Vertiefung zu Kapitel 3

Reguläre Ausdrücke (RE) und ihre Verwendung in Django

Einschub: Reguläre Ausdrücke

- **Reguläre Ausdrücke** (Regular Expressions, RE)
  - Sind **Muster** (engl. Pattern), zu denen Strings passen (können)
    - Ähnlich zu einer Suchfunktion in einer Textverarbeitung „suchen“ wir nach einem **Match** des Regulären Ausdrucks auf dem String
    - Beispiel: RE 'ge' matcht 'Wege' und 'gehen', da 'ge' in beiden vorkommt
    - Beispiel: RE 'ge' matcht nicht 'Straßen', da 'ge' nicht darin vorkommt
  - Diese Muster können Steuerzeichen enthalten
    - So bedeutet '^', dass es nur am Anfang des Strings passen kann
    - Beispiel: RE '^ge' matcht 'gehen', da 'ge' am Anfang steht steht
    - Beispiel: RE '^ge' matcht nicht 'Wege' und 'Straßen', da 'ge' nicht am Anfang steht
  - In Python schreiben wir Reguläre Ausdrücke in String-Literalen meist mit Präfix 'r' (z.B. r'^ge')
    - Dadurch werden Sonderzeichen („\“) im String nicht von Python interpretiert

Einschub: Reguläre Ausdrücke

- **Reguläre Ausdrücke**
  - Es gibt viele RE-Steuerzeichen. Hier eine Auswahl.
    - Siehe https://docs.python.org/3/library/re.html
  - *Grundregel: Jedes nicht-Steuerzeichen* matcht sich selbst (z.B. „a“)
  - '**.**' Punkt = Beliebiges Zeichen
  - '**^**', '**$**' Anfang bzw. Ende des Strings
  - '**?**', '*****', '**+**' Das vorangehende Muster darf 0...1 mal ('?'), >=0 mal ('*') bzw. >=1 mal ('+') auftreten
    - Beispiel: r'ab*c' bedeutet „einmal a, beliebig oft b, einmal c“
    - Der String 'xabbcy' wird also gematcht von r'ab*c' und r'ab+c', nicht aber von r'ab?c'
  - '**{**n,m**}**', '**{**n**}**' Das vorangehende Zeichen oder Muster muss n bis m mal bzw. genau n mal auftreten
    - Beispiel: Die RE r'ab{3,3}c', r'ab{3}c' und r'abbbc' sind äquivalent

Einschub: Reguläre Ausdrücke

- **Reguläre Ausdrücke**
  - '`(…)`' Der RE in der Klammer gehört zusammen, das Ergebnis des enthalten RE wird namenlos im Ergebnis des Matchvorgangs gespeichert
    - Beispiel RE r'^(ab)+$' matcht 'ab', 'abab', 'abababab' usw.
  - '`[…]`' Eines der Zeichen in der Klammer
    - Beispiel: RE r'^[ab]+$' matcht 'a', 'b', 'aa', 'ab', 'ba', 'bb' usw.
    - In der eckigen Klammer haben „`.`“, „$“, „?“ etc. keine besondere Bedeutung mehr
  - '`[…-…]`' Eines der Zeichen im Bereich von … bis
    - z.B. '`[1-4]`' entspricht '`[1234]`'
  - '`[^…]`' Keines der Zeichen aus …
    - z.B. '`[^0-9.]`' matcht ein beliebiges Zeichen, das keine Ziffer und kein „`.`“ ist
  - '`\d`' Dezimalziffern (entspricht '`[0-9]`')
  - '`\w`' Buchstaben (entspricht '`[a-zA-Z0-9_]`')
  - '`(?P<name>...)`' Das Ergebnis des RE '`...`' wird auch unter der Bezeichnung name im Ergebnis des Matchvorgangs gespeichert

Einschub: Reguläre Ausdrücke

- **Reguläre Ausdrücke verwenden**
  - Unix-Kommandozeile: z.B. mit **grep**
    - **grep**: ein Tool, das aus Dateien alle Zeilen ausgibt, die zum Pattern passen
    - z.B. liste alle Zeilen aus x.txt auf, die mit einer Ziffer anfangen:

```
grep '^[0-9]' x.txt
```

    - z.B. liste alle Zeilen aus x.txt auf, die mindestens eine Ziffer enthalten:

```
grep '[0-9]' x.txt
```

    - z.B. liste alle Zeilen aus x.txt auf, die nur aus Ziffer bestehen:

```
grep '^[0-9][0-9]*$' x.txt
```

      Fragen:
      - Was ist der Unterschied zwischen „`[0-9][0-9]*`“ und „`[0-9]+`“?
      - Warum dann „`[0-9][0-9]*`“ statt „`[0-9]+`“?
        - Es gibt verschieden Sprachumfänge für REs
        - grep kennt u.a. „`+`“ nicht (es ist in grep's RE ein normales Zeichen)
        - Wie sieht dann eine Zeile aus, die in grep zu „`^[0-9]+$`“ passt?

Einschub: Reguläre Ausdrücke

- **Reguläre Ausdrücke verwenden**
  - Python: Bibliothek **re** (siehe https://docs.python.org/3/library/re.html)
    - Die Funktion re.**match**(pattern, string) liefert ein **Match-Objekt** oder None
    - Die Match-Object-Methode **groups**() liefert die Matches als Tupel:

```
import re
for s in ('0x123', '0x1f', '0x1g', '01d'):
    m = re.match(r'^0x([0-9a-f]+)$', s)
    if m is not None:
        print('%s enthält Hex-Ziffern %s' % (s, m.groups()))
```

```
0x123 enthält Hex-Ziffern ('123',)
0x1f enthält Hex-Ziffern ('1f',)
```

    - Die Match-Object-Methode **groupdict**() liefert benannte Matches als Dictionary:

```
m = re.match(r'(?P<name>\w+)=(?P<wert>\d+)', 'counter=17')
print(m.groups())
print(m.groupdict())
```

```
('counter', '17')
{'name': 'counter', 'wert': '17'}
```

Django: Reguläre Ausdrücke

- Anwendung von REs: **Queryset-Filter**
  - `Professor.objects.filter(name__regex=r'^W.*[hr]$')`
    - Liefert alle Professoren, deren Name mir „W“ beginnt und mit „h“ oder „r“ endet.
- Anwendung von REs: **Modelfield-Validatoren**
  - `validators = [ RegexValidator(r'^#[0-9a-f]{6}$') ]`
    - `models.CharField` mit diesem validators-Parameter erlaubt nur 6-stellige CSS-Hex-Farbangaben z.B. `#ffaa37`
- Anwendung von REs: **Custom Path Converter**

```
class FourDigitYearConverter:
    regex = '[0-9]{4}'
    def to_python(self, value):
        return int(value)
    def to_url(self, value):
        return '%04d' % value
```

```
register_converter(
    FourDigitYearConverter, 'yyyy'
)
urlpatterns = [
    path('articles/<yyyy:year>/', ...),
]
```

  - https://docs.djangoproject.com/en/4.2/topics/http/urls/#registering-custom-path-converters

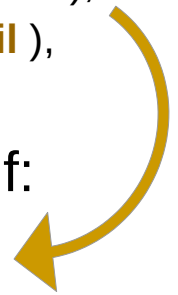
Django: Reguläre Ausdrücke

- Anwendung von REs: `re_path` statt `path` in **urls.py (1)**
  - https://docs.djangoproject.com/en/4.2/topics/http/urls/#using-regular-expressions

Positions-URL-Argumente

- from django.conf.urls imprt url, include
  import news.views

```
urlpatterns = [
    re_path(r'^articles/(\d{4})/$',                 news.views.year_archive ),
    re_path(r'^articles/(\d{4})/(\d{2})/$',         news.views.month_archive ),
    re_path(r'^articles/(\d{4})/(\d{2})/(\d+)/$',   news.views.article_detail ),
]
```



- Beim Zugriff der URL '/articles/2015/12/ resultiert folgender Aufruf:
  - `news.views.month_archive(request, '2015', '12')`
  - Die Reihenfolge der Parameter entspricht der im RE

Verständnisfrage: *Was passiert, wenn man oben das „$“ weglässt?*
Kann man das Problem vermeiden?

Django: Reguläre Ausdrücke

- Anwendung von REs: `re_path` statt `path` in **urls.py** **(2)**

  Benannte-URL-Argumente

  - Man kann die Argumente auch **benennen**
    - siehe RE-Pattern: `'(?P<name>...)'`
  - z.B.
    - `re_path(r'^articles/(?P<year>\d{4})/(?P<month>\d{2})/$', ...)`
    - Der RE matcht genau wie `r'^articles/\d{4}/\d{2}/$'`
    - Hier werden die Ergebnisse den benannten Parametern year und month zugewiesen
    - Beim Aufruf der URL '/articles/2015/12/' erfolgt folgender Funktionsaufruf:
      - `news.views.month_archive(request, year='2015', month='12')`
    - Analog zu `path('articles/<int:year>/<int:month>/$', ...)`
      - Aber bei re_path mehr Kontrolle über Format (hier: Anzahl der Ziffern).