1. Validation in Models:

1.1.Built-in Field Validators:

Django provides build-in field validators allowing to validate the data entered in the model fields,
ensure that the data meets specific requirements before saving it

MaxValueValidator
MinValueValidator
MaxLengthValidator
MinLengthValidator
RegexValidator

accept 2 arguments:
- limit_value: a required, first positional argument that specifies the limit value
- message: a default argument, message=None is being passed by default

raise a ValidationError(if the field value does noe meet the requirements)

1.2.Custom Validators:

Create custom validators when you need to implement a custom validation logic.

Steps: Define a function that:
- takes the field's value
- applies some validation logic
- raises a ValidationError(if the value does not meet the requirements)

```
def validate_even(value):
      if value % 2 != 0:
            raise ValidationError('Value must be an even number!')

class MyModel(models.Model):
      number = models.IntegerField(validators=[validate_even])
```

2. Meta Options and Meta Inheritance:

- Meta class allows you to provide additional information about the model.

Used to specify model-level options like:
- DB table name(db_table): overrides the default DB table name for the model
- ordering: defines a default order when a collection of model objects is obtained
- unique constraints(unique_together): defines a set of field names that their values
  together must be unique
- abstract: if True, indicates that the model will be an Abstract Base Class
- proxy: if True, indicates that the model will be a proxy model
- verbose_name: defines a human-readable name for the object(singular)
- verbose_name_plural: defines a plural name for the object. By default, Django uses the model
  or verbose name(if given) + 's'

2.1.DB Table Name

2.2.Default order, unique constraint

2.3.Meta Inheritance:
- Meta inheritance refers to the ability to inherit Meta options from a

```
parent abstract model to its child models.
            - When an abstract model(parent) is defined in Django, it can include
an inner Meta class, where you specify various
                  options related to the behaviour and configuration of that model
            - It can be subclassed so that the child will inherit the same Meta
options if it does not define its own Meta class
            - When you create a child model that inherits from an abstract model,
it can also define its own Meta class. Own Meta
                  class completely overrides the parent's one unless it extends it
by subclassing
            - If the child model does not have its own Meta class Django will look
for the Meta class in the parent abstract model
                  and inherit its options
            example for extending Meta options:
            class ChildModel(BaseModel):
                  description = model.TextField()

                  class Meta(BaseModel.Meta):
                        verbose_name_plural = 'Child Models'

3.Indexing:
      - In Django models, indexing is used to optimize DB queries for specific
fields
      - By adding an index to a field, you can speed up search operations on that
field
      - DB uses indexes to locate rows much faster, significantly reducing the time
to retrieve data
      - By default, the DB creates an index for the PK
      - It is possible to specify additional indexes manually on other fields by
using the db_index attribute,
            by using Meta class option indexes

4.Django Model Mixins:
      -     Mixins are a way to extend the functionality of Django models:
            - by creating a reusable piece of code
            - that can be mixed into multiple models
      - A Mixin is essentially a Python class that:
            - contains additional fields, methods, or behaviour
            - can be combined with other Django models
      - By using Mixins, you can avoid code duplication and keep your models clean
and organized
      - Model Mixins are abstract classes that can be added as a parent class of a
model
      - Python supports multiple inheritance - you can list any number of parent
classes for a model
      - Mixins have to be simpe and easy composable
      - Smaller mixins are better - if a mixin becomes large and violates the
single responsibility principle,
            consider refactoring it to smaller classes
```