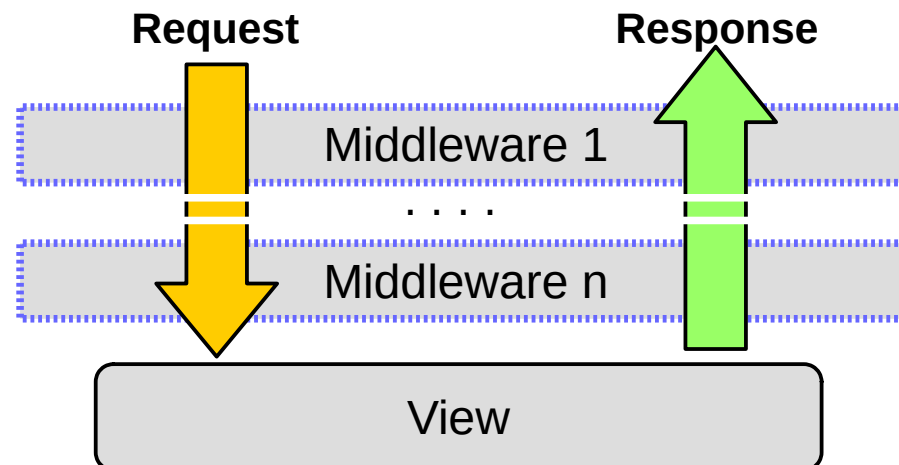


Django: Middlewares

- Django beinhaltet einen **Middleware-Mechanismus**
 - Im System ist eine Menge von **aktivierten** Middlewares eingestellt
 - Parameter **MIDDLEWARE_CLASSES** in settings.py
 - Middlewares bearbeiten eingehende **Requests** und ausgehende **Responses**
 - Man kann so z.B.
 - beim Response die Kommentare aus dem ausgehenden HTML-Text entfernen
 - Im Request zusätzliche Template-Variablen hinzufügen
 - Responses Cachen und bei erneuten Anfragen aus dem Cache beantworten
 - usw.



Django: Middlewares

- **Standardmäßig aktive Middlewares sind u.a.**
 - **SessionMiddleware**
 - Verwaltet **Sessions** und stellt **request.session** bereit
 - **CsrfViewMiddleware**
 - Verhindert **Cross-Site-Request-Forgeries (CSRFs)**
 - **AuthenticationMiddleware, SessionAuthenticationMiddleware**
 - Verwaltet **Sessions** und stellt **request.user** bereit
 - **MessageMiddleware**
 - Verwaltet einmalig angezeigte Nachrichten an den Benutzer, die in der Webseite angezeigt werden.
 - z.B. „Der Datensatz wurde gespeichert.“

Django: Middlewares

- Manche Middlewares brauchen Zugriff auf den **Request** beim Rendern der Antwort

- Deshalb haben wir **request** an **render()** übergeben

- **render(request, template_name, parameter_dict, ...)**
erhält als ersten Parameter immer das **Request-Objekt**

```
def edit_vl(request, id):  
    # ...  
    return render(request, 'vl_edit.html', dict(vl=vl, form=form))
```

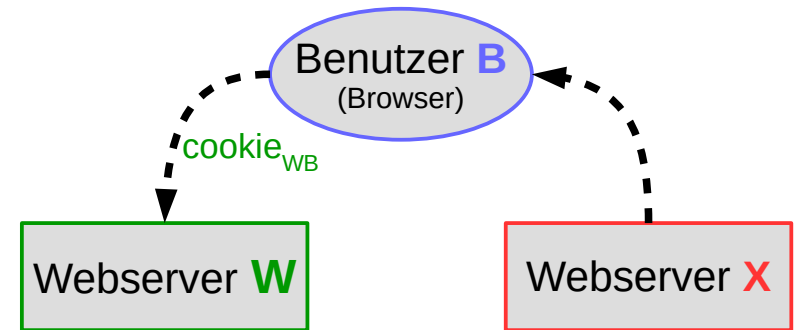
- Mit Hilfe des Request-Kontext hat man z.B. in den Templates dann ...
 - Zugriff auf `{% csrf_token %}` bei der **CsrfViewMiddleware**
 - Zugriff auf `{{ user }}` bei der **AuthenticationMiddleware**
 - Zugriff auf `{{ request }}`

Django: Middlewares

- Was sind **CSRFs**?

