Reguläre Ausdrücke in Python

Prof. Dr. Rüdiger Weis

Beuth Hochschule für Technik Berlin

Reguläre Ausdrücke

Reguläre Ausdrücke

Reguläre Ausdrücke (engl. regular expression) beschreiben eine Familie von formalen Sprachen.

- Typ-3 Chomsky-Hierarchie
- Endliche Automaten
- Unix Shell, grep, emacs, vi, PERL, Python

Python re Module

- Python Library Reference
  - **4.2 re – Regular expression operations**
  - `http://docs.python.org/lib/module-re.html`
- A.M. Kuchling, Regular Expression HOWTO
  - `http://www.amk.ca/python/howto/regex/`
- Wikipedia-Artikel
  - Reguläre Ausdrücke
  - `http://de.wikipedia.org/w/index.php?title=Regul%C3%A4rer_Ausdruck`

Raw Strings

- r'String'
- Escapesequenzen werden nicht interpretiert.

Reguläre Ausdrücke

Verwenden sie grundsätzlich raw Strings, wenn sie mit regular expressions arbeiten.

compile

compile

compile(pattern[,flags])

```
cpat = re.compile(pat)
result = cpat.match(str)
```

ist äquivalent zu

```
result = re.match(pat,str)
```

Flags I, L, U

re.I re.IGNORECASE
Nichtberücksichtigung von Gross- und Kleinschreibung.

re.L re.LOCALE
Macht

\w \W \b \B \s \S

abhängig von der Lokalisierungseinstellung.

re.U re.UNICODE
Macht

\w \W \b \B \d \D \s \S

abhängig von der Unicode Charakter properties database.

Flags M, S

re.M re.MULTILINE

^ erkennt am Beginn des Strings und von jeder Zeile.

$ erkennt am Ende des Strings und von jeder Zeile.

re.S re.DOTALL

Der Punkt . erkennt alle Zeichen inklusive newline Zeichen (\n).

compile flag X

re.X re.VERBOSE

Ermöglicht lesbarer Formatierung der RE

- # Kommentar bis zum Zeilen Ende
- Suchen nach # mittels [#] oder \#
- Whitespaces werden ignoriert
- Suchen von Leerzeichen mittels [ ] oder \

Beispiel VERBOSE

```
ref = re.compile(r"""
 (
    [1-9][0-9]*[^0-9]          # Dezimal-Zahl
    |  0[0-7]+[^0-7]     # Oktal-Zahl
    |  x[0-9a-fA-F]+[^0-9a-fA-F] # Hexadezimal-Zahl
 )
""", re.VERBOSE)
```

```
ref = re.compile("([1-9][0-9]*[^0-9]"
                 "|0[0-7]+[^0-7]"
                 "|x[0-9a-fA-F]+[^0-9a-fA-F])")
```

search

search

search(pattern, string[, flags])
p.search(string[,pos[,endpos]])

- Innerhalb des String (beziehungsweise zwischen pos und (endpos-1)) werden 0 oder mehr Zeichen mit dem Regulären Ausdruck verglichen und ein MatchObject zurückgeliefert.
- Liefert `None` zurück, falls keine Übereinstimmung vorliegt.

match

match

match(pattern, string[, flags])
p.match(string[,pos[,endpos]])

- Ab Beginn (beziehungsweise zwischen pos und (endpos-1)) des String werden 0 oder mehr Zeichen mit dem Regulären Ausdruck verglichen und ein MatchObject zurückgeliefert.
- Liefert `None` zurück, falls keine Übereinstimmung vorliegt.

Beispiel

```
>>> import re
>>> s = "Spam, eggs and spam."
>>> m = re.match("Spam", s)
>>> print m
<_sre.SRE_Match object at 0xb7df7f70>
>>> print m.group()
Spam
>>> m = re.match("eggs", s)
>>> print m
None
>>> m = re.search("eggs", s)
>>> print m
<_sre.SRE_Match object at 0xb7df7f70>
```

RegexObject

```
>>> import re
>>> p = re.compile("spam+")
>>> p
<_sre.SRE_Pattern object at 0xb7e2f4e0>
>>> p.search("Spam, spammmmm...")
<_sre.SRE_Match object at 0xb7df1020>
>>> p.search("Spam, spammmmm...").group()
'spammmmm'
```

Metazeichen

Metazeichen

. ^ $ * + ? { } [ ] \ | ( )

Auswahlbereiche

```
\d: Alle Ziffern, [0-9]
\D: Alles ausser Ziffern, [^0-9]

\s: Whitespace, [ \t\n\r\f\v]
\S: Alles ausser Whitespace, [^ \t\n\r\f\v]

\w: Alphanumerische Zeichen, [a-zA-Z0-9]
\W: Alles ausser alphanumerische Zeichen, [^a-zA-Z0-9]

\b: Wortgenze
\B: Alles ausser Wortgrenze

|: Oder Verknuepfung
```

Auswahl

. beliebiges Zeichen

[Beginn einer Auswahl von Zeichen

]Ende einer Auswahl von Zeichen

| Oder-Verknüpfung Auswahl

[^ Negation Auswahl

\ Hebt Sonderbehandlung für das folgende Zeichen auf.

Beispiel

- [abcd] Matcht ein einzelne Zeichen a, b, c oder d
- [a-z] Matcht Kleinbuchstaben (ASCII)
- [a-zA-Z] Bereich: Matcht Gross- und Kleinbuchstaben (ASCII)
- [abc$^] Matcht a, b, c oder $ oder ^.
- [ab\]cd] Matcht a, b,], c, oder d

Beispiel: ^

```
>>> import re
>>> s = "Das Leben des Brian"
>>> m = re.search('^Das', s) # String Begin
>>> m.group()
'Das'
>>> m = re.search('^des', s)
>>> m
>>> print m
None
```

Beispiel: $

```
>>> import re
>>> s = "Das Leben des Brian"
>>> m = re.search('Brian$', s) # String Ende
>>> m.group()
'Brian'
>>> m = re.search('Bri$', s)
>>> print m
None
```

^ Zirkus

```
>>> re.search('\^','^').group()
'^'
>>> re.search('^\^','^').group()
'^'
>>> print re.search('^\^','O^')
None
>>> re.search('^\^','^O').group()
'^'
>>> print re.search('^[^]','^O')
None
>>> re.search('^[^]','X^X').group()
'X'
>>> print re.search('^[^]','^')
None
```

Quantoren

*	Wiederholung 0, 1 oder viele
?	Wiederholung 0 oder 1
+	Wiederholung 1 oder viele
{	Beginn gezählter Wiederholung
}	Ende gezählter Wiederholung
{m,n}	Wiederholung mindestens m mal und maximal n mal

Gruppierung

(Beginn Gruppierung

) Ende Gruppierung

^ Anfang String oder Zeile

$ Ende String oder Zeile

\A Matcht Begin des Strings auch im `MULTILINE` Mode

findall

findall

findall(pattern, string[, flags])
p.findall(string[, pos[, endpos]])

- Keine Gruppe: Liste von allen nicht-überlappenden Musterübereinstimmungen.
- Eine Gruppe: Musterübereinstimmungen bezüglich Gruppe.
- Mehrere Gruppen: Liste mit Tupeln der Musterübereinstimmungen bezüglich der Gruppen.

Beispiel: findall

```
>>> s = "Kreuz 6, Schippe 9, Herz 7, Karo 8"
>> re.findall("\d", s)
['6', '9', '7', '8']
>>> re.findall("Herz \d", s)
['Herz 7']
>>> re.findall("Her(z \d)", s)
['z 7']
>>> re.findall("z \d", s)
['z 6', 'z 7']
>>> re.findall("(z)( \d)", s)
[('z', ' 6'), ('z', ' 7')]
```

finditer

finditer

finditer(pattern, string[, flags])
finditer(string[, pos[, endpos]])

- Liefert einen Iterator von allen nicht-überlappenden Musterübereinstimmungen.

Beispiel: finditer

```
>>> s = "Kreuz 6, Schippe 9, Herz 7, Karo 8"
>>> mi=re.finditer("(\d)", s)
>>> mi.next().group()
'6'
>>> mi.next().group()
'9'
>>> mi.next().group()
'7'
>>> mi.next().group()
'8'
>>> mi.next().group()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
8
```

Beispiel: finditer for Schleife

```
>>> s="Kreuz 6, Schippe 9, Herz 7, Karo 8"
>>> mi=re.finditer("(\d)", s)
>>> for hit in mi:
        print hit.group()
...
6
9
7
8
```

sub

sub

sub(pattern,repl,string[,count])
p.sub(repl, string[, count = 0])

- Liefert einen String mit Ersetzungen der nicht-überlappenden Musterübereinstimmungen mit dem repl String.

subn

subn

subn(repl,string[, count = 0])
p.subn(repl, string[, count = 0])

- Führt sub() Ersetzung aus und liefert ein Tupel (*new_string*, *number_of_subs_made*).

Beispiel: sub, subn

```
>>> s = "Windows mit MS Office. Windows ist installiert."
>>> besser = re.sub("MS", "Open", s)
>>> besser
'Windows mit Open Office. Windows ist installiert.'
>>> re.subn("Windows", "Linux", besser)
('Linux mit Open Office. Linux ist installiert.', 2)
```

split

split

split(pattern, string[, maxsplit = 0])
p.split(string[, maxsplit = 0])

- Spaltet string mit dem pattern als Trennpunkte.
- Mit () eingeklammerte Gruppen im Pattern werden im resultierenden String mit zurückgeliefert.
- Falls maxsplit gesetzt wir, werden höchstend maxsplit Schnitte durchgeführt. Der restliche String wird als letzter String in der Liste zurückgeliefert.

Beispiel: split

```
>>> s = "Die Gedanken sind      frei"
>>> re.split("\W+", s)
['Die', 'Gedanken', 'sind', 'frei']
>>> re.split("(\W+)", s)
['Die', ' ', 'Gedanken', ' ', 'sind', '      ', 'frei']
>>> re.split("\W+", s, 2)
['Die', 'Gedanken', 'sind      frei']
```

escape

escape

escape(string)

- Liefert String zurück, in dem alle non-alphanumerischen Zeichen ge-backslashed werden.

```
>>> re.escape(r"\section{}")
'\\\\section\\{\\}'
>>> re.escape("...$")
'\\.\\.\\.\\$'
```

Match Objects

match und search liefern ein `MatchObject` zurück.

Methoden

- start
- end
- span
- group
- groups

start, end, span

- start([group])
  Liefert Start der Muststerübereinstimmung.
- end([group])
  Liefert Ende der Muststerübereinstimmung.
- span([group])
  Liefert `(m.start(group), m.end(group)`

Beispiel: Match Object Methoden

```
>>> s = "Spam, spam, eggs and spam."
>>> m = re.search('spam', s)

>>> m.start()
6
>>> m.end()
10
>>> m.span()
(6, 10)
```

group

group

p.group([group1, ...])

- Liefert ein oder mehrere Untergruppen der Musterübereinstimmung.
- Defaultwert von `group1` ist 0, was die gesamte Musterübereinstimmung liefert.

groups

groups

p.groups([default])

- Liefert ein Tupel der Untergruppen der Musterübereinstimmung.
- Das `default` Argument wird benutzt für Gruppen die nicht bei Musterübereinstimmung berücksichtigt werden. Default in `None`.

Beispiel: Match Object Methoden

```
>>> s = "Spam, spam, eggs and spam."
>>> m = re.search('(spam)', s)
>>> m.groups()
('spam',)
>>> m = re.search("(spam)(, )(eggs and spam)", s)
>>> m.groups()
('spam', ', ', 'eggs and spam')
>>> m.span()
(6, 25)
```

Gierig und Nicht-Gierig

greedy und non-greedy

- *, +, ?, {n,m}
  Gieriges Matchen
  So viel wie möglich
- *?, +?, ??, {n,m}?
  Nicht-gieriges Matchen
  So wenig wie möglich.

Bsp: greedy und non-greedy

```
>>> s = "<title>Technische Fachhochschule Berlin</title>"
>>> m = re.search("<.*>", s) # Gierig
>>> m.group()
'<title>Technische Fachhochschule Berlin</title>'
>>> m = re.search("<.*?>", s) # Nicht-gierig
>>> m.group()
'<title>'
```

```
>>> print re.search("<.*?>", "1 <> 2").group() # Vorsicht *
<>
```

Gruppen Nummern \1 ...

\1 ...

\1 ...

- Matcht den Inhalt der entsprechenden Gruppe.
- Zählung beginnt mit 1
- raw String verwenden

Beispiel: \1 ...

```
>>> import re
>>> s = 'Breakfast: spam spam, spam eggs'
>>> pattern = re.compile(r'(\w+) (\1)')
>>> matchObj = pattern.search(s)
>>> matchObj.group()
'spam spam'
>>> satz="Hund beisst Mann"
>>> re.sub(r'(\w+) (\w+) (\w+)', r'\3 \2 \1', satz)
'Mann beisst Hund'
```

(?P<name>) und (?P=name) Gruppen-Namen

(?P<name>...)

(?P<name>...)
Zuweisung von Gruppen-Namen

(?P=name)

(?P=name)
Ansprechen von Gruppen-Namen

groupdict

groupdict

p.groupdict([default])

- Liefert Dictonary aller benannten Untergruppen mit dem Gruppen-Namen als Schlüssel.
- Das `default` Argument wird benutzt für Gruppen, welche nicht bei Musterübereinstimmung berücksichtigt werden. Default in `None`.

Beispiel: Gruppen-Namen

```
>>> import re
>>> s = "http://www.bht-berlin.de/index.html"
>>> pat = re.compile('(?P<Protokol>\w+)\W+(?P<Adresse>[\w.-]+)\W')
>>> pat.search(s).groupdict()
{'Protokol': 'http', 'Adresse': 'www.bht-berlin.de'}
>>> pat.search(s).groups()
('http', 'www.bht-berlin.de')
>>> pat.search(s).group()
'http://www.bht-berlin.de/'
>>> pat.search(s).group(1) # !Achtung!
'http'
>>> pat.search(s).group("Protokol")
'http'
>>> pat.search(s).group(2)
'www.bht-berlin.de'
```

(?=...) Positive lookahead assertion

(?=...) Positive lookahead assertion

(?=...)

- Sucht Übereinstimmung mit dem folgenden String
- Verbraucht keine Zeichen (zero-width assertion)

(?!...) Negative lookahead assertion

(?!...) Negative lookahead assertion

(?!...)

- Liefert Übereinstimmung falls folgender String nicht übereinstimmt.
- Verbraucht keine Zeichen (zero-width assertion)

Beispiel: Nicht .txt enden

Alle Dateien, welche nicht auf `.txt` enden

```
.*[.]([^t].?.?|.[^x]?.?|..?[^t]?)$
```

oder

```
.*[.](?!txt$).*$
```

(?<=...) Positive lookbehind assertion

(?<=...) Positive lookbehind assertion

(?<=...)

- Sucht Übereinstimmung mit dem vorangehenden String
- Verbraucht keine Zeichen (zero-width assertion)

(?<!...) Negative lookbehind assertion

(?<!...) Negative lookahead assertion

(?<!...)

- Liefert Übereinstimmung falls vorausgehender String nicht übereinstimmt.
- Verbraucht keine Zeichen (zero-width assertion)

Beispiel: lookbehind assertion

```
>>> import re
>>> s = 'spam eggs, spam spam, python eggs'
>>> m = re.search("(?<=python) eggs", s)
>>> m.span()
(28, 33)
>>> m = re.search("(?<!python) eggs", s)
>>> m.span()
(4, 9)
```

(?(id/name)yes-pattern|no-pattern)

(?(id/name)yes-pattern|no-pattern)

- Falls die id/name Gruppe existiert, versuche gegen das yes-pattern zu matchen sonst gegen das no-pattern.
- no-pattern ist optional

Beispiel (?(id/name)yes-pattern|no-pattern)

```
>>> import re
>>> muster = "(<)?(\w+@([\w-])+\.\w+)(?(1)>)"
>>> print re.match(muster, "<rweis@bht-berlin.de>").group()
<rweis@bht-berlin.de>
>>> print re.match(muster, "rweis@bht-berlin.de").group()
rweis@bht-berlin.de
>>> print re.match(muster, "<rweis@bht-berlin.de")
None
```

©opyleft