

VOL 1

# Iniciando no **django**

BEATRIZ COSTA

# SUMÁRIO

## 01. Introdução ao Django (p. 5)

- Conceito (p. 6)
- Estrutura de um projeto (p. 7)
- Estrutura de um app (p. 8)
- O que faz cada parte do app (p. 9)
- 3 Boas Práticas Básicas (p. 10)

## 02. Model (p. 11)

- A Estrutura dos Dados (p. 12)
- CharField - Texto Curto (p. 13)
- TextField - Texto Longo (p. 13)
- DecimalField - Valores com Casas Decimais (p. 14)
- IntegerField - Números Inteiros (p. 14)
- BooleanField - Verdadeiro ou Falso (p. 15)
- DateField e DateTimeField - Datas (p. 15)

# SUMÁRIO

## 03. View (p. 16)

- A Lógica da Página (p. 17)
- O Que a View Faz? (p. 17)
- Exemplo Real: Listando Produtos (p. 18)
- Extra: exibindo um item específico (p. 19)

## 04. Template (p. 20)

- O HTML Dinâmico da Página (p. 21)
- Configurações (p. 23)
- Herança de Templates (p. 24)

## 05. Criando um CRUD: To-Do List (p. 26)

- Criando o Projeto (p. 28)
- Criando o App (p. 29)
- Model (p. 30)
  - Definindo o model (p. 31)

# SUMÁRIO

## 05. Criando um CRUD: To-Do List (p. 26)

- Template (p. 33)
  - Criando os Templates (p. 34)
  - HTML de Listar (p. 36)
  - HTML de Criar (p. 38)
  - HTML de Editar (p. 39)
- View (p. 40)
  - Criando as Regras de Lógica: Views (p. 41)
  - Listar tarefas (p. 42)
  - Criar tarefas (p. 43)
  - Editar tarefas (p. 44)
  - Excluir tarefas (p. 45)
- URL (p. 46)
  - URLs: Ligando tudo (p. 47)
  - Configurando as Rotas do App (p. 48)

## Agradecimentos (p. 49)

01

# Introdução ao Django

# INTRODUÇÃO AO DJANGO

## Conceito

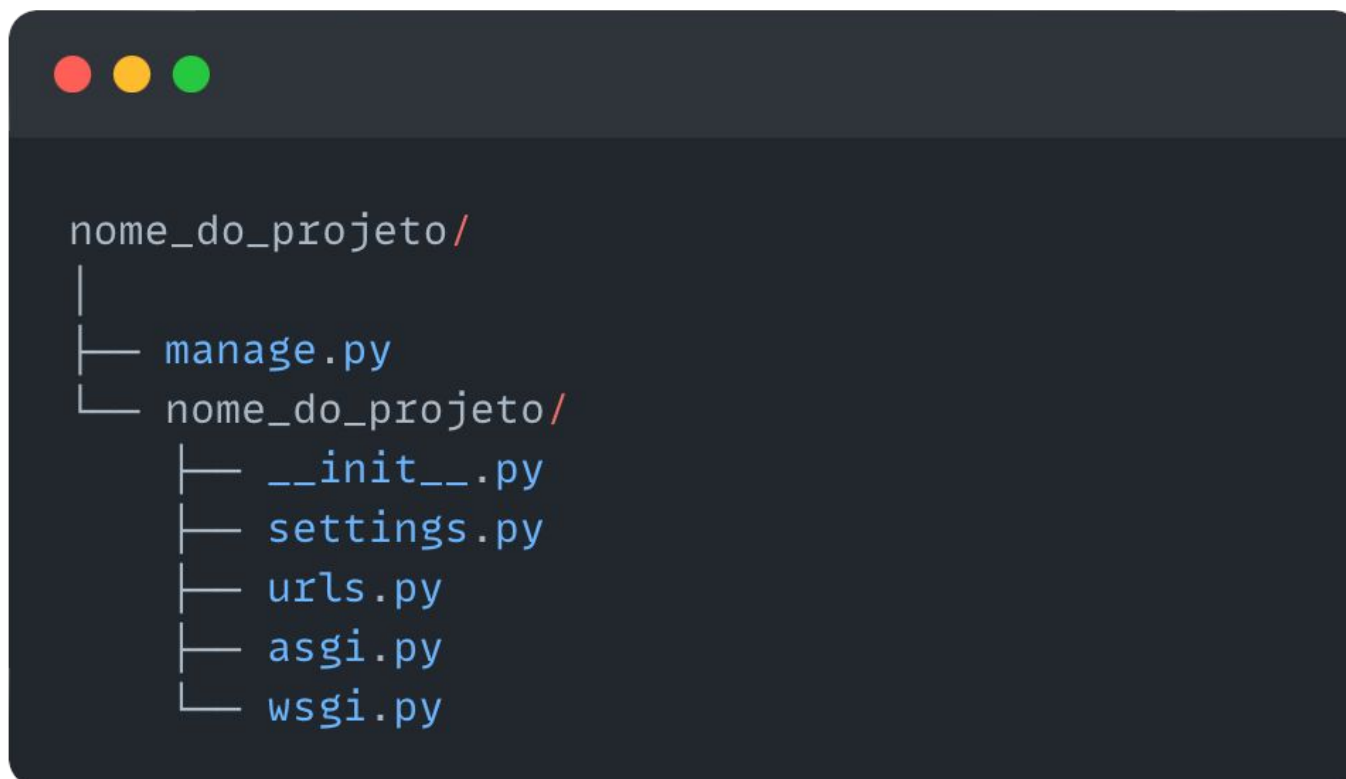
Django é um framework web em Python que ajuda você a criar aplicações web de forma rápida, segura e escalável. Ele segue o padrão **MVC (Model-View-Controller)**, mas com nomes ligeiramente diferentes: **Model-Template-View**.

- **Model:** representa os dados (estrutura das tabelas).
- **View:** processa a lógica e responde às requisições.
- **Template:** cuida do que o usuário vê (HTML dinâmico).

# INTRODUÇÃO AO DJANGO

## Estrutura de um projeto

Quando você cria um projeto Django, o comando ***django-admin startproject nomedoprojeto*** gera uma estrutura padrão com os arquivos mostrados abaixo.



```
nome_do_projeto/  
├── manage.py  
└── nome_do_projeto/  
    ├── __init__.py  
    ├── settings.py  
    ├── urls.py  
    ├── asgi.py  
    └── wsgi.py
```

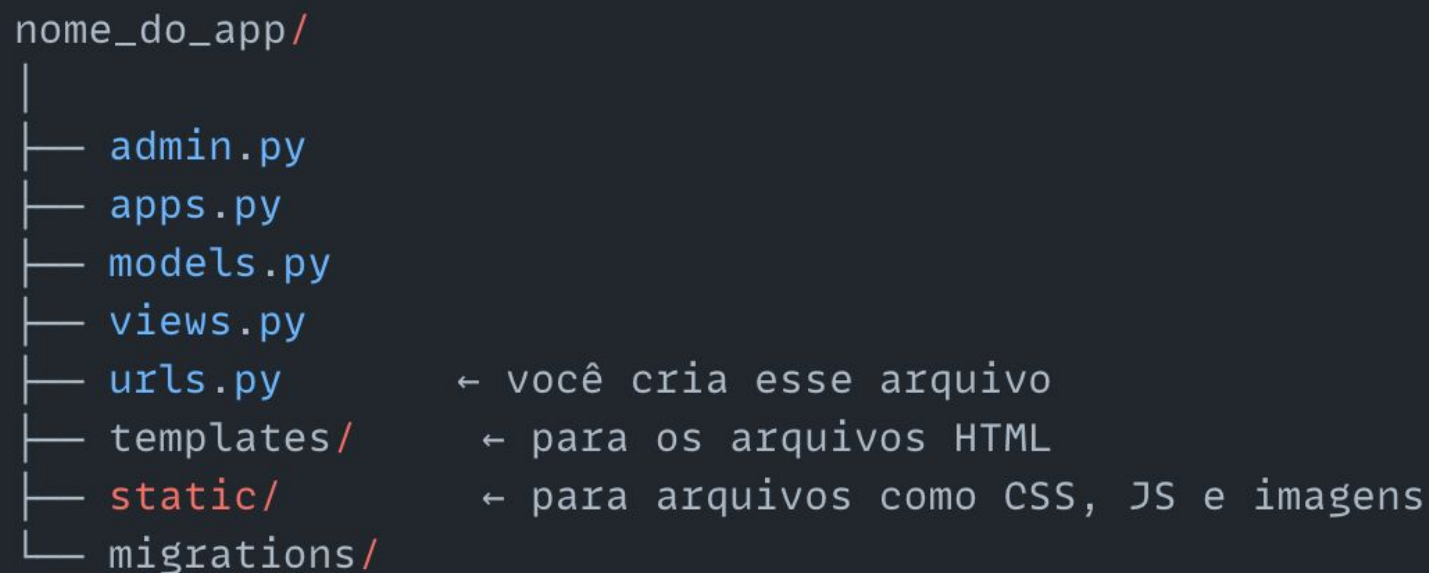
### O que cada parte faz?

- ***manage.py***: Comando principal para rodar o servidor, migrar dados, criar apps etc.
- ***settings.py***: Configurações gerais do projeto (apps instalados, banco de dados, templates, etc).
- ***urls.py***: Define as rotas principais do projeto (o caminho para as páginas).
- ***asgi.py*** e ***wsgi.py***: Arquivos para o servidor, usados em produção.

# INTRODUÇÃO AO DJANGO

## Estrutura de um app

Quando você cria um app dentro do projeto Django, o comando ***python manage.py startapp nomedoapp*** gera uma estrutura padrão para cada app criado com os seguintes arquivos:



```
nome_do_app/  
|  
|— admin.py  
|— apps.py  
|— models.py  
|— views.py  
|— urls.py      ← você cria esse arquivo  
|— templates/   ← para os arquivos HTML  
|— static/      ← para arquivos como CSS, JS e imagens  
|— migrations/
```

Essa separação torna o projeto **modular e organizado**, facilitando manutenção e crescimento.



# INTRODUÇÃO AO DJANGO

## Estrutura de um app

### O que faz cada parte do app

- ***admin.py***: registra os modelos para o painel administrativo.
- ***apps.py***: configurações do app.
- ***models.py***: onde ficam as classes que representam o banco de dados (Model).
- ***views.py***: onde escrevemos as funções ou classes que controlam o que é exibido ao usuário.
- ***urls.py***: você cria esse arquivo para organizar as rotas do seu app.
- ***templates/***: pasta onde ficam os arquivos HTML que formam o front-end (Template).
- ***static/***: arquivos estáticos (imagens, CSS, JS etc.).
- ***migrations/***: histórico das mudanças no banco de dados com base nos seus models.

# INTRODUÇÃO AO DJANGO

## 3 Boas Práticas Básicas

### 1. Organize por App

Divida sua aplicação em apps por domínio funcional: ex. usuarios, produtos, tarefas. Isso facilita a manutenção e o reuso de código.

### 2. Separe templates e estáticos por app

Exemplo:

A terminal window with a dark background and three colored window control buttons (red, yellow, green) in the top-left corner. It displays two lines of text: 'tarefas/templates/' and 'tarefas/static/'.

```
tarefas/templates/  
tarefas/static/
```

Evita conflito de nomes e melhora a organização.

### 3. Use URLs nomeadas

Ao definir rotas, sempre use ***name="alguma\_coisa"***. Isso permite usar ***{% url 'alguma\_coisa' %}*** nos templates e evita quebras se a rota mudar.


02

# Model

# MODEL

## A Estrutura dos Dados

O **Model** define a estrutura do banco de dados. É onde você cria as "tabelas" usando Python.

A screenshot of a code editor window with a dark background. The window has a title bar with three colored circles (red, yellow, green) on the left and a tab labeled 'models.py' with a Python logo icon. The code inside the editor is a Django model definition for a class named 'Produto'. It inherits from 'models.Model'. The class has three attributes: 'nome' of type 'CharField' with 'max\_length=100', 'preco' of type 'DecimalField' with 'max\_digits=6' and 'decimal\_places=2', and 'em\_estoque' of type 'BooleanField' with 'default=True'.

```
class Produto(models.Model):  
    nome = models.CharField(max_length=100)  
    preco = models.DecimalField(max_digits=6, decimal_places=2)  
    em_estoque = models.BooleanField(default=True)
```

Esse código cria uma tabela chamada **Produto** com campos de nome, preço e estoque.

No Django, os **fields** (campos) definem os tipos de dados que serão armazenados no banco de dados através dos Models. Cada campo representa uma coluna em uma tabela, com um tipo específico (texto, número, data etc.). A seguir, veja os principais fields e como usá-los.

# ENTENDENDO OS FIELDS NO DJANGO


## CharField – Texto Curto



```
nome = models.CharField(max_length=100)
```

Usado para armazenar strings pequenas, como nomes, títulos ou cidades. ***max\_length*** é obrigatório e define o número máximo de caracteres.

## TextField – Texto Longo



```
descricao = models.TextField()
```

Ideal para descrições ou textos grandes, como posts de blog. Não precisa de ***max\_length***.

# ENTENDENDO OS FIELDS NO DJANGO

## DecimalField – Valores com Casas Decimais

```
preco = models.DecimalField(max_digits=6, decimal_places=2)
```

Usado para preços, notas ou medidas exatas.

***max\_digits***: número total de dígitos.

***decimal\_places***: quantas casas decimais.

## IntegerField – Números Inteiros

```
estoque = models.IntegerField()
```

Guarda números sem casas decimais, como idade, quantidade, estoque.

# ENTENDENDO OS FIELDS NO DJANGO

## BooleanField – Verdadeiro ou Falso

```
ativo = models.BooleanField(default=True)
```

Armazena um valor True ou False, como se um produto está ativo ou não.

## DateField e DateTimeField – Datas

```
# Apenas data
data_publicacao = models.DateField(auto_now_add=True)

# Data + hora
criado_em = models.DateTimeField(auto_now_add=True)
```

Usados para armazenar datas ou data e hora juntas.

***auto\_now\_add = True***: define a data na criação.

***auto\_now = True***: atualiza sempre que o objeto for salvo.





# View



# VIEW

## A Lógica da Página

A **View** recebe requisições e retorna respostas. Pode buscar dados do banco e enviá-los para o HTML.

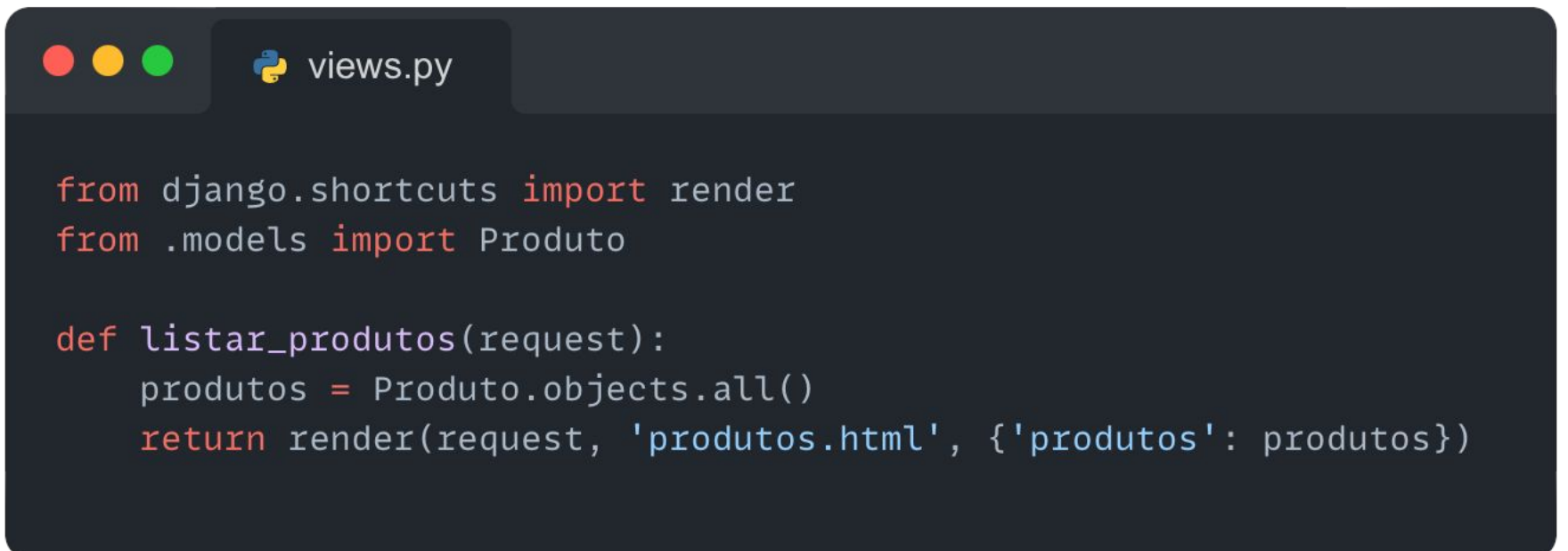
Ela funciona como o “meio do caminho” entre o que o usuário pede e o que o sistema entrega. A View pode buscar informações no banco de dados (através dos Models), organizar os dados e passá-los para os Templates (a parte visual do site, o HTML).

### O Que a View Faz?

- Recebe a **requisição HTTP** do usuário (como quando ele acessa uma página).
- Executa alguma **lógica de negócio**, como buscar itens no banco.
- Retorna uma **resposta**, geralmente um HTML.

# VIEW

## Exemplo Real: Listando Produtos

A screenshot of a code editor window with a dark theme. The title bar shows three colored circles (red, yellow, green) and a tab labeled 'views.py' with a Python logo. The code is written in a monospaced font with syntax highlighting: keywords are in red, identifiers in blue, and strings in green. The code defines a function 'listar\_produtos' that takes a 'request' object, queries all 'Produto' objects from the database, and renders a template named 'produtos.html' with the query results.

```
from django.shortcuts import render
from .models import Produto

def listar_produtos(request):
    produtos = Produto.objects.all()
    return render(request, 'produtos.html', {'produtos': produtos})
```

### Explicando:

- ***request***: É o que chega do navegador (como "me mostra a página /produtos").
- ***Produto.objects.all()***: Consulta todos os produtos no banco de dados.
- ***render()***: Monta a resposta, unindo os dados com um HTML (template).
- ***'produtos.html'***: É o arquivo HTML que vai ser exibido.
- ***{'produtos': produtos}***: Envia os dados para o template com a variável produtos.

# VIEW

Extra: exibindo um item específico

A screenshot of a code editor window with a dark theme. The window has a title bar with three colored circles (red, yellow, green) and a tab labeled 'views.py' with a Python logo. The code inside is a Django view function named 'detalhe\_produto' that takes 'request' and 'id' as arguments. It uses 'Produto.objects.get(id=id)' to fetch a product and 'render' to return a response with the template 'detalhe.html' and the product data.

```
def detalhe_produto(request, id):  
    produto = Produto.objects.get(id=id)  
    return render(request, 'detalhe.html', {'produto': produto})
```

Essa View mostra apenas um produto, com base no ID enviado pela URL.

04

# Template

# TEMPLATE

## O HTML Dinâmico da Página

O Template é o HTML que recebe os dados da view e os mostra ao usuário.

No Django, o **Template** é a **parte visual da aplicação** — o que o usuário realmente vê na tela. Ele é baseado em HTML, mas com um “superpoder”: pode receber dados dinâmicos vindos das Views e exibi-los de forma automática.

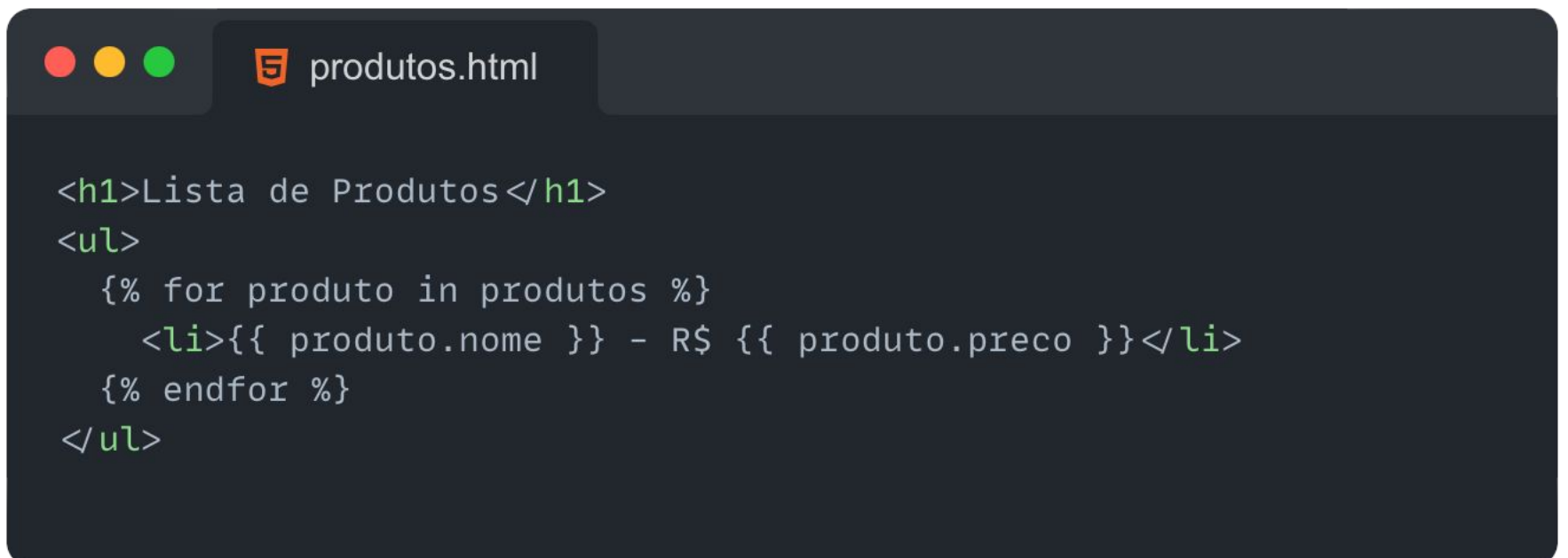
Isso significa que você pode montar uma página que muda o conteúdo dependendo dos dados do sistema, como uma lista de produtos, um nome de usuário logado, ou uma mensagem personalizada.

### Como funciona?

- A **View** envia os dados para o Template.
- O Template usa a linguagem do Django Template Language (DTL) para **mostrar esses dados**.
- O que era só HTML vira uma **página dinâmica**.

# TEMPLATE

## O HTML Dinâmico da Página

A screenshot of a code editor window titled 'produtos.html'. The editor has a dark background and shows Django template code. The code includes an opening h1 tag, an opening ul tag, a for loop that iterates over 'produtos', and a closing li tag that displays the product name and price. The code is as follows:

```
<h1>Lista de Produtos</h1>
<ul>
  {% for produto in produtos %}
    <li>{{ produto.nome }} - R$ {{ produto.preco }}</li>
  {% endfor %}
</ul>
```

### Explicando:

- ***{% for produto in produtos %}***: É um loop que percorre cada item da lista produtos enviada pela View.
- ***{{ produto.nome }}***: Exibe o valor do campo 'nome' do produto.
- ***{{ produto.preco }}***: Monta a resposta, unindo os dados com um HTML (template).

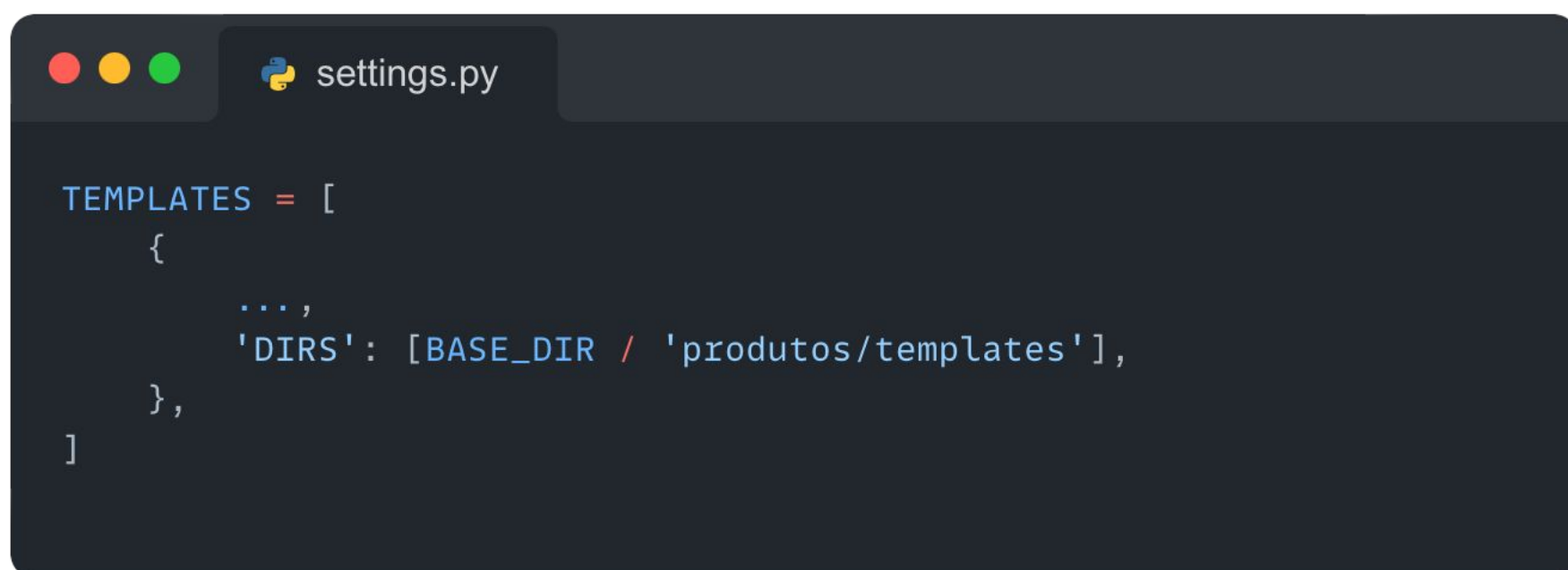
# TEMPLATE

## Configurações

### Onde ficam os templates?

Por padrão, você cria uma pasta chamada 'templates' dentro do seu app (ex: produtos/templates/) e coloca os arquivos .html dentro.

Para garantir que o Django encontre os templates, no **settings.py**, adicione a localização (path/caminho) dos seus arquivos .html, como o exemplo abaixo:

A screenshot of a code editor window titled 'settings.py'. The editor shows a Python list named 'TEMPLATES' containing a dictionary. The dictionary has a key 'DIRS' with a value that is a list containing the path '[BASE\_DIR / 'produtos/templates']'.

```
TEMPLATES = [  
    {  
        ...,  
        'DIRS': [BASE_DIR / 'produtos/templates'],  
    },  
]
```

# TEMPLATE

## Herança de Templates

No Django, usamos herança de templates para reaproveitar a estrutura HTML entre páginas diferentes. Isso evita repetir cabeçalho, rodapé, menu etc.

### Como funciona

1. Criamos um arquivo base (ex: *base.html*).
2. As páginas herdam esse arquivo com ***{% extends 'base.html' %}***.
3. Cada parte personalizável fica dentro de um ***{% block %}***.

A screenshot of a code editor window with a dark theme. The title bar shows three colored window control buttons (red, yellow, green) and a tab labeled 'base.html'. The code is written in a light green monospace font on a dark background. It is an HTML template for a blog, featuring Django template tags for block inheritance and content insertion. The structure includes a DOCTYPE declaration, html, head, and body tags. The head contains a title block. The body contains a main heading, a navigation menu, and a content block.

```
<!DOCTYPE html>
<html>
<head>
  <title>{% block title %}Blog{% endblock %}</title>
</head>
<body>
  <h1>Meu Blog</h1>
  <nav><a href="/">Início</a> | <a href="/posts/">Posts</a></nav>
  <hr>

  {% block content %}
  {% endblock %}
</body>
</html>
```



# TEMPLATE

## Herança de Templates

No exemplo a seguir, queremos estender a página `list.html` a partir do `base.html` para reaproveitar o título da aba e a navbar. Veja abaixo como ficaria o código de `list.html`.

A screenshot of a code editor window with a dark theme. The title bar shows three colored window control buttons (red, yellow, green) and a tab labeled 'list.html' with a small icon. The code is written in a light gray font on a dark background. It shows Django template syntax for inheritance and block rendering. The code includes tags for extending a base template, defining a title block, and a content block that contains HTML for a list of posts or a message if no posts are found.

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

{% block title %}Lista de Posts{% endblock %}

{% block content %}
<h2>Últimos Posts</h2>
<ul>
  {% for post in posts %}
    <li>{{ post.titulo }}</li>
  {% empty %}
    <li>Nenhum post encontrado.</li>
  {% endfor %}
</ul>
{% endblock %}
```

### O que cada tag faz

- **`{% extends 'base.html' %}`** - Herda o layout principal do projeto (como cabeçalho, rodapé, navbar etc.)
- **`{% block title %}`** - Define o título da aba do navegador para a página atual
- **`{% block content %}`** - Define o conteúdo principal da página que será substituído em cada template filho

05

# Criando um CRUD: To-Do List

# CRIANDO UM CRUD: TO-DO LIST

## O HTML Dinâmico da Página

Um CRUD (Create, Read, Update, Delete) é a base de qualquer aplicação que salva dados. Vamos criar um sistema de **tarefas** com Django onde o usuário pode:

- Criar novas tarefas
- Ver a lista de tarefas
- Editar uma tarefa
- Apagar tarefas

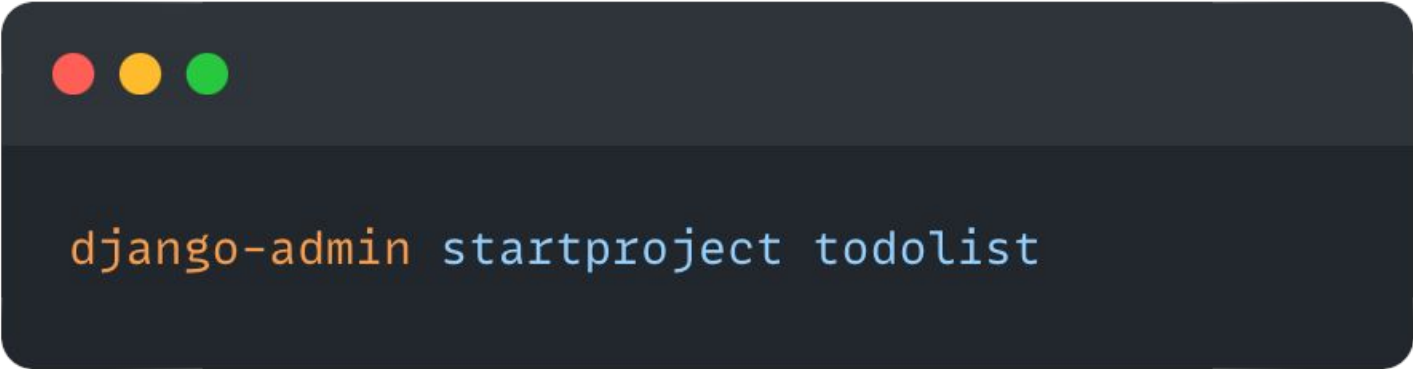
# CRIANDO UM CRUD: TO-DO LIST

## Criando o Projeto

É aconselhável já abrir a sua IDE de preferência, para seguir com os próximos passos, aconselho o VS Code.

### Criando o projeto

Para criar o projeto, abra o terminal e digite:

A terminal window with a dark background and three colored window control buttons (red, yellow, green) in the top-left corner. The command `django-admin startproject todoist` is entered in a light blue monospace font.

```
django-admin startproject todoist
```

Para abrir o projeto, digite no terminal:

A terminal window with a dark background and three colored window control buttons (red, yellow, green) in the top-left corner. The command `cd todoist` is entered in a light blue monospace font.

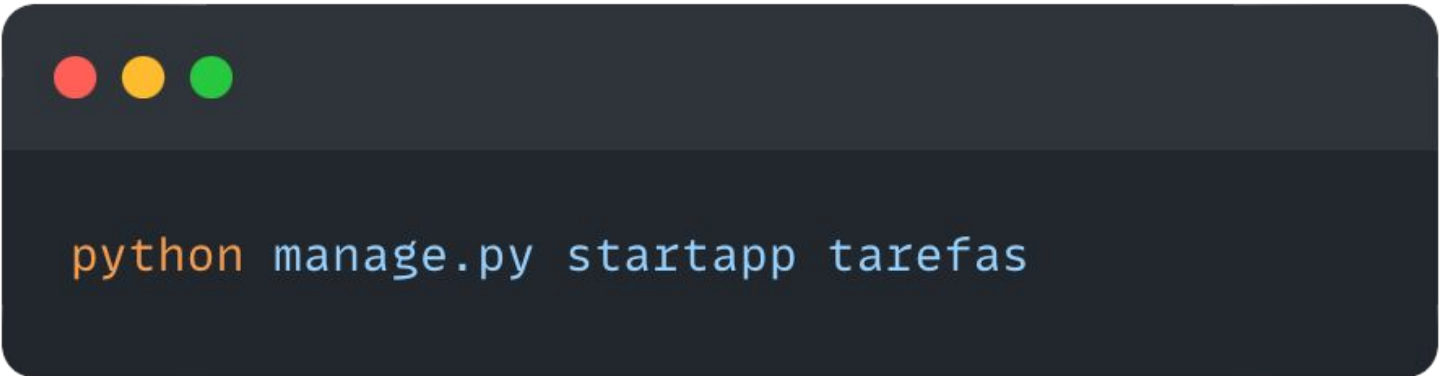
```
cd todoist
```

# CRIANDO UM CRUD: TO-DO LIST

## Criando o App

### Criando o app

Para criar o app, digite no terminal:



```
python manage.py startapp tarefas
```

Registre o app em todolist/settings.py:



```
INSTALLED_APPS = [  
    ...,  
    'tarefas',  
]
```



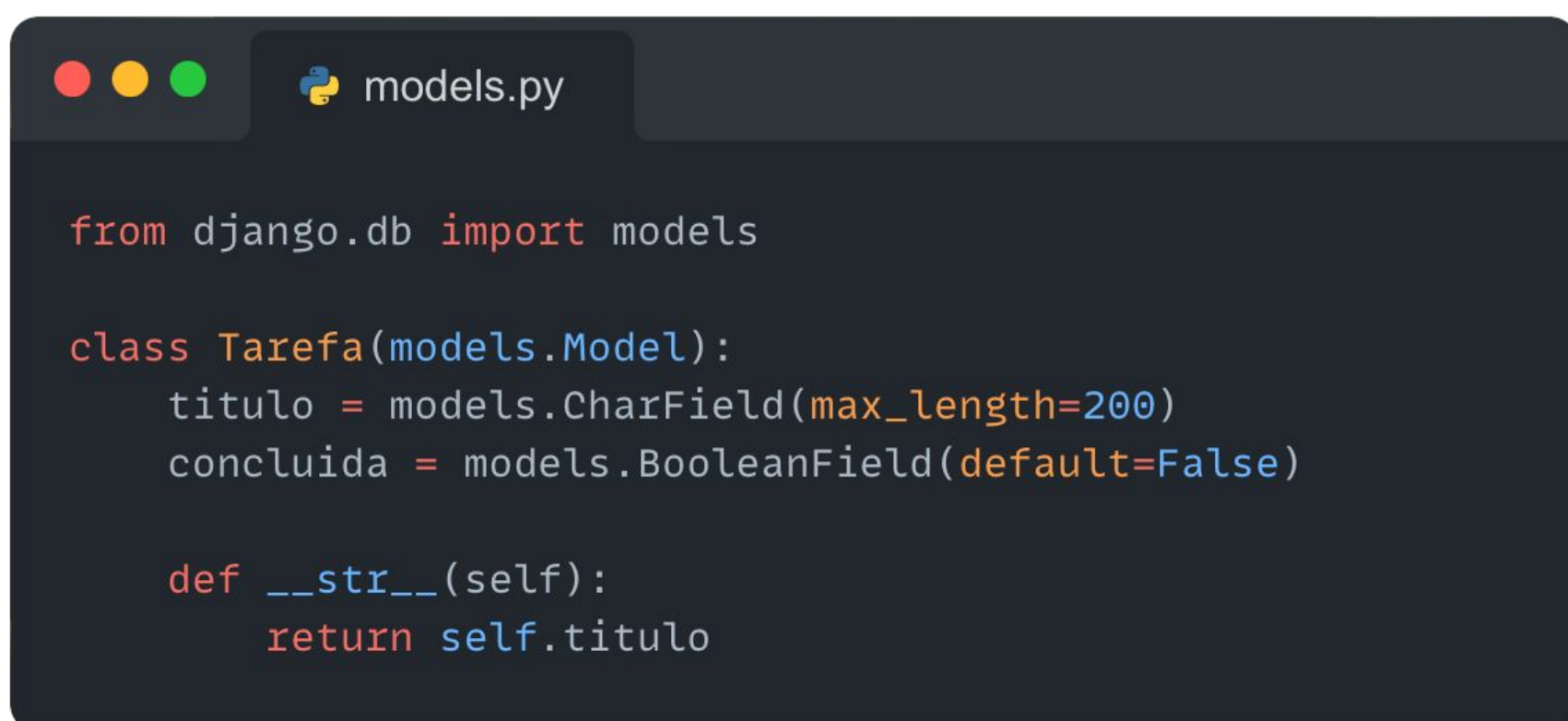
**05**

# Criando um CRUD: To-Do List Model

# CRIANDO UM CRUD: TO-DO LIST

## Definindo o model

O model define a **estrutura da tabela** no banco. Vamos criar uma tabela chamada Tarefa, com campos de título e status. Para isso, abra o arquivo **models.py** dentro da pasta tarefas.



```
from django.db import models

class Tarefa(models.Model):
    titulo = models.CharField(max_length=200)
    concluida = models.BooleanField(default=False)

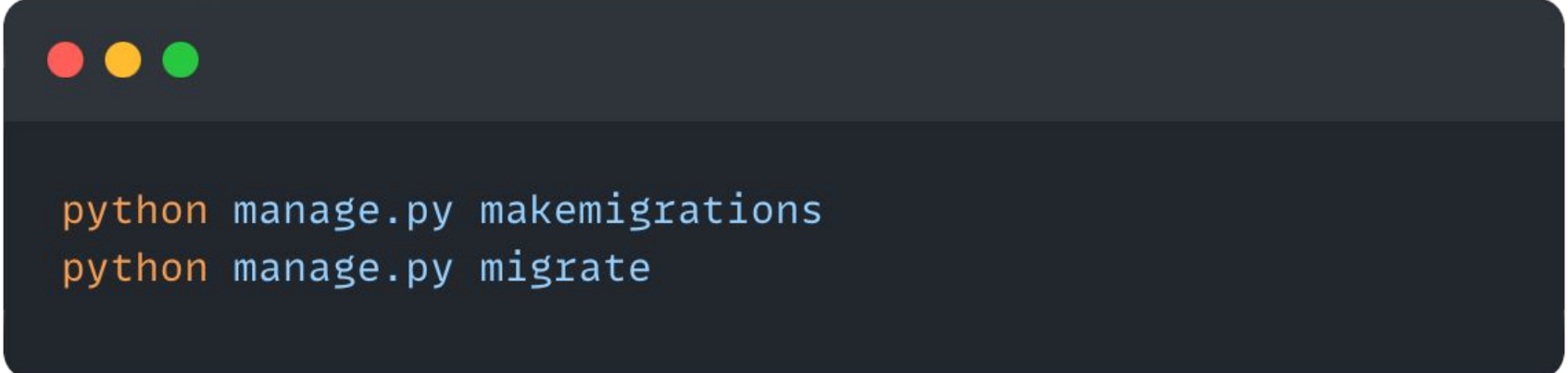
    def __str__(self):
        return self.titulo
```

Esse código cria uma tabela **Tarefa** com dois campos: 'titulo' (texto curto) e 'concluida' (boolean - verdadeiro ou falso).

# CRIANDO UM CRUD: TO-DO LIST

## Definindo o model

Depois, rode no terminal:

A terminal window with a dark background and three colored window control buttons (red, yellow, green) in the top-left corner. It contains two lines of code in a monospaced font.

```
python manage.py makemigrations  
python manage.py migrate
```

O comando ***makemigrations*** registra as mudanças feitas nos models e o ***migrate*** aplica essas mudanças no banco de dados.



05

# Criando um CRUD: To-Do List Template

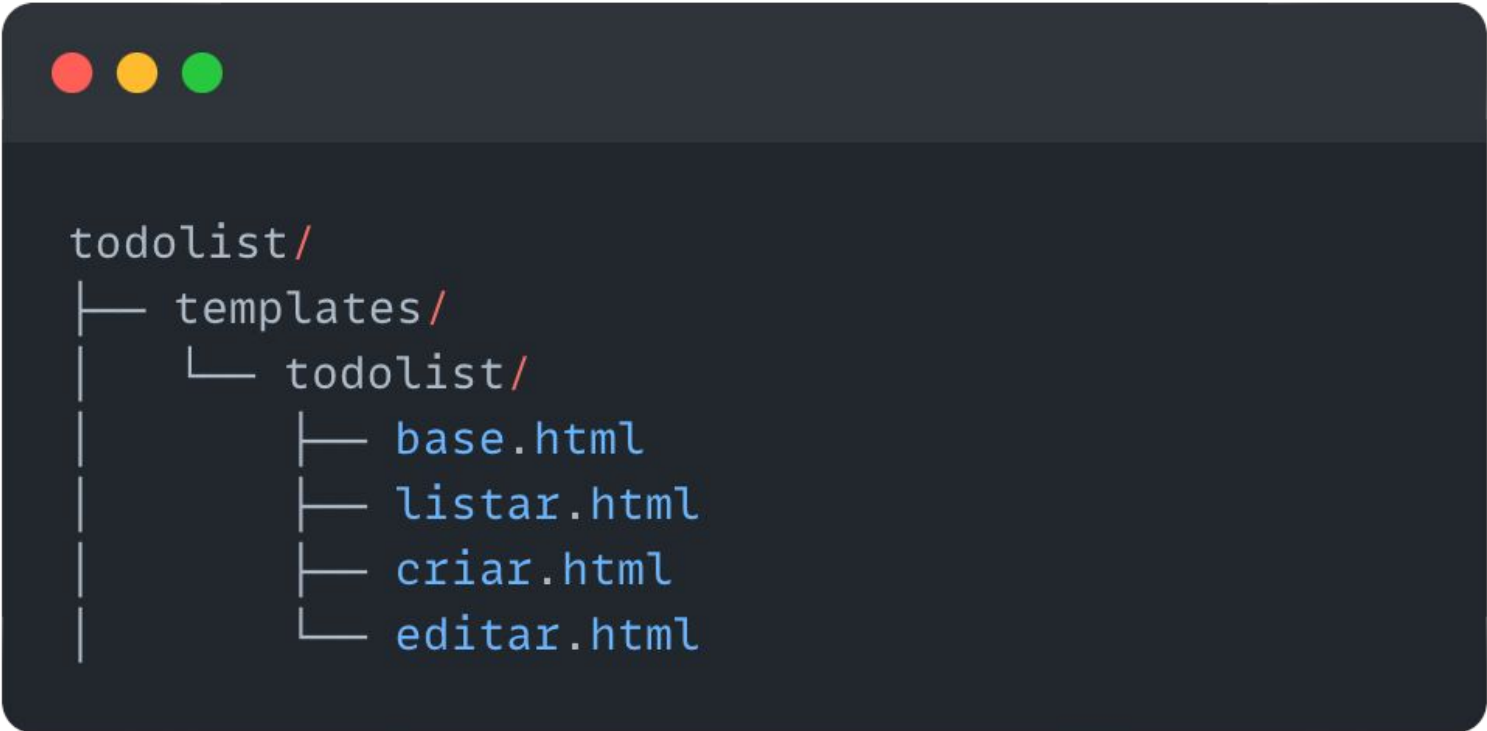
# CRIANDO UM CRUD: TO-DO LIST

## Criando os Templates

Os templates são os arquivos *.html* que formam o visual da nossa aplicação. No Django, usamos o recurso de **herança de templates** para reaproveitar estruturas como navbar, títulos e rodapés. Vamos criar primeiro um layout base.

### Estrutura recomendada:

Coloque os arquivos HTML dentro de uma pasta chamada *templates/todolist/*, assim:



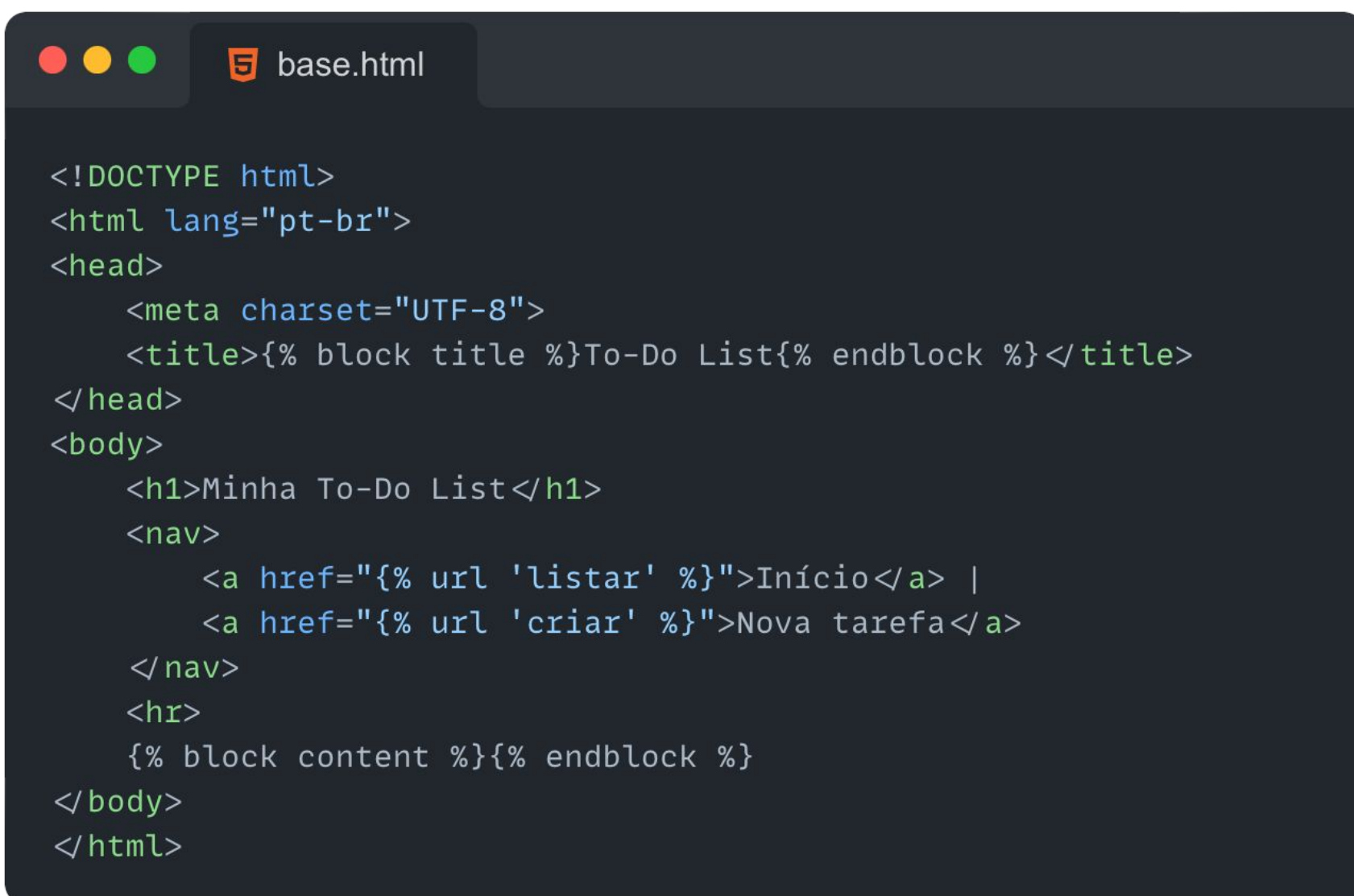
```
todolist/  
├── templates/  
│   └── todolist/  
│       ├── base.html  
│       ├── listar.html  
│       ├── criar.html  
│       └── editar.html
```

O diagrama mostra a estrutura de arquivos de templates para o Django. A pasta principal é `todolist/`. Dentro dela, há uma subpasta `templates/`. Dentro de `templates/`, há uma subpasta `todolist/`. Dentro de `todolist/`, há quatro arquivos HTML: `base.html`, `listar.html`, `criar.html` e `editar.html`.

# CRIANDO UM CRUD: TO-DO LIST

## Criando os Templates

Primeiramente, vamos criar o arquivo `base.html`, que será o layout base para outras páginas.

A screenshot of a code editor window with a dark theme. The title bar shows three colored circles (red, yellow, green) and a tab labeled 'base.html'. The code is written in a light green font on a dark background. It is an HTML template for a To-Do List application, using Django's template syntax with curly braces for variables and double braces for blocks. The code defines the document type, language, charset, title, and body structure, including a navigation bar with links for 'listar' and 'criar' tasks.

```
<!DOCTYPE html>
<html lang="pt-br">
<head>
    <meta charset="UTF-8">
    <title>{% block title %}To-Do List{% endblock %}</title>
</head>
<body>
    <h1>Minha To-Do List</h1>
    <nav>
        <a href="{% url 'listar' %}">Início</a> |
        <a href="{% url 'criar' %}">Nova tarefa</a>
    </nav>
    <hr>
    {% block content %}{% endblock %}
</body>
</html>
```

### O que esse código faz:

- Os links usam `{% url 'listar' %}` e `{% url 'criar' %}`, que devem existir no `urls.py`.
- Exibe links de navegação para listar e criar tarefas.
- Usa `{% block title %}` para permitir que cada página defina seu próprio `<title>`.

# CRIANDO UM CRUD: TO-DO LIST

## Criando os Templates

### HTML de Listar

Mostra a lista de tarefas salvas.

```
listar.html

{% extends 'base.html' %}

{% block title %}Lista de Tarefas{% endblock %}

{% block content %}
<h2>Lista de Tarefas</h2>

{% if tarefas %}
<ul>
    {% for tarefa in tarefas %}
    <li>
        {% if tarefa.concluida %}
        <s>{{ tarefa.titulo }}</s>
        {% else %}
        {{ tarefa.titulo }}
        {% endif %}
        - <a href="{% url 'editar' tarefa.id %}">Editar</a>
        - <a href="{% url 'excluir' tarefa.id %}">Excluir</a>
    </li>
    {% endfor %}
</ul>
{% else %}
<p>Nenhuma tarefa cadastrada.</p>
{% endif %}
{% endblock %}
```

### O que esse código faz:

- Lista todas as tarefas usando for.
- Mostra o título riscado se a tarefa estiver concluída (<s>).

# CRIANDO UM CRUD: TO-DO LIST

## Criando os Templates

### HTML de Listar

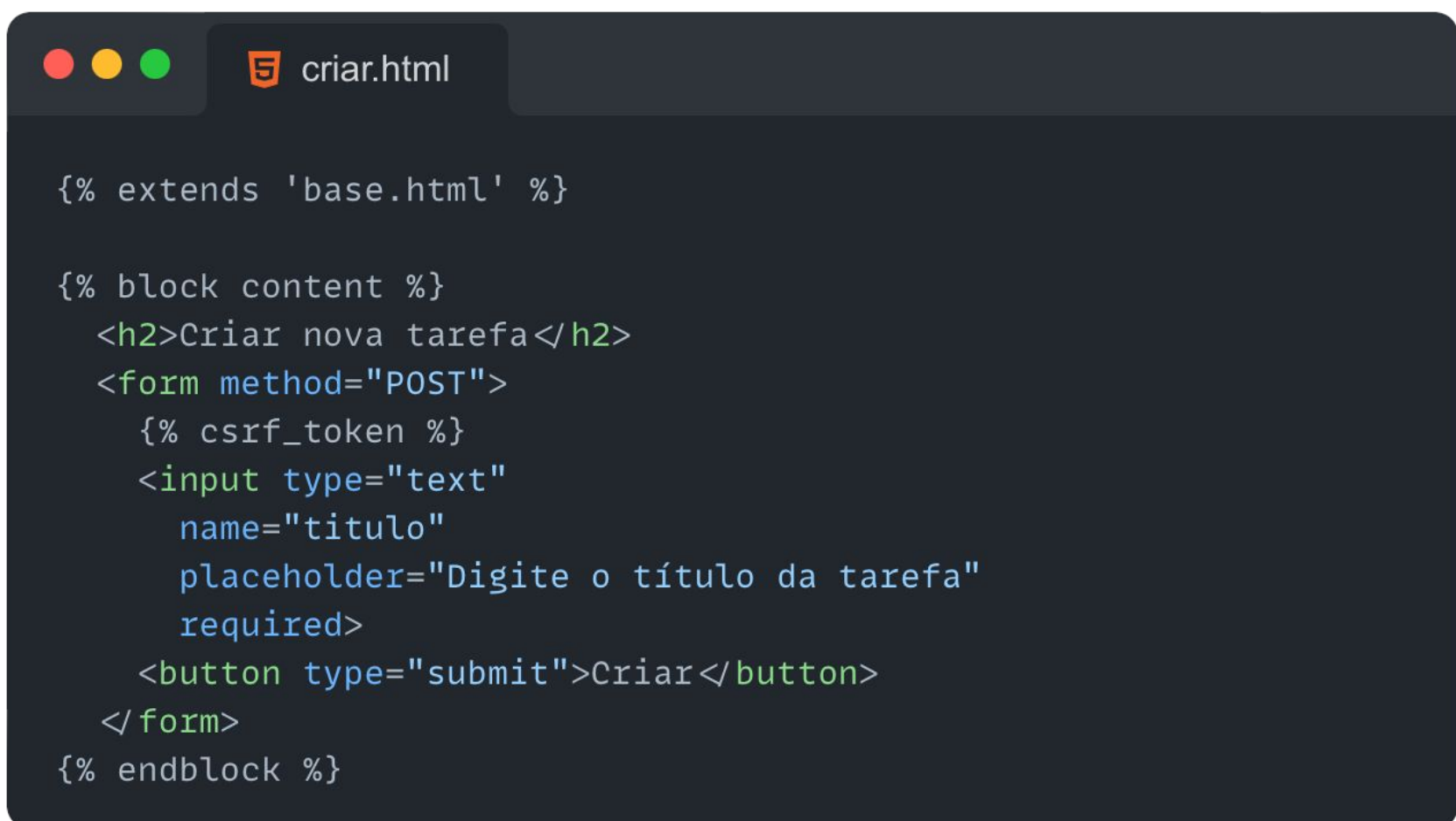
- Os links de editar e excluir funcionam com URLs nomeadas: *editar* e *excluir*.
- ***{% for tarefa in tarefas %}*** - Percorre a lista de tarefas trazida pela view e exibe cada uma em um `<li>`.
- ***{% url 'editar\_tarefa' tarefa.id %}*** - Gera o link para a página de edição daquela tarefa usando a URL nomeada *editar\_tarefa*.
- ***{% empty %}*** - Mostra uma mensagem caso não existam tarefas cadastradas.

# CRIANDO UM CRUD: TO-DO LIST

## Criando os Templates

### HTML de Criar

Formulário para adicionar uma nova tarefa.

A screenshot of a code editor window titled 'criar.html'. The editor has a dark background and shows Django template code. The code includes a block for content, an h2 tag 'Criar nova tarefa', a form with method 'POST', a CSRF token, a text input for 'titulo' with a placeholder 'Digite o título da tarefa' and 'required' attribute, and a submit button labeled 'Criar'.

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

{% block content %}
    <h2>Criar nova tarefa</h2>
    <form method="POST">
        {% csrf_token %}
        <input type="text"
            name="titulo"
            placeholder="Digite o título da tarefa"
            required>
        <button type="submit">Criar</button>
    </form>
{% endblock %}
```

O que esse código faz:

- ***<form method="post">*** - Envia os dados digitados pelo usuário via método POST (útil para criação/edição).
- ***{% csrf\_token %}*** - Protege o formulário contra ataques CSRF (boas práticas de segurança no Django).
- ***name="titulo"*** é essencial para funcionar com ***request.POST.get('titulo')***

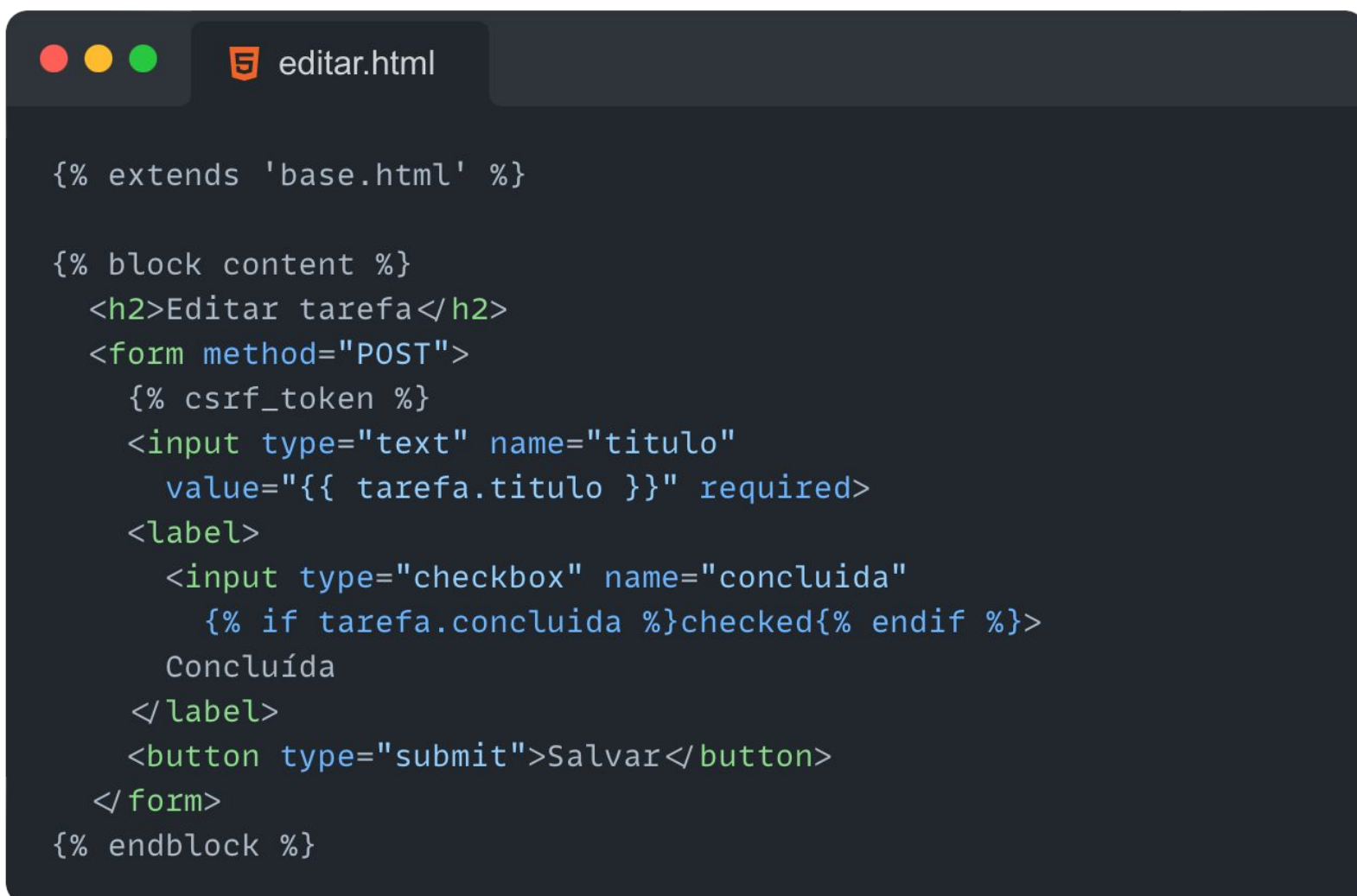


# CRIANDO UM CRUD: TO-DO LIST

## Criando os Templates

### HTML de Editar

Formulário para editar uma tarefa existente.

A screenshot of a code editor window titled 'editar.html'. The editor shows Django template code for editing a task. The code includes a form with a text input for the title, a checkbox for 'concluída' (completed), and a 'Salvar' (Save) button. The checkbox is checked if the task is already completed. The code uses Django's {% csrf\_token %} and {% if %} tags for security and conditional rendering.

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

{% block content %}
<h2>Editar tarefa</h2>
<form method="POST">
  {% csrf_token %}
  <input type="text" name="titulo"
    value="{{ tarefa.titulo }}" required>
  <label>
    <input type="checkbox" name="concluída"
      {% if tarefa.concluída %}checked{% endif %}>
    Concluída
  </label>
  <button type="submit">Salvar</button>
</form>
{% endblock %}
```

### O que esse código faz:

- Estrutura praticamente igual à de *criar.html*, mas a **view envia os dados da tarefa já existente para serem editados**.
- O checkbox de conclusão já aparece marcado se a tarefa estiver concluída.
- Quando o botão “Salvar” é clicado, o Django atualiza a tarefa no banco de dados.

**05**

# Criando um CRUD: To-Do List **View**

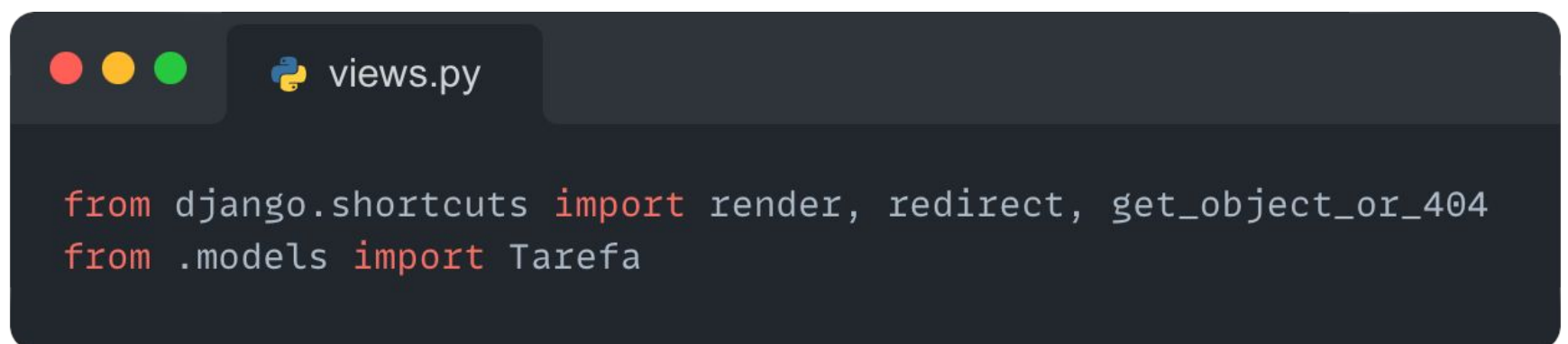


# CRIANDO UM CRUD: TO-DO LIST

## Criando as Regras de Lógica: Views

As views são responsáveis por **buscar, salvar, editar e apagar** as tarefas. Cada ação vai ser definida por uma função. Veja abaixo como criar cada uma no mesmo arquivo **views.py** na pasta tarefas.

Primeiramente, vamos fazer as importações:

A screenshot of a code editor window with a dark theme. The title bar shows three colored circles (red, yellow, green) and a tab labeled 'views.py' with a Python logo. The code inside the editor is:

```
from django.shortcuts import render, redirect, get_object_or_404
from .models import Tarefa
```

### O que cada uma faz:

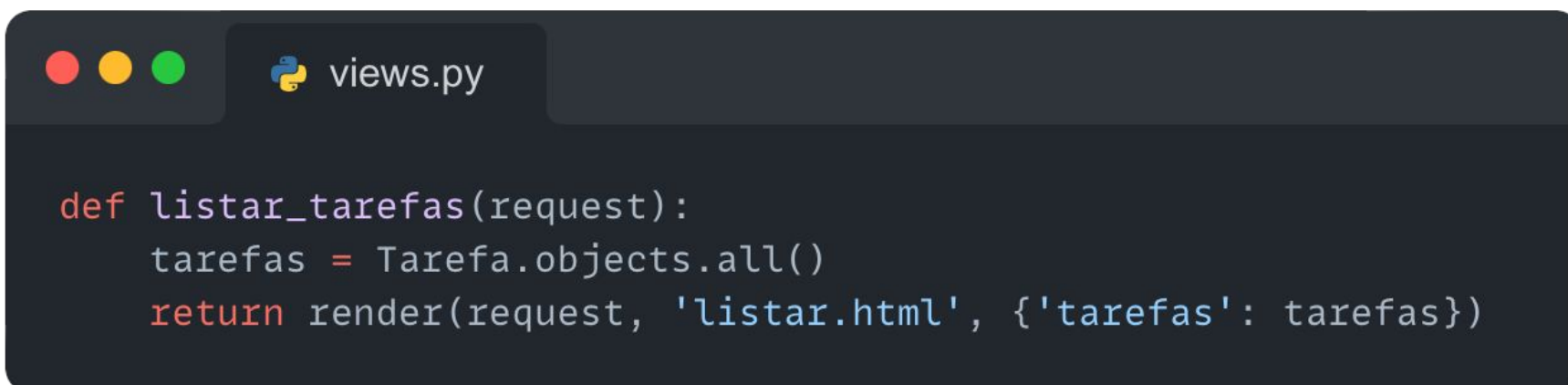
- ***render***: Carrega um template e envia dados para ele.
- ***redirect***: Redireciona o usuário para outra rota (por exemplo, após salvar uma tarefa).
- ***get\_object\_or\_404***: Busca um item no banco; se não existir, mostra erro 404.
- ***Tarefa***: Importa o model criado, para usar nas consultas e operações.

# CRIANDO UM CRUD: TO-DO LIST

## Criando as Regras de Lógica: Views

Ainda no mesmo arquivo **views.py** na pasta tarefas, vamos criar as funções abaixo.

### Listar tarefas



```
def listar_tarefas(request):  
    tarefas = Tarefa.objects.all()  
    return render(request, 'listar.html', {'tarefas': tarefas})
```


### Explicação:

- Busca todas as tarefas no banco (***Tarefa.objects.all()***).
- Envia essa lista para o template listar.html usando a variável tarefas.
- Essa é a **página inicial**, que mostra todas as tarefas.

# CRIANDO UM CRUD: TO-DO LIST

## Criando as Regras de Lógica: Views

### Criar tarefas

A screenshot of a code editor window with a dark theme. The title bar shows three colored circles (red, yellow, green) and a tab labeled 'views.py' with a Python logo. The code is written in Python and defines a function 'criar\_tarefa' that handles POST requests to create a new task and redirect to the list view, or renders a form for creating a task.

```
def criar_tarefa(request):  
    if request.method == 'POST':  
        titulo = request.POST.get('titulo')  
        Tarefa.objects.create(titulo=titulo)  
        return redirect('listar')  
    return render(request, 'criar.html')
```

### Explicação:

- Se o formulário for enviado (POST), ele pega o título digitado e cria uma nova tarefa.
- Depois redireciona para a lista de tarefas.
- Se for só para exibir o formulário, mostra o HTML criar.html.

# CRIANDO UM CRUD: TO-DO LIST

## Criando as Regras de Lógica: Views

### Editar tarefas

A screenshot of a code editor window titled 'views.py'. The code defines a function 'editar\_tarefa' that takes 'request' and 'id' as arguments. It first retrieves the task object using 'get\_object\_or\_404'. Then, it checks if the request method is 'POST'. If so, it updates the task's title and completion status from the request data and saves it. Finally, it either redirects to the 'listar' view or renders the 'editar.html' template with the task data.

```
def editar_tarefa(request, id):  
    tarefa = get_object_or_404(Tarefa, id=id)  
    if request.method == 'POST':  
        tarefa.titulo = request.POST.get('titulo')  
        tarefa.concluida = 'concluida' in request.POST  
        tarefa.save()  
        return redirect('listar')  
    return render(request, 'editar.html', {'tarefa': tarefa})
```

### Explicação:

- Busca a tarefa pelo ID (ou mostra erro 404 se não existir).
- Se for POST, atualiza o título e o status de conclusão com os novos valores e salva.
- Depois redireciona para a lista.
- Se for uma visita normal, só exhibe o formulário editar.html com os dados preenchidos.

# CRIANDO UM CRUD: TO-DO LIST

## Criando as Regras de Lógica: Views

### Excluir tarefas

A screenshot of a code editor window with a dark theme. The window has a title bar with three colored circles (red, yellow, green) and a tab labeled 'views.py' with a Python logo. The code is written in Python and defines a function 'excluir\_tarefa' that takes 'request' and 'id' as arguments. It uses 'get\_object\_or\_404' to find a task, calls 'delete()' on it, and then redirects to the 'listar' view.

```
def excluir_tarefa(request, id):  
    tarefa = get_object_or_404(Tarefa, id=id)  
    tarefa.delete()  
    return redirect('listar')
```

### Explicação:

- Busca a tarefa pelo ID.
- Apaga do banco de dados com delete().
- Redireciona o usuário de volta para a lista de tarefas.

**05**

# Criando um CRUD: To-Do List

## URL



# CRIANDO UM CRUD: TO-DO LIST

## URLs: Ligando tudo

No Django, o **arquivo de URLs** define **quais funções (views)** devem ser chamadas quando o usuário acessa determinada **rota** (endereço no navegador).

É como um **mapa** que liga o que o usuário digita na barra de endereço às funções Python que devem responder.

### Como o sistema de URLs funciona?

Existem **dois níveis** principais de arquivos de URL:

1. urls.py do **projeto** (ex: todolist/urls.py)
2. urls.py do **app** (ex: tarefas/urls.py)

O projeto organiza o todo. O app define as rotas específicas daquela funcionalidade (ex: rotas da to-do list).

Em seguida, vamos ver como configurar essas rotas no código.



# CRIANDO UM CRUD: TO-DO LIST

## Configurando as Rotas do App

Para configurar as rotas do app você deve acessar **urls.py** na pasta **/tarefas** e digitar as rotas, como no código abaixo.

```
from django.urls import path
from .views import listar_tarefas, criar_tarefa, editar_tarefa, excluir_tarefa

urlpatterns = [
    path('', listar_tarefas, name='listar'),
    path('criar/', criar_tarefa, name='criar'),
    path('editar/<int:id>/', editar_tarefa, name='editar'),
    path('excluir/<int:id>/', excluir_tarefa, name='excluir'),
]
```

### Explicando cada rota:

- **' '**: Rota raiz (✓) - mostra a lista de tarefas (**listar\_tarefas**).
- **'criar/'**: Mostra o formulário para nova tarefa (**criar\_tarefa**).
- **'editar/<int:id>/'**: Edita a tarefa com o ID informado (**editar\_tarefa**).
- **'excluir/<int:id>/'**: Exclui a tarefa com o ID informado (**excluir\_tarefa**).
- **<int:id>**: Espera um número como identificador da tarefa.
- **name='...'**: Permite usar **{% url 'nome' %}** nos templates.

# Agradecimentos

# OBRIGADA POR LER ATÉ AQUI!

Este eBook foi gerado por IA, diagramado e editado por humano.

O conteúdo foi criado com fins de aprendizado. Não passou por uma revisão humana cuidadosa e pode conter erros gerados pela IA.

Espero que este material tenha sido útil para você. Bons estudos!

Conheça mais sobre a autora em:  
**[www.linkedin.com/in/beatriz-costaa](https://www.linkedin.com/in/beatriz-costaa)**