Tokenisierer[1]

1. Überblick

Tokenisierer (engl. *Tokenizers*) sind Programme, die Texte für die Weiterverarbeitung durch anderer Programme (z.B. *Tagger*, *Stemmer*, *Lemmatisierer* oder *Parser*) in Teilketten, sog. *Token* zerlegen. Was dabei als *Token* gelten soll, hängt von der jeweiligen Anwendung ab. Im Allgemeinen geht es darum, Text in Listen von Wortformen umzuwandeln. In einem ersten Schritt kann es auch darum gehen. Längere Texte zunächst in Sätze aufzuspalten.

Das NLTK-Paket stellt eine Reihe von Tokenisierern zur Verfügung, die alle eine Methode `tokenize()` zur Verfügung stellen, sich aber in den Parametern unterscheiden können. Ein Problem ist beispielsweise der Umgang mit Interpunktionszeichen. Für Satzanalyseprogramme (sog. *Parser*) können Satzzeichen wichtige Hinweise liefern, so dass sie als *Token* erhalten bleiben müssen. In anderen Fällen ist es sinnvoll, sie auszufiltern.

Der im Folgenden verwendete *Tokeniser* liefert Satzzeichen als separate *Token* zurück.[2]

```
>>> from nltk.tokenize import *
>>> tokenizer = WordPunctTokenizer()
>>> tokenizer.tokenize("She said 'hello'.")
['She', 'said', "'", 'hello', "'."]
```

Die Anweisung in der ersten Zeile setzt voraus, dass das NLTK bereits importiert worden ist, andernfalls müssen Sie das mit **`import nltk`** nachholen. Der Stern '*' in der Anweisung bedeutet, dass alle Mitglieder des Moduls `tokenize` importiert werden sollen. Dazu gehören u.a.

- **`PunktSentenceTokenizer`**
- **`PunktWordTokenizer`**
- **`RegexpTokenizer`**
- **`BlanklineTokenizer`**
- **`WordPunctTokenizer`**
- **`WordTokenizer`**
- **`LineTokenizer`**
- **`SpaceTokenizer`**
- **`TabTokenizer`**
- **`WhitespaceTokenizer`**

Viele der Methoden zur Tokenisierung stehen auch als eigenständige Funktionen zur Verfügung, z.B. **`wordpunct_tokenize()`**. Das spart einem die Mühe, jedes Mal ein Tokenizer-Objekt erzeugen zu müssen, wenn man eine Zeichenkette "tokenisieren" will. Beispiel:

```
>>> wordpunct_tokenize("She said 'hello'.")
['She', 'said', "'", 'hello', "'."]
```

Im Folgenden sollen die wichtigsten Verfahren etwas genauer unter die Lupe genommen werden.

---

[1] Dieser Text ist eine Bearbeitung der englischen Fassung eines "Guide" aus der NTLK-Dokumentation. Ich habe, wo ich es für angebracht hielt, sowohl Ergänzungen als auch Kürzungen vorgenommen.

[2] Sie sollten alle Beispiele im Text ausprobieren und auch darüber hinaus experimentieren.

2. Einfache Tokenisierer

Die folgenden Tokenisierer aus dem Modul **`nltk.tokenize.simple`** zerlegen eine Zeichenkette mit Hilfe der bereits bekannten und häufig eingesetzten für den Datentyp string definierten **`split()`** Methode. Diese kann also in allen Fällen ersatzweise verwendet werden.

Bei den verschiedenen Verfahren geht es u.a. um die Frage, welche Wortgrenzsignale zu Grunde gelegt werden. Im ersten Tokenisierer-Beispiel **`WhitespaceTokenizer`** ist dies, wie der Name schon sagt *Whitespace*, d.h. Leerraum. Dazu zählen normalerweise Leerzeichen, Tabulatorzeichen und Zeilenumbrüche. Im folgenden Beispiel wird der Zeilenumbruch durch die "Escape-Sequenz"[3] **`'\n'`** (= *newline*) dargestellt. Das Tabulatorzeichen wird durch **`'\t'`** ausgedrückt. Ein *Token* ist hier also das, was zwischen zwei *Whitespace*-Zeichen steht:

```
>>> s = ("Good muffins cost $3.88\nin New York.  Please buy me\n"
...      "two of them.\n\nThanks.")
```

Die der Variablen s zugewiesene Zeichenkette dient für die folgenden Beispiele als Testbeispiel. Durch die öffnende Klammer in **`s = ("Good`** .... ist es möglich, die Eingabe der Zeichenfolge auf mehrere Zeilen zu verteilen. Python fügt dies dann zu einer Zeichenkette zusammen.

```
>>> # identisch mit s.split():
>>> WhitespaceTokenizer().tokenize(s)
['Good', 'muffins', 'cost', '$3.88', 'in', 'New', 'York.',
 'Please', 'buy', 'me', 'two', 'of', 'them.', 'Thanks.']
```

Der *Whitespace* wird damit faktisch entfernt. Die String-Methode **`split()`** liefert ein identisches Ergebnis, was leicht nachzuvollziehen ist, wenn man weiß, dass die einfachen *Tokenizer*, um die es hier geht, intern die **`split()`**-Methode verwenden:

```
>>> s.split()
['Good', 'muffins', 'cost', '$3.88', 'in', 'New', 'York.',
 'Please', 'buy', 'me', 'two', 'of', 'them.', 'Thanks.']
```

Im nächsten Fall kommt der **`SpaceTokenizer`** zum Einsatz, bei dem nur das Leerzeichen als Trennzeichen verwendet wird. Zeilenumbrüche und Tabulatoren bleiben erhalten. Verwendet man die String-Methode **`split()`**, muss das Leerzeichen explizit angegeben werden:

```
>>> # identisch mit s.split(' '):
>>> SpaceTokenizer().tokenize(s)
['Good', 'muffins', 'cost', '$3.88\nin', 'New', 'York.', '',
 'Please', 'buy', 'me\ntwo', 'of', 'them.\n\nThanks.']
```

In den beiden nachstehenden Beispielen mit dem **`LineTokenizer`** wird als Trennparameter das *newline*-Zeichen **`\n`** verwendet. Dabei sind zwei Anwendungsfälle zu unterscheiden, je nachdem, ob bei zwei unmittelbar aufeinanderfolgende **`\n`**-Zeichen ein "leeres" *Token* (= **`''`**) erzeugt werden soll, oder nicht. Dies wird durch einen Parameter **`blanklines`** mit den

[3] Eine "Escape-Sequenz" ist eine Zeichenfolge, die mit einem "Escape"-Zeichen beginnt, das ausdrückt, dass das darauf folgende Zeichen (oder die folgende Zeichenkette) nicht als normale Zeichen stehen, sondern eine besondere Funktion hat. In diesem Kontext ist das Escapezeichen der sog. "backslash" **`'\'`**. Beispiele für solche Sequenzen: **`\t`** (t < tab, ASCII Nr. 9), **`\r`** (r < return [Wagenrücklauf], ASCII Nr. 13), **`\n`** (n < newline, ASCII Nr. 10), etc. Wenn der "backslash" selbst als Zeichen verwendet werden soll, muss er "maskiert" werden, d.h. selbst als Escapefolge **`'\\'`** geschrieben werden.

Werten **'keep'** (Leerzeilen bleiben erhalten) bzw. **'discard'** (Leerzeilen werden entfernt) festgelegt, der an den Tokenizer übergeben wird.

Beispiel mit **blanklines ='keep'**

```
>>> # identisch mit s.split('\n'):
>>> LineTokenizer(blanklines='keep').tokenize(s)
['Good muffins cost $3.88', 'in New York.  Please buy me',
 'two of them.', '', 'Thanks.']
```

Leeres Token

Beispiel mit **blanklines ='discard'**

```
>>> #identisch mit [z for z in s.split('\n') if z.strip()]:
>>> LineTokenizer(blanklines='discard').tokenize(s)
['Good muffins cost $3.88', 'in New York.  Please buy me',
 'two of them.', 'Thanks.']
```

Der Ausdruck in der Kommentarzeile des vorstehenden Beispieles ist für Python-Novizen wohl erklärungsbedürftig: **[z for z in s.split('\n') if z.strip()]**. Es handelt es sich hierbei um ein für die Python-Programmierung außerordentlich wichtiges und allgegenwärtiges Programmkonstrukt, das *list comprehension* (dt. Listen-Abstraktion) genannt wird. Genauere Informationen dazu finden Sie unter: http://www.fb10.uni-bremen.de/homepages/hackmack/clst1/lc.pdf.

Als letztes bleibt nun noch der **TabTokenizer**, bei dem als Trennindikator das Tabulator-Zeichen verwendet wird:

```
>>> # identisch mit s.split('\t'):
>>> TabTokenizer().tokenize('a\tb c\n\t d')
['a', 'b c\n', ' d']
```

Für die Tokenisierer des Moduls **nltk.tokenize.simple** stehen keine separaten Funktionen zur Verfügung. Statt dessen sollte hier die String-Methode **split()**direkt eingesetzt werden:

```
>>> s.split()
['Good', 'muffins', 'cost', '$3.88', 'in', 'New', 'York.',
 'Please', 'buy', 'me', 'two', 'of', 'them.', 'Thanks.']
>>> s.split(' ')
['Good', 'muffins', 'cost', '$3.88\nin', 'New', 'York.',
'',
 'Please', 'buy', 'me\ntwo', 'of', 'them.\n\nThanks.']
>>> s.split('\n')
['Good muffins cost $3.88', 'in New York.  Please buy me',
 'two of them.', '', 'Thanks.']
```

3. Tokenisierung mit Regulären Ausdrücken

Wir haben gesehen, dass es für den **WhitespaceTokenizer** aus dem Modul **nltk.tokenize.simple** für die Trennzeichen **' '**, **'\t'** und **'\n'** Spezialisierung gibt (**SpaceTokenizer, LineTokenizer, TabTokenizer**). Viel flexibler wäre es natürlich, wenn man je nach Bedarf ein Trennzeichen oder beliebige Kombinationen davon

vorgeben könnte, eingeschlossen Interpunktionszeichen. Hier kommen sog. **reguläre Ausdrücke** ins Spiel und ein *Tokenizer*, der darauf spezialisiert ist, der **`RegexpTokenizer`**. Ausführliche Erläuterungen zu den regulären Ausdrücken finden Sie unter: http://www.fb10.uni-bremen.de/homepages/hackmack/clst1/ra.pdf.

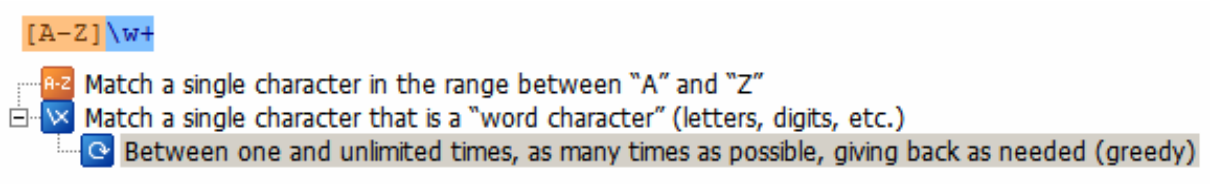

Abbildung 1: Erklärung des Ausdrucks '[A-Z]\w+' mit RegexBuddy

Wenn man einem *RegexpTokenizer* nur einen regulären Ausdruck als Parameter übergibt, liefert er eine Liste der Teilketten zurück, die mit dem regulären Ausdruck in Einklang stehen. Der folgende *Tokenizer* beispielsweise wählt nur Wörter mit großem Anfangsbuchstaben aus und filtert alles Andere aus:

```
>>> capword_tokenizer = RegexpTokenizer('[A-Z]\w+')
>>> capword_tokenizer.tokenize(s)
['Good', 'New', 'York', 'Please', 'Thanks']
```

Der RA **`'\w+|\$[\d\.]+|\S+'`** fasst 3 alternative RAs zusammen, was durch den Operator **`'|'`** ausgedrückt wird. Das entspricht dem nebenstehenden Klammerausdruck. Die Zeichen **`'$'`** und **`'.'`** (Punkt) haben einen Sonderstatus: **`'$'`** markiert das Zeilenende und der Punkt **`'.'`** steht für ein beliebiges Zeichen aus dem Zeichenvorrat. Wenn sie selbst als Zeichen verwendet werden sollen, müssen sie maskiert werden: **`\.`** und **`\$`**. **`\S`** ist das Komplement von **`\s`**, d.h. **`\S = [^\s]`** (alles was kein *Whitespace* ist). Die genaue Beschreibung findet sich in **Abbildung 2.** Diese Alternativen werden dem folgenden **`RegexpTokenizer`**-Beispiel als Parameter übergeben.

```
\w+
\$[\d\.]+
\S+
```

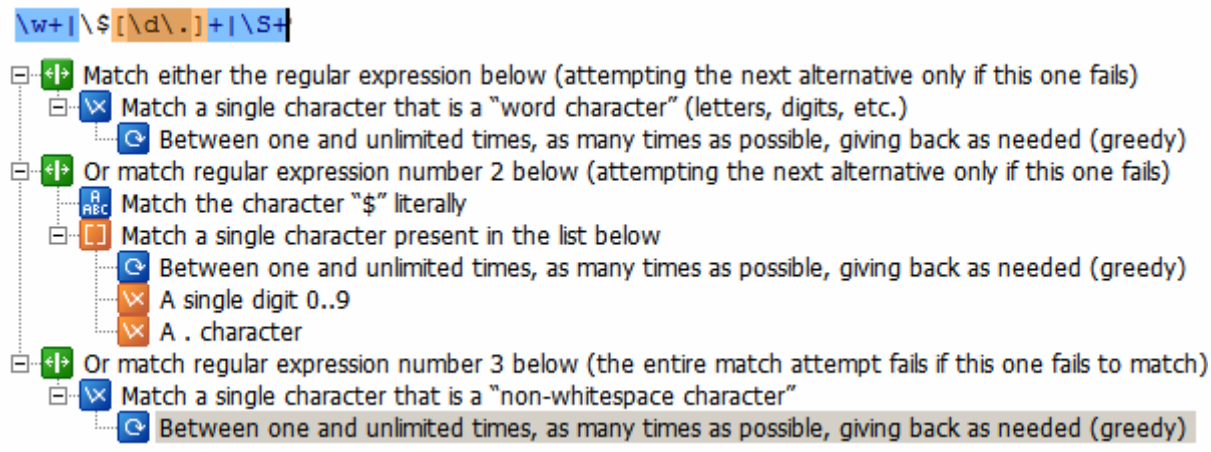

Abbildung 2: Erklärung des RA `\w+ | \$[\d\.]+ | \S+`

Der folgende Tokenisierer bildet *Token*, die aus alphabetischen Zeichen bestehen, Währungsausdrücke und anderen druckbaren Zeichen (z.B. Interpunktions-Zeichen).

```
>>> tokenizer = RegexpTokenizer('\w+|\$[\d\.]+|\S+')
>>> tokenizer.tokenize(s)
['Good', 'muffins', 'cost', '$3.88', 'in', 'New', 'York', '.',
 'Please', 'buy', 'me', 'two', 'of', 'them', '.', 'Thanks', '.']
```

Sollen die regulären Ausdrücke als Separatoren (Trennzeichen) verwendet werden, muss dies dem **RegexpTokenizer** mit dem Ausdruck **gaps=True** als zusätzlichem Parameter mitgeteilt werden. Die Escape-Sequenz **'\s'** steht für die Klasse der *Whitespace*-Zeichen (Leerzeichen, Tabulatorzeichen, *newline*-Zeichen (**\r** und/oder **\n**).

```
>>> tokenizer = RegexpTokenizer('\s+', gaps=True)
>>> tokenizer.tokenize(s)
['Good', 'muffins', 'cost', '$3.88', 'in', 'New', 'York.',
 'Please', 'buy', 'me', 'two', 'of', 'them.', 'Thanks.']
```

Das Modul *nltk.tokenize.regexp* enthält eine Reihe von Subklassen die von **RegexpTokenizer** abgeleitet sind und vordefinierte reguläre Ausdrücke verwenden: **WordTokenizer**, **WordPunctTokenizer**, **BlanklineTokenizer**.

```
>>> # Verwendet '\w+':
>>> WordTokenizer().tokenize(s)
['Good', 'muffins', 'cost', '3', '88', 'in', 'New', 'York',
'Please', 'buy', 'me', 'two', 'of', 'them', 'Thanks']
```

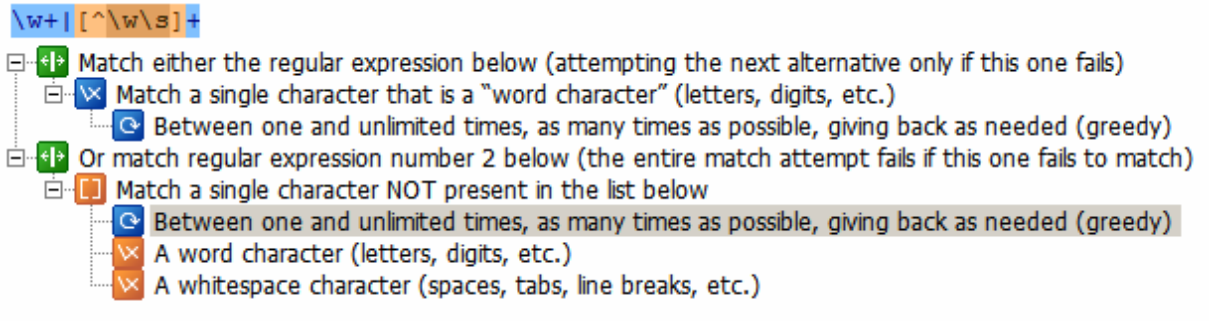

Abbildung 3: Beschreibung des RA \w+ | [^\w\s]+

```
>>> # Verwendet '\w+|[^\w\s]+':
>>> WordPunctTokenizer().tokenize(s)
['Good', 'muffins', 'cost', '$', '3', '.', '88', 'in',
'New', 'York', '.', 'Please', 'buy', 'me', 'two', 'of',
'them', '.', 'Thanks', '.']
```

Abbildung 4: Beschreibung des RA \s*\n\s*\n\s*

```
>>> # Verwendet '\s*\n\s*\n\s*':
>>> BlanklineTokenizer().tokenize(s)
['Good muffins cost $3.88\nin New York.  Please buy me\ntwo
of them.', 'Thanks.']
```

Alle *Tokenizer* der Klasse **RegexpTokenizer** sind auch als einfache Funktionen verfügbar:

```
>>> regexp_tokenize(s, pattern='\w+|\$[\d\.]+|\S+')
['Good', 'muffins', 'cost', '$3.88', 'in', 'New', 'York',
'.', 'Please', 'buy', 'me', 'two', 'of', 'them', '.',
'Thanks', '.']
```

```
>>> word_tokenize(s)
['Good', 'muffins', 'cost', '3', '88', 'in', 'New', 'York',
'Please', 'buy', 'me', 'two', 'of', 'them', 'Thanks']
```

```
>>> wordpunct_tokenize(s)
['Good', 'muffins', 'cost', '$', '3', '.', '88', 'in',
'New', 'York', '.', 'Please', 'buy', 'me', 'two', 'of',
'them', '.', 'Thanks', '.']
```

```
>>> blankline_tokenize(s)
['Good muffins cost $3.88\nin New York.  Please buy me\ntwo
of them.', 'Thanks.']
```

4. Punkt Tokenizer

In der Einleitung wurde schon darauf hingewiesen, dass Satzzeichen wichtige Hinweise liefern können, z.B für Satzanalyseprogramme (sog. *Parser*). Für diese Einsatzgebiete müssen sie daher als *Token* erhalten bleiben. Der **`PunktSentenceTokenizer`** ist in der Lage, verschiedene Funktionen von Punkten zu unterscheiden und Abkürzungen und Satzendemarkierungen zu entdecken. Dieser *Tokenizer* muss jedoch mit den Daten eines hinreichend großen Textkorpus trainiert werden, bevor er verwendet werden kann.[4]

[4] Der Algorithmus für diesen *Tokenizer* wird beschrieben in: KISS, TIBOR and STRUNK, JAN (2006): Unsupervised Multilingual Sentence Boundary Detection. *Computational Linguistics* **32**: 485–525.

Das *NLTK data package* enthält einen vortrainierten **`PunktTokenizer`** für Englisch.

```
>>> import nltk.data
>>> text = """
... Punkt knows that the periods in Mr. Smith and Johann S. Bach
... do not mark sentence boundaries.  And sometimes sentences
... can start with non-capitalized words.  i is a good variable
... name.
... """
>>> tokenizer =
nltk.data.load('tokenizers/punkt/english.pickle')
>>> print '\n-----\n'.join(tokenizer.tokenize(text.strip()))
Punkt knows that the periods in Mr. Smith and Johann S. Bach
do not mark sentence boundaries.
-----
And sometimes sentences
can start with non-capitalized words.
-----
i is a good variable
name.
```

(Man beachte, dass der Leerraum des Originaltextes, inklusive Zeilenumbrüchen, im Output erhalten bleibt.)