Django의 핵심 기능

장고 파이썬 쉘로 데이터 조작하기

❖ 쉘 기동

- $ python manage.py shell
  - 내부적으로 mysite/settings.py 모듈을 임포트함

장고 파이썬 쉘로 데이터 조작하기

❖ Create - 데이터 생성/입력

```
>>> from polls.models import Question, Choice
>>> from django.utils import timezone
>>> q = Question(question_text = "What's new?", pub_date=timezone.now())
>>> q.save()
```

장고 파이썬 쉘로 데이터 조작하기

❖ Read - 데이터 조회

- 쿼리셋
  - 모든 모델은 objects 속성을 가짐

```
>>> Question.objects.all()
```

  - 쿼리셋의 메서드
    - all()　　: 모든 레코드 추출
    - get()　　: 단일 레코드 추출
    - filter_by() : 주어진 조건에 맞는 객체 추출(=연산자 이용)
    - filter()　: 주어진 조건에 맞는 객체 추출(기타 연산자 이용가능)
    - exclude()　: 주어진 조건에 맞지 않는 개체 추출

장고 파이썬 쉘로 데이터 조작하기

❖ Read - 데이터 조회

- 메서드 체인 가능

```
>>> Question.objects.filter(
    question_text__startswith='What'
).exclude(
    pub_date__gte=datetime.date.today()
).filter(
    pub_date__gte=datetime(2005, 1, 30)
)
```

장고 파이썬 쉘로 데이터 조작하기

❖ Read - 데이터 조회

- 단일 레코드 추출

```
>>>  one_entry = Question.objects.get(pk=1)
```

- 슬라이싱을 이용한 offset, limit 처리

```
>>> Question.objects.all()[:5]
>>> Question.objects.all()[5:10]
>>> Question.objects.all()[:10:2]
```

장고 파이썬 쉘로 데이터 조작하기

❖ Update - 데이터 수정

- 단일 레코드 수정
  - 모델 인스턴스의 필드 수정 후 save() 호출

```
>>> q.question_text = 'What is your favorite hobby ?'
>>> q.save()
```

- 다중 레코드 수정
  - 쿼리 셋의 filter로 대상 지정후 update() 호출

```
>>> Question.objects.filter(pub_date__year=2007).update(
     question_text='Everything is the same')
```

장고 파이썬 쉘로 데이터 조작하기

❖ Delete - 데이터 삭제

- 단일 데이터 삭제

```
q.delete()
```

- 다중 데이터 삭제

```
Question.objects.filter(pub_date__year=2005).delete()
```

- 모든 데이터 삭제

```
Question.objects.delete()
```

장고 파이썬 쉘로 데이터 조작하기

❖ polls 애플리케이션의 데이터 실습

```
>>> from polls.models import Question, Choice

>>> Question.objects.all()

>>> Choice.objects.all()

>>> from django.utils import timezone
>>> q = Question(question_text="What's up?", pub_date=timezone.now())
>>> q.save()
>>> q.id

>>> q.question_text

>>> q.pub_date

>>> q.question_text="What's new ?"
>>> q.save()
>>> Question.objects.all()
```

장고 파이썬 쉘로 데이터 조작하기

❖ polls 애플리케이션의 데이터 실습

```
>>> from polls.models import Question, Choice

>>> Question.objects.filter(id=1)

>>> Question.objects.filter(question_text__startswith='What')

>>> from django.util import timezone
>>> current_year = timezone.now().year
>>> Question.objects.get(pbu_date__year=current_year)

>>> Question.objects.get(id=100)# 예외 발생


>>> Question.objects.get(pk=1)  # get(id=1)와 동일

# Foreign Key가 지정되면
# 대상 모델에도 해당 모델명_set으로 QuerySet이 생성됨

>>> q = Question.objects.get(pk=2)
>>> q.choice_set.all()
```

장고 파이썬 쉘로 데이터 조작하기

❖ polls 애플리케이션의 데이터 실습

```
>>> q.choice_set.create(choice_text='Sleeping', votes=0)

>>> q.choice_set.create(choice_text='Eating', votes=0)

>>> c = q.choice_set.create(choice_text='Playing', votes=0)
>>> c.question

>>> q.choice_set.all()

>>> q.choice_set.count()


>>> Choice.object.filter(question_pub_date__year = current_year)

>>> c = q.choice_seet.filter(choice_text__startswith='Reading')
>>> c.delete()
```

템플릿 시스템

❖ 템플릿 변수

- {{변수}}
  - 변수를 평가하여 변수 값을 출력

- .연산자 처리 절차(foo.bar)
  - foo가 사전 타입인지 확인 -> foo['bar']로 해석
  - 사전이 아닌경우 속성을 확인 --> foo.bar로 해석
  - 그것도 아니면 리스트인지 확인 --> foo[bar]로 해석

- 정의 되지 않은 변수는 ''로 처리

템플릿 시스템

❖ 템플릿 필터

- {{ 표현식 | 필터명 }}

- 내장 필터
  - lower `{{ name | lower }}`
  - escape `{{ text | escape }}`
  - linebreaks `{{ text | escape | linebreaks }}`
  - truncatewords:길이 `{{ bio | truncatewords:30 }}`
  - join `{{ list | join: " // " }}`
  - default `{{ value | default:'nothing' }}`
  - length `{{ value | length }}`
  - striptags `{{ value | striptags }}`
  - pluralize `{{ value | pluralize }}`
  - add `{{ value | add:"2" }}`

템플릿 시스템

❖ 템플릿 태그

- {% tag %}

- {% for %} 태그

```
<ul>
{% for athlete in athlete_list %}
    <li>{{athlet.name}}</li>
{% endfor %}
</ul>
```

- for 태그에 사용되는 변수
  - forloop.counter 현재까지 루프를 실행한 루프 카운트(1부터)
  - forloop.counter0 현재까지 루프를 실행한 루프 카운트(0부터)
  - forloop.recounter 루프 끝에서 현재가 면 번째인지 카운트한 숫자(1부터)
  - forloop.recounter0 루프 끝에서 현재가 면 번째인지 카운트한 숫자(0부터)
  - forloop.first 루프에서 첫 번째 실행이면 True 값을 가짐
  - forloop.last 루프에서 마지막 실행이면 True 값을 가짐
  - forloop.parentloop 중첩된 루프에서 현재의 루프 바로 상위 루프

템플릿 시스템

❖ 템플릿 태그

- % if %} 태그

```
{% if athlete_list %}
    Number of athletes: {{ athlete_list | length }}
{% elif athlete_in_locker_room_list %}
    Athletes should be out of the locker room soon!
{% else %}
    No athletes.
{% endif %}
```

- {% if athlete_list|length > 1%}
  - 사용 가능한 연산자
    - and, or, not, and not, ==, !=, <, >, <=, >=, in, not in

템플릿 시스템

❖ 템플릿 태그

- {% csrf_token %} 태그
  - CSRF(Cross Site Request Forgery) 공격 방지

```
<form>{% csrf_token %}
```

  - CSRF 토큰값 검증 실패시 403 에러 발생

템플릿 시스템

❖ 템플릿 태그

- {% url %} 태그
  - {% url '네임스페이스:뷰이름' [파라미터] %}

템플릿 시스템

❖ 템플릿 태그

- {% with %} 태그
  - 특정 값을 변수에 저장해 두는 기능

```
{% with total=business.employees.count %}
    {{total}} people works at business
{% endwith %}
```

```
{% with business.employees.count as total %}
    {{total}} people works at business
{% endwith %}
```

템플릿 시스템

❖ 템플릿 태그

- {% load %} 태그
  - 사용자 정의 태그 및 필터를 로딩

```
{% load somelibrary package.otherlibrary %}
```

템플릿 시스템

❖ 템플릿 주석

- 단일 라인 주석 {# #}

```
{# greeting #}hello

{# {% if foo %}bar{% else %} #}
```

- 여러 줄 주석 {% comment %}

```
{% comment "Optional note" %}
<p>Commented out text here</p>
{% endcomment %}
```

템플릿 시스템

❖ HTML 이스케이프

- 디폴트로 태그 문자는 이스케이프됨

- HTML 태그를 살리고자 하는다면 safe 필터 사용

```
{{ data | safe }}

{% autoescape off %}
Hello {{name}}
{% endautoescape %}

{{ data | default: "3 < 4" }} // 스트링 리터럴 에서는 자동 이스케이프 안됨
{{ data | default: "3 &lt; 4" }}
```

템플릿 시스템

❖ 템플릿 상속

- 부모 템플릿
  - 기본 구조와 블럭의 위치 지정

```
{% block 블럭명 %}{%endblock %}
```

  - 주로 templates/ 디렉토리에 정의

- 자식 템플릿
  - 부모 템플릿 상속

```
{% extends "base.html" %}
```

재정의할 블럭 구현

```
{% block 블럭명 %}{%endblock %}
```

폼 처리하기

❖ HTML에서의 폼

- GET 정보의 추출
- POST 정보의 저장

폼 처리하기

❖ 장고의 폼 기능

- 폼 생성에 필요한 데이터를 폼 클래스로 구조화하기(forms.py)
- 폼 클래스의 데이터를 렌더링하여 HTML 폼 만들기(템플릿.html)
- 사용자로부터 제출된 폼과 데이털르 수신하고 처기하기(views.py)

폼 처리하기

❖ 폼 클래스로 폼 생성

- 폼 클래스

```
from django import forms

class NameForm(forms.Form):
    your_name = forms.CharField(label="Your name", max_length=100)
```

- 렌더링 결과

```
<label for="your_name">Your name: </label>
<input id="your_name" type="text" name="your_name" maxlength="100">
```

- 위젯 지정하기

```
your_name = forms.CharField(label="Your name", max_length=100,
                            widget=forms.Textarea)
```

폼 처리하기

❖ 폼 클래스로 폼 생성

- is_valid() 메서드
  - 모든 필드에 대해 유효성 검사 루틴 실행
  - 유효성 검사 성공시
    - 폼 데이터를 cleaned_data 속성에 저장(필드명을 키로하는 사전)
    - True를 리턴

- 폼 클래스 사용하기

```
<form action="..." method="POST">
{% csrf_token %}
{{form}}
<input type="submit" value="Submit">
</form>
```

폼 처리하기

❖ 뷰에서 폼 클래스 처리

- 2개의 뷰가 필요
  - 폼을 보여주는 뷰(GET)
  - 제출된 폼을 처리 하는 뷰(POST)

  --> 하나의 뷰로 통합하여 운영 권장

폼 처리하기

❖ 뷰에서 폼 클래스 처리

- ㅇ 통합 뷰 코드

```
from django.shortcuts import render
from django.http import HttpResponseRedirect

def get_name(request):
    if request.method == 'POST':  # POST 요청
        # request에 담긴 데이터로, 클래스 폼을 생성
        form = NameForm(request.POST)
        if form.is_valid():
            new_name = form.cleaned_data['name']
            # 로직에 따라 추가 처리

            return HttpResponseRedirect('/thanks/')

    else: # GET 요청
        form = NameForm()

    # GET 요청시, POST 요청시 유효성 검사 실패시
    return render(request, 'name.html', {'form': form })
```

폼 처리하기

❖ 폼 클래스를 템플릿으로 변환

- {{form}} 구문
  - <label>과 <input> 엘리먼트 상으로 렌더링됨
  - 다른 옵션
    - {{form.as_table}} : <tr> 태그로 감싸서 테이블 셀로 렌더링
    - {{form.as_p}} : <p> 태그로 감싸서 렌더링
    - {{form.as_ul}} : <li> 태그로 감싸서 렌더링

폼 처리하기

❖ 폼 클래스를 템플릿으로 변환

- ContactForm 폼 클래스 정의

```
from django import forms

class ContactForm(forms.Form):
    subject = forms.CharField(max_length=100)
    message = forms.CharField(widget=forms.Textarea)
    sender = forms.EmailField()
    cc_myself = forms.BooleanField(required=False)
```

- 렌더링 결과

```
<p><label for="id_sbject">Subject:<label>
      <input id="id_subject" type="text" name="subject" maxlength="100"/></p>

<p><label for="id_message">Subject:<label>
      <textaread id="id_message" name="message"></textarea></p>

<p><label for="id_sender">Subject:<label>
      <input id="id_sender" type="email" name="sender"/></p>

<p><label for="id_cc_myself">Subject:<label>
      <input id="id_cc_myself" type="text" name="cc_myself"/></p>
```

클래스형 뷰

❖ 클래스형 뷰의 시작점

- urls.py

```
from django.conf.urls import patterns
from myapp.views import MyView

urlpatterns = patterns('', MyView.as_view())
```

- as_view()
  - 클래스의 인스턴스를 생성
  - 인스턴스의 dispatch() 메서드 호출
  - GET/POST 구분하여 해당 이름을 갖는 메서드 요청
  - 해당 메소드가 정의되지 않은 경우 HttpResponseNotAllowed 예외 발생

클래스형 뷰

❖ 클래스형 뷰의 장점 - 효율적인 메서드 구분

- GET, POST 등의 HTTP 메소드에 따른 처리 기능을 코딩할 때, 메서드 명으로 구분
  - get(), post(), head() 등
- 다중 상속 기능 가능, 클래스형 제너릭 뷰 및 믹스인 클래스 등 사용 가능

```
class MyView(View):
    def get(self, request):
        # 뷰 로직 작성
        return HttpResponse('result')
```

클래스형 뷰

❖ **클래스형 뷰의 장점 - 상속 기능 가능**

- 장고의 제너릭 뷰 상속

```
from django.views.generic import TemplateView

class AboutView(TemplateView):
    template_name = "about.html"
```

클래스형 뷰

❖ 클래스형 제너릭 뷰

- 제너릭 뷰
  - 공통 로직을 미리 개발해 놓고 제공하는 뷰

- BaseView
  - 뷰 클래스를 생성하고, 다른 제너릭 뷰의 부모 클래스 역할
- Generic DisplayView
  - 객체의 리스트를 보여주거나, 특정 객체의 상제 정보 출력
- Generic Edit View
  - 폼을 통해 객체를 생성, 수정, 삭제하는 기능
- Generic Date View
  - 날짜 기반 객체의 년/월/일 페이지로 구분

클래스형 뷰

❖ 클래스형 제너릭 뷰

- BaseView
  - View : 가장 기본이 되는 최상위 제너릭 뷰
  - TemplateView : 템플릿이 주어지면 해당 템플릿을 렌더링
  - RedirectView : 지정한 URL로 리다이렉트

- Generic Display View
  - DetailView : 객체 하나에 대한 상세한 정보 출력
  - ListView : 조건에 맞는 여러 개의 객체 출력

- Generic Edit View
  - FormView : 폼을 보여줌
  - CreateView : 객체를 생성하는 폼 출력
  - UpdateView : 객체를 수정하는 폼 출력
  - DeleteView : 객체를 삭제하는 폼 출력

- Generic Date View
  - YearArchiveView : 주어진 년도에 해당하는 객체를 출력
  - MonthArchiveView : 주어진 월에 해당하는 객체를 출력
  - DayArchiveView : 주어진 날짜에 해당하는 객체를 출력

클래스형 뷰

❖ 클래스형 뷰에서 폼 처리

- 폼 처리 과정
  - 최초의 GET 요청: 폼은 비어 있거나 미리 채워진 데이터를 가짐
  - 유효한 데이터를 가진 POST: 데이터를 처리함. 처리 후 리다이렉트 함
  - 유효하지 않은 데이터를 가진 POST: 에러 메시지와 함께 폼을 다시 출력 함

클래스형 뷰

❖ 클래스형 뷰에서 폼 처리

- 함수형 뷰

```
def myview(request):
    if request.method == 'POST':
        form = MyForm(request.POST)
        if form.is_valid():
            # form.cleaned_data로 관련 로직 처리
            return HttpResponseRedirect('/success/')
    else:
        form = MyForm(initial={'key': 'value'})

    return render(request, 'form_template.html', {'form': form})
```

클래스형 뷰

❖ 클래스형 뷰에서 폼 처리

- 클래스형 뷰

```
class MyFormView(View):
    form_class = MyForm
    initial = { 'key': 'value' }
    template_name = 'form_template.html'

    def get(self, request, *args, **kwargs): # 최초의 GET 요청
        form = self.form_class(initial = self.initial)
        return render(request, self.templatge_name, {'form': form}

    def post(self, request, *args, **kwargs):# POST 요청
        form = self.form_class(request.POST)
        if form.is_valid():  # 유효성 검사 통과
            # form.clelan_data로 관련 로직 처리
            return HttpResponseRedirect('/success/')

        #  유효성 검사 실패
        return render(request, self.template_name, {'form': form})
```

클래스형 뷰

❖ 클래스형 뷰에서 폼 처리

- 제너릭 뷰 상속

```
class MyFormView(FormView):
    form_class = MyForm
    template_name = 'form_template.html'
    success_url = '/success/'

    def form_valid(self, form):
        # self.clean_data로 관련 로직 처리
        return super(MyFormView, self).form_valid(form)
```

클래스형 뷰

❖ 클래스형 뷰에서 폼 처리

- 주요 클래스 변수 및 메서드
  - form_class: 사용자에 보여줄 폼 클래스 지정
  - template_name: 폼을 포함하여 렌더링할 템플릿 파일 이름
  - success_url: 처리가 정상적으로 완료되었을 때 리다이렉트시킬 URL

  - form_valid() : 유효한 폼 데이터를 처리할 로직 코딩. 반드시 super() 함수를 호출해야 함