Informatik I

20. Reguläre Ausdrücke

Bernhard Nebel

Albert-Ludwigs-Universität Freiburg

07.01.2014

Informatik I

07.01.2014 — 20. Reguläre Ausdrücke

20.1 Motivation

Reguläre Ausdrücke - wozu?

- Ein Großteil von (Skript-)Programmierung besteht darin, Textdateien zu überprüfen und/oder zu analysieren.
- Zum Beispiel:
  - feststellen, ob ein String eine gültige Telefonnummer oder eine gültige E-Mail-Adresse ist;
  - bestimmte Muster in Texten finden, die vermutlich fehlerhaft sind („der der“);
  - aus einer HTML-Seite alle Links extrahieren.
- Wir können dies mit den String-Methoden und Einzelzugriffen auf Strings realisieren . . . aber das wird ziemlich schnell ziemlich umständlich.

Beispiel: E-Mail-Adresse

- Eine E-Mail-Adresse besteht aus
  - dem lokalen Teil,
  - gefolgt vom „@“-Zeichen,
  - gefolgt vom Domänenteil.
- Der lokale Teil darf alle Zeichen außer „@“ und Leerzeichen enthalten.
- Der Domänenteil darf nur die Zeichen „A“–„Z“, „a“–„z“, „0“–„9“, „-“ und„.“ enthalten und nicht auf einem Punkt oder Strich enden und muss mindestens einen Punkt enthalten.
  - OK: `nebel@uni-freiburg.de`, `m!'',mustermann@gmx.t`
  - Nicht OK: `b.nebel@uni`, `m@mustermann@gmx.de`
- Tatsächlich ist das eine grobe Vereinfachung!

Eine E-Mail-Adresse testen

email.py

```
def emailIsValid(str):
    i=0; dot = False
    while (i<len(str) and not str[i].isspace() and
           not str[i] == "@"):
        i +=1
    if i==len(str) or str[i] != '@': return False
    i += 1
    while i<len(str) and (str[i].isalnum() or
                          str[i] == "-" or str[i] == "."):
        if str[i] == ".": dot = True
        i += 1
    if i != len(str) or nor dot: return False
    if str[i - 1] == "-" or str[i - 1] == ".":
        return False
    return True
```

E-Mail-Adresse testen – kompakt

email.py

```
import re

def emailIsValidRe(str):
    return re.match(
      r'[^@\s]+@[0-9a-z\-]*\.[.0-9a-z\-]*[0-9a-z]$',
      str,re.I+re.A) != None
```

- Diese Funktion macht dasselbe, wie die auf der letzten Folie,
- is aber sehr viel kompakter beschrieben.

→ Einsatz von regulären Ausdrücken

- Ein regulärer Ausdruck beschreibt eine Menge von Strings:
- Wir beschreiben nicht, wie wir den String analysieren, sondern was für eine Gestalt er haben soll.

20.2 Das Modul re

Das re-Modul: Match-Funktionen

- `re.match(pattern, string, flags=0)`:
  Prüft, ob `pattern` auf ein Anfangsstück von `string` passt – ob das Pattern den String matcht. Dabei ist `flags` optional. Ergebnis ist `None` wenn nicht erfolgreich, sonst ein *Match*-Objekt.
- `re.search(pattern, string, flags=0)`:
  Wie `match`, aber es wird innerhalb von `string` gesucht, statt nur den Anfang zu testen.
- `re.findall(pattern, string, flags=0)`:
  Wie `search`, aber liefert eine Liste mit allen nicht-überlappenden in String gematchten Teilstrings (oder Tupeln mit *Gruppenzugehörigkeiten*).
- `re.finditer(pattern, string, flags=0)`:
  Wie `findall`, aber liefert einen Iterator, der alle *Match*-Objekte erzeugen kann.

Das re-Modul: Modifikations-Operationen

- `re.split(pattern, string, flags=0)`:
  Zerlegt `string` an den Stellen, an denen es eine Übereinstimmung mit `pattern` gibt (u.U. noch mehr Resultate, wenn Gruppen vorhanden).
- `re.sub(pattern, repl, string, count=0, flags=0)`:
  Ersetzt innerhalb von `string` alle *Matches* durch `repl`, wobei das ein String oder ein Funktionsobjekt sein kann, das das *Match*-Objekt als Parameter nehmen muss. Der optionale Parameter `count` begrenzt die Anzahl der Ersetzungen.
- `re.subn(pattern, repl, string, count=0, flags=0)`:
  Wie `sub`, aber es wird ein Tupel aus Anzahl und neuem String zurück gegeben.

20.3 Die Syntax regulärer Ausdrücke

Ein String ist ein regulärer Ausdruck

- Jeder String ist ein regulärer Ausdruck, der genau den String matcht, der mit ihm identisch ist.

Python-Interpreter

```
>>> re.match('aba', 'abababababa')
<_sre.SRE_Match object at 0x10e29f920>
>>> re.findall('aba', 'abababababa')
['aba', 'aba'] # nicht überlappende Strings!
```

- Da das Zeichen „\“ häufig zum Einsatz kommt, gibt man reguläre Ausdrücke besser mit Hilfe von rohen Strings an, d.h. `r'string'`. Ansonsten müsste man immer „\\“ schreiben, wenn man einen Rückstrich meint.

Verallgemeinerung: Beliebige Zeichen

- Um ein beliebiges Zeichen zu matchen (in Übereinstimmung zu bringen), kann man den Punkt benutzen. Meint man tatsächlich den Punkt, muss man „\." nutzen.
- Wie könnte man die bestimmten Artikel aus einem Text fischen?

Python-Interpreter

```
>>> re.findall(r'd..','der Hund, die Katze, damit')
['der', 'd, ', 'die', 'dam']
>>> re.findall(r'd.. ','der Hund, die Katze,damit')
['der ', 'die ']
>>> re.findall(r'.....\.', 'der Hund, die Katze.')
['Katze.']
```

- Beachte: Normalerweise matcht der Punkt nicht das Newline-Zeichen \n, außer das DOTALL-Flag ist gesetzt (kommt noch . . .).

Verallgemeinerung: Zeichenklassen

- Ein Buchstabe matcht genau den Buchstaben, ein Punkt alles. Man bräuchte etwas dazwischen.
- Beispiel: Wir wollen ein Datum wie „12.11.1998“ matchen.
- Um eine Menge von möglichen Zeichen zu matchen, kann man diese in eckigen Klammern aufzählen:

Python-Interpreter

```
>>> re.findall(r'[0123][0123456789]\.','12.11.1998')
['12.', '11.']
```

- Einen Bereich von Zeichen kann man mit dem Bindestrich aufzählen.

Python-Interpreter

```
>>> re.findall(r'[0-3][6-950-4]\.', '31.12.1998 ')
['31.', '12.']
```

Sonderzeichen in Zeichenklassen

- Der Punkt „.“ ist kein Sonderzeichen in Zeichenklassen.
- „\“ ist das Quotierungszeichen, mit dem man andere Sonderzeichen (z.B. „-“ und „\“) präfigiert, um diese in die Zeichenklasse aufzunehmen, z.B. bezeichnet `[.\-\\]` die Klasse bestehend aus Punkt, Minus und Rückstrich.
- „-“ ist das Bereichssonderzeichen, das aber am Anfang oder Ende einer Zeichenklassenbeschreibung kein Quotierungszeichen benötigt.
- „]“ ist das Abschlusszeichen für Zeichenklassenbeschreibungen, das am *Anfang* keine Quotierung benötigt, d.h. sowohl `[]\]{}()]` als auch `[][{}()]` beschreiben die Klasse aller Klammern.
- „^“ ist das Komplementzeichen, wenn es das erste Zeichen ist. An allen anderen Stellen wird es als normales Zeichen verstanden. Z.B. ist `[^[\]{}()^]` die Klasse aller Zeichen, die keine Klammern oder Hochpfeile sind.

Vordefinierte Zeichenklassen

Es gibt eine Menge von vordefinierten Zeichenklassen, die auch wiederum in Zeichenklassenbeschreibungen auftauchen können.

- \d matcht alle Ziffern, d.h. im ASCII-Fall `0-9`. Normalerweise aber (Unicode) auch alle Ziffern in anderen Schriftsystemen.
- \D matcht alles, was keine Ziffer ist, d.h. ist äquivalent zu `[^\d]`.
- \s matcht alle Weißraum-Zeichen, d.h. ist im ASCII-Fall äquivalent zu `[ \t\n\r\f\v]`. Bei Unicode können weitere Zeichen hinzukommen.
- \S ist äquivalent zu `[^\s]`.
- \w ist im ASCII-Fall `[a-zA-Z0-9_]`, im Unicode-Fall kommen alle Buchstaben und Ziffern aus anderen Schriftsystemen dazu.
- \W ist äquivalent zu `[^\w]`.

Den leeren String an bestimmten Positionen matchen

Normalerweise wollen wir immer nicht-leere Strings matchen. Aber hin- und wieder kann es sinnvoll sein, den leeren String an bestimmten Positionen zu matchen. Z.B., wenn wir alle Worte mit drei Buchstaben, die mit einem Großbuchstaben beginnen, matchen wollen.

- \A passt nur am Anfang des Strings, d.h. `re.match(regex, string)` ist äquivalent zu `re.search(r'\A' + regex, string)`.
- \b passt nur vor und nach jedem Wort (bestehend aus \w-Zeichen D.h. `r'\b[A-Z]\w\w\b'` würde das oben gewünscht matchen.
- \B matcht nur dann, wenn \b nicht matcht.
- \Z passt nur am Ende des Strings.
- ^ passt wie \A nur am Anfang eines Strings. Wenn `MULTILINE`-Flag gesetzt, passt es an jedem Zeilenanfang.
- $ passt am Ende des Strings, bzw. an jedem Zeilenende, wenn MULTILINE-Flag gesetzt.

Gruppenbildung und Alternativen

Beim Prüfen des Datums wollen wir zulassen, dass Monat und Tag ein- und zweistellig sein dürfen. Dafür gibt es den Alternativ-Operator „|“. Dieser Operator bindet weniger stark als die Konkatenation. Runde Klammern können zur Gruppenbildung eingesetzt werden. Will man runde Klammer matchen, muss man das Quotierungszeichen voran stellen.

Python-Interpreter

```
>>> re.match(r'P(ython|eter)', 'Peter')
<_sre.SRE_Match object at 0x12a77fcf0>
>>> r = r'([0-3][0-9]|[1-9])\.([0-1][0-9]|[1-9])\.'
>>> re.match(r, '31.1.')
<_sre.SRE_Match object at 0x10e11fcf0>
>>> re.match(r, '41.1.')
None
>>> re.match(r, '31.0.')
None
```

Einfache Quantoren

Oft soll eine Gruppe *wiederholt* werden oder *optional* sein. Dafür gibt es die Quantoren „?“, „*“ und „+“, die einem Zeichen oder einer Gruppe nachgestellt werden – und die stärker als Konkatenation binden.

- „?“: Das vorangegangene Zeichen oder die vorangegangene Gruppe ist optional. Unseren Beispielausdruck fürs Datum von der letzten Folie könnten wir damit so schreiben: `r'[0-3]?[0-9]\.[0-1]?[0-9]\.'` Allerdings gibt es einen kleinen Unterschied zur letzten Folie!
- „*“: Das Zeichen oder die Gruppe kann beliebig oft (auch 0-mal) wiederholt werden.
- „+“: Das Zeichen oder die Gruppe kann beliebig oft (aber mindestens einmal) wiederholt werden.

→ Damit kann man das Format für deutsche Telefonnummern wie folgt beschreiben: `r'(0\d+/)?\d+'`

Zählquantoren

Oft kann man genauer sagen, wie oft etwas wiederholt werden darf.
Beispiel PLZ: 5-stellig. Datum 1-2-stellig, bzw. 4-stellig.

- Mit {*i*} spezifiziert man, dass die voran gegangene Gruppe (bzw. Zeichen) genau *i*-mal wiederholt werden soll.
- Mit {*min*, *max*} gibt man an, dass die Gruppe oder das Zeichen zwischen *min* und *max* mal wiederholt werden soll.
- Mit { , *max*} gibt man eine Obergrenze an.
- Mit {*min* , } gibt man eine untere Grenze an.

Python-Interpreter

```
>>> re.match(r'^\d{5}$', '78110')
<_sre.SRE_Match object at 0x55e11ffff>
>>> re.match(r'^(\d{1,2}\.){2}\d{4}$', '1.1.201')
None
```

Die E-Mail-Adresse nochmal

Mittlerweile können wir den regulären Ausdruck aus unserem Eingangsbeispiel völlig problemlos verstehen:

```
r'[^@\s]+@[0-9a-z\-]*\.[.0-9a-z\-]*[0-9a-z]$'
```

1. Die Zeichenklasse, die alle Zeichen außer Leerzeichen und @ enthält
2. Beliebige Wiederholung, aber mindestens einmal
3. Das @-Zeichen
4. Ziffern, Kleinbuchstaben, Minus-Zeichen beliebig oft
5. Ein Punkt
6. Beliebig viele Kleinbuchstaben, Ziffern, Minus-Zeichen und Punkte
7. Zum Abschluss muss ein Buchstabe oder eine Ziffer stehen.
8. Hier muss der String zu Ende sein.

Gruppen und Rückbeziehung

Jedes Mal, wenn wir eine Gruppe mit den runden Klammern bilden, wird der damit gematchte Teilstring referenzierbar gemacht. Auf diesen Teilstring kann man dann mit der Notation $\backslash n$ zugreifen, wobei n eine Zahl zwischen 1 und 99 entsprechend der Stellung der Gruppe im regulären Ausdruck ist (Position der öffnenden Klammer zählt!).

- Wir wollen alle Ziffernfolgen finden, die mit der gleichen Zahl anfangen und enden.

Python-Interpreter

```
>>> f='129 337873 78324 43938 9388 824998 349734'
>>> re.findall(r'\b((\d)\d*\2)\b', f)
[('337873', '3'), ('824998', '8')]
```

- Gibt es Gruppen im regulären Ausdruck, dann werden von `findall` statt dem gematchten Text die von den Gruppen gematchten Teilstrings in Tupeln zurück gegeben!

Gieriges Matchen

- Der *Matcher* versucht immer, möglichst viel im String zu überdecken. Man nennt das auch gieriges Matching (*greedy matching*).
- Wenn man nur wissen will, ob der reguläre Ausdruck den gegeben String matcht, ist das einerlei.
- Will man alle möglichen Matches in einem String finden, kann das problematisch sein.
- Beispiel: Sei der folgende Text Teil einer HTML-Datei:

  ```
  Bla <a href="http://www.ex.de/" target="_blank">
  Linktext</a> und <a href="dat.txt">mehr</a>
  ```

  Wir wollen die URLs und den Linktexte extrahieren. Sei der Text in der Variablen `html` gespeichert.

Gieriges und genügsames Matchen

- Ein regulärer Ausdruck zum Extrahieren von URLs und Linktexten könnte so aussehen:

Python-Interpreter

```
>>> re.findall(r'<a href="(.*)".*>(.*)</a>', html)
[('http://www.ex.de/" target="_blank">Linktext</a> und <a
href="dat.txt', 'mehr')]
```

- Da ist was schief gegangen!
- Es gibt für alle Quantoren eine genügsame (*non-greedy*) Version, die einen möglichst kurzen Teilstring versucht zu überdecken: Nachgestelltes „?“ führt zu genügsamen Matchen.

Python-Interpreter

```
>>> re.findall(r'<a href="(.*?)".*?>(.*?)</a>', html)
[('http://www.ex.de/', 'Linktext'), ('dat.txt', 'mehr')]
```

Erweiterungen

- Damit kennen wir die Syntax der regulären Ausdrücke im Wesentlichen.
- Es gibt einige so genannte Erweiterungen, die alle die Gestalt `(? ...)` haben.
- Die meisten matchen den leeren String — und haben nur bestimmte Seiteneffekte, einige matchen aber bestimmte Dinge.

Erweiterungen I: Flags

- `(?aiLmsux)`:
  Diese Erweiterung erlaubt es, eine oder mehrere Flags für den gesamten Ausdruck zu setzen (alternativ können die beim Aufruf der Match-Funktion angegeben werden):
  - a: `re.A` – ASCII Matching
  - i: `re.I` – ignoriere Groß-/Kleinschreibung
  - L: `re.L` – *Locale*, vordefinierte Zeichenklassen werden von aktueller Lokalisierung abhängig gemacht – vom Gebrauch wird abgeraten!
  - m: `re.M` – Multi-Zeilen Matching (betrifft $ und ^)
  - s: `re.S` – Der Punkt matcht auch `\n`
  - u: `re.U` – nur für Rückwärtskompatibilität, da auf allen Unicode-Strings per Default mit Unicode-Matching gearbeitet wird.
  - x: re.X – Weißraum-Zeichen im regulären Ausdruck werden ignoriert, wenn sie nicht mit einem \ eingeleitet werden. Nach # kann man kommentieren.

Erweiterungen I: Flags – Beispiele

Python-Interpreter

```
>>> re.match(r'''(?xai)
... [^@\s]* # der lokale Teile
... @ # das at-Zeichen
... [\w-]*\.[.\w-]*\w # der Domänenteil
...  ''','ABX@GMX.DE')
<_sre.SRE_Match object at 0x10c55d9f0> >>>
re.findall(r'\b\w+\b','Das ist die Hölle')
['Das', 'ist', 'die', 'Hölle']
>>> re.findall(r'(?a)\b\w+\b','Das ist die Hölle')
['Das', 'ist', 'die', 'H', 'lle']
```

Erweiterungen II: Anonyme Gruppen und Gruppennamen

- `(?:...)`:
  Hiermit wird eine „anonyme“ Gruppe gebildet, auf die kein Bezug genommen werden kann!
- `(?P<`*name*`>...)`:
  Hiermit wird eine benannte Gruppe erzeugt, auf die man sich mit dem Namen *name* beziehen kann.
- `(?P=`*name*`)`:
  Dies ist das Gegenstück, mit dem man sich auf die benannte Gruppe beziehen kann.

Python-Interpreter

```
>>> f='129 337873 78324 43938 9388 824998 349734'
>>> re.findall(r'\b((?P<digit>\d)\d*(?P=digit))\b', f)
[('337873', '3'), ('824998', '8')]
```

Erweiterungen II: Kommentare

- `(?#...)`:
  Alles in der Klammer nach dem #-Zeichen ist Kommentar!

Python-Interpreter

```
>>> f='Abba hat es aber'
>>> r=r'(?i)(?#Nur ein Beispiel)ab.'
>>> re.findall(r, f)
['Abb', 'abe']
```

Erweiterungen III: Positive Rück- und Vorausschau (1)

- `(?=...)`:
  matcht, falls der Ausdruck ... als nächstes matcht! Dabei wird dieser aber nicht konsumiert, sondern kann wieder verwendet werden. D.h. wir schauen voraus. Man redet von positiver Vorausschau (*positive lookahead*).
- `(?<=...)`:
  matcht, falls der Ausdruck ... den String vor der aktuellen Position matcht! Dabei wird dieser aber nicht konsumiert, sondern kann wieder verwendet werden. D.h. wir schauen zurück. Man redet von positiver Rückschau (*positive lookbehind*). Achtung: Der Ausdruck muss Strings fester Länge beschreiben!

Erweiterungen III: Positive Rück- und Vorausschau (2)

- **Beispiel**: Wir wollen in einer Ziffernreihe alle dreistelligen Zahlen, die keine Nullen enthalten, aber durch Nullen eingefasst sind, finden.

Python-Interpreter

```
>>> f='00056640358000034505005230157090604009280 0'
>>> r=r'(?<=0)[1-9]{3}(?=0)'
>>> re.findall(r, f)
['358', '345', '523', '157', '928']
```

- Beachte: 523 und 157 hätte man nicht als Ergebnis erhalten können, wenn die 0 Bestandteil des Patterns gewesen wäre.

Erweiterungen IV: Negative Rück- und Vorausschau

- `(?!...)`:
  matcht, falls der Ausdruck ... als nächstes nicht matcht! Dabei wird dieser aber nicht konsumiert.
- `(?<!...)`:
  matcht, falls der Ausdruck ... den String vor der aktuellen Position nicht matcht! Dabei wird dieser aber nicht konsumiert. Auch hier muss der Ausdruck Strings einer festen Länge beschreiben!
- **Beispiel erweitert**: keine zwei Nullen vorher oder hinterher.

Python-Interpreter

```
>>> f='00056640358000034505005230157090604009280 0'
>>> r=r'(?<=0)(?<!00)[1-9]{3}(?!00)(?=0)'
>>> re.findall(r, f)
['157']
```

Erweiterungen V: Konditionale Matches

- *(?(id/name)yes-pattern|no-pattern)*: Falls die Gruppe mit dem angegeben Index oder Namen einen Wert erhalten hat, wird der erste Teil zum Matchen benutzt, sonst der zweite Teil (der optional ist).

Python-Interpreter

```
>>> r = r'(?P<klammer>\()?Python(?(klammer)\))$'
>>> re.match(r, 'Python') != None
True
>>> re.match(r, '(Python)') != None
True
>>> re.match(r, '(Python') != None
False
```

- Generell werden konditionale Pattern aber als schwierig zu lesen und eher überflüssig angesehen.

Debuggen von regulären Ausdrücken

- Wenn Matches nicht erfolgreich sind oder aus den falschen Gründen erfolgreich sind, ist es oft schwierig, die Ursachen dafür zu verstehen.
- Eine Vorgehensweise ist: Erst einmal kleinere reguläre Ausdrücke zum Matchen benutzen und dann zusammen setzen.
- Alternativ: Einen Online-Debugger/Interpreter nutzen, z.B. `https://www.debuggex.com/`
  - Gibt eine Visualisierung des Ausdrucks als Zustandsmaschine (ein nicht-deterministischer endlicher Automat)
  - Erlaubt Anfangspunkt zu wählen und visualisiert dann die verschiedenen möglichen Punkte.
  - Enthält einen *Cheatsheet* für reguläre Ausdrücke.

Reguläre Ausdrücke und reguläre Sprachen

- Reguläre Ausdrücke werden auch in vielen anderen Systemen mit ganz ähnlicher Syntax eingesetzt.
- In der *Theoretischen Informatik* werden auch reguläre Ausdrücke und die von ihnen erzeugten regulären Sprachen untersucht.
- Dort bestehen reguläre Ausdrücke aber nur aus einzelnen Zeichen, Alternativ-Operator, Gruppierung und den Quantoren * und +.
- Die meisten zusätzlichen Ausdrucksmittel in Sprachen wie Python sind „syntaktischer Zucker“.
  - Allerdings erlauben die Zählquantoren eine exponentiell kürzere Darstellung.
  - Der Rückbezug auf Gruppen durch $\backslash n$ geht dann echt über die Ausdrucksfähigkeit normaler regulärer Ausdrücke hinaus.

20.4 Weiteres zur re-Schnittstelle

Kompilieren von regulären Ausdrücken

- Bisher haben wir den regulären Ausdruck direkt als String beim Aufruf der Match-Funktion angegeben.
- Man kann, wenn der reguläre Ausdruck öfter verwendet werden soll, diesen kompilieren und dann das entsprechende Regular-Expression-Objekt benutzen.
- `re.compile(pattern, flags=0)`

Python-Interpreter

```
>>> rx = re.compile(r'^(\d{1,2}\.){2}\d{4}$')
>>> rx
<_sre.SRE_Pattern object at 0x10c566180>
>>> rx.match('1.12.2014')
<_sre.SRE_Match object at 0x10c561288>
```

- Die Regular-Expression-Objekte besitzen Methoden entsprechend zu den re-Methoden, d.h. `match`, `search`, usw.

Match-Objekte

- Die Match-Objekte besitzen eine Menge von Methoden und Attributen, die den Match genauer beschreiben. Einige werden jetzt aufgeführt. Sei `m` ein Match-Objekt:
  - `m.group(*groups)`:
    Gibt die gematchten Teilstrings für die angegebenen Gruppen zurück. Gruppe 0 ist der gesamte gematchte String.
  - `m.groups(default=None)`:
    Gibt alle Teilstrings aller Gruppen zurück, wobei leere (nicht gematchte) Gruppen den Defaultwert erhalten.
  - `m.groupdict(default=None)`:
    Gibt ein dict mit allen benannten Gruppen zurück.
  - `m.start()`:
    Anfangsindex des Matches.
  - `m.end()`:
    Endindex des Matches.
  - `m.span()`:
    = (m.start(), m.end()).

Match-Objekte: Beispiele

Python-Interpreter

```
>>> f='00056640358000034505005230157090604009280 0'
>>> rx = re.compile(r'(?<=0)[1-9]{3}(?=0)')
>>> for m in rx.finditer(f):
...     print(m.group(0), m.span(), '', end='')
358 (8, 11) 345 (15, 18) 523 (22, 25) 157 (26, 29) 928 (37,
40)
>>> t='Python ist eine langweilige Sprache'
>>> ry = re.compile('langweilig')
>>> ry.sub('toll',t)
'Python ist eine tolle Sprache'
```

20.5 Ausführliches Beispiel

Beispiel: Worthäufigkeiten zählen (1)

- Isolieren von Worten (unter Weglassen von Satzzeichen): `\b(\w+)\b`.
- Worte mit Bindestrichen und Apostrophen?

→ Hinzunahme von Apostroph und Bindestrich: `\b([\w'-]+)\b`.

- Ist `tree_branch` ein Wort?

→ Ohne Unterstriche: `\b([a-zA-ZäöüÄÖÜß'-]+)\b`

- Was, wenn spezielle Buchstaben aus anderen Sprachen berücksichtigt werden sollen?

→ Alle Wortzeichen ohne Unterstrich: `\b(([^\W_]|['-])+)\b`

- Trennen Unterstriche Worte?

→ Unterstrich als Worttrenner:
`(\b|(?<=_))(([^\W_]|['-])+)(\b|(?=_))`

Beispiel: Worthäufigkeiten zählen (2)

count.py

```
import re
import operator
def count(fn, maxitems=10):
  rx = re.compile(
         r"(\b|(?<=_))(([^\W_]|['-])+)(\b|(?=_))")
  hist = dict()
  with open(fn, encoding='utf8') as f:
    text = f.read()
  for m in rx.finditer(text):
    hist[m.group(0).lower()] =
      hist.setdefault(m.group(0).lower(),0) + 1
  return((sorted(hist.items(),
                 key=operator.itemgetter(1),
                 reverse=True))[0:maxitems])
```

Beispiel: Worthäufigkeiten zählen (3)

- Die Konstruktion
  `with open(name) as f: ...`
  ist äquivalent zu der längeren Konstruktion:
  `try: f = open(name); ...`
  `finally: f.close()`
- `encoding` ist ein benannter Parameter für `open`, mit dem man die Kodierung der Textdatei spezifizieren kann (u.a., `ascii`, `utf8`, `latin9`, `cp1252`)
- Den `key`-Parameter und die `itemgetter`-Funktion hatten wir schon besprochen.