Teil VII

Reguläre Ausdrücke

Was wir uns (vielleicht) schon immer gefragt haben:

- Wie funktioniert Suche nach einer Zeichenfolge in einem Text?
- Wie wird Auto-Vervollständigung in Entwicklungsumgebungen realisiert?
- Woher weiß ein email client, ob eine Emailadresse valide ist?

Was wir uns (vielleicht) schon immer gefragt haben:

- Wie funktioniert Suche nach einer Zeichenfolge in einem Text?
- Wie wird Auto-Vervollständigung in Entwicklungsumgebungen realisiert?
- Woher weiß ein email client, ob eine Emailadresse valide ist?

$\Rightarrow$ **Da sind reguläre Ausdrücke im Spiel**

reguläre Ausdrücke = regular expressions = regex

Was wir uns (vielleicht) schon immer gefragt haben:

- Wie funktioniert Suche nach einer Zeichenfolge in einem Text?
- Wie wird Auto-Vervollständigung in Entwicklungsumgebungen realisiert?
- Woher weiß ein email client, ob eine Emailadresse valide ist?

$\Rightarrow$ **Da sind reguläre Ausdrücke im Spiel**

reguläre Ausdrücke = regular expressions = regex

Unterbau: Formale Sprachen (Theoretische Informatik)

Formale Sprachen

Syntax

- Jede Sprache (natürliche und formale) hat Regeln.
- Für einen Text lässt sich feststellen, ob er zu einer Sprache gehört:
  - Deutsch: Die Studierenden sitzen in der Vorlesung.
  - Python: `print "Hello World"`
- Eine Grammatik beschreibt, nach welchen Regeln eine Sprache syntaktisch aufgebaut ist.
- Bei einer Programmiersprache prüft der Compiler/Interpreter die Syntax.

Semantik

- Auch bei grammatikalisch korrektem Aufbau kann ein Text sinnlos sein
  - Deutsch: Die Vorlesung sitzt in den Studierenden.
  - Python: `print "sin(x) = %f" % cos(x)`
- Eine Semantik-Korrektur für Programmiersprachen gibt es noch nicht.

Formale Grammatik

- Eine formale Grammatik ist ein 4-Tupel $G = (V, \Sigma, P, S)$

Formale Grammatik

- Eine formale Grammatik ist ein 4-Tupel $G = (V, \Sigma, P, S)$
- V ist eine endliche Menge, das Vokabular

Formale Grammatik

- Eine formale Grammatik ist ein 4-Tupel $G = (V, \Sigma, P, S)$
- V ist eine endliche Menge, das Vokabular
- $\Sigma \subset V$ ist das Alphabet, die Elemente heißen Terminalsymbole („Student“, „Vorlesung“; a, b, ...)

Formale Grammatik

- Eine formale Grammatik ist ein 4-Tupel $G = (V, \Sigma, P, S)$
- V ist eine endliche Menge, das Vokabular
- $\Sigma \subset V$ ist das Alphabet, die Elemente heißen Terminalsymbole („Student“, „Vorlesung“; a, b, ...)
- $N = V \backslash \Sigma$ sind die Nichtterminalsymbole/Variablen (<Subjekt>, <Verb>; A, B, ...)

Formale Grammatik

- Eine formale Grammatik ist ein 4-Tupel $G = (V, \Sigma, P, S)$
- V ist eine endliche Menge, das Vokabular
- $\Sigma \subset V$ ist das Alphabet, die Elemente heißen Terminalsymbole („Student“, „Vorlesung“; a, b, ...)
- $N = V \backslash \Sigma$ sind die Nichtterminalsymbole/Variablen (<Subjekt>, <Verb>; A, B, ...)
- X^* ist die Kleensche Hülle der Menge X. Enthält beliebige Konkatenationen von Elementen aus der Menge.

Formale Grammatik

- Eine formale Grammatik ist ein 4-Tupel $G = (V, \Sigma, P, S)$
- V ist eine endliche Menge, das Vokabular
- $\Sigma \subset V$ ist das Alphabet, die Elemente heißen Terminalsymbole („Student“, „Vorlesung“; a, b, ...)
- $N = V \backslash \Sigma$ sind die Nichtterminalsymbole/Variablen (<Subjekt>, <Verb>; A, B, ...)
- X^* ist die Kleensche Hülle der Menge X. Enthält beliebige Konkatenationen von Elementen aus der Menge.
- $P \subset (V^* \backslash \Sigma^*) x V^*$ ist die Menge der Produktionsregeln. Überführung eines Wortes/Texts R, das mindestens ein Nichtterminal enthält ($R \in V^* \backslash \Sigma^*$) in ein beliebiges Wort $Q \in V^*$

```
<Subj><Verb><Obj>  -->  Der Student<Verb><Objt>
AB --> AaB
```

Formale Grammatik

- Eine formale Grammatik ist ein 4-Tupel $G = (V, \Sigma, P, S)$
- V ist eine endliche Menge, das Vokabular
- $\Sigma \subset V$ ist das Alphabet, die Elemente heißen Terminalsymbole („Student“, „Vorlesung“; a, b, ...)
- $N = V \backslash \Sigma$ sind die Nichtterminalsymbole/Variablen (<Subjekt>, <Verb>; A, B, ...)
- X^* ist die Kleensche Hülle der Menge X. Enthält beliebige Konkatenationen von Elementen aus der Menge.
- $P \subset (V^* \backslash \Sigma^*) x V^*$ ist die Menge der Produktionsregeln. Überführung eines Wortes/Texts R, das mindestens ein Nichtterminal enthält ($R \in V^* \backslash \Sigma^*$) in ein beliebiges Wort $Q \in V^*$

```
<Subj><Verb><Obj>  -->  Der Student<Verb><Objt>
AB --> AaB
```

- $S \in (V \backslash \Sigma)$ ist das Startsymbol

Chomsky-Hierarchie

- Eingeführt von Noam Chomsky
- Eine Hierarchie von Klassen formaler Grammatiken

Chomsky-Hierarchie

- Eingeführt von Noam Chomsky
- Eine Hierarchie von Klassen formaler Grammatiken

Typ 0: unbeschränkte Grammatik

- enthält alle formalen Grammatiken
- zugehörige Sprache wird von Turingmaschine akzeptiert

Chomsky-Hierarchie

- Eingeführt von Noam Chomsky
- Eine Hierarchie von Klassen formaler Grammatiken

Typ 0: unbeschränkte Grammatik

- enthält alle formalen Grammatiken
- zugehörige Sprache wird von Turingmaschine akzeptiert

Typ 1: kontextsensitive Grammatik

- Produktionsregeln der Form $\alpha B \gamma \rightarrow \alpha \beta \gamma$ (B Nichtterminal, griechische Buchstaben Worte aus V^*)
- Sprache wird von linear beschränkter Turingmaschine erkannt

Chomsky-Hierarchie

- Eingeführt von Noam Chomsky
- Eine Hierarchie von Klassen formaler Grammatiken

Typ 0: unbeschränkte Grammatik

- enthält alle formalen Grammatiken
- zugehörige Sprache wird von Turingmaschine akzeptiert

Typ 1: kontextsensitive Grammatik

- Produktionsregeln der Form $\alpha B \gamma \rightarrow \alpha \beta \gamma$ (B Nichtterminal, griechische Buchstaben Worte aus V^*)
- Sprache wird von linear beschränkter Turingmaschine erkannt

Typ 2: kontextfreie Grammatik

- Produktionsregeln der Form $A \rightarrow \alpha$
- erkannt von Kellerautomat, Programmiersprachen sind Typ 2

Chomsky-Hierarchie

- Eingeführt von Noam Chomsky
- Eine Hierarchie von Klassen formaler Grammatiken

Typ 0: unbeschränkte Grammatik

- enthält alle formalen Grammatiken
- zugehörige Sprache wird von Turingmaschine akzeptiert

Typ 1: kontextsensitive Grammatik

- Produktionsregeln der Form $\alpha B \gamma \rightarrow \alpha \beta \gamma$ (B Nichtterminal, griechische Buchstaben Worte aus V^*)
- Sprache wird von linear beschränkter Turingmaschine erkannt

Typ 2: kontextfreie Grammatik

- Produktionsregeln der Form $A \rightarrow \alpha$
- erkannt von Kellerautomat, Programmiersprachen sind Typ 2

Typ 3: reguläre Grammatik

- Produktionsregeln der Form $A \rightarrow a$ und $A \rightarrow aB$
- erkannt durch endliche Automaten / reguläre Ausdrücke

Reguläre Grammatik der Exponentialzahlen

- Einschränkende Annahmen:
  - Beide Vorzeichen (Anfang+Exponent) zwingend anzugeben
  - Genau eine Vor- und mindestens eine Nachkommastelle
- $G = (V, \Sigma, P, S)$
- $V = \Sigma \cup N$
- $N = S, M, N1, P, N2, E1, E2$
- $\Sigma = -, +, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, ., e$
- Platzhalter: $z \in \{0 - 9\}$

Produktionsregeln (rechts Beispiel: -3.81e-11)

```
S  -> -M     S -> +M     | S -> -M
M  -> zP                 |   -> -3P
P  -> .N1                |   -> -3.N1
N1 -> zN2                |   -> -3.8N2
N2 -> zN2    N2 -> eE1   |   -> -3.81N2      -> -3.81eE1
E1 -> -E2    E1 -> +E2   |   -> -3.81e-E2
E2 -> zE2    E2 -> z     |   -> -3.81e-1E2   -> -3.81e-11
```

Reguläre Sprachen und Ausdrücke???

Die Menge der regulären Sprachen über einem Alphabet Σ und die zugehörigen regulären Ausdrücke sind rekursiv definiert:

- Die leere Sprache $\emptyset$ ist eine reguläre Sprache und der zugehörige reguläre Ausdruck ist $\emptyset$.
- Der leere String $\{\wedge\}$ ist eine reguläre Sprache und der zugehörige reguläre Ausdruck ist $\wedge$.
- Für jedes a in Σ, ist die einelementige Sprache $\{$ a $\}$ eine reguläre Sprache und der zugehörige reguläre Ausdruck ist a
- Wenn A und B reguläre Sprachen sind mit zugehörigen regulären Audrücken $r1$ and $r2$, dann
  - ist $A \cup B$ (Vereinigung) eine reguläre Sprache und der zugehörige reguläre Ausdruck ist $(r1|r2)$
  - ist AB (Verknüpfung) eine reguläre Sprache und der zugehörige reguläre Ausdruck ist $(r1r2)$
  - ist A^* (Kleene star) eine reguläre Sprache und der zugehörige reguläre Ausdruck ist $(r1^*)$

Regulärer Ausdruck für Exponentialzahlen

- Es gibt unendlich viele Sprachen über jedem endlichen nichtleeren Alphabet
- z.B. für $\Sigma = \{a\}$ sind Sprachen: $\{\}$, $\{a\}$, $\{aa\}$, $\{a, aa\}$, $\{aaa\}$, ...

Regulärer Ausdruck für Exponentialzahlen

- Es gibt unendlich viele Sprachen über jedem endlichen nichtleeren Alphabet
- z.B. für $\Sigma = \{a\}$ sind Sprachen: $\{\}$, $\{a\}$, $\{aa\}$, $\{a, aa\}$, $\{aaa\}$, ...
- Wir wollen zeigen, dass sich die Sprache der Exponentialzahlen und der zugehörige reguläre Ausdruck rekursiv herleiten lassen

Regulärer Ausdruck für Exponentialzahlen

- Es gibt unendlich viele Sprachen über jedem endlichen nichtleeren Alphabet
- z.B. für $\Sigma = \{a\}$ sind Sprachen: $\{\}$, $\{a\}$, $\{aa\}$, $\{a, aa\}$, $\{aaa\}$, ...
- Wir wollen zeigen, dass sich die Sprache der Exponentialzahlen und der zugehörige reguläre Ausdruck rekursiv herleiten lassen
- Alphabet: $\{-, +, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, ., e\}$
- Einelementige Sprachen: $S_- = \{-\}$, $S_+ = \{+\}$, $S_0 = \{0\}$, $S_1 = \{1\}$, ..., $S_. = \{.\}$, $S_e = \{e\}$
- zugehörige regulären Ausdrücke: $r_- = -, r_+ = +, r_0 = 0, r_1 = 1, ..., r_. = ., r_e = e$

Regulärer Ausdruck für Exponentialzahlen

- Es gibt unendlich viele Sprachen über jedem endlichen nichtleeren Alphabet
- z.B. für $\Sigma = \{a\}$ sind Sprachen: $\{\}$, $\{a\}$, $\{aa\}$, $\{a, aa\}$, $\{aaa\}$, ...
- Wir wollen zeigen, dass sich die Sprache der Exponentialzahlen und der zugehörige reguläre Ausdruck rekursiv herleiten lassen
- Alphabet: $\{-, +, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, ., e\}$
- Einelementige Sprachen: $S_- = \{-\}$, $S_+ = \{+\}$, $S_0 = \{0\}$, $S_1 = \{1\}$, ..., $S_. = \{.\}$, $S_e = \{e\}$
- zugehörige regulären Ausdrücke: $r_- = -, r_+ = +, r_0 = 0, r_1 = 1, ..., r_. = ., r_e = e$
- Vereinigung zweier Sprachen: $\{a, b\} \cup \{0, 1, 2\} = \{a, b, 0, 1, 2\} \Rightarrow |S_1| + |S_2|$ Elemente
- Verknüpfung zweier Sprachen: $\{a, b\}\{0, 1, 2\} = \{a0, a1, a2, b0, b1, b2\} \Rightarrow |S_1| \cdot |S_2|$ Elemente
- Kleensche Hülle einer Sprache: $\{0, 1\}^* = \{\emptyset, 0, 1, 00, 01, 10, 11, 000, ...\} \Rightarrow \infty$ Elemente

Erzeugung weiterer Sprachen aus den einelementigen

Operation	Sprache	Ausdruck
$S_- \cup S_+$	$S_A = \{-, +\}$	$r_A = (-\|+)$
$S_0 \cup S_1 \cup$	$S_B = \{0, 1, ..., 9\}$	$r_B = (0\|1\|...\|9) = (0-9)$
$S_A S_B S.$	$S_C = \{-0., ..., -9., ...\}$	$r_C = (-\|+)(0-9).$
$S_C S_B$	$S_D = \{-0.0, -0.1, ...\}$	$r_D = (-\|+)(0-9).(0-9)$
$S_D S_B^*$	$S_E = \{-0.0, -0.00, ...\}$	$r_E = (-\|+)(0-9).(0-9)(0-9)^*$
		$r_E = (-\|+)(0-9).(0-9)^+$
$S_E S_e$	$S_F = \{-0.0e, ...\}$	$r_F = (-\|+)(0-9).(0-9)^+e$
$S_F S_A$	$S_G = \{-0.0e-, ...\}$	$r_G = (-\|+)(0-9).(0-9)^+e(-\|+)$
$S_F S_B$	$S_H = \{-0.0e-0, ...\}$	$r_H = (-\|+)(0-9).(0-9)^+e(-\|+)(0-9)$
$S_H S_B^*$	$S_I = \{-0.0e-00, ...\}$	$r_I = (-\|+)(0-9).(0-9)^+e(-\|+)(0-9)^+$

Erzeugung weiterer Sprachen aus den einelementigen

Operation	Sprache	Ausdruck
$S_- \cup S_+$	$S_A = \{-, +\}$	$r_A = (-\|+)$
$S_0 \cup S_1 \cup$	$S_B = \{0, 1, ..., 9\}$	$r_B = (0\|1\|...\|9) = (0-9)$
$S_A S_B S.$	$S_C = \{-0., ..., -9., ...\}$	$r_C = (-\|+)(0-9).$
$S_C S_B$	$S_D = \{-0.0, -0.1, ...\}$	$r_D = (-\|+)(0-9).(0-9)$
$S_D S_B^*$	$S_E = \{-0.0, -0.00, ...\}$	$r_E = (-\|+)(0-9).(0-9)(0-9)^*$
		$r_E = (-\|+)(0-9).(0-9)^+$
$S_E S_e$	$S_F = \{-0.0e, ...\}$	$r_F = (-\|+)(0-9).(0-9)^+e$
$S_F S_A$	$S_G = \{-0.0e-, ...\}$	$r_G = (-\|+)(0-9).(0-9)^+e(-\|+)$
$S_F S_B$	$S_H = \{-0.0e-0, ...\}$	$r_H = (-\|+)(0-9).(0-9)^+e(-\|+)(0-9)$
$S_H S_B^*$	$S_I = \{-0.0e-00, ...\}$	$r_I = (-\|+)(0-9).(0-9)^+e(-\|+)(0-9)^+$

- S_I ist genau die Sprache der Exponentialzahlen, für die bereits eine Grammatik erstellt wurde
- $r_I = (-|+)(0-9).(0-9)^+e(-|+)(0-9)^+$ ist der zugehörige reguläre Ausdruck

Erzeugung weiterer Sprachen aus den einelementigen

Operation	Sprache	Ausdruck
$S_- \cup S_+$	$S_A = \{-, +\}$	$r_A = (-\|+)$
$S_0 \cup S_1 \cup$	$S_B = \{0, 1, ..., 9\}$	$r_B = (0\|1\|...\|9) = (0-9)$
$S_A S_B S.$	$S_C = \{-0., ..., -9., ...\}$	$r_C = (-\|+)(0-9).$
$S_C S_B$	$S_D = \{-0.0, -0.1, ...\}$	$r_D = (-\|+)(0-9).(0-9)$
$S_D S_B^*$	$S_E = \{-0.0, -0.00, ...\}$	$r_E = (-\|+)(0-9).(0-9)(0-9)^*$
		$r_E = (-\|+)(0-9).(0-9)^+$
$S_E S_e$	$S_F = \{-0.0e, ...\}$	$r_F = (-\|+)(0-9).(0-9)^+e$
$S_F S_A$	$S_G = \{-0.0e-, ...\}$	$r_G = (-\|+)(0-9).(0-9)^+e(-\|+)$
$S_F S_B$	$S_H = \{-0.0e-0, ...\}$	$r_H = (-\|+)(0-9).(0-9)^+e(-\|+)(0-9)$
$S_H S_B^*$	$S_I = \{-0.0e-00, ...\}$	$r_I = (-\|+)(0-9).(0-9)^+e(-\|+)(0-9)^+$

- S_I ist genau die Sprache der Exponentialzahlen, für die bereits eine Grammatik erstellt wurde
- $r_I = (-|+)(0-9).(0-9)^+e(-|+)(0-9)^+$ ist der zugehörige reguläre Ausdruck
- in Python: `rI = r'[-+][0-9]\.[0-9]+e[-+][0-9]+'`

Erzeugung weiterer Sprachen aus den einelementigen

Operation	Sprache	Ausdruck
$S_- \cup S_+$	$S_A = \{-, +\}$	$r_A = (-\|+)$
$S_0 \cup S_1 \cup$	$S_B = \{0, 1, ..., 9\}$	$r_B = (0\|1\|...\|9) = (0-9)$
$S_A S_B S.$	$S_C = \{-0., ..., -9., ...\}$	$r_C = (-\|+)(0-9).$
$S_C S_B$	$S_D = \{-0.0, -0.1, ...\}$	$r_D = (-\|+)(0-9).(0-9)$
$S_D S_B^*$	$S_E = \{-0.0, -0.00, ...\}$	$r_E = (-\|+)(0-9).(0-9)(0-9)^*$
		$r_E = (-\|+)(0-9).(0-9)^+$
$S_E S_e$	$S_F = \{-0.0e, ...\}$	$r_F = (-\|+)(0-9).(0-9)^+e$
$S_F S_A$	$S_G = \{-0.0e-, ...\}$	$r_G = (-\|+)(0-9).(0-9)^+e(-\|+)$
$S_F S_B$	$S_H = \{-0.0e-0, ...\}$	$r_H = (-\|+)(0-9).(0-9)^+e(-\|+)(0-9)$
$S_H S_B^*$	$S_I = \{-0.0e-00, ...\}$	$r_I = (-\|+)(0-9).(0-9)^+e(-\|+)(0-9)^+$

- S_I ist genau die Sprache der Exponentialzahlen, für die bereits eine Grammatik erstellt wurde
- $r_I = (-|+)(0-9).(0-9)^+e(-|+)(0-9)^+$ ist der zugehörige reguläre Ausdruck
- in Python: `rI = r'[-+][0-9]\.[0-9]+e[-+][0-9]+'`
- noch kürzer: `rI = r'[-+]\d\.\d+e[-+]\d+'`

Reguläre Ausdrücke!!!

What does all this mean to you, as a user? Absolutely nothing. As a user, you don't care if it's regular, nonregular, unregular, irregular, or incontinent. So long as you know what you can expect from it, you know all you need to care about. — Jeffrey Friedl

Reguläre Ausdrücke!!!

What does all this mean to you, as a user? Absolutely nothing. As a user, you don't care if it's regular, nonregular, unregular, irregular, or incontinent. So long as you know what you can expect from it, you know all you need to care about. — Jeffrey Friedl

Wo fast jeder sie schon verwendet hat:

- Suche nach einer Zeichenfolge in einem Text
- Tabulator-Vervollständigung
- Mit * eine Gruppe von Dateien auszuwählen (*.txt)

Reguläre Ausdrücke!!!

What does all this mean to you, as a user? Absolutely nothing. As a user, you don't care if it's regular, nonregular, unregular, irregular, or incontinent. So long as you know what you can expect from it, you know all you need to care about. — Jeffrey Friedl

Wo fast jeder sie schon verwendet hat:

- Suche nach einer Zeichenfolge in einem Text
- Tabulator-Vervollständigung
- Mit * eine Gruppe von Dateien auszuwählen (*.txt)

Einschränkungen

- Für Anfänger sehr kryptisch
- Nach reiner Lehre „Zählen“ nicht möglich (z.B. $a^n b^n$ geht nicht)

Reguläre Ausdrücke!!!

What does all this mean to you, as a user? Absolutely nothing. As a user, you don't care if it's regular, nonregular, unregular, irregular, or incontinent. So long as you know what you can expect from it, you know all you need to care about. — Jeffrey Friedl

Wo fast jeder sie schon verwendet hat:

- Suche nach einer Zeichenfolge in einem Text
- Tabulator-Vervollständigung
- Mit * eine Gruppe von Dateien auszuwählen (*.txt)

Einschränkungen

- Für Anfänger sehr kryptisch
- Nach reiner Lehre „Zählen“ nicht möglich (z.B. $a^n b^n$ geht nicht)

Vorteile

- Sehr nützliches Werkzeug zum Finden von Pattern
- Vergleichbare „manuelle“ Implementierung viel aufwändiger
- Python hat Erweiterungen, die sogar „Zählen“ ermöglichen

Reguläre Ausdrücke in Python

Pattern Syntax

- Reguläre Ausdrücke immer als „raw string“
- E.g. `r'([0-9]*\.[0-9]*|[0-9]*)'`

Reguläre Ausdrücke in Python

Pattern Syntax

- Reguläre Ausdrücke immer als „raw string“
- E.g. `r'([0-9]*\.[0-9]*|[0-9]*)'`

`.`	entspricht beliebigem Zeichen außer newline
`^`	entspricht dem Beginn eines strings
`$`	entspricht dem Ende eines strings
`*`	voriger Ausdruck beliebig oft (incl. Null mal) (gierig)
`+`	voriger Ausdruck beliebig oft (excl. Null mal) (gierig)
`?`	voriger Ausdruck Null- oder einmal (gierig)
`*?, +?, ??`	nicht gierige Versionen von `*, +, ?`
`{m}`	voriger Ausdruck genau m Mal
`{m,n}`	voriger Ausdruck m bis n Mal (gierig)
`{m,n}?`	nicht gierige Version von `{m,n}`
`[...]`	ein beliebiges Zeichen aus der Menge (...)
`[^...]`	ein beliebiges Zeichen, das nicht in der Menge ist
`A|B`	A oder B (A und B sind reguläre Ausdrücke)
`(...)`	speichert den gefundenen Inhalt der Klammern

Reguläre Ausdrücke - Maskierungszeichen

- Zeichen mit einer speziellen Bedeutung (`.`, `*`, `...`) können mit ihrer eigentlichen Bedeutung durch voranstellen eines \verwendet werden, z.B. `\.`, `\*`

Reguläre Ausdrücke - Maskierungszeichen

- Zeichen mit einer speziellen Bedeutung (`.`, `*`, `...`) können mit ihrer eigentlichen Bedeutung durch voranstellen eines \verwendet werden, z.B. `\.`, `\*`
- Normale Maskierungszeichen (`\n`, `\t`, `...`) funktionieren wie erwartet, z.B. `r'\n+'` entspricht einem oder mehreren Zeilenumbrüchen

Reguläre Ausdrücke - Maskierungszeichen

- Zeichen mit einer speziellen Bedeutung (`.`, `*`, `...`) können mit ihrer eigentlichen Bedeutung durch voranstellen eines \verwendet werden, z.B. `\.`, `\*`
- Normale Maskierungszeichen (`\n`, `\t`, `...`) funktionieren wie erwartet, z.B. `r'\n+'` entspricht einem oder mehreren Zeilenumbrüchen

`\`*number*	enspricht dem n-ten zuvor gefundenen Text (starting from 1)
`\d`	entspricht `[0-9]`
`\D`	entspricht `[^0-9]`
`\s`	enspricht beliebigem whitespace (= `[\t\n\r\f\v]`)
`\S`	alles außer whitespace
`\w`	beliebiges alphanumerisches Zeichen (= `[a-zA-Z0-9_]`)
`\W`	beliebiges nicht-alphanumerisches Zeichen
`\A`	entspricht dem Beginn eines strings
`\Z`	entspricht dem Ende eines strings

Reguläre Ausdrücke - erste Python-Beispiele

- `findall(patt, string)` – finde alle nicht überlappenden Vorkommen von `patt` in `string`
- `sub(patt, repl, string)` – ersetze Vorkommen durch `repl`

```
import re
regstr = r'\d+\.\d+|\d+'
s = "12.3/5/4.4;5.7;6"
re.findall(regstr,s)
```

Reguläre Ausdrücke - erste Python-Beispiele

- `findall(patt, string)` – finde alle nicht überlappenden Vorkommen von `patt` in `string`
- `sub(patt, repl, string)` – ersetze Vorkommen durch `repl`

```
import re
regstr = r'\d+\.\d+|\d+'
s = "12.3/5/4.4;5.7;6"
re.findall(regstr,s)
```

```
regstr = r'\d*\.\d*|\d*'
s = "12.3/5/4.4;5.7;6"
re.findall(regstr,s)
```

Reguläre Ausdrücke - erste Python-Beispiele

- `findall(patt, string)` – finde alle nicht überlappenden Vorkommen von `patt` in `string`
- `sub(patt, repl, string)` – ersetze Vorkommen durch `repl`

```
import re
regstr = r'\d+\.\d+|\d+'
s = "12.3/5/4.4;5.7;6"
re.findall(regstr,s)
```

```
regstr = r'\d*\.\d*|\d*'
s = "12.3/5/4.4;5.7;6"
re.findall(regstr,s)
```

```
regstr = r'(\d*\.\d*|\d*)'
re.sub(regstr, r'\1xxx', s)
```

Reguläre Ausdrücke - gierig/greedy

- Was bedeutet gierig/nicht-gierig (bzw. greedy/non-greedy)?
- Beispiel:

```
reg_greedy = r'<.*>'       #match as many chars as possible
s = "<H1>title</H1>"
findall(reg_greedy,s)      #whole '<H1>title</H1>' matched
reg_nongreedy = r'<.*?>'   #match as few chars as possible
findall(reg_nongreedy,s)   #only '<H1>' matched
```

- gierig/greedy: so viele Zeichen wie möglich gematched; nur, wenn das schiefgehen würde, gibt es ein backtracking durch die regex engine

Reguläre Ausdrücke - Match-Objekte

- Manche Funktionen aus `re` geben keine strings, sondern „Match-Objekte“(m) zurück
- `match(patt, string)` Sucht nach Übereinstimmung am Beginn des strings
- `search(patt, string)` Sucht die erste Übereinstimmung im string
- `finditer(patt, string)` wie findall, gibt aber einen iterator zurück
- `m.group(i)` Gibt Übereinstimmungsgruppe i zurück ($i \geq 1$)
- `m.groups()` Gibt Tupel mit allen Übereinstimmungsgruppen zurück

```
import re
s = "Ferien␣12/24/2010␣-␣01/06/2011"
patt = r'(\d+)/(\d+)/(\d+)'
for m in re.finditer(patt,s):
  print("%s.%s.%s" % (m.group(2), m.group(1), m.group(3)))
```

Reguläre Ausdrücke - Rätsel zu Primzahlen

Wieso “berechnet” folgender Code Primzahlen?

```
import re

def isPrime(p):
    m = re.search(r'^1?$|^(11+?)\1+$', p)
    if(m):
        return False
    else:
        return True

for i in xrange(0,1024):
    if(isPrime('1' * i)):
        print i
```