RegEx

Reguläre Ausdrücke - Motivation: Ein regulärer Ausdruck kann als Beschreibung für das Aussehen eines Strings gesehen werden. Hat man diese Beschreibung nun gegeben, kann man in erster Linie überprüfen ob ein String dieser Beschreibung entspricht oder Teil-strings finden die dieser Beschreibung entsprächen. Man spricht oft davon, dass ein Ausdruck einen String *matched*, wenn der Ausdruck den String, oder Teile davon, beschreibt. Die Art solche Beschreibungen zu formulieren (also die Syntax von regulären Ausdrücken) ist weitestgehend genormt und die meisten Programmiersprachen biten eine Möglichkeit mit diesen zu arbeiten.

Reguläre Ausdrücke - Syntax: Grundsätzlich Matchen alle Buchstaben auf sich selbst. Also der reguläre Ausdruck `H` beschreibt den String `H` sind.
Es gibt allerdings einige sogenannte Meta-Characters die eine spezielle Bedeutung haben.
Konkret sind dies:

- `.` matched irgendein Zeichen bis auf Newline (\n).
- `^` matched den Zeilenanfang.
- `$` matched das Zeilenende.
- `\d` matched eine Ziffer.
- `\D` matched alles was keine Ziffer ist.
- `\` wandelt einen Meta-Character in das Normale Zeichen um. Also `\.` ist einfach der Punkt. Zuletzt kann nun noch mehrere reguläre Ausdrücke kombinieren:
- `reg_1reg_2` wobei `reg_1` und `reg_2` beliebige reguläre Ausdrücke sind, matched zuerst `reg_1` dann `reg_2`.
- `reg_1|reg_2` wobei `reg_1` und `reg_2` beliebige reguläre Ausdrücke sind, matched entweder `reg_1` oder `reg_2`.
- `(reg)` wobei `reg` ein beliebiger regulärer Ausdrück ist, kann genutzt werden Ausdrücke zu Gruppieren. (Wie bei Mathematischen Ausdrücken also)
- `reg*` wobei `reg` ein beliebiger regulärer Ausdrück ist, matched den Ausdruck `reg` null Mal oder öfters.
- `reg+` wobei `reg` ein beliebiger regulärer Ausdrück ist, matched den Ausdruck `reg` einmal oder öfters.
- `reg?` wobei `reg` ein beliebiger regulärer Ausdrück ist, matched den Ausdruck `reg` höchstens einmal.
- `reg{m,n}` wobei `reg` ein beliebiger regulärer Ausdrück ist, matched den Ausdruck `reg` mindestens m Mal und maximal n Mal. Man kann dabei auch m oder n leer lassen um die jeweilige Schranke zu entfernen.
- `reg{m}` wobei `reg` ein beliebiger regulärer Ausdrück ist, matched den Ausdruck `reg` genau m Mal.
- `[reg_1reg_2]` wobei `reg_1` und `reg_2` beliebige reguläre Ausdrücke sind, matched entweder `reg_1` oder `reg_2`. ACHTUNG: Innerhalb dieser Eckigen Klammern, wird die Bedeutung von ALLE Zeichen einfach das Zeichen selbst.

- `[reg_1-reg_2]` wobei `reg_1` und `reg_2` beliebige reguläre Ausdrücke sind. Dies ist ein Spezialfall der oberen Regel. Dabei kann man eine Range angeben. Also z.B, matched `[0-9]` alle Ziffern.

Raw-Strings - Python: Ein Raw-String ist eine spezielle art von String in Python, innerhalb dessen Escape Charaktere ignoriert werden. Also in einem Raw-String ist ein Backslash(\) einfach ein Backslash ohne besondere Bedeutung. Man erstellt einen solchen String wie folgt

```
r"Ich bin ein Raw\String"
```

Wenn man Regex Ausdrücke Schreibt ist dies sehr praktisch. Will man z.B. ein Backslash matchen, so ist der reguläre ausdruck dafür der folgende: `\\`. Will man diesen in einen Python String speichern so muss man jedes Backslash durch ein Backslash escapen. Dies muss man bei Raw-Strings nicht. Also

```
pattern = "\\\\"
pattern_raw = r"\\" # besser
```

Reguläre Ausdrücke - Python: In Python muss man eine library importieren um Regex zu nutzen.

```
import re
```

Dann werden die folgenden Funktionen zur Verfügung gestellt:

- `re.match(pattern, string)` Ist erfolgreich wenn das Pattern auf den *Anfang* des Strings matched und gibt dabei ein Match-Objekt zurück. Dieses Objekt stellt die Methode `.span()` zur Verfügung, die dann ein Tupel mit Start und End Postion des gematchend Teilstrings zurück gibt. ACHTUNG. Wie in Python üblich ist die End-Position der Index des ersten Buchstaben, welcher NICHT mehr im Match ist.
- `re.search(pattern, string)` Ist ist erfolgreich, wenn das Pattern auf einen beliebigen Teilstring matched und gibt dabei ein Match-Objekt zurück.
- `re.findall(pattern, string)` Findet alle nicht-überlappenden matchenden Teilstrings und gibt eine Liste von Strings - bei der Verwendung von Gruppen auch Tupel - zurück.
- `re.finditer(pattern, string)` Wie `re.findall` aber gibt ein iterierbares Match-Objekt zurück.
- `re.split(pattern, string)` Zerlegt den String in Teilstrings an den Stellen, an denen das Pattern matched.
- `re.sub(pattern, repl, string)` Ersetzt alle matchenden Teilstrings im String durch 'repl'.
- `re.compile(pattern)` Erzeugt ein Objekt von Typ Pattern. Dieses Objekt bietet dann alle obigen Funktionen wobei man nun natürlich kein Pattern mehr angeben muss.