Empfohlene Unterrichtsinhalte als Vorbereitung für dieses Thema: 02-01 (Strings), 05-03 (Module), 09-01 (Regular Expressions schreiben), 09-02 (Regular Expressions in Python verarbeiten)

Einführung in die computerlinguistische Programmierung mit Python

09-03: Regular Expressions und Gruppen 

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 das zu erreichen, arbeiten wir mit **Gruppen**, die bei regulären Ausdrücken durch runde Klammern `(...)` markiert werden.

Die Funktion `group()` kann auf RegEx-Matches angewendet werden und ermöglicht uns, auf die verschiedenen Gruppen des Matches **zuzugreifen**. Im folgenden Beispiel besteht der reguläre Ausdruck aus zwei Gruppen und wird auf den Beispielstring gematcht. Die Gruppe mit dem Index 0 entspricht dem **gesamten Match**. Danach folgen in aufsteigender Reihenfolge die Teil-Matches für jede Gruppe, gezählt von links vom Start des Strings. Wie viele `groups` wir verwenden können, hängt davon ab, wie viele Gruppen im regulären Ausdruck enthalten sind.

In [ ]:
```python
import re

# Unser Test-String
drink = 'warm tea'

# Muster erstellen und in einer Variable speichern:
# Dieser Ausdruck enthält eine Gruppe mit den alternativen
# Teilstrings hot, warm oder cold;
# und eine zweite Gruppe mit den alternativen Teilstrings
# milk, coffee, water oder tea.
description = re.compile('(hot|warm|cold)\s(milk|coffee|water|tea)')

# Wir kennen re.search schon: Hier prüfen wir,
# ob der Test-String zum Muster passt.
# Das Ergebnis wird als Variable m gespeichert.
m = re.search(description, drink)

# Der Inhalt der Variable m hat den Typ re.Match.
# Der Inhalt von m zeigt uns, welcher Slice des Strings gematcht wurde
# ( = von welchem Startindex bis zu welchem Endindex
# der String dem Muster entspricht).
print(type(m))
print(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)))
```

```
# Und was ist hiermit?
#print("Gruppe 3: {}".format(m.group(3)))
```

Falls Gruppen ineinander verschachtelt sind, zählt für die Nummerierung die Reihenfolge der öffnenden Klammern. Hier das gleiche Beispiel von oben noch einmal, nur mit einer zusätzlichen Gruppe im regulären Ausdruck:

In [ ]:
```
import re

# Unser Test-String
drink = 'warm tea'

description = re.compile('((hot|warm|cold)\s)(milk|coffee|water|tea)')

m = re.search(description, drink)

# 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)))

# Gruppe 3: Dritte gematchte Gruppe aus dem Muster
print("Gruppe 3: {}".format(m.group(3)))
```

Die Gruppen sind am nützlichsten, wenn wir verschiedene Informationen aus einem String extrahieren und sie in einer **strukturierten Form** abspeichern wollen. Im folgenden Beispiel erhalten wir einen String mit einer Deklinationstabelle für ein lateinisches Substantiv. Ziel des Codes ist es, ein **Dictionary** zu erstellen, in dem die Informationen aus dem String abgespeichert werden und dann jederzeit wieder abgerufen werden können.

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 fünf Worte
# "Nominativ", "Genitiv", 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)

        print("Aktueller Kasus: " + case)
        print("Aktuelle Form: " + form)

        # 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)
```

Gruppen und Teilmatches im Brief von Heinrich Heine

Im letzten Thema, 09-02, haben wir einen Brief von Heinrich Heine mit `re.split()` in seine einzelnen Sätze zerlegt, aber dabei gingen die **Satzzeichen** verloren.

Hier ein Vorschlag, wie es stattdessen gehen könnte. Diesmal definieren wir den regulären Ausdruck so, dass jeder Satz komplett von einer **eigenen Gruppe** abgedeckt wird. `re.split()` verhält sich dann anders, weil wir explizit definiert haben, dass das jeweilige Satzzeichen zum Satz dazugehört und deshalb beim Zerteilen nicht verloren gehen soll. `re.split()` berücksichtigt alle Elemente, die Teil einer Gruppe sind, und verhindert, dass Elemente innerhalb von Gruppen verloren gehen.

In [ ]:
```
# VARIANTE 1: re.split() verwenden

import re

# Der Text enthält mehrere Sätze, die durch Satzendzeichen
# voneinander getrennt sind
heine_brief = """Sehr liebenswürdige und charmante Person! Ich bedauere sehr, daß ic

# alter regulärer Ausdruck:
# satzende = re.compile('[!\.?]\s-?\s*')

# neuer regulärer Ausdruck:
satz = re.compile("([\w,\'\s]+[!\.?]\s-?\s?)")

# Jede Gruppe beginnt mit einer beliebigen Zahl von
# Buchstaben, Leerzeichen oder Kommas (mehr als 0).
# Danach folgt eins der Satzendzeichen.
# Optional folgt ein Gedankenstrich.

brief_saetze = re.split(satz, heine_brief)
for satz in brief_saetze:
    print(satz)
    print("+++")
```

`re.findall()` im Brief von Heinrich Heine

Schließlich gibt es noch die Möglichkeit, mit `re.findall()` **alle Teilstrings, die zum Muster passen**, zu finden. Da unser regulärer Ausdruck jetzt immer genau einen Satz abdeckt, können wir das leicht umsetzen. Die RegEx bleibt so, wie wir sie eben schon definiert haben; nur die Verarbeitung des Strings läuft jetzt etwas anders.

In [ ]:
```
# VARIANTE 2: re.findall() verwenden

import re

# Der Text enthält mehrere Sätze, die durch Satzendzeichen
# voneinander getrennt sind
heine_brief = """Sehr liebenswürdige und charmante Person! Ich bedauere sehr, daß ic

# alter regulärer Ausdruck:
# satzende = re.compile('[!\.?]\s-?\s*')

# neuer regulärer Ausdruck:
satz = re.compile("([\w,\'\s]+[!\.?]\s?-?\s?)")

# Jede Gruppe beginnt mit einer beliebigen Zahl von
# Buchstaben, Leerzeichen oder Kommas (mehr als 0).
# Danach folgt eins der Satzendzeichen.
# Optional folgt ein Gedankenstrich.

brief_saetze = re.findall(satz, heine_brief)

for satz in brief_saetze:
    print(satz)
    print("+++")
```

Hier gilt es zu beachten, dass `re.findall()` uns **nur Strings** zurückgibt, keine zusätzlichen Informationen wie z.B. bei `re.search()`. Das ist aber kein Problem, solange uns sowieso nur der Stringinhalt der Teilmatches interessiert.

Zusammenfassung

- Gruppen in regulären Ausdrücken können nach einem erfolgreichen Match einzeln verarbeitet werden.
- Die Gruppe mit dem Index 0 enthält immer den gesamten Match. Jede weitere Gruppe enthält den Inhalt des jeweils nächsten Teilmatches.
- Beim Verarbeiten von Gruppen können wir die Informationen aus dem String neu strukturieren und z.B. in ein Dictionary schreiben.
- Mit Gruppen können wir erreichen, dass bei `re.split()` der Teilstring, der dem Muster entspricht, nicht verlorengeht.
- Mit `re.findall()` erhalten wir alle Teilstrings, die zum Muster passen, und können sie weiter verarbeiten, z.B. in einer Schleife.

Weitere Themen dieser Woche:

- 09-01: Regular Expressions schreiben
- 09-02: Regular Expressions in Python verarbeiten