**Django Validation**

Built-in methods are provided by Django for automatic validation of data of the form. Forms in Django are submitted only if it has a CSRF token. An easy approach is used by it for the form validation in Django. The is\_valid() method is defined in the form class of Django and it is for the validation of the data of every field that the form has. If the data in the field is valid then true is returned by it and then it places data into an attribute cleaned\_data

When you enter data, the browser and/or the web server will check to see that the data is in the correct format and within the constraints set by the application. Validation done in the browser is called client-side validation(JavaScript, HTML, and CSS.), while validation done on the server is called server-side validation(PHP, Python, Node.js). client-side validation.

**We can implement validation logic by using the following 2 ways.**

1. Explicitly by the programmer by using clean methods
2. By using Django inbuilt validators

**Note: All validations should be implemented in the forms.py file**

**Build\_in Validators**

Django's form (and model) fields support use of utility functions and classes known as validators. A validator is a callable object or function that takes a value and returns nothing if the value is valid or raises a ValidationError if not.

Built-in field validations in Django models are predefined in Django. They are default validations. Every field in Django has validations that are unique to it. The module django.core.validators contain variables that can be called. We can use the callable variables for the model and field. These callable variables are for internal use but can also be used for our field.

**PROJNAME-->APPNAME-->forms.py**

from django import forms

from django.core import validators

class NAMEForm(forms.Form):

field1 = forms.DATATYPE(validators = [validators.ANYValidator, validators.ANYValidator2])

field2 = forms.DATATYPE()

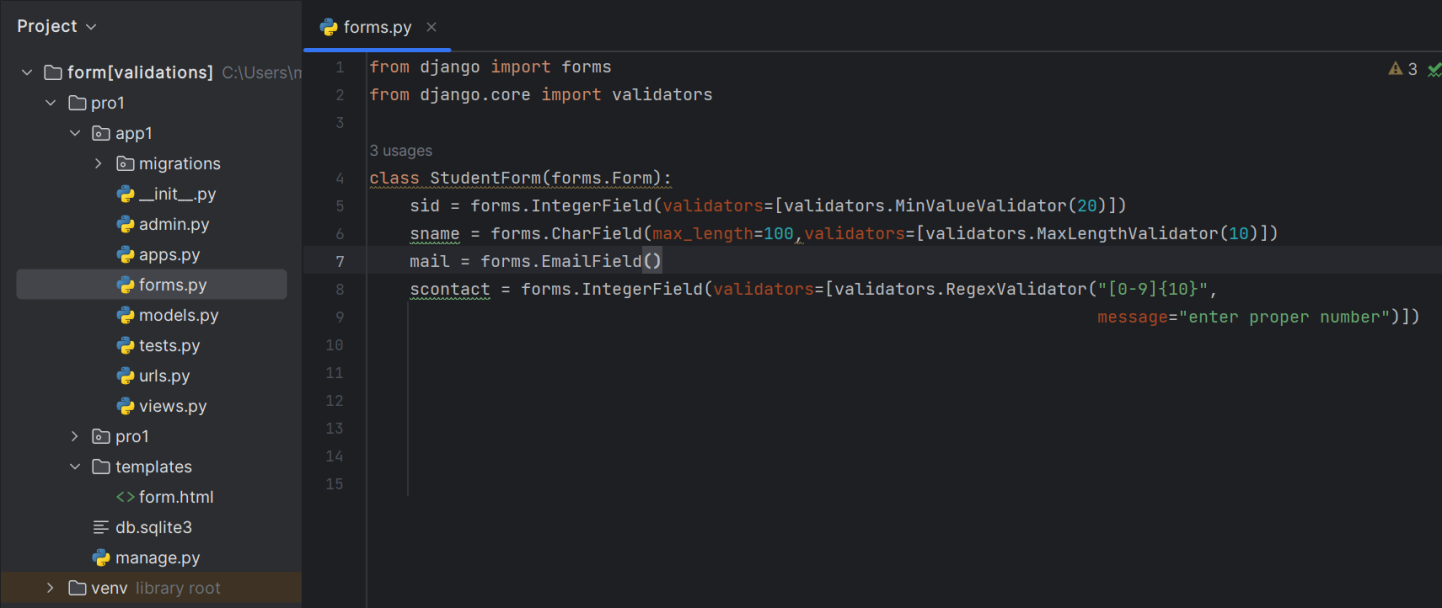
|  |  |  |
| --- | --- | --- |
| **Built-in Field Validations** | **Description** | **Syntax** |
| **RegexValidator** | If the regex is not assigned as None, it allows regex overrides. This can be a pre-compiled regular expression.  If the message is not set as none, it will override the message. If the code is not assigned as none, it will enable overriding code.  If regex is not a regular expression string, a TypeError is raised.  The RegexValidator searches the provided value with re.search(). The RegexValidator, by default, gives a ValidationError with message ad code. This is done when the match is not found. If the inverse\_match is set to True, we can obtain an opposite result. | **Class**  **RegexValidator(regex=None,message=None)** |
| **EmailValidator** | When not set to None, the message allows overriding of the message. Similarly, when code and allowlist are not set to None, it will enable overriding code and allowlist. | **Class**  **EmailValidator(code=None, message=None,**  **allowlist= None)** |
| **MaxValueValidator** | A ValidationError is set with “max\_value”  when the limit\_value is greater than the value. | **class MaxValueValidator(limit\_value, message=None)** |
| **MinValueValidator** | A ValidationError is set with “min\_value”  when the limit\_value is less than the value. | **class MinValueValidator(limit\_value, message=None)** |
| **MaxLengthValidator** | A ValidationError is set with “max\_length”  when the limit\_value is greater than the value. | **class MaxLengthValidator(limit\_value, message=None)** |
| **MinLengthValidator** | A ValidationError is set with “min\_length”  when the limit\_value is lesser than the value. | **class MinLengthValidator(limit\_value, message=None)** |

Some of the built-in field validations in Django models that be used to make changes are:-

|  |  |
| --- | --- |
| **Fields Options** | **Description** |
| Null | Default is False. Django will store the values as Null in the database if set True. |
| Blank | Default is False. If True, the field is allowed to be blank. |
| Db\_column | Django will use the field's name if no values are given. |
| Default | Every time a new object is created, this will be called. |
| Help text | Used to display text with the form widget. |
| Primary key | If set to True, then the field is the model's primary key. |
| Editable | If set to false, Django will not display the field. |
| Error message | Lets us override the default message of the field. |
| Verbose name | Name for the field that humans can read. |
| Unique | If set to true, then the field must be unique. |

**Form Validation**

In Django, **form validation** can be extended using validators, which are functions that perform specific validation checks on form fields. These validators can be applied to form fields in the form class definition to ensure data integrity and enforce specific constraints.



* **EmailValidator** is applied to the **mail** field to ensure it follows the correct email format.
* **MinValueValidator** is used with the **IntegerField** for the **sid** field to ensure the entered value is at least 20.
* **MaxLengthValidator** is applied to the **Sname** field to ensure the entered character is greater than 10.
* **RegexValidator** is applied to the **scontact** field to ensure the entered number should contain 10 digit.

A screenshot of a computer

Description automatically generated

**Custom Validators:**

You can also create custom validators for more complex validation requirements. These are functions that take the field's value as input and raise a **ValidationError** if the validation fails.

In Django, both clean() and clean\_<field\_name>() methods are used for form validation within Django forms or model forms.

**Clean() Method:**

1. This method is used for general validation across multiple fields in a form
2. This method returns the clean data, which is then inserted into the cleaned\_data dictionary of the form.
3. When you override the clean() method in a Django form or model form, you're able to perform validation that involves multiple fields. This method is called after all the individual field validations (clean\_<field\_name>() methods) have been executed successfully.
4. You might use this method for cross-field validation, where the validity of one field depends on the value of another field.

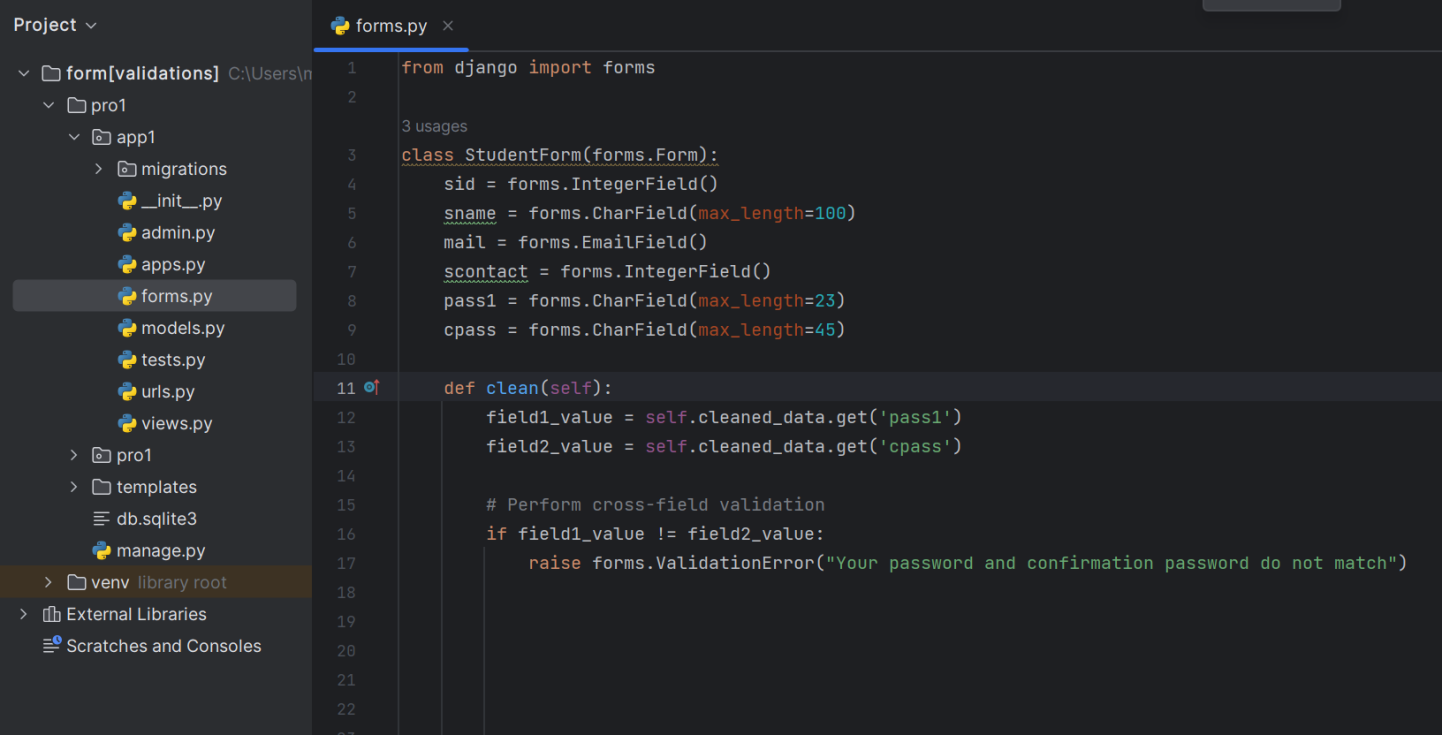
**Syntax:**

**def clean(self):**

**all\_data = super().clean()**

**entered\_data1 = all\_data['field1']**

**entered\_data2 = all\_data['field2']#validation logic NO RETURN STATEMENT SHOULD BE USED**



A screenshot of a computer

Description automatically generated

**Clean\_<fieldname>( ) Method:**

1. This method is used for field-specific validation.
2. Django allows you to define methods named clean\_<field\_name>() for individual fields. These methods perform validation specific to that field.
3. These methods are automatically called by Django during form validation.
4. In the FormClass for any field if we define clean method then at the time of submit the form, Django will call this method automatically to perform validations. If the clean method won't raise any error then only form will be submitted**.**
5. This method has no parameters.

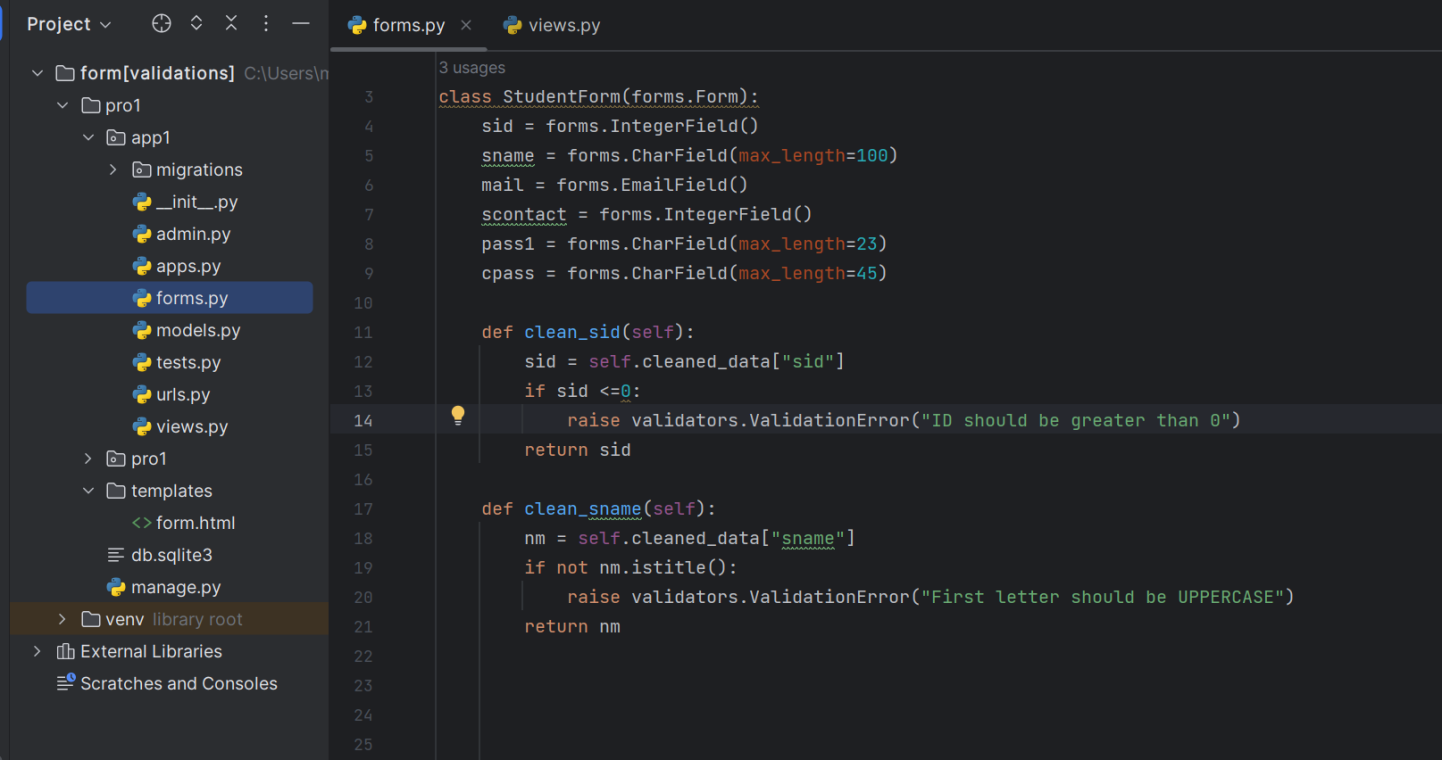
**Syntax:**

**def clean\_field1(self):**

**entered\_data= self.cleaned\_data['field1']**

**return value**

**#validation logic with entered\_data MANDATORILY RETURN VALUE**



A screenshot of a computer

Description automatically generated

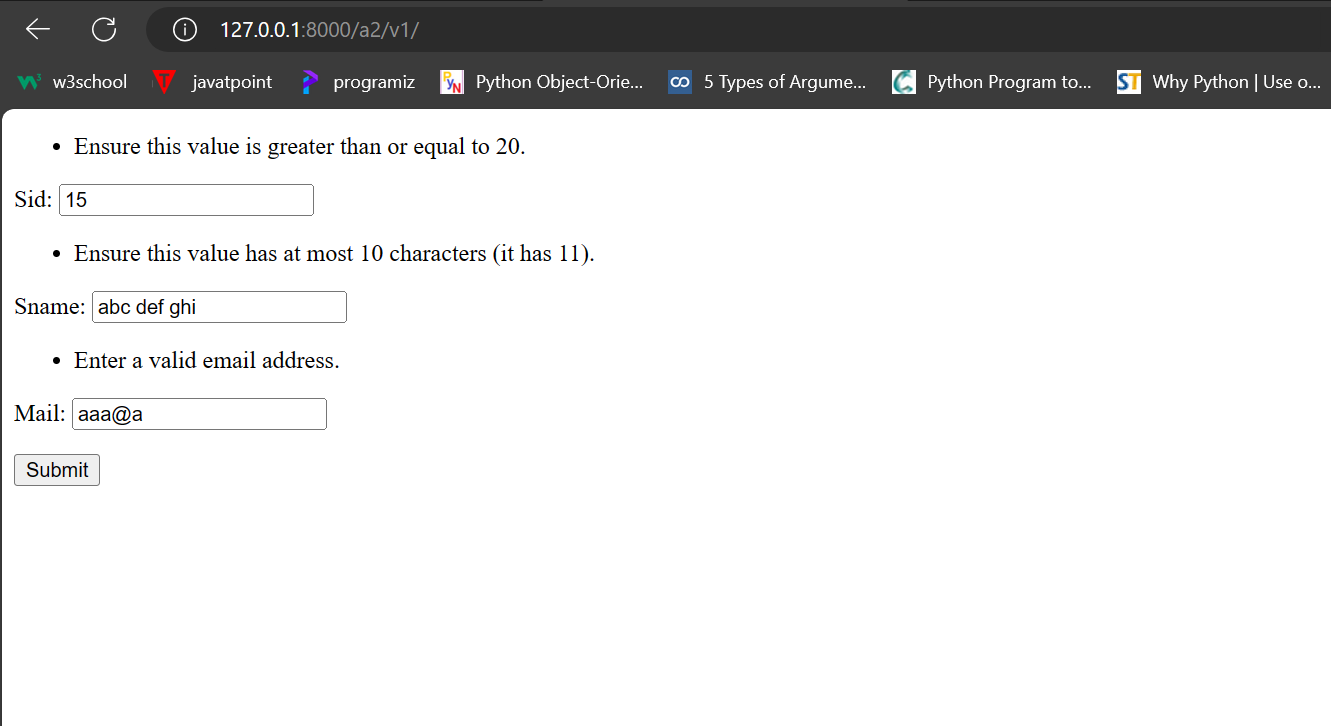
**ModeForm Validation**

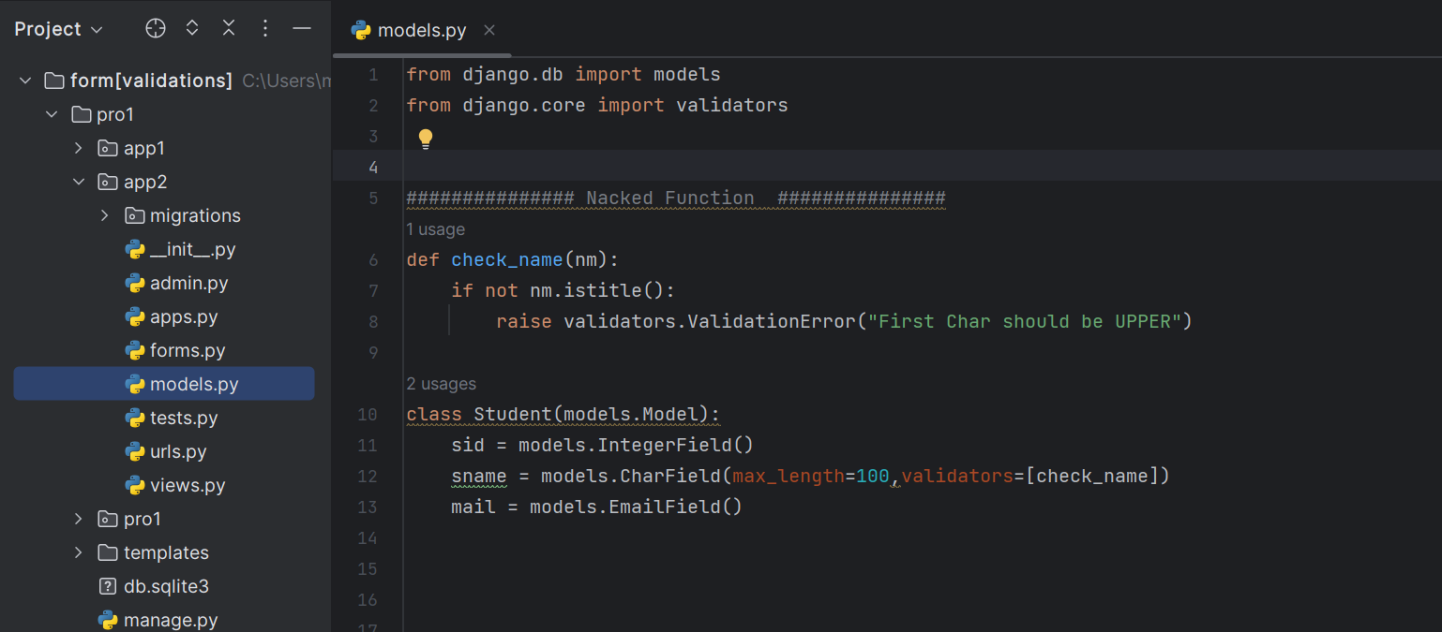
1. Similarly we can apply validations on modelform by defining clean, clean\_<fieldname> or using validators.
2. We also can apply validators on model class fields , see below.

**Build\_in Validators:**

A screenshot of a computer program

Description automatically generated



**Custom Validators** 

A screenshot of a computer

Description automatically generated

A screen shot of a computer program

Description automatically generated

