Editor-Trick des Tages: Zen Mode

Wenn Sie an Ihren Programmen schreiben, brauchen Sie nicht immer alle Funktionalitäten von VSCode. Die Balken an den Seiten und unten können stören, wenn Sie sich nur auf den aktuellen Code konzentrieren wollen.

Dazu können Sie den *Zen Mode* von VSCode aktivieren. Drücken Sie `Ctrl + K` und dann `Z`. Jetzt wird der Editor maximiert und die Teile des Editors, die nicht unmittelbar mit dem Schreiben der aktuellen Datei zu tun haben, werden ausgeblendet.

Um den Zen Mode zu beenden, drücken Sie zweimal `Escape`.

String Formatting in Python

Wie Sie wissen, können die Datentypen, mit denen wir zu tun haben, mit der Methode `str()` explizit in Strings umgewandelt werden. So können wir Strings und Nicht-Strings zusammenfügen und schließlich als einen Gesamtstring weiter verarbeiten:

In [ ]:

```
frequencies = {"und": 1002, "oder": 876, "nicht": 311}
for word in frequencies:
    output = "Das Wort '" + word + "' kommt " + str(frequencies[word]) + "-mal vor."
    print(output)
```

Es gibt einen sehr viel bequemeren Mechanismus, um Nicht-Strings in Strings einzubauen, und zwar die Methode `format()`. `format()` ist eine Stringmethode und nimmt beliebig viele Argumente an. Die Methode wird folgendermaßen verwendet:

In [ ]:

```
frequencies = {"und": 1002, "oder": 876, "nicht": 311}
for word in frequencies:
    output = "Das Wort '{}' kommt {}-mal vor.".format(word, frequencies[word])
    print(output)
```

Beachten Sie, dass in diesem Codebeispiel nur noch ein einziger String und keine Verknüpfung von Strings mit `+` mehr vorkommt.

Der String enthält zwei Platzhalter, die jeweils mit `{}` markiert werden. Die Methode `format()` wird auf den String mit den Platzhaltern angewendet. Die Argumente, die in den Klammern von `format()` übergeben werden, werden nun von links nach rechts in die Platzhalter eingefügt.

Aufgabe

1. Was passiert, wenn die Anzahl der Platzhalter größer ist als die Anzahl der Argumente? Was, wenn es mehr Argumente als Platzhalter gibt? Probieren Sie es aus, indem Sie den Code im Beispiel oben verändern.

Ein weiterer Vorteil der Format-Methode ist, dass praktisch jeder Datentyp als Argument übergeben werden kann. Solange es eine mögliche String-Repräsentation für das übergebene Objekt gibt, wird diese String-Repräsentation an der Stelle des jeweiligen Platzhalters eingefügt.

`format()` ist sogar noch mächtiger und erlaubt uns, zusätzliche Angaben zum Format des Strings zu machen. So können Kommazahlen gerundet werden, einzelne Elemente können links oder rechts ausgerichtet werden etc. Eine ausführliche Erklärung finden Sie unter https://pyformat.info/ (https://pyformat.info/).

Reguläre Ausdrücke

Vor einigen Wochen haben wir die Stringmethode `replace()` kennengelernt, bei der Teilstrings in einem größeren String durch andere Zeichenfolgen ersetzt werden können:

In [ ]:

```
print("Call me Ishmael.".replace(" ", "! "))
```

Wie würden Sie mit der Methode `replace()` eine Aufgabe lösen, bei der ein Text anonymisiert werden soll, indem alle Zahlen durch ein X ersetzt werden?

Es könnte etwa so funktionieren:

In [ ]:

```
secret = 'Sozialversicherungsnr.: 968127490567'
secret = secret.replace('0', 'X')
secret = secret.replace('1', 'X')
secret = secret.replace('2', 'X')
secret = secret.replace('3', 'X')
secret = secret.replace('4', 'X')
secret = secret.replace('5', 'X')
secret = secret.replace('6', 'X')
secret = secret.replace('7', 'X')
secret = secret.replace('8', 'X')
secret = secret.replace('9', 'X')
print(secret)
```

So ein Programmierstil ist äußerst fehleranfällig, schwer lesbar und ineffizient. Wenn wir alle Zahlen durch ein X ersetzen wollen, egal, um welche Zahl es sich jeweils handelt, dann wollen wir nicht jede Zahl einzeln behandeln.

Deshalb beschäftigen wir uns heute mit einer Möglichkeit, Strings nicht nach konkreten Vorgaben, sondern nach beliebig allgemeinen Mustern zu untersuchen: **Reguläre Ausdrücke** (englisch *Regular Expressions*, *regex*).

Mit regulären Ausdrücken können wir alle Zahlen im Text mit nur einer Operation zensieren:

In [ ]:

```
import re     # das Modul importieren, das Methoden für regexes zur Verfügung stellt

secret = 'Sozialversicherungsnr.: 968127490567'

# Hier definieren wir das Muster für Zahlen:
zahlen = re.compile('(0|1|2|3|4|5|6|7|8|9)')

# re.sub ersetzt Teilstrings durch beliebige andere Zeichensequenzen:
print(re.sub(zahlen, 'X', secret))
```

Die Funktion `re.sub(pattern, replacement, string)` sucht im übergebenen `string` nach dem übergebenen `pattern` und ersetzt jede Stelle im String, die dem Muster entspricht, durch das gewünschte `replacement`. Der Rückgabewert ist vom Typ String.

Der reguläre Ausdruck `zahlen` ist als Gruppe definiert, erkennbar an den runden Klammern an Anfang und Ende des Ausdrucks. Die einzelnen Elemente in der Gruppe sind durch das Zeichen `|` separiert, das in diesem Kontext für das logische "oder" steht.

Sobald im String `secret` eins der Elemente aus der Gruppe gefunden wird, wird die entsprechende Stelle im String durch ein X ersetzt.

Wenn wir über das Wiederfinden von Mustern in Strings sprechen, bezeichnen wir die Stelle im String, die dem Muster entspricht, als *Match*. Das hier verwendete Muster `zahlen` *matcht* auf jede Stelle im String `secret`, die der Zahl 0 oder der Zahl 1 oder der Zahl 2 oder... usw. entspricht.

Und es geht sogar noch kompakter:

In [ ]:

```
import re

secret = 'Sozialversicherungsnr.: 968127490567'
zahlen = re.compile('[0-9]')

print(re.sub(zahlen, 'X', secret))
```

Hier wird der reguläre Ausdruck als *Menge* von Symbolen (die Zahlen von 0 bis 9) definiert, erkennbar an den eckigen Klammern. Das Programm verhält sich immer noch so wie vorher, ist aber durch die abgekürzte Schreibweise `[0-9]` noch übersichtlicher geworden.

Einige Arten von Zeichen, wie z.B. die Zahlen von 0 bis 9, bilden eine eigene Klasse von Symbolen. Für sie gibt es vordefinierte Sonderzeichen, mit denen reguläre Ausdrücke sich kompakter schreiben lassen:

Zeichen	Matcht auf...
`\d`	…alle Zahlen von 0 bis 9.
`\D`	…sämtliche Zeichen außer den Zahlen von 0 bis 9.
`\s`	…sämtliche Arten von Whitespace, unter anderem Leerzeichen, Tabs und Zeilenumbrüche.
`\S`	…sämtliche Zeichen außer Whitespace-Symbole.
`\w`	…alle alphanumerischen Zeichen; gleichbedeutend mit der Menge `[A-Za-z0-9_]`.
`\W`	…sämtliche Zeichen außer alphanumerischen Zeichen.
`^`	…den Anfang des Strings. Wird verwendet, wenn der reguläre Ausdruck nur am Anfang der Zeichenkette gematcht werden soll.
`$`	…das Ende des Strings. Wird verwendet, wenn der reguläre Ausdruck nur am Ende der Zeichenkette gematcht werden soll.
`.`	…genau ein beliebiges Zeichen. Achtung, kein Punkt!
`\.`	…den Punkt (`.`)

Mit einem dieser Sonderzeichen wird unser Code von oben sogar noch kürzer:

In [ ]:

```
import re

secret = 'Sozialversicherungsnr.: 968127490567'
print(secret)

zahlen = re.compile('\d')

print(re.sub(zahlen, 'X', secret))
```

Verarbeitung von Matches

Wenn ein regulärer Ausdruck ein komplexes Muster beschreibt, ist es oft hilfreich, die Teil-Matches aus dem analysierten String zu extrahieren und z.B. in Variablen zu speichern. Um dies zu tun, arbeiten wir mit **Gruppen**. Mit der Methode `group()` können wir auf den Teil des Strings, der auf die Gruppe gematcht hat, zugreifen:

In [ ]:

```
import re

drink = 'warm tea'

# Muster erstellen und als description speichern:
description = re.compile('(hot|warm|cold)\s(milk|coffee|water|tea)')

# wenn wir den Inhalt und Typ der Variable description ausgeben,
# sehen wir, dass es sich um ein Objekt des Typs _sre.SRE_Pattern
# handelt
print(description)
print(type(description))

print("-----------------------------------")

# Im String drink nach dem Muster description suchen
m = re.search(description, drink)

# Der Inhalt der Variable m hat den Typ _sre.SRE_Match.
# Der Wert zeigt uns, welcher Slice des Strings gematcht wurde
# ( = dem Muster entspricht).
print(m)
print(type(m))

print("-----------------------------------")

# Gruppe 0: Gesamter Match
print("Gruppe 0: {}".format(m.group(0)))

# Gruppe 1: Erste gematchte Gruppe aus dem Muster
print("Gruppe 1: {}".format(m.group(1)))

# Gruppe 2: Zweite gematchte Gruppe aus dem Muster
print("Gruppe 2: {}".format(m.group(2)))
```

Reguläre Ausdrücke entwerfen und testen

Im Internet gibt es eine Menge Webseiten, auf denen man reguläre Ausdrücke eingeben kann und die für Teststrings anzeigen, ob es einen Match gibt bzw. welche Teile des Strings auf welche Teile der Regex matchen. Wir empfehlen beispielsweise RegexPal (https://www.regexpal.com/).

Diese Seiten können ein erster Anlaufpunkt sein, wenn Sie einen regulären Ausdruck schreiben müssen. Der Vorteil ist, dass Sie so den regulären Ausdruck von Ihrem restlichen Code isolieren und ihn separat bearbeiten können. Wenn Sie fertig sind, übernehmen Sie Ihre Regex in Ihren Code, indem Sie mit `re.compile()` eine neue Variable anlegen, deren Inhalt Ihr definiertes Muster ist.

Gruppen und Mengen

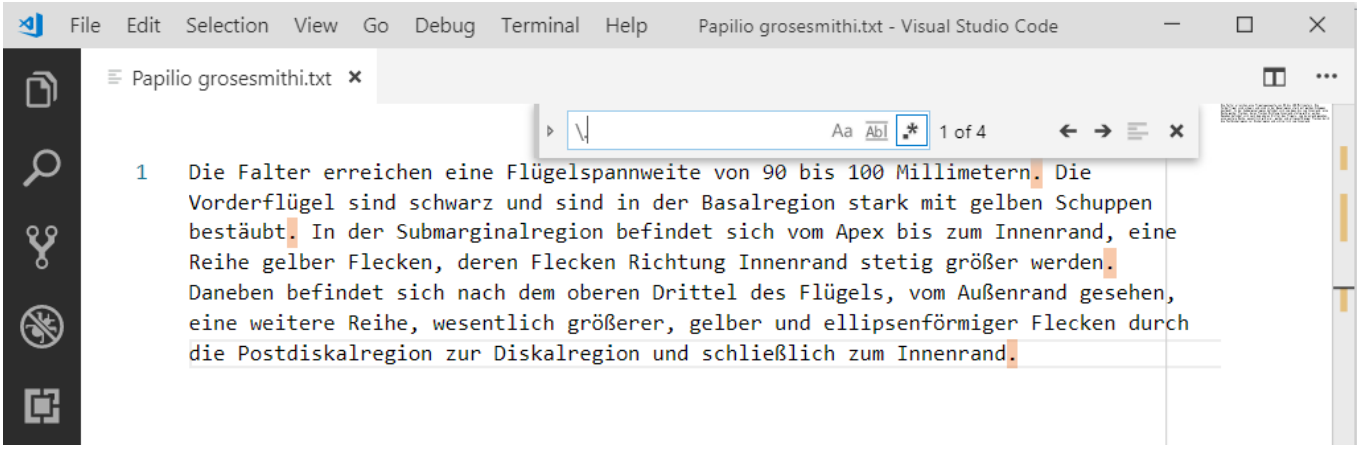
Die Syntax von regulären Ausdrücken belegt runde und eckige Klammern mit einer Spezialfunktion, nämlich dem Erzeugen von Gruppen und Mengen. Die folgende Tabelle zeigt, wie diese Funktionen einzusetzen sind und wie ein regulärer Ausdruck die Klammersymbole selbst als Schriftzeichen matchen kann.

Ausdruck	Funktion
`A`	Findet alle Vorkommen des Zeichens `A` im String.
`(A | B)`	Findet alle Stellen im String, die dem Zeichen `A` oder dem Zeichen `B` entsprechen.
`(...)`	Definiert eine Gruppe. Innerhalb von Gruppen werden Zeichenabfolgen in der vorgegebenen Reihenfolge gematcht. Die runden Klammern sind Teil der Syntax von regulären Ausdrücken und finden sich nicht im String wieder.
`\(` und `\)`	Matcht auf öffnende bzw. schließende runde Klammern im String.
`[...]`	Definiert eine Menge. Die Reihenfolge der Zeichen spielt keine Rolle. Jedes Vorkommen von einem der Zeichen in der Menge führt zu einem Match. Groß- und Kleinschreibung wird unterschieden!
`[A-Z]`	Definiert die Menge aller Großbuchstaben von A bis Z (beide inklusive).
`[e-h]`	Definiert die Menge aller Kleinbuchstaben von e bis h (beide inklusive).
`[^...]`	Definiert eine auszuschließende Menge. Ein Match findet an jeder Stelle des Strings statt, an der *keins* der Zeichen in der Menge steht.
`\[` und `\]`	Matcht auf öffnende bzw. schließende eckige Klammern im String.

Reguläre Ausdrücke und VSCode

Wir können im Editor einen Suchmodus einstellen, der reguläre Ausdrücke beherrscht. Drücken Sie dazu `Ctrl + F`, oder öffnen Sie die Command Palette, um den Befehl `>find` einzugeben.

Ein kleines Eingabefenster öffnet sich. Hier können wir ein Stichwort eingeben, das wir suchen wollen. Um nach regulären Ausdrücken zu suchen, aktivieren wir den RegEx-Modus durch einen Klick auf die Schaltfläche mit der Beschriftung `.*`.



Aufgabe

1. Schreiben Sie einen regulären Ausdruck, der in einem Text alle großgeschriebenen Vokale findet! Tipp: Sie können eine Menge dafür definieren (siehe Tabelle "Gruppen und Mengen"). Im Suchfeld gibt es auch eine Schaltfläche mit der Beschriftung `Aa`. Diese Einstellung schaltet um, ob zwischen Groß- und Kleinschreibung unterschieden wird.

2. Wieviele verschiedene Möglichkeiten finden Sie, um mit einem regulären Ausdruck großgeschriebene Vokale zu ermitteln? Mindestens zwei sollten Ihnen einfallen.

Quantifizierer für Teilausdrücke

Soll ein Teil eines regulären Ausdrucks optional sein oder mehrfach nacheinander vorkommen können, werden die folgenden Zeichen verwendet. Sie beziehen sich immer auf den direkt vor ihnen stehenden Teilausdruck, also auf das vorangegangene Zeichen, die vorangegangene Gruppe oder die vorangegangene Menge.

Ausdruck	Funktion
`+`	Voriges Element kommt **entweder einmal oder mehrmals** direkt nacheinander im String vor.
`*`	Voriges Element kommt **entweder nullmal oder mehrmals** direkt nacheinander im String vor.
`?`	Voriges Element kommt **entweder nullmal oder einmal** im String vor, ist also optional.
`{3}`	Voriges Element kommt **genau 3 mal** nacheinander im String vor.
`{3,5}`	Voriges Element kommt **zwischen 3 und 5 mal** nacheinander im String vor.
`{3,}`	Voriges Element kommt **mindestens 3 mal** nacheinander String vor.
`{,5}`	Voriges Element kommt **höchstens 5 mal** nacheinander im String vor.

Anwendungsbeispiel für reguläre Ausdrücke

Sie können alle oben gelisteten Elemente nach Belieben in einem einzigen regulären Ausdruck kombinieren. Das so definierte Muster wird dann beim Aufruf von `re.sub()` oder `re.search()` von links nach rechts mit dem String abgeglichen.

Falls der String alle Bedingungen erfüllt, ist das Ergebnis von `re.search(muster, string)` ein Objekt vom Typ `Match` (ein spezieller Datentyp für Regexes). Andernfalls gibt die Funktion `None` zurück.

Im folgenden Beispiel sollen lateinische Substantivformen aus dem String `lexicon` gelesen und dann als Dictionary abgespeichert werden.

Mit regulären Ausdrücken kann die Aufgabe elegant gelöst werden:

In [ ]:

```
import re

lexicon = """amicus (Freund):

Nominativ: amicus
Genitiv: amici
Dativ: amico
Akkusativ: amicum
Ablativ: amico"""

# Jede Zeile, die uns interessiert, beginnt mit einem der vier Worte
# "Nominativ", "Dativ", "Akkusativ" oder "Ablativ".
# Nach dem ersten Wort folgt ein Doppelpunkt.
# Nach dem Doppelpunkt steht ein Whitespace (Leerzeichen).
# Alle verbleibenden Zeichen werden mit der Gruppe (.*) "eingefangen".
case_and_form = re.compile('(Nominativ|Genitiv|Dativ|Akkusativ|Ablativ):\s(.*)')

# leeres Dictionary anlegen
forms = {}

# Wir schauen uns jede Zeile nacheinander an
for line in lexicon.split("\n"):

    # Hier prüfen wir, ob das Muster case_and_form in der aktuellen
    # Zeile gefunden wird
    m = re.search(case_and_form, line)

    if m:
        # Falls das Muster auf die Zeile passt, können wir den Inhalt
        # ins Dictionary schreiben.
        # Die erste Gruppe enthält den Kasusnamen.
        # Die zweite Gruppe enthält das dazugehörige Wort.
        case = m.group(1)
        form = m.group(2)

        # Schreibe für den Schlüssel case den Inhalt der Variable form
        # ins Dictionary.
        forms[case] = form

    else:
        # Manche Zeilen passen nicht zum Muster. Diese Zeilen werden
        # nicht weiter verarbeitet.
        print("kein Match gefunden: {}".format(line))

print(forms)
```

Im Ausdruck `case_and_form` werden genau zwei Gruppen definiert: Die eine enthält alle erwarteten Kasusnamen, die andere enthält alle Zeichen, die nach dem Doppelpunkt und dem Whitespace noch in der Zeile folgen. Die erste Gruppe ist also sehr viel einschränkender als die zweite. Doppelpunkt und Whitespace stehen außerhalb der Gruppen, weil sie nicht weiter verarbeitet werden sollen.

Die Methode `re.split()`

Sie können reguläre Ausdrücke auch verwenden, um Strings zu zerlegen. Sie kennen bereits die Funktion `string.split(sep)`, bei der der gegebene String an allen Vorkommen des Substrings `sep` aufgeteilt wird. Das Ergebnis von `split()` ist eine Liste (vgl. Vorlesungsskript vom 15.10.2019).

Fast analog dazu funktioniert die Funktion `re.split(pattern, string)`. Das übergebene `pattern` beschreibt in der Syntax regulärer Ausdrücke den Separator, der jetzt nicht mehr exakt bekannt sein muss, sondern mithilfe von Bedingungen beschrieben werden kann.

Im folgenden Code wird eine logische Formel überall dort zerteilt, wo logische Operatoren vorkommen. Das Ziel ist es, eine Liste aller Teilformeln zu erhalten. (In diesem vereinfachten Beispiel gehen wir davon aus, dass alle großgeschriebenen Elemente im String logische Operatoren sind.)

In [ ]:

```
import re

# die Query enthält die Operatoren AND, OR, AND NOT
query = 'casus=nom AND (genus=f OR genus=m) AND NOT numerus=sg'

# Das Muster soll auf alle Vorkommen von Großbuchstaben,
# öffnenden oder schließenden Klammern,
# und dazugehörige Whitespaces matchen.
# Das + am Schluss der Menge besagt, dass mehr als ein Zeichen
# aus der Menge gematcht werden kann.
operator = re.compile('[A-Z\(\)\s]+')

print(re.split(operator, query))
```

Bei `re.split()` gibt es eine Besonderheit, die zu Verwirrung führen kann: Wenn im regulären Ausdruck Gruppen definiert sind, enthält das Ergebnis der Operation nicht nur die Teile des Strings, die zwischen den Vorkommen von `pattern` gefunden wurden, sondern auch diejenigen Teile, die dem `pattern` entsprechen (also die Separatoren). Darin unterscheidet sich `re.split()` von der Stringmethode `split()`, die ja die Separatoren abschneidet.

In [ ]:

```
import re

query = 'casus=nom AND (genus=f OR genus=m) AND NOT numerus=sg'

operator2 = re.compile('([A-Z\(\)\s]+)') # Hier ist das Muster von runden Klammern umge
ben
                                         # (Definition einer Gruppe)
print(re.split(operator2, query))
```

Aufgabe: Wählen zwischen regulären Ausdrücken und Stringoperationen

Reguläre Ausdrücke sind im Vergleich zu den Ihnen bereits bekannten Stringoperationen sehr mächtig. Ob Sie mit Stringoperationen oder mit Regexes arbeiten, hängt hauptsächlich von der Art der Aufgabe ab, die Sie lösen möchten. Was würden Sie beispielsweise in den folgenden Situationen tun?

1. Einlesen einer Visitenkarte, deren Inhalte in einem Dictionary gespeichert werden sollen:

   Eingabe:

   ```
   Vorname: Petunia
   Nachname: Dursley
   Straße: Privet Drive
   Hausnummer: 4
   ```

   Ausgabe als Dictionary:

   ```
   {'Vorname': 'Petunia', 'Nachname': 'Dursley', 'Hausnummer': '4', 'Straße':
    'Privet Drive'}
   ```

2. Ermitteln, ob eine IP-Adresse wohlgeformt ist (vier Zahlenblöcke, getrennt durch je einen Punkt, jeder Block entspricht einer Zahl zwischen 0 und 255)

   ```
   gute IP: 192.168.178.1
   gute IP: 10.10.10.10
   schlechte IP: 10.10.10
   schlechte IP: 300.188.10.6
   ```

3. Ersetzen aller E-Mail-Adressen in einem Text durch Platzhalterstrings

   ```
   Sie können mich jederzeit unter der Adresse [REDACTED@REDACTED.COM] erreich
   en.
   ```

4. Alle HTML-Tags aus dem Code einer Webseite entfernen (z.B. `<h1>`)

   Eingabe:

   ```
   <h1>Überschrift</h1>
   ```

   Ausgabe:

   ```
   Überschrift
   ```

5. Aus einem kompletten Dateipfad nur den Namen der Datei extrahieren (alle Zeichen nach dem letzten `/` , z.B. `notizen.txt` aus dem Pfad `D:/Dateien/Python/notizen.txt`)
6. Ermitteln, ob ein String ein Palindrom ist

Verwenden Sie auch die Python-Dokumentation und ggf. StackOverflow, um Antworten für die einzelnen Beispiele zu finden.

Sie müssen die Aufgaben nicht fertig implementieren! Entscheiden Sie nur, ob reguläre Ausdrücke jeweils geeignet sind oder nicht.

Regex Crossword

Wenn Sie noch nicht mit regulären Ausdrücken gearbeitet haben, wirkt die Syntax auf Sie vielleicht etwas abschreckend, weil teilweise viele Zeichen verwendet werden, um auf genau ein Zeichen im String zu matchen. Das ist nicht immer leicht lesbar.

Um die einzelnen Bestandteile der Syntax zu trainieren, können Sie einige Rätsel auf https://regexcrossword.com/ (https://regexcrossword.com/) lösen. Einige der Rätsel werden Ihnen in den Übungsaufgaben für diese Woche noch begegnen.

Wenn Sie mit konkreten regulären Ausdrücken arbeiten, können Sie jederzeit in der Python-Dokumentation nachlesen, wie die Syntax für eine bestimmte regex aussieht.

Zusammenfassung

Sie haben heute gelernt,

- wie Strings mit Platzhaltern definiert werden können, bei denen mit `format()` beliebige Werte eingefügt werden können
- wie Sie mit regulären Ausdrücken Strings anhand bestimmter Muster beschreiben können
- wie Sie Gruppen in regulären Ausdrücken definieren können, deren Matches Sie dann weiterverarbeiten können
- wie Sie mit `re.split()` Strings zerteilen können, indem Sie komplexe Muster als Separator-Substrings definieren
- in welchen Situationen reguläre Ausdrücke besser/schlechter geeignet sind, um bestimmte Aufgaben zu lösen