

Coding II

Ronierison Maciel

Agosto 2024

Quem sou eu?



Nome: Ronierison Maciel / Roni

Formação: Mestre em Ciência da Computação

Ocupação: Pesquisador, Professor e Desenvolvedor de Software

Hobbies: Jogar cartas, ficar com a família no final de semana conversando sobre diversos temas

Interesses: Carros, aprimoramento na área educacional, desenvolvimento de software, data science e machine learning

Email: ronierison.maciел@pe.senac.br

GitHub: <https://github.com/ronierisonmaciel>

Conteúdo

- 1 Introdução
- 2 Tipos de dados e operadores em python
- 3 Condicionais em python
- 4 Estruturas de repetição em python
- 5 Funções em python
- 6 Estruturas de dados
- 7 Entrada e saída de dados em python
- 8 Manipulação de arquivos com python
- 9 (POO) em python
- 10 Módulos e pacotes em python
- 11 Tratamento de exceções em python
- 12 Introdução ao Django
- 13 Configuração de aplicações e views básicas



Objetivos:

- **Compreender** os conceitos básicos da programação e a história do Python.
- **Configurar** o ambiente de desenvolvimento para começar a programar em Python.
- **Escrever** e executar o primeiro script em Python.

O que é programação?

Programação é o processo de escrever instruções que serão executadas por um computador. Componentes da Programação:

- **Código-Fonte:** As instruções que escrevemos.
- **Compilador/Interpretador:** Ferramenta que traduz o código para o computador.
- **Output:** O resultado das instruções executadas.

A Origem da linguagem Python



Por volta de 1989, **Guido van Rossum** trabalhava no CWI (Centrum Wiskunde & Informatica) e estava envolvido no desenvolvimento da linguagem ABC. A linguagem ABC era uma linguagem interpretada, ou seja, seu código era executado diretamente sem a necessidade de compilação prévia. Além disso, os blocos de código em ABC eram delimitados por indentação, uma característica que promove a legibilidade do código. ABC também oferecia tipos de dados de altíssimo nível, o que facilitava a programação para os usuários.

Limitações da linguagem ABC

Apesar de suas vantagens, a linguagem **ABC** tinha uma limitação significativa: não permitia extensões. Isso significava que os programadores não podiam adicionar novas funcionalidades ou adaptar a linguagem para atender a necessidades específicas que surgissem durante o desenvolvimento de projetos mais complexos.

Surgimento da necessidade de uma nova linguagem

Diante dessas dificuldades, surgiu a necessidade de uma linguagem de script semelhante à **ABC**, mas que permitisse extensões genéricas. **Guido** queria uma linguagem que combinasse a simplicidade e a clareza do ABC com a flexibilidade de extensões e o poder de manipulação do sistema que eles precisavam para o Amoeba.

```
1 HOW TO total sum list:
2     PUT 0 IN total
3     FOR number IN list:
4         ADD number TO total
5     RETURN total
6
```


Criação da linguagem Python

Durante os feriados de **Natal** de 1989, **Guido** começou a trabalhar nessa nova linguagem. Ele dedicou seu tempo disponível no ano seguinte ao desenvolvimento dessa linguagem, que eventualmente se tornaria o Python. A linguagem foi projetada para ser fácil de usar e aprender, incorporando a clareza do **ABC** e a robustez necessária para o Amoeba.

Python foi usado com sucesso crescente no projeto Amoeba, mostrando-se uma ferramenta poderosa e flexível. Em fevereiro de 1991, depois de pouco mais de um ano de desenvolvimento, Python foi publicado na USENET, tornando-se disponível para a comunidade de desenvolvedores.

Características e uso do Python

Python é uma linguagem de programação que apresenta muitas particularidades tornando-a utilizada e apreciada pela comunidade de desenvolvedores. Vamos explorar essas características e o impacto que elas têm no desenvolvimento de software.

- **Interpretada:** O código é executado diretamente pelo interpretador, sem necessidade de compilação prévia, facilitando a execução e teste rápido.
- **Orientada a Objetos:** Suporta a definição de classes e criação de objetos, organizando e estruturando o código de maneira modular e reutilizável.
- **Altíssimo Nível:** Abstrai complexidades do hardware, permitindo que os programadores foquem mais na lógica do problema e menos nos detalhes técnicos.

Características e uso do Python

- **Multiplataforma:** Executável em diversos sistemas operacionais (Windows, macOS, Linux) sem necessidade de modificações.
- **Interativa:** Oferece um modo interativo para executar comandos linha a linha, tornando o desenvolvimento e depuração mais dinâmicos e eficientes.

- Desenvolvimento Web,
- Ciência de Dados,
- Inteligência Artificial,
- Automação, entre outros.

Instalação do Python

Windows:

- Baixar o instalador em python.org
- Executar o instalador e marcar a opção “Add Python to PATH”.
- Verificar a instalação usando o comando `python --version` no terminal.

Mac/Linux:

- Verificar a versão pré-instalada.
- Atualizar ou instalar usando gerenciadores de pacotes (brew, apt-get, etc.).
- Comando: `python3 --version`.

VSCode:

- Leve e flexível.
- Suporte para extensões e Python integrado.

PyCharm:

- Completa, com ferramentas de depuração avançadas.

Jupyter Notebook:

- Ideal para prototipação rápida e análise de dados.

O que é REPL?: Read, Eval, Print, Loop.

Ferramenta interativa do Python para testar comandos em tempo real. Comandos Básicos:

- Operações matemáticas: $2 + 2$
- Trabalhar com strings: `"Hello" * 3`
- Importar bibliotecas: `import math`

O que é o IDLE?:

- IDLE é um ambiente de desenvolvimento integrado simples que vem instalado com o Python, oferecendo uma interface gráfica para escrever e executar código Python.

Características do IDLE:

- **Editor de código:**
- Permite escrever, editar e salvar scripts Python.

Shell Interativo:

- Uma interface interativa que permite executar comandos Python em tempo real, semelhante ao REPL, mas com uma interface gráfica.

Continua...

Depuração:

- Oferece uma ferramenta de depuração integrada para ajudar a encontrar e corrigir erros no código.

Como iniciar o IDLE: Iniciando:

- No Windows, você pode iniciar o IDLE pesquisando "IDLE" no menu Iniciar.
- No Mac ou Linux, pode ser iniciado a partir do terminal com o comando `idle` ou `import idlelib.idle`

Exemplo prático:

- **Escrevendo e Executando Código:**
- Digite um código simples, como `print("Olá, mundo!")`, no editor e execute-o para ver a saída na janela de shell interativa.

Criação de um script

Abra sua IDE ou editor de texto.

- Digite o código: `print("Hello, World!")`.
- Salve o arquivo com extensão `.py` (ex.: `hello.py`).
- Executando o Script:
- Via terminal: `python hello.py`.
- Na IDE: Usando a funcionalidade de execução integrada.

- **Tarefa 1:**

Configurar o ambiente de desenvolvimento seguindo as instruções.

- **Tarefa 2:**

Escrever e executar o script Hello, World!.

- **Tarefa 3:**

Explorar o REPL e testar comandos básicos.

- **Discussão:** Compartilhe dificuldades e soluções encontradas durante a configuração e execução.

Revisão:

- Conceitos básicos de programação.
- História e características do Python.
- Instalação do Python e configuração de ambiente.
- Criação e execução do primeiro script em Python.

Próximos passos:

- Continuar explorando o ambiente.
- Preparar-se para estudar tipos de dados e operadores na próxima semana.

Leitura recomendada:

- Documentação oficial do Python: docs.python.org

Tutoriais online:

- Codecademy, W3Schools.

Comunidades:

- Stack Overflow, Reddit - r/learnpython.

Objetivos:

- **Compreender** os diferentes tipos de dados em Python.
- **Aprender** a utilizar operadores aritméticos, relacionais e lógicos.
- **Aplicar** esses conceitos na criação de expressões e scripts simples.

Tipos de dados primitivos

Tipos de dados são categorias que especificam quais tipos de valores uma variável pode armazenar.

Tipos de dados em Python:

- **Inteiros (int):** Números inteiros, positivos ou negativos.
- **Flutuantes (float):** Números decimais.
- **Strings (str):** Sequência de caracteres.
- **Booleanos (bool):** Verdadeiro (True) ou Falso (False).

Inteiros (int):

- ex. $x = 10$, $y = -3$
- Usado para contar, iterar, etc.

Flutuantes (float):

- ex. $\pi = 3.14$, $\text{temperature} = -5.5$
- Usado para medições, cálculos precisos, etc.

Exemplo prático:

- $a = 7$
- $b = 3.5$
- $c = a + b$

Trabalhando com strings

Strings são sequências de caracteres delimitadas por aspas simples ('...') ou duplas ("...").

Operações básicas:

- **Concatenação:** `greeting = "Hello" + " " + "World!"`
- **Repetição:** `laugh = "ha" * 3` → Resultado: `"hahaha"`
- **Indexação:** `first_letter = "Python"[0]` → Resultado: `"P"`

Exemplo prático:

- `name = "Roni"`
- `welcome_message = "Hello, " + name + "!"`

Tipos booleanos e comparações

Booleanos representam valores verdadeiros ou falsos: True ou False.

Operações lógicas:

- **Comparações:** `==`, `!=`, `>`, `<`, `>=`, `<=`
- **Lógicos:** `and`, `or`, `not`

Exemplo prático:

- `is_adult = age >= 18`
- `can_vote = is_adult and citizenship == "Brazil"`

Lista de operadores:

- **Adição:** (+) $\rightarrow 5 + 3 = 8$
- **Subtração:** (-) $\rightarrow 9 - 4 = 5$
- **Multiplicação:** (*) $\rightarrow 7 * 6 = 42$
- **Divisão:** (/) $\rightarrow 8 / 2 = 4.0$
- **Divisão inteira:** (//) $\rightarrow 8 // 3 = 2$
- **Módulo:** (%) $\rightarrow 8 \% 3 = 2$
- **Exponenciação:** (**) $\rightarrow 2 ** 3 = 8$

Lista de operadores:

- Igual a: `==`
- Diferente de: `!=`
- Maior que: `>`
- Menor que: `<`
- Maior ou igual a: `>=`
- Menor ou igual a: `<=`

Exemplo prático:

- `result = (5 > 3) → Resultado: True`
- `result = (7 == 8) → Resultado: False`

Lista de operadores:

- E (and): Retorna True se ambos os operandos forem verdadeiros.
- Ou (or): Retorna True se pelo menos um dos operandos for verdadeiro.
- Não (not): Inverte o valor lógico.

Exemplo prático:

- `result = (5 > 3) and (8 > 6) → Resultado: True`
- `result = (7 > 10) or (5 < 2) → Resultado: False`
- `result = not (4 == 4) → Resultado: False`

Criando expressões combinadas

Combinação de operadores:

Operadores aritméticos, relacionais e lógicos podem ser combinados para criar expressões complexas.

Precedência de operadores:

- Parênteses: ()
- Exponenciação: **
- Multiplicação, divisão, módulo: *, /, %
- Adição e subtração: +, -
- Comparações: ==, !=, >, <, >=, <=
- Lógicos: not, and, or

Exemplo prático:

- `x = (5 + 2) * 3 > 10 and (4 % 2 == 0)`

- **Tarefa 1:**

Criar variáveis usando diferentes tipos de dados.

- **Tarefa 2:**

Escrever expressões aritméticas e lógicas, e prever seus resultados.

- **Tarefa 3:**

Implementar um script que peça ao usuário dois números e mostre os resultados das operações aritméticas básicas entre eles.

Revisão:

- Tipos de dados primitivos em Python: inteiros, flutuantes, strings, booleanos.
- Uso de operadores aritméticos, relacionais e lógicos.
- Criação de expressões combinadas e compreensão da precedência de operadores.

Próximos passos:

- Aplicar o conhecimento em estruturas condicionais na próxima semana.

Leitura recomendada:

- Documentação oficial de tipos de dados em Python: docs.python.org

Exercícios online:

- Exercícios sobre tipos de dados e operadores em plataformas como Codecademy, HackerRank.

Comunidades:

- Stack Overflow, Reddit - r/learnpython.

Objetivos:

- **Entender** como usar estruturas condicionais em Python para tomar decisões no código.
- **Aprender** a utilizar `if`, `elif`, e `else` para controlar o fluxo do programa.
- **Aplicar** esses conceitos em problemas práticos.

Introdução às estruturas condicionais

Estruturas condicionais permitem que seu programa execute determinadas partes do código com base em condições específicas.

- **Fluxo de Decisão:** Dependendo de uma condição (True ou False), diferentes blocos de código serão executados.
- **Importância:** Fundamental para criar programas dinâmicos que reagem de maneiras diferentes a diferentes entradas.

sintaxe básica:

```
1 if condição:
2     # bloco de código a ser executado se a condição for
    verdadeira
```

Exemplo prático:

```
1 age = 18
2 if age >= 18:
3     print("Você é maior de idade.")
```

Explicação: Se `age` for maior ou igual a 18, a mensagem "Você é maior de idade." será exibida.

Nota: A indentação é crucial em Python para definir blocos de código.

Usando else

sintaxe básica:

```
1     if condição:
2         # bloco de código 1
3 else:
4     # bloco de código 2
```

Exemplo prático:

```
1 age = 16
2 if age >= 18:
3     print("Você é maior de idade.")
4 else:
5     print("Você é menor de idade.")
```

Explicação: Se a condição do **if** não for verdadeira, o bloco **else** será executado.

Bloco elif

sintaxe básica:

```
1 if condição1:
2     # bloco de código 1
3 elif condição2:
4     # bloco de código 2
5 else:
6     # bloco de código 3
```

Exemplo prático:

```
1 score = 85
2 if score >= 90:
3     print("Nota A")
4 elif score >= 80:
5     print("Nota B")
6 else:
7     print("Nota C")
```

Explicação: `elif` permite testar múltiplas condições em sequência, parando na primeira que for verdadeira.

Aninhando estruturas condicionais

Estruturas condicionais podem ser aninhadas dentro de outras para lidar com cenários mais complexos.

Exemplo prático:

```
1 age = 20
2 if age >= 18:
3     if age >= 65:
4         print("Você é um idoso.")
5     else:
6         print("Você é um adulto.")
7 else:
8     print("Você é menor de idade.")
```

Explicação: O bloco `if` interno verifica uma segunda condição se a primeira for verdadeira.

Condições complexas

Condições podem ser combinadas usando operadores lógicos (and, or, not) para formar expressões mais complexas.

```
1 temperature = 22
2 is_sunny = True
3
4 if temperature > 20 and is_sunny:
5     print("Ótimo dia para um passeio!")
6 elif temperature <= 20 or not is_sunny:
7     print("Talvez ficar em casa seja uma boa ideia.")
8 else:
9     print("Que dia estranho.")
```

Explicação: O programa decide o que exibir com base em duas condições diferentes.

Evitando erros comuns

- Indentação:

Regra: Cada bloco de código em Python deve ser corretamente indentado.

```
1 if condição:
2     print("Isso causará um erro!")    # Sem indentação correta
```

- Comparação com = ao invés de ==:

Erro Comum: Usar = (atribuição) ao invés de == (comparação) em uma condição.

```
1 if x = 5:    # Erro: deve ser '=='
2     print("Cinco")
```

- **Tarefa 1:**

Escreva um script que leia uma nota (0-100) e imprima o conceito (A, B, C, D, F) baseado em faixas de valores.

- **Tarefa 2:**

Crie um programa que peça ao usuário sua idade e decida se ele pode votar e/ou se ele é aposentado.

- **Tarefa 3:**

Desenvolva um pequeno jogo de adivinhação onde o usuário deve acertar um número entre 1 e 10, com dicas baseadas na entrada do usuário.

Revisão:

- **Compreendemos** como usar estruturas condicionais if, elif, e else.
- **Exploramos** aninhamento de condições e combinações com operadores lógicos.
- **Aplicamos** esses conceitos em exemplos práticos.

Leitura recomendada:

- Documentação oficial do Python: docs.python.org

Exercícios online:

- Exercícios sobre estruturas condicionais em plataformas como Codecademy, LeetCode.

Comunidades:

- Stack Overflow, Reddit - [r/learnpython](https://www.reddit.com/r/learnpython).

Objetivos:

- **Compreender** como usar estruturas de repetição para automatizar tarefas em Python.
- **Aprender** a utilizar os laços for e while para iterar sobre sequências e executar blocos de código repetidamente.
- **Entender** o uso de comandos de controle como break, continue e pass.

Introdução às estruturas de repetição

Estruturas de repetição permitem que um bloco de código seja executado múltiplas vezes com base em uma condição ou sobre uma sequência.

- **Importância:**

Essencial para automatizar tarefas repetitivas, processar listas de dados, e realizar iterações controladas.

O laço for

sintaxe básica:

```
1 for item in sequência:  
2     # bloco de código a ser repetido
```

Funcionamento:

O laço for itera sobre cada item em uma sequência (como listas, strings, tuplas).

Exemplo prático:

```
1 fruits = ["apple", "banana", "cherry"]  
2 for fruit in fruits:  
3     print(fruit)
```

Explicação: O código imprime cada fruta na lista.

Iterando com range()

A função range() gera uma sequência de números, muito usada com for.

sintaxe:

```
1 for i in range(start, stop, step):  
2     # bloco de código a ser repetido
```

Exemplo prático:

```
1 for i in range(1, 6):  
2     print(i)
```

Explicação: Imprime números de 1 a 5.

O laço while

sintaxe básica

```
1 while condição:  
2     # bloco de código a ser repetido
```

Funcionamento:

O laço while repete o bloco de código enquanto a condição for verdadeira.

Exemplo prático:

```
1 count = 1  
2 while count <= 5:  
3     print(count)  
4     count += 1
```

Explicação: O código imprime números de 1 a 5, incrementando count a cada iteração.

Comandos de controle: break, continue, pass

O **break** encerra o laço imediatamente.

Exemplo:

```
1 for i in range(1, 10):  
2     if i == 5:  
3         break  
4     print(i)
```

Explicação: O laço é interrompido quando *i* é igual a 5. Já o **continue** pula para a próxima iteração do laço.

Exemplo:

```
1 for i in range(1, 6):  
2     if i == 3:  
3         continue  
4     print(i)
```

Explicação: Quando *i* é 3, a impressão é pulada.

Comandos de controle: break, continue, pass

O **pass** não faz nada; é usado como placeholder.

Exemplo:

```
1 for i in range(1, 6):  
2     if i == 3:  
3         pass  
4     print(i)
```

Explicação: O **pass** não afeta o fluxo do laço.

Iterando sobre strings

Strings são iteráveis, o que significa que podemos iterar sobre cada caractere usando um laço for.

Exemplo prático:

```
1 word = "Python"
2 for letter in word:
3     print(letter)
```

Explicação: O código imprime cada letra da string "Python" em uma nova linha.

Laços aninhados

Um laço pode ser colocado dentro de outro, criando um laço aninhado, útil para trabalhar com matrizes ou listas de listas.

Exemplo prático:

```
1 for i in range(1, 4):  
2     for j in range(1, 4):  
3         print(f"i={i}, j={j}")
```

Explicação: Para cada valor de *i*, o laço interno itera *j* de 1 a 3.

Evitando loops infinitos

Um loop infinito ocorre quando a condição de um laço while nunca se torna falsa.

Exemplo:

```
1 while True:
2     print("Isso vai continuar para sempre... ou até você interromper.")
```

Dicas para Evitar:

- Verifique cuidadosamente as condições de saída.
- Garanta que a variável de controle seja corretamente atualizada.

- **Tarefa 1**

Escreva um laço for que percorre uma lista de números e imprime apenas os números pares.

- **Tarefa 2**

Crie um script que solicite ao usuário um número e use um laço while para calcular o fatorial desse número.

- **Tarefa 3**

Desenvolva um jogo simples em que o usuário deve adivinhar um número aleatório entre 1 e 100, com dicas fornecidas após cada tentativa. Use while para repetir até que o usuário acerte.

Revisão:

- Compreendemos o uso de laços for e while para iterar sobre sequências e condições.
- Exploramos comandos de controle como break, continue, e pass.
- Praticamos a criação de laços aninhados e discutimos a importância de evitar loops infinitos.

Próximos passos:

- Na próxima semana, aprenderemos sobre funções em Python, que permitem modularizar e reutilizar código.

Leitura recomendada:

- Documentação oficial do Python: docs.python.org

Exercícios online:

- Exercícios sobre loops em plataformas como LeetCode, HackerRank.

Comunidades:

- Stack Overflow, Reddit - r/learnpython.

Objetivos:

- **Compreender** o conceito de funções e sua importância em programação.
- **Aprender** a definir e chamar funções em Python.
- **Entender** o uso de parâmetros e valores de retorno.
- **Aplicar** o conceito de funções em problemas práticos.

Definição:

- Funções são blocos de código reutilizáveis que executam uma tarefa específica.

Importância:

- Facilita a organização e a modularização do código.
- Promove a reutilização e a manutenção eficiente do código.

Analogia:

- Pense em uma função como uma "receita": ela recebe ingredientes (parâmetros), executa um conjunto de instruções e pode retornar um resultado.

Definindo uma função

sintaxe básica:

```
1 def nome_da_funcao(param1, param2):  
2     # bloco de código a ser executado  
3     return valor_de_retorno
```

Componentes:

- def: Palavra-chave que indica a definição de uma função.
- nome_da_funcao: Nome da função.
- param1, param2: Parâmetros (opcionais).

Exemplo prático:

```
1 def saudacao(nome):  
2     return f"Olá, {nome}!"
```

Chamando uma função

sintaxe básica:

```
1 nome_da_funcao(argumento1 , argumento2)
```

Funcionamento:

Quando uma função é chamada, o controle do programa passa para a função, os parâmetros são passados, e o bloco de código é executado.

```
1 mensagem = saudacao("Roni")  
2 print(mensagem)
```

Parâmetros e argumentos

Definição:

- **Parâmetros:** Variáveis listadas na definição da função.
- **Argumentos:** Valores passados para os parâmetros na chamada da função.

Tios de Parâmetro:

- **Obrigatórios:** Devem ser fornecidos ao chamar a função.
- **Opcionais (com valores padrão):** Têm um valor padrão se não forem fornecidos.

```
1 def saudacao(nome, saudacao="Olá"):  
2     return f"{saudacao}, {nome}!"  
3  
4 print(saudacao("Roni"))  
5 print(saudacao("Roni", "Bom dia"))
```

Retornando valores de funções

Definição:

- Uma função pode retornar um valor usando a palavra-chave **return**.
- **return** finaliza a execução da função e retorna o valor especificado.

Exemplo prático:

```
1 def soma(a, b):  
2     return a + b  
3  
4 resultado = soma(5, 3)  
5 print(resultado)  # Saída: 8
```

Nota: Se uma função não tiver **return**, ela retorna **None** por padrão.

Escopo de variáveis

Definição:

- **Escopo Local:** Variáveis definidas dentro de uma função só existem dentro dessa função.
- **Escopo Global:** Variáveis definidas fora de qualquer função podem ser acessadas em qualquer lugar no código.

Exemplo prático:

```
1 x = 10  # Variável global
2
3 def funcao():
4     x = 5  # Variável local
5     print(x)
6
7 funcao()  # Saída: 5
8 print(x)  # Saída: 10
```

Nota: Evita conflitos de nomes e facilita a manutenção do código.

Funções Aninhadas e Closures

Definição:

- **Funções Aninhadas:** Uma função pode ser definida dentro de outra função.
- **Closure:** Uma função interna que se lembra das variáveis no escopo de sua função externa, mesmo depois que a função externa foi executada.

Exemplo prático:

```
1 def pai(num):  
2     def filho(x):  
3         return x + num  
4     return filho  
5  
6 adicao = pai(10)  
7 print(adicao(5))    # Saída: 15
```

Documentando funções

Docstrings:

- **Definição:** Docstrings são cadeias de caracteres que documentam uma função.
- **Uso:** Colocadas logo após a definição da função para descrever seu propósito e parâmetros.

Exemplo prático:

```
1 def saudacao(nome):  
2     """  
3     Esta função retorna uma saudação personalizada.  
4  
5     Parâmetros:  
6     nome (str): Nome da pessoa a ser saudada.  
7  
8     Retorno:  
9     str: Uma saudação na forma de string.  
10    """  
11    return f"Olá, {nome}!"
```

Tarefa 1:

- Escreva uma função `media` que recebe três números e retorna a média aritmética deles.

Tarefa 2:

- Crie uma função fatorial que calcule o fatorial de um número usando um loop `while`.

Tarefa 3:

- Desenvolva uma função `conversor_temperatura` que converte graus Celsius para Fahrenheit e vice-versa, dependendo de um parâmetro adicional.

Revisão:

- Aprendemos a definir e chamar funções, a trabalhar com parâmetros e retornos.
- Discutimos escopo de variáveis e vimos a importância das docstrings.
- Aplicamos esses conceitos em exercícios práticos.

Próximos passos:

- Na próxima semana, aprenderemos sobre estruturas de dados mais avançadas, como listas e dicionários.

Leitura recomendada:

- Documentação oficial do Python: docs.python.org

Exercícios online:

- Exercícios sobre funções em plataformas como Codecademy, LeetCode.

Comunidades:

- Stack Overflow, Reddit - [r/learnpython](https://www.reddit.com/r/learnpython).

Objetivos:

- **Compreender** o conceito de listas e sua importância como estrutura de dados em Python.
- **Aprender** a criar, acessar, modificar e iterar sobre listas.
- **Explorar** métodos e operações comuns em listas.
- **Aplicar** listas em problemas práticos.

Definição:

- Uma lista é uma coleção ordenada e mutável de itens em Python, que podem ser de tipos variados.

Importância:

- Listas são uma das estruturas de dados mais versáteis e amplamente usadas, permitindo armazenar e manipular múltiplos itens em uma única variável.

Criando listas

sintaxe básica:

```
1 nome_da_lista = [item1, item2, item3, ...]
```

Exemplo prático:

```
1 frutas = ["maçã", "banana", "cereja"]  
2 numeros = [1, 2, 3, 4, 5]  
3 misturada = ["texto", 42, 3.14, True]
```

Características:

- **Ordenada:** A ordem dos itens é preservada.
- **Mutável:** Itens podem ser modificados, adicionados ou removidos.

Acessando itens em listas

Indexação:

- Cada item em uma lista tem uma posição (índice) que começa em 0.
- sintaxe: nome_da_lista[indice]

Exemplo prático:

```
1 frutas = ["maçã", "banana", "cereja"]  
2 print(frutas[0])    # Saída: maçã  
3 print(frutas[2])    # Saída: cereja
```

Indexação negativa:

- Permite acessar itens de trás para frente.
- Exemplo: frutas[-1] retorna "cereja".

Modificando listas

Modificando itens:

- Listas são mutáveis, o que significa que podemos modificar seus itens.
- sintaxe: `nome_da_lista[indice] = novo_valor`

Exemplo Prático:

```
1 frutas = ["maçã", "banana", "cereja"]
2 frutas[1] = "laranja"
3 print(frutas)    # Saída: ["maçã", "laranja", "cereja"]
```

`append(item):`

- Adiciona um item ao final da lista.
- **Exemplo:** `frutas.append("uva")`

`insert(indice, item):`

- Insere um item em uma posição específica.
- **Exemplo:** `frutas.insert(1, "abacaxi")`

`remove(item):`

- Remove a primeira ocorrência de um item na lista.
- **Exemplo:** `frutas.remove("banana")`

`pop(indice):`

- Remove e retorna o item no índice especificado (ou o último item, se o índice não for fornecido).
- **Exemplo:** `ultimo = frutas.pop()`

sort():

- Ordena a lista em ordem crescente.
- **Exemplo:** `numeros.sort()`

reverse():

- Inverte a ordem dos itens na lista.
- **Exemplo:** `frutas.reverse()`

Operações comuns com listas

Concatenando listas:

- sintaxe: `lista1 + lista2`

Exemplo:

```
1 lista1 = [1, 2, 3]
2 lista2 = [4, 5, 6]
3 lista3 = lista1 + lista2 # Saída: [1, 2, 3, 4, 5, 6]
```

Repetindo listas:

- sintaxe: `lista * n`

Exemplo:

```
1 frutas = ["maçã", "banana"]
2 nova_lista = frutas * 2 # Saída: ["maçã", "banana", "maçã",
    "banana"]
```

Operações comuns com listas

Verificando existência de itens:

- sintaxe: item in lista

Exemplo:

```
1 if "maçã" in frutas:  
2     print("A lista contém maçã.")
```

Iterando sobre listas

Uso de for:

- Iteração sobre cada item da lista.
- sintaxe:

```
1 for item in lista:  
2     # bloco de código a ser executado para cada item
```

Exemplo prático:

```
1 frutas = ["maçã", "banana", "cereja"]  
2 for fruta in frutas:  
3     print(fruta)
```

Saída

```
1 maçã  
2 banana  
3 cereja
```

List comprehensions

Definição:

- Uma maneira concisa de criar listas usando uma expressão.
- **sintaxe:** [expressao for item in lista if condicao]

Exemplo prático:

```
1 quadrados = [x**2 for x in range(10)]  
2 print(quadrados)    # Saída: [0, 1, 4, 9, 16, 25, 36, 49, 64,  
                        81]
```

Explicação:

- A expressão $x**2$ é aplicada a cada x no intervalo de 0 a 9, gerando uma nova lista de quadrados.

Fatiamento de listas

Definição:

- Fatiamento permite acessar subpartes de uma lista.
- **sintaxe:** lista[início:fim:passo]

Exemplo prático:

```
1 numeros = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
2 print(numeros[2:5])      # Saída: [2, 3, 4]
3 print(numeros[:4])       # Saída: [0, 1, 2, 3]
4 print(numeros[::2])      # Saída: [0, 2, 4, 6, 8]
```

Explicação:

- **início:** posição inicial (inclusiva).
- **fim:** posição final (exclusiva).
- **passo:** salto entre elementos.

Tarefa 1:

- Crie uma lista de números de 1 a 10 e, em seguida, faça um loop para calcular a soma de todos os números.

Tarefa 2:

- Escreva um script que recebe uma lista de nomes e retorna uma nova lista contendo apenas os nomes que começam com uma vogal.

Tarefa 3:

- Desenvolva um programa que solicita ao usuário uma lista de números e, em seguida, remove todos os números duplicados da lista.

Revisão:

- **Compreendemos** como criar, acessar e modificar listas.
- **Exploramos** métodos comuns e operações em listas, como concatenação, repetição e fatiamento.
- **Aplicamos** o conceito de listas em exercícios práticos.

Próximos passos:

- Na próxima semana, aprenderemos sobre outras estruturas de dados importantes em Python: tuplas e dicionários.

Leitura recomendada:

- Documentação oficial do Python: docs.python.org

Exercícios online:

- Exercícios sobre listas em plataformas como HackerRank, LeetCode.

Comunidades:

- Stack Overflow, Reddit - r/learnpython.

Objetivos:

- **Compreender** o conceito de tuplas e dicionários e suas diferenças em relação a listas.
- **Aprender** a criar, acessar e modificar tuplas e dicionários.
- **Explorar** operações e métodos comuns em tuplas e dicionários.
- **Aplicar** esses conceitos em problemas práticos.

Definição:

- Uma tupla é uma coleção ordenada e imutável de itens em Python.

Características:

- **Imutável:** Após a criação, os itens não podem ser modificados.
- **Ordenada:** A ordem dos itens é preservada.

Quando usar:

- Quando os dados não precisam ser alterados e você deseja garantir sua integridade.

Criando tuplas

sintaxe básica:

```
1 nome_da_tupla = (item1, item2, item3, ...)
```

Exemplo prático:

```
1 coordenadas = (10.0, 20.0)
2 cores = ("vermelho", "verde", "azul")
3 misturada = ("texto", 42, 3.14, True)
```

Nota:

- Tuplas com um único item devem incluir uma vírgula: (item,)

Acessando itens em tuplas

Indexação:

- Semelhante às listas, cada item em uma tupla tem uma posição (índice) que começa em 0.
- **sintaxe:** nome_da_tupla[indice]

Exemplo prático:

```
1 cores = ("vermelho", "verde", "azul")
2 print(cores[0])    # Saída: vermelho
3 print(cores[2])    # Saída: azul
```

Indexação negativa:

- **Exemplo:** cores[-1] retorna "azul".

Métodos comuns de tuplas

`count(item):`

- Retorna o número de vezes que um item aparece na tupla.
- **Exemplo:** `coordenadas.count(10.0)`

`index(item):`

- Retorna o índice da primeira ocorrência do item na tupla.
- **Exemplo:** `cores.index("verde")`

Operações com tuplas

Concatenando tuplas:

- sintaxe: `tupla1 + tupla2`

Exemplo:

```
1 tupla1 = (1, 2, 3)
2 tupla2 = (4, 5, 6)
3 tupla3 = tupla1 + tupla2 # Saída: (1, 2, 3, 4, 5, 6)
```

Repetindo tuplas:

- sintaxe: `tupla * n`

Exemplo:

```
1 cores = ("vermelho", "verde")
2 nova_tupla = cores * 2 # Saída: ("vermelho", "verde", "vermelho", "verde")
```

Iterando sobre tuplas

Uso de for:

- Iteração sobre cada item da tupla.

sintaxe:

```
1 for item in tupla:
2     # bloco de código a ser executado para cada item
```

Exemplo prático:

```
1 cores = ("vermelho", "verde", "azul")
2 for cor in cores:
3     print(cor)
```

Saída

```
1 vermelho
2 verde
3 azul
```

Fatiamento de tuplas

Definição:

- Fatiamento permite acessar subpartes de uma tupla.
- **sintaxe:** `tupla[inicio:fim:passo]`

Exemplo prático:

```
1 numeros = (0, 1, 2, 3, 4, 5, 6, 7, 8, 9)
2 print(numeros[2:5])      # Saída: (2, 3, 4)
3 print(numeros[:4])       # Saída: (0, 1, 2, 3)
4 print(numeros[::2])      # Saída: (0, 2, 4, 6, 8)
```

Definição:

- Um dicionário é uma coleção desordenada de pares chave-valor em Python.

Características:

- **Mutável:** Pode-se adicionar, modificar ou remover itens.
- **Desordenado:** A ordem dos itens não é garantida.

Quando usar:

- Quando precisamos mapear valores a chaves únicas e de fácil acesso.

Criando dicionários

sintaxe básica:

```
1 nome_do_dicionario = {chave1: valor1, chave2: valor2, ...}
```

Exemplo prático:

```
1 aluno = {"nome": "Roni", "idade": 25, "curso": "Engenharia"}  
2 estoque = {"maçã": 10, "banana": 15, "laranja": 8}
```

Acessando e modificando itens em dicionários

Acessando valores:

- sintaxe: nome_do_dicionario[chave]

Exemplo:

```
1 print(aluno["nome"]) # Saída: Roni
```

Modificando valores:

- sintaxe: nome_do_dicionario[chave] = novo_valor

Exemplo:

```
1 aluno["idade"] = 26
```

`get(chave, valor_padrao):`

- Retorna o valor associado à chave, ou um valor padrão se a chave não existir.
- **Exemplo:** `aluno.get("cidade", "Não informado")`

`keys():`

- Retorna uma lista das chaves do dicionário.
- **Exemplo:** `aluno.keys()`

`values():`

- Retorna uma lista dos valores do dicionário.
- **Exemplo:** `aluno.values()`

items():

- Retorna uma lista de pares (chave, valor).
- **Exemplo:** `aluno.items()`

pop(chave):

- Remove e retorna o valor associado à chave.
- **Exemplo:** `aluno.pop("curso")`

Adicionando e removendo itens

Adicionando Itens:

- sintaxe: `nome_do_dicionario[nova_chave] = valor`

Exemplo:

```
1 aluno["cidade"] = "Recife"
```

Removendo itens:

- sintaxe: `del nome_do_dicionario[chave]`

Exemplo:

```
1 del aluno["idade"]
```

Iterando sobre dicionários

Iteração sobre chaves:

sintaxe:

```
1 for chave in dicionario:  
2     # bloco de código a ser executado para cada chave
```

Iteração sobre pares chave-valor:

sintaxe:

```
1 for chave, valor in dicionario.items():  
2     # bloco de código a ser executado para cada par
```

Exemplo prático:

```
1 for chave, valor in aluno.items():  
2     print(f"{chave}: {valor}")
```

Saída

```
1 nome: Roni  
2 curso: Engenharia  
3 cidade: Recife
```

Dicionários aninhados

Definição:

- Dicionários podem conter outros dicionários, formando uma estrutura mais complexa.

Exemplo prático:

```
1 alunos = {  
2     "aluno1": {"nome": "Roni", "idade": 25},  
3     "aluno2": {"nome": "Maria", "idade": 22},  
4 }  
5 print(alunos["aluno1"]["nome"])    # Saída: Roni
```

Tarefa 1:

- Crie uma tupla com as 12 horas de um relógio e itere sobre ela para exibir cada hora.

Tarefa 2:

- Escreva um script que solicita informações de um livro (título, autor, ano) e armazene-as em um dicionário.

Tarefa 3:

- Desenvolva um programa que recebe um dicionário de produtos e seus preços, e retorna o produto mais caro.

Revisão:

- **Compreendemos** como criar, acessar e modificar tuplas e dicionários.
- **Exploramos** métodos e operações comuns em tuplas e dicionários.
- **Aplicamos** esses conceitos em exercícios práticos.

Próximos passos:

- Na próxima semana, aprenderemos sobre entrada e saída de dados em Python, incluindo leitura e gravação de arquivos.

Leitura recomendada:

- Documentação oficial do Python: docs.python.org

Exercícios online:

- Exercícios sobre tuplas e dicionários em plataformas como HackerRank, LeetCode.

Comunidades:

- Stack Overflow, Reddit - [r/learnpython](https://www.reddit.com/r/learnpython).

Objetivos:

- **Compreender** como capturar e manipular a entrada de dados do usuário.
- **Aprender** a realizar operações de entrada e saída com arquivos.
- **Entender** o uso de funções integradas para formatação de saída.
- **Aplicar** esses conceitos em problemas práticos que envolvem interação com o usuário e manipulação de arquivos.

Entrada de dados com input()

Definição:

- A função `input()` captura a entrada do usuário como uma string.

sintaxe:

```
1 variavel = input("Mensagem para o usuário: ")
```

Exemplo prático:

```
1 nome = input("Digite seu nome: ")  
2 print(f"Olá, {nome}!")
```

Nota:

- O valor retornado por `input()` é sempre uma string; conversões de tipo podem ser necessárias.

Convertendo entrada de dados

Conversões comuns:

- Inteiro: `int()`
- Flutuante: `float()`

Exemplo prático:

```
1 idade = int(input("Digite sua idade: "))  
2 altura = float(input("Digite sua altura em metros: "))  
3 print(f"Você tem {idade} anos e {altura} metros de altura.")
```

Entrada de dados validada

Validação de entrada:

- Garantir que os dados fornecidos pelo usuário sejam válidos antes de prosseguir.

Exemplo prático:

```
1 while True:
2     idade = input("Digite sua idade: ")
3     if idade.isdigit():
4         idade = int(idade)
5         break
6     else:
7         print("Por favor, insira um número válido.")
```

Nota:

- A função `isdigit()` verifica se uma string contém apenas dígitos.

Saída de dados com print()

Definição:

- A função print() exibe a saída de dados no console.

sintaxe:

```
1 print(valor1, valor2, ..., sep=' ', end='\n')
```

Exemplo prático:

```
1 print("Python", "é", "divertido", sep="-", end="!\n")
```

Saída: Python-é-divertido!

Formatação de saída

Métodos de Formatação:

- Concatenação de Strings:

```
1 nome = "Roni"  
2 print("Olá, " + nome + "!")
```

Formatação com %:

```
1 print("Eu tenho %d anos." % 25)
```

Método .format():

```
1 print("Meu nome é {} e tenho {} anos.".format("Roni", 25))
```

F-strings (Python 3.6+):

```
1 nome = "Roni"  
2 idade = 25  
3 print(f"Meu nome é {nome} e tenho {idade} anos.")
```

Abrindo e fechando arquivos

Abertura de arquivos:

- A função `open()` abre um arquivo para leitura, gravação ou anexação.

sintaxe:

```
1 arquivo = open("nome_do_arquivo.txt", modo)
```

Modos comuns:

- `'r'`: Leitura (padrão).
- `'w'`: Escrita (sobrescreve o arquivo existente).
- `'a'`: Anexação (escreve ao final do arquivo).

Fechamento de arquivos:

- É importante fechar o arquivo após a operação.

sintaxe:

```
1 arquivo.close()
```

Leitura de arquivos

Métodos de Leitura:

- **read():** Lê o arquivo inteiro.

```
1 conteudo = arquivo.read()
```

- **readline():** Lê uma linha de cada vez.

```
1 linha = arquivo.readline()
```

- **readlines():** Lê todas as linhas e retorna uma lista.

```
1 linhas = arquivo.readlines()
```

Exemplo prático:

```
1 with open("exemplo.txt", "r") as arquivo:  
2     conteudo = arquivo.read()  
3     print(conteudo)
```

Nota: Usar `with` garante que o arquivo seja fechado automaticamente.

Escrita em arquivos

Métodos de escrita:

- **write(texto):** Escreve uma string no arquivo.

```
1 arquivo.write("Texto a ser escrito.")
```

- **writelines(lista_de_linhas):** Escreve uma lista de strings no arquivo.

```
1 arquivo.writelines(["Linha 1\n", "Linha 2\n"])
```

Exemplo prático:

```
1 with open("exemplo.txt", "w") as arquivo:  
2     arquivo.write("Olá, mundo!\n")  
3     arquivo.write("Esta é uma nova linha.")
```


Manipulação de arquivos com with

Uso de with:

- O uso de with é a maneira mais segura de abrir e manipular arquivos, garantindo que eles sejam fechados corretamente.

sintaxe:

```
1 with open("nome_do_arquivo.txt", "modo") as arquivo:  
2     # operações com o arquivo
```

Exemplo prático:

```
1 with open("dados.txt", "a") as arquivo:  
2     arquivo.write("Dados adicionados ao arquivo.\n")
```

Tarefa 1:

- Escreva um script que solicita ao usuário uma lista de compras e as salva em um arquivo de texto.

Tarefa 2:

- Desenvolva um programa que lê um arquivo de texto contendo números, um por linha, e calcula a soma desses números.

Tarefa 3:

- Crie um sistema simples de registro de notas, onde o usuário insere o nome do aluno e a nota, e o programa salva essas informações em um arquivo.

Revisão:

- **Compreendemos** como capturar entrada do usuário e realizar saídas formatadas.
- **Exploramos** a leitura e escrita de arquivos em Python, incluindo o uso seguro de with.
- **Aplicamos** esses conceitos em exercícios práticos.

Próximos passos:

- Na próxima semana, começaremos a explorar o conceito de Programação Orientada a Objetos (POO) em Python.

Leitura recomendada:

- Documentação oficial do Python: docs.python.org

Exercícios online:

- Exercícios sobre entrada e saída de dados em plataformas como HackerRank, LeetCode.

Comunidades:

- Stack Overflow, Reddit - [r/learnpython](https://r.python.org).

Objetivos:

- **Compreender** as operações fundamentais para manipulação de arquivos em Python.
- **Aprender** a ler, escrever e manipular diferentes tipos de arquivos.
- **Explorar** operações avançadas como leitura e escrita de arquivos CSV e JSON.
- **Aplicar** esses conceitos em problemas práticos envolvendo manipulação de dados.

Abertura de arquivos

Revisão:

- A função `open()` é usada para abrir arquivos para leitura, gravação ou anexação.

sintaxe:

```
1 arquivo = open("nome_do_arquivo.txt", "modo")
```

Modos comuns:

- `'r'`: Leitura (padrão).
- `'w'`: Escrita (sobrescreve o arquivo existente).
- `'a'`: Anexação (adiciona ao final do arquivo).
- `'b'`: Modo binário (para arquivos binários como imagens).

Fechamento:

- Use `arquivo.close()` para fechar o arquivo após a operação.

Manipulando arquivos CSV

O que é um arquivo CSV:

- **CSV** (Comma-Separated Values) é um formato de arquivo usado para armazenar dados tabulares.

Leitura de arquivos CSV:

- Uso do módulo `csv`:

```
1 import csv
2
3 with open("dados.csv", "r") as arquivo:
4     leitor = csv.reader(arquivo)
5     for linha in leitor:
6         print(linha)
```

Escrita em arquivos CSV:

- Exemplo:

```
1 with open("dados.csv", "w", newline='') as arquivo:
2     escritor = csv.writer(arquivo)
3     escritor.writerow(["Nome", "Idade", "Cidade"])
4     escritor.writerow(["Roni", 25, "Recife"])
```

Manipulando arquivos JSON

O que é um arquivo JSON:

- **JSON** (JavaScript Object Notation) é um formato leve de troca de dados.

Leitura de arquivos JSON:

- **Uso do módulo json:**

```
1 import json
2
3 with open("dados.json", "r") as arquivo:
4     dados = json.load(arquivo)
5     print(dados)
```

Escrita em arquivos JSON:

- **Exemplo:**

```
1 dados = {"nome": "Roni", "idade": 25, "cidade": "Recife"}
2
3 with open("dados.json", "w") as arquivo:
4     json.dump(dados, arquivo)
```


Manipulação de arquivos binários

O que é um Arquivo Binário:

- Arquivos binários são qualquer arquivo que não seja de texto, como imagens, vídeos, executáveis, etc.

Leitura de Arquivos Binários:

- Exemplo:

```
1 with open("imagem.jpg", "rb") as arquivo:  
2     conteudo = arquivo.read()
```

Escrita em Arquivos Binários:

- Exemplo:

```
1 with open("copia_imagem.jpg", "wb") as arquivo:  
2     arquivo.write(conteudo)
```

Operações com arquivos e diretórios

Uso do módulo os:

- `os.path.exists(caminho)`: Verifica se um arquivo ou diretório existe.
- `os.remove(caminho)`: Remove um arquivo.
- `os.mkdir(caminho)`: Cria um novo diretório.
- `os.listdir(caminho)`: Lista todos os arquivos e diretórios em um diretório.

Exemplo prático:

```
1 import os
2
3 if not os.path.exists("novo_diretorio"):
4     os.mkdir("novo_diretorio")
5
6 arquivos = os.listdir(".")
7 print(arquivos)
```

Tarefa 1:

- Escreva um script que lê um arquivo CSV contendo dados de alunos (nome, idade, nota) e calcula a média das notas.

Tarefa 2:

- Desenvolva um programa que lê um arquivo JSON com informações de um catálogo de produtos e permite ao usuário buscar um produto pelo nome.

Tarefa 3:

- Crie um script que copia uma imagem de um diretório para outro e renomeia a cópia.

Revisão:

- **Compreendemos** as operações fundamentais de manipulação de arquivos, incluindo leitura e escrita de arquivos CSV e JSON.
- **Exploramos** a manipulação de arquivos binários e operações com diretórios.
- **Aplicamos** esses conceitos em exercícios práticos.

Próximos Passos:

- Na próxima semana, começaremos a estudar Programação Orientada a Objetos (POO) em Python, uma abordagem poderosa para organizar e estruturar seu código.

Objetivos:

- **Compreender** os conceitos fundamentais da Programação Orientada a Objetos (POO).
- **Aprender** a definir classes, objetos, atributos e métodos em Python.
- **Explorar** conceitos como encapsulamento, herança e polimorfismo.
- **Aplicar** POO para resolver problemas práticos e organizar o código de maneira eficiente.



Definição



Conceitos
Principais



Vantagens



Exemplo Real

- **Definição:**

- A Programação Orientada a Objetos é um paradigma de programação que usa "objetos" – que são instâncias de classes – para modelar e representar entidades do mundo real ou conceitos.

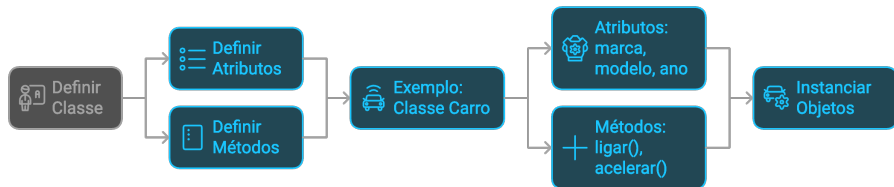
- **Conceito:**

- **Objeto:** Uma instância de uma classe.
- **Classe:** Um molde ou definição para criar objetos.
- **Atributo:** Propriedade ou característica de um objeto.
- **Método:** Função associada a um objeto.

- **Vantagem:**

- Organiza o código de maneira modular e reutilizável, facilitando a manutenção e expansão.

O que é Programação Orientada a Objetos (POO)?



Definindo classes e objetos

Classe:

- Uma classe define a estrutura e o comportamento de objetos.

sintaxe:

```
1 class NomeDaClasse:  
2     # atributos e métodos
```

Objeto:

- Um objeto é uma instância de uma classe.

sintaxe:

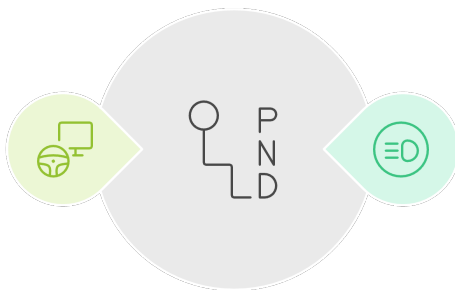
```
1 objeto = NomeDaClasse()
```

Exemplo prático:

```
1 class Carro:  
2     pass  
3  
4 meu_carro = Carro()
```

Componentes de uma Classe Carro

Métodos
Ações que o carro
pode realizar



Atributos
Características que
definem o estado do
carro

Atributos de Instância

Definição:

- Atributos de instância são variáveis associadas a cada objeto individualmente.

Inicializando Atributos com `__init__()`:

- O método `__init__()` é o construtor da classe e é chamado automaticamente quando um objeto é criado.

sintaxe:

```
1 class Carro:
2     def __init__(self, marca, modelo):
3         self.marca = marca
4         self.modelo = modelo
```

Continua na próxima página...

Exemplo prático:

```
1 class Carro:
2     def __init__(self, marca, modelo):
3         self.marca = marca
4         self.modelo = modelo
5
6 meu_carro = Carro("Toyota", "Corolla")
7 print(meu_carro.marca, meu_carro.modelo)    # Saída: Toyota
        Corolla
```

Métodos

Definição:

- Métodos são funções definidas dentro de uma classe, que descrevem os comportamentos dos objetos.

sintaxe:

```
1 class NomeDaClasse:
2     def metodo(self):
3         # código do método
```

Exemplo prático:

```
1 class Carro:
2     def __init__(self, marca, modelo):
3         self.marca = marca
4         self.modelo = modelo
5
6     def exibir_info(self):
7         print(f"Carro: {self.marca} {self.modelo}")
8
9 meu_carro = Carro("Toyota", "Corolla")
10 meu_carro.exibir_info() # Saída: Carro: Toyota Corolla
```

Atributos e métodos de classe

Atributos de classe:

- Pertencem à classe e são compartilhados por todas as instâncias.

sintaxe:

```
1 class Carro:  
2     rodas = 4 # atributo de classe
```

Métodos de classe:

- Definidos com o decorador @classmethod.

Exemplo:

```
1 class Carro:  
2     rodas = 4  
3  
4     @classmethod  
5     def exibir_rodas(cls):  
6         print(f"Todos os carros têm {cls.rodas} rodas.")  
7  
8 Carro.exibir_rodas() # Saída: Todos os carros têm 4 rodas.
```

Conceitos avançados de POO

Encapsulamento

- O encapsulamento esconde detalhes internos de um objeto e protege seus dados.

Atributos Privados:

- Atributos podem ser tornados privados prefixando seus nomes com dois underscores `__`.

Exemplo:

```
1 class ContaBancaria:
2     def __init__(self, saldo):
3         self.__saldo = saldo
4
5     def exibir_saldo(self):
6         print(f"Saldo: R$ {self.__saldo}")
7
8 conta = ContaBancaria(1000)
9 conta.exibir_saldo()    # Saída: Saldo: R$ 1000
```

Encapsulamento em Python

Consulta de Saldo

Fornecer uma maneira de visualizar o saldo da conta, destacando o uso de métodos para acessar dados privados.



Atributo Privado

Representa a natureza privada do atributo `__saldo`, inacessível diretamente de fora da classe.



Método de Saque

Facilita a remoção de fundos da conta, garantindo que o atributo privado seja modificado de forma segura.



Método de Depósito

Permite adicionar fundos à conta, demonstrando acesso controlado ao atributo privado.



Herança

- Herança permite que uma classe herde atributos e métodos de outra classe, promovendo a reutilização de código.
- **Classe base e classe derivada:**

sintaxe:

```
1 class ClasseBase:  
2     # atributos e métodos  
3  
4 class ClasseDerivada(ClasseBase):  
5     # atributos e métodos adicionais
```

Continua...

Exemplo prático:

```
1 class Veiculo:
2     def __init__(self, marca):
3         self.marca = marca
4
5     def mover(self):
6         print("O veículo está se movendo.")
7
8 class Carro(Veiculo):
9     def __init__(self, marca, modelo):
10         super().__init__(marca)
11         self.modelo = modelo
12
13     def exibir_info(self):
14         print(f"Carro: {self.marca} {self.modelo}")
15
16 meu_carro = Carro("Toyota", "Corolla")
17 meu_carro.exibir_info()  # Saída: Carro: Toyota Corolla
```

Polimorfismo

Função fazer_barulho

Uma função que chama o método de som apropriado com base no tipo do objeto.



Classe Gato

Uma subclasse que representa um gato, implementando o som do animal.

Classe Animal

A classe base que define uma interface comum para todos os animais.

Classe Cachorro

Uma subclasse que representa um cachorro, implementando o som do animal.

Polimorfismo permite que métodos em diferentes classes compartilhem o mesmo nome, mas tenham comportamentos diferentes.

Continua...

Exemplo prático:

```
1 class Animal:
2     def fazer_som(self):
3         pass
4
5 class Cachorro(Animal):
6     def fazer_som(self):
7
8 class Gato(Animal):
9     def fazer_som(self):
10
11     def fazer_barulho(animal):
12         animal.fazer_som()
13
14 cachorro = Cachorro()
15 gato = Gato()
16
17 fazer_barulho(cachorro)
18 fazer_barulho(gato)
```

Método `__str__()`

Definição:

- O método `__str__()` define o que será exibido quando o objeto for convertido em string ou impresso.

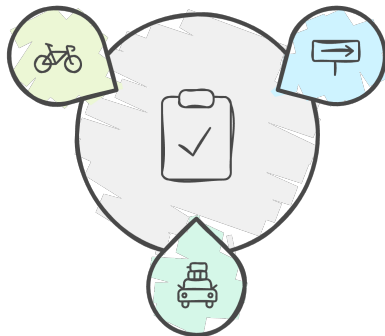
Exemplo prático:

```
1 class Carro:
2     def __init__(self, marca, modelo):
3         self.marca = marca
4         self.modelo = modelo
5
6     def __str__(self):
7         return f"{self.marca} {self.modelo}"
8
9 meu_carro = Carro("Toyota", "Corolla")
10 print(meu_carro)    # Saída: Toyota Corolla
```

Métodos Abstratos em Subclasses

Bicicleta

A subclasse que implementa o método mover para bicicletas



MeioDeTransporte

A classe abstrata que define o contrato para movimento

Carro

A subclasse que implementa o método mover para carros

Abstração

- A abstração permite criar uma estrutura genérica focada nas funcionalidades essenciais.
- Em Python, é possível criar classes abstratas usando a classe base ABC e decoradores como @abstractmethod.
- **Classes e métodos abstratos:**
 - Uma classe abstrata define métodos que devem ser implementados pelas classes derivadas.
 - No exemplo abaixo, MeioDeTransporte é uma classe abstrata com o método mover(), que deve ser implementado por qualquer classe derivada.

Sintaxe:

```
1 from abc import ABC, abstractmethod
2
3 class MeioDeTransporte(ABC):
4     @abstractmethod
5     def mover(self):
6         pass
```

Implementação em classes concretas:


```
1 class Carro(MeioDeTransporte):
2     def mover(self):
3         print("O carro está se movendo.")
4
5 class Bicicleta(MeioDeTransporte):
6     def mover(self):
7         print("A bicicleta está se movendo.")
```

Prática:

- Crie instâncias das classes Carro e Bicicleta e passe-as para a função abaixo:

```
1 def iniciar_movimento(transporte):
2     transporte.mover()
```

Resultado esperado: O método mover() é executado com base na classe concreta passada, ilustrando o conceito de abstração.

Tarefa 1:

- Crie uma classe Pessoa com atributos nome e idade. Crie uma instância da classe e exiba as informações.

Tarefa 2:

- Desenvolva uma classe Funcionario que herda de Pessoa e adicione um atributo salario. Crie um método que exibe o salário do funcionário.

Tarefa 3:

- Crie um sistema simples de gerenciamento de veículos, onde diferentes tipos de veículos (como carro e moto) herdam de uma classe Veiculo. Implemente polimorfismo para os métodos de movimento de cada veículo.

Revisão:

- **Compreendemos** os conceitos fundamentais da Programação Orientada a Objetos (POO) em Python.
- **Exploramos** classes, objetos, atributos, métodos e conceitos avançados como herança e polimorfismo.
- **Aplicamos** POO em exercícios práticos, organizando o código de maneira modular e eficiente.

Próximos Passos:

- Na próxima semana, vamos aprofundar a utilização de POO com mais exemplos práticos, incluindo o uso de herança múltipla e tratamento de exceções em POO.

Leitura recomendada:

- Documentação oficial do Python: docs.python.org

Exercícios online:

- Exercícios sobre POO em plataformas como HackerRank, LeetCode.

Comunidades:

- Stack Overflow, Reddit - r/learnpython.

Objetivos:

- **Compreender** o conceito de módulos e pacotes em Python.
- **Aprender** a criar e importar módulos para organizar e reutilizar código.
- **Explorar** pacotes e a estrutura de diretórios em projetos Python.
- **Aplicar** esses conceitos para modularizar projetos e gerenciar dependências de forma eficiente.

O que é um módulo?

Definição:

- Um módulo é um arquivo contendo código Python (funções, classes, variáveis) que pode ser importado e reutilizado em outros scripts ou programas.

Vantagens:

- Facilita a organização do código.
- Promove a reutilização e a manutenção do código.
- Reduz a complexidade dos programas ao dividir o código em partes menores e gerenciáveis.

Exemplo:

- Arquivo `meu_modulo.py` com funções que podem ser usadas em diferentes projetos.

Importando módulos

Importando módulos: sintaxe:

```
1 import nome_do_modulo
```

Exemplo prático:

```
1 import math
2 print(math.sqrt(16))    # Saída: 4.0
```

Importando funções específicas: sintaxe

```
1 from nome_do_modulo import nome_da_funcao
```

Exemplo:

```
1 from math import sqrt
2 print(sqrt(16))    # Saída: 4.0
```

Criando seu próprio módulo

Criando um módulo:

- Um módulo é simplesmente um arquivo Python (.py) contendo funções, classes e variáveis que podem ser importadas.

Exemplo:

- Crie um arquivo chamado meu_modulo.py com o seguinte código:

```
1 def saudacao(nome):  
2     return f"Olá, {nome}!"  
3  
4 def soma(a, b):  
5     return a + b
```

Usando o módulo:

- Importação:

```
1 import meu_modulo  
2 print(meu_modulo.saudacao("Roni"))    # Saída: Olá, Roni!  
3 print(meu_modulo.soma(5, 3))          # Saída: 8
```


Usando `__name__ == "__main__"`

Definição:

- `__name__` é uma variável especial em Python que define o contexto em que um módulo está sendo executado.
- Se `__name__ == "__main__"` for True, o módulo está sendo executado diretamente e não importado.

Por que usar?:

- Permite que um módulo execute código de teste quando executado diretamente, mas oculte esse código quando importado como um módulo.

Exemplo Prático:

```
1 def saudacao(nome):  
2     return f"Olá, {nome}!"  
3  
4 if __name__ == "__main__":  
5     print(saudacao("Roni"))
```

Introdução aos pacotes

O que é um Pacote?

- Um pacote é uma coleção de módulos organizados em uma estrutura de diretórios.
- Cada diretório contendo um arquivo `__init__.py` é tratado como um pacote em Python.

Vantagens:

- Organiza melhor módulos relacionados.
- Facilita a manutenção e o desenvolvimento de projetos maiores.
- Permite a criação de hierarquias de módulos.

Exemplo de pacote:

- Estrutura de diretórios:

```
1 meu_pacote/  
2     __init__.py  
3     modulo1.py  
4     modulo2.py
```

Criando e usando pacotes

Criando um pacote:

- Crie um diretório com um arquivo `__init__.py` vazio ou com código de inicialização.
- Adicione módulos (arquivos `.py`) dentro desse diretório.

Estrutura de exemplo:

- Diretório `meu_pacote` com os arquivos:

```
1 meu_pacote/  
2     __init__.py  
3     calculadora.py  
4     saudacoes.py
```

Arquivo `calculadora.py`

```
1 def soma(a, b):  
2     return a + b
```

Continua...

Arquivo saudacoes.py

```
1 def saudacao(nome):  
2     return f"Olá, {nome}!"
```

Usando o Pacote:

- Importação:

```
1 from meu_pacote.calculadora import soma  
2 from meu_pacote.saudacoes import saudacao  
3  
4 print(soma(5, 3))           # Saída: 8  
5 print(saudacao("Roni"))    # Saída: Olá, Roni!
```

Importando de pacotes

Importando módulos de um pacote: sintaxe:

```
1 import pacote.modulo
```

Exemplo:

```
1 import meu_pacote.calculadora
2 print(meu_pacote.calculadora.soma(5, 3))  # Saída: 8
```

Importando funções específicas: sintaxe:

```
1 from pacote.modulo import funcao
```

Exemplo:

```
1 from meu_pacote.saudacoes import saudacao
2 print(saudacao("Roni"))  # Saída: Olá, Roni!
```

Gerenciando dependências e pacotes externos

Instalando Pacotes Externos com pip

- pip é o instalador de pacotes padrão para Python, usado para instalar e gerenciar pacotes Python que não são distribuídos com o Python padrão.

Instalando um Pacote: sintaxe:

```
1 pip install nome_do_pacote
```

Exemplo:

```
1 pip install requests
```

Usando um pacote externo:

```
1 import requests
2 response = requests.get("https://api.github.com")
3 print(response.status_code)
```

Criando e distribuindo seus próprios pacotes

Estrutura Básica para um Pacote Distribuível:

Diretório do projeto:

```
1 meu_projeto/  
2     meu_pacote/  
3         __init__.py  
4         modulo1.py  
5         modulo2.py  
6     setup.py
```

Arquivo setup.py:

```
1 from setuptools import setup, find_packages  
2  
3 setup(  
4     name="meu_pacote",  
5     version="0.1",  
6     packages=find_packages(),  
7     install_requires=[],  
8 )
```

Continua...

Distribuindo o pacote:

Passos:

- 1 Execute `python setup.py sdist` para criar um pacote distribuível.
- 2 Distribua através do PyPI ou use `pip install` para instalá-lo localmente.

Tarefa 1:

- Crie um módulo chamado `operacoes.py` que contenha funções básicas de matemática (adição, subtração, multiplicação, divisão). Importe e use esse módulo em um script separado.

Tarefa 2:

- Desenvolva um pacote chamado `meu_calculador` com dois módulos: `aritmetica.py` (com funções aritméticas) e `geometria.py` (com funções para cálculo de área e perímetro). Importe e use as funções desses módulos em um script separado.

Tarefa 3:

- Instale um pacote externo (por exemplo, `requests`) usando `pip` e use-o para fazer uma solicitação HTTP a um site de sua escolha.

Revisão:

- **Compreendemos** o conceito de módulos e pacotes em Python.
- **Exploramos** como criar e usar módulos e pacotes para organizar e reutilizar código.
- **Aprendemos** a instalar pacotes externos com pip e a distribuir nossos próprios pacotes.
- **Aplicamos** esses conceitos em exercícios práticos para modularizar projetos e gerenciar dependências.

Próximos passos:

- Na próxima semana, exploraremos o tratamento de exceções em Python para melhorar a robustez e a confiabilidade do código.

Leitura recomendada:

- Documentação oficial do Python: docs.python.org

Exercícios online:

- Exercícios sobre módulos e pacotes em plataformas como HackerRank, LeetCode.

Comunidades:

- Stack Overflow, Reddit - [r/learnpython](https://r.python.org).

Objetivos:

- **Compreender** o conceito de exceções em Python e por que elas ocorrem.
- **Aprender** a usar blocos `try`, `except`, `else`, e `finally` para capturar e tratar exceções.
- **Explorar** a criação de exceções personalizadas para situações específicas.
- **Aplicar** o tratamento de exceções em exemplos práticos para melhorar a robustez e confiabilidade do código.

Introdução ao tratamento de exceções

O que é uma Exceção?

- Uma exceção é um evento que ocorre durante a execução de um programa e interrompe o fluxo normal do código.
- Exceções ocorrem geralmente devido a erros como divisão por zero, tentativa de acessar um índice inexistente em uma lista, ou erro de sintaxe.

Importância:

- Capturar e tratar exceções é essencial para evitar que o programa quebre inesperadamente e para lidar de forma controlada com situações de erro.

Exemplo de Exceção:

```
1 numero = int("texto") # Gera um ValueError
```

Usando blocos try e except

Bloco try e except:

- O bloco try permite que você teste um bloco de código que pode gerar uma exceção.
- O bloco except permite que você capture e trate a exceção.

sintaxe:

```
1 try:
2     # código que pode gerar uma exceção
3 except TipoDeExcecao:
4     # código que será executado se ocorrer uma exceção
```

Exemplo prático:

```
1 try:
2     numero = int(input("Digite um número: "))
3     resultado = 10 / numero
4     print(f"Resultado: {resultado}")
5 except ZeroDivisionError:
6     print("Erro: Divisão por zero não é permitida.")
7 except ValueError:
8     print("Erro: Por favor, insira um número válido.")
```

Bloco else:

- Executado se nenhum erro for encontrado no bloco try.

sintaxe:

```
1 try:
2     # código que pode gerar uma exceção
3 except TipoDeExcecao:
4     # código que será executado se ocorrer uma exceção
5 else:
6     # código que será executado se não ocorrer uma exceção
```

Bloco finally:

- Sempre executado, independentemente de ocorrer uma exceção ou não. Usado para liberar recursos ou executar código de limpeza.

sintaxe:

```
1 try:
2     # código que pode gerar uma exceção
3 except TipoDeExcecao:
4     # código que será executado se ocorrer uma exceção
5 finally:
6     # código que sempre será executado
```

Exemplo prático:

```
1 try:
2     arquivo = open("dados.txt", "r")
3     conteudo = arquivo.read()
4 except FileNotFoundError:
5     print("Erro: Arquivo não encontrado.")
6 else:
7     print(conteudo)
8 finally:
9     arquivo.close()
10    print("Arquivo fechado.")
```


Capturando múltiplas exceções

Múltiplas Exceções:

- Você pode capturar diferentes tipos de exceções no mesmo bloco try usando vários blocos except.

sintaxe:

```
1 try:
2     # código que pode gerar exceções
3 except TipoDeExcecao1:
4     # código para tratar TipoDeExcecao1
5 except TipoDeExcecao2:
6     # código para tratar TipoDeExcecao2
```

Exemplo prático:

```
1 try:
2     numero = int(input("Digite um número: "))
3     resultado = 10 / numero
4 except (ZeroDivisionError, ValueError) as e:
5     print(f"Ocorreu um erro: {e}")
```

Criando exceções personalizadas

Definição:

- Você pode criar suas próprias exceções em Python para lidar com situações específicas do seu programa.

sintaxe:

```
1 class MinhaExcecao(Exception):  
2     pass
```

Continua...

Exemplo prático:

```
1 class SaldoInsuficienteError(Exception):
2     def __init__(self, mensagem="Saldo insuficiente para
3         realizar a operação."):
4         self.mensagem = mensagem
5         super().__init__(self.mensagem)
6
7 def sacar(saldo, valor):
8     if valor > saldo:
9         raise SaldoInsuficienteError()
10    saldo -= valor
11    return saldo
12
13 try:
14     saldo_atual = 100
15     saldo_atual = sacar(saldo_atual, 150)
16 except SaldoInsuficienteError as e:
17     print(e)
```

Aplicando tratamento de exceções em projetos

Melhores práticas para tratamento de exceções

Seja específico:

- Capture exceções específicas em vez de usar um except genérico. Isso ajuda a evitar a captura de exceções inesperadas e facilita a depuração.

Evite silenciar exceções:

- Não ignore exceções sem registrar ou lidar com elas. Isso pode mascarar problemas importantes no código.

Use finally para limpeza:

- Sempre libere recursos (como arquivos ou conexões de rede) no bloco finally para garantir que eles sejam liberados, independentemente de ocorrer uma exceção.

Documente exceções:

- Documente as exceções que uma função pode lançar, para que os usuários do código saibam como lidar com possíveis erros.



Exemplo prático de tratamento de exceções em um projeto

Projeto: Sistema de Gerenciamento de Biblioteca (SGB):

- **Descrição:** Um sistema simples que permite o empréstimo e devolução de livros, com controle de estoque.

Cenários de exceção:

- Tentativa de emprestar um livro que não está disponível.
- Devolução de um livro que não foi emprestado.
- Tentativa de acessar um livro que não existe no sistema.

Continua...

Exemplo de código:

```
1 class LivroIndisponivelError(Exception):
2     pass
3
4 class LivroNaoEncontradoError(Exception):
5     pass
6
7 class Biblioteca:
8     def __init__(self):
9         self.livros = {"Python Básico": 3, "POO Avançado":
10                        2}
11
12     def emprestar_livro(self, titulo):
13         if titulo not in self.livros:
14             raise LivroNaoEncontradoError(f"Livro '{titulo}'
15             não encontrado.")
16         if self.livros[titulo] == 0:
17             raise LivroIndisponivelError(f"Livro '{titulo}'
18             está indisponível.")
19         self.livros[titulo] -= 1
20         print(f"Livro '{titulo}' emprestado com sucesso.")
21
22     def devolver_livro(self, titulo):
```

```
20         if titulo not in self.livros:
21             raise LivroNaoEncontradoError(f"Livro '{titulo}'
22             não encontrado.")
23             self.livros[titulo] += 1
24             print(f"Livro '{titulo}' devolvido com sucesso.")
25
26 biblioteca = Biblioteca()
27 try:
28     biblioteca.emprestar_livro("Python Básico")
29     biblioteca.emprestar_livro("Python Avançado")
30 except (LivroIndisponivelError, LivroNaoEncontradoError) as
    e:
        print(e)
```

Tarefa 1:

- Crie um programa que leia números de um arquivo e calcule a média. Implemente tratamento de exceções para lidar com possíveis erros, como arquivo não encontrado ou dados inválidos.

Tarefa 2:

- Desenvolva um sistema simples de login que trate exceções como usuário não encontrado e senha incorreta. Crie exceções personalizadas para essas situações.

Tarefa 3:

- Implemente um sistema de carrinho de compras que capture exceções relacionadas a produtos fora de estoque e preços inválidos. Utilize blocos try, except, else, e finally para garantir o funcionamento robusto do sistema.

Revisão:

- Compreendemos o conceito de exceções e como elas afetam o fluxo de um programa.
- Exploramos o uso de blocos `try`, `except`, `else`, e `finally` para capturar e tratar exceções.
- Aprendemos a criar exceções personalizadas para lidar com situações específicas.
- Aplicamos o tratamento de exceções em exemplos práticos para melhorar a robustez e a confiabilidade do código.

Próximos Passos:

- Na próxima semana, concluiremos o curso com uma revisão geral dos principais conceitos abordados e discutiremos práticas recomendadas para escrever código Python de alta qualidade.

Leitura recomendada:

- Documentação oficial do Python: docs.python.org

Exercícios online:

- Exercícios sobre tratamento de exceções em plataformas como HackerRank, LeetCode.

Comunidades:

- Stack Overflow, Reddit - [r/learnpython](https://www.reddit.com/r/learnpython).

Objetivos:

- Introdução ao framework Django.
- Entender sua estrutura e os principais componentes.
- Criar o primeiro projeto Django.

Nota:

- Django: O framework para perfeccionistas com prazos.

O que é Django?

Django é um framework web de alto nível em Python que promove o desenvolvimento rápido e um design limpo e pragmático.

Principais características:

- **Rápido:** Facilitando o desenvolvimento de aplicações web rapidamente.
- **Seguro:** Inclui segurança contra ataques comuns, como CSRF, XSS, SQL Injection.
- **Escalável:** Projetado para escalar de pequenos projetos para aplicações complexas.
- **Versátil:** Usado para tudo, de sistemas de gerenciamento de conteúdo a aplicações científicas e de dados.

História e popularidade

- **Lançamento:** Criado em 2003, oficialmente lançado em 2005.
- **Uso em grandes empresas:** Utilizado por empresas como Instagram, Spotify, YouTube, The Washington Post.
- **Popularidade:** Classificado entre os principais frameworks web devido à sua simplicidade e robustez.

Configurando o ambiente de desenvolvimento

Passos para instalação:

- **Instalação do Python:** Verifique se o Python está instalado usando `python -version`.
- **Instalação do Django:** Usar o comando `pip install django`.
- **Verificação:** Checar a instalação usando `django-admin -version`.

Nota: Sempre usar um ambiente virtual (venv) para isolar as dependências do projeto.

Criando um projeto Django

Comando para criar um projeto:

No terminal, execute `django-admin startproject config`

Estrutura do projeto criado:

- `config/`: diretório raiz do projeto.
 - `manage.py`: script de linha de comando para interações administrativas.
 - `config/`: diretório interno com configuração principal.
 - `__init__.py`: marca o diretório como um pacote Python.
 - `settings.py`: configurações do projeto.
 - `urls.py`: definições de URL para o projeto.
 - `wsgi.py`: entrada para servidores WSGI.
 - `asgi.py`: entrada para servidores ASGI (opcional).

Explicação dos componentes do projeto

`manage.py`:

- Utilizado para várias operações, como iniciar um servidor de desenvolvimento, criar novas aplicações, realizar migrações, entre outros.

`settings.py`:

- Contém todas as configurações do projeto, como configuração de banco de dados, configurações de segurança, templates, middleware, etc.

`urls.py`:

- Mapeamento de URLs do projeto para diferentes aplicações e views.

`wsgi.py` e `asgi.py`:

- Configurações de entrada para servidores web, garantindo que a aplicação possa ser servida.

Configurando e rodando o servidor de desenvolvimento

Step:

- 1 Navegar para o diretório do projeto: `cd config.`
- 2 Rodar o servidor de desenvolvimento: `python manage.py runserver.`

Resultado esperado:

- Uma mensagem no terminal indicando que o servidor está rodando em `http://127.0.0.1:8000/`.
- Acessar este endereço no navegador deve exibir uma página padrão do Django, confirmando que a instalação foi bem-sucedida.

O que são aplicações em Django?

- Módulos independentes dentro de um projeto, cada um responsável por uma parte específica da lógica. Permite reuso e modularização de código.

Comando para criar uma nova aplicação:

- Dentro do diretório do projeto: `python manage.py startapp tarefa`.

Estrutura de diretórios da aplicação:

- Diretório `blog/` com arquivos como `models.py`, `views.py`, `admin.py`, `tests.py`.

Ligando a aplicação ao projeto

Passos para configurar a aplicação:

- 1 Adicionar a aplicação ao projeto em settings.py:

```
1 INSTALLED_APPS = [  
2     ...,  
3     'tarefa',  
4 ]
```

- 1 Definir URLs da aplicação no arquivo principal urls.py:

```
1 from django.urls import include, path  
2  
3 urlpatterns = [  
4     path('blog/', include('tarefa.urls')),  
5     path('admin/', admin.site.urls),  
6 ]
```

Tarefa 1:

- Criar um novo projeto Django chamado.

Tarefa 2:

- Rodar o servidor de desenvolvimento e acessar a página inicial.

Tarefa 3:

- Criar uma nova aplicação dentro do projeto.

Tarefa 4:

- Configurar a nova aplicação para ser reconhecida pelo projeto principal.

Revisão:

- Introdução ao Django como um framework web robusto e escalável.
- Criação de um projeto Django e entendimento da sua estrutura de diretórios e arquivos.
- Execução do servidor de desenvolvimento e configuração básica de aplicações.

Dica: Sempre testar as alterações usando o servidor de desenvolvimento antes de mover para produção.

Leitura recomendada:

- Documentação oficial do Django: docs.djangoproject.com

Comunidades:

- Stack Overflow, Django Girls Tutorial.

Objetivos:

- Entender o conceito de aplicações dentro de um projeto Django.
- Aprender a configurar URLs e criar views para exibir dados.
- Implementar uma estrutura básica de uma aplicação Django.

Nota: Django torna fácil criar e organizar funcionalidades em módulos reutilizáveis.

Revisão da estrutura de projeto Django

Projeto Django:

- O ponto central que agrupa várias aplicações.
- Inclui configurações globais (settings.py), URLs principais (urls.py), e arquivos de gerenciamento (manage.py).

Aplicações Django:

- Módulos individuais que encapsulam funcionalidades específicas.
- Cada aplicação possui seu próprio conjunto de modelos, views, templates e arquivos estáticos.

Criando uma nova aplicação em Django

O que é uma aplicação?

- Um módulo autônomo e reutilizável que adiciona funcionalidades ao projeto Django.
- Pode ser reutilizado em diferentes projetos.

Comando para criar uma aplicação: No terminal, dentro do diretório do projeto, executar:

```
python manage.py startapp tarefa
```

Estrutura de diretórios da aplicação:

- blog/
 - migrations/
 - __init__.py
 - admin.py
 - apps.py
 - models.py
 - tests.py
 - views.py

Integrando a aplicação ao projeto principal

Passo 1: Registrar a aplicação em settings.py

- Adicionar o nome da aplicação na lista de INSTALLED_APPS

```
1 INSTALLED_APPS = [  
2     ...,  
3     'tarefa',  
4 ]
```

Passo 2: Configurar URLs para a aplicação

- Criar um arquivo urls.py na pasta da aplicação (blog/urls.py):

Continua...

```
1 from django.urls import path
2 from . import views
3
4 urlpatterns = [
5     path('', views.index, name='index'),
6 ]
```

- Incluir as URLs da aplicação no urls.py do projeto principal:

```
1 from django.urls import include, path
2
3 urlpatterns = [
4     path('tarefa/', include('tarefa.urls')),
5     path('admin/', admin.site.urls),
6 ]
```

O que são Views?

- Componentes que contêm a lógica que é executada quando uma URL específica é acessada.
- As views pegam dados do modelo e os enviam para os templates.

Tipos de Views:

- **Funções:** Definidas como funções Python simples.
- **Classes:** Usando class-based views para lógica mais complexa.

Criando uma View básica

Exemplo de View simples em `blog/views.py`:

```
1 from django.http import HttpResponse
2
3 def index(request):
4     return HttpResponse("Bem-vindo a To-Do List!")
```

Step:

- 1 Importar `HttpResponse` do módulo `django.http`.
- 2 Definir uma função `index` que aceita um `request` como parâmetro.
- 3 Retornar um objeto `HttpResponse` com uma mensagem de texto.

Ligando a View à URL

Configuração de URL para a View index:

Em `blog/urls.py`:

```
1 from django.urls import path
2 from . import views
3
4 urlpatterns = [
5     path('', views.index, name='index'),
6 ]
```

Verificando no navegador:

- Iniciar o servidor de desenvolvimento: `python manage.py runserver`.
- Acessar `http://127.0.0.1:8000/blog/` para ver a mensagem "Bem-vindo ao Blog!".

Renderizando templates com Views

Uso de Templates:

Em vez de retornar uma `HttpResponse` simples, usamos templates para HTML estruturado.

Configurando a View para usar um Template:

```
1 from django.shortcuts import render
2
3 def index(request):
4     return render(request, 'index.html')
```

Step:

- Importar a função `render` do módulo `django.shortcuts`.
- Usar `render(request, 'template.html')` para buscar e renderizar um template.



Criando templates para a aplicação

Estrutura de Templates:

- Dentro da pasta blog/, criar um diretório templates/.
- Dentro de templates/, criar uma subpasta blog/.
- Criar o arquivo index.html dentro de blog/templates/blog/.

Exempo do index.html

```
1 <!DOCTYPE html>
2 <html>
3 <head>
4     <title>Lista de Tarefas</title>
5 </head>
6 <body>
7     <h1>Bem-vindo a Minha Lista de Tarefas!</h1>
8     <p>Este é o conteúdo da página inicial.</p>
9 </body>
10 </html>
```

Revisão:

- Criação e configuração de novas aplicações dentro de um projeto Django.
- Entendimento de views e sua integração com URLs.
- Uso de templates para renderizar páginas HTML.
- Dica Importante: Sempre verificar o mapeamento correto de URLs para evitar erros ao acessar views.

Leitura recomendada:

- Tutorial sobre templates
- Documentação oficial do Django sobre views

Comunidades:

- Stack Overflow, Django Girls Tutorial.

Objetivos:

- Compreender o sistema de templates do Django.
- Aprender a utilizar a herança de templates para criar layouts reutilizáveis.
- Implementar templates para melhorar a organização e manutenção do código.

Nota:

- Com Django Templates, você separa a lógica da apresentação e cria um código mais limpo e organizado.

O que são templates em Django?

Templates são arquivos de texto que definem a estrutura e layout de uma página. Eles podem incluir HTML, CSS, JavaScript e sintaxe específica de Django.

Funcionalidade dos templates:

- Permitir a exibição dinâmica de conteúdo.
- Separar a lógica de negócios da apresentação visual.
- Facilitar a reutilização de código e organização de layouts.

Configurando templates no projeto Django

Passo 1: Estrutura de diretórios

- Criar um diretório chamado templates dentro de cada aplicação (ex.: blog/templates/blog).
- Alternativamente, criar um diretório global de templates no diretório raiz do projeto.

Passo 2: Configuração no settings.py

- Adicionar o caminho para templates na variável TEMPLATES:

Continua...

Continuação

```
1 TEMPLATES = [  
2     {  
3         'BACKEND': 'django.template.backends.django.  
DjangoTemplates',  
4         'DIRS': ['templates'],  
5         'APP_DIRS': True,  
6         'OPTIONS': {  
7             'context_processors': [  
8                 'django.template.context_processors.debug',  
9                 'django.template.context_processors.request',  
10                ,  
11                'django.contrib.auth.context_processors.auth',  
12                ,  
13                'django.contrib.messages.context_processors.  
messages',  
14            ],  
15        },  
16    ],  
17 ]
```

Criando um template básico

Passo a passo:

- Criar um arquivo de template (ex.: index.html) dentro do diretório templates/blog/.
- Adicionar estrutura básica de HTML ao arquivo:

```
1 <!DOCTYPE html>
2 <html>
3 <head>
4     <title>Meu Site</title>
5 </head>
6 <body>
7     <h1>Bem-vindo a minha lista de tarefas!</h1>
8     <p>Conteúdo dinâmico será exibido aqui.</p>
9 </body>
10 </html>
```


Criando um template básico

Ligando o template à View:

- Em views.py, usar a função render para renderizar o template:

```
1 from django.shortcuts import render
2
3 def index(request):
4     return render(request, 'index.html')
```

Passando contexto para templates

- O que é contexto?
 - Dados dinâmicos passados das views para os templates.
- Exemplo de passagem de contexto:
 - Em views.py, definir variáveis e passá-las para o template:

```
1 def index(request):
2     context = {
3         'titulo': 'Página Inicial',
4         'mensagem': 'Seja bem-vindo!',
5     }
6     return render(request, 'index.html', context)
```

No template index.html, usar as variáveis de contexto:

```
1 <h1>{{ titulo }}</h1>
2 <p>{{ mensagem }}</p>
```

Conceito de herança:

- Permite criar uma estrutura base de layout e reutilizá-la em várias páginas, garantindo consistência e facilitando a manutenção.

Como funciona:

- Templates filhos herdam o layout e conteúdo do template pai, podendo adicionar ou sobrescrever partes específicas.

Criando um template base

Exemplo de template base (base.html):

```
1 <!-- estrutura html -->
2 <title>{% block title %}Meu Site{% endblock %}</title>
3 <header>
4     <h1>Minhas Tarefas</h1>
5 </header>
6 <nav>
7     <a href="/">Home</a>
8     <a href="/task/">Tarefa</a>
9 </nav>
10 <main>
11     {% block content %}
12     <!-- Conteúdo específico será inserido aqui -->
13     {% endblock %}
14 </main>
15 <footer>
16     <p>&copy; 2024 Meu Site</p>
17 </footer>
18 <!-- estrutura html -->
```

Usando a herança de templates

Template filho herdando de base.html:

- No template específico (ex.: index.html):

```
1 {% extends 'base.html' %}
2
3 {% block title %}Página Inicial{% endblock %}
4
5 {% block content %}
6 <h2>Bem-vindo ao Blog!</h2>
7 <p>Conteúdo da página inicial do blog.</p>
8 {% endblock %}
```

Nota:

- `{% extends 'base.html' %}` indica que este template usará o layout do template base.
- `{% block title %}` e `{% block content %}` permitem inserir ou modificar conteúdo dentro dos blocos definidos no template base.



Tags de template úteis em Django

Tags comuns:

- {% if %}: Estruturas condicionais.

```
1 {% if usuario.is_authenticated %}  
2     <p>Olá, {{ usuario.username }}!</p>  
3 {% else %}  
4     <p>Bem-vindo, visitante!</p>  
5 {% endif %}
```

- {% for %}: Estruturas de repetição.

```
1 <ul>  
2 {% for post in posts %}  
3     <li>{{ post.titulo }}</li>  
4 {% endfor %}  
5 </ul>
```

- {% include %}: Inclui outro template dentro do atual.

```
1 {% include 'partials/footer.html' %}
```



Revisão:

- Uso de templates para criar e renderizar páginas HTML.
- Passagem de contexto para exibir dados dinâmicos.
- Criação de templates base e uso de herança para reutilização de layouts.

Nota: Manter a organização dos templates em diretórios separados por aplicação para facilitar a manutenção e escalabilidade.

Leitura recomendada:

- Tutorial sobre templates
- Documentação oficial do Django sobre views

Comunidades:

- Stack Overflow, Django Girls Tutorial.

Objetivos:

- Compreender o papel dos modelos em Django e sua relação com o banco de dados.
- Aprender a definir e usar modelos para armazenar e manipular dados.
- Entender o processo de migrações para refletir mudanças no banco de dados.

Nota:

- Modelos são a ponte entre o código e o banco de dados, permitindo que dados sejam manipulados com facilidade.

O que são modelos em Django

Modelos são classes Python que definem a estrutura de dados do aplicativo, incluindo campos e comportamentos. Eles representam e gerenciam os dados armazenados no banco de dados.

Papel dos modelos:

- Definem a estrutura do banco de dados (tabelas).
- Fornecem uma interface Python para manipulação de dados.
- Utilizam o ORM (Object-Relational Mapping) do Django para realizar consultas e operações no banco de dados.

Configurando o banco de dados no Django

Passo 1: Configuração Padrão no settings.py

- Configuração padrão para usar SQLite:

```
1 DATABASES = {  
2     'default': {  
3         'ENGINE': 'django.db.backends.sqlite3',  
4         'NAME': BASE_DIR / 'db.sqlite3',  
5     }  
6 }
```

Configurando o banco de dados no Django

Passo 2: Alteração para outros bancos de dados (opcional!)

- Configurações para usar PostgreSQL, MySQL, etc.

```
1 DATABASES = {  
2     'default': {  
3         'ENGINE': 'django.db.backends.postgresql',  
4         'NAME': 'nome_do_banco',  
5         'USER': 'usuario',  
6         'PASSWORD': 'senha',  
7         'HOST': 'localhost',  
8         'PORT': '5432',  
9     }  
10 }
```

Criando um modelo em Django

Exemplo de modelo:

- No arquivo `models.py` da aplicação `blog`:

```
1 from django.db import models
2
3 class Tarefa(models.Model):
4     titulo = models.CharField(max_length=200)
5     descricao = models.TextField()
6     data_criacao = models.DateTimeField(auto_now_add=True)
7     status = models.CharField(max_length=20, choices=[('Pendente', 'Pendente'), ('Concluída', 'Concluída')])
8
9     def __str__(self):
10         return self.titulo
```

Componentes do modelo:

- **Fields:** Definem os tipos de dados e as propriedades (ex.: `CharField`, `TextField`, `DateTimeField`).
- **Método `__str__`:** Retorna uma string representativa do objeto.



Tipos de campos comuns em modelos

- **Exemplos de campos:**

- CharField: texto curto, com limite de caracteres.
- TextField: texto longo, sem limite de caracteres.
- IntegerField: números inteiros.
- DateField e DateTimeField: datas e horários.
- BooleanField: valores verdadeiros/falsos.
- ForeignKey: relacionamento de muitos para um entre modelos.

Uso de opções em campos:

- **Exemplo:** Definindo um campo obrigatório e único:

```
1 email = models.EmailField(unique=True)
```

Migração: Criando e aplicando alterações no banco de dados

O que são migrações?

- Migrações são formas de sincronizar os modelos Python com o esquema do banco de dados.
- Geram arquivos de migração que especificam as mudanças no banco de dados.

Passos para criar e aplicar migrações

- Criar migrações: `python manage.py makemigrations`
- Aplicar migrações: `python manage.py migrate`

Exemplo de saída:

- Mensagem confirmando que a tabela Post foi criada no banco de dados.

Trabalhando com o Django ORM

Criando objetos:

```
1 from tarefa.models import Tarefa
2
3 nova_tarefa = Tarefa(titulo='Comprar leite', descricao='Ir
   ao mercado e comprar leite', status='Pendente')
4 nova_tarefa.save()
```

Consultando objetos:

```
1 tarefa = Tarefa.objects.all()
2 tarefa = Tarefa.objects.get(id=1)
```

Continua...

Atualizando objetos:

```
1 tarefa = Tarefa.objects.get(id=1)
2 tarefa.titulo = 'Tarefa Atualizada'
3 tarefa.save()
```

Deletando objetos

```
1 tarefa = Tarefa.objects.get(id=1)
2 tarefa.delete()
```

Relacionamentos entre modelos

Tipos de relacionamentos:

- One-to-Many (um para muitos): Usado com ForeignKey.

```
1 class Categoria(models.Model):  
2     nome = models.CharField(max_length=100)  
3  
4 class Tarefa(models.Model):  
5     categoria = models.ForeignKey(Categoria, on_delete=  
        models.CASCADE)
```

- Many-to-Many (muitos para muitos): Usado com ManyToManyField.

```
1 class Prioridade(models.Model):  
2     nome = models.CharField(max_length=50)  
3  
4 class Tarefa(models.Model):  
5     prioridades = models.ManyToManyField(Prioridade)
```

- Exemplos de uso: Relacionar um autor a vários posts ou associar múltiplas tags a um post.

Registro de modelos no Django Admin

O que é o Django Admin?

- Interface de administração automática para gerenciar dados de forma interativa.

Registrando um modelo no Admin:

- No arquivo `admin.py` da aplicação:

```
1 from django.contrib import admin
2 from .models import Tarefa
3
4 admin.site.register(Tarefa)
```

Personalizando a exibição:

- Usando classes para customizar como as tarefas são exibidas no Admin

```
1 class TarefaAdmin(admin.ModelAdmin):
2     list_display = ('titulo', 'data_criacao', 'status')
3     search_fields = ('titulo', 'descricao')
4
5 admin.site.register(Tarefa, TarefaAdmin)
```

Revisão

- Definição e criação de modelos em Django para representar e gerenciar dados.
- Uso de migrações para sincronizar o esquema do banco de dados.
- Uso do Django ORM para operações CRUD (Criar, Ler, Atualizar, Deletar).
- Implementação de relacionamentos entre diferentes modelos.

Nota: Testar consultas e alterações no shell do Django (`python manage.py shell`) antes de aplicá-las no código.

Leitura recomendada:

- Tutorial sobre templates
- Documentação oficial do Django sobre views

Comunidades:

- Stack Overflow, Django Girls Tutorial.

Objetivos:

- Entender o papel do Admin do Django para o gerenciamento de dados.
- Aprender a registrar e personalizar modelos no Django Admin.
- Explorar funcionalidades avançadas do Django Admin para melhorar a eficiência no gerenciamento de aplicações.

Nota:

- O Django Admin oferece uma interface poderosa para administrar sua aplicação com facilidade e segurança.

O que é o Django Admin?

O Django Admin é uma interface de administração automática gerada para gerenciar o conteúdo dos modelos do Django.

Características principais:

- Fornece uma interface gráfica para gerenciar dados e usuários.
- Totalmente personalizável para atender às necessidades específicas do projeto.
- Inclui recursos de autenticação e autorização.

Benefícios:

- Rápida configuração e fácil manutenção.
- Reduz o tempo de desenvolvimento para construir interfaces de administração.
- Aumenta a segurança ao gerenciar acesso a dados sensíveis.

Ativando o Django Admin

Configuração básica:

- O Django Admin já está incluído por padrão nas instalações do Django.

Passo 1: Adicionar 'django.contrib.admin' em INSTALLED_APPS:

```
1 INSTALLED_APPS = [  
2     'django.contrib.admin',  
3     'django.contrib.auth',  
4     'django.contrib.contenttypes',  
5     'django.contrib.sessions',  
6     'django.contrib.messages',  
7     'django.contrib.staticfiles',  
8     'tarefa', # Aplicação do exemplo  
9 ]
```

Continua...

Passo 2: Incluir URLs do Admin em urls.py:

```
1 from django.contrib import admin
2 from django.urls import path, include
3
4 urlpatterns = [
5     path('admin/', admin.site.urls),
6     path('tarefa/', include('tarefa.urls')),
7 ]
```

Criando um superusuário

Por que criar um superusuário?

- Um superusuário tem acesso a todos os recursos do Django Admin e pode gerenciar usuários e permissões.

Comando para criar um superusuário:

- Executar no terminal:

```
1 python manage.py createsuperuser
```

Atenção!: Nome de usuário, e-mail e senha.

Acessando o Django Admin

Iniciando o servidor de desenvolvimento:

- Executar este comando no terminal:

```
1 python manage.py runserver
```

Acessar o Django Admin

- Navegar para <http://127.0.0.1:8000/admin/>.
- Fazer login com as credenciais do superusuário criadas.

Tela inicial do Admin:

- Interface gráfica para gerenciar usuários, grupos, e dados do projet

Registrando modelos no Admin

Por que registrar modelos?

- Para permitir a manipulação de dados dos modelos através da interface do Django Admin.

Passos para registrar um modelo:

- No arquivo `admin.py` da aplicação (`blog/admin.py`):

```
1 from django.contrib import admin
2 from .models import Post
3
4 admin.site.register(Post)
```

Visualizando o modelo registrado:

- Após registrar, o modelo `Post` aparecerá na interface do Django Admin, permitindo criar, editar e deletar registros.

Personalizando a interface do Django Admin

Personalização básica:

- Usar classes de administração para customizar a exibição dos modelos.

Exemplo de personalização de modelo:

- Em `blog/admin.py`:

```
1 from django.contrib import admin
2 from .models import Post
3
4 class PostAdmin(admin.ModelAdmin):
5     list_display = ('titulo', 'autor', 'data_publicacao')
6     search_fields = ('titulo', 'conteudo')
7     list_filter = ('data_publicacao', 'autor')
8
9 admin.site.register(Post, PostAdmin)
```

Componentes personalizáveis:

- `list_display`: Define quais campos são exibidos na lista de objetos.
- `search_fields`: Campos em que a pesquisa é habilitada.
- `list_filter`: Filtros para refinar a visualização dos objetos.



Recursos avançados do Django Admin

Ações customizadas:

- Criar ações personalizadas para realizar operações em lote.
- Exemplo de ação para publicar múltiplos posts:

```
1 def publicar_posts(modeladmin, request, queryset):
2     queryset.update(publicado=True)
3 publicar_posts.short_description = "Publicar posts
4     selecionados"
5
6 class PostAdmin(admin.ModelAdmin):
7     actions = [publicar_posts]
```

Inline models:

- Exibir e editar modelos relacionados diretamente na página do modelo pai.
- Exemplo:

```
1 from .models import Comentario
2
3 class ComentarioInline(admin.TabularInline):
4     model = Comentario
```

Configurações de permissões e segurança

Controle de acesso:

- O Django Admin permite configurar permissões específicas para diferentes usuários e grupos.

Definir permissões em modelos:

- No modelo:

```
1 class Post(models.Model):  
2     ...  
3     class Meta:  
4         permissions = [  
5             ("pode_publicar_post", "Pode publicar post"),  
6         ]
```

Gerenciamento de usuários e grupos:

- Criar e atribuir permissões para controlar o que cada usuário pode ver e modificar no Django Admin.

Revisão:

- Configuração e acesso ao Django Admin.
- Criação de superusuários para administração completa.
- Registro e personalização de modelos para gerenciamento de dados.
- Uso de recursos avançados para melhorar a administração de aplicações.

Nota: Usar o Django Admin como uma ferramenta de desenvolvimento rápida, mas considerar construir interfaces customizadas para aplicações com requisitos específicos de UI.

Leitura recomendada:

- Tutorial sobre templates
- Documentação oficial do Django sobre views

Comunidades:

- Stack Overflow, Django Girls Tutorial.

Objetivos:

- Entender o papel dos formulários em Django.
- Aprender a criar e utilizar formulários para entrada de dados.
- Explorar a validação de dados e manipulação de formulários com Django Forms e ModelForms.

Nota: Com formulários, o Django facilita a captura e a validação de dados do usuário de maneira segura e eficiente.

O que são formulários em Django?

Formulários são componentes para coletar e processar entrada de dados dos usuários.

Papel dos formulários em aplicações web:

- Captura de dados de forma estruturada.
- Validação de dados antes de processá-los.
- Segurança ao evitar entrada de dados maliciosos.

Tipos de formulários no Django:

- Django Forms: Formulários personalizados usando a classe Form.
- ModelForms: Formulários baseados em modelos que integram diretamente com o banco de dados.

Criando um formulário personalizado

Definição de formulário com a Classe Form:

- Exemplo de um formulário simples em `forms.py` dentro da aplicação blog:

```
1 from django import forms
2
3 class ContatoForm(forms.Form):
4     nome = forms.CharField(max_length=100)
5     email = forms.EmailField()
6     mensagem = forms.CharField(widget=forms.Textarea)
```

Explicação dos componentes:

- `CharField`: Campo de texto para entrada de dados.
- `EmailField`: Campo de entrada para e-mail, com validação embutida.
- `widget`: Personalização do campo, como uso de área de texto (`Textarea`).

Renderizando o formulário em uma View

Passos para renderizar o formulário:

- Criar uma view para exibir o formulário:

