1) Django Template Language (DTL)

Django doesn’t use normal Python syntax inside HTML — it uses its own “template language”.

Syntax examples

Loop

```
{% for item in sequence %}
    {{ item }}
{% endfor %}
```

Variable output

```
{{ item }}
```

---

2) Template Inheritance (Extending One File in Another)

We usually create `base.html` with common structure (header, footer, css links) and extend it.

base.html

```
<!DOCTYPE html>
<html>
<head>
    <title>My Site</title>
</head>
<body>
    {% block content %}{% endblock %}
</body>
</html>
```

child.html

```
{% extends 'base.html' %}

{% block content %}
```

```
    <h1>This is child page</h1>
{% endblock %}
```

This avoids repeating the same header/footer in every page.

---

3) Static Files in Django (CSS, JS, Images)

Settings in `settings.py`

```
STATIC_URL = '/static/'   # URL prefix to access static files

STATICFILES_DIRS = [
    os.path.join(BASE_DIR, 'static')   # Folder where *you* store
static files
]

STATIC_ROOT = os.path.join(BASE_DIR, 'assets')   # Folder where Django
collects files for production
```

Why do we need `assets` (STATIC_ROOT)?

- `static/` → during development (you work here)
- `collectstatic` copies everything into → `assets/` for deployment

You **cannot** serve files directly from your `static/` folder in production — Django needs a final collected folder (`assets/`) which a web server (Nginx, Apache) can serve.

Collect command (only before deployment)

```
python manage.py collectstatic
```

---

4) Using static files in templates

Step 1: Add at top of template

```
{% load static %}
```

Step 2: Use `{% static %}` instead of hardcoding paths

```
<link rel="stylesheet" href="{% static 'home.css' %}">
<script src="{% static 'script.js' %}"></script>
<img src="{% static 'images/logo.png' %}">
```

Without `{% static %}`, Django won't know the correct path.

📌 CRUD on a List inside Django Views — NOTES

---

1) Where is the "database" here?

Instead of using a real database or Django Model, we use:

```
products = [
    {'id': 1, 'name': 'product1', 'cost': 230},
    ...
]
```

This behaves like an **in-memory pseudo-database**.

- Data is lost on server restart
- No migrations, no ORM — just list operations

Useful for **learning CRUD concepts** without touching DB.

---

2) CREATE — Add new product (`products_view` POST logic)

```
if req.method == 'POST':
    p_name = req.POST.get('product_name')
    p_cost = req.POST.get('cost')
    id = len(products) + 1               # generate new ID
    new_prod = {'id': id, 'name': p_name, 'cost': p_cost}
    products.append(new_prod)
 return render(req, 'home.html', {'products': products})
```

Flow:

- Receives POST from form
- Extracts form data using `req.POST.get()`
- Creates new dict and appends to list
- Same function returns `render(...)` so UI reloads with new item (So also works for get)

---

3) Show all products in home.html

Template loops using:

```
 {% for product in products %}
    {{product.id}} {{product.name}} {{product.cost}}
{% endfor %}
```

---

4) DELETE — Remove by ID (`delete_product` view)

```
def delete_product(req, id):
    """
    Deletes a product from the global `products` list using its ID.
    Then reloads the home page with an updated list.
    """
    global products                          # required to reassign
    products = [p for p in products if p['id'] != id]   # filtering
    return redirect('/')#better than render- avoids double form submit
```

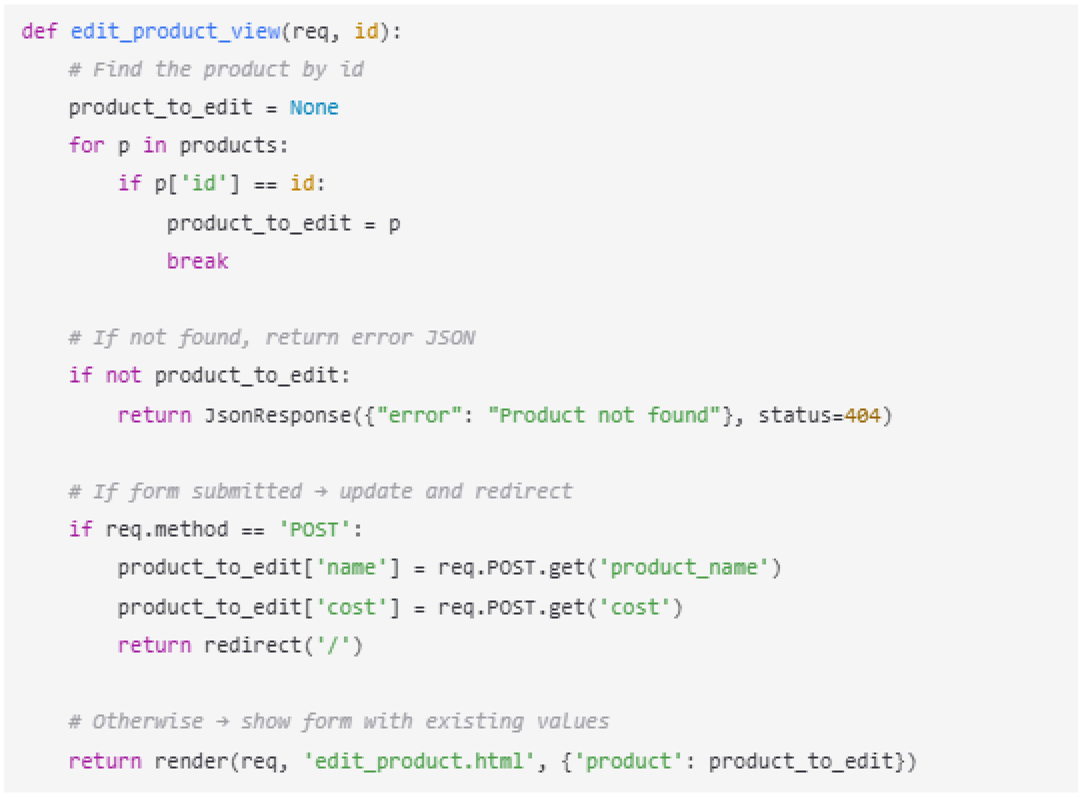
Why a new list?

You should NOT remove from a list **while iterating over it**.
Instead, create a *filtered* new list and assign back.

---

5) UPDATE — Edit existing product (`edit_product_view`)

Two parts: First task to get details of an individual product and second task is to edit the previous values (POST)



```python
def edit_product_view(req, id):
    # Find the product by id
    product_to_edit = None
    for p in products:
        if p['id'] == id:
            product_to_edit = p
            break

    # If not found, return error JSON
    if not product_to_edit:
        return JsonResponse({"error": "Product not found"}, status=404)

    # If form submitted → update and redirect
    if req.method == 'POST':
        product_to_edit['name'] = req.POST.get('product_name')
        product_to_edit['cost'] = req.POST.get('cost')
        return redirect('/')

    # Otherwise → show form with existing values
    return render(req, 'edit_product.html', {'product': product_to_edit})
```

Note: `redirect('/')` does a GET to home instead of rendering the edit page again — good UX.

---

6) URL Routing based on View Name + Params

`path('delete/<int:id>', views.delete_product, name='delete-product')`

Template:

`<a href="{% url 'delete-product' product.id %}">`

- Django resolves URL by **view name**, not raw path string
- `product.id` fills `<int:id> in the URL`

`<a href="/delete/{{product.id}}"><button>Delete</button></a>`

1. You can also write like this but the previous approach is much better.