Skriptsprachen: Python
Regular Expressions

Jan Krüger, Alexander Sczyrba

Technische Fakultät
Universität Bielefeld

18. Februar 2020

Reguläre Ausdrücke

Was wir machen wollen OHNE irgendeinen Texteditor zu verwenden (`STRG+F` ist keine Textsuche):

- Text-Dateien zeilenweise einlesen

Reguläre Ausdrücke

Was wir machen wollen OHNE irgendeinen Texteditor zu verwenden (`STRG+F` ist keine Textsuche):

- Text-Dateien zeilenweise einlesen
- Zeilen nach Mustern durchsuchen

Reguläre Ausdrücke

Was wir machen wollen OHNE irgendeinen Texteditor zu verwenden (`STRG+F` ist keine Textsuche):

- Text-Dateien zeilenweise einlesen
- Zeilen nach Mustern durchsuchen
- Zeilen in Teilworte zerlegen

Reguläre Ausdrücke

Was wir machen wollen OHNE irgendeinen Texteditor zu verwenden (`STRG+F` ist keine Textsuche):

- Text-Dateien zeilenweise einlesen
- Zeilen nach Mustern durchsuchen
- Zeilen in Teilworte zerlegen
- Textstücke ersetzen

Reguläre Ausdrücke

- Beschreiben einfache Sprachen
- „schwächste“ Chomsky-Grammatik
- In Python: *Regular Expressions* (RE), nach Perl-Syntax
- Regular Expressions in Python deutlich mächtiger als Reguläre Sprachen

Reguläre Ausdrücke

Wir arbeiten mit dem `re`-Modul, dass uns die Funktionalität von Perl-RE zur Verfügung stellt.

In Python geht alles von einem `pattern`-Objekt aus, das entsprechende Funktionen anbietet.

```
> import re

> story = 'In a hole in the ground there lived a boggit.'
> p = re.compile(r"in")

> m = p.match(story)      #Sucht am Stringanfang
> m                       #Kein Treffer -> None
> m = p.search(story)     #Sucht im gesamten String
> m                       #Match-Objekt
<_sre.SRE_Match at 0x1042a9648>
```

Pattern-Objekt

Das Pattern wird mit der `re.compile(<RE_STR>, <FLAGS>)` erstellt.

- Das Pattern beginnt mit einem vorangestellten `r` gefolgt von der eigentlichen RE als String (Bsp: `re.compile(r"[a-Z]*")`)
- Häufigst verwendeter *Modifier* (Flag):

  re.IGNORECASE Ignoriere Groß- und Kleinschreibung

Aus dem obigen Beispiel folgt:

```
re.compile(r"[a-z]*", re.IGNORECASE)
```

Match-Objekt

Auf dem `match`-Objekt sind folgende Methoden definiert:

- group() Rückgabewert: Der gematchte String
- start() Rückgabewert: Startindex des Matches
- end() Rückgabewert: Endindex des Matches
- span() Rückgabewert: Start-/Endindex als Tupel

Match-Objekt

Wenn ein Treffer gefunden wurde, so wird ein `match`-Objekt zurückgeliefert, ansonsten `None`.

Generelles Vorgehen zur RE-Verarbeitung in Python

```
> p = re.compile(<PATTERN>)
> m = p.match( 'string␣goes␣here' )
> if m:
>     print('Match␣found:␣', m.group(), '␣with␣indices␣', m.span())
> else:
>     print('No␣match')
```

Pattern – findall() & finditer()

Um alle Vorkommen zu finden, verwendet `findall()` bzw. `finditer()`:

```
> p = re.compile('\d+')
> p.findall('12␣drummers␣drumming,␣11␣pipers␣piping,
␣␣10␣lords␣a-leaping')
['12', '11', '10'] #Alle gefundenen Matches als Liste

> iterator = p.finditer('12␣drummers␣drumming,␣11␣...␣10␣...')
> iterator #Iterator mit match-Objekten
<callable-iterator object at 0x...>
> for match in iterator:
>     print(match.span())
(0, 2)
(22, 24)
(29, 31)
```

sub() – Patternersetzung

Mit `sub(<REPL>, <STR>[, count=0])` (*Substitute*) können Pattern mit `REPL` ersetzt werden. Die maximale Ersetzungshäufigkeit kann mit `count` angegeben werden.

```
> t = '12 drummers drumming, 11 pipers piping, ...'
> p = re.compile('umm')

> p.sub("ift", t)
'12 drifters drifting, 11 pipers piping, ...'

> p.sub("rift", t, count=1)
'12 drifters drumming, 11 pipers piping, ...'
```

Übung – Suchen und Ersetzen

Die nachfolgend genannten Textdateien findet ihr in */vol/lehre/python/*. Es handelt sich um Theaterstücke von Wayne Anthoney.

- Wieviele Zeilen von *romeo.txt* enthalten das Wort „Gold“?
- Gib die jeweiligen Indexpositionen der Treffer pro Zeile aus.
- Ersetze in dem Text *eric.txt* das Wort „Estragon“ durch „Basil“ und „Vladimir“ durch „Ilych“.

RegEx

- Alternativen: `r"Huey|Dewey|Louie"`
- Gruppierung: `r"(Hu|Dew)ey|Louie"`
- Quantoren

```
r"ab?a" # aa, aba
r"ab*a" # aa, aba, abba, abbba, abbbba, ...
r"ab+a" # aba, abba, abbba, abbbba, ...
r"ab{3,6}a" # abbba, abbbba, abbbbba, abbbbbba
r"a(bab)+a" # ababa, ababbaba, ababbabbaba, ...
```

RegEx

- Zeichenklassen

```
r"hello\s+world" # whitespace
r"es␣ist␣\d+␣Uhr" # digits
r"name:␣\w+" # letters (words)
```

- „Gegenteile“: `\S`, `\D`, `\W`
- Selbst erstellte Zeichenklasse

```
r"M[ea][iy]er" # Meier, Meyer, Maier, Mayer
r"[a-z]{2,8}" # Account-Namen
r"[A-Z][^0-9]+"
```

- Passt auf alles: `.`

Anker

Muster an bestimmte Position binden:

- Zeilenanfang/-ende

```
r"^LOCUS.+" # LOCUS Zeile aus GenBank Datei
r"\s+$" # all trailing whitespace
r"^\d+␣\d+␣\d+$" # 3d coord
```

- Wortzwischenraum

```
r"\bmit\b" # "nicht mit mir", "Kommen Sie mit!"
r"\bmit\B" # "mittendrin", nicht "vermitteln"
```

Übung – Zeilen extrahieren

- Lies die Datei */etc/services* zeilenweise ein.
  - Gib alle Zeilen aus, die sich auf das Protokoll *TCP* beziehen.
  - Gib alle Zeilen aus, die einen Service deviieren (also keine Kommentarzeile sind).
  - Gib alle Zeilen aus, die eine vier- oder fünfstellige Port-Nummer enthalten.
- Wie können alle Zeilen aus *romeo.txt* extrahiert werden, in denen die Worte „club“ oder „clubs“ vorkommen, aber nicht „clubroom“?

Pattern Capturing

- Bis jetzt: Muster kommen im Text vor!

Pattern Capturing

- Bis jetzt: Muster kommen im Text vor!
- Aber: Wie sah der Treffer aus? $\rightarrow$ `findall` nur halbe Wahrheit

Pattern Capturing

- Bis jetzt: Muster kommen im Text vor!
- Aber: Wie sah der Treffer aus? $\rightarrow$ `findall` nur halbe Wahrheit
- Interessanten Bereich markieren:

```
r"^LOCUS\s+(\S+)"
r"^VERSION\s+(\S+)\.(\d+)\s+GI:(\d+)$"
```

Pattern Capturing

- Bis jetzt: Muster kommen im Text vor!
- Aber: Wie sah der Treffer aus? $\rightarrow$ `findall` nur halbe Wahrheit
- Interessanten Bereich markieren:

```
r"^LOCUS\s+(\S+)"
r"^VERSION\s+(\S+)\.(\d+)\s+GI:(\d+)$"
```

- Treffer landen bei `match` im `match`-Objekt

```
> p = re.compile(r'a(b((c)d))')
> m = p.match('abcd')
> m.group()     #Gesamter Match -> m.group(0)
'abcd'
> m.groups()    #Markierte Gruppen
('bcd', 'cd', 'c')
> m.group(2)
'cd'
```

Pattern Capturing

- Bis jetzt: Muster kommen im Text vor!
- Aber: Wie sah der Treffer aus? $\rightarrow$ `findall` nur halbe Wahrheit
- Interessanten Bereich markieren:

```
r"^LOCUS\s+(\S+)"
r"^VERSION\s+(\S+)\.(\d+)\s+GI:(\d+)$"
```

- Treffer landen bei `match` im `match`-Objekt

```
> p = re.compile(r'a(b((c)d))')
> m = p.match('abcd')
> m.group()     #Gesamter Match -> m.group(0)
'abcd'
> m.groups()    #Markierte Gruppen
('bcd', 'cd', 'c')
> m.group(2)
'cd'
```

- Quantoren richtig einsetzen: `(\w)+` != `(\w+)`

Übung – Romeo, oh Romeo...

In *romeo.txt* findet sich die Regieanweisung der „ROMEO enters“.

Extrahiert die Namen der Personen, die auf diese Weise die Bühne betreten. Verwendet einen geeigneten Datentyp, um die Personen lediglich einmalig zu speichern.

Pattern Capturing

- Das Muster muss vollständig passen:

```
> p = re.compile(r'a(b((c)d))')
> m = p.match('abcd')
> type(m)
NoneType
```

- Unterschied zwischen *grouping* und *capturing*:

```
r"\d+(-\d+)*"        # -12345
r"\d+(?:-\d+)*"      # 12345
```

Übung – Servicelist as a Service

Lies */etc/services* ein. Stelle die Informationen über die Dienste etwas umgangssprachlicher dar.

Für die Zeile `ftp 21/tcp` soll die Ausgabe wie folgt aussehen:

```
Der Dienst "ftp"verwendet TCP auf Port 21
```

Eventuelle weitere Informationen (Alias-Name oder Kommentare) sollen ignoriert werden.

Greedy Matches

- Was passiert, wenn Pattern nicht eindeutig sind?

```
> t = "aaaaaaaaaaa"
> p = re.compile(r"(a+)(a+)")
> m = p.match(t)
> m.group()      #???
> m.groups()     #???
```

Greedy Matches

- Was passiert, wenn Pattern nicht eindeutig sind?

```
> t = "aaaaaaaaaaa"
> p = re.compile(r"(a+)(a+)")
> m = p.match(t)
> m.group()      #???
> m.groups()     #???
```

- Ausprobieren:

```
r"(a+)(a*)"
r"(a*)(a+)"
r"(a*)(a*)"
r"(a?)(a*)"
r"(a{2,4})(a*)"
```

Greedy Matches

- Was passiert, wenn Pattern nicht eindeutig sind?

```
> t = "aaaaaaaaaaa"
> p = re.compile(r"(a+)(a+)")
> m = p.match(t)
> m.group()      #???
> m.groups()     #???
```

- Ausprobieren:

```
r"(a+)(a*)"
r"(a*)(a+)"
r"(a*)(a*)"
r"(a?)(a*)"
r"(a{2,4})(a*)"
```

- Setze hinter den ersten Quantoren: `+?  *?  ??  {2,4}?`

Text zerteilen

- Sequenzkomponenten zu einen String vereinen:

```
> l = ['b', 1, 'ffeeg']
> "".join(map(str, l))
'b1ffeeg'
```

- Entgegengesetzte Funktion mit RE: `split(string[, count=0])`
- Trennung durch Pattern:

```
> story = "In␣a␣hole␣in␣the␣ground␣there␣lived␣a␣hobbit."
> p = re.compile(r"\s")
> p.split(story)
['In', 'a', 'hole', 'in', 'the', 'ground', 'there', 'lived',
   'a', 'hobbit.']
```