# 4 Programmierung mit regulären Ausdrücken

Als *regulären Ausdruck* bezeichnet man ein Textsuchmuster, das mittels eines bestimmten Formalismus spezifiziert wird. Die Bezeichnung „regulärer Ausdruck“ stammt ursprünglich aus der Computerlinguistik und Theoretischen Informatik. Die regulären Ausdrücke der Theoretischen Informatik entsprechen grundsätzlich den hier behandelten regulären Ausdrücken, und beide sind auch grundsätzlich gleich ausdrucksstark. Jedoch sind die hier behandelten regulären Ausdrücke gegenüber denen der Theoretischen Informatik um einige Konstrukte erweitert.

Oberflächlich haben wir reguläre Ausdrücke schon in Abschnitt 2.5.2 im Rahmen der Behandlung des Unix-Kommandos `grep` kennengelernt. Reguläre Ausdrücke, in ähnlicher Form, wie sie in Abschnitt 2.5.2 behandelt wurden, können so auch in Python verwendet werden – Abschnitt 4.1 beschreibt, wie das geht. Abschnitt 4.2 gibt zusätzlich einen umfassenderen Überblick über reguläre Ausdrücke.

Ein gutes Verständnis regulärer Ausdrücke ist für einen Informatiker wichtig: Zum einen kann das Durchdenken, Verstehen und Anwenden regulärer Ausdrücke als „Modell“ für das Verständnis vieler weiterer Konzepte der Informatik betrachtet werden. Zum anderen kann eine sinnvolle Verwendung regulärer Ausdrücke in erstaunlich vielen Situationen eine Menge Programmierarbeit sparen, insbesondere wenn es darum geht, Dateien oder Texte zu manipulieren, zu durchsuchen oder zu analysieren. Es gibt ganze Bücher, die sich ausschließlich dem Thema „reguläre Ausdrücke“ widmen[5], die dem interessierten Leser empfohlen werden können.

## 4.1 Verwendung Regulärer Ausdrücke in Python

Pythons `re`-Modul bietet Funktionen an, reguläre Ausdrücke zu verarbeiten und anzuwenden. Ein Python-Skript, das reguläre Ausdrücke verarbeitet, sollte die Zeile

```
import re
```

enthalten. Reguläre Ausdrücke werden in Form von Strings angegeben. Üblicherweise verwendet man dazu *Raw*-Strings, denen ein „`r`“ vorangestellt ist. Im Gegensatz zu den üblichen Python-Strings interpretieren Raw-Strings den Backslash „\“ nicht, sondern behandeln diesen als einfaches Zeichen – ein erwünschtes Verhalten bei der Spezifikation regulärer Ausdrücke, die häufig Backslashes enthalten. Folgendes Beispiel verdeutlicht den Unterschied zwischen Raw-Strings und herkömmlichen Python-Strings:

```
>>> len('\n')
1
>>> len(r'\n')
2
```

```
>>> r'\n'[0]
'\\'
```

Der Python-String `'\n'` enthält genau ein Zeichen, nämlich das Newline-Zeichen. Dagegen enthält der Raw-String `r'\n'` zwei Zeichen, nämlich den Backslash `\` und das Zeichen `n`. Das Präfix `r` schaltet also die Sonderbedeutung des Backslashs aus. Vergleichbar ist dies mit der Verwendung der einfachen Ticks (`'...'`) in der Bash, die jegliche Interpretation bestimmter Zeichen (in der Bash sind dies etwa `$`, `&`, und `?`) durch die Bash ausschaltet.

Bevor wir im Detail in Abschnitt 4.2 den Aufbau regulärer Ausdrücke darstellen, stellen wir im Folgenden einige Kommandos aus Pythons `re`-Modul vor, die diese verwenden. Wir setzen in diesem Abschnitt die in Abschnitt 2.5.2 vermittelten Kenntnissen über reguläre Ausdrücke voraus.

### 4.1.1 Das Kommando *re.findall*

#### Syntax

Ein Kommando aus dem Modul `re`, das reguläre Ausdrücke verwendet, lautet `re.findall`. Es hat die folgende Syntax:

`re.findall(`⟨*regexp*⟩, ⟨*string*⟩ [,⟨*flags*⟩]`)`

Das Kommando `re.findall` durchsucht den String ⟨*string*⟩ von links nach rechts und liefert die Liste aller maximalen *nicht-überlappenden* Übereinstimmungen des regulären Ausdrucks ⟨*regexp*⟩ im String ⟨*string*⟩. Als *Match* bezeichnet man eine Zeichenfolge, die auf einen bestimmten regulären Ausdruck passt. Wir klären in Kürze, was mit den Eigenschaften „maximal“ und „nicht-überlappend“ gemeint ist.

#### Beispiele

Der reguläre Ausdruck `'[0-9]+ '` passt beispielsweise auf jede Folge von Ziffern denen ein Leerzeichen folgt. Der Aufruf `re.findall('[0-9]+ ', str)` findet folglich alle von einem Leerzeichen gefolgten Ziffernfolgen in einem String.

```
>>> re.findall(r'[0-9]+ ', '99 Luftballons in 10 Zimmern in 1nem Haus')
>>> ['99 ', '10 ']
```

Zwar passt der reguläre Ausdruck `'[0-9]+ '` auch auf die im String enthaltene Zeichenfolge `'0 '`, jedoch wird standardmäßig immer nach maximalen Matches, d. h. Matches maximaler Länge, gesucht, und die Zeichenfolge `'10 '` passt ebenfalls auf den regulären Ausdruck, ist jedoch länger.

**Aufgabe 4.1**

Geben Sie eine Python-Kommandozeile an, die die Summe aller von einem Leerzeichen gefolgten Zahlen (d. h. Ziffernfolgen) eines Strings zurückliefert.

Der reguläre Ausdruck `r'[a-z]+'` passt auf eine nicht-leere Folge von Kleinbuchstaben; der reguläre Ausdruck `r'\s'` passt auf alle Whitespace-Zeichen, d. h. auf Leerzeichen, Tabulatoren, Zeilenumbrüche, usw. Folglich passt der reguläre Ausdruck `r'[a-z]+\s+[a-z]+'` auf alle Wortpaare, die durch ein Whitespace-Zeichen getrennt sind. Folgendes Beispiel

```
>>> re.findall(r'[a-z]+\s+[a-z]+','ein kleiner test mit regexps')
>>> ['ein kleiner', 'test mit']
```

veranschaulicht, was mit „nicht überlappenden Matches“ gemeint ist: Das Kommando `re.findall` durchläuft den String von links nach rechts und liefert alle zu dem regulären Ausdruck passenden Treffer zurück – hierbei wird etwa der Teilstring `'mit regexps'` nicht zurückgeliefert, da `'mit'` bereits im zweiten Treffer `'test mit'` gematcht wurde.

**Aufgabe 4.2**

(a) Verwenden Sie `re.findall` um zu bestimmen, wie viele Ziffern die Datei `test.txt` enthält.
(b) Verwenden Sie `re.findall` um zu bestimmen, wie viele aus Groß- und/oder Kleinbuchstaben bestehende Wörter die Datei `test.txt` enthält.
(c) Verwenden Sie `re.findall`, um alle „inneren Klammerungen“ der Datei `test.txt` zurückzuliefern.

Aus dem String `(Hallo (Welt)), (hier (bin (ich)))` sollten also die Strings `['(Welt)', '(ich)']` extrahiert werden.
(d) Verwenden Sie `re.findall`, um zu bestimmen, wie viele Klammerpaare es in der Datei `test.txt` insgesamt gibt.

### 4.1.2 Das Kommando *re.sub*

**Syntax**

Das Kommando `re.sub` verwendet reguläre Ausdrücke, um Ersetzungen in Strings vorzunehmen. Die Syntax des Kommandos kann folgendermaßen dargestellt werden:

`re.sub(`⟨*regexp1*⟩, ⟨*replacement*⟩, ⟨*string*⟩ [,⟨*flags*⟩]`)`

Das Kommando `re.sub` ersetzt jedes Vorkommen des Musters ⟨*regexp1*⟩ in ⟨*string*⟩

durch den String ⟨*replacement*⟩[1]. Die ersetzten Vorkommen des Musters sind grundsätzlich nicht-überlappend.

**1. Beispiel**
Wir wollen alle Leerzeilen aus einem String entfernen.

**1. (Falscher) Ansatz:** Wir ersetzen alle Stellen, an denen eine Leerzeile auftaucht, d. h. an denen zwei aufeinanderfolgende Newline-Zeichen auftreten, durch ein einziges Newline-Zeichen und verwenden das Kommando `re.sub(r'\n\n','\n', string)`. Dies führt jedoch nicht zum gewünschten Ergebnis:

```
>>> re.sub(r'\n\n','\n','Dies\nist ein\n\n\n String')
>>> 'Dies\nist ein\n\n String'
```

Aus der Newline-Folge `'\n\n\n'` werden durch das Muster nur die ersten beiden Newline-Zeichen gematcht und durch ein einziges Newline-Zeichen ersetzt. Überlappende Matches sind ausgeschlossen. Das übriggebliebene dritte Newline-Zeichen kann nicht mehr durch das Muster gematcht werden, bleibt also stehen. Um das gewünschte Ergebnis zu erhalten, könnte man genau diese Ersetzungs-Aktion so oft wiederholen, bis es keine aufeinanderfolgenden Newline-Zeichen mehr gibt. Es gibt aber eine bessere Möglichkeit:

**2. Richtiger Ansatz**: Wir ersetzen alle Newline-Folgen – dies entspricht dem Muster `r'\n+'` – durch ein einziges Newline-Zeichen. Dies führt zu dem gewünschten Ergebnis:

```
>>> re.sub(r'\n+','\n','Dies\nist ein\n\n\n String')
>>> 'Dies\nist ein\nString'
```

**Aufgabe 4.3**

(a) Verwenden Sie das `re.sub`-Kommando, um alle Kommentare in der Datei `test.py` zu entfernen.
(b) Verwenden Sie das `re.sub`-Kommando, um alle Zeilen der Datei `test.py` auszukommentieren.

1 Hier wäre auch eine Funktion zugelassen, die ein Match-Objekt als Argument erhält und einen String zurückliefert; wir betrachten – da wir uns auf die Verwendung regulärer Ausdrücke konzentrieren wollen – jedoch nur den Fall, dass dieses Argument ein einfacher String ist

**2. Beispiel**
Wir wollen alle Doppelvorkommen von Wörtern, also Wörter, die zweimal direkt hintereinander in einem String auftauchen, suchen und bereinigen. Wir wollen etwa einen String `'das ist ist gut'` in den String `'das ist gut'`, und einen String `'ich ich bin gut'` in `'ich bin gut'` umwandeln. Ein regulärer Ausdruck, der zwei nebeneinander stehende Wörter matcht, ist schnell gefunden – nämlich der Ausdruck `r'[a-zA-Z]+\s+[a-zA-Z]+'`. Wir können jedoch unter Verwendung der bisher vorgestellten Methoden nicht sicherstellen, dass das zweite Wort mit dem ersten identisch ist.

Durch Verwendung von *Rückwärtsreferenzen* ist dies jedoch einfach umzusetzen. Man kann Teile eines regulären Ausdrucks durch Einschließen in runde Klammern gruppieren. Auf diese Gruppierungen kann man in nachfolgenden Teilen eines regulären Ausdrucks (oder im ⟨*replacement*⟩-Argument des `re.sub`-Kommandos) Bezug nehmen. Der reguläre Ausdruck `r'([a-zA-Z]+)\s+\1'` matcht alle Doppelvorkommen von Wörtern. Die Rückwärtsreferenz `'\1'` nimmt Bezug auf den ersten geklammerten Teilausdruck des regulären Ausdrucks und steht für den durch diesen Teilausdruck gematchten String. Wir können alle Doppelvorkommen von Wörtern bereinigen, wenn wir diesen regulären Ausdrucks durch `r'\1'` ersetzen, also auch im Ersetzungs-String auf den geklammerten Teilausdruck Bezug nehmen.

```
>>> re.sub(r'([a-zA-Z]+)\s+\1\b',r'\1',
           'hallo hallo ich ich stottere')
>>> 'hallo ich stottere'
```

Der Ausdruck `'b'` matcht den Anfang oder das Ende eines Wortes. Ohne das `'b'` nach dem `'1'` hätten wir nicht sichergestellt, dass die durch `'1'` gemachte Zeichenfolge nicht nur den Anfang eines längeren Wortes darstellen würde.

**Aufgabe 4.4**

Diese Lösung löscht jedoch keine Dreier-, Vierer- usw. Vorkommen von Wörtern. Erweitern Sie die Lösung entsprechend so, dass auch diese Fälle bereinigt werden.

**Aufgabe 4.5**

(a) Verwenden Sie das `re.sub`-Kommando, um alle runden Klammern in einem String zu entfernen.
(b) Verwenden Sie das `re.sub`-Kommando, um alle *inneren* runden Klammern in einem String zu entfernen.

(c) Verwenden Sie das `re.sub`-Kommando, um alle objektorientierten Methodenaufrufe in Funktionsaufrufe umzuwandeln, die das Objekt als erstes Argument erhalten. Beispiel: `'xy.findall(pat,str)'` soll umgewandelt werden in `'findall(xy,pat,str)'` oder `'[1,2,3].count(1)'` soll umgewandelt werden in `'count([1,2,3],1)'`.

### 4.1.3 Das Kommando *re.search*

**Syntax**

Das Kommando `re.search` bietet eine allgemeinere Schnittstelle zum Suchen regulärer Ausdrücke. Die Syntax dieses Kommandos lässt sich wie folgt darstellen:

`re.search(`⟨*regexp*⟩, ⟨*string*⟩, [,⟨*flags*⟩]`)`

Diese Methode läuft durch den String ⟨*string*⟩ und sucht nach Stellen, in denen das Muster ⟨*regexp*⟩ passt. Das Kommando liefert ein *Match-Objekt* zurück, das Informationen über den oder die Matches enthält. Ein Match-Objekt bietet die in folgender Tabelle gezeigten Methoden an:

| Methode | Rückgabewert |
|---|---|
| *group*() | Liefert den String, der durch den ⟨*regexp*⟩ gematcht wurde; |
| *start*() | liefert die Startposition des Matches; |
| *end*() | liefert die Endposition des Matches. |

**Beispiel**

Wir wollen in einem Text alle Python-Listen (angegeben als Konstanten) finden[2]. Ein passender regulärer Ausdruck

```
r'\[([^,\]]+)((,[^,\]]+)*)\]'
```

ist recht komplex. Insbesondere in diesem Fall bietet es sich an, in den regulären Ausdruck Kommentare einzufügen. Hierfür verwendet man einen zeilenübergreifenden String (eingeschlossen in jeweils drei Anführungszeichen am Anfang und Ende) und setzt das Flag `re.X` – erst durch Setzen dieses Flags können Kommentare in regulären Ausdrücken verarbeitet werden:

2 Unter der Annahme, die Python-Listen sind nicht geschachtelt. Reguläre Ausdrücke sind nämlich grundsätzlich nicht in der Lage mit geschachtelten Strukturen umzugehen, d. h. diese zu erkennen und zu analysieren.

```
>>> m = re.search(r'''\[ # Listen-Anfang
            ([^,\]]+)     # 1. Element: Alle Zeichen bis , oder ] kommt.
            ((,[^,\]]+)*) # Restliste
            \]            # Listen-Ende
            ''',string, re.X)}
```

`'\['` matcht den Listenanfang; der reguläre Ausdruck `'[^,\]]'` matcht das erste Element der Liste – also die Folge der Zeichen, bis zum nächsten `','` oder `']'`. Das erste Element sowie die Restliste sind zusätzlich durch runde Klammern gruppiert.

Die Variable `m` enthält ein Match-Objekt, das die in obiger Tabelle vorgestellten Methoden anbietet. Nehmen wir an, die erste in `string` enthaltene Python-Liste ist `[1,2,3]`, so ergibt sich:

```
>>> m.group()
>>> '[1,2,3]'
```

Werden der Methode `group` keine Argumente übergeben, so wird der gesamte, durch den regulären Ausdruck gematchte String zurückgeliefert. Enthält der reguläre Ausdruck Gruppierungen (d. h. durch runde Klammern gruppierte Teile des regulären Ausdrucks), so kann auf die `i`-te Gruppe durch Aufruf von `group(i)` zugegriffen werden. Der erste durch runde Klammern gruppierte reguläre Ausdruck – nämlich `'[^,\]]+'` – matcht das erste Listenelement:

```
>>> m.group(1)
'1'
```

Der zweite durch runde Klammern gruppierte reguläre Ausdruck – nämlich `'(,[^,\]+)*)'` – matcht die restlichen Listenelemente:

```
>>> m.group(2)
',2,3'
```

**Aufgabe 4.6**

Was liefert der Aufruf `m.group(3)` als Ergebnis? Erklären Sie!

## 4.2 Komponenten Regulärer Ausdrücke

In Abschnitt 4.2.1 wiederholen wir die einfachsten Komponenten regulärer Ausdrücke, die wir teilweise schon in früheren Abschnitten verwendet haben. In Abschnitt 4.2.2 gehen wir auf die Möglichkeit der Verwendung von Rückwärtsreferenzen ein, die wir auch schon im vorigen Abschnitt verwendet haben. In Abschnitt 4.2.3

klären wir den Unterschied zwischen Greedy- und Non-Greedy-Wiederholungsoperatoren und in Abschnitt 4.2.4 beschreiben wir die Verwendung von Lookahead- und Lookbehind-Operatoren.

Alle hier vorgestellten regulären Ausdrücke können in Form von Raw-Strings mit den im letzten Abschnitt vorgestellten Befehlen `re.findall`, `re.search` und `re.sub` verwendet werden.

### 4.2.1 Einfache Konstrukte

Ein Teil der in diesem Abschnitt vorgestellten regulären Ausdrücke wurden bereits in Kapitel 2.5.2 kurz beschrieben. Dieser Abschnitt hier gibt einen systematischeren und umfassenderen Einblick in die Möglichkeiten, reguläre Ausdrücke zu erstellen.

Viele Konstrukte, die wir im Folgenden wiederholen und neu kennenlernen werden, sind eigentlich *Kombinatoren* regulärer Ausdrücke, d. h. sie können auf einen – manchmal auch auf mehreren – bestehenden regulären Ausdrücken angewandt werden und erzeugen daraus einen komplexeren regulären Ausdruck. Im Folgenden sei $\langle re \rangle$ stets ein Platzhalter für einen beliebigen regulären Ausdruck und $\langle re_1 \rangle$, $\langle re_2 \rangle$, … seien Platzhalter für mehrere reguläre Ausdrücke, falls an einem Konstrukt mehrere reguläre Ausdrücke beteiligt sind.

#### Spezielle Zeichen und Positionen

`.` Matcht jedes Zeichen außer dem Newline-Zeichen `'\n'`. Falls das Python-Kommando, das den regulären Ausdruck verwendet, das `re.DOTALL`-Flag setzt, dann matcht `'.'` auch das Newline-Zeichen.

**Beispiele:**

`1.2` : Matcht jedes Vorkommen einer `'1'` gefolgt von einem beliebigen Zeichen (außer dem Newline-Zeichen) gefolgt von einer `'2'`.

`1\.2` : Matcht den String `'1.2'`.

`^` Matcht den Anfang eines Strings. Wenn das `re.MULTILINE`-Flag gesetzt wird, dann matcht `'^'` zusätzlich jede Position *nach* dem Newline-Zeichen `'\n'`, also Zeilenanfänge. Der reguläre Ausdruck `'^'` matcht also kein bestimmtes Zeichen bzw. keine bestimmte Zeichenfolge, sondern eine *Position* innerhalb eines Strings.

**Beispiele:**

`^a` : Matcht alle `'a'`s, die an einem Zeilenanfang stehen, vorausgesetzt das `re.MULTILINE`-Flag ist gesetzt. Andernfalls matcht dieser reguläre Ausdruck den entsprechenden String nur dann, wenn der mit einem `'a'` beginnt.

`^...x` : Matcht alle Zeilen, deren viertes Zeichen ein `'x'` ist, vorausgesetzt das `re.MULTILINE`-Flag ist gesetzt. Andernfalls matcht dieser reguläre Ausdruck den entsprechenden String nur dann, wenn dessen viertes Zeichen ein `'x'` ist.

`$` Matcht das Ende eines Strings. Wenn das `re.MULTILINE`-Flag gesetzt wird, dann matcht `'$'` zusätzlich jede Position vor einem Newline-Zeichen `'\n'`, also Zeilenenden.

**Beispiele:**

`^...$` : Matcht alle Zeilen, die aus genau drei Zeichen bestehen, vorausgesetzt das `re.MULTILINE`-Flag ist gesetzt. Andernfalls matcht dieser reguläre Ausdruck nur Strings, die aus genau drei Zeichen bestehen.

`^$` : Matcht alle Leerzeilen, vorausgesetzt das `re.MULTILINE`-Flag ist gesetzt. Andernfalls matcht dieser reguläre Ausdruck nur den leeren String.

`\s` Matcht Whitespace-Zeichen, also Leerzeichen, Tabulatoren, Newline-Zeichen, usw.

**Beispiele:**

`\s[0-9][0-9]\s` : Matcht alle zweistelligen Ganzzahlen, die durch Whitespace-Zeichen begrenzt sind.

`\ser\s` : Matcht alle alleinstehenden Wörter `'er'`, die durch Whitespace-Zeichen begrenzt sind. Dagegen wird das `'er'` als Vorsilbe, wie in `'erscheinen'`, `'erlauben'` usw., so nicht gematcht; auch Vorkommen von `'er'`, auf denen ein Komma oder ein Satzendezeichen folgt, werden so nicht gematcht.

`\S` Matcht Nicht-Whitespace-Zeichen, also alles *außer* Leerzeichen, Tabulatoren, Newline-Zeichen usw.

`\w` Matcht alphanumerische Zeichen. Dieser reguläre Ausdruck ist äquivalent mit `[a-zA-Z0-9 ]`.

`\W` Matcht alle Zeichen, die nicht alphanumerisch sind.

**Beispiel:**

`\W\w+\W` : Matcht ein Wort, also eine nicht leere Folge alphanumerischer Zeichen `'\w+'`, begrenzt durch ein nicht-alphanumerisches Zeichen `'\W'`.

`\b` Matcht keine Zeichen, sondern Positionen in einem String – und zwar alle Positionen, die sich am Anfang oder Ende eines Wortes befinden. Als Wort gilt hierbei eine Folge alphanumerischer Zeichen (inklusive des `'_'`-Zeichens.). Der reguläre Ausdruck `'\b'` matcht also immer die Position zwischen einem `'\w'`- und einem `'\W'`-Zeichen.

**Beispiele:**

`\ba[a-z]+` : Matcht alle Wörter, die mit einem `'a'` beginnen.

`\ba\w*a\b` : Matcht alle Wörter, die mit einem `'a'` beginnen und mit einem `'a'` enden.

**Aufgabe 4.7**

Sie möchten alle Wörter matchen, die mit einem `'e'` beginnen und mit einem `'e'` enden. Erklären Sie, was dagegen spricht, den regulären Ausdruck

```
r'\\be.*e\\b'
```

zu verwenden.

**Aufgabe 4.8**

Geben Sie einen regulären Ausdruck an, der …

(a) alle Zeilen matcht, die genau ein Wort enthalten.
(b) alle Wörter matcht, die kein `'a'` enthalten.
(c) alle Wörter matcht, die mit einem `'a'` beginnen.

### Wiederholungen

⟨*re*⟩`*` Der `'*'` bezieht sich immer auf den vorhergehenden regulären Ausdruck ⟨*re*⟩ und matcht immer eine beliebig lange Folge (auch Nullfolge) des regulären Ausdrucks ⟨*re*⟩.

**Beispiele:**

`a*` : Matcht beliebig lange `'a'`-Folgen – dies schließt auch den leeren String als Nullfolge ein.

`(aa)*` : Matcht alle `'a'`-Folgen mit einer geraden Anzahl `'a'`s – dies schließt wiederum den leeren String ein.

⟨*re*⟩`+` Dieser Wiederholungsoperator verhält sich ähnlich wie `'*'`, nur werden hier Folgen der Länge von mindestens eins gematcht.

⟨*re*⟩? Auch `'?'` bezieht sich immer auf den vorhergehenden regulären Ausdruck ⟨*re*⟩ und matcht immer höchstens eine (oder keine) Wiederholung des regulären Ausdrucks ⟨*re*⟩.

**Beispiele:**

`re?` : Matcht entweder den String `'r'` oder den String `'re'`.

`r?e?` : Matcht entweder den leeren String, den String `'r'`, den String `'e'`, oder den String `'re'`.

`</?p>` : Matcht einen HTML-Paragraph-Tag, und zwar entweder den öffnenden `<p>`, oder den schließenden `</p>`.

⟨*re*⟩{*m*} Matcht genau *m* Kopien des vorhergehenden regulären Ausdrucks ⟨*re*⟩.

**Beispiel.**

`re{3}` : Matcht `'reee'`, jedoch weder `'ree'` noch `'reeee'`.

`\b\w{5}+\b` : Matcht alle Worte, deren Länge durch fünf teilbar ist.

`\b(\w{5}?\w{7}?)*\b` : Matcht alle Wörter *s* mit $len(s) = 5 \cdot k + 7 \cdot k'$, mit $k, k' \in \mathbb{N}_0$, d. h. deren Länge eine Linearkombination der Zahlen 5 und 7 ist.

⟨*re*⟩{*n*, *m*} Matcht mindestens *n* und höchstens *m* Wiederholungen des vorhergehenden regulären Ausdrucks ⟨*re*⟩. Grundsätzlich werden durch diesen regulären Ausdruck immer möglichst viele Wiederholungen gematcht. Gleiches gilt auch für die Operatoren `'*'`, `'+'` und `'?'`; sie alle haben die Eigenschaft, *greedy* (engl. für „gierig“) zu sein, d. h. soviel Zeichen wie möglich zu matchen.

**Beispiel.**

`(re){2,3}` : Matcht die Strings `'rere'` und `'rerere'`.

`\b\w{1,3}\b` : Matcht alle Worte, die eine Länge von höchstens drei haben.

**Aufgabe 4.9**

Zählen Sie alle Strings auf, die durch den regulären Ausdruck `r'(ab?)?a?'` gematcht werden.

**Aufgabe 4.10**

Geben Sie einen regulären Ausdruck an, der …

(a) alle Wörter matcht, die mindestens 3 und höchstens 5 Buchstaben haben.
(b) alle Wörter matcht, die *genau* ein `'a'` enthalten.
(c) alle Wörter matcht, die *genau* zwei `'a'` enthalten.

**Aufgabe 4.11**

Geben Sie einen regulären Ausdruck an, der …

(a) alle Wörter matcht, die mehr als 10 Zeichen enthalten.
(b) alle Zeilen matcht, die ein Wort enthalten, das eine Länge von mindestens 10 Zeichen hat.
(c) alle Zeilen matcht, die keine Wörter enthalten, die eine Länge von mehr als 5 Zeichen haben.

### Alternativen

`[`⟨*charlst*⟩`]` Dieser reguläre Ausdruck matcht *eines* der Zeichen aus ⟨*charlst*⟩. Neben einzelnen Zeichen können in ⟨*charlst*⟩ unter Verwendung von „–“ auch Zeichen-Bereiche angegeben werden.

**Beispiele:**
`[aeiou]` : Matcht einen kleinen Vokal.
`[0-9]` : Matcht eine Ziffer.
`[()$]` : Matcht eines der Zeichen `'('`, `')'`, oder `'$'`.

Wie man am letzten Beispiel sieht, werden die meisten Sonderzeichen innerhalb der eckigen Klammern nicht interpretiert, sondern stellen einfach die entsprechenden Zeichen dar.

`[^`⟨*charlst*⟩`]` Das innerhalb der eckigen Klammern verwendete `'^'` steht für eine Negation: Dieser reguläre Ausdruck matcht alle Zeichen *außer* den in ⟨*charlst*⟩ angegebenen.

**Beispiele:**

`[^0-9]` : Matcht alle Zeichen außer Ziffern.

`\([^)]*\)` : Matcht einen Klammerausdruck, beginnend mit einer öffnenden Klammer `'('` (da das Zeichen `'('` innerhalb eines regulären Ausdrucks eine Sonderbedeutung hat, muss ein Backslash vorangestellt werden, um diese Sonderbedeutung auszuschalten), gefolgt von einer Folge von nicht-`')'`-Zeichen, gefolgt von einer schließenden Klammer. Da innerhalb der eckigen Klammern grundsätzlich die Sonderbedeutungen der Zeichen ausgeschaltet sind, braucht hier der schließenden Klammer kein Backslash vorangestellt werden.

`[^ ]+` : Matcht eine Folge von nicht Leerzeichen, d. h. kann in bestimmten Situationen dazu verwendet werden, ein Wort zu matchen – meist gibt es aber bessere Möglichkeiten hierzu.

$\langle re_1 \rangle | \langle re_2 \rangle$ Erzeugt einen regulären Ausdruck, der entweder den regulären Ausdruck $\langle re_1 \rangle$, oder den regulären Ausdruck $\langle re_2 \rangle$ matcht.

**Beispiele:**

`meier|meyer` : Matcht entweder den String `'meier'`, oder `'meyer'`.

`[0-9]+|[a-z]+` : Matcht entweder eine Ziffernfolge oder eine Folge von Kleinbuchstaben.

`\b(1[0-2]|[1-9])\b` : Matcht eine Zahl zwischen 1 und 12.

**Aufgabe 4.12**

Wahr oder falsch? – Bitte begründen Sie jeweils.

(a) `r'(re)*'` matcht genau die gleichen Strings wie `r'(re)*(re)?'`
(b) `r'[re]*'` matcht genau die die gleichen Strings wie `r'((re)?)*'`
(c) `r'[re]*'` matcht genau die gleichen Strings wie `r'(r?e?)*'`
(d) `r'((re)*)*'` matcht genau die gleichen Strings wie `r're*'`
(e) `r'((re)*)*'` matcht genau die gleichen Strings wie `r'(re)*'`
(f) `r'(a?b?)?'` matcht genau die gleichen Strings wie `r'a?b?'`

**Aufgabe 4.13**

Geben Sie einen regulären Ausdruck an, der …

(a) alle Wörter matcht, die entweder mit `'a'` anfangen oder mit `'z'` enden.
(b) alle Zeilen matcht, die entweder ein `'a'` oder ein `'z'` enthalten.

**Aufgabe 4.14**

Geben Sie einen regulären Ausdruck `s` an, der alle äußeren Klammerungen (mit runden Klammern) matcht. Beispielsweise sollte gelten:

```
>>> re.findall(s,'(al(l)e) Klammern (sollten (ge(funden))) werde(n)')
>>> ['(al(l)e)', '(sollten (ge(funden)))', '(n)']
```

**Aufgabe 4.15**

Geben Sie einen regulären Ausdruck an, der …

(a) Zahlen zwischen 1 und 24 matcht.
(b) Zahlen zwischen 1 und 31 matcht.
(c) Zahlen zwischen 1 und 365 matcht.
(d) Zahlen zwischen 0 und 59 matcht.

### 4.2.2 Rückwärtsreferenzen (Backreferences)

In einigen Fällen ist es notwendig, auf vorher gematchte Teile zu einem späteren Zeitpunkt bzw. in einem weiter rechts stehenden Teil des regulären Ausdrucks Bezug zu nehmen. Dazu muss man wissen, dass immer dann, wenn ein Teil eines regulären Ausdrucks mittels der einfachen runden Klammern `(...)` gruppiert wird, zugleich eine Referenz definiert wird, auf die man später mittels einer sogenannten Backreference Bezug nehmen kann.

Folgende Konstrukte sind für die Verwendung von Backreferences relevant:

| | |
|---|---|
| $(\langle re \rangle)$ | Matcht den regulären Ausdruck $\langle re \rangle$. Die runden Klammern haben eine spezielle Bedeutung: sie markieren den Beginn und das Ende einer Gruppierung. Auf den durch die Gruppierung gematchten String kann im Weiteren mittels einer Rückwärtsreferenz zugegriffen werden. |
| $(?:\langle re \rangle)$ | Matcht den regulären Ausdruck $\langle re \rangle$. Im Unterschied zu den „einfachen“ Klammern $(\langle re \rangle)$ kann auf den mittels $(?:\langle re \rangle)$ gematchten String nicht mehr über Rückwärtsreferenzen zugegriffen werden. |
| $\backslash\langle zahl \rangle$ | Stellt eine Rückwärtsreferenz (engl.: Backreference) dar und steht immer für den durch die $\langle zahl \rangle$-te Gruppierung gematchten String. |

**Beispiele:**

`\b(zu|auf)\w*\b`: Matcht alle Wörter, die entweder mit `'zu'` oder mit `'auf'` beginnen.

`\b(?:zu|auf)\w*\b`: Matcht, genau wie im vorigen Beispiel, alle Wörter, die entweder mit `'zu'` oder mit `'auf'` beginnen; nur dass jetzt keine referenzierbare Gruppierung erzeugt wird.

`\b(\w+)\1\b`: Matcht alle Wörter die aus zwei identischen Teilwörtern bestehen, beispielsweise `'wikiwiki'` oder `'wauwau'`.

`b([0-9])\1*([0-9])(?:\1?\2?)*b`: Matcht alle Zahlen, die höchstens 2 unterschiedliche Ziffern enthalten.

`re.sub(r'\(([^)]*)\)',r'-- \1 --',s)`: Ersetzt Klammern durch Gedankenstriche – unter der Voraussetzung, die Klammern seien nicht geschachtelt. Die Rückwärtsreferenz bezieht sich dabei auf den Teil `'[^)]*'` des regulären Ausdrucks.

**Aufgabe 4.16**

(a) Erweitern Sie das `re.sub`-Kommando im letzten Beispiel so, dass ein Klammereinschub am Ende eines Satzes lediglich durch *einen* Gedankenstrich ersetzt wird. So sollte beispielsweise der String

```
'Alle (bis auf wenige) werden ersetzt (ok)? Gut so.'
```

ersetzt werden durch

```
'Alle -- bis auf wenige -- werden ersetzt -- ok? Gut so.'
```

(b) Funktioniert der reguläre Ausdruck im letzten Beispiel auch mit geschachtelten Klammerausdrücken? Was passiert, wenn das `re.sub`-Kommando aus diesem Beispiel auf den String

```
'Das sind (sollte man aber (fast) nie machen) Schachtelungen'
```

angewendet wird?

**Aufgabe 4.17**

Geben Sie einen regulären Ausdruck an, der alle Wörter matcht, die entweder kein `'a'` oder kein `'z'` enthalten.

**Aufgabe 4.18**

Verwenden Sie einen regulären Ausdruck, um allen Referenzen der Form `\ref{`⟨*label*⟩`}` einer LaTeX-Datei zusätzlich eine Seitenreferenz der Form `\pageref{`⟨*label*⟩`}` hinzuzufügen.

### 4.2.3 Greedy vs. Non-Greedy

Alle bisher behandelten Wiederholungsoperatoren sind *greedy* (engl.: gierig). Das bedeutet, sie versuchen, eine möglichst *lange* Sequenz von Zeichen zu matchen. Angenommen, wir möchten alle mit dem Klammerpaar `(...)` eingeklammerten Wörter matchen. Der reguläre Ausdruck `r'\(.*\)'` ist aufgrund der Greedy-Eigenschaft des Wiederholungsoperators hierfür nicht geeignet. Angewandt beispielsweise auf den String

```
'Jetzt wird (mehrmals) geklammert (ok?)!'
```

wird der Teil `'(mehrmals) geklammert (ok?)'` gematcht.

Es gibt gelegentlich Situationen, da möchte man, dass Wiederholungsoperatoren eine möglichst *kurze* Sequenz von Zeichen matchen. Für jeden der bisher behandelten Wiederholungsoperatoren gibt es eine Version, die non-greedy ist:

⟨*re*⟩`*?` Non-Greedy-Version des Wiederholungsoperators ⟨*re*⟩`*`;

⟨*re*⟩`+?` Non-Greedy-Version des Wiederholungsoperators ⟨*re*⟩`+`;

⟨*re*⟩`??` Non-Greedy-Version des Wiederholungsoperators ⟨*re*⟩`?`;

⟨*re*⟩`{`*n*`,`*m*`}?` Non-Greedy-Version des Wiederholungsoperators ⟨*re*⟩`{`*n*`,`*m*`}`.

**Beispiele:**

`'\(.*?\)'`: Matcht alle mit dem Klammerpaar `(...)` eingeklammerten Einschübe.

`'<p>.*?</p>'`: Matcht Paragraphen einer HTML-Datei; ein Paragraph beginnt immer mit dem *Tag* `'<p>'` und endet mit dem Tag `'</p>'`. Die Verwendung der Non-Greedy-Version des Wiederholungsoperators stellt hier sicher, dass immer nur bis zum nächsten Vorkommen von `'</p>'` gematcht wird. Eine Beispielanwendung:

```
>>> htmltxt = '''<p> Erster Paragraph </p>
...             <p> Zweiter Paragraph </p>'''
>>> re.findall(r'<p>.*?</p>',htmltxt)
>>> ['<p> Erster Paragraph </p>',
     '<p> Zweiter Paragraph </p>']
```

**Aufgabe 4.19**

Welche Matches würde es im Text

```
'<p> Erster Paragraph </p> <p> Zweiter Paragraph </p>'
```

geben, wenn versehentlich die Greedy-Variante des Wiederholungsoperators `'*'` verwendet worden wäre?

**Aufgabe 4.20**

Erklären sie folgende Ausgabe:

```
>>> re.findall(r'r??e??','re')
['', '', '']
```

**Aufgabe 4.21**

Gibt es einen String, der durch den regulären Ausdruck `'r?e?'` einen Match liefert, nicht jedoch durch `'r??e??'`?

### 4.2.4 Lookahead

Manchmal ist es notwendig, ein oder mehrere Zeichen im Eingabestring vorauszuschauen und einen String nur dann matchen zu lassen, wenn diese Zeichen bestimmte Eigenschaften erfüllen. Eine solche Vorausschau wird in der Informatik (insbesondere in der Computerlinguistik und im Compilerbau) als *Lookahead* bezeichnet. Ebenso ist es gelegentlich notwendig, ein oder mehrere Zeichen im Eingabestrom zurückzuschauen und die aktuelle Stelle eines Strings nur dann matchen zu lassen, wenn diese Zeichen bestimmte Eigenschaften erfüllen. Dieses Zurückschauen wird auch als *Lookbehind* bezeichnet.

Man sollte sich stets bewusst sein, dass alle im Folgenden vorgestellten Lookahead- und Lookbehind-Operatoren *keine* Zeichen matchen, sondern lediglich Positionen innerhalb eines Strings.

`(?=`⟨*re*⟩`)` Dieser reguläre Ausdruck matcht genau dann an der aktuellen Stelle im Eingabestring, wenn die nachfolgenden Zeichen von ⟨*re*⟩ gematcht werden. Diese Zeichen sind jedoch kein Teil des Matches und werden von diesem regulären Ausdruck nicht „verbraucht".

**Beispiele:**

`\b\w*\b(?=[.!?;])` : Matcht Wörter am Satzende. Ob das Wort mit einem Satzende-Zeichen endet, wird mittels des Lookaheads `(?=[.!?;])` überprüft. Der Match beinhaltet das Satzendezeichen selbst nicht – da es sich um ein Lookahead handelt, der lediglich Positionen, jedoch keine Zeichen matcht.

`\b(?=ein|da|w)\w{5,7}\b` : Die Lookahead-Möglichkeit kann dazu verwendet werden, nacheinander mehrere Tests auf einer Zeichenkette durchzuführen. Dieser reguläre Ausdruck matcht alle Wörter, die entweder mit `'ein'` oder `'da'` oder `'w'` beginnen (überprüft durch Lookahead) *und* deren Länge 5 und 7 Zeichen ist. Ohne Verwendung der Lookahead-Prüfung wäre ein sehr viel komplexerer regulärer Ausdruck notwendig.

`(?<=`⟨*re*⟩`)` Dieser reguläre Ausdruck matcht genau dann an der aktuellen Stelle im Eingabestring, wenn die vorhergehenden Zeichen von ⟨*re*⟩ gematcht werden. Diese Zeichen wurden evtl. bereits von einem vorhergehenden regulären Ausdruck gematcht und sind in diesem Fall kein Teil des Matches. Da hier nicht vorausgeschaut, sondern nach „hinten" geschaut wird, bezeichnet man `(?<=`⟨*re*⟩`)` oft als *Lookbehind*-Operator.

**Beispiel:**

`(?<=\^|\s)[0-9]+(?=$|\s)` : Matcht alle einzelstehenden Dezimalzahlen in einem Text.

`(?!`⟨*re*⟩`)` Dieser reguläre Ausdruck matcht genau dann an der aktuellen Stelle im Eingabestring, wenn die nachfolgenden Zeichen den regulären Ausdruck ⟨*re*⟩ *nicht* matchen. Es handelt sich also um einen negierten Lookahead-Operator.

**Beispiele:**

`\b(der|die|das)(?!\s+[A-Z])` : Matcht alle Vorkommen eines der Artikel `der`, `die` oder `das`, auf den *kein* großgeschriebenes Wort folgt.

`(?<!`⟨*re*⟩`)` Dieser reguläre Ausdruck matcht genau dann an der aktuellen Stelle im Eingabestring, wenn die vorhergehenden Zeichen ⟨*re*⟩ *nicht* matchen – es handelt sich also um einen negierten Lookbehind-Operator.

**Beispiele:**

`(?<!,\s)\bdass\b` : Matcht Vorkommen des Wortes `'dass'`, sofern dieses nicht auf ein Komma folgt.

**Aufgabe 4.22**

Geben Sie einen regulären Ausdruck an, der …

(a) alle Wörter matcht, die nicht mit einem `'a'` beginnen.
(b) alle Wörter matcht, denen ein großgeschriebenes Wort folgt.
(c) alle Wörter matcht, außer dem Wort `'Wort'`.
(d) alle Wörter matcht, die nicht die Zeichenfolge `'reg'` als Teilwort enthalten.
(e) alle Wörter matcht, die entweder mit `'re'` oder `'erg'` beginnen, das Teilwort `'reg'` enthalten und eine Länge von höchstens 10 haben.
(f) alle Wörter matcht, die von Whitespace-Zeichen eingeschlossen sind.
(g) alle Wörter matcht, die nicht von Whitespace-Zeichen eingeschlossen sind.

**Aufgabe 4.23**

Geben Sie einen regulären Ausdruck *ohne* Lookahead-Operatoren an, der alle Wörter matcht, die mit einem `'ein'` oder `'da'` oder `'w'` beginnen *und* deren Länge zwischen 5 und 7 Zeichen beträgt.

**Aufgabe 4.24**

Eine Möglichkeit, in einem LaTeX-Dokument Umlaute zu erzeugen, ist die, das Zeichen `'"'` vor den entsprechenden Nicht-Umlautvokal zu stellen: Ein „Ü“ wird in LaTeX also durch die Zeichenfolge `"U` erzeugt; ein „ä“ wird in LaTeX durch Zeichenfolge `"a` erzeugt, usw.

Angenommen, Sie wollen alle Wörter in einem LaTeX-Dokument finden. Der reguläre Ausdruck `r'\b\w+\b'` ist aber leider wegen der möglicherweise in einem Wort enthaltenen `"`-Zeichen nicht geeignet. Geben Sie einen geeigneten regulären Ausdruck an, der alle in einem LaTeX-Dokument enthaltenen Wörter matcht.

**Aufgabe 4.25**

(a) Erzeugen Sie die Liste aller `for`-Schleifen-Variablen des Python-Skripts `test.py`.
(b) Geben Sie alle im Python-Skript `test.py` enthaltenen statischen Integer-Listen aus (d. h. es sollen keine dynamisch erzeugte Listen betrachtet werden, sondern nur als Konstanten angegebene Listen).
(c) Geben Sie alle Zuweisungen eines Strings (mit einfachen Anführungszeichen, also ¸...¸ oder "...") an eine Variable in einem Pythonskript an. Zurückgegeben werden sollen Tupel, bestehend aus der Variablen und dem jeweiligen String.

## 4.3 Reguläre Ausdrücke vs. Suchausdrücke mit Listenkomprehensionen

Nicht immer benötigt ein Python-Skript reguläre Ausdrücke um Operationen auf Strings auszuführen oder Eigenschaften von Strings zu untersuchen. Oft jedoch werden Lösungen durch Verwendung regulärer Ausdrücke vereinfacht und meistens werden Lösungen durch Verwendung regulärer Ausdrücke performanter.

Grundsätzlich kann man sagen, dass Lösungen über Listenkomprehensionen und/oder einfache String-Funktionen oft dann einfacher sind als die Verwendung regulärer Ausdrücke, wenn die zu findenden Textstellen durch eine Reihe mehrerer, relativ einfacher Eigenschaften beschrieben werden können. Andererseits sind reguläre Ausdrücke praktisch immer dann überlegen, wenn zum Suchen der entsprechenden Textstellen Wiederholungsoperatoren notwendig werden. Wir werden im Folgenden mehrere Beispiele betrachten, um ein Gefühl dafür zu entwickeln, wann der Einsatz regulärer Ausdrücke sinnvoll ist, in welchen Situation man sie nicht unbedingt benötigt, und wann man gar davon abraten sollte sie zu verwenden.

Dies soll uns auch als Beispiel für eine für Informatiker häufig anzutreffende Problemstellung dienen: Oft gibt es nämlich mehrere Programmiertechniken oder Technologien, um eine Problemstellung anzugehen. Die Entscheidung für eine bestimmte Technologie kann nur erfolgen, wenn man einen tieferen Einblick besitzt und die Technologien auch selbst schon einmal ausprobiert und angewendet hat.

**Beispiel 1**

Angenommen, wir möchten alle Zeilen einer Datei `test.txt` ausgeben, die den String `'Wort'` enthalten. Pythons `in`-Operator ermöglicht eine kompakte und einfach ver-

ständliche Lösung mittels einer Listenkomprehension ohne eine Verwendung regulärer Ausdrücke:

```
>>> [zeile for zeile in open('test.txt') if 'Wort' in zeile ]
```

Eine entsprechende Lösung unter Verwendung eines regulären Ausdrucks ist komplexer und – zumindest für Programmierer, die nicht regelmäßig mit regulären Ausdrücken arbeiten – unverständlicher:

```
>>> dat = open('test.txt').read()
>>> re.findall(r'^.*Wort.*$',dat,re.MULTILINE)
```

**Beispiel 2**

Angenommen, wir möchten alle Zeilen einer Datei `test.txt` ausgeben, die den String `'Wort'` *nicht* enthalten. Wiederum ermöglicht Pythons `in`-Operator eine kompakte und einfach verständliche Lösung ohne eine Verwendung regulärer Ausdrücke:

```
>>> [zeile for zeile in open('test.txt') if 'Wort' not in zeile ]
```

Hier ist eine entsprechende Lösung unter Verwendung eines regulären Ausdrucks deutlich komplexer und vermutlich für die meisten Programmierer schlechter zu verstehen:

```
>>> dat = open('test.txt').read()
>>> re.findall(r'^(?:(?!\bWort\b).)*$',dat,re.MULTILINE)
```

Der reguläre Ausdruck `'^(?:(?!\bWort\b).)*$'` nutzt den negierenden Lookahead-Operator `'(?!...)'` um jede Position einer Zeile darauf zu testen, ob der String `'Wort'` folgt. In jeder Position wird also 4 Zeichen nach vorne geschaut; die jeweilige Position wird genau dann gematcht, wenn die folgenden 4 Positionen nicht dem Wort `'Wort'` entsprechen.

**Beispiel 3**

Angenommen, wir möchten alle in der Datei `test.txt` enthaltenen Wörter extrahieren, die *nicht* das Teilwort `'re'` enthalten und zwischen 3 und 8 Buchstaben lang sind. Mittels einer Listenkomprehension ist dies relativ einfach möglich:

```
>>> [wort for wort in open('test.txt').read().split()
        if 're' not in wort and len(wort)>=3 and len(wort)<=8]
```

Im Gegensatz dazu scheint der reguläre Ausdruck komplexer und für meisten Programmierer wahrscheinlich schwerer (wenn überhaupt) verständlich:

```
>>> re.findall(r'\b(?=\w{3,8})(?:(?!re)\w)+\b',open('test.txt').read())
```

Durch den Lookahead-Operator `r'\w{3,8}'` wird sichergestellt, dass das gematchte Wort mehr als 3 und weniger als 8 Zeichen lang ist. Der reguläre Ausdruck `r'(?:(?!re)\w)'` matcht jeden Buchstaben des Wortes genau dann, wenn dort *nicht* das Teilwort `'re'` beginnt.

Auch wenn diese Lösung schwerer verständlich sein mag, als die mittels Listenkomprehensionen, liefert in diesem Fall die Verwendung eines regulären Ausdrucks eine robustere Lösung: Der reguläre Ausdruck verwendet als Wortgrenze `r'\b'`, also den übergang zwischen einem Wort-Zeichen `r'\w'` und einem nicht-Wort-Zeichen `r'\W'`. Die in der Listenkomprehension verwendete `split()`-Funktion verwendet als Wort-Trenner lediglich Whitespace-Zeichen. Die Listenkomprehension so anzupassen, dass diese ebenfalls Wortgrenzen erkennt, würde einen größeren Aufwand bedeuten. Dahingegen wäre es ein Einfaches, den regulären Ausdruck so anzupassen, dass er exakt die gleichen Ergebnisse liefert, wie die Listenkomprehension. Allgemein kann gesagt werden, dass Lösungen über reguläre Ausdrücke flexibler sind als Lösungen, die keine regulären Ausdrücke verwenden.

Einen vernünftigen Kompromiss beider Lösungen, der die Flexibilität regulärer Ausdrücke und die Verständlichkeit der obigen Listenkomprehension vereinigt, erhält man durch Verwendung der flexibleren Funktion `re.split`. Ein Aufruf von

`re.split(`⟨*re*⟩, ⟨*str*⟩`)`

zerlegt ⟨*str*⟩ an den Stellen, wo ⟨*re*⟩ matcht – dahingegen zerlegt die **`str`**`.split`-Funktion nur an den Whitespace-Stellen.

```
>>> [wort for wort in re.split(r'\W+',open('test.txt').read())
        if 're' not in wort and len(wort)>=3 and len(wort<=8)]
```

**Aufgabe 4.26**

Passen Sie die Lösung so an, dass alle in einem LaTeX-Dokument enthaltenen Wörter gefunden werden, die den String `'re'` enthalten und eine Länge zwischen drei und acht haben. Gehen Sie davon aus, dass Umlaute durch ein dem entsprechenden Vokal vorangestelltes doppeltes Anführungszeichen `"` dargestellt sind, d. h. die Darstellung eines „ü“ erfolgt im LaTeX-Quelltext durch die Zeichen `"u`, die Darstellung eines „Ä“ durch die Zeichen `"A` usw.

**Beispiel 4**

Angenommen, wir möchten alle in der Datei `test.txt` enthaltenen Ziffern extrahieren. Wir können hierzu die Funktion **`str`**`.isdigit()` verwenden, die testet, ob ein Zeichen eine Ziffer ist:

```
>>> [c for c in open('test.txt').read() if c.isdigit()]
```

Ebenso einfach, kann man hierfür einen regulären Ausdruck verwenden:

```
>>> re.findall(r'[0-9]', open('test.txt').read())
```

**Beispiel 5**

Angenommen, wir möchten alle in der Datei `test.txt` enthaltenen Ganzzahlen (sprich: Ziffernfolgen) extrahieren. Möchte man die Verwendung eines regulären Ausdrucks vermeiden, dann wird eine Lösung, vor allem aufgrund der vielen beteiligten Variablen und Schleifen, unangenehm kompliziert. Listing 4.1 zeigt die Implementierung einer solchen Lösung:

```
erg = []
i=0
zahl=''
txt = open('test.txt').read()
while i<len(txt):
  if txt[i].isdigit():
    zahl = txt[i]
    i += 1
    while txt[i].isdigit():
      zahl += txt[i]
      i += 1
    erg.append(zahl)
    zahl = ''
  else:
    i+=1
```

**Listing 4.1.** Suche aller in der Datei `test.txt` enthaltenen Ganzzahlen; nach Ausführung enthält die Liste `erg` alle gefundenen Ganzzahlen. Immer dann, wenn die Bedingung der `if`-Abfrage in Zeile 6 mit `True` ausgewertet wird, befinden wir uns am Anfang einer Ziffernfolge. Die `while`-Schleife in Zeile 9 läuft die Ziffernfolge ab, sammelt alle enthaltenen Ziffern in der String-Variablen `zahl` auf und hängt diese nach Beendigung der `while`-Schleife (in Zeile 12) an die Liste `erg` an.

Eine sehr viel einfachere Lösung erhält man, wenn man einen regulären Ausdruck verwendet:

```
>>> re.findall(r'\b[0-9]+\b',open('test.txt').read())
```

Genaugenommen werden so jedoch möglicherweise mehr Ziffernfolgen gefunden als durch das Programm in Listing 4.1: Das Programm in Listing 4.1 extrahiert beispielsweise aus dem String `'Soll36$'` die Ziffernfolge `'36'`, der reguläre Ausdruck `r'\b[0-9]+\b'` jedoch findet diese Ziffernfolge nicht, da er nur Ziffern matcht, die von Wortgrenzen eingeschlossen sind. Ein regulärer Ausdruck, der *genau* dieselben Ziffernfolgen findet, wie das in Listing 4.1 gezeigte Programm, ist jedoch noch einfacher:

```
>>> re.findall(r'[0-9]+',open('test.txt').read())
```

Der Wiederholungsoperator `'+'` ist greedy; es werden also möglichst viele Ziffern gematcht. Dieses `re.findall`-Kommando findet also genau dieselben Ziffernfolgen, wie das in Listing 4.1 gezeigte Skript.