- Betrachten wir folgendes Szenario:

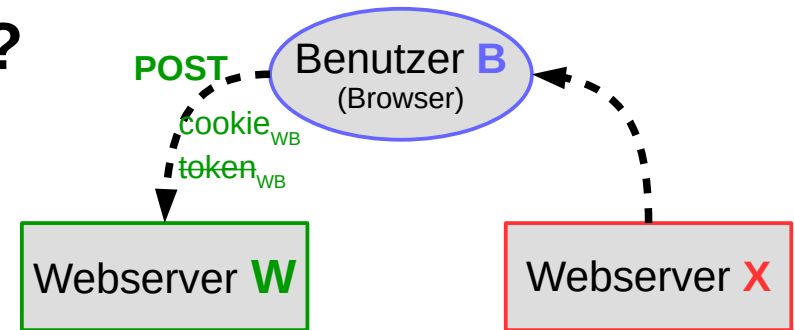
- Benutzer **B** ist auf einer **Webseite W** eingeloggt (gültiges Session-Cookie)
 - Bei jedem Zugriff auf diese Webseite **W** wird das Session-Cookie übertragen
 - Damit erkennt der Webserver **W**, dass die Zugriffe vom Benutzer **B** stammen
- Anschließend / Parallel ruft Benutzer **B** die **Webseite X** auf
 - Auf Webseite **X** befindet sich ein Link auf eine URL in **W**
 - z.B. ein IMG-Tag mit entsprechendem SRC-Parameter oder über Javascript
- Dadurch entsteht vom Browser von **B** ein Request auf Webserver **W**
 - GET-Request oder ein POST-Request
 - Dieser enthält das gültige Session-Cookie von **B**
- Der Webserver **W** kann nicht erkennen, dass der (von **B** authentifizierte) Request nicht von **B** explizit gewünscht war, sondern von **X** provoziert wurde
 - **X** könnte so im Namen von **B** auf Webserver **W** agieren
 - z.B. Daten manipulieren, Bestellungen auslösen, indirekte Angriffe vorbereiten
 - Dabei sind (normalerweise) **GET-Requests** unkritisch
 - Warum?



Django: Middlewares

- **Was kann man gegen CSRFs tun?**

- Kritisch sind v.a. POST-Requests
 - also Formulare mit method POST
- Es ist kein Geheimhaltungsproblem
 - Niemand außer **B** und **W** kennen das Session-Cookie cookie_{WB}
- Trotzdem ist ein **fremder POST-Request** mit cookie_{WB} versehen
 - Weil **X** im Browser von **B** ein POST-Request an **W** auslösen kann.
- Man möchte also erkennen können, **woher** der POST-Request ursprünglich stammt, nicht wer ihn **verschickt** hat (*in beiden Fällen **B***)
 - **Idee:** **W** integriert in sein Formular an **B** eine **geheime Information** (token_{WB})
 - Diese ist spezifisch für die Paarung (**W**, **B**) und nur in dem Formular enthalten
 - Diese Information wird später Teil des POST-Requests
 - Wenn **B** das „gute“ **W**-Formular an **W** POSTet, ist das Geheimnis enthalten
 - **X** kennt diese geheime Information nicht
 - Also kann **X** keinen POST-Request mit token_{WB} erzeugen



Django: Middlewares

- **Wie funktioniert die CSRF-Protection von Django?**

- In jedes Formular wird das Geheimnis `tokenWB` eingebettet
 - Dazu hatten wir oben im Formular das Tag „`{% csrf_token %}`“ eingefügt
 - Dieses erzeugt ein hidden-Input-Field mit `tokenWB`

```
<div style='display:none'>
  <input type='hidden'
        name='csrfmiddlewaretoken'
        value='q392uasopdhjqp92asdkqp23rfasd' />
</div>
```

- Bei einem eingehenden Request ...
 - wird von der **CsrfViewMiddleware** geprüft, ob der entsprechende Parameter mit dem korrekten Wert enthalten ist
 - Ist er nicht enthalten wird eine 403-Response zurück geschickt.
 - Ist er enthalten wird der entfernt und der Request wird normal weiter verarbeitet

Django: Transaktionen

- **Wir kennen aus SQL bereits Transaktionen**
 - Transaktionen definieren einen **semantisch atomaren Zustandsübergang** der Datenbank
 - von einem **konsistenten Zustand** zu einem anderen
 - inkonsistente Zustände sollen für andere Transaktionen nicht sichtbar werden
 - Die Modell-Definition in Django definiert Konsistenzbedingungen
 - Attributtypen, unique, unique_together, ...
- **Wir wollen in Django auch DB-Transaktionen steuern können**
 - Beginn / Ende einer Transaktion, Abbruch einer Transaktion, ...
 - Standard-Transaktions-Verhalten:
 - **Auto-Commit** jeder Django-DB-Operation
 - z.B. bei jedem Aufruf von `model.save()` oder `model.delete()`

Django: Transaktionen

- **Atomarität von Views**

- Views verarbeiten einen Request und erzeugen einen Response
- Aus **Nutzersicht** haben diese Transaktionseigenschaften
 - Löse ich eine Funktion „Warenkorb bestellen“ aus, so soll z.B.
 - *geprüft werden, ob die Waren im Warenkorb gerade verfügbar sind,*
 - *die Waren als **gekauft** markiert aus dem verfügbaren Bestand entfernt werden,*
 - *die **Zahlung** veranlasst werden (Kundenkonto belasten, Gutscheine, etc.),*
 - *die Waren zum **Versand** vorgesehen werden*
 - Alle Operationen sollen **atomar**, also alle oder keine davon erfolgen
 - Auch für den Benutzer ist „**ganz oder gar nicht**“ auf Request-Ebene leicht zu verstehen.
- Aus **Systemsicht** sollen diese Operationen **ACID** erfolgen
 - Das Autocommit-Modell der Einzel-Operationen ist oft nicht sinnvoll.
- Wir möchten also ein **Auto-Commit** eines **kompletten View-Aufrufs**
 - also **eines Requests**

Django: Transaktionen

- **Views in einer DB-Transaktion abwickeln**
 - Um dies zum Default zu machen dient der settings-Parameter **ATOMIC_REQUESTS** ¶
 - Default ist False → in settings.py auf True setzen
 - Genaueres: <https://docs.djangoproject.com/en/4.2/topics/db/transactions/#tying...>
 - Ablauf:
 - **Request-Behandlung** (also **View-Aufruf**) ≡ **Transaktion**
 - Endet der View-Aufruf normal, erfolgt automatisch ein **Commit**
 - Endet sie mit einer unbehandelten Exception, so wird erfolgt **Rollback**
 - Vorteil:
 - Man muss keine unvollständigen Zwischenzustände im Fehlerfall behandeln
 - Nachteil:
 - Transaktionen dauern so lange wie die gesamte Request-Behandlung
 - Vorsicht beim **Logging** in die Datenbank (wird evtl. mit zurück gesetzt)

Django: Transaktionen

- Man kann die Transaktionsbehandlung auch individuell steuern

- z.B. durch einen **Decorator** der View-Funktion

- **@atomic**

- Nur bestimmte Views atomic ausführen

```
from django.db import transaction

@transaction.atomic
def viewfunc(request):
    # ....
```

- z.B. nur einen Code-Abschnitt atomar ausführen

```
from django.db import transaction

with transaction.atomic():
    a.save()
    b.save()
```

- Die Transaktionssteuerung ist sehr präzise möglich (*Savepoints, ...*)

- Siehe <https://docs.djangoproject.com/en/4.2/topics/db/transactions/>

Django: Benutzerverwaltung

- Die **Benutzerverwaltung** in Django
 - wird u.a. im Admin-Interface benutzt
 - Hier kann man auch **Benutzer**, **Benutzergruppen** und **Rechte** (für Benutzern oder für Gruppen) zuordnen
 - Rechte sind u.a. für jede Modell-Klassen ...
 - das Recht ein Objekt **anzulegen**
 - das Recht ein Objekt zu **ändern**
 - das Recht ein Objekt zu **löschen**
 - Darüber hinaus hat jeder Benutzer die Bool-Attribute
 - **Active** (wenn False kann der Benutzer nicht authentifiziert werden)
 - **Staff-Status** (darf man sich im Admin-Interface anmelden)
 - **Superuser-Status** (hat man alle Rechte implizit)
 - Sie basiert auf der **AuthenticationMiddleware**
 - Ist diese aktiv (default), so hat jeder Request die Komponente **request.user**, die auf ein User-Objekt verweist
 - ggf. auf den **AnonymousUser**, wenn niemand authentifiziert ist

Django: Benutzerverwaltung

- **Methoden der User-Objekte**

- Um einen echt angemeldeten User vom `AnonymousUser`, zu unterscheiden, dient das Attribut `is_authenticated`

- Möchte man die Funktion einer View (z.B. Änderungs-Formular) nur angemeldeten Nutzern bereitstellen, so kann man das in der View auch folgendermaßen prüfen:

```
def viewfunc(request):  
    if request.user.is_authenticated:  
        # nur authentifizierte User
```

- Eleganter ist dieser **Dekorator**:

```
@login_required  
def viewfunc(request):  
    # nur authentifizierte User
```

- Ist der Benutzer nicht angemeldet wird man auf eine Login-Seite umgeleitet

- In `Templates` gibt es die Variable „`user`“

```
{% if user.is_authenticated %}  
    {# nur authentifizierte User #}  
{% endif %}
```

- Weitere Einzelheiten dazu: <https://docs.djangoproject.com/en/4.2/topics/auth/>

Django: Benutzerverwaltung

- **Tipp: Eine Login-Seite für Prototyp-Applikationen**

- Der `login_required`-Dekorator benötigt ggf. eine Login-Seite ...
 - auf die er den (noch nicht eingeloggten) Nutzer umleiten kann
 - Standardmäßig nutzt er „`/accounts/login/`“
- Wenn man kein Login-Template oder keine View dafür anlegen will, kann man folgende URL-Mapper-Regel benutzen

```
path( 'accounts/login/',  
      'django.contrib.auth.views.login',  
      dict(template_name='admin/login.html')  
)
```

- Sie nutzt
 - eine vom System bereitgestellte view-Funktion und
 - das Login-Seiten-Template des Admin-Interfaces
 - Letzteres kann natürlich leicht ersetzt werden um eine eigene Login-Seite passend zum Applikations-Design anzulegen

Django: Benutzerverwaltung

- **Feingranulare Benutzerrechte**

- Durch die feingranularen Benutzerrechte kann man weitaus präziser Zugriffe steuern.

- Beispiel:

```
def viewfunc(request):  
    if request.user.has_permission('pruefungsamt.change_vorlesung'):  
        # nur wenn man Vorlesungen ändern darf
```

- Oder wiederum als Dekorator:

```
@permission_required('pruefungsamt.change_vorlesung')  
def viewfunc(request):  
    # nur wenn man Vorlesungen ändern darf
```

- In Templates gibt es dazu die Kontext-Variable „**perms**“:

```
{% if perms.pruefungsamt.change_vorlesung %}  
    {# nur wenn man Vorlesungen ändern darf #}  
{% endif %}
```

- Man kann auch beliebig neue Rechte anlegen und darauf testen
 - Auch hier gilt dann: Superuser haben immer alle Rechte
 - Mehr dazu: <https://docs.djangoproject.com/en/4.2/topics/auth/default/>

Django: Migrationen

- **Schema-Migration**

- ist die **Anpassung der DB-Strukturen** an ein geändertes Daten-Modell
 - **Beispiel:** Die Professoren erhalten ein neues Attribut „Vorname“
- Schema-Migrationen treten durch die Weiterentwicklung der Applikation regelmäßig auf
 - Die Anpassung soll bei der Aktivierung einer neuen Software-Version weitgehend oder ganz automatisch erfolgen

- **Daten-Migration**

- ist die **Anpassung der Daten in der DB**
 - **Beispiel:** Alle Namen sollen ab jetzt mit Großbuchstaben anfangen
 - Neue Eingabe erfüllen das bereits, die Altdaten müssen angepasst werden
- Daten-Migrationen sind oft die Folge von Schema-Migrationen
 - **Beispiel:** Nach Einführung des Vornamen-Attributs sollen die Namensfelder am enthaltenen Komma in Name und Vorname aufgeteilt werden

Django: Migrationen

- **Migrationen sollen ...**
 - zuverlässig,
 - weitgehend automatisch und
 - umkehrbar ablaufen
- **Wunschvorstellung**
 - Bei der Aktivierung einer neuen Software-Version finden automatisch auch alle nötigen Migrationen statt
 - Danach ist das System ohne Nacharbeiten sofort wieder betriebsbereit.
 - Auch die Rückmigration nach einem Downgrade auf eine ältere Software-Version soll automatisch stattfinden.
 - selbst wenn zwischenzeitlich auf dem neuen System Daten verändert wurden
 - Die Erstellung der Schema-Migrations-Scripte erfolgt weitgehend automatisch
 - Man kann aber eingreifen

Django: Migrationen

- **Django legt zu jeder Migration ein Script an**
 - Für App „pruefungsamt“ z.B. in „pruefungsamt/migrations“
 - Die Scripte sind aufsteigend von 0001 durchnummeriert, z.B.
 - 0001_initial.py
 - 0002_auto__add_field_professor_vorname.py
 - In jedem der Scripte wird definiert, welche Änderungen erfolgen müssen
 - Damit ist es möglich, den Migrationsschritt von der nächst kleineren Stufe zu der Stufe der Scriptnummer hin machen oder diesen Schritt umzukehren
 - Zusätzlich speichert Django in der Datenbank die Nummer der aktuellen Migrationsstufe
 - Damit ist klar, in welcher Stufe das DB-Schema sich befindet ...
 - ... und was zu tun ist um jeweils eine Stufe auf- oder abzusteigen
 - Siehe <https://docs.djangoproject.com/en/4.2/topics/migrations/>

Django: Migrationen

- **Beispiel:**

- Wir legen in der App „pruefungsamt“ im Modell „Professor“ ein neues Attribut „**vorname**“ an

```
class Professor(models.Model):  
    persnr = models.IntegerField(max_length=10, unique=True)  
    name = models.CharField(max_length=64)  
    vorname = models.CharField(max_length=64, blank=True)
```

- Wir lassen manage.py das zugehörige Migrations-Script erzeugen

```
./manage.py makemigrations pruefungsamt --auto
```

→ 0002_auto__add_field_professor_vorname.py

```
class Migration(migrations.Migration):  
    # ...  
    operations = [  
        migrations.AddField(  
            model_name='Professor',  
            name='vorname',  
            field=models.CharField(blank=True, max_length=64),  
        ),  
    ]
```

Django: Migrationen

- **Wie generiert man Schemamigrations-Scripte?**

- Folgender Aufruf erzeugt automatisch ein Migrationsscript:

```
./manage.py makemigrations pruefungsamt --auto
```

- „--auto“ bewirkt dabei, dass der Script-Name automatisch erzeugt wird
 - z.B. „0002_auto__add_field_professor_vorname.py“ im obigen Beispiel
- Hier wird nichts an dem DB-Schema geändert, nur das Script erzeugt

- **Wie aktualisiert man das DB-Schema?**

- Folgender Aufruf bringt das DB-Schema auf Stufe 0002

```
./manage.py migrate pruefungsamt 0002
```

- Um alle Apps auf die jeweils **neueste** Stufe zu bringen:

```
./manage.py migrate
```

- Um zu sehen, welche Migrationen noch durchzuführen sind:

```
./manage.py showmigrations
```

Django: Migrationen

- **Datenmigration**

- Neben Schema-Anpassungen müssen bei Migrationen gelegentlich auch Daten angepasst werden.
- Das erfolgt in einer **Datenmigration** mit den gleichen Mechanismen wie oben in der **Schemamigration**, es gibt also auch ein Migrationsscript
- Die Daten-Migrationen müssen allerdings manuell erstellt werden
 - Das System kann ja nicht wissen was wir an den Daten ändern wollen

- **Beispiel:** Umwandlung *Personalnummer* (Integer) in *Personalkennung*
(Struktur *Abteilungs-Kennung* „-“ *Personalnummer*, z.B. „**FBINF-3587**“) 

- 1) **Schemamigration:** Attribut Personalkennung (Charfield) hinzufügen (Null=True)
- 2) **Datenmigration:** Für jeden Mitarbeiter: Personalkennung berechnen und setzen
- 3) **Schemamigration:** Null=True aus Attribut Personalkennung entfernen
- 4) **Schemamigration:** Altes Attribut Personalnummer löschen

→ Bei Aufruf von „./manage.py **migrate**“ werden dann alle Schritte ausgeführt.

- Mehr dazu:

<https://docs.djangoproject.com/en/4.2/topics/migrations/#data-migrations>

Web 2.0 Technologien 2

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
 - 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 enthaltenen RE wird namenlos im Ergebnis des Matchvorgangs gespeichert
 - Beispiel RE `r'^(ab)+$'` matcht `'ab'`, `'abab'`, `'ababab'` usw.
- **'[...]'** Eines der Zeichen in der Klammer
 - Beispiel: RE `r'^[ab]+$'` 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:

- Gibt es einen 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 mit „W“ beginnt und mit „h“ oder „r“ endet.

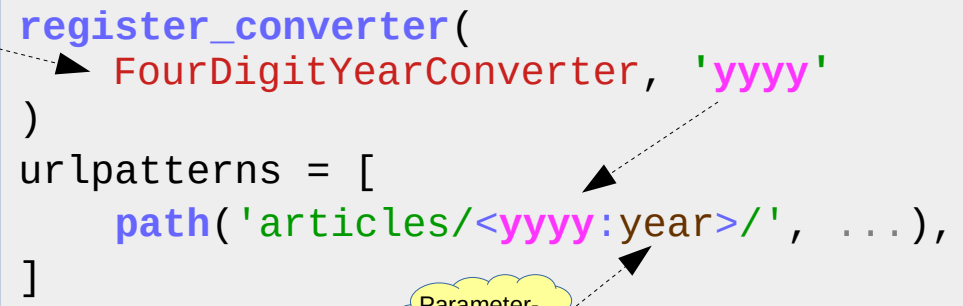
- Anwendung von REs: **Modelfield-Validatoren**

- `validators = [ RegexValidator(r'^#[0-9a-f]{6}$') ]`
 - `models.CharField` mit diesem `validators`-Parameter erlaubt nur 6-stellige CSS-Hex-Farangaben 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>

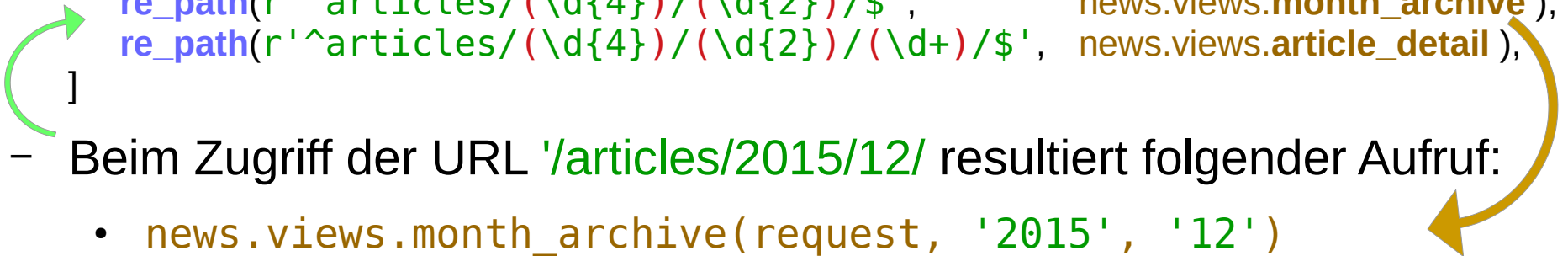
Django: Reguläre Ausdrücke

- 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 import 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).