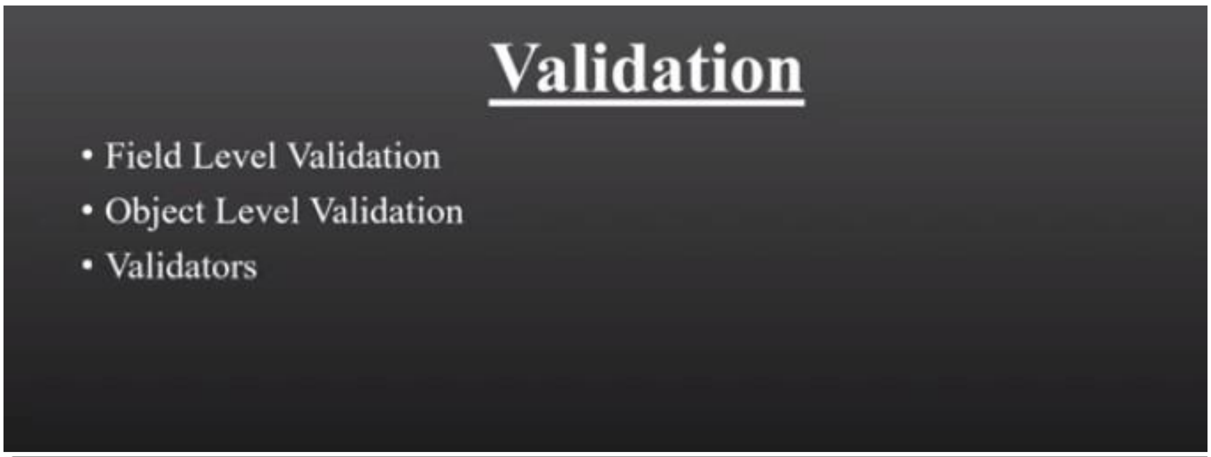
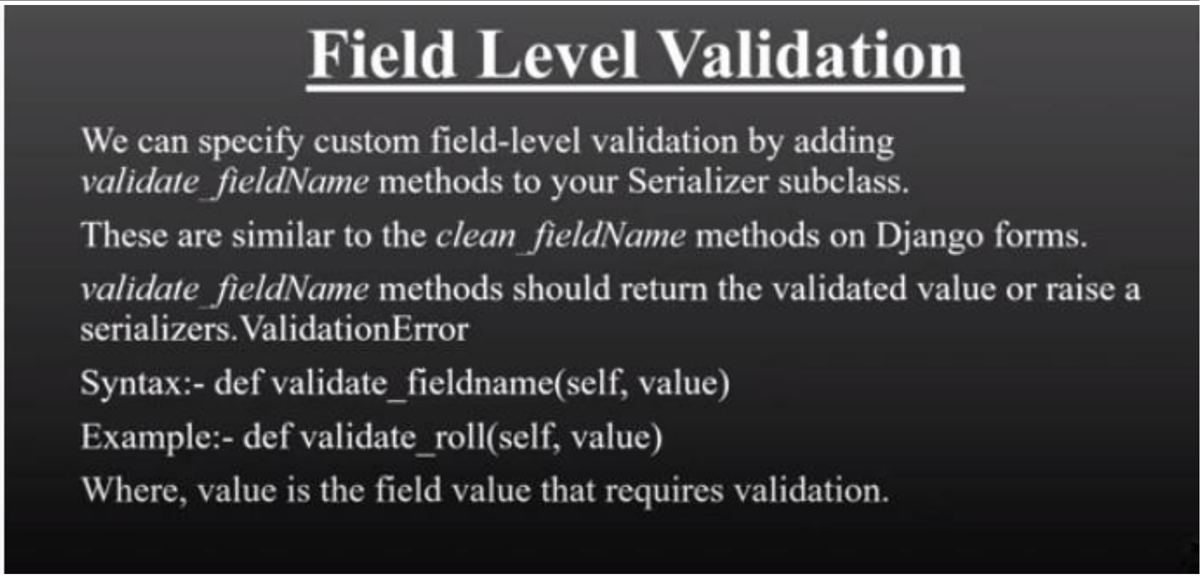
Priority oder

1. Validators
2. Field level
3. Object level



Validation

- Field Level Validation
- Object Level Validation
- Validators



Field Level Validation

We can specify custom field-level validation by adding *validate_fieldName* methods to your Serializer subclass.

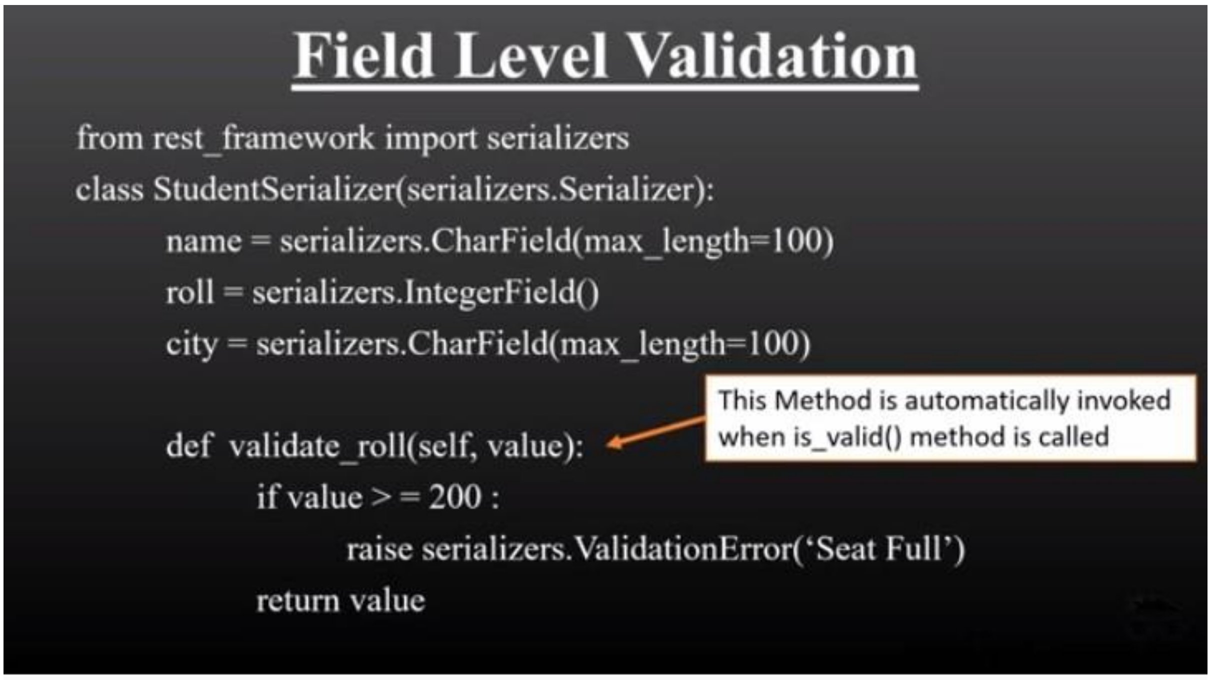
These are similar to the *clean_fieldName* methods on Django forms.

validate_fieldName methods should return the validated value or raise a serializers.ValidationError

Syntax:- def validate_fieldname(self, value)

Example:- def validate_roll(self, value)

Where, value is the field value that requires validation.

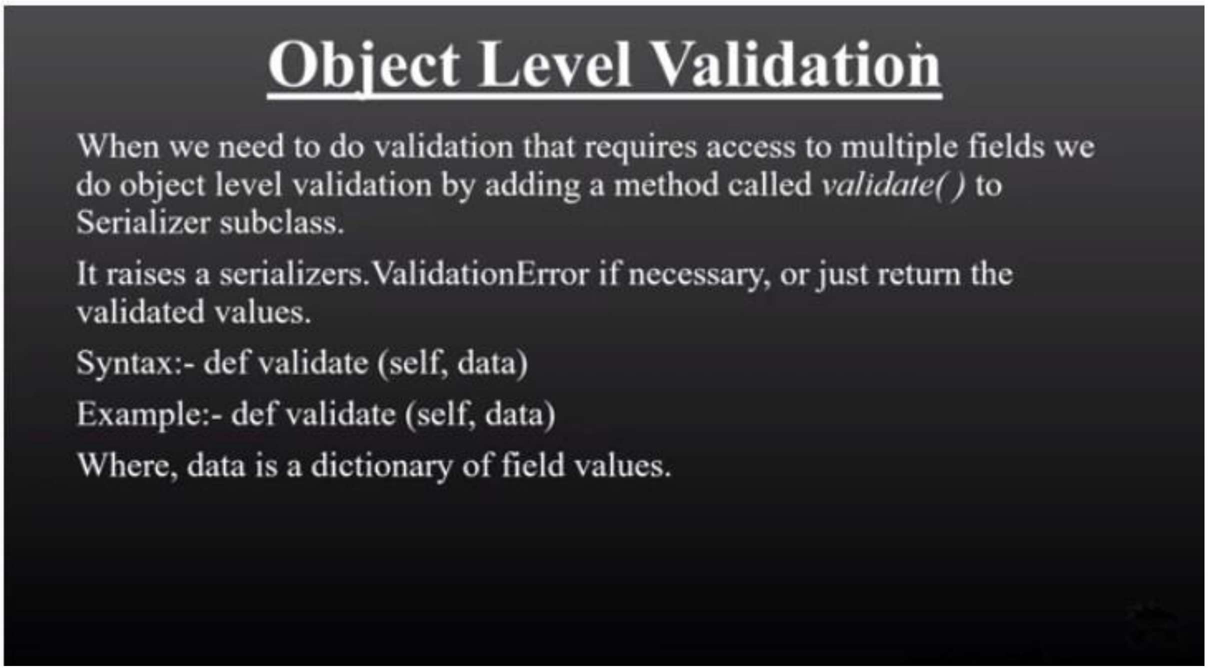


Field Level Validation

```
from rest_framework import serializers
class StudentSerializer(serializers.Serializer):
        name = serializers.CharField(max_length=100)
        roll = serializers.IntegerField()
        city = serializers.CharField(max_length=100)

        def  validate_roll(self, value):
                if value > = 200 :
                        raise serializers.ValidationError('Seat Full')
                return value
```

This Method is automatically invoked when is_valid() method is called



Object Level Validation

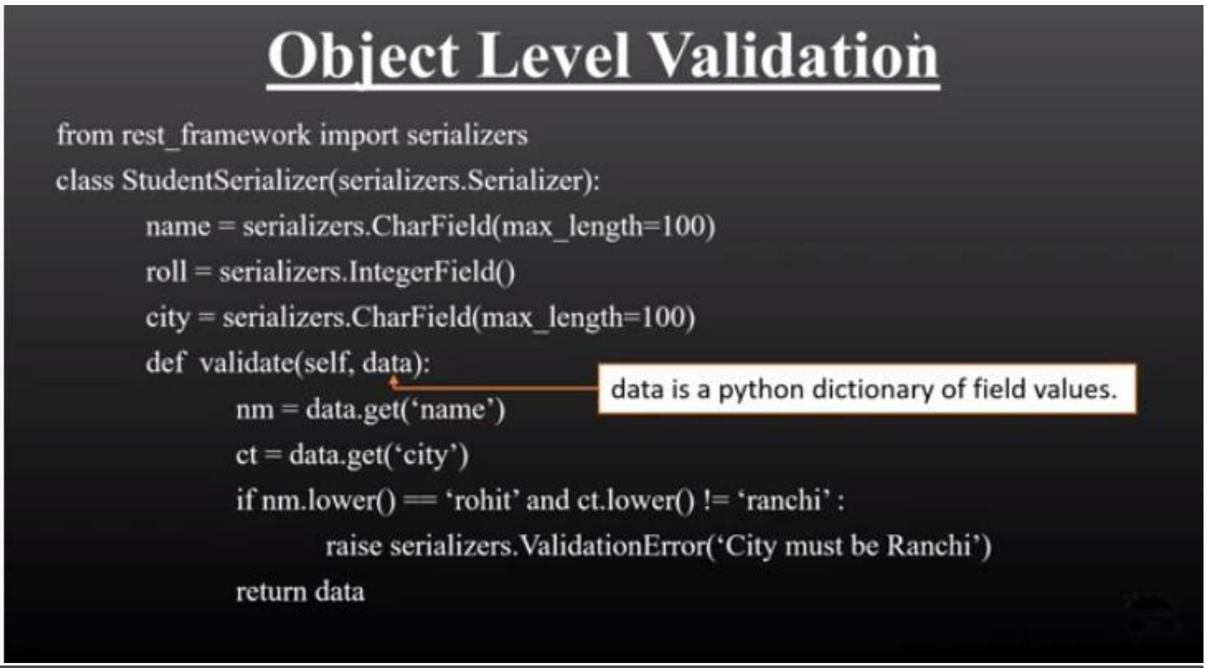
When we need to do validation that requires access to multiple fields we do object level validation by adding a method called *validate( )* to Serializer subclass.

It raises a serializers.ValidationError if necessary, or just return the validated values.

Syntax:- def validate (self, data)

Example:- def validate (self, data)

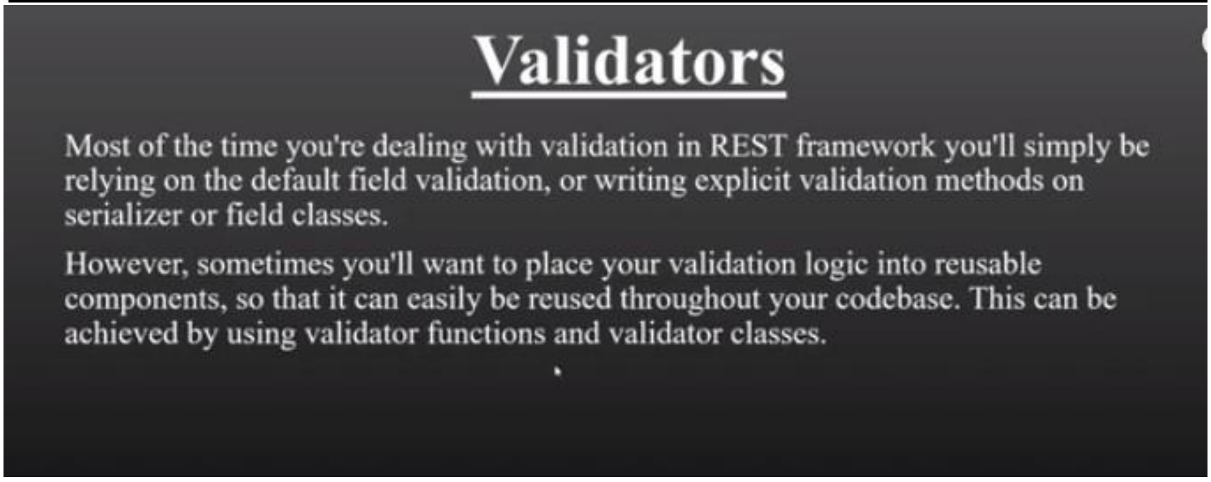
Where, data is a dictionary of field values.



Object Level Validation

```
from rest_framework import serializers
class StudentSerializer(serializers.Serializer):
        name = serializers.CharField(max_length=100)
        roll = serializers.IntegerField()
        city = serializers.CharField(max_length=100)
        def  validate(self, data):
                nm = data.get(‘name’)
                ct = data.get(‘city’)
                if nm.lower() == ‘rohit’ and ct.lower() != ‘ranchi’ :
                        raise serializers.ValidationError(‘City must be Ranchi’)
                return data
```

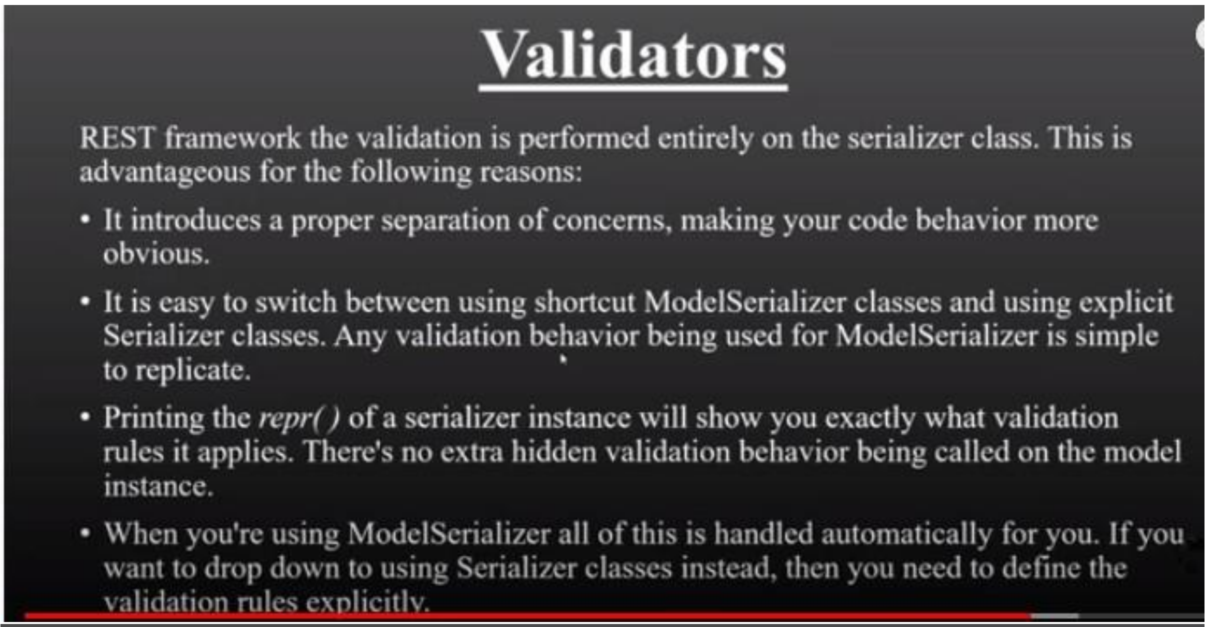
data is a python dictionary of field values.



Validators

Most of the time you're dealing with validation in REST framework you'll simply be relying on the default field validation, or writing explicit validation methods on serializer or field classes.

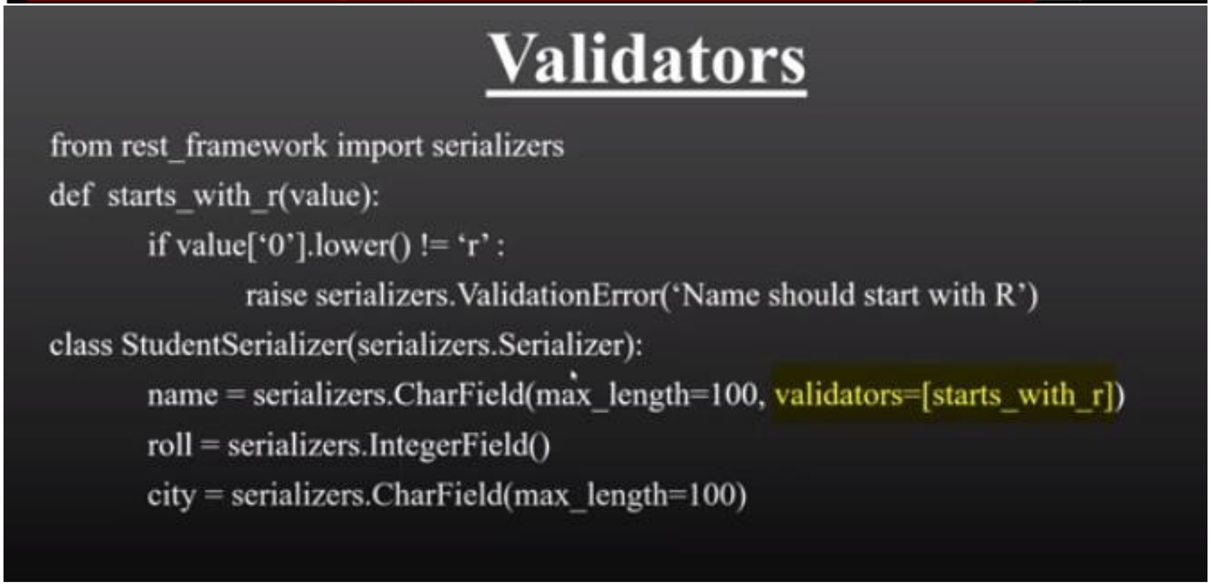
However, sometimes you'll want to place your validation logic into reusable components, so that it can easily be reused throughout your codebase. This can be achieved by using validator functions and validator classes.



Validators

REST framework the validation is performed entirely on the serializer class. This is advantageous for the following reasons:

- It introduces a proper separation of concerns, making your code behavior more obvious.
- It is easy to switch between using shortcut ModelSerializer classes and using explicit Serializer classes. Any validation behavior being used for ModelSerializer is simple to replicate.
- Printing the *repr()* of a serializer instance will show you exactly what validation rules it applies. There's no extra hidden validation behavior being called on the model instance.
- When you're using ModelSerializer all of this is handled automatically for you. If you want to drop down to using Serializer classes instead, then you need to define the validation rules explicitly.



Validators

```
from rest_framework import serializers
def  starts_with_r(value):
        if value['0'].lower() != 'r' :
                raise serializers.ValidationError('Name should start with R')
class StudentSerializer(serializers.Serializer):
        name = serializers.CharField(max_length=100, validators=[starts_with_r])
        roll = serializers.IntegerField()
        city = serializers.CharField(max_length=100)
```