Template Filters:

Django template filters are built-in or custom functions used in Django templates to transform the values of variables before displaying them. They allow you to modify data directly in templates, keeping your views and models clean. For example, they can be used to format dates, change text case, and truncate strings. Filters are applied by using the pipe symbol |.

Examples of Django Template Filters:

1. date: Formats a date object.

{{ my_date|date:"Y-m-d" }}

2. lower: Converts text to lowercase.
{{ my_text|lower }}

3. length: Gets the length of a string or list.
{{ my_list|length }}

4. truncatechars: Truncates text to a specific number of characters.
{{ my_text|truncatechars:10 }}

5. default: Sets a default value if the variable is empty.
{{ my_var|default:"Default text" }}

6. pluralize: Adds an "s" to a word based on a condition, useful for singular/plural forms.
{{ item_count }} item{{ item_count|pluralize }}

Template filters in Django offer several benefits:

1. Separation of Concerns: Filters keep logic out of templates, letting templates focus on presentation while views and models handle data.
2. Code Readability: Using filters makes templates cleaner and more readable by reducing complex formatting code.
3. Reusability: Filters are reusable across templates, avoiding redundant code and making it easier to maintain.
4. Customization: Django allows custom filters, so you can tailor filters to fit specific formatting needs in your application.

5. Simplified Formatting: Filters make it easy to format and manipulate data directly in templates (e.g., dates, text capitalization), which is especially useful for rapid development.
6. Error Reduction: Filters handle many common data transformations directly, reducing errors that might arise from repeatedly writing formatting logic.

Django provides a variety of built-in template filters that cover common needs. Here's a list of some of the most commonly used built-in filters:

Text Filters:

1. add: Adds a value to the variable.
   {{ number|add:"10" }}
2. capfirst: Capitalizes the first character.
   {{ "hello"|capfirst }} # Output: Hello
3. center: Centers the text within a given width.
   {{ "Django"|center:10 }}
4. cut: Removes all occurrences of a specified string.
   {{ "hello world"|cut:"o" }} # Output: hell wrld
5. join: Joins a list with a specified string.
   {{ my_list|join:", " }}
6. length: Returns the length of a list, string, or dictionary.
   django
   Copy code
   {{ my_list|length }}
7. lower: Converts the string to lowercase.
   {{ "HELLO"|lower }}
8. slugify: Converts text into a slug (URL-friendly format).
   {{ "Hello World!"|slugify }} # Output: hello-world
9. title: Capitalizes each word.
   {{ "hello world"|title }} # Output: Hello World
10. truncatechars: Truncates a string to a specified number of characters.
    {{ "Hello, world!"|truncatechars:5 }} # Output: Hello...

Date Filters

date: Formats a date or datetime.
{{ my_date|date:"Y-m-d" }}

time: Formats a time object.
{{ my_time|time:"H:i" }}

timesince: Returns the time since a specific date.
{{ my_date|timesince }}

Logic and Conditional Filters

default: Provides a default value if the variable is empty.
{{ my_var|default:"Default value" }}

default_if_none: Similar to default, but only applies if the variable is None.
{{ my_var|default_if_none:"None value" }}

divisibleby: Returns True if the variable is divisible by the given value.
{{ number|divisibleby:3 }}

yesno: Converts Boolean values into human-readable strings.
{{ value|yesno:"yes,no,maybe" }}

List and Dictionary Filters

first: Returns the first item in a list.
{{ my_list|first }}

last: Returns the last item in a list.
{{ my_list|last }}

length_is: Checks if the length of a list or string matches a given value.
{{ my_list|length_is:3 }}

random: Returns a random item from a list.
{{ my_list|random }}

Formatting and Number Filters

floatformat: Rounds a float to a specified number of decimal places.
{{ my_float|floatformat:2 }}

intcomma: Adds commas to an integer.

{{ 1000000|intcomma }} # Output: 1,000,000

linebreaks: Converts line breaks into <p> tags.

{{ my_text|linebreaks }}

pluralize: Adds "s" to a word if the variable is greater than 1.
{{ count }} item{{ count|pluralize }}

HTML Escaping and Formatting Filters

escape: Escapes HTML.
{{ my_html|escape }}

safe: Marks a string as safe, bypassing HTML escaping.
{{ my_html|safe }}

urlize: Converts URLs in plain text into clickable links.

{{ "Visit https://www.example.com"|urlize }}

Other Useful Filters

filesizeformat: Formats a number as a human-readable file size.
{{ 1024|filesizeformat }} # Output: 1 KB

dictsort: Sorts a list of dictionaries by a specified key.
{{ my_list|dictsort:"name" }}

unordered_list: Renders a list as an unordered HTML list.
{{ my_nested_list|unordered_list }}

Custom Template Filters:

A custom filter in Django is a user-defined template filter that you can create to apply specific, customized transformations to data within your templates. While Django provides a variety of built-in template filters, custom filters allow you to handle specialized data formatting or processing that built-in filters may not cover.

Filter chaining in Django templates refers to the practice of applying multiple filters sequentially to a variable. You can chain filters together by using multiple pipe symbols (|), with each filter transforming the output of the previous one.

Example of Filter Chaining:

Suppose you have a variable my_text with the value " hello WORLD ":

{{ my_text|trim|lower|capfirst }}

In this example:

1. trim removes any leading or trailing whitespace.
2. lower converts the text to lowercase.
3. capfirst capitalizes the first character.

The output would be:

arduino

"Hello world"

Benefits of Filter Chaining:

- Efficiency: Apply multiple transformations in a single line.
- Readability: Make templates cleaner by keeping formatting in one place.
- Flexibility: Combine various filters to achieve complex transformations without writing custom code.
- Filter chaining allows for efficient and readable data formatting directly within Django templates.

The order of filters in Django matters because each filter acts on the result of the previous one, creating a chain where each filter's output becomes the next filter's input. Changing the order can lead to different results, especially when filters affect data formatting, whitespace, case, or conditional handling.

Example of Filter Order Impact

Let's consider a variable my_text with the value " Django TEMPLATE FILTERS ":

django

Copy code

{{ my_text|lower|trim }}

In this case:

1. lower first converts the entire string to lowercase.

2. trim then removes any leading and trailing whitespace.

Result: "django template filters"

Now, if we reverse the order:

{{ my_text|trim|lower }}

Here:

1. trim first removes the whitespace around the string.
2. lower then converts the result to lowercase.

Result: "django template filters"

While the output may appear the same in this case, changing filter order can make a significant difference when specific transformations are required.

Why Filter Order Matters

1. Dependency on Previous Filters: Each filter acts on the transformed result of the previous one. Some filters may not behave as intended if they receive unexpected input.
2. Intended Output: Certain transformations must occur first for later filters to have the correct input.
3. Avoiding Errors: Filters like default or safe depend on the input's state, so placing them correctly is important to ensure no unintended results.

Filter order is thus essential to achieve the desired final output and avoid unexpected behavior in your templates.

Project - django-admin startproject FilterProject

Then cd FilterProject

App - filterapp

settings.py:

Import os

INSTALLED_APPS = [

```
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'filterapp',
]
TEMPLATES = [
    {
        'BACKEND': 'django.template.backends.django.DjangoTemplates',
        'DIRS': [os.path.join(BASE_DIR,'templates')],
        'APP_DIRS': True,
        'OPTIONS': {
            'context_processors': [
                'django.template.context_processors.debug',
                'django.template.context_processors.request',
                'django.contrib.auth.context_processors.auth',
                'django.contrib.messages.context_processors.messages',
            ],
        },
    },
]
```

Create templates folder in application.

Create templatetags folder in application.

In templatetags folder - customs_filter.py file:

```
from django  import template


import math

register=template.Library()


@register.filter

def add_hello(value):

    return f"hello {value}"

@register.filter

def square(value):

    return value ** 2

@register.filter

def reverse_string(value):

    return value [::-1]

@register.filter

def absolute(value):

    return  abs(value)

@register.filter
```

```
def even_or_odd(value):
    if value % 2 ==0:
        return "even"
    else:
        return "odd"
@register.filter
def factorial(value):
    return  math.factorial(value)
```

In templates folder create html file - templates.html file:

```
{% load custom_filters %}
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Document</title>
</head>
<body>
    <h2>Template Filters</h2>
    <p>{{name|add_hello}}</p>
    <p>Square value of 5{{5|square}}</p>
    <p>Reverse string of 'Django'{{'Django'|reverse_string}}</p>
    <p>Absolute of -7{{-7|absolute}}</p>
```

```
    <p>Even or odd for 8{{8|even_or_odd}}</p>
    <p>Factorial of 10{{10|factorial}}</p>
</body>
</html>
```

views.py file:

```
from django.shortcuts import render
def filter_view(request):
    return render(request,'template.html',{'name':'world'})
```

urls.py file in application:

```
from django.urls import path
from . import views
urlpatterns=[
    path('filters/',views.filter_view),
]
```

Project level urls.py file:

```
from django.contrib import admin
from django.urls import path,include
urlpatterns = [
    path('admin/', admin.site.urls),
    path('', include('filterapp.urls')),
]
```

Python manage.py runserver

Output: