​Important Questions?

How to do Migrations?

Using database Queries in Django ORM.

Django Testing

Django debug toolbar.

Doubts?

1. Adding static files- - F:\Python Tutorials\Python-Django\01. Basics\04- Django Templates – Static assets
2. How to use ‘url’ inside the templates?

**BASICS**

* A web framework for python
* Its mainly used for creating CMS like Applications but it can be used to create any webapplication that we want to.
* pip3 install django
* python -m django —version to check the version
* django installs a handy command line tool named ‘djando-admin’ for doing administrative tasks.
* Type python manage.py to get all the commands.:

**Django Admin uses:**

* django-admin startproject <name> : creates new project

Put your code in some directory outside of the document root, such as /home/mycode.

* django-admin help to get all the available commands in django admin
* manage.py is our entry point to Django commands inside of our project?
* Running server: Navigate to the root directory.

- use following command:

python3 manage.py runserver

python manage.py runserver 8080

python3 manage.py runserver 0.0.0.0:8000

The server restarts when change a code.

But some actions like restarting don’t trigger a restart and we have to do it manually.

* Manage.py:
* manage.py does the same thing as django-admin
* It puts your project’s package on sys.path.
* It sets the DJANGO\_SETTINGS\_MODULE environment variable so that it points to your project’s settings.py file.
* ‘python manage.py’ gives list of all commands that we need.
* A Project Vs an App
* An app is a Web application that does something – e.g., a Weblog system, a database of public records or a simple poll app.
* A project is a collection of configuration and apps for a particular website.
* A project can contain multiple apps.
* An app can be in multiple projects. (that is apps can be pluggabe)
* An app name is generally the plural form of the main model that we want to use in the app.
* Use command ‘python manage.py startapp <appname>’ to create an app.
* Add the app in ‘settings.py> INSTALLED\_APPS’ tuple to use it inside the project.
* What are migrations?
* Migration is way of moving our database from one design to another.
* We are changing the orgaization and sturcture of the database from one configuration to another.

Migrations are a way of moving a database from one design, a specific set of tables and columns, to a new one. Migrations are reversible, too. The fact that they can be done backwards and forwards is what gives them their name.

By running makemigrations, you’re telling Django that you’ve made some changes to your models (in this case, you’ve made new ones) and that you’d like the changes to be stored as a migration.

* **Django:**

Its an MVC - Model View Controller

Django calls templates as templates and the functions that return rendered templates as views.

* Django apps

A project can contain many apps

App names are generally the plural forms of the main thing the project is about.

We can create a new app by being on the same directory as Manage.py and using :

python3 manage.py startapp courses

To add apps to our project , we can go to ‘setting.py’ and add the app name in the ‘INSTALLED\_APPS’ list.

When you add new apps to INSTALLED\_APPS, be sure to run manage.py migrate, optionally making migrations for them first with manage.py makemigrations.

Django apps are “pluggable”: You can use an app in multiple projects, and you can distribute apps, because they don’t have to be tied to a given Django installation.

* **Models in Django:**

Important references :

Fields - https://docs.djangoproject.com/en/1.10/ref/models/fields/#model-field-types

#A model is the single, definitive source of information about your data. It contains the essential fields and behaviors of the data you’re storing. #Generally, each model maps to a single database table

In Djangos ORM:

* Models are classes that represent a Database table.
* Each model is a table.
* Each attribute on a class is a column inside the table.
* When we create a new instance of the model class to the database, the ORM creates a new row in the table.
* By convention, model names should be singular.
* **The basics:**

Each model is a Python class that subclasses django.db.models.Model.

Each attribute of the model represents a database field.

With all of this, Django gives you an automatically-generated database-access API that we can access via the django shell

* **Django Shell:**

There are two ways of using the python shell:

Run python manage.py shell

a) Set the DJANGO\_SETTINGS\_MODULE environment variable to mysite.settings,

b)start a plain Python shell, and

c) set up Django: (import django and django.setup() )

* **Steps to Do :**

1**. Create a Model**

When we create an app in django, the app folder by default has a models.py module.

Our models (database) for the app goes into it.

from django.db import models

class Course(models.Model):

created\_at = models.DateTimeField(auto\_now\_add=True)

title = models.CharField(max\_length=255)

description = models.TextField()

  def \_\_str\_\_(self):

return self.title

# this \_\_str\_\_ method is used to return the model object as a string that is easily readable in the shell and in the admin.

**2. Migrate it.**

After the Model (database) is created(or edited), we need to migrate the app using.

python manage.py makemigrations <app\_name>

python manage.py migrate courses

**3. Add data (instance) to it.**

There are two ways of adding data to model:

* Using the manage.py shell.

we can use the powerful command line shell of manage.py to add and retrieve data.

Manage.py shell can be used to explore the ORM

Run the command “python3 manage.py shell” to open the mangae.py shell

Adding data to the Model using Manage.py shell.

>>> from courses.models import Course

>>> Course.objects.all() # returns a queryset for list of entries inside the model.

<QuerySet []>

>>> c = Course() # Creating an instance of our model.

>>> c.title = "Python Basics"

>>> c.description = "Learn the baiscs of Python"

>>> c.save()

>>> Course.objects.all()

<QuerySet [<Course: Course object>]>

>>>

Easier way to add all the data at once:

from courses.models import Course

>>> Course(title = "Python collections", description="Learn about python collections").save()

>>> Course.objects.all()

<QuerySet [<Course: Course object>, <Course: Course object>]>

>>>

Yet another easier way to add data in on go:

Course.objects.create(title = "Python Django", description = "Python Web Framework")

<Course: Course object>

(This returns an object as well so can be useful)

**Some Important Methods for managing Models in Manage.py shell**

* python manage.py shell opens a Python shell with Django's configuration already loaded.

Model.save() will save an in-memory instance of a model to the database.

* Model.create() will save an in-memory instance of a model to the database and return the newly-created object.
* Model.filter(attribute=value) will get a QuerySet of all instances of the Model that match the attribute values. You can change these values with other comparisons, too, like gte or in
* Model.objects.all() – returns the queryset of all the entries inside our app.

<https://docs.djangoproject.com/en/1.8/topics/db/queries/#retrieving-objects>

## Using the Django’s admin:

* Go the url/admin. where you will see the username asking page.
* “python3 manage.py createsuperuser” to create user
* **Go to admin.py module and register the model using**

admin.site.register(Course).

This is way to tell the admin that ‘Course’ Object has an admin interface.

* Now login to the Django Admin using the username and password.Here, we can see all the models that we have created.

### Making Views:

* What is a View: A view is a “type” of Web page in your Django application that generally serves a specific function and has a specific template

Making URLs for the Views:

* create a new urls.py in the app
* Import url and views in this file
* add URL of the page using ‘url\_patterns’

from django.conf.urls import url

from . import views

urlpatterns = [  url(r'^$', views.course\_list) ]

**But django won’t look in to this ‘urls.py’ of our app, it looks in to the urls.py of the main project**. So we have to mention in the projects urls.py about our apps url

* Go to urls.py of the project
* in the url\_pattern add the following to ask django to look into this url.py file when it sees a specific url pattern :
* url(r'^courses/', include('courses.urls')),

—————————————————— ——————————————————

## Django Templates:

Here we do two things,

a) We Create a Template

- In Django templates can be any language that we want like html, xml, json

- **By default django looks for the template directory inside our app directory**. So we need to create a new on in our app directory if we don’t have one.

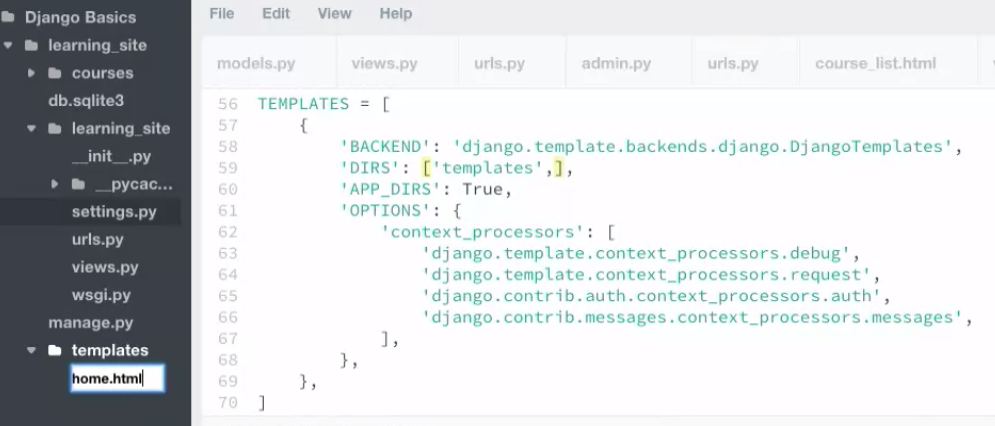
- Also, inside the ’templates’ django looks for the folder with same name as the app (so make it if not present). So all the apps are namespace in the templates folder.

b) Make the ‘view’ to use our template:

- To show templates from our views, we use the ‘render’ function as a return value.

- ‘return render(request, ‘<template: ‘course/course\_list.html’>, <context>: { ‘courses’: course})

To create a templates for our project, we need to create a new ‘template’ folder in the project directory and update the ‘TEMPLATES’ list in Settings.py

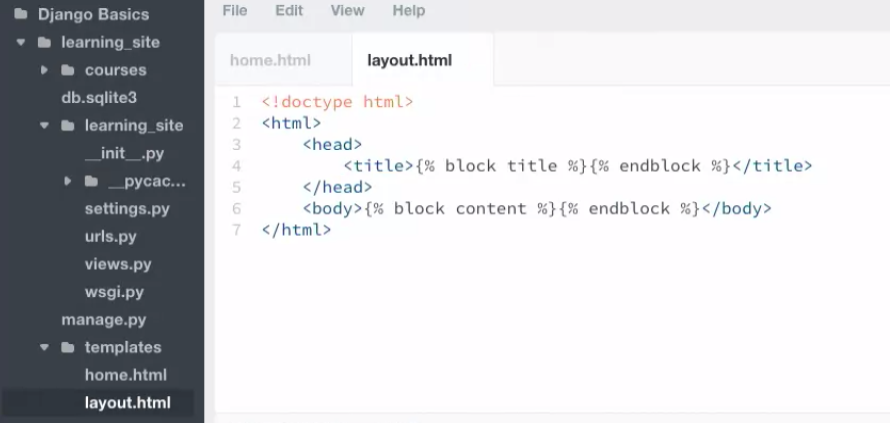


Template Inheritance:

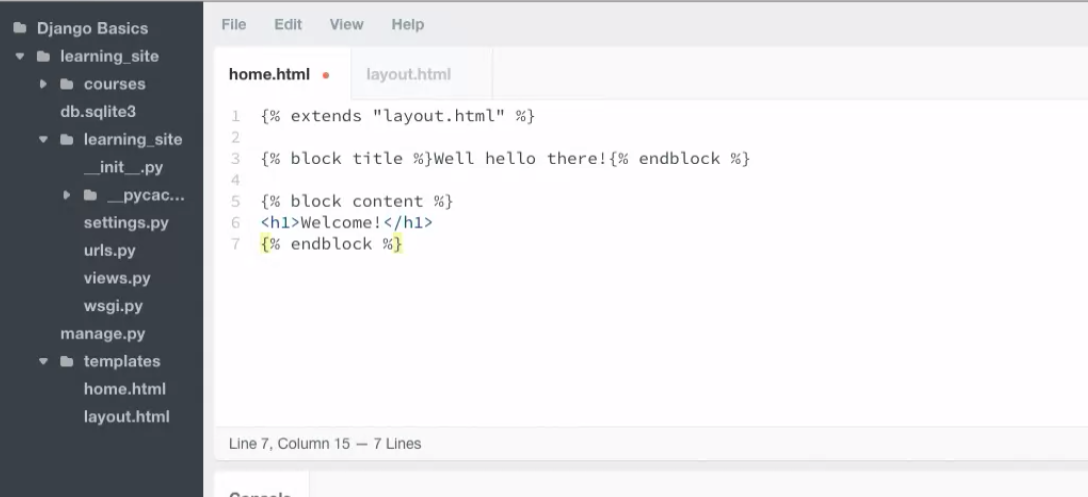
* A template is simply the design part of our application.
* We can avoid repeating our html code in different templates.
* This can done using template inheritance.
* We can inherit the html of the parent template and then override the block of things

syntax: {% extends "layout.html" %}

Parent Template:



Child template that extends Parent Template:



Using Static Assets in our template:

In the project directory, create a new folder ‘assets’

In the settings.py, scroll down to the - STATIC\_URL = '/static/'

Add a new attribute over here :

STATICFILES\_DIRS = (  os.path.join(BASE\_DIR, 'assets'), )

This tells django where to look for the static files.

Now go to the urls.py of the project

from django.contrib.staticfiles.urls import staticfiles\_urlpatterns

from django.http import HttpResponse

def index(request):

return HttpResponse("Hello Treehouse")

— The template system uses dot-lookup syntax to access variable attributes. In the example of {{ question.question\_text }}, first Django does a dictionary lookup on the object question. Failing that, it tries an attribute lookup – which works, in this case. If attribute lookup had failed, it would’ve tried a list-index lookup.

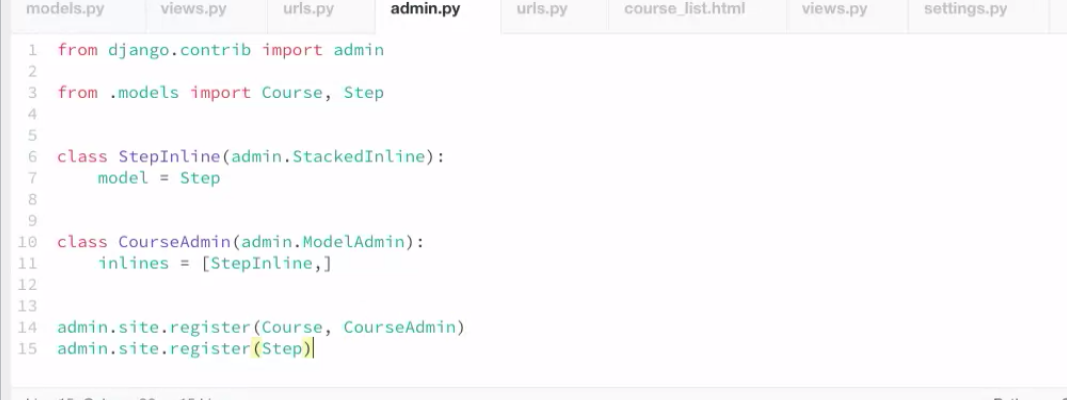
## Django Admin:

1. Creating Inlines:

* We can use one Model into another inside the admin, for easy editing:

1. Create an inline class with the smaller form that we want to add into the larger form.
2. Add that class to the admin class of the larger form.

Below we are adding steps to our Courses model using Stackedinline.



## Django ORM:

* Object Relational Mapper
* Each class the extends the django.db.model represents a table in the database and the attributes of the class represents the table columns.
* Queries: (Lets take a model as ‘Course’)

-Lookups are the functions that take keyword arguments and perform some quries into our database. We have filter(), exclude(), get() as some of the lookups that we use.

Insertion Queries:

1. Course.objects.create(<data>)

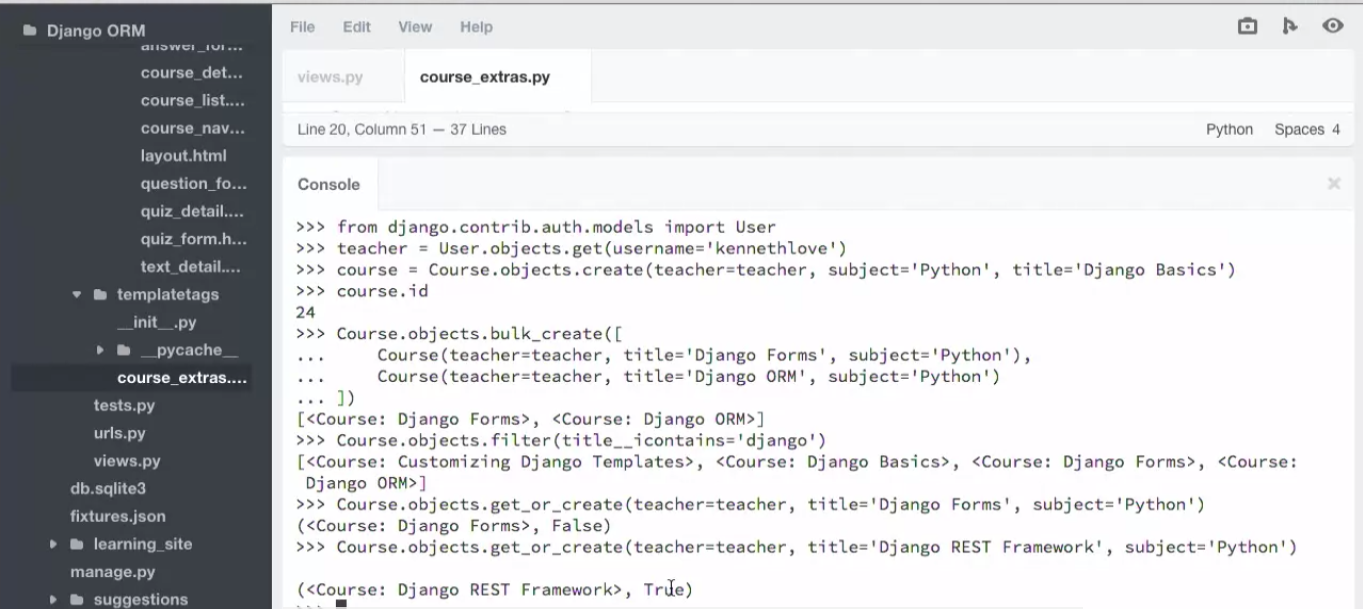
-create method inserts into database and gives us back the record.

b) Course.objects.bulk\_create(<Iterable having instances of the models that are not saved>)

c) Course.objects.get\_or\_create(<data>)

-creates an object into database if its not there , and returns the newly created data and ‘true’.

- if data is already there, then it fetches the data returns it with ‘false’ (meaning that the data was not created).



1. Courses.objects.all()
2. Course.objects.none()
3. Courses.objects.get()
4. Get\_object\_or\_404(Courses, <some equality>)
5. Course.objects.filter(<some condition>) –to get only those results that match the condition.
6. Course.objects.exclude(<some conditions>) – to get only those results that donot match the condtion.
7. Course.objects.update(<attribute=new\_value>)- To update the records using some new value. This by defaults happens on all the objects of the queryset and not on a single object.
8. .delete() method on a Model object can be used to delete all the obejects form the query set. So we normally filter our results based on what we want and we then use delete () on the query set.
9. filter(<columnName>\_\_icontains=<Something>) – is the query to filter rows where a column contains some value ( ‘i’ meaning that the value is case insensitive).
10. Selecting on specific columns from the queryset:
    * Course.objects.filter(<condition>).values(<column1>, <column2>).
    * values\_list() ------- ???
11. Limiting the results:
    * Course.objects.filter(<condition>).values(<column1>, <column2>)[:5]

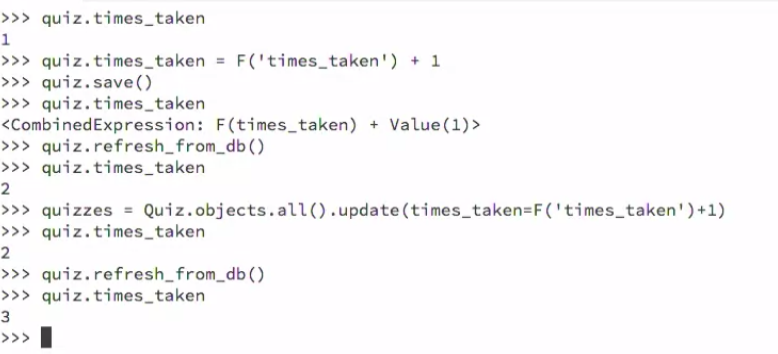
12 . Ordering our results:

* + Course.objects.filter(<condition>).order\_by(‘<columnName>’).values(‘<column1>, <column2>)[:5]
  + Use ‘-<columnname>’ to sort in descending order.

1. Course.objects.latest(<column>) – get the lastest entry from the database based on the column name

F Objects:

* F Objects in Django helps to get the value of an database column from the database itself instead of an object that we created as the object might well be outdated.
* Its helps to pervent race condition inside the database.
* Suppose we have a ‘Quiz’ Model and quiz is an instance of that model and ‘time\_taken’ is a field that represents the number of the times the quiz was taken.



## Q Objects:

* Q objects can be used to make complex queries like those using OR condition.
* Noramlly the keywords args passed into lookup functions are ‘And’ed.
* A [Q object](https://docs.djangoproject.com/en/1.11/ref/models/querysets/#django.db.models.Q) (django.db.models.Q) is an object used to encapsulate a collection of keyword arguments.
* For example, this Q object encapsulates a single LIKE query:
  + from django.db.models import Q
  + Q(question\_\_startswith='What')
* Q objects can be combined using the & and | operators. When an operator is used on two Q objects, it yields a new Q object.
* For example, this statement yields a single Q object that represents the “OR” of two "question\_\_startswith" queries:

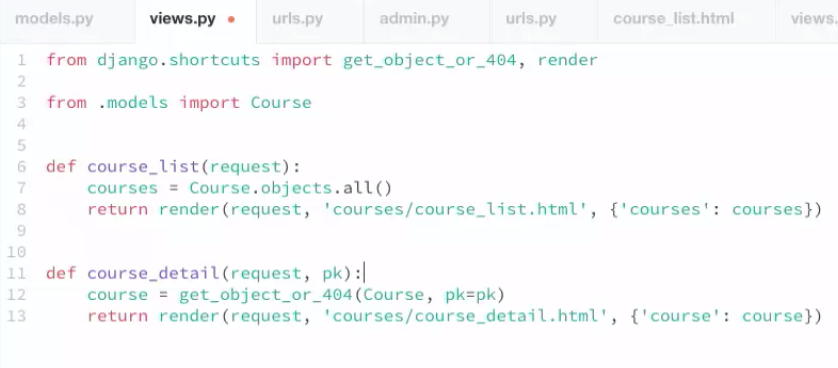
Q(question\_\_startswith='Who') | Q(question\_\_startswith='What')

* This is equivalent to the following SQL WHERE clause:

WHERE question LIKE 'Who%' OR question LIKE 'What%'

* A queryset is a collections of objects returned from a database.
* A queryset is lazy, and is used whenever we use them.

Handling 404 errors in Django:



Making References to foreign key relations in reverse:

The use of the "\_set" suffix is to reference foreign key relationships in reverse. Since Step has a foreign key to Course and is referred to as step.course, the reverse relationship is automatically created for you by Django. A Course can refer to all Step objects that point to it using the "\_set" notation as course.step\_set.

The ".all()" part means all records. It could also have been a filter or a get