চমৎকার! এখন আমরা **Django Model Validation — Deep Level** শিখব।
এটা Django-র সবচেয়ে powerful কিন্তু অনেক beginner properly বুঝে না।
আমি আপনাকে *শুরু → advanced → production best practice* সব দেখাবো।

🧠 মডেল ভ্যালিডেশন — বড় ৩টি জায়গা

Django model-এর validation মোট ৩ভাবে করা যায়:

1. **Field-level validation**
2. **Model-level (clean()) validation**
3. **Full model validation (full_clean())**

এগুলো কীভাবে কাজ করে—deeply দেখি।

🧩 1) Field-level Validation (Automatic)

Django model fields নিজেই কিছু validation করে।

Example:

```
age = models.IntegerField()
```

❌ যদি age = "abc" দাও → ValidationError
✔️ IntegerField → only integer allowed

আরো built-in validator:

```
age = models.IntegerField(validators=[MinValueValidator(18)])
```

🧩 2) Custom Field Validator (Advanced)

এখানে আপনি নিজেই rule বানাবেন।

Example: name এ সংখ্যা থাকতে পারবে না

```
from django.core.exceptions import ValidationError

def validate_name(value):
```

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

Model:

```
name = models.CharField(max_length=100, validators=[validate_name])
```

এই validator শুধুমাত্র **name** field-এর জন্য চলবে।

🧩 3) Model-level Validation (clean() method)

এটাই advanced স্তর।
এখানে আপনি পুরো model-এর উপর একসাথে validation চালাতে পারবেন।

Why powerful?

✔️ multiple field একসাথে check
✔️ complex logic
✔️ conditional validation (age → marks)
✔️ global rules

Example:

```
def clean(self):
    if self.age < 18 and self.marks > 80:
        raise ValidationError("Underage students cannot score above 80.")
```

🔥 এখন Deep Dive করি — Model clean() কীভাবে কাজ করে?

🧬 Step-by-step flow:

যখন আপনি model validate করেন, Django এই sequence follow করে:

✔️ 1. Field-level validation

প্রতি field নিজের validation চালায়
(min value, type check, custom validator)

✔️ 2. Model-level clean() method

আপনার custom logic এখানে চলে

✔️ 3. Model.clean_fields()

সব fields individually validate করে

✔️ 4. Model.validate_unique()

unique constraints check করে

✔️ 5. সব errors collect করে → ValidationError throw করে

⚠️ Important: Model clean() automatically run হয় না save() এর সময়

এই কোড error উঠাবে না:

```
s = Student(name="A", age=10, marks=99)
s.save()   # NO validation here
```

কেন?

Django **validate automatically during save() করে না!**

⭐ How to force validation? (VERY IMPORTANT)

✔️ Correct way:

```
s = Student(name="A", age=10, marks=99)
s.full_clean()   # validate everything
s.save()
```

full_clean() = clean_fields + clean + validate_unique

এটাই **best practice**।

🛠️ Want automatic validation?

save() override করে full_clean() কল করা যায়:

```
def save(self, *args, **kwargs):
    self.full_clean()     # automatic validation
```

```
        super().save(*args, **kwargs)
```

এখন:

```
Student.objects.create(name="B", age=10)
```

→ ValidationError
(because age < 18)

📘 Deep Best Practice Model Example

```
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):
        # Field-level custom validation
        if any(char.isdigit() for char in self.name):
            raise ValidationError({"name": "Name cannot contain digits!"})

        if self.age < 18:
            raise ValidationError({"age": "Age must be at least 18."})

        # Multi-field logic
        if self.age < 18 and self.marks > 80:
            raise ValidationError(
                "Underage students cannot have marks above 80."
            )

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

    def save(self, *args, **kwargs):
        self.full_clean()  # ensure validation before saving
        super().save(*args, **kwargs)

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

🎯 Summary of Deep Learning

Concept	Purpose
Validators	Single-field rules
clean()	Complex multi-field rules
clean_fields()	Automatic field validation
validate_unique()	Unique check
full_clean()	Full validation pipeline
save() override	Automatic model validation

📌 পরের ধাপে আপনি কী শিখতে চান?

1. **clean() এর মধ্যে multiple errors কীভাবে return করবেন?**
2. **Field-specific error message কীভাবে দেবেন?**
3. **clean() + ModelForm integration?**
4. **Django Admin validation?**
5. **REST API validation (Django REST Framework)?**

যেটা বলবেন—আমরা সেটাই গভীরভাবে শিখব!

`full_clean()` **কবে, কখন, কোথায় কল করতে হয়**—এইটা Django model validation-এর সবচেয়ে গুরুত্বপূর্ণ জিনিস।
আমি খুব পরিষ্কারভাবে ব্যাখ্যা করছি যাতে আপনি ১০০% বুঝে যান।

✅ full_clean() কী করে?

`full_clean()` চলার সময় ৩টি জিনিস ঘটে:

1. **clean_fields()** → Field-level validation
2. **clean()** → Model-level validation
3. **validate_unique()** → Unique constraints

মানে: **Model-এর সব validation একবারে running হয়।**

🧠 কখন full_clean() কল করতে হবে? (Manual & Automatic)

🔵 1. When manually creating objects (Shell or scripts)

যখন শেল থেকে model তৈরি করছেন:

```
s = Student(name="A", age=10)
s.full_clean()   # <-- always needed!
s.save()
```

✔️ Data validated
❌ না করলে → Invalid data database-এ চলে যাবে

🔵 2. When saving objects manually in your views

Example: আপনি form ব্যবহার করছেন না, ডেটা নিজে process করছেন:

```
student = Student(name=request.POST["name"], age=request.POST["age"])
student.full_clean()  # must
student.save()
```

🔵 3. Before saving in bulk operations

Custom scripts, cron jobs, data import code → must validate.

🔵 4. When you want to validate entire model (not just a single field)

যখন model-level clean() ব্যবহার করেছেন:

```
def clean(self):
    if self.age < 18 and self.marks > 80:
        raise ValidationError("Underage students cannot score above 80.")
```

তখন **validate করতে full_clean() অবশ্যই কল করতে হবে।**

🔥 full*clean() কখন* স্বয়ংক্রিয়ভাবে_ কল হয়?

✔️ 1. Django Forms / ModelForms

Form → clean() → validation চলে
ModelForm.save() → internally model.full_clean() কল করে

তাই Form ব্যবহার করলে আপনাকে full_clean() কল করতে হয় না।

✔️ 2. Admin panel

Django Admin → ModelForm ব্যবহার করে
→ তাই validation automatically হয়

⚠️ Saved during `.save()` → full_clean() Auto Run হয় না (by default)

```
s = Student(age=5)
s.save()   # ❌ validation will NOT run
```

👉 **এই কারণে অনেকেই ভুল করে ভুল ডেটা ডাটাবেজে ঢুকিয়ে দেয়।**

⭐ Best Practice: save() override করে full_clean() enforce করা

```
def save(self, *args, **kwargs):
    self.full_clean()  # validation runs every time before save
    super().save(*args, **kwargs)
```

এখন:

```
Student.objects.create(age=10)
```

Automatically:

```
ValidationError
```

✔️ safest
✔️ production best practice
✔️ consistent validation everywhere

🎯 Summary (সবচেয়ে গুরুত্বপূর্ণ সারাংশ)

পরিস্থিতি	full_clean() লাগে?
Shell / script থেকে data save	✅ অবশ্যই
Views-এ manually model save	✅ অবশ্যই
ModelForm ব্যবহার করলে	❌ লাগে না (Form নিজেই করে)
Admin panel	❌ লাগে না
save() override করলে	auto validate হয়
REST API DRF (Serializer)	❌ লাগে না (serializer করে)

👉 এখন আপনি বলুন:

আপনি কি দেখতে চান:

1) full_clean() exception কীভাবে catch করবেন?

2) ModelForm + clean() কিভাবে কাজ করে?

3) Unique validation + full_clean()?

4) save() override-এর best practice?

যেটা চান, আমি গভীরভাবে শেখাবো।

Perfect! Let’s go step by step, deeply explaining all 4 points with examples so you can fully master Django model validation.

1️⃣ Catching exceptions from `full_clean()`

`full_clean()` raises **`ValidationError`** if validation fails. You can catch it using `try...except`.

```
from django.core.exceptions import ValidationError
from students.models import Student

s = Student(name="Ali123", age=15, marks=105)

try:
    s.full_clean()  # Runs all validations: fields + clean() + unique
    s.save()
except ValidationError as e:
    print("Validation errors:", e.message_dict)
```

Output

```
Validation errors: {
    'name': ['Name cannot contain digits!'],
    'age': ['Age must be at least 18.'],
    'marks': ['Marks must be between 0 and 100.']
}
```

💡 **Tip:** `e.message_dict` gives a dictionary with field-specific errors.

2️⃣ ModelForm + `clean()`

When you use a **ModelForm**, Django automatically calls:

1. **Field validators**
2. **Form clean_fieldname()**
3. **Form clean()**
4. **Model clean()** (via `full_clean()` internally)

Example:

```
from django import forms
from students.models import Student

class StudentForm(forms.ModelForm):
    class Meta:
        model = Student
        fields = ['name', 'age', 'marks']

    # Field-level clean
```

```
    def clean_age(self):
        age = self.cleaned_data.get('age')
        if age < 18:
            raise forms.ValidationError("You must be at least 18 years
old.")
        return age

    # Form-level clean (multi-field)
    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("Underage students cannot have
marks above 80.")
```

Usage in View:

```
form = StudentForm(data=request.POST)
if form.is_valid():
    form.save()  # full_clean() internally called
else:
    print(form.errors)
```

💡 **Key point:**
ModelForm automatically calls **model-level** `clean()` when `save()` is called.

3️⃣ Unique validation + `full_clean()`

Django supports **unique fields**. Validation occurs in:

- ModelForm → automatically
- `full_clean()` → manually

Example Model

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

Shell

```
s1 = Student.objects.create(name="Ali", age=20)
s2 = Student(name="Ali", age=25)
```

```
try:
    s2.full_clean()  # validates unique constraint
    s2.save()
except ValidationError as e:
    print(e.message_dict)
```

Output

```
{'name': ['Student with this Name already exists.']}
```

💡 **Tip:** `validate_unique()` is automatically called inside `full_clean()`.

4️⃣ `save()` override — Best Practice

To **ensure model-level validation always runs**, override `save()`:

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

    def clean(self):
        if self.age < 18:
            raise ValidationError({'age': 'Age must be 18 or above!'})
        if self.marks < 0 or self.marks > 100:
            raise ValidationError({'marks': 'Marks must be 0-100'})

    def save(self, *args, **kwargs):
        self.full_clean()  # validate every time save() is called
        super().save(*args, **kwargs)
```

Now

```
Student.objects.create(name="Ali", age=15, marks=120)
```

✅ Automatically raises ValidationError
✔️ Ensures no invalid data ever reaches DB

✅ Summary Table

Feature	How it works
`full_clean()`	Runs field, model, and unique validation
Catch ValidationError	`try...except ValidationError` & use `message_dict`
ModelForm	Automatically runs `full_clean()` on save; supports field & form clean
Unique field	Checked during `full_clean()` / ModelForm validation
save() override	Best practice: call `full_clean()` inside save() to prevent invalid DB entries

---

If you want, I can **create a complete production-ready Student model** with:

- field-level validators
- model-level clean()
- save() override
- proper error handling

…so you can **copy-paste and practice directly in shell or admin**.

Do you want me to do that?

You should override the `save()` method in a Django model when you want to add extra logic every time an object is saved.

In the context of **validation**, the most common reason is to **automatically call `full_clean()` before saving** so that invalid data never gets into the database.

✅ Why override `save()`

1. **Automatic validation**
   Without overriding, `full_clean()` does **not run automatically** during `save()`.
   By overriding, you enforce that **every save validates the model**.
2. **Preprocessing or modifying fields**
   Example: auto-generate slug from name before saving.

3. **Logging or auditing**
   Example: track last_modified timestamps, create history records.

✅ Example: save() override for validation

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

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

    def clean(self):
        if self.age < 18:
            raise ValidationError({'age': 'Age must be at least 18.'})
        if not (0 <= self.marks <= 100):
            raise ValidationError({'marks': 'Marks must be between 0 and
100.'})

    def save(self, *args, **kwargs):
        self.full_clean()  # ensure all validation runs
        super().save(*args, **kwargs)
```

Usage

```
Student.objects.create(name="Ali", age=15, marks=120)
# → ValidationError raised automatically
```

✅ No invalid data enters the database.

⚡ Key Points When Overriding `save()`

1. Always call `super().save(*args, **kwargs)` at the end to actually save the object.
2. Call `self.full_clean()` **before** `super().save()` to enforce validation.
3. Only override `save()` when you want **logic applied automatically every time** the model is saved.

🧠 When you don't need to override `save()`

- If you are always using **ModelForms** or **Admin**, validation already happens.
- If no preprocessing or automatic validation is needed, overriding `save()` is optional.

If you want, I can create a **complete example Student model** that uses:

- field validators
- model-level `clean()`
- `save()` override
- unique constraints
- automatic exception handling

…so that it's **ready for shell, forms, and admin**.

Do you want me to do that?

---

how to override save() and delete()

Great! Let's go **deeply** into Django's `save()` and `delete()` methods. I'll explain **when they are called, how to use them, and why you might override them**, with examples.

1️⃣ save() method

What it does

- `save()` stores or updates a model instance in the database.
- If the object is new → it **inserts** a row.
- If the object already exists (has a primary key) → it **updates** the row.

When it is called

1. Explicitly by you:

```
student = Student(name="Ali", age=20)
student.save()
```

2. Implicitly via:

- `objects.create()` → internally calls `save()`

- ModelForms → `form.save()`
- Admin → saves object

How to use it

```
# Normal save
s = Student(name="Rafi", age=19)
s.save()

# Update existing object
s.age = 20
s.save()  # updates row in DB
```

Why you might override save()

1. **Automatic validation**

```
def save(self, *args, **kwargs):
    self.full_clean()  # validate before saving
    super().save(*args, **kwargs)
```

2. **Preprocessing fields**

```
from django.utils.text import slugify

def save(self, *args, **kwargs):
    self.slug = slugify(self.name)
    super().save(*args, **kwargs)
```

3. **Logging / auditing**

```
def save(self, *args, **kwargs):
    print(f"Saving {self.name}")
    super().save(*args, **kwargs)
```

2️⃣ delete() method

What it does

- `delete()` removes the object from the database.

```
s = Student.objects.get(id=1)
s.delete()
```

- Deletes **one object**.
- You can also delete **querysets**:

```
Student.objects.filter(age__lt=18).delete()
```

- Deletes all students with age < 18 in one query.

When it is called

- Explicitly by you:

```
s = Student.objects.get(name="Ali")
s.delete()
```

- Implicitly via cascading deletes:

```
class Enrollment(models.Model):
    student = models.ForeignKey(Student, on_delete=models.CASCADE)
```

- If a Student is deleted → all related Enrollment objects are deleted automatically.

Why you might override delete()

1. **Cleanup related data**

```
def delete(self, *args, **kwargs):
    print(f"Deleting student {self.name}")
    # delete related files if any
    super().delete(*args, **kwargs)
```

2. **Soft delete (mark as inactive instead of removing)**

```
def delete(self, *args, **kwargs):
    self.is_active = False
```

```
        self.save()
```

3. **Logging / auditing**

```
def delete(self, *args, **kwargs):
    print(f"{self.name} deleted")
    super().delete(*args, **kwargs)
```

✅ Summary Table

Method	When Called	How to Use	Why Override / Use
`save()`	New object or updating existing	`obj.save()` or `objects.create()`	Validation, preprocessing, logging, auto-slug, audit
`delete()`	When you want to remove object	`obj.delete()` or `QuerySet.delete()`	Cleanup, logging, soft delete, cascading actions

---

⚡ Tips

1. Always call `super().save()` or `super().delete()` in overrides.
2. Overriding is optional — only if you need **custom behavior**.
3. `save()` triggers signals like `pre_save` and `post_save`.
4. `delete()` triggers `pre_delete` and `post_delete` signals.

If you want, I can make a **complete Student model example** showing:

- `save()` with validation & slug
- `delete()` with logging & soft delete
- signals + full validation

…so you can **practice all together**.

Do you want me to do that?

---

some note important

Absolutely! Let's go **deep dive** into Django's `save()` and `delete()` signals, the importance of `super()` in overrides, and how to use them with examples.

1️⃣ Why always call `super().save()` or `super().delete()`

When you override `save()` or `delete()`, Django **doesn't automatically call the parent implementation**.

- `super().save()` → actually writes to the database.
- `super().delete()` → actually removes the row from the database.

❌ If you don't call `super()`, your custom logic runs but **the database is not updated**.

Example: Overriding save() correctly

```
from django.db import models
from django.core.exceptions import ValidationError
from django.utils.text import slugify

class Student(models.Model):
    name = models.CharField(max_length=100)
    age = models.IntegerField()
    slug = models.SlugField(blank=True)

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

    def save(self, *args, **kwargs):
        self.slug = slugify(self.name)  # preprocess field
        self.full_clean()               # validate model
        super().save(*args, **kwargs)  # actually save to DB
```

✅ Here:

- `slug` is automatically generated
- Validation runs before saving
- Database is updated because of `super().save()`

Example: Overriding delete() correctly

```python
def delete(self, *args, **kwargs):
    print(f"Deleting student: {self.name}")
    super().delete(*args, **kwargs)  # actually remove from DB
```

❌ Without `super().delete()`, the student **won't be deleted**, only your print runs.

2️⃣ Signals: pre_save, post_save, pre_delete, post_delete

Django provides **hooks** that trigger **before or after save/delete**:

Signal	When triggered
`pre_save`	Before `save()` is executed
`post_save`	After `save()` is executed
`pre_delete`	Before `delete()` is executed
`post_delete`	After `delete()` is executed

2a) Using pre_save / post_save

```python
from django.db.models.signals import pre_save, post_save
from django.dispatch import receiver

@receiver(pre_save, sender=Student)
def before_saving_student(sender, instance, **kwargs):
    print(f"About to save: {instance.name}")

@receiver(post_save, sender=Student)
def after_saving_student(sender, instance, created, **kwargs):
    if created:
        print(f"New student created: {instance.name}")
    else:
        print(f"Student updated: {instance.name}")
```

Test:

```python
s = Student(name="Ali", age=20)
s.save()
```

Output:

```
About to save: Ali
New student created: Ali
```

2b) Using pre_delete / post_delete

```python
from django.db.models.signals import pre_delete, post_delete

@receiver(pre_delete, sender=Student)
def before_deleting_student(sender, instance, **kwargs):
    print(f"About to delete: {instance.name}")

@receiver(post_delete, sender=Student)
def after_deleting_student(sender, instance, **kwargs):
    print(f"Deleted student: {instance.name}")
```

Test:

```python
s = Student.objects.get(name="Ali")
s.delete()
```

Output:

```
About to delete: Ali
Deleted student: Ali
```

3️⃣ Why signals are useful

1. Decouple logic from model methods → you don't need to override save/delete.
2. Run background tasks after creation, update, or deletion.
   - Example: send email after student created
   - Example: log deletion to a file
3. Keep code DRY — multiple listeners can act on the same signal.

4️⃣ Important Tips

1. Always use `super()` in overrides if you still want default DB behavior.

2. `pre_save` happens **before** save(), so you can modify fields here.
3. `post_save` happens **after** save(), can check `created` flag.
4. Signals are **global**, so all code in your project listening to the signal will run.

✅ Example: Complete Student Model with save, delete, signals

```
class Student(models.Model):
    name = models.CharField(max_length=100)
    age = models.IntegerField()
    slug = models.SlugField(blank=True)

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

    def save(self, *args, **kwargs):
        self.slug = self.name.lower().replace(" ", "-")
        self.full_clean()
        super().save(*args, **kwargs)

    def delete(self, *args, **kwargs):
        print(f"Deleting: {self.name}")
        super().delete(*args, **kwargs)
```

Signals:

```
from django.db.models.signals import pre_save, post_save, pre_delete,
post_delete
from django.dispatch import receiver

@receiver(pre_save, sender=Student)
def before_save(sender, instance, **kwargs):
    print(f"[pre_save] {instance.name}")

@receiver(post_save, sender=Student)
def after_save(sender, instance, created, **kwargs):
    print(f"[post_save] {instance.name}, created={created}")

@receiver(pre_delete, sender=Student)
def before_delete(sender, instance, **kwargs):
    print(f"[pre_delete] {instance.name}")

@receiver(post_delete, sender=Student)
```

```
def after_delete(sender, instance, **kwargs):
    print(f"[post_delete] {instance.name}")
```

In Django, signals can be defined in several ways, but the recommended and cleanest way is not directly inside `models.py`. Let me explain clearly.

1️⃣ Recommended way — signals.py

1. Create a **`signals.py`** file inside your app folder:

```
students/
├── models.py
├── signals.py
├── apps.py
```

2. Example `signals.py`:

```
from django.db.models.signals import pre_save, post_save, pre_delete,
post_delete
from django.dispatch import receiver
from .models import Student

@receiver(pre_save, sender=Student)
def before_saving_student(sender, instance, **kwargs):
    print(f"[pre_save] About to save {instance.name}")

@receiver(post_save, sender=Student)
def after_saving_student(sender, instance, created, **kwargs):
    print(f"[post_save] Saved {instance.name}, created={created}")

@receiver(pre_delete, sender=Student)
def before_deleting_student(sender, instance, **kwargs):
    print(f"[pre_delete] About to delete {instance.name}")

@receiver(post_delete, sender=Student)
def after_deleting_student(sender, instance, **kwargs):
    print(f"[post_delete] Deleted {instance.name}")
```

2️⃣ Make sure signals are loaded — use apps.py

If you just put signals in `signals.py`, Django **won't automatically load them** unless you import the file somewhere.

The recommended way is to **import signals in the `ready()` method** of your app's AppConfig:

```
# students/apps.py
from django.apps import AppConfig

class StudentsConfig(AppConfig):
    default_auto_field = 'django.db.models.BigAutoField'
    name = 'students'

    def ready(self):
        import students.signals  # import signals when app is ready
```

3️⃣ Why not directly in models.py?

- Putting signals in `models.py` can **cause circular import errors**, especially when your signals import models from other apps.
- Keeping `signals.py` separate is **clean, maintainable, and scalable**.

4️⃣ Quick Checklist

Step	Action
1	Create `signals.py` in your app
2	Define your signals there (`pre_save`, `post_save`, etc.)
3	Import `signals.py` in `apps.py` inside `ready()`
4	Make sure your AppConfig is used in `INSTALLED_APPS`

Example Complete Setup

students/models.py

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

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

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

students/signals.py

```python
from django.db.models.signals import pre_save, post_save
from django.dispatch import receiver
from .models import Student

@receiver(pre_save, sender=Student)
def pre_save_student(sender, instance, **kwargs):
    print(f"About to save {instance.name}")

@receiver(post_save, sender=Student)
def post_save_student(sender, instance, created, **kwargs):
    print(f"Saved {instance.name}, created={created}")
```

students/apps.py

```python
from django.apps import AppConfig

class StudentsConfig(AppConfig):
    default_auto_field = 'django.db.models.BigAutoField'
    name = 'students'

    def ready(self):
        import students.signals
```

✅ Now, **every time you save a Student** in shell, form, or admin, signals will automatically run.
