Perfect! You want a **complete A to Z guide for Django models**. Let's go step by step, covering everything you need for **creating, managing, and using Django models**. I'll make it structured for clarity.

🔹 Django Models A to Z

---

1. What is a Django Model?

- A **model** is a Python class that represents a database table.
- Each **attribute** of the model corresponds to a **database field**.
- Defined in `models.py` of your Django app.

```
from django.db import models

class Student(models.Model):
    name = models.CharField(max_length=100)
    age = models.IntegerField()
    grade = models.CharField(max_length=10)
```

---

2. Field Types

Django provides many field types:

Field Type	Description	Example
`CharField`	Short text, max_length required	`name = models.CharField(max_length=50)`
`TextField`	Long text	`bio = models.TextField()`
`IntegerField`	Integer	`age = models.IntegerField()`
`FloatField`	Floating point number	`price = models.FloatField()`
`BooleanField`	True/False	`is_active = models.BooleanField(default=True)`
`DateField`	Date only	`dob = models.DateField()`

Field Type	Description	Example
`DateTimeField`	Date + Time	`created_at = models.DateTimeField(auto_now_add=True)`
`EmailField`	Email address	`email = models.EmailField()`
`URLField`	URL	`website = models.URLField()`
`ForeignKey`	Many-to-One relation	`author = models.ForeignKey(Author, on_delete=models.CASCADE)`
`OneToOneField`	One-to-One relation	`profile = models.OneToOneField(User, on_delete=models.CASCADE)`
`ManyToManyField`	Many-to-Many relation	`tags = models.ManyToManyField(Tag)`

---

3. Field Options

- `null=True` → database can store NULL.
- `blank=True` → field can be empty in forms.
- `default=value` → default value.
- `choices=[('A','A Grade'), ('B','B Grade')]` → limit to choices.
- `unique=True` → value must be unique.

```
grade = models.CharField(max_length=2, choices=[('A','A Grade'),('B','B Grade')], default='B')
```

---

4. Model Meta Options

Configure model behavior with inner `Meta` class:

```
class Student(models.Model):
    name = models.CharField(max_length=50)
    age = models.IntegerField()

    class Meta:
        ordering = ['name']            # default ordering
        verbose_name = 'Student'
        verbose_name_plural = 'Students'
```

Common `Meta` options:

- `db_table` → custom table name

- `ordering` → default ordering
- `unique_together` → composite unique constraints
- `verbose_name` / `verbose_name_plural` → human-readable names

---

5. Model Methods

- You can define methods inside your model:

```
class Student(models.Model):
    name = models.CharField(max_length=50)
    age = models.IntegerField()

    def __str__(self):
        return self.name

    def is_adult(self):
        return self.age >= 18
```

- `__str__()` → string representation
- Custom methods → custom logic for the object

আপনি চান যে **age < 18 হলে ডেটা ডাটাবেসে সেভ না হয়** — অর্থাৎ বয়স ১৮ এর নিচে হলে validation error দেখাবে।

এটির জন্য Django-তে দুটি জনপ্রিয় পদ্ধতি আছে:

✅ পদ্ধতি–১: Validators ব্যবহার করা (সবচেয়ে সহজ ও সুন্দর উপায়)

`IntegerField`-এ validator যোগ করে দিন:

```
from django.db import models
from django.core.validators import MinValueValidator

class Students(models.Model):
    name = models.CharField(max_length=100)
    age = models.IntegerField(validators=[MinValueValidator(18)])

    def __str__(self):
        return self.name
```

✔️ এখন age < 18 হলে ডেটা সেভ হবে না

Django shell এ চেষ্টা করলে:

```
Students.objects.create(name="Rafi", age=10)
```

Output:

```
ValidationError: {'age': ['Ensure this value is greater than or equal to
18.']}
```

✅ পদ্ধতি–২: clean() মেথড ব্যবহার করা (custom logic লাগলে)

```
from django.core.exceptions import ValidationError

class Students(models.Model):
    name = models.CharField(max_length=100)
    age = models.IntegerField()

    def clean(self):
        if self.age < 18:
            raise ValidationError("Age must be 18 or above!")

    def __str__(self):
        return self.name
```

✔️ সেভ করার সময় validation চলবে:

```
s = Students(name='Ali', age=12)
s.full_clean()   # <-- validation
s.save()         # invalid হলে save হবে না
```

🚀 কোনটি ব্যবহার করবেন?

পদ্ধতি	কখন ব্যবহার করবেন
Validators	শুধু মিনিমাম / ম্যাক্সিমাম ভ্যালু লাগলে
clean()	জটিল validation লাগলে (একাধিক ফিল্ড মিলিয়ে, কাস্টম লজিক ইত্যাদি)

You can validate any field in Django using 3 main approaches:

✅ 1. Field Validators (Simple & Recommended)

Django allows built-in or custom validators directly in a field.

✔️ Example: Validate length, minimum value, etc.

```
from django.db import models
from django.core.validators import MinValueValidator, MaxLengthValidator

class Student(models.Model):
    name = models.CharField(
        max_length=100,
        validators=[MaxLengthValidator(100)]
    )
    age = models.IntegerField(
        validators=[MinValueValidator(18)]
    )
```

➤ When to use?

- Minimum / maximum age
- Max length
- Regex validation
- Simple one-field checks

✅ 2. Custom Validators

If built-in validators are not enough:

✔️ Example: name must not contain numbers

```
from django.core.exceptions import ValidationError

def validate_name(value):
    if any(char.isdigit() for char in value):
        raise ValidationError("Name cannot contain numbers!")
```

```python
class Student(models.Model):
    name = models.CharField(
        max_length=100,
        validators=[validate_name]
    )
    age = models.IntegerField()
```

✅ 3. Model-level Validation (`clean()` method)

This allows **multiple fields together** or **complex logic**.

✔️ Example: Validate based on multiple fields

```python
from django.core.exceptions import ValidationError

class Student(models.Model):
    name = models.CharField(max_length=100)
    age = models.IntegerField()
    marks = models.IntegerField()

    def clean(self):
        if self.age < 18:
            raise ValidationError("Age must be 18 or above!")

        if not (0 <= self.marks <= 100):
            raise ValidationError("Marks must be between 0 and 100")
```

✔️ Validation runs when you call this:

```python
student.full_clean()
student.save()
```

🔥 Bonus: Forms Validation (Django Form / ModelForm)

Field-level validation inside forms

```python
def clean_age(self):
    age = self.cleaned_data['age']
    if age < 18:
```

```
            raise forms.ValidationError("Age must be 18+")
        return age
```

Form-level validation

```
def clean(self):
    cleaned_data = super().clean()
    age = cleaned_data.get("age")
    marks = cleaned_data.get("marks")

    if age and marks and age < 18 and marks > 80:
        raise forms.ValidationError("Under-18 cannot score above 80")
```

⭐ Summary Table

Method	Best For	Where it Runs
Field Validators	Simple rules (min, max, regex)	Model & Forms
Custom Validators	Special one-field logic	Model & Forms
clean() method	Rules needing multiple fields	Model
Forms clean()	Form-based validation	Forms only

Yes — you can use only the `clean()` method inside the model class to validate fields.

This method gives you **full control** over validation logic.

Below is the **correct structure** and **how it works**.

✅ Example: Using `clean()` only (inside model)

```
from django.db import models
from django.core.exceptions import ValidationError

class Student(models.Model):
    name = models.CharField(max_length=100)
    age = models.IntegerField()
    marks = models.IntegerField()
```

```
    def clean(self):
        # Validate age
        if self.age < 18:
            raise ValidationError("Age must be at least 18.")

        # Validate marks
        if not (0 <= self.marks <= 100):
            raise ValidationError("Marks must be between 0 and 100.")

        # Validate name
        if any(char.isdigit() for char in self.name):
            raise ValidationError("Name cannot contain digits.")

    def __str__(self):
        return self.name
```

⚠️ Important: clean() does NOT run automatically during save()

To apply validation, you must call:

```
student.full_clean()
student.save()
```

Or use:

```
student = Student(name="Rahim", age=15, marks=50)
student.full_clean()  # <-- runs clean()
student.save()
```

If age < 18 → you get:

```
ValidationError: ['Age must be at least 18.']
```

⭐ Recommended pattern for models using clean()

```
obj = Student(name="John", age=16, marks=120)
obj.full_clean()   # validate all fields using clean()
obj.save()         # save if valid
```

✔️ Why use clean() method?

- You want validation **only at model level**
- You want to check **multiple fields together**
- You want **custom validation** logic
- Works for: Django Admin, ModelForms, DRF serializers (when full_clean() is triggered)

Short answer:

❌ **না**, প্রতিটি model field-এর জন্য আলাদা custom method বানাতে *হবে না*।
✔️ আপনি চাইলে **একটাই `clean()` method** ব্যবহার করে **সব field** validate করতে পারবেন।

✅ 1. একটিমাত্র `clean()` — সব ফিল্ডের ভ্যালিডেশন

```
def clean(self):
    if self.age < 18:
        raise ValidationError("Age must be at least 18.")

    if any(char.isdigit() for char in self.name):
        raise ValidationError("Name cannot contain numbers.")

    if not (0 <= self.marks <= 100):
        raise ValidationError("Marks must be between 0 and 100.")
```

➡️ এখানে age, name, marks— সব কিছুর ভ্যালিডেশন **এক method-এই** হচ্ছে।

✅ 2. চাইলে আলাদা আলাদা custom validation method বানাতে পারেন

(কিন্তু বাধ্যতামূলক না)

Example:

```
def validate_age(self):
    if self.age < 18:
        raise ValidationError("Age must be at least 18.")

def validate_name(self):
```

```
        if any(char.isdigit() for char in self.name):
            raise ValidationError("Name cannot contain digits.")

    def validate_marks(self):
        if not (0 <= self.marks <= 100):
            raise ValidationError("Marks must be between 0 and 100.")
```

তারপর `clean()` এর ভিতরে এগুলো কল করবেন:

```
def clean(self):
    self.validate_age()
    self.validate_name()
    self.validate_marks()
```

✔️ এতে কোড clean ও readable হয়।
❌ কিন্তু Django আপনাকে এভাবে করতে বাধ্য করে না।

⭐ Conclusion (সবচেয়ে গুরুত্বপূর্ণ)

প্রশ্ন	উত্তর
প্রতিটি ফিল্ডের জন্য আলাদা custom method বানানো লাগবে?	❌ না
একটাই `clean()` এর মধ্যে সব ভ্যালিডেশন করা যাবে?	✔️ হ্যাঁ
চাইলে প্রতিটি ফিল্ডের জন্য আলাদা method লিখে clean() থেকে কল করা যাবে?	✔️ হ্যাঁ

চাইলে আমি আপনার model দেখে পুরো **best practice অনুযায়ী** validation setup করে দিতে পারি। Debug করতে হলে model টা পাঠান।

---

6. Relationships

- **One-to-Many** → `ForeignKey`
- **Many-to-Many** → `ManyToManyField`
- **One-to-One** → `OneToOneField`

```
class Author(models.Model):
    name = models.CharField(max_length=100)

class Book(models.Model):
```

```
    title = models.CharField(max_length=200)
    author = models.ForeignKey(Author, related_name='books',
on_delete=models.CASCADE)
```

- Access reverse relation using `related_name`:

```
author = Author.objects.get(id=1)
books = author.books.all()
```

---

7. Querying Models

- Use **ORM** to query data:

```
# All objects
students = Student.objects.all()

# Filter
adults = Student.objects.filter(age__gte=18)

# Get single object
student = Student.objects.get(id=1)

# Order
students = Student.objects.order_by('-age')
```

- Common lookups: `exact`, `iexact`, `contains`, `icontains`, `gt`, `lt`, `gte`, `lte`, `in`, `startswith`, `istartswith`

---

8. Creating / Updating / Deleting

```
# Create
student = Student.objects.create(name='John', age=20)

# Update
student.age = 21
student.save()

# Delete
student.delete()
```

---

9. Model Managers

- Default manager → `objects`
- Can create **custom managers** for extra methods:

```
class AdultManager(models.Manager):
    def get_queryset(self):
        return super().get_queryset().filter(age__gte=18)

class Student(models.Model):
    name = models.CharField(max_length=50)
    age = models.IntegerField()

    objects = models.Manager()   # default
    adults = AdultManager()      # custom
```

---

10. Migrations

- Create migration:

```
python manage.py makemigrations
```

- Apply migration:

```
python manage.py migrate
```

- Inspect SQL:

```
python manage.py sqlmigrate app_name 0001
```

---

11. Signals (Optional)

- Trigger actions on model events (save, delete):

```
from django.db.models.signals import post_save
from django.dispatch import receiver

@receiver(post_save, sender=Student)
def student_saved(sender, instance, created, **kwargs):
```

```
    if created:
        print(f'New student created: {instance.name}')
```

---

12. Best Practices

1. Keep models **lean**; complex logic → move to services or model methods.
2. Use `related_name` for clarity in reverse relations.
3. Avoid using `null=True` on `CharField`; prefer `blank=True`.
4. Use **choices** for fields with limited options.
5. Use **custom managers** for reusable querysets.

If you want, I can also make a **visual diagram of "Models A to Z"** showing:
Fields → Relations → Queries → Methods → Managers → Signals, all in one flow.

Do you want me to make that diagram?