Q-1 Why Django should be used for webdevelopment? Explain how youcan create a project in Django?

Ans. Django is the best framework for web applications, as it allows developers to use modules for faster development. As a developer, you can make use of these modules to create apps, websites from an existing source. It speeds up the development process greatly, as you do not have to code everything from scratch.

Using Django, you can create professional web apps and websites in a short window. The platform is known for its advanced functionality like admin panels, authentication support, comment boxes, file upload support, contact forms, app management, and more.

There is a large collection of modules that you can use for Django website development projects. It is among the best framework for web development, according to developers in the community due to its simplicity and rich pool of features.

ince Django is written in pure Python, it was originally built around the Model View Controller (MVC) framework. The concept is still applicable in the current version of the framework, as well. But when it comes to Django, developers usually refer to the architecture of Django as Model View Template (MVT). Three distinct layers are responsible for supporting the architecture, and they can be used separately in Python Django framework.

Models hold information about all your data, and they are represented using attributes/fields. Models have no

information about Django layers.Communication between multiple layers is made possible only through an API.

According to the official documentation, the Django model is designed to be a single source for all your data. It has all the fields and behaviours for storing data. All models within Django direct to a single database table, making the development process streamlined regardless of the data type you are storing. The The Django web framework supports major database frameworks which include:

PostgreSQL

SQLite

Oracle

DRY Philosophy

Django's DRY (don't repeat yourself) philosophy is what helps developers achieve rapid development. You can work on more than one iteration of an app at a time without working on the code from scratch. You can also reuse existing code for future apps while maintaining authenticity and unique features.

Security Measures

One of the biggest priorities for Django as a platform is the security systems. You do not have to implement security features manually to keep web development going. You don't have to worry about common security concerns like SQL injection, cross-site scripting, and clickjacking using the platform. There are frequent security patches that are

deployed on the platform to ensure all of the latest security standards are met.

Usable on Any Web App Project

You can tackle just about any web app development project, whether it is a basic website or a high-end web app. Django is full of features and is completely scalable. You can build apps that are designed to handle high volumes of information. Django web application is also cross-platform, which makes them usable across Linux, Windows, and Mac. It is also compatible with all major databases and even works with multiple database management systems at the same time.

Reliability

Django has been around for a while now, and its large community makes the platform even better. There are dedicated websites for the platform where you can find help for any issues that you may run into. If you need help with your projects, the community support can always be banked on.

Even if we put aside the community support available, the technical documentation available for Django is fleshed out, making it easy for developers to refer to in case of any difficulties. Django receives constant updates with all the latest packages to make working with Django a streamlined experience.

Creating the project

To create the project:

Open a command shell (or a terminal window), and make sure you are in your virtual environment.

Navigate to where you want to store your Django apps (make it somewhere easy to find like inside your Documents folder), and create a folder for your new website (in this case: django_projects). Then change into your newly-created directory:

mkdir django_projects

cd django_projects

Create the new project using the django-admin startproject command as shown, and then change into the project folder:

django-admin startproject locallibrary

cd locallibrary

The django-admin tool creates a folder/file structure as follows:

locallibrary/

manage.py

locallibrary/

__init__.py

settings.py

urls.py

wsgi.py

asgi.py

Our current working directory should look something like this:

../django_projects/locallibrary/

The locallibrary project sub-folder is the entry point for the website:

__init__.py is an empty file that instructs Python to treat this directory as a Python package.

settings.py contains all the website settings, including registering any applications we create, the location of our static files, database configuration details, etc.

urls.py defines the site URL-to-view mappings. While this could contain all the URL mapping code, it is more common to delegate some of the mappings to particular applications, as you'll see later.

wsgi.py is used to help your Django application communicate with the web server. You can treat this as boilerplate.

asgi.py is a standard for Python asynchronous web apps and servers to communicate with each other. ASGI is the asynchronous successor to WSGI and provides a standard for both asynchronous and synchronous Python apps (whereas WSGI provided a standard for synchronous apps only). It is backward-compatible with WSGI and supports multiple servers and application frameworks.

The manage.py script is used to create applications, work with databases, and start the development web server.

Q-2 How to check installed version of Django?

Ans

you will also get to know how to install a specific version of Django.

Method 1

The very first method to check the version of Django is simply to write python -m django –version in your terminal.

python -m django --version

This code will return the version of your Django in the format 3.2.9. If python doesn’t work replace it with python3.

Method 2

This method is almost similar to the first one and also gives the same output as the above method. The second method is django-admin –version

django-admin --version

Again it will only return 3.2.9 as output.

Method 3

The third method is known to almost all of us, but we might not have paid much attention to it. Do you remember what is the output when you write python manage.py runserver in your terminal.

python manage.py runserver

Here is the output,

tching for file changes with StatReloader

Performing system checks...

System check identified no issues (0 silenced).

February 22, 2022 - 20:32:37

Django version 4.0.2, using settings 'myproject.settings'

Starting development server at http://127.0.0.1:8000/

Quit the server with CTRL-BREAK.

E:\codespeedy\codespeedy_final\myproject\myapiapp\views.py changed, reloading.

Watching for file changes with StatReloader

Performing system checks...

You might have noticed version 4.0.2 in the above output.

Method 3

You can also check the Django version using pip. Do you want to know how? Let us move to the code section.

pip freeze

This will show you all the versions of all the modules installed in yThis will show you all the versions of all the modules installed in your system.

The output somewhat looks like:

Django==3.2.9

djangorestframework==3.13.1

Method 4

The next method in this tutorial is by using python IDLE. So, quickly open your IDLE and import django then write django.get_version().

```
import django

django.get_version()
```

It will result in the same output like the above method 1 and 2.

our system.

'3.2.9'

Method 5

The above method can be modified a little bit to know the version of Django. The slight change in syntax is after importing django we need to write django.VERSION but the output varies.

```
import django

django.VERSION
```

The output looks like:

(3, 2, 9, 'final', 0)

So, we have learnt overall 5 methods to know the version of Django installed in your system.

The additional information in this tutorial is about how to install any specific version of Django.

The first technique is to upgrade. You can upgrade Django using the syntax:

pip install --upgrade django

Q-3 Explain what does django-admin.py make messages command is used for?

Ans

django-admin - Utility script for the Django Web framework

django-admin is Django's command-line utility for administrative tasks. This document

outlines all it can do.

In addition, manage.py is automatically created in each Django project. It does the same

thing as django-admin but also sets the DJANGO_SETTINGS_MODULE environment variable so

that it points to your project's settings.py file.

The django-admin script should be on your system path if you installed Django via its

setup.py utility. If it's not on your path, you can find it in site-packages/django/bin

within your Python installation. Consider symlinking it from some place on your path, such

as /usr/local/bin.

For Windows users, who do not have symlinking functionality available, you can copy

django-admin.exe to a location on your existing path or edit the PATH settings (under

Settings - Control Panel - System - Advanced - Environment...) to point to its installed

location.

Generally, when working on a single Django project, it's easier to use manage.py than

django-admin. If you need to switch between multiple Django settings files, use

django-admin with DJANGO_SETTINGS_MODULE or the --settings command line option.

The command-line examples throughout this document use django-admin to be consistent, but

any example can use manage.py or python -m django just as well.

USAGE

$ django-admin <command> [options]

$ manage.py <command> [options]

$ python -m django <command> [options]

command should be one of the commands listed in this document. options, which is

optional, should be zero or more of the options available for the given command.

Getting runtime help

django-admin help

Run django-admin help to display usage information and a list of the commands provided by

each application.

Run django-admin help --commands to display a list of all available commands.

Run django-admin help <command> to display a description of the given command and a list

of its available options.

App names

Many commands take a list of "app names." An "app name" is the basename of the package

containing your models. For example, if your INSTALLED_APPS contains the string

'mysite.blog', the app name is blog.

Determining the version

django-admin version

Run django-admin version to display the current Django version.

The output follows the schema described in PEP 440:

1.4.dev17026

1.4a1

1.4

Displaying debug output

Use --verbosity to specify the amount of notification and debug information that

django-admin prints to the console.

AVAILABLE COMMANDS

check

django-admin check [app_label [app_label ...]]

Uses the system check framework to inspect the entire Django project for common problems.

By default, all apps will be checked. You can check a subset of apps by providing a list

of app labels as arguments:

django-admin check auth admin myapp

If you do not specify any app, all apps will be checked.

--tag TAGS, -t TAGS

The system check framework performs many different types of checks that are categorized

with tags. You can use these tags to restrict the checks performed to just those in a

particular category. For example, to perform only models and compatibility checks, run:

```
django-admin check --tag models --tag compatibility
```

--list-tags

Lists all available tags.

--deploy

Activates some additional checks that are only relevant in a deployment setting.

You can use this option in your local development environment, but since your local

development settings module may not have many of your production settings, you will

probably want to point the check command at a different settings module, either by setting

the DJANGO_SETTINGS_MODULE environment variable, or by passing the --settings option:

```
django-admin check --deploy --
settings=production_settings
```

Or you could run it directly on a production or staging deployment to verify that the

correct settings are in use (omitting --settings). You could even make it part of your

integration test suite.

--fail-level {CRITICAL,ERROR,WARNING,INFO,DEBUG}

Specifies the message level that will cause the command to exit with a non-zero status.

Default is ERROR.

compilemessages

django-admin compilemessages

Compiles .po files created by makemessages to .mo files for use with the built-in gettext

support. See /topics/i18n/index.

--locale LOCALE, -l LOCALE

Specifies the locale(s) to process. If not provided, all locales are processed.

--exclude EXCLUDE, -x EXCLUDE

Specifies the locale(s) to exclude from processing. If not provided, no locales are

excluded.

--use-fuzzy, -f

Includes fuzzy translations into compiled files.

Example usage:

```
django-admin compilemessages --locale=pt_BR
django-admin compilemessages --locale=pt_BR --locale=fr -f
django-admin compilemessages -l pt_BR
```

django-admin compilemessages -l pt_BR -l fr --use-fuzzy

django-admin compilemessages --exclude=pt_BR

django-admin compilemessages --exclude=pt_BR --exclude=fr

django-admin compilemessages -x pt_BR

django-admin compilemessages -x pt_BR -x fr

createcachetable

django-admin createcachetable

Creates the cache tables for use with the database cache backend using the information

from your settings file. See /topics/cache for more information.

--database DATABASE

Specifies the database in which the cache table(s) will be created. Defaults to default.

--dry-run

Prints the SQL that would be run without actually running it, so you can customize it or

use the migrations framework.

dbshell

django-admin dbshell

Runs the command-line client for the database engine specified in your ENGINE setting,

with the connection parameters specified in your USER, PASSWORD, etc., settings.

- For PostgreSQL, this runs the psql command-line client.
- For MySQL, this runs the mysql command-line client.
- For SQLite, this runs the sqlite3 command-line client.
- For Oracle, this runs the sqlplus command-line client.

This command assumes the programs are on your PATH so that a simple call to the program

name (psql, mysql, sqlite3, sqlplus) will find the program in the right place. There's no

way to specify the location of the program manually.

--database DATABASE

Specifies the database onto which to open a shell. Defaults to default.

diffsettings

django-admin diffsettings

Displays differences between the current settings file and Django's default settings (or

another settings file specified by --default).

Settings that don't appear in the defaults are followed by "###". For example, the default

settings don't define ROOT_URLCONF, so ROOT_URLCONF is followed by "###" in the output of

diffsettings.

--all

Displays all settings, even if they have Django's default value. Such settings are

prefixed by "###".

--default MODULE

The settings module to compare the current settings against. Leave empty to compare

against Django's default settings.

--output {hash,unified}

Specifies the output format. Available values are hash and unified. hash is the default

mode that displays the output that's described above. unified displays the output similar

to diff -u. Default settings are prefixed with a minus sign, followed by the changed

setting prefixed with a plus sign.

dumpdata

django-admin dumpdata [app_label[.ModelName] [app_label[.ModelName] ...]]

Outputs to standard output all data in the database associated with the named

application(s).

If no application name is provided, all installed applications will be dumped.

The output of dumpdata can be used as input for loaddata.

Note that dumpdata uses the default manager on the model for selecting the records to

dump. If you're using a custom manager as the default manager and it filters some of the

available records, not all of the objects will be dumped.

--all, -a

Uses Django's base manager, dumping records which might otherwise be filtered or modified

by a custom manager.

--format FORMAT

Specifies the serialization format of the output. Defaults to JSON. Supported formats are

listed in serialization-formats.

--indent INDENT

Specifies the number of indentation spaces to use in the output. Defaults to None which

displays all data on single line.

--exclude EXCLUDE, -e EXCLUDE

Prevents specific applications or models (specified in the form of app_label.ModelName)

from being dumped. If you specify a model name, the output will be restricted to that

model, rather than the entire application. You can also mix application names and model

names.

If you want to exclude multiple applications, pass --exclude more than once:

```
django-admin dumpdata --exclude=auth --exclude=contenttypes
```

--database DATABASE

Specifies the database from which data will be dumped. Defaults to default.

--natural-foreign

Uses the natural_key() model method to serialize any foreign key and many-to-many relationship to objects of the type that defines the method. If you're dumping contrib.auth Permission objects or contrib.contenttypes ContentType objects, you should probably use this flag. See the natural keys documentation for more details on this and the next option.

--natural-primary

Omits the primary key in the serialized data of this object since it can be calculated

during deserialization.

--pks PRIMARY_KEYS

Outputs only the objects specified by a comma separated list of primary keys. This is

only available when dumping one model. By default, all the records of the model are

output.

--output OUTPUT, -o OUTPUT

Specifies a file to write the serialized data to. By default, the data goes to standard

output.

When this option is set and --verbosity is greater than 0 (the default), a progress bar is

shown in the terminal.

flush

django-admin flush

Removes all data from the database and re-executes any post-synchronization handlers. The

table of which migrations have been applied is not cleared.

If you would rather start from an empty database and re-run all migrations, you should

drop and recreate the database and then run migrate instead.

--noinput, --no-input

Suppresses all user prompts.

--database DATABASE

Specifies the database to flush. Defaults to default.

inspectdb

django-admin inspectdb [table [table ...]]

Introspects the database tables in the database pointed-to by the NAME setting and outputs

a Django model module (a models.py file) to standard output.

You may choose what tables or views to inspect by passing their names as arguments. If no

arguments are provided, models are created for views only if the --include-views option is

used. Models for partition tables are created on PostgreSQL if the --include-partitions

option is used.

Use this if you have a legacy database with which you'd like to use Django. The script

will inspect the database and create a model for each table within it.

As you might expect, the created models will have an attribute for every field in the

table. Note that inspectdb has a few special cases in its field-name output:

• If inspectdb cannot map a column's type to a model field type, it'll use TextField and

will insert the Python comment 'This field type is a guess.' next to the field in the

generated model. The recognized fields may depend on apps listed in INSTALLED_APPS. For

example, django.contrib.postgres adds recognition for several PostgreSQL-specific field

types.

• If the database column name is a Python reserved word (such as 'pass', 'class' or

'for'), inspectdb will append '_field' to the attribute name. For example, if a table

has a column 'for', the generated model will have a field 'for_field', with the

db_column attribute set to 'for'. inspectdb will insert the Python comment 'Field

renamed because it was a Python reserved word.' next to the field.

This feature is meant as a shortcut, not as definitive model generation. After you run it,

you'll want to look over the generated models yourself to make customizations. In

particular, you'll need to rearrange models' order, so that models that refer to other

models are ordered properly.

Django doesn't create database defaults when a default is specified on a model field.

Similarly, database defaults aren't translated to model field defaults or detected in any

fashion by inspectdb.

By default, inspectdb creates unmanaged models. That is, managed = False in the model's

Meta class tells Django not to manage each table's creation, modification, and deletion.

If you do want to allow Django to manage the table's lifecycle, you'll need to change the

managed option to True (or simply remove it because True is its default value).

Database-specific notes

Oracle

• Models are created for materialized views if --include-views is used.

PostgreSQL

- Models are created for foreign tables.

- Models are created for materialized views if --include-views is used.

- Models are created for partition tables if --include-partitions is used.

Support for foreign tables and materialized views was added.

--database DATABASE

Specifies the database to introspect. Defaults to default.

--include-partitions

If this option is provided, models are also created for partitions.

Only support for PostgreSQL is implemented.

--include-views

If this option is provided, models are also created for database views.

loaddata

django-admin loaddata fixture [fixture ...]

Searches for and loads the contents of the named fixture into the database.

--database DATABASE

Specifies the database into which the data will be loaded. Defaults to default.

--ignorenonexistent, -i

Ignores fields and models that may have been removed since the fixture was originally

generated.

--app APP_LABEL

Specifies a single app to look for fixtures in rather than looking in all apps.

--format FORMAT

Specifies the serialization format (e.g., json or xml) for fixtures read from stdin.

--exclude EXCLUDE, -e EXCLUDE

Excludes loading the fixtures from the given applications and/or models (in the form of

app_label or app_label.ModelName). Use the option multiple times to exclude more than one

app or model.

What's a fixture ?

A fixture is a collection of files that contain the serialized contents of the database.

Each fixture has a unique name, and the files that comprise the fixture can be distributed

over multiple directories, in multiple applications.

Django will search in three locations for fixtures:

1. In the fixtures directory of every installed application

2. In any directory named in the FIXTURE_DIRS setting

3. In the literal path named by the fixture

Django will load any and all fixtures it finds in these locations that match the provided

fixture names.

If the named fixture has a file extension, only fixtures of that type will be loaded. For

example:

```
django-admin loaddata mydata.json
```

would only load JSON fixtures called mydata. The fixture extension must correspond to the

registered name of a serializer (e.g., json or xml).

If you omit the extensions, Django will search all available fixture types for a matching

fixture. For example:

```
django-admin loaddata mydata
```

would look for any fixture of any fixture type called mydata. If a fixture directory

contained mydata.json, that fixture would be loaded as a JSON fixture.

The fixtures that are named can include directory components. These directories will be

included in the search path. For example:

```
django-admin loaddata foo/bar/mydata.json
```

would search <app_label>/fixtures/foo/bar/mydata.json for each installed application,

<dirname>/foo/bar/mydata.json for each directory in FIXTURE_DIRS, and the literal path

foo/bar/mydata.json.

When fixture files are processed, the data is saved to the database as is. Model defined

save() methods are not called, and any pre_save or post_save signals will be called with

raw=True since the instance only contains attributes that are local to the model. You may,

for example, want to disable handlers that access related fields that aren't present

during fixture loading and would otherwise raise an exception:

```
from django.db.models.signals import post_save
from .models import MyModel

def my_handler(**kwargs):
    # disable the handler during fixture loading
    if kwargs['raw']:
        return
    ...

post_save.connect(my_handler, sender=MyModel)
```

You could also write a simple decorator to encapsulate this logic:

```
from functools import wraps


def disable_for_loaddata(signal_handler):
    """
    Decorator that turns off signal handlers when loading fixture data.
    """
    @wraps(signal_handler)
    def wrapper(*args, **kwargs):
        if kwargs['raw']:
            return
        signal_handler(*args, **kwargs)
    return wrapper


@disable_for_loaddata
def my_handler(**kwargs):
    ...
```

Just be aware that this logic will disable the signals whenever fixtures are deserialized,

not just during loaddata.

Note that the order in which fixture files are processed is undefined. However, all

fixture data is installed as a single transaction, so data in one fixture can reference

data in another fixture. If the database backend supports row-level constraints, these

constraints will be checked at the end of the transaction.

The dumpdata command can be used to generate input for loaddata.

Compressed fixtures

Fixtures may be compressed in zip, gz, or bz2 format. For example:

```
django-admin loaddata mydata.json
```

would look for any of mydata.json, mydata.json.zip, mydata.json.gz, or mydata.json.bz2.

The first file contained within a zip-compressed archive is used.

Note that if two fixtures with the same name but different fixture type are discovered

(for example, if mydata.json and mydata.xml.gz were found in the same fixture directory),

fixture installation will be aborted, and any data installed in the call to loaddata will

be removed from the database.

MySQL with MyISAM and fixtures

The MyISAM storage engine of MySQL doesn't support transactions or constraints,

so if you use MyISAM, you won't get validation of fixture data, or a rollback if

multiple transaction files are found.

Database-specific fixtures

If you're in a multi-database setup, you might have fixture data that you want to load

onto one database, but not onto another. In this situation, you can add a database

identifier into the names of your fixtures.

For example, if your DATABASES setting has a 'master' database defined, name the fixture

mydata.master.json or mydata.master.json.gz and the fixture will only be loaded when you

specify you want to load data into the master database.

Loading fixtures from stdin

You can use a dash as the fixture name to load input from sys.stdin. For example:

```
django-admin loaddata --format=json -
```

When reading from stdin, the --format option is required to specify the serialization

format of the input (e.g., json or xml).

Loading from stdin is useful with standard input and output redirections. For example:

```
django-admin dumpdata --format=json --database=test app_label.ModelName | django-admin loaddata --format=json --database=prod -
```

makemessages

django-admin makemessages

Runs over the entire source tree of the current directory and pulls out all strings marked

for translation. It creates (or updates) a message file in the conf/locale (in the Django

tree) or locale (for project and application) directory. After making changes to the

messages files you need to compile them with compilemessages for use with the builtin

gettext support. See the i18n documentation for details.

This command doesn't require configured settings. However, when settings aren't

configured, the command can't ignore the MEDIA_ROOT and STATIC_ROOT directories or include

LOCALE_PATHS.

--all, -a

Updates the message files for all available languages.

--extension EXTENSIONS, -e EXTENSIONS

Specifies a list of file extensions to examine (default: html, txt, py or js if --domain

is js).

Example usage:

```
django-admin makemessages --locale=de --extension xhtml
```

Separate multiple extensions with commas or use -e or --extension multiple times:

```
django-admin makemessages --locale=de --extension=html,txt --extension xml
```

--locale LOCALE, -l LOCALE

Specifies the locale(s) to process.

--exclude EXCLUDE, -x EXCLUDE

Specifies the locale(s) to exclude from processing. If not provided, no locales are

excluded.

Example usage:

```
django-admin makemessages --locale=pt_BR
django-admin makemessages --locale=pt_BR --locale=fr
django-admin makemessages -l pt_BR
django-admin makemessages -l pt_BR -l fr
django-admin makemessages --exclude=pt_BR
django-admin makemessages --exclude=pt_BR --exclude=fr
django-admin makemessages -x pt_BR
django-admin makemessages -x pt_BR -x fr
```

--domain DOMAIN, -d DOMAIN

Specifies the domain of the messages files. Supported options are:

- django for all *.py, *.html and *.txt files (default)

Q-4 What is Django URLs?make program to create django urls?

n Django, views are Python functions which take a URL request as parameter and return an HTTP response or throw an exception like 404. Each view needs to be mapped to a corresponding URL pattern. This is done via a Python module called URLConf(URL configuration)

Let the project name be myProject. The Python module to be used as URLConf is the value of ROOT_URLCONF in myProject/settings.py. By default this is set to 'myProject.urls'. Every URLConf module must contain a variable urlpatterns which is a set of URL patterns to be matched against the requested URL. These patterns will be checked in sequence, until the first match is found. Then the view corresponding to the first match is invoked. If no URL pattern matches, Django invokes an appropriate error handling view.

Including other URLConf modules
It is a good practice to have a URLConf module for every app in Django. This module needs to be included in the root URLConf module as follows:

```
from django.contrib import admin
from django.urls import path, include


urlpatterns = [
```

```
    path('admin/', admin.site.urls),
    path('', include('books.urls')),
]
```

This tells Django to search for URL patterns in the file books/urls.py.

URL patterns

Here’s a sample code for books/urls.py:

```
from django.urls import path
from . import views

urlpatterns = [
    path('books/<int:pk>/', views.book_detail),
    path('books/<str:genre>/', views.books_by_genre),
    path('books/', views.book_index),
]
```

For example,

A URL request to /books/crime/ will match with the second URL pattern. As a result, Django will call the function views.books_by_genre(request, genre = "crime").

Similarly a URL request /books/25/ will match the first URL pattern and Django will call the function views.book_detail(request, pk =25).

Here, int and str are path convertors and capture an integer and string value respectively.

Path convertors:
The following path convertor types are available in Django

int – Matches zero or any positive integer.

str – Matches any non-empty string, excluding the path separator('/').

slug – Matches any slug string, i.e. a string consisting of alphabets, digits, hyphen and under score.

path – Matches any non-empty string including the path separator('/')

uuid – Matches a UUID(universal unique identifier).

Q-5 What is a QuerySet?Write program to create a new Post object in
database

Ans queryset is simply a list of objects from the Django Models. When using the Django ORM creating a new row in the table is simply like creating a new object of Model class. Django ORM maps these Model objects to Relational database query.

Any SQL query can be written easily as Django queryset. You can separately test the query for Create, Filter, Update, Order, etc. in Django Shell or in the views.py.

The syntax for a queryset is always like:

ModelName.objects.method_name(arguments)

For example, to get all data from the model Blog under app named myapp, start a Python shell and run the following:

python manage.py shell

```
>>> from myapp.models import Blog
>>> queryset = Blog.objects.all()
QuerySet [...]
```

Methods In Queryset API

There are several methods under the Django Queryset API. Some methods return another queryset object and thus can be followed by another methods.

queryset_object = Model.objects.all().filter()

These queryset objects needs to be accessed iteratively in the template pages with the following syntax:

{ % for i in queryset_object % }

{ {i.fieldname} }

...

{ % endfor % }

On the other hand, some queryset methods do not return a new queryset or simply return data items. For example:

ueryset_object = Model.objects.get(id=1)

The result of a get query can be accessed in the template pages without using any loop, or simply with the below syntax.

{ {queryset_object.fieldname} }

Q-6 Mention what command line can be used to load data into Django?

Ans. Just before we get started, let's take a moment to familiarize with Django's command line interface. You are probably already familiar with commands
like startproject, runserver or collectstatic. To see a complete list of commands you can run the command below:

python manage.py help

Output:

Type 'manage.py help <subcommand>' for help on a specific subcommand.

Available subcommands:

[auth]

changepassword

createsuperuser

[contenttypes]

remove_stale_contenttypes

[django]

check

compilemessages

createcachetable

dbshell

diffsettings

dumpdata

flush

    inspectdb
    loaddata
    makemessages
    makemigrations
    migrate
    sendtestemail
    shell
    showmigrations
    sqlflush
    sqlmigrate
    sqlsequencereset
    squashmigrations
    startapp
    startproject
    test
    testserver

[sessions]
    clearsessions

[staticfiles]
    collectstatic

findstatic

runserver

We can create our own commands for our apps and include them in the list by creating
a management/commands directory inside an app directory, like below:

```
mysite/                          <-- project directory
 |-- core/                       <-- app directory
 |    |-- management/
 |    |    +-- commands/
 |    |         +-- my_custom_command.py  <-- module where command is going to live
 |    |-- migrations/
 |    |    +-- __init__.py
 |    |-- __init__.py
 |    |-- admin.py
 |    |-- apps.py
 |    |-- models.py
 |    |-- tests.py
 |    +-- views.py
 |-- mysite/
 |    |-- __init__.py
 |    |-- settings.py
```

```
|    |-- urls.py
|    |-- wsgi.py
+-- manage.py
```

The name of the command file will be used to invoke using the command line utility. For example, if our command was called my_custom_command.py, then we will be able to execute it via:

python manage.py my_custom_command

Let’s explore next our first example.

Basic Example

Below, a basic example of what the custom command should look like:

management/commands/what_time_is_it.py

```
from django.core.management.base import BaseCommand
from django.utils import timezone

class Command(BaseCommand):
    help = 'Displays current time'

    def handle(self, *args, **kwargs):
        time = timezone.now().strftime('%X')
        self.stdout.write("It's now %s" % time)
```

Basically a Django management command is composed by a class named Command which inherits from BaseCommand. The command code should be defined inside the handle() method.

See how we named our module what_time_is_it.py. This command can be executed as:

python manage.py what_time_is_it

Output:

It's now 18:35:31

You may be asking yourself, how is that different from a regular Python script, or what's the benefit of it. Well, the main advantage is that all Django machinery is loaded and ready to be used. That means you can import models, execute queries to the database using Django's ORM and interact with all your project's resources.

Handling Arguments

Django make use of the argparse, which is part of Python's standard library. To handle arguments in our custom command we should define a method named add_arguments.

Positional Arguments

The next example is a command that create random user instances. It takes a mandatory argument named total, which will define the number of users that will be created by the command.

management/commands/create_users.py

```
from django.contrib.auth.models import User
from django.core.management.base import BaseCommand
from django.utils.crypto import get_random_string


class Command(BaseCommand):
    help = 'Create random users'


    def add_arguments(self, parser):
        parser.add_argument('total', type=int, help='Indicates
the number of users to be created')


    def handle(self, *args, **kwargs):
        total = kwargs['total']
        for i in range(total):


User.objects.create_user(username=get_random_string(),
email='', password='123')
```

Here is how one would use it:

python manage.py create_users 10

Optional Arguments

The optional (and named) arguments can be passed in any order. In the example below you will find the definition of an

argument named “prefix”, which will be used to compose the username field:

management/commands/create_users.py

```
from django.contrib.auth.models import User
from django.core.management.base import BaseCommand
from django.utils.crypto import get_random_string


class Command(BaseCommand):
    help = 'Create random users'


    def add_arguments(self, parser):
        parser.add_argument('total', type=int, help='Indicates the number of users to be created')


        # Optional argument
        parser.add_argument('-p', '--prefix', type=str, help='Define a username prefix', )


    def handle(self, *args, **kwargs):
        total = kwargs['total']
        prefix = kwargs['prefix']


        for i in range(total):
```

```
        if prefix:
            username = '{prefix}_{random_string}'.format(prefix=prefix, random_string=get_random_string())
        else:
            username = get_random_string()
        User.objects.create_user(username=username, email='', password='123')
```

Usage:

```
python manage.py create_users 10 --prefix custom_user
```

or

```
python manage.py create_users 10 -p custom_user
```

If the prefix is used, the username field will be created as custom_user_oYwoxtt4vNHR. If not prefix, it will be created simply as oYwoxtt4vNHR – a random string.

Flag Arguments

Another type of optional arguments are flags, which are used to handle boolean values. Let's say we want to add an --admin flag, to instruct our command to create a super user or to create a regular user if the flag is not present.

management/commands/create_users.py

```
from django.contrib.auth.models import User
from django.core.management.base import BaseCommand
from django.utils.crypto import get_random_string
```

```
class Command(BaseCommand):
    help = 'Create random users'


    def add_arguments(self, parser):
        parser.add_argument('total', type=int, help='Indicates
the number of users to be created')
        parser.add_argument('-p', '--prefix', type=str,
help='Define a username prefix')
        parser.add_argument('-a', '--admin', action='store_true',
help='Create an admin account')


    def handle(self, *args, **kwargs):
        total = kwargs['total']
        prefix = kwargs['prefix']
        admin = kwargs['admin']


        for i in range(total):
            if prefix:
                username =
'{prefix}_{random_string}'.format(prefix=prefix,
random_string=get_random_string())
            else:
```

```
            username = get_random_string()


        if admin:


User.objects.create_superuser(username=username,
email='', password='123')
        else:
            User.objects.create_user(username=username,
email='', password='123')
```

Usage:

```
python manage.py create_users 2 --admin
```

Or

```
python manage.py create_users 2 -a
```