Python: Reguläre Ausdrücke Extended Format (C) 2014-2024 T.Birnthaler OSTC GmbH

Reguläre Ausdrücke (Regex) sind eine in praktisch allen Programiersprachen eingebettete MINI-PROGRAMMIERSPRACHE zum Vergleich (MATCHEN) von Strings (Zeilen) mit einem durch den Regex vorgegebenen FORMAT und zur Extraktion von Teilstrings daraus über den Regex im Falle eines Matches.

Reguläre Ausdrücke bestehen sowohl aus normalen Zeichen (LITERAL) als auch speziellen Zeichen (METAZEICHEN) mit einer besonderen Funktionalität. Die normalen Zeichen wie "A", "a" oder "0" bilden die einfachsten Regulären Ausdrücke, sie matchen einfach sich selbst. Per Verkettung kann man komplexere Reguläre Ausdrücke bilden, z.B. matcht der Reguläre Ausdruck "last" den String "last".

Normalerweise hat in Regulären Ausdrücken JEDES EINZELNE ZEICHEN eine Bedeutung (Literale steht für sich selbst, Metazeichen stehen für eine Funktionalität) und daher sind sie in einer einzigen Zeile und ohne Whitespaces (Leerzeichen und Zeilenvorschub) zu schreiben.

Das macht Regex sehr unübersichtlich und damit schwer lesbar und wartbar. Man sagt auch, Regex sind eine "WRITE-ONLY" Sprache: Das Hinschreiben eines Regex geht noch, das Lesen die Pflege eines Regex ist hingegen sehr fehleranfällig.

Extended Format

Das von der Sprache "Perl" eingeführte EXTENDED FORMAT erlaubt auch in Python die FORMATIERUNG, EINRÜCKUNG und KOMMENTIERUNG von Regulären Ausdrücken wie bei einem normalen Programm.

Das wesentliche Problem von Regex ist damit von Tisch --- dass sie so unleserlich sind, weil sie am Stück in einer Zeile zu schreiben sind und standardmäßig JEDES EINZELNE Zeichen in ihnen eine Bedeutung hat (auch Leerzeichen). Sie werden mit Hilfe dieses Formats deutlich besser LESBAR und damit auch WARTBAR.

Durch folgende Kombination erhält man die bessere Lesbarkeit:

* Mit Begrenzer R"""...""" oder R'''...''' umrahmen (R klein oder groß erlaubt)
  + R=RAW/REGEX: Escape-Sequenz "\X" nicht als Sonderzeichen interpretiert
  + Aufteilen auf MEHRERE Zeilen erlaubt

* re.VERBOSE oder re.X als FLAG (3. Parameter) eintragen (eXtended Format)
  + KOMMENTAR #... bis zum Zeilenende wird ignoriert
    --> Zu matchendes Zeichen "#" schützen: \# oder [#]
  + WHITESPACE [\t\v\n\r\f] wird ignoriert
    (Space, Tabulator Vertical Tabulator, Newline, Return, FormFeed):
    --> Leerraum darf zum Einrücken und Formatieren benutzt werden
    --> Zu matchendes Leerzeichen " " schützen: \ oder [ ]
    --> Zu matchenden Whitespace "..." so schreiben: \s
  + Nach "(" einrücken, vor ")" ausrücken --> Merken + Gruppieren übersichtlich
  + Analog "(?: ...) --> Nur Gruppieren (NICHT Merken)
    ACHTUNG: Syntaxfehler: (:? ...) ist falsch!

Für die Art der Einrückung und Aufteilung eines Regex auf mehrere Zeilen gibt es keine Vorschriften. Typischerweise sollten thematisch zusammengehörende Teilstücke eines Regex in getrennte Zeilen und geklammerte Elemente per Einrückung visualisiert werden. Weiterhin sollte man jede Zeile kommentieren, um ihre Bedeutung klar zu machen.

Beispiel:

```
  # Klassischer Regulärer Ausdruck (JEDES EINZELNE Zeichen hat eine Bedeutung)
  mo = re.search(r"^\s+      #(.*)\s+\1(?:.*)\s*$", line)
  #                    ^^^^^^ 6 Leerzeichen!

  # Gleicher Regulärer Ausdruck im Extended-Format (formatiert + kommentiert):
  mo = re.search(        # MO=MatchObject
       R"""              # raw/regex + Zeilenumbruch erlaubt
           ^             # Zeilen-Anfang
           \s+           # 1-n Leerraum ( hiteSpace)
           \ \ \         # 3 Leerzeichen
           [ ][ ][ ]     # 3 Leerzeichen
           \#            # Zeichen "#"
           (             # Merken 1: Einrücken
```

```
            .*        # Merken 1: 0-n beliebige Zeichen
          )           # Merken 1: Ausrücken
          \s+         # 1-n Leerräume (WhiteSpace)
          \1          # Text von "Merken 1" nochmal
          (?: .* )    # Klammern, aber nicht merken (z.B. weil optional)
          \s*         # 0-n Leerräume (WhiteSpace)
          $           # Zeilen-Ende
      """, line, re.VERBOSE)   # re.X = re.VERBOSE = Extended Format
if (mo):
    print "Matchteile:          ", len(mo.groups())
    print "Vollständiger Match:",  mo.group(0)
    print "Teilmatch:           ", mo.group(1)
else:
    print "KEIN match"
```

Python-Funktionen zur Nutzung Regulärer Ausdrücke

P ist ein Regex-Muster (Pattern), S, T sind Strings, F ist eine Flag-Kombination, R ist ein Ersetzungs-String, MAX ist ein Integer (maximale Anzahl an Ersetzungen oder Zerlegungen), MO ist ein Match-Object, L ist eine Liste von Strings, IT ist ein Iterator, RO ist ein Regex-Object.

MO = match(P,S,F)	Muster passt am Anfang?
MO = fullmatch(P,S,F)	Muster passt vollständig?
MO = search(P,S,F)	Muster passt irgendwo?
T = sub(P,R,S,MAX,F)	Muster P durch R in String S ersetzen
(T,N) = subn(...)	Analog sub() --> (Ergebnis-String, Anz. Ersetz.)
L = split(P,S,MAX,F)	String durch Muster in Stücke zerlegen
L = findall(P,S,F)	Alle zum Muster passenden Strings als Liste
IT = finditer(P,S,F)	Iterator, gibt zum Muster pass. Strings zurück
RO = compile(P,F)	Muster in Regex-Objekt übersetzen
purge()	Regex-Cache löschen
T = escape(P)	Metazeichen im Muster mit Backslash "\" versehen
error(MSG,P,POS)	Exception falls ein Muster ungültig ist

Regex-Metazeichen

Folgende Elemente in einem Regex stehen nicht für sich selbst, sondern lösen eine bestimmte Funktionalität aus (Quotierung, Zeichenklasse, Anker, Quantifier, Oder, Klammerung, Gruppierung, Flag, Benennung, Fallunterscheidung, Positive/Negative Look Behind/Before, ...).

\X	Wandelt um: Metazeichen <-> Normales Zeichen	
.	Ein beliebiges Zeichen (außer "\n")	
[...]	1 Zeichen aus Zeichen-Menge	
[^...]	1 Zeichen aus Inverser Zeichen-Menge	
^	String-Anfang	
$	String-Ende oder vor "\n" am String-Ende	
R*	0 oder mehr Wiederholungen des Regex R (greedy)	
R+	1 oder mehr Wiederholungen des Regex R (greedy)	
R?	0 oder 1 Wiederholung des Regex R (greedy)	
R{m,n}	M bis N Wiederholungen des Regex R (greedy)	
*? +? ??	Non-greedy (lazy) Version der Wiederholungen * + ?	
R{m,n}?	Non-greedy (lazy) Version der Wiederholung {M,N}	
*+ ++ ?+	Possessive Version der Wiederholungen * + ? (kein Backtracking)	3.11
R{m,n}+	Possessive Version der Wiederholung {M,N} (kein Backtracking)	3.11
R1|R2	Matcht Regex R1 oder Regex R2	
(...)	Inhalt kann abgefragt oder später wiederverwendet werden	
(?aiLmsux)	Flag aktivieren	
(?:...)	Non-grouping Version von (...) (merkt sich nichts)	
(?#...)	Kommentar (wird ignoriert).	

(?=...)	Matcht falls ... danach (look behind)	
(?!...)	Matcht falls ... NICHT danach (negative look behind)	
(?<=...)	Matcht falls ... davor (look before)	
(?<!...)	Matcht falls ... NICHT davor (negative look before)	
(?>...)	Atomic grouping (kein Backtracking darin, analog possesive)	3.11
(?P<N>...)	Von dieser Gruppe gematchter String per N)NAME nutzbar	
(?P=NAME)	Vorher von Gruppe NAME gematchten Text erneut matchen	
(?(I/N)Y|N)	Matcht Muster Y(es) falls Gruppe mit I(d) oder N(ame) matcht sonst (optionales) N(o) Muster	

Greedy = So viel wie möglich matchen (gierig)
Lazy = So wenig wie möglich matchen (faul)
Possesive = Matches nicht mehr hergeben (kein Backtracking, besitzergreifend)

Regex-Metazeichen durch Escapen eines normalen Zeichens

Einige Literale werden durch ein vorangestellten "\" in ein Metazeichen umgewandelt um spezielle Anker oder Zeichenklassen übersichtlich darstellen zu können.

\A	Matcht nur am String-Anfang
\Z	Matcht nur am String-Ende
\b	Matcht Wort-Anfang/Ende (leerer Match)
\B	Matcht Wort-Inneres (leerer Match)
\d	Matcht alle Ziffern [0-9]
\D	Matcht alle Nicht-Ziffern [^\d]
\s	Matcht alle Leerzeichen [\t\n\r\f\v]
\S	Matcht alle Nicht-Leerzeichen [^\s]
\w	Matcht alle Alphanumerischen Zeichen [a-zA-Z0-9_]
\W	Matcht alle Nicht-Alphanumerischen Zeichen [^\w]
\NUMBER	Matcht Inhalt der Klammer-Gruppe mit dieser Nummer
\\	Matcht den Backslash \

Regex-Flags

Über Regex-Flags lässt sich das generelle Verhalten von Regex oder von Teilen einer Regex steuern. Sie sind beim Funktionsaufruf zusätzlich mit anzugeben oder per "(?aiLmsux)" in einen Regex einzubetten.

Flag-Name		Inline	Bedeutung
DOTALL	S	(?s)	"." matcht auch "\n" (sonst nicht)
IGNORECASE	I	(?i)	Groß/Kleinschreibung ignorieren
MULTILINE	M	(?m)	^ matcht String-Anfang + Zeilen-Anfang $ matcht String-Ende + Zeilen-Ende
VERBOSE	X	(?x)	Leerraum + #-Kommentare bis Zeilende ignorieren
ASCII	A	(?a)	Nur ASCII-Zeichen bei \w \W \b \B \d \D \s \S
LOCALE	L	(?L)	Groß/Kleinschreibung und \w \W \b \B hängt von akt. Locale-Einstellung ab (nur bei Bytes relevant)
UNICODE	U	(?u)	Unicode-Zeichen bei \w \W \b \B \d \D \s \S (Std)
DEBUG			Debug-Info zu übersetztem Regex anzeigen
TEMPLATE	T		? ? ?

Flags können per "|" kombiniert werden (auch per "+").
Die Flags A, L, und U schließen sich gegenseitig aus.

TIPPS
---
* Öffnende Merkklammern "(" werden von links nach rechts durchnumeriert von 1 .. n
  --> Mit \1 \2 ... \n kann auf davon gematchten Inhalt zugegriffen werden
      (in re.search, re.match, re.sub(...))

* mo.groups() enthält Liste von gemerkten Textstücke (inkl. leere = None!)
  --> Klammern im Regex kennzeichnen die zu merkenden Textstücke

```
  --> Optionale Teile enthalten "None", wenn es keinen Treffer gab
  + mo.groups()      enthält ALLE gemerkten Textstücke
  + len(mo.groups()) enthält ANZAHL gemerkter Textstücke
  + mo.group(0)      enthält GANZEN gematchten Text (nicht unbedingt ganze Zeile)
  + mo.group(1)      enthält 1. gemerkten Textteil (1. Klammer (...))
  + mo.group(2)      enthält 2. gemerkten Textteil (2. Klammer (...))
  + ...
  + mo.lastindex     enthält letzte Gruppennummer

* Zu jeder Gruppe mo.group(N) gibt es folgende weiteren Informationen:
  + mo.start(N)   Start-Index der Gruppe
  + mo.end(N)     End-Index der Gruppe
  + mo.span(N)    Tupel (Start, End)-Index der Gruppe
  + mo.len(N)     Länge der Gruppe

* re.match() und re.fullmatch() NICHT verwenden, sie sind Teil von re.search()
  + re.match("...")     = re.search("^...")    # Anker ^ = ab Zeilenanfang
  + re.fullmatch("...") = re.search("^...$")   # Anker ^...$ = vollständig

* Reguläre Ausdrücke sind "GREEDY"!
  --> Suchen längsten Treffer
  --> .* alleine ist sinnlos ("passt auf alles und nichts"),
      es sollte immer etwas aussenrum stehen
  --> .*? ist "NON-GREEDY" oder "LAZY" und matcht kürzestmöglichen Text
  --> .*+ ist "POSSESIVE" und gibt gemachten Text nicht mehr frei (KEIN Backtracking)

* Reguläre Ausdrücke sind "LEFT-MOST"!
  --> Suchen von LINKS NACH RECHTS den ersten Treffer

* Erst "Schmutz" wegwerfen, dann Verarbeitung per Regex.
  + Z.B. Leerzeilen und Kommentarzeilen ignorieren per:

        if re.search(r"^\s*[#]?\s*$", line): continue

  + Whitespace am Zeilenanfang oder Ende entfernen
    (inkl. Newline, Carriage Return, Tab, ...).

        line = line.strip()
        line = re.sub(r"^\s+", "", line)
        line = re.sub(r"\s+$", "", line)

* Statt .* besser [^FOLGEZEICHEN]* schreiben.
  --> "greedy"-Verhalten wird gestoppt

* Ein "." matcht ALLE Zeichen AUSSER "\n" (Newline).
  --> Flag re.S/re.DOTALL schaltet dieses Verhalten ab (Newline wird auch gematcht).
  --> Alternative: "[\s\S]" matcht auch JEDES Zeichen.

* "^" und "$" matchen String-Anfang/Ende.
  --> Flag re.M/re.MULTILINE sorgt dafür, dass sie auch "\n" (Zeilenanfang/ende)
matchen
      (für mehrzeilige Daten in einem String interessant).
  --> Die bisherige Aufgabe der beiden übernehmen dann "\A" und "\Z" bzw. "\z".
```