Django's ORM (Object-Relational mapping):

Defination and purpose:

- What is an ORM?

→ ORM stands for 'Object Relational mapping'. Its a technique in software that allows you to work with relational database (e.g. MySQL, SQLite) using the object-oriented paradigm of your programming language, in this case python.

- Why does Django have an ORM?

- Django's ORM abstracts away much of the sql you would normally write. Instead of writing sql queries directly you use python classes to represent database model, and django automatically translate your python code into the necessary SQL.

- Where does it work/ How does it functions?

→ The Django's ORM works within django projects. Whenever you define a model class in 'models.py', you are effectively defining a table schema.

Ex:

```
From django.db import models

Class Book (models.Model):
    title = models.CharField (max_length = 200)
    author = models.charfield (max_length = 100)
    published_date = models.DateField()
    ...
```

Behind the scenes, Django creates database tables (columns, constraints, etc.) for those classes when you run migrations.
When you fetch or create data, ORM converts that to SQL.
Ex:
```
Python
Book.objects.all()
Book.objects.create(title="Django_unleased",
                    author2="Andrew Pinkham")
```
- The ORM also handles relationships through special field types (e.g. 'ForeignKey, ~~for~~ ManytoManyField).

Benifits of Django's ORM:

1) Less SQL: You do not have to write raw sql statements for most common operations.

2) Database Portability: If you swithch from one database to another, your code is more easily portable.

3) Validation and secureity: The ORM helps prevent common issues with SQL injection by escaping parameters automatically.

4) Easier to maintain: your model class definations are clean and can be version-controlled well,

3) Django calls the appropriate view if the request is allowed through.
4) After the view returns a response, that response goes back through middleware again.

- Common Examples of Django middleware:

- Authentication Middleware: Associates users with requests.
- Session Middleware: Manages user sections across requests.
- CSRF middleware: Adds protection against Cross-site Request Forgery.
- Security Middleware: Adds common security headers.

The concept is similar across frameworks, but the implementation details ~~way~~ vary, essentially. Its about looking into the flow of data.

3) Migration and 'migrate' in Django:

Overview:

Django has a builtin system to handle changes to your database schema over time. This system consist of two main steps/commands:

1) 'makemigration' - Creates migration files based on changes you've made to your model.

2) 'Migrate' - Applies those migration files to the actual database.

What is migration ?

- A migration is a python file that describes the changes you want to make your database schema like creating a new table, adding a new column, or changing a field's type.

- Location of migration files : By default, Django stores these files in a folder called 'migrations' within each app.

A typical migration file might look like this :

```python
# books/migrations/0001_initial.py

from django.db import migrations, models

class Migration(migrations.Migration):
    initial = True

    dependencies = [
        ]

    Operations = [
        migrations.CreateModel(
        name='Book',
        Fields = [
        ('id', models.AutoField(primary_key=True),
        ('title', models.CharField(max_length=200)),
        ('auther' models.CharField( max_length=100)),
        ('published-date', Models.DateField()),
        ],
```

```
),
]
'''
```

This tells django to create a new table named 'Book' with the defined fields

What is the migrate process?

- The 'migrate' command reads the migration files and applies them to your database. This means:

1) If a new table is specified, it creates it.
2) If a field is added, it modifies the table structure to add that column.
3) If a column is renamed or removed, it performs those operations, etc..

Difference Between Migration and 'Migrate':

"Migration": Usually refers to the defination of the changes you want in your database. Its the plan or blueprint

- You create or update this plan by running "python manage.py makemigrations".
- This command compares your current model definations with the previously recorded state in your migration then writes out new migration files if there are differences.

'Migrate': Reffers to actualy applying those migration files to the database.
- You do this by runnig 'python manage.py migrate".
- This command looks at all the migration files and executes the pending changes against the database.

Why Two steps:

- By having a seperate step to create migration files, Django gets you:
① Generate migration files in your version controll.
② Inspect or edit them before they are applied to your database.
③ Roll back or roll forward to specific version

- The 'migrate' step then safely applies those changes This seperation provides better controll and transperancy about what database changes are being made.

Summary:

1) Django's ORM:
- An abstraction layer to interact with database through python classes and methods.
- Helps avoid raw sql and support multiple database easily.

2) Middleware:

- Software that intercepts request / response to add cross-cutting functionality.
- In Django, middleware are classes that process requests / responces as they travel in and out of django's view layer.

③ Migrations vs Migrate:

- Migrations are python files that describe changes to the database schema. You generate or update them with 'python manage.py makemigrations'.

- 'Migrate' is the command that applies these migrations files to the actual database.

With these cocept in mind, you can see how Django supports a clean, maintainable workflow - from defining your datamodel, to managing schema changes, to hooking into HTTP requests/responces with middleware.

1) Middlewar (Simple explaination)

- Middleware is like a security checkpoint in a django application. It sits between the users request and the response sent by the server. It can modify, block, or add extra features to request and response.

Example use of middleware:

- Checking if a user is logged in before allowing acess.
- protecting against security threats like CSRF attacks.
- Recording logs of all requests and responces for debugging.
- Adding extra data to requests or responces.

In a simple words, middleware act as a filter that controls what happens to the request before it reaches the main logic and what happens to the response before ~~that~~ it is sent back to the user.

2. ORM :

ORM is a way to interact with database using python objects instead of writing sql queries. Instead of writing commands like 'select * from users', you just ask django's ORM to get all users with a simple python command.

Why ORM is useful.?

- You dont have to write SQL.
- It works with different databases like MySQL, Postgre SQL, and sqlite without changing your code.
- It helps prevent security issues like SQL, injections.

- In simple words, ORM is like a translator that helps Django talk to the database using python instead of sql.

HTTP:

HTTP is the language that web browsers and servers use to communicate. Whenever you open a website, your browser sends an HTTP request, and the server responds with the HTTP response.

How HTTP works in django:

- When a user visit a webpage, their browser sends a request to the django server.
- Django processes the request and decides what data to sent back.
- The server then sends an HTTP response, which can be a webpage, data, or an error message.

Common HTTP methods:

- GET → Used to request data from the server
- POST → Used to send data to the server
- PUT/PATCH → Used to update existing data.
- DELETE → Used to delete data.

- In simple words, HTTP is like messaging system where browsers and servers exchange requests and responses to show webpage, send forms or update data.