus **`nltk. corpus`**
Weisen Sie der Variable **`text`** den folgenden Text zu:

```
the dog barked.
the boy sang beautifully.
the likelihood for that result is unexpectedly low.
John knows that she extracted the information carefully.
the neighbourhood evolved slowly.
Mary thinks that Bill took her car.
the guy from the garage said that the car is unrepairable.
that book is abolutely readable.
the students guessed that the task involves tagging.
```

Schritt (B): Das Brown Corpus mit angenehmen Tags ausstatten

In diesem Schritt geht es darum, aus einem Input wie
`[('Scotty', 'NP'), ('did', 'DOD'), ('not', '*'), ('go', 'VB'), ('back', 'RB'), ('to', 'IN'), ('school', 'NN'), ('.', '.')]`
einen Output wie
`[('Scotty', 'noun'), ('did', 'verb'), ('not', '*'), ('go', 'verb'), ('back', 'adv'), ('to', 'prep'), ('school', 'noun'), ('.', '.')]`
zu erzeugen, bei dem die Originaltags des Corpus durch benutzerfreundliche Tags substituiert wurden. Um diese Aufgabe zu bewerkstelligen, sind drei Fragen zu beantworten:

a. (theoretisch) wie soll das neue Tagset aussehen, und welche Tags aus dem Brown Corpus werden auf welche neuen Tags abgebildet?

b. (praktisch) in welcher Form muss die Information vorliegen, die besagt, welches Brown Tag auf welches neue Tag abgebildet ist?

c. (praktisch) wie muss eine Funktion aussehen, die den Vorgang für einen mit Brown Tags getaggten Datensatz automatisiert?

zu (a): bei einer 'echten' Analyse müssten Sie sich zunächst – sozusagen auf dem Papier – überlegen, auf welche Weise das Brown Tagset verschlichtet werden kann. Dieses ist keine triviale Angelegenheit, sondern erfordert einiges an linguistischen Grundkenntnissen. Diese Aufgabe ist zu zeitaufwändig für die praktische Übung. Wir verwenden daher das folgende, vereinfachte Verfahren:

1. Bildung von 'Großklassen' im Brown Tagset:

Wenn Sie sich das Brown Tagset ansehen, stellen Sie fest, dass sich Klassen wie Adjektiv, Nomen, Pronomen etc. zahlreiche Subkategorien enthalten, die im wesentlichen auf den Flexionseigenschaften ihrer einzelnen Mitglieder basieren. Wenn wir von den Kategorienbezeichnern des Brown Corpus nur die ersten beiden Zeichen berücksichtigen, landen wir bei einer sehr viel kleineren Zahl von Tags, da z.B. BE, BED, BED*, BEDZ, BEDZ* usw. allesamt der Großklasse 'BE' angehörten; VB, VB+AT, VB+IN, VB+JJ, VB+PPO etc. allesamt der Großklasse 'VB' und so weiter.

2. Abbildung der Großklassen auf traditionelle Klassen:

Dieser Schritt ist einfach, da wir folgende Abbildungen vornehmen können:

Brown	Neu	Brown	Neu	Brown	Neu	Brown	Neu
AB	predet	EX	pro	NP	name	TO	inf
AT	det	HV	verb	NR	name	VB	verb
BE	cop	IN	prep	OD	ord	WD	det
CC	conj	JJ	adj	PN	pro	WP	pro
CD	card	MD	modal	PP	pro	WQ	adv
DO	verb	NN	noun	RB	adv	WR	adv
DT	det						

wobei predet = *predeterminer*, card = *cardinal number*, ord = *ordinal number* und cop = *copula*

Bei diesem Verfahren gibt es zwei Punkte, die geklärt werden müssen:

1. Die Klassifikation der Negation *not*. Diese ist im Brown Corpus als * klassifiziert. Somit entzieht sich dieses Tag dem Verfahren zur Großklassenbildung (das ja lautet "berücksichtige nur die ersten beiden Zeichen des Originaltags"), denn es besteht nur aus einem einzigen Zeichen. Wir werden deshalb dieses Tag, wie auch die Tags für die Satzzeichen, extra erfassen
2. Die Klasse PP$. Diese besteht aus abhängigen Possessivpronomina wie *his*, *her*, *my* usw., die in entsprechenden NP als Determinatoren auftreten. Wenn wir bei dem Originaltag nur die ersten beiden Zeichen berücksichtigen, erhalten wir PP – diese Klasse aber wird auf Pronomina abgebildet. Wir werden deshalb dieses Tag besonders berücksichtigen

zu (b): Hier geht es darum, die in der oa. Tabelle kodierte Information in Python aufzubereiten. Für die Abbildung einer Menge von Daten auf eine Zielmenge ist in Python das Dictionary die Datenstruktur der Wahl. Ein Dictionary besteht aus einer Liste von Einträgen der Form **`<key>:<value>`**, also <Schlüssel>: <Wert>. Im Gegensatz zu normalen Listen werden diese aber in { } eingeschlossen: **`{<key>:<value>}`**, z.B. wie **`{'boy':'noun', 'barks':'verb'}`**

⇨ für mehr Information zum Datentyp Dictionary siehe z.B Abschnitt 2.6.1. im NLTK-Buch (http://nltk.org/doc/en/programming.html)

Weisen Sie der Variablen **`tag_table`** ein Dictionary zu, das die in der Tabelle weiter oben aufgeführten Information kodiert, also **`tag_table = {'AB':'predet', 'AP':'det', 'AT':'det'…}`**

zu (c): Hier müssen wir uns fragen, wie wir den Vorgang der Tagsubstitution implementieren können. Dafür müssen wir zunächst einmal wissen, in welcher Datenform die getaggten Sätze aus dem Brown Corpus vorliegen.

Im ursprünglichen getaggten Brown Corpus haben die Sätze auf Dateiebene die folgende Form:

`Scotty/np did/dod not/* go/vb back/rb to/in school/nn ./.`
(zweiter Satz von Text *ck01* in .../nltk/data/corpora/brown/).

Die Tags sind also mit den jeweiligen Wörtern durch ein **'/'** getrennt verknüpft und werden mit Kleinbuchstaben geschrieben

Die Methode **`tagged_sents()`** im Modul **`nltk.corpus.brown`** liefert die Sätze jedoch in folgender Form:

```
>>> brown.tagged_sents(categories='k')[1]
[('Scotty', 'NP'), ('did', 'DOD'), ('not', '*'), ('go', 'VB'), ('back',
'RB'), ('to', 'IN'), ('school', 'NN'), ('.', '.')]
```

Das ist eine Liste aus Paaren (i.e. Tupel aus zwei Elementen) mit dem Tag als zweitem Element.

⇨ für mehr Information zum Datentyp Tupel siehe z.B. http://www.rg16.asn-wien.ac.at/~python/how2think/kap09.htm.

[1] Mit dem Parameter *categories* werden verschiedene Textkategorien identifiziert, die auch in den Dateinamen zu finden sind. Texte der Kategorie *k* haben den Dateinamen ck<Nr.>, z.B. *ck01*.

Die Abfrage mehrerer Sätze durch *Slicing* liefert dann eine Liste von Sätzen:

```
>>> brown.tagged_sents(categories='k')[0:2]
[[('Thirty-three', 'CD-HL')], [('Scotty', 'NP'), ('did', 'DOD'), ('not', '*'), ('go', 'VB'), ('back', 'RB'), ('to', 'IN'), ('school', 'NN'), ('.', '.')]]
```
[2]

Wir benötigen also ein Verfahren, die Brown-Form in unsere Form zu konvertieren.
Dabei gehen wir so vor, dass wir das Verfahren für einen einzelnen Tupel definieren und es dann dann auf alle Tupel in den Datensätzen anwenden.

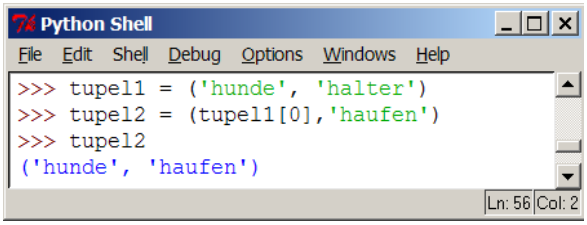
Die Verarbeitung eines einzelnen Tupels

Input `('Scotty','NP')`
Output `('Scotty','noun')`.

Die Elemente eines Tupels können wie andere Sequenzen durch Indexausdrücke referenziert werden. Es gilt: `('Scotty','NN')[0] == 'Scotty'` und `('Scotty','NN')[1]=='NN'`.

Da Tupel, ähnlich wie Zeichenketten, aber genau anders als Listen, unveränderlich sind, können ihren Elementen keine Werte zugewiesen werden, d.h. `('Scotty','NN')[1]='noun'` ist nicht zulässig. Das bedeutet *de facto*, dass wir nicht das ursprüngliche Tupel ändern, sondern auf seiner Grundlage eine neues Tupel erzeugen.

Beispiel: Erzeugen eines Tupels auf Grundlage eines anderen Tupels



Das Objekt **`tupel2`** ist so zu lesen: erstes Element dieses Tupels ist gleich dem ersten Element des **`tupel1`**, zweites Element ist die Zeichenkette 'haufen'.

Zurück zur Aufgabe. Uns liegt an dieser Stelle folgendes vor:

a) ein Tupel der Form ('WORT', 'LANGES_BROWNTAG') und

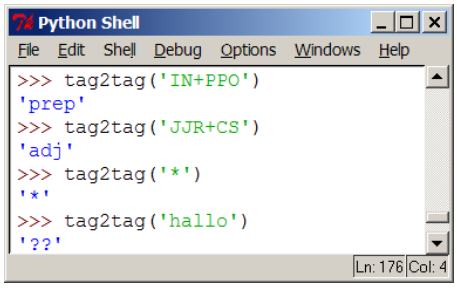
b) ein Dictionaryeintrag der Form {'KURZES_BROWNTAG': 'NEUESTAG'}

Um nun die Tag-Ersetzung vornehmen zu können, definieren wir eine Funktion **`tag2tag()`**, die als Input das Original-Browntag hat und als Output das wie oben in der Tabelle zugeordnete neue Tag zurückliefert.

Diese Funktion muss…

1. überprüfen, ob **`tag`** Element der Zeichenkette **`'()*,.:'`** ist, in welchem Fall das neue Tag gleich dem Originaltag ist (d.h. es wird der gleiche Wert zurückgegeben und die Prozedur ist beendet)
2. überprüfen, ob **`tag`** == PP$, in welchem Fall als neuer Wert **`det`** zurückgegeben wird und die Prozedur beendet ist
3. auf der Grundlage von **`tag`** ein neues Objekt **`temp`** erzeugen, in dem es von **`tag`** alles außer den ersten beiden Buchstaben abschneidet (das geht ganz einfach mit der Indexfunktion)
4. überprüfen, ob **`temp`** im Dictionary **`tag_table{}`** vorhanden ist und wenn
   a. ja: den **`temp`** im Dictionary **`tag_table{}`** zugeordneten Wert zurückgeben.
   b. nein: zwei Fragezeichen zurückgeben

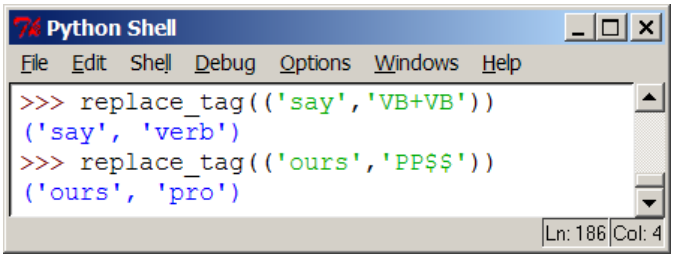
Schreiben Sie eine Funktion **`tag2tag(tag)`**, die die beschriebenen Schritte vollzieht.
Der Screenshot zeigt, was diese Funktion tut:



⇨ für mehr Information zu **`tag2tag()`** siehe den Appendix

[2] Wie man sehen kann, ist das erste Element in dieser Liste wohl so etwas wie die Kapitelüberschrift

Wir können die Funktion **`tag2tag()`** nun für unser Ziel verwenden, aus einem gegebenen Tupel ein neues Tupel zu erstellen. Dafür definieren wir die Funktion **`replace_tag(tupel)`**, die folgendes leistet:

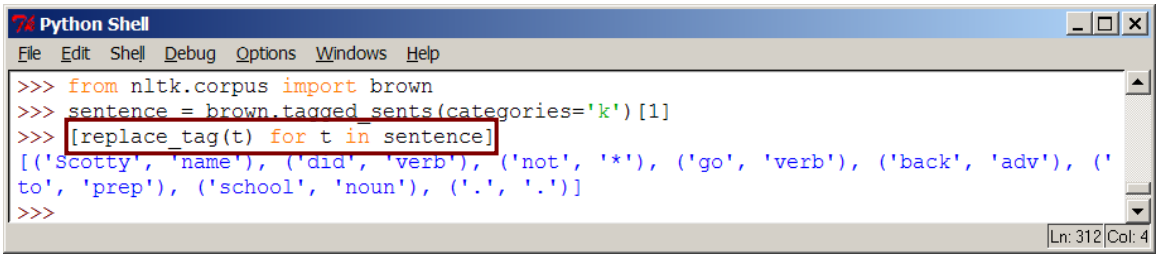


Schreiben Sie eine Funktion **`replace_tag(tupel)`**, die auf der Grundlage eines Tupels ein neues Tupel erzeugt, dessen erstes Element gleich dem ersten Element des Ursprungstupels ist und dessen zweites Element per **`tag2tag()`** ermittelt wird.

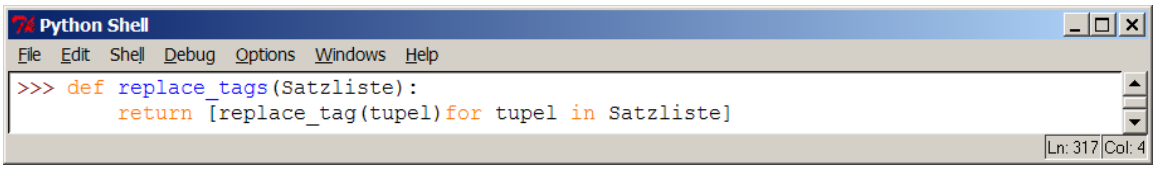
⇨ für mehr Information zu **`replace_tag()`** siehe den Appendix

Die Verarbeitung einer Tupelliste

Unter Verwendung der Funktion **`replace_tag(tupel)`** können wir diese Aufgabe leicht lösen. Wie oben bereits erwähnt, stehen die Daten, um die es uns geht, in Brown Corpus in Form von Listen zur Verfügung: ein Text ist eine Liste von Sätzen, die selber wiederum als Listen von Tupeln aufgebaut sind. Um nur einen einzelnen Satz zu behandeln, in dem jedes Tupel einzeln abgearbeitet wird, 'bauen' wir unter Zuhilfenahme der *List Comprehension* Methode (siehe dazu die Erklärungen auf unserer Webseite) eine neue Liste von Tupeln auf, indem wir einfach folgenden Ausdruck verwenden:

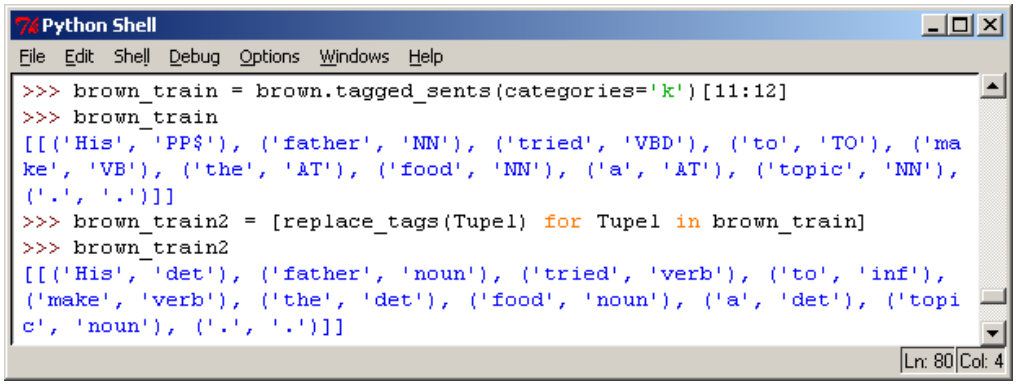


Um eine Satzliste, die Teil einer Liste von Satzlisten ist, zu verarbeiten, bietet es sich an, wieder eine eigene Funktion zu schreiben, die wir **`replace_tags(Liste)`** nennen:



Schreiben Sie die Funktion **`replace_tags(Satzliste)`**, die eine Satzliste konvertiert, d.h. darin die entsprechenden Ersetzungen vornimmt.

Jetzt haben wir alles beisammen, um aus einem Originaldatenausschnitt des Brown Corpus einen neuen Datensatz mit angenehmen Tags zu erzeugen. In Anlehnung an die Terminologie im NLTK-Buch und mit Bezug auf den weiteren Verwendungszweck nennen wir diese Objekte **`brown_train`** (Ausschnitt der Originaldaten) und **`brown_train2`** (Originaldatensatz mit guten Tags) nennen:



Prüfen Sie anhand verschiedener Datenausschnitte, ob alles funktioniert.

Schritt (C): Tagger-Training

Auf der Grundlage der Vorarbeit und Schritten A und B können wir jetzt eine Reihe von Taggern ausprobieren. Wir wollen im Einzelnen folgendes testen:

- einen Regex-Tagger
- einen mit **`brown_train2`** trainierten Unigrammtagger
- eine Kombination von beiden, die wir Unireg-Tagger nennen
- eine Kombination aus dem Unireg-Tagger und einem Bigrammtagger

Der Regex-Tagger.

Ein Regex-Tagger ist ein Tagger, der zum Einsatz kommen kann, wenn es darum geht, Wörter aufgrund bestimmter Wortmerkmale zu taggen. Ein Regex-Tagger wird nicht trainiert, stattdessen wird sich der Umstand zu Nutze gemacht, dass Wörter, insbesondere nach einem Derivationsprozess, wortartenspezifische Endungen aufweisen können. Die folgende Tabelle zeigt einige konkrete Beisipiele:

Affix	Informal description[3]	X → Y	Examples
-ism, -ity, -ness	property of being X	A → N	realism, sensitivity, kindness
-ance, -ment, -al	activity or result of Xing	V → N	deferance, engagement, refusal
-ful / -less	full of / lacking X	N → A	joyful, joyless
-able	able to be Xed	V → A	breakable, readable
-en	(cause to) become (more) X	A → V	redden, loosen, tighten
de-	remove X from	N → V	debug, delouse
dis-, de-, un-	not X or reverse X	V → V	disagree, decompose, unlock
-let, -ette, -ie	small X	N → N	piglet, cigarette, girlie
un-	not X	A → A	untidy, unsound, unsafe

(S. Hackmack *Introducing Linguistics: A Basic Course*, 47)

An dieser Tabelle kann man bestimmte Hypothesen über die lexikalische Kategorie einer bestimmten Wortform ableiten – z.B. dass Wörter, die auf -ism und -ness enden, höchstwahrscheinlich Nomina sind (*communism, sadism, racism, formalism, sadness, childness, soreness, lumpiness*) oder dass Wörter, die auf -able oder -ful enden, höchstwahrscheinlich Adjektive (*careful, revengeful, soulful, joyful, drinkable, deniable, laughable, revokable*).

In einem Regex-Tagger nun werden Wörter mit Bezug darauf untersucht, ob in ihnen derartige Affixe (genauer gesagt: Suffixe) auszumachen sind. Wenn ja, wird eine entsprechende Zuordnung zur passenden Wortart getroffen. Diese Untersuchung verläuft über ein Pattern-Match mit regulären Ausdrücken.

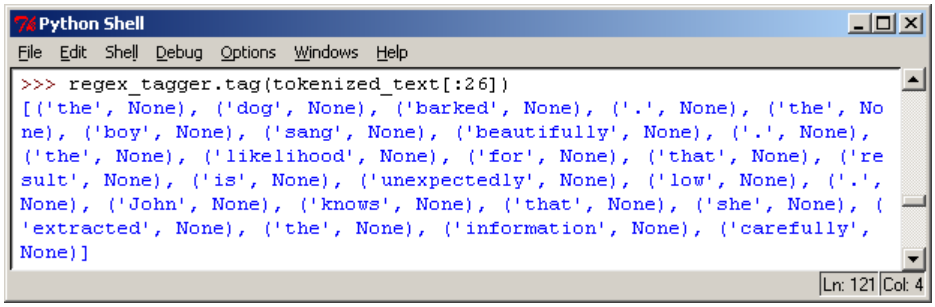
Was wir uns also überlegen müssen, ist: wie müssen diese regulären Ausdrücke aussehen? Nach welchen Mustern soll gesucht werden und welche Tags sollen diesen jeweils zugewiesen werden?

Für diese Situation bietet es sich an, den Beispieltext als Grundlage zu nehmen. Dafür müssen wir zunächst den Text tokenisieren, und zwar am besten mit dem **`WordPunctTokenizer`**.

Erzeugen Sie in der Datei **`tagging.py`** ein Objekt namens **`tokenized_text`**, indem Sie **`text`** mit dem WordPunctTokenizer tokenisieren.

Weisen Sie anschließend in der Datei **`tagging.py`** der Variablen **`regex_tagger`** das Objekt **`nltk.RegexpTagger([])`** zu. Achten Sie darauf, dass der Parameter eine leere Liste ist.

Nun können wir die Datei kompilieren und auf die Shell zurückkehren. Wenn wir unseren tokenisierten Text mit dem Regex-Tagger untersuchen, ist das Ergebnis natürlich grauenvoll, denn wir haben keinerlei Angaben über mögliche Wortendungen gemacht:



[3] These descriptions are taken from Andrew Carstairs-McCarthy's *Introduction to English Morphology* (Edinburgh: Edinburgh University Press 2002). 'X' is to be substituted with the base to which the affix attaches.

Dieses aber können wir ändern, indem wir entsprechende reguläre Ausdrücke formulieren und diesen die entsprechenden Kategorien zuweisen. Formal werden regulärer Ausdruck und Kategorie als Tupel notiert, und der Regex-Tagger mit einer einer Liste solcher Tupel als Parameter aufgerufen. Das Prozedere soll an einem kleinen Beispiel demonstriert werden.

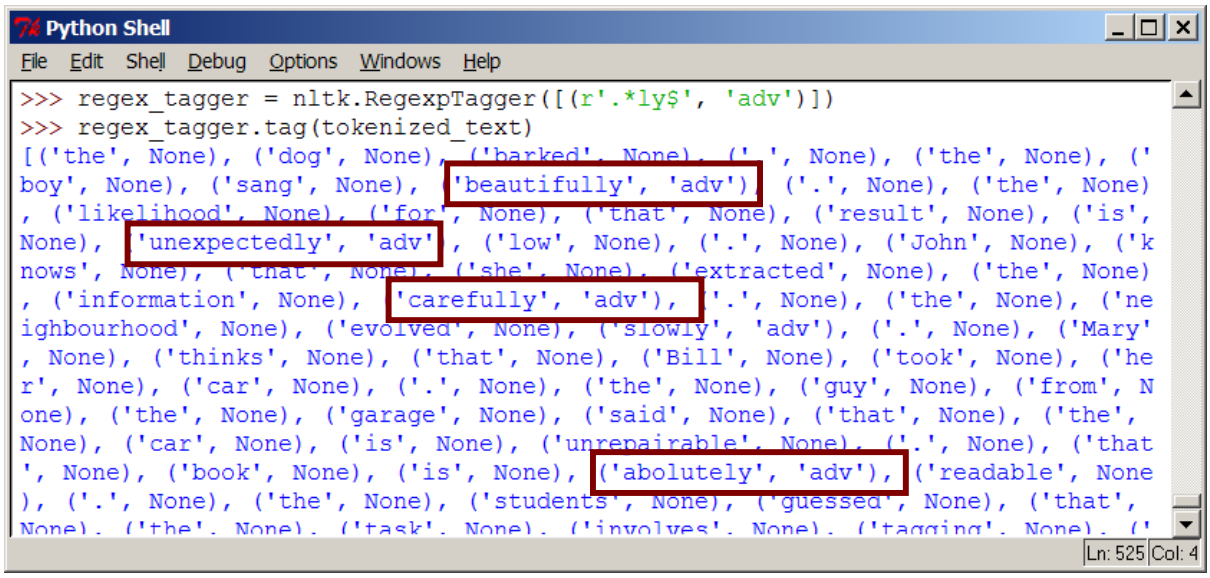
Im obigen Datensatz treten die Wörter *beautifully*, *unexpectedly* usw. auf: allesamt Adverbien mit der typischen Endung *-ly*. Wir erstellen ein Tupel nach dem Muster ('RA', 'KAT'): wie folgt:

(r'.*ly$', 'adv')

Der reguläre Ausdruck RA ist eine Zeichenkette folgender Elemente:

- ein Punkt (welcher für ein beliebiges Zeichen außer \n steht),
- ein Kleene-Stern,
- die Zeichenfolge *ly*,
- das Dollarzeichen, das hier das Tokenende markiert.

Rufen wir unseren Regex-Tagger mit diesem Tupel auf, sind die fraglichen Tokens entsprechend getaggt:



Analysieren Sie den Text mit Bezug auf die Frage, welche Wörter sich für einen Regex-Tagger eignen. Formalisieren Sie Ihre Ergebnisse dann in Form einer Tupelliste [('RA1', 'KAT1'), ('RA2', 'KAT2')]. Weisen Sie in der Datei **`tagging.py`** der Variablen **`regex_tagger`** das Objekt **`nltk.RegexpTagger([])`** zu und geben Sie Ihre Tupelliste als Parameterwert ein. Prüfen Sie Ihr Ergebnis.

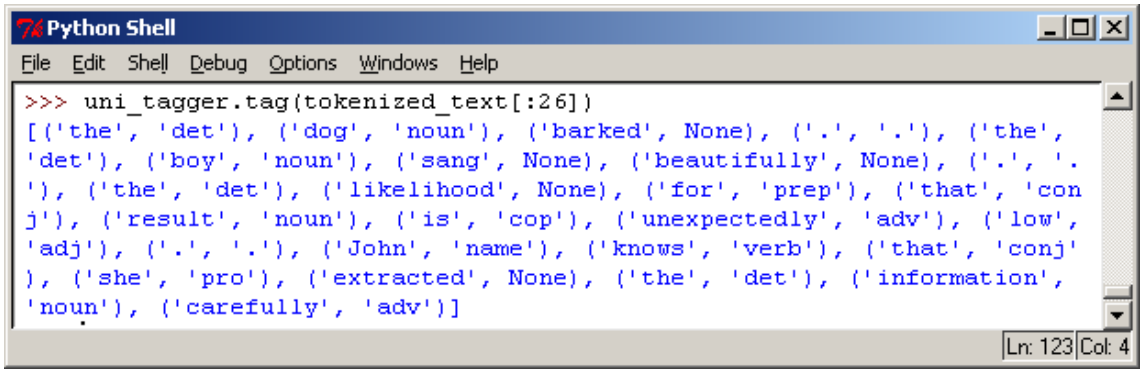
Der Unigramm-Tagger.

Ein Unigramm-Tagger muss trainiert werden und kommt zu einem Ergebnis, wenn im zu taggenden Text Token auftreten, die auch im Trainingscorpus vorhanden sind.

Erstellen Sie in der Datei **`tagging.py`** die Objekte **`brown_train`** und **`brown_train2`** (s.o., S. 5). Achten Sie darauf, dass diese unterhalb der Funktionen stehen, auf die sie sich berufen (konkret: auf **`replace_tags()`**).

Weisen dann Sie das Objekt **`nltk.UnigramTaggger(brown_train2)`** der Variablen **`uni_tagger`** zu. Damit erstellen Sie einen mit **`brown_train2`** trainierten Unigrammtagger.

Wenn wir den Bespieltext mit unserem Unigramm-Tagger taggen, kommt dabei folgendes heraus:



Der Unigramm-Tagger mit Regex-Tagger als Backoff

Interessant wird die Sache dann, wenn es darum geht, verschiedene Tagging-Formalismen zu kombinieren. Wir beginnen mit einer Kombination aus Unigram-Tagger und Regex-Tagger. Informell könnte man sagen, dass unser neuer Unigramm-Tagger nun einen Regex-Tagger in der Hinterhand hat: in den Fällen, in denen der Unigram_Tagger nicht weiterkommt, wird der Regex-Tagger aktiv und versucht sein Glück. Ausgedrückt wird das als *backoff*-Parameter. Vorausetzung für das Funktionieren ist, dass Sie die weiter oben beschriebene Aufgabe bezüglich des Regex-Tagger-Parameters gelöst haben

Weisen Sie in der Datei **`tagging.py`** der Variablen **`unireg_tagger`** das folgende Objekt zu:

```
nltk.UnigramTaggger(brown_train2, backoff = regex_tagger)
```

Testen Sie diesen Tagger am Beispieltext. Was fällt Ihnen auf? In welcher Hinsicht liefert der Uniregtagger bessere Ergebnisse als der Unigrammtagger?

Ein Bigramm-Tagger mit dem Unireg-Tagger als Backoff

Ein Bigramm-Tagger berücksichtigt bei seiner Analyse, welches Token dem gerade bearbeiteten Token vorausgeht (für mehr Info zu N-Gramm-Taggern allgemein und Bigramm-Taggern im Besonderen siehe den Text *Automatischen lexikalische Analyse*, den Sie über unsere Webseite abrufen können). Für sich genommen zeitigt er genau so schlechte Resulate wie der Regex-Tagger, aber mit einem Unigramm- und Regex-Tagger als Backoff sieht das schon anders aus.

Weisen Sie in der Datei **`tagging.py`** der Variablen **`kombi_tagger`** das folgende Objekt zu:

```
nltk.BigramTaggger(brown_train2, backoff = unireg_tagger)
```

Testen Sie diesen Tagger am Beispieltext. Was fällt Ihnen auf? In welcher Hinsicht liefert der Kombitagger bessere Ergebnisse als der Uniregtagger?

Teil 2: Tokenübergreifendes Taggen: Chunk-Parsing

Grundlagen für diesen Teil: alles, was für den ersten Teil relevant ist, sowie ggf. Kapitel 7 im NLTK Buch und Übung 2, insbesondere die dazugehörigen Kommentare

Kurzeinleitung: Shallow Parsing vs Deep Parsing

Parsing im traditionellen Sinn, in der Informatik und der Computerlinguistik

Der Begriff 'Parsing', der vom Lat. *pars orationis* (Redeteil) abgeleitet ist, taucht im Rahmen der Informatik und der Computerlinguistik häufig auf, hat allerdings eine lange Tradition und existierte schon zu einer Zeit, als es weder Informatik noch Computerlinguistik gab.

- Parsing im traditionellen Sinne liegt vor, wenn ein Mensch den Wörtern eines Satzes nacheinander einen Redeteil zuweist, die grammatischen Kategorien bestimmt und die grammatischen Beziehungen zwischen den Wörtern (Subjekt und verschiedene Arten von Objekten, Kongruenzbeziehungen etc.) identifiziert. In diesem Sinne ist Parsing
  - eine Operation, die Menschen auf Segmente einer natürlichen Sprache (gewöhnlich Sätze in geschriebener Form) ausführen,
  - mit einer Beschreibung dieser Segmente als Resultat. Diese ist grammatischer Natur, d.h. sie beschreibt Fakten, die sich auf das Vorkommen von Einheiten einer bestimmten Sprache und die Beziehungen ihnen beziehen.
- Parsing in der Informatik bedeutet, dass auf einen symbolischen, formalisierten Input bestimmte Operationen angewendet werden, die eine sukzessive (partielle) Anordnung dieser Symbole in größeren Einheiten eines bestimmten Typs und die Interpretation dieser Anordnungen als Zustandsänderungen in der Maschine zum Resultat hat.
- Parsing in der Computerlinguistik bedeutet, dass die im traditionellen Parsing von Menschen durchgeführte Analyse eines natürlichsprachlichen Inputs von Computern übernommen wird und in einer formalen Repräsentation der Satzstruktur bzw. Teilen davon resultiert.

Für uns relevant ist nur das Parsing im Rahmen der Computerlinguistik. Wir können hier diverse Parser unterscheiden, die sich hinsichtlich einer Reihe von Faktoren unterscheiden, z.B.

- Zweckbestimmung
- verwendeter Algorithmus & Verarbeitungsstrategie
- zugrundeliegender Grammatikformalismus
- Programmiersprache etc.

Die Gretchenfrage ist – wie bei jeder maschineller Analye natürlicher Sprache – die erste dieser Fragen, also "Welchem Zweck dient das Parsing?"

In Abhängigkeit davon, wozu ein Text bzw. Satz analysiert wird, ändern sich die für die Analyse verwendeten Verarbeitungsstrategien und das bei der Analyse verwendete Regel- und Kategoriensystem.

Deep Parsing

Angenommen, es ginge darum, einen bestimmten Grammatikformalismus wie z.B. die Kategorialgrammatik, die HPSG, eine PS-Grammatik usw. auf seine 'formale' Tauglichkeit zu überprüfen, würde eben dieser Formalismus implementiert und auf eine Reihe von Eingabesätzen angewendet. Das Ergebnis wäre eine vollausgebaute syntaktische Analyse der Sätze, die alle Eigenschaften natürlicher Sprache, die vom Formalismus abgedeckt werden, wie z.B. Rekursivität, Fernabhängigkeiten usw. reflektiert. Eine Analyse des Satzes 'Heute haben die Kinder das Brot gegessen', basierend auf einer der Modellvarianten generativer Grammatik chomskyscher Prägung, müsste z.B. das folgende Resultat zeitigen:
$[Heute_i\ [[haben]_C{}^0\ [[die\ Kinder]_N{}^2\ [[e']_I{}^0\ [t_i\ [das\ Brot]_N{}^2\ gegessen\ e]_V{}^2]_I{}^1]_I{}^2]_C{}^2$.
Eine derartige, quasi tiefgehende Analyse fällt unter den Begriff *deep parsing*.

Shallow Parsing

Im Rahmen des *Information Retrieval* und *Text Mining* hingegen geht es nicht primär darum, einen Text vollständig syntaktisch zu analysieren, sondern z.B. darum, bestimmte Information zu extrahieren (z.B. bei einer Internetsuche). Für diese Aufgabe ist es nicht wichtig, die genau Struktur der Sätze zu erzeugen, sondern bei der Analyse auf bestimmte (z.T. semantisch motivierte) Kategorien wie z.B. Eigennamen, Handlungen, Ortsbezeichner und den Relationen zwischen diesen zu achten. Für den gleichen Satz wie weiter oben wäre also eher eine Analyse angebracht, die auf einen Telegrammstil hinausläuft und dabei z.B. nur alle NP und das Verb identifziert:

$Heute\ haben\ [die\ Kinder]_{NP}\ [das\ Brot]_{NP}\ gegessen_{Verb}$

Eine derartige, sozusagen flache Analyse fällt unter den Begriff *shallow* parsing. Das Beispiel ist dabei ein Vertreter des sogenannten *partial parsings*, also einer Variante des *shallow parsings*, bei dem aus einem Satz bzw. Text bestimmte Elemente (wie alle NP, alle VP, alle ortsbezeichnenden PP usw.) herausgepickt

werden und andere Elemente dagegen unberücksichtigt bleiben können (deshalb die Bezeichnung 'partielles Parsing'). Eine Gruppe von Elementen, wie im o.a. Beispiel die Wörter 'die' und 'Kinder' sowie 'das' und 'Brot' werden zusammengehörig als Chunk bezeichnet, daher auch der Name ***chunk-parser***. Wie Sie in der Aufgabe sehen werden, können wir auch noch verschiedene Typen von Chunkparser unterscheiden, z.B. in Abhängigkeit davon, auf wievielen Strukturebenen die Analyse angesiedelt ist.

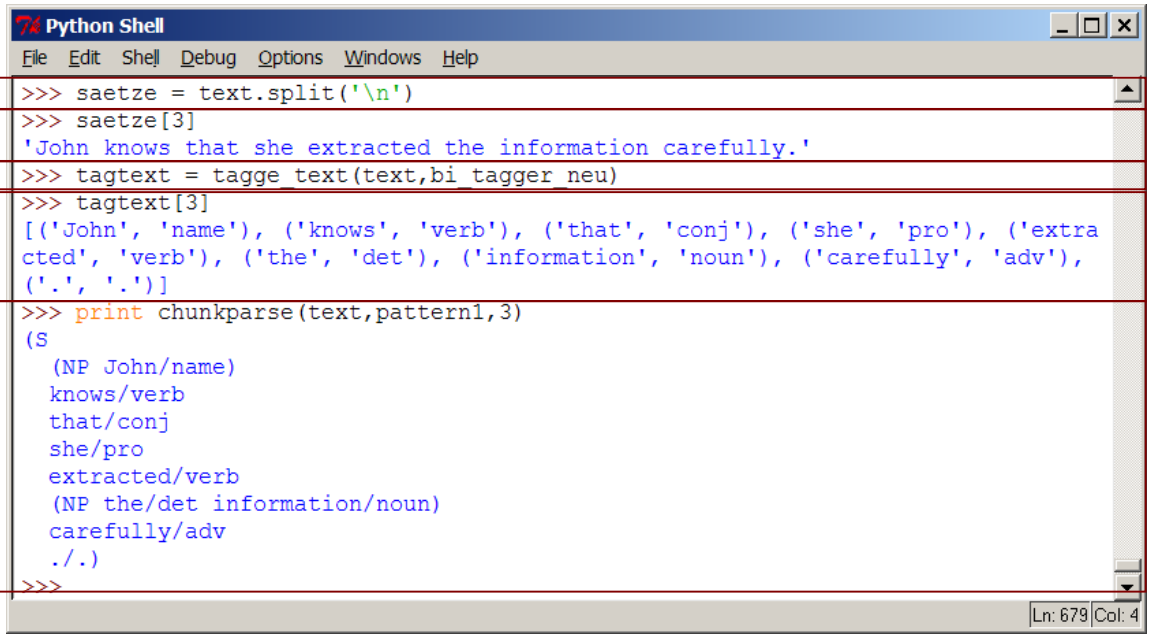
✧✧✧✧✧

Vorbemerkung:

Diese Aufgabe verfolgt mehr als eine Zielsetzung. Sie soll nicht nur in den primären Gegenstand, das Chunk-Parsing mit Python, einführen, sondern auch bestimmte Python-Verfahren illustrieren, die eigentlich erst in den folgenden Aufgaben relevant werden. Wundern Sie sich deshalb nicht, wenn Ihnen bestimmte Teilaufgaben als überflüssig für das Erreichen des eigentlichen Zieles erscheinen: diese dienen dann möglicherweise dazu, grundlegende Techniken auszuprobieren.

Beschreibung der Aufgabe:

In dieser Aufgabe sollen in einem getaggten Text mithilfe regulärer Ausdrücke bestimmte tokenübergreifende Muster identifiziert und mit Kategorialsymbolen wie 'NP' oder 'PP' annotiert werden. Der folgende Screenshot zeigt, worum es in etwa geht:



Zur Erklärung:

1. der Variablen **`saetze`** wird die Auswertung des Ausdrucks **`text.split('\n')`** zugewiesen. **`text`** ist die Liste von Sätzen, die Sie für die erste Aufgabe eingegeben hatten.
2. der 4. Satz von **`saetze`** wird angezeigt: **`'John knows… carefully.'`**.
   Anmerkung: 1 und 2 dienen nur dazu, Ihnen im Screenshot zu zeigen, um welches Objekt es sich in 3 und 4 handelt.
3. der Variablen **`tagtext`** wird die Auswertung des Ausdrucks **`tagge_text()`** zugewiesen. Die Funktion **`tagge_text()`** ist im Programm definiert und erzeugt auf Grundlage eines Textes den diesem entsprechenden getaggten Text. 'Neu' heißt, dass dafür nicht die Brownschen, sondern die aus diesen abgeleiteten neuen Tags aus der ersten Aufgabe verwendet werden.
4. **`tagtext[3]`** zeigt das 4. Element aus **`tagtext`**.
   Anmerkung: 3 und 4 dienen nur dazu, Ihnen im Screenshot zu zeigen, um welches Objekt es sich in 5 handelt
5. dies ist der Kernpunkt: das 4. Element aus **`tagtext`** wird chunkgeparst und das Ergebnis gedruckt. Im konkreten Beispiel geht es nur darum, in dem getaggten Satz die NP zu identifizieren.

Um diese Aufgabe zu lösen, müssen wir verschiedene Funktionen definieren: wie bereits gesehen z.B. eine Funktion, die als Input einen Text erhält und als Output eine getaggte Version dieses Textes liefert, dazu müssen wir unsere Tagger neu bestimmen und jeweils unterscheiden, ob das Brown-Tagset oder unser neues Tagset verwendet wird, natürlich muss auch die Funktion **`chunkparse()`** definiert werden, die mit 3 Argumenten aufgerufen wird, nämlich (a) dem zu chunkparsenden Text, (b) dem Regex-Pattern, das verwendet wird und (c) dem Index desjenigen getaggten Satzes im getaggten Text, der chunkgeparst werden soll.

Vorarbeit

Legen Sie in Ihrem Python-Verzeichnis einen Ordner **`chunking`** an. Alle Dateien, die wir für die Aufgabe benötigen, sollen in diesem Ordner zu finden sein. Kopieren Sie die Datei **`tagaufg.py`** in diesen Ordner und benennen Sie sie in **`chunkauf.py`** um.

Schritt (A): Erzeugen der Datengrundlage

1.: Zu importierende Module

Geben Sie die folgenden Anweisungen an den Beginn der der Datei **`chunkauf.py`**:

```
import nltk
from nltk.corpus import brown
from nltk.tokenize import *
from nltk.chunk import *
from nltk.chunk.regexp import *
from re import *
from re.show import *
```

2.: Die textliche Datengrundlage

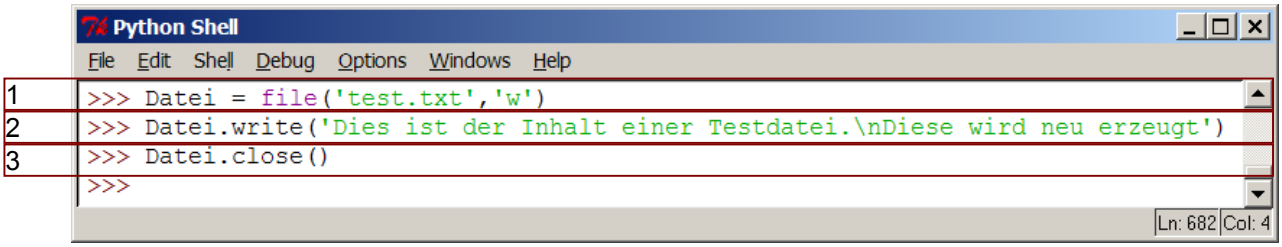
In der Datei **`chunkauf.py`** befindet sich bereits ein Objekt namens **`text`**, dem eine Zeichenkette mit diversen Sätzen zugeordnet ist. Mit diesem Objekt wollen wir weiterhin arbeiten, d. h. es wird darum gehen, die darin befindlichen Sätze zu taggen und anschließend chunkzuparsen. Um einen Einstieg in das Chunkparsen zu bekommen, bei dem von vorneherein bestimmte Schwierigkeiten umgangen werden, benötigen wir einen korrekt getaggten Text.
Dabei gibt es allerdings ein Problem: die Tagger, die wir im Rahmen von **`tagaufg.py`** erstellt haben, liefern für diesen Text kein 100%ig korrektes Ergebnis. Dieses Problem hätten wir auch mit jedem weiteren Text, der getaggt werden soll, es ist mithin, wie in der Veranstaltung bereits erwähnt, i.d.R. immer nötig, einen maschinell getaggten Text per Hand nachzueditieren. Das aber geht in unserem Fall gar nicht so ohne weiteres: bis jetzt, d.h. über **`tagaufg.py`**, ist die getaggte Version des Objektes **`text`** quasi virtuell: sie existiert nur im Arbeitsspeicher von Python. Selbst wenn wir diesem Objekt einen Namen zuweisen, haben wir doch keinen Zugriff darauf. Was wir also machen müssen, ist, die getaggte Version in eine Datei schreiben zu lassen, die wir dann per Hand editieren und abspeichern können, um die darin befindlichen Daten dann wieder – für die Weiterverarbeitung – in Python zu laden. Die folgenden Abschnitte befassen sich damit, wie diese Verfahren in Python zu ermöglichen sind.

Dateimanagement in Python

In Python gibt es eine Reihe von Methoden, die das Erzeugen und Einlesen von Dateien bzw. Dateiinhalten ermöglichen. Wir konzentrieren uns ausschließlich auf diejenigen Methoden, die für unsere Aufgabe relevant sind.

⇨ für mehr Information zum Python File Input/Output siehe z.B.
http://en.wikibooks.org/wiki/Non-Programmer's_Tutorial_for_Python/File_IO
Abschnitt 1.8 (Dateioperationen) in
http://www.informatik.uni-frankfurt.de/~sschaef/python/Tutorial_python.pdf

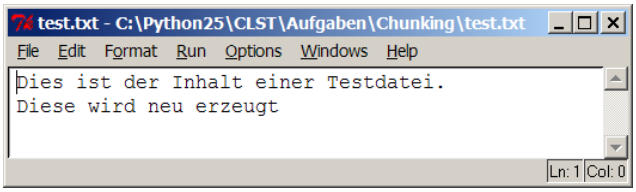
Die zentrale Methode in diesem Kontext ist **`file(Dateiname,Parameter)`**
Fangen wir von hinten an: **`Parameter`** gibt an, ob die Datei gelesen wird (**`'r'`**, für *read*), oder ob sie geschrieben wird (**`'w'`** für *write*).[4] Der folgende Screenshot zeigt ein Beispiel:



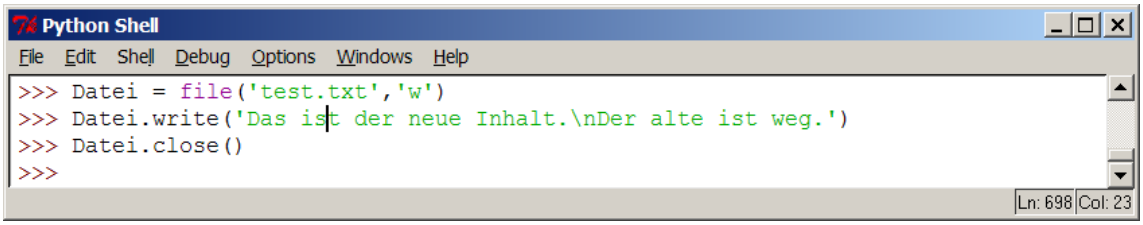
1. Der Variablen **`Datei`** wird ein Fileobjekt zugewiesen und dabei eine Datei namens **`'test.txt'`** zum Schreiben (Parameter **`'w'`**) geöffnet. Wenn eine Datei dieses Namens nicht vorhanden ist, wird sie neu angelegt, andernfalls – wenn sie bereits besteht – wird ihr Inhalt gelöscht.
2. Verwendet die Methode **`.write()`** auf **`Datei`**, um eine Zeichenkette in die Datei zu schreiben.
3. Verwendet die Methode **`.close()`** auf **`Datei`**, um **`'test.txt'`** zu schießen. Erst jetzt wird der in 2. aufgeführte Inhalt in die Datei geschrieben.

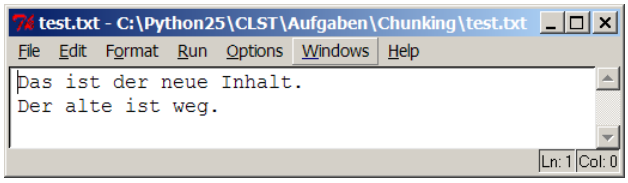
Nach diesen Schritten können wir die Datei 'test.txt' öffnen und uns ihren Inhalt anzeigen lassen:

[4] Es gibt noch weitere Parameter, die für uns aber nicht relevant sind.



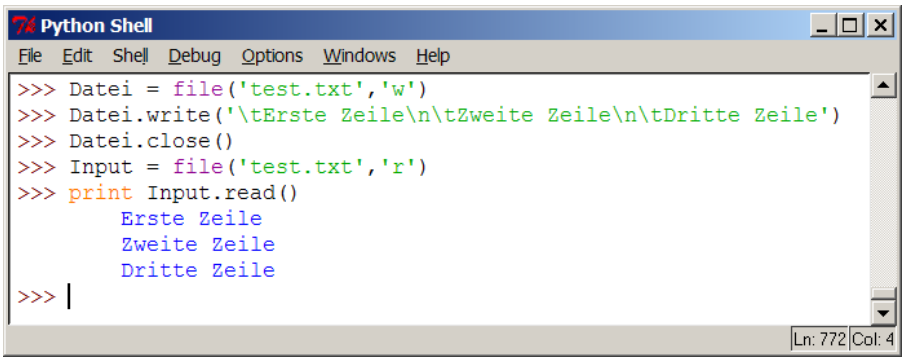
Bei erneutem Aufruf der o.a. Methoden wird der Inhalt von 'test.txt' durch den neuen Inhalt überschrieben:



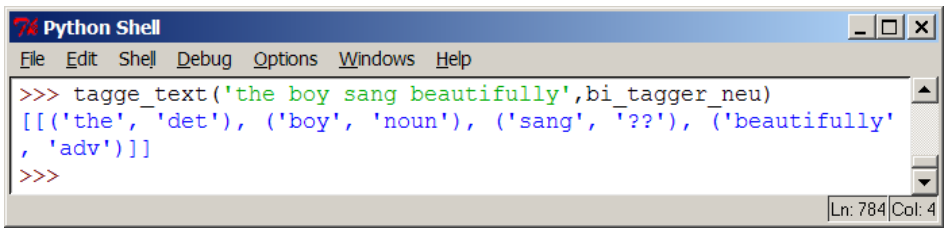


- Testen Sie die Methode **`file()`** mit eigenen Variablen und Dateinamen aus.
- Geben Sie bei **`.write()`** auch einmal eine Liste aus Zahlen ein, z.B. wie in **`Datei.write([1,2,3,4])`**. Was sagt die Fehlermeldung aus?

Wenn Sie die letzte Miniübung gemacht haben, werden Sie festellen, dass **`.write()`** nicht mit jedem Datentyp funktioniert: das Argument sollte – vereinfacht gesagt – eine Zeichenkette sein. Diesen wichtigen Punkt behalten wir im Hinterkopf.
Um nun den Inhalt einer Datei einzulesen, verwenden wir die gleiche Methode wie beim Schreiben, allerdings mit anderem Parameter, nämlich 'r' für *read*. Der nachstehende Screenshot zeigt das ganze Verfahren 'Datei schreiben & Dateiinhalt einlesen' auf einmal:



Wenn es nun um die Aufgabe geht, einen Text taggen zu lassen und das Ergebnis in eine Datei zu schreiben, haben wir ein Problem. Dazu sehen wir uns den folgenden Screenshot an, in dem deutlich wird, worum es geht:



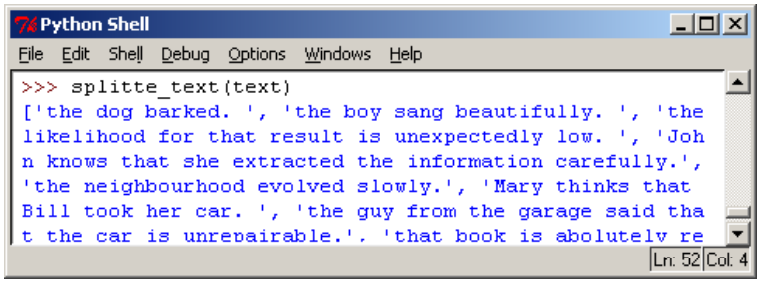
Das, was beim Taggen eines Textes nach unseren bisherigen Arbeiten herauskommt, ist eine Gesamtliste von Tupeln. Was wir haben wollen, ist, aber, dass als Ergebnis pro Satz eine Liste getaggter Tokens zurückgegeben wird. Wir re-definieren nun **`tagge_text()`** derart, dass genau dieses Ergebnis erzielt wird. Dafür definieren wir zunächst eine Reihe von Hilfsfunktionen.

Ein Schritt zurück

splitte_text(Text)

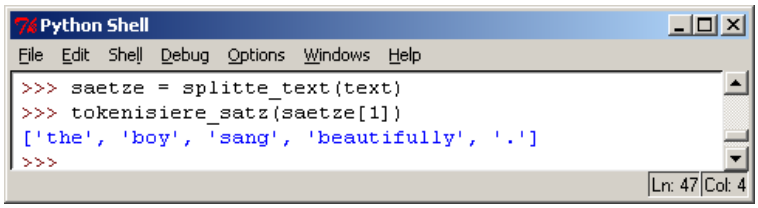
Diese Methode soll einen Text so aufbereiten, dass die einzelnen Sätze als Zeichenkette in einer Liste erscheinen

Voraussetzung ist, dass der Text entsprechende Absatzmarkierungen aufweist. Diese Methode kann einfach per `.split('\n')` definiert werden.



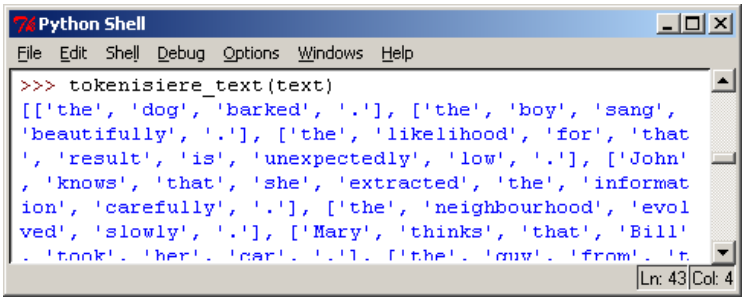
tokenisiere_satz(Satz):

Diese Methode ist bereits definiert. Angewendet auf das Ergebnis von **`splitte_text(Text)`** mit einem entsprechendem Index (im Screenshot der zweite Satz von **`text`**) käme dabei heraus:



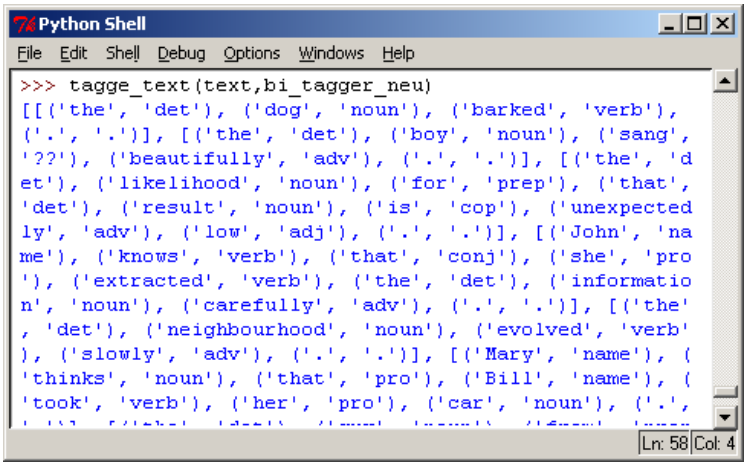
tokenisiere_text(Text)

Diese Methode wird mit einer leeren Liste 'gestartet', die per **`.append()`** sukzessive, also mit einer **`for`**-Schleife, um die einzelnen Elemente eines gesplittenen Textes erweitert wird. Sie müssen bei dieser Methode also **`splitte_text()`** einsetzen. Da die einzelnen Elemente, sprich die Sätze, ihrerseits tokenisiert sein sollen, müssen Sie entsprechend auch die Methode **`tokenisiere_satz()`** verwenden:



tagge_text(Text,Tagger)

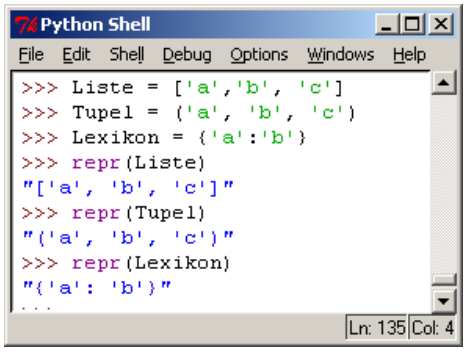
Diese Methode soll einen Text **`Text`** mit einem bestimmten **`Tagger`** (z.B. **`kombi_tagger`**, **`bi_tagger`** etc.) so taggen, dass eine Liste von Listen von **`(Token,Tag)`**-Tupeln zurückgegeben wird. Sie wird ebenfalls mit einer leeren Liste gestartet. Voraussetzung ist, dass der zu taggende Text in Form einer Liste von Listen von Zeichenketten vorliegt, also müssen Sie bei der Definition dieser Methode auf **`tokenisiere_text()`** zurückgreifen. Beim Programmablauf wird dann in einer for-Schleife die zunächst leere Liste sukzessive per **`.append()`** aufgebaut:



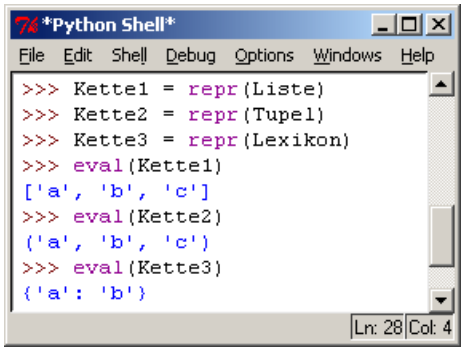
Implementieren Sie die o.a. Methoden.

Nun zurück zum Problem, nämlich der Frage, wie man bestimmten Inhalt, der nicht vom Datentyp 'Zeichenkette' ist, in eine Datei schreiben und aus dieser wieder auslesen kann.
Dafür verwenden wir zwei sehr praktische vordefinierte Funktionen: **`repr()`** und **`eval()`.**

`repr()`
erzeugt eine Zeichenkettenrepräsentation eines beliebigen Objektes:

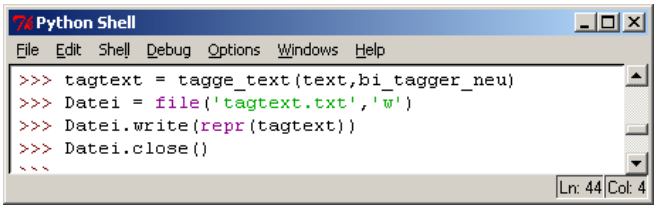


`eval()`
ist eine Methode, die eine Zeichenkette entsprechend auswertet:



⇨ Die beiden Methoden **`repr()`** und **`eval()`** haben mit der sogenannten 'Serialisierung' und 'Deserialisierung' in der Informationsverarbeitung zu tun. Für mehr Hintergrund siehe http://en.wikipedia.org/wiki/Serialization

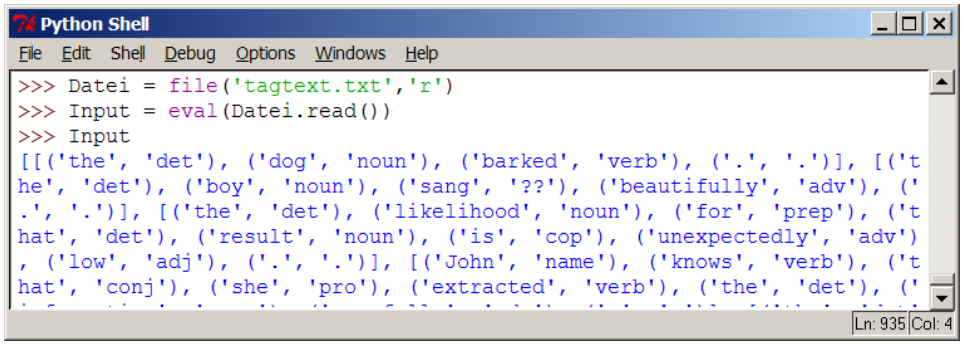
Im nebenstehenden Screenshot wird gezeigt, wie ein ein Objekt **`tagtext`** in eine Datei namens **`tagtext.txt`** geschrieben wird. Bei **`tagtext`** handelt es sich um die getaggte Version unseres Ursprungstextes, den wir in **`chunking.py`** aufgeführt haben. Da es sich hierbei also um eine Liste von Listen von Tupeln handeln, wird die Methode **`repr()`** auf **`tagtext`** angewendet.



Jetzt können wir die Textdatei 'tagtext.txt' öffnen und ändern.

Erzeugen Sie – wie im Screenshot – eine Datei 'tagtext.txt' auf der Basis des getaggten Textes **`text`**. Ändern Sie darin alle falschen Tags und ergänzen Sie für Tokens, die nicht erkannt wurden ('??') die Tags. Speichern Sie, wenn Sie fertig sind, 'tagtext.txt' ab.

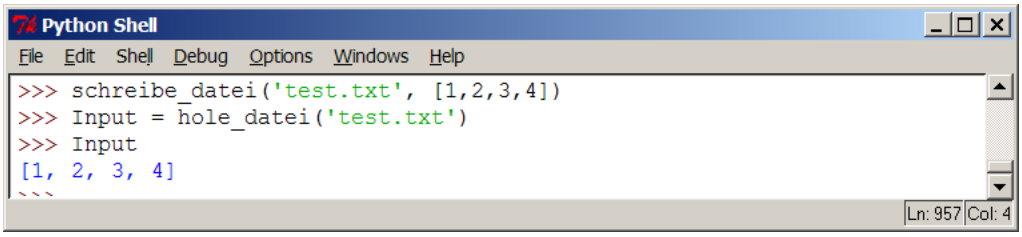
Nun stellt sich die Frage, wie man den Dateiinhalt von 'tagtext.txt' wieder in den Arbeitsspeicher bekommt, und dazu verwenden wir das Verfahren mit **`eval():`**



Prüfen Sie – wie im Screenshot – ob Ihre modifizierte Datei eingelesen werden kann.

Ein eigenes I/O-Modul

Für die weiteren Aufgaben wäre es sehr angenehm, wenn wir zwei Input/Output Methoden zur Verfügung hätten, nämlich eine Methode **`schreibe_Datei()`**, die zwei Argumente hat: **`schreibe_Datei(Dateiname,Objekt)`** und deren Aufgabe es ist, **`Objekt`** als Zeichenkette in **`Datei`** zu schreiben, sowie eine Methode **`hole_datei()`**, die ein Argument hat (**`hole_datei(Dateiname)`**) und den in der Datei namens **`Dateiname`** befindlichen Inhalt, also eine Zeichenkette, auswertet und in den Arbeitsspeicher lädt. Der folgende Screenshot zeigt, worum es geht:



Implementieren Sie auf der Grundlage der auf Seite 7 dargestellten Verfahren zum Schreiben und Lesen von Dateien die Methoden **`schreibe_datei()`** und **`hole_datei()`**.
Legen Sie für diese Aufgabe ein eigenes Modul namens **`datei.py`** im Chunking-Ordner an.

Schritt (B): Chunking per regulärer Ausdrücke: Wir bauen einen super (eingeschränkten…) Shift-Reduce Parser

Im Nachfolgenden werden wir eine Chunking-Methode anwenden, deren Grundlage wir im Prinzip bereits beim Tagging bzw. den Übungen zu regulären Ausdrücken kennengelernt haben: dem Chunking per regulären Ausdruckes.
Es soll darum gehen, mit Hilfe von regulären Ausdrücken (RAs) in einem Tagset, d.h. einer als Folge von Tags repräsentierten präterminalen[5] Kette, Ausschnitte – sog. *chunks* – zu finden, die syntaktischen Kategorien wie Nominalphrase, Verbalphrase, Adjektivphrase etc. entsprechen. So entspräche dem Satz *the boy kicked the colourful ball* die präterminale Kette *det/noun/verb/det/adj/noun/* und mit einem geeigneten RA würden Nominalphrasen wie folgt markiert werden: **`{det/noun/}verb/{det/adj/noun/}`**.
In einem weiteren Schritt wird es darum gehen, solche *chunks* schrittweise durch die entsprechenden Kategorialsymbole zu ersetzen, z.B.
`det/noun/verb/det/adj/noun/`
`np/verb/np/`
`np/vp/`
`s/`
Dieses ist die syntaktische Analyse des Satzes *the boy kicked the colourful ball*. Dem Prinzip nach implementieren wir hier einen *Shift-Reduce-Parser*.

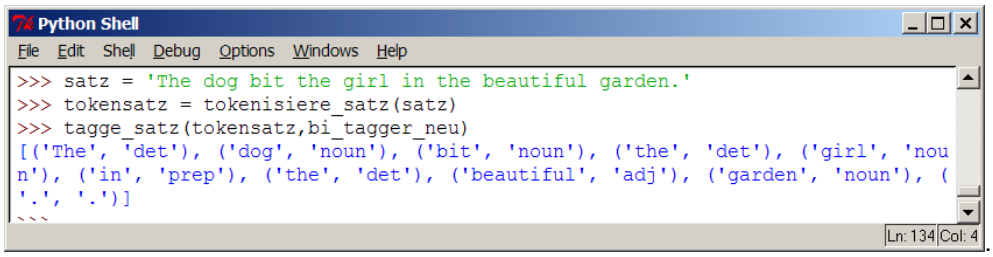
⇨ Für mehr Information zu *Shift-Reduce*-Parsern siehe z.B. den Beitrag in der Wikipedia (http://en.wikipedia.org/wiki/Bottom-up_parsing) oder Kapitel 8 (*Context Free Grammars and Parsing*), Abschnitt 8.5.3, im NLTK-Buch (http://nltk.org/doc/en/parse.html)

Wie aber in der Kurzeinleitung zu *Shallow vs Deep Parsing* in diesem Text zu lesen ist, kann diese syntaktische Analyse aufgrund des verwendeten Formalismus, nämlich dem Chunking, nur sehr schmalbrüstig ausfallen, entsprechend eingeschränkt ist das Ergebnis. Das Schöne an dieser Aufgabe ist, dass Sie in ihr den Unterschied zwischen *Shallow Parsing* und *Deep Parsing* sozusagen hautnah erfahren.

Vorarbeit: Tag-Zeichenketten

Wie Sie gleich sehen werden, geht es uns beim Chunking darum, Sätze – keine Texte, also Listen von Sätzen – einzeln abzuarbeiten. Deshalb bietet es sich zunächst an, unser Modul **`chunking.py`** um eine Funktion zu ergänzen, die als Input einen einzelnen, tokenisierten Satz nimmt und als Output diesen Satz in Form einer Liste von Token/Tag-Tupeln zurückgibt. Wir nenen diese Funktion **`tagge_satz(Tokensatz,Tagger)`** und rufen sie mit dem tokenisierten Satz und dem Tagger unserer Wahl als Argumente auf:

[5] Die präterminale Kette eines Satzes ist eine Kette der der Wortfolge entsprechenden lexikalischen Kategorien. Der Satz *the horse jumped* hat die präterminale Kette *det noun verb*.

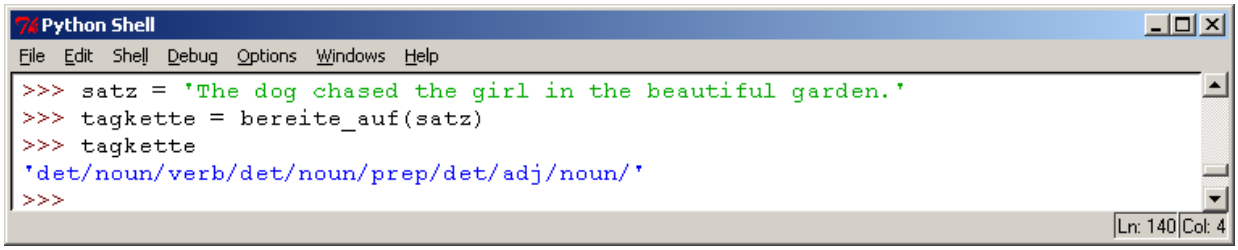


Definieren Sie die Funktion **`tagge_satz()`** wie oben angegeben.

Nun können wir Daten in der für uns interessanten Form erzeugen. Wie gesehen ist in Satz als eine Liste von Token-Tag-Tupeln repräsentiert, z.B.
`[('the', 'det'), ('boy', 'noun'), ('kicked','verb'), ('colourful','adj'), ('ball','noun')]`
Reguläre Ausdrücke operieren jedoch über Zeichenketten. Wir müssen also zunächst die Tags aus den Listen extrahieren und zu einer Zeichenfolge zusammenbauen. Dabei soll das Zeichen **'/'** als Taggrenze verwendet werden. Aus technischen Gründen ist es notwendig, dass auch das letzte Tag auf eine Taggrenze endet.

Schreiben Sie eine Funktion **`mache_tagkette()`**, die wie im obenstehenden Screenshot aus einer Liste von Token-Tag-Paaren die Tags extrahiert und zu einer Zeichenfolge der Form **`tag1/tag2/.../tag_n/`** zusammenfügt. Achten Sie darauf, dass mögliche Satzzeichen am Listenende ausgefiltert werden!

Wie Sie im Screenshot sehen können, umfasst die Prozedur, einen ungetaggten Satz für das Chunk-Parsing vorzubereiten, eine Reihe von Methoden. Wir können uns die Sache für die weitere Arbeit sehr vereinfachen, wenn wir diese Methoden in einer Funktion **`bereite_auf(Satz)`** zusammenfassen, die als Input einen ungetaggten Satz nimmt und als Output eine Zeichenkette der in diesem Satz anzufindenden Tags zurückgibt:



Schreiben Sie eine Funktion **`bereite_auf()`**, die wie im obenstehenden Screenshot zu sehen einen Satz als Input nimmt und als Output eine Zeichenkette der in diesem Satz anzutreffenden Tags zurückgibt. Es bietet sich an, die Eingabe darin auch gleich zu normalisieren, d.h. in Kleinbuchstaben umzuwandeln.

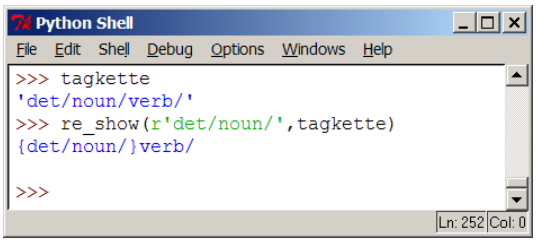
Die Formulierung regulärer Ausdrücke über Tag-Zeichenketten

Sie benötigen für diese Aufgabe die Funktion **`re_show()`**, die wir im Zusammenhang mit den Übungen zu regulären Ausdrücken bereits definiert haben. Zur Erinnerung:

```
def re_show(pat,s):
    print re.compile(pat,re.M).sub("{\g<0>}",s.rstrip()),'\n'
```

Bei der Formulierung von regulären Ausdrücken ist zu beachten, dass diese eigentlich über Zeichenfolgen operieren, während Tags selber Symbole sind, die aus mehreren Zeichen bestehen können. Um den damit verbundenen Probleme aus dem Wege zu gehen, haben wir die Taggrenze **'/'** eingeführt.

Ein einfacher RA für eine Nominalphrase mit Determinator und Nomen wäre z.B. **'det/noun/'**. Die Anwendung des oben definierten regulären Ausdruck auf die Tagkette zeigt in der Tat die erste Nominalphrase an:

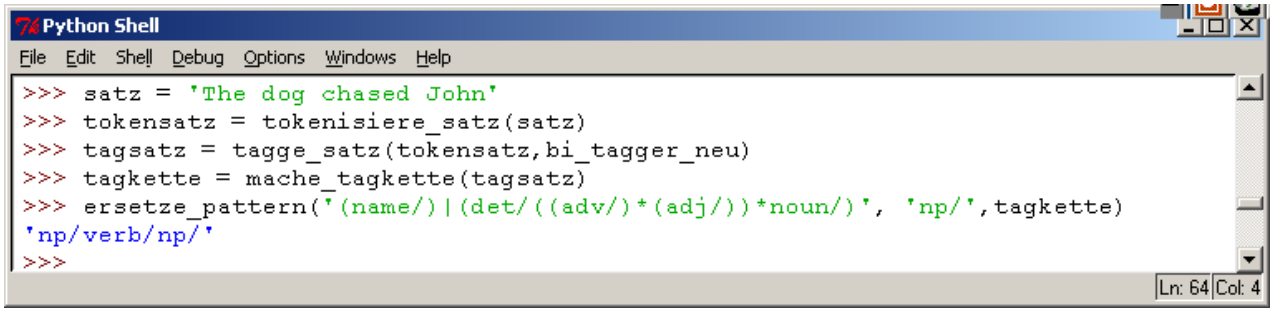


Um auch andere Typen von Nominalphrase zu erkennen, muss das Muster entsprechend modifiziert werden.

Modifizieren und erweitern Sie den RA für Nominalphrasen so, dass die nebenstehenden NP erkannt werden. Testen Sie Ihr Ergebnis aus.:

{ He | The dog | A small dog | A large beautiful dog | A terribly small dog }

Als nächstes soll es darum gehen, das in einem Satz erkannte Muster wie im nachstehenden Screenshot durch ein entsprechendes Kategorialsymbol zu ersetzen:



Definieren Sie eine Funktion **`ersetze_pattern(Pattern, Ersetzung, Ziel)`**, die bei einem erfolgreichen Mustervergleich **`Pattern`** durch **`Ersetzung`** substituiert.

Wenn Sie sich Ihren RA für die NP ansehen, stellen Sie fest, dass dieser einer PS-Regel wie der folgenden entspricht:

NP → { (Det) ((Adv)Adj) Noun | Pro }

Worum es als nächstes gehen soll ist, mehr RAs zu formulieren, nämlich solche, in denen die folgende Grammatik 1 erfasst ist:[6]

Grammatik 1:

S → NP VP
NP → { (Det) (AP) Noun | Pro }
VP → Verb ({ NP | PP })
PP → Prep NP
AP → (Adv) A

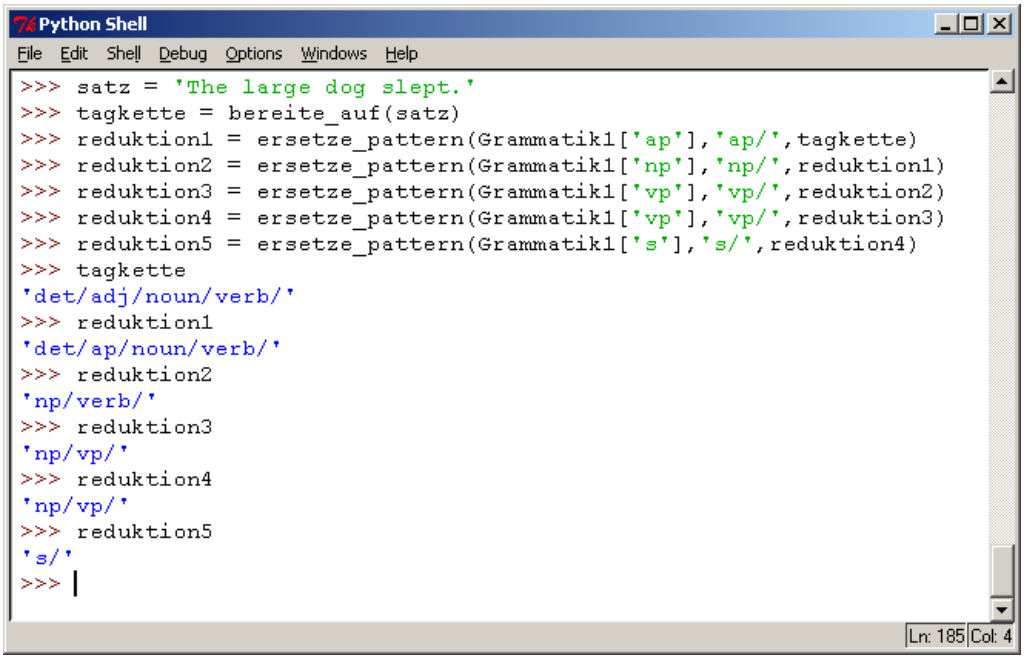
Für diese Aufgabe bietet es sich an, ein Dictionary der Form **`Grammatik1 = {Regelkopf:Regelrumpf}`** anzulegen, in denen der Schlüssel **`Regelkopf`** das Kategorialsymbol der jeweiligen Regel in Grammatik 1 ist und der Wert **`Regelrumpf`** der reguläre Ausdruck, der das Muster für diese Kategorie beschreibt.
Auf diese Weise können wir den RA über **`Grammatik1[Regelkopf]`** referenzieren.

Weisen Sie nach diesem Muster der Variablen **`Grammatik1`** ein Dictionary zu, in welchem die oa. Grammatik 1 beschrieben ist.

Sie können sich beim Aufruf von **`ersetze_pattern()`** jetzt auf das Dictionary beziehen, d.h. dass als erstes Argument anstelle eines vollen RAs nun **`Grammatik1['np']`** oder **`Grammatik1['vp']`** etc. angegeben werden kann.

Wenn Sie einen Satz, genauer gesagt die aus diesem abgeleitete Tagkette, mehrfach dem **`ersetze_pattern()`**-Verfahren unterziehen, reduzieren Sie also diese Tagkette sukzessive bis zu dem Punkt, an dem Sie beim Startsymbol der Grammatik landen, sprich bei 'S'. Sie müssen dafür aber die vorzunehmenden Reduktionen in der richtigen Reihenfolge ausführen: da wir z.B. innerhalb der VP-Regel auf eine NP Bezug nehmen, müssen Sie in der Tagkette zuerst die NP identifizieren und dann erst die VP. Der nachstehende Screenshot zeigt das Verfahren am Beispiel des Satzes *The brown dog slept*:

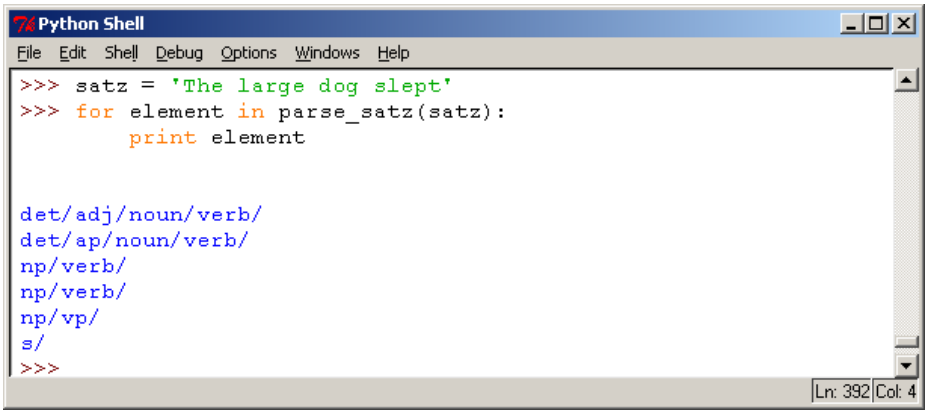
[6] Kongruenz- und Subkategorisierungsphänomene sind hier nicht berücksichtigt



Wie Sie sehen können, beinhaltet die sukzssive Reduktion, dass **`ersetze_pattern()`** zu Beginn mit **`tagkette`** und anschließend immer mit dem Ergebnis der vorherigen Reduktion (**`reduktion1`** – **`reduktion4`**) als 3. Argument aufgerufen wird.

- Testen Sie an den folgenden Sätzen aus, ob alles funktioniert und alle syntaktischen Kategorien erkannt werden. Voraussetzung ist natürlich, dass die Sätze richtig getaggt sind. *She slept.—The dog slept.—The brown dog slept.—The large beautiful dog slept—A very large dog slept.— The dog slept in the garden.—The dog bit him.*

Das Verfahren zur sukzessiven Reduktion der Tagkette ist natürlich reichlich umständlich. Darum schreiben wir eine Funktion **`parse_satz()`**, deren einziges Argument der zu analysierende Satz ist und der jeden einzelnen Schritt der sukzessiven Reduktion protokolliert:



Um diese Funktion schreiben zu können, müssen die folgenden Fragen beantwortet sein:

1. Wie kann ein Protokoll geführt werden darüber, welche Reduktionen erfolgen?
2. Wie kann auf die einzelnen Elemente der zu reduzierenden Tagkette zugegriffen werden?
3. Wie kann die korrekte Reihenfolge der Reduktionen gesteuert werden?

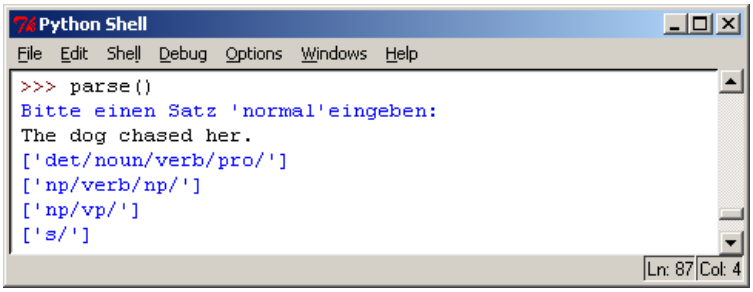
Zu 1.:
Dieses ist relativ leicht zu bewerkstelligen, indem wir in der Funktion ein Objekt namens 'Protokoll' anlegen, dem zunächst die Tagkette zugewiesen wird. Tagkette muss hier als Liste vorgegeben sein: **`Protokoll = [Tagkette]`**, damit **`Protokoll`** dann, in einer for-Schleife, per **`.append()`** jeweils um das Ergebnis eines einzelnen Reduktionsschrittes ergänzt werden kann. Informell sähe das so aus:
Protokoll = [Tagkette]
Wiederhole Folgendes so lange, bis das Ende erreicht ist:
    Tagkette = Ergebnis der Reduktion
    Hänge Tagkette an Protokoll an
Gib Protokoll aus
Gretchenfrage hier ist: worauf bezieht sich diese **`for`**-Schleife?

Zu 2. und 3.:
Praktischerweise können wir diese beiden Fliegen ganz elegant mit einer Klappe schlagen.
Zunächst zu 2: was wir machen müssen, ist, die in der Grammatik formulierten RAs auf eine Zeichenkette, nämlich eine Tagkette, anzuwenden. Wir gehen also bei der Fomulierung unserer Funktion in diesem Punkt von den RA aus und lassen diese Muster in **`Tagkette`** finden. Informell können wir das Prozedere so beschreiben: um das Ergebnis der Reduktion zu ermitteln, gehe **`Tagkette`** mit Bezug auf den in Regel X der Grammatik formulierten RA durch und nehme – so möglich – die entsprechende Ersetzung vor. Der RA wird durch den Schlüssel des jeweiligen Dictionary-Eintrages referenziert. In Python sähe das so aus:
`Tagkette = ersetze_pattern(Grammatik1['xp'],'xp/',Tagkette)`
Und nun zu 3: wir müssen, wie oben gesagt, die Ersetzungsschritte in einer bestimmten Reihenfolge vornehmen, damit ein ganzer Satz abgearbeitet werden kann. Bis dato, beim Abarbeiten eines Satzes per Hand, haben wir einfach die richtige Reihenfolge genommen, um zum gewünschten Resultat zu gelangen (sehen Sie dazu einfach nochmal den ersten der beiden Screenshots auf der vorigen Seite: erst AP, dann NP, dann VP, dann Satz). Das, was wir dabei eingesetzt haben, um die RA zu referenzieren, sind die Kategorialsymbole der Grammatik, die ja in der als Dictionary implementieren Grammatik1 als Schlüssel auftreten. Das Problem bei der Sache ist darin zu sehen, dass wir nun nicht einfach das Dictionary durchgehen und entsprechende Ersetzungen vornehmen können: selbst wenn Sie die Regeln im Dictionary in der Reihenfolge angeben, in der sie letztendlich angwendet werden sollten, funktioniert es nicht, weil Python Daten vom Typ Dictionary intern auf nicht nachvollziehbare Weise sortiert.
Wir können die Fragen 1. (worauf bezieht sich die for-Schleife), 2. und 3. nun so lösen, dass wir in der Funktion **`parse_satz()`** eine Liste oder Zeichenkette verwenden, die die Kategorialsymbole des Dictionaries in der richtigen Reihenfolge angibt und darüber dann die for-Schleife implementieren. Eine informelle Beschreibung der Funktion sieht so aus:

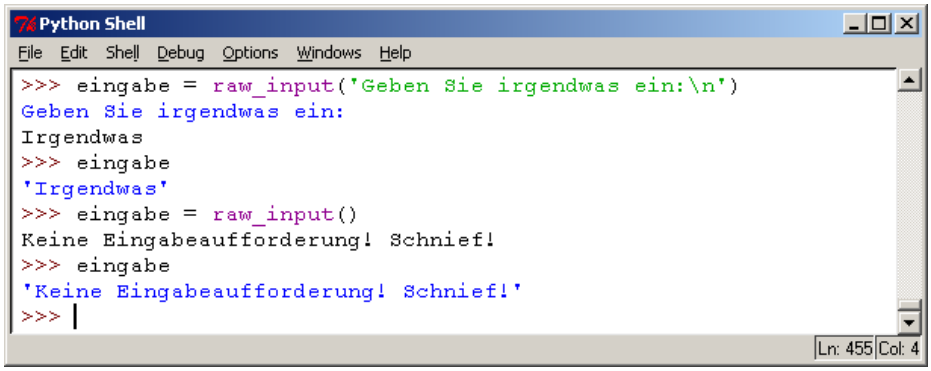
parse_satz(Satz):
    Erzeuge aus Satz eine Tagkette
    Setze Protokoll auf [Tagkette]
    Nehme für jedes Element aus der Liste ['ap', 'pp', 'np' usw] folgendes vor:
        Setze Tagkette auf das Ergebnis der Substition von 'Element' durch den durch 'Element' referenzierten RA in Tagkette
        Hänge Tagkette an Protokoll
    Gebe Protokoll zurück

Implementieren Sie die Funktion **`parse_satz()`**

Wenn Sie alles richtig gemacht haben, können wir jetzt – sozusagen zur Versüßung der Eingabe – noch rasch eine Funktion **`parse()`** definieren, die kein Argument hat sondern den User zur Eingabe eines Satzes auffordert, der dann mit **`parse_satz()`** analysiert wird. Aussehen soll das wie im nachstehenden Screenshot:



Für diese Funktion verwenden wir die Methode **`raw_input()`**. Das (optionale) Argument ist eine Zeichenkette, die den User zur Eingabe auffordert:



Wie Sie hier sehen können, wird die Auswertung des Ausdrucks **`raw_input()`** an eine Variable gebunden (im Screenshot **`eingabe`**), die dann für weitere Zwecke zur Verfügung steht.

Implementieren Sie die Funktion **`parse()`**, die den User auffordert, einen Satz ganz normal (also nicht in Anführungszeichen) einzugeben und diese Eingabe dann an **`parse_satz()`** übergibt.

Jetzt haben wir alles beisammen, um die eingangs gestellte Aussage zu prüfen, nach der "das Schöne an dieser Aufgabe ist, dass Sie in ihr den Unterschied zwischen *Shallow Parsing* und *Deep Parsing* sozusagen hautnah erfahren."

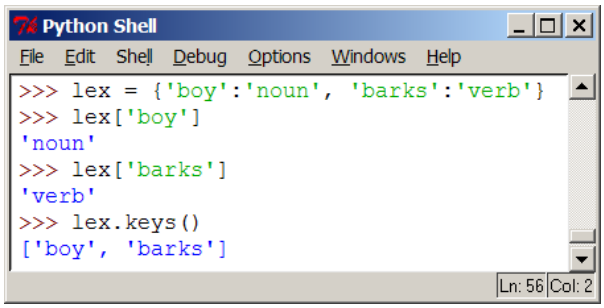
Erklären Sie, warum unser Shift-Reduce-Parser als 'eingeschränkt' bezeichnet wurde. Ändern Sie dafür Ihre Grammatik so, dass auch die folgenden Sätze erfasst werden können. Beginnen Sie dabei mit dem linken Satz des Satzpaares und versuchen Sie sich dann am rechten Satz. Was läuft dabei schief? Welche Eigenschaften natürlicher Sprache können mit unserem Parser nicht erfasst werden?

- *The girl in the garden slept* vs. *The girl in the garden behind the house slept*
- *The boy who slept dreamt* vs. *The boy who chased the girl jumped*
- *The boy chased her in the garden* vs. *The boy chased the girl in the garden*

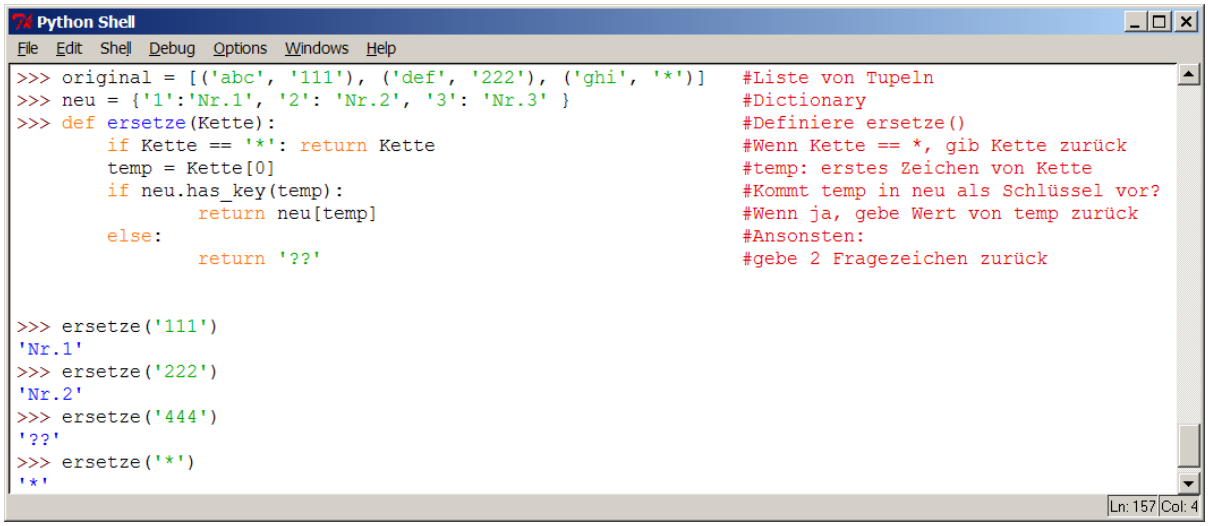
Appendix

Hilfestellung `für tag2tag(tag)`:

Die Elemente eines Tupels können wir, wie gesehen, durch die Indexfunktion referenzieren. Der Wert eines Dictionaryeintrages kann durch seinen jeweiligen Schlüssel angesprochen werden, einfach indem dieser als Index verwendet wird, wie der nebenstehende Screenshot zeigt. Über die Methode **`.keys()`** werden die Schlüssel als Liste ausgegeben.



Der nachstehende Screenshot zeigt, wie die oben Funktion aufgebaut sein kann und sich in der Anwendung verhält:



Hilfestellung `für replace_tag(tupel)`:

Wir verwenden hier die Indexmethode um auf das erste Element des Tupels, sprich das Wort, zuzugreifen und die Funktion **`tag2tag()`**, um das zweite Element des neuen Tupels zu ermitteln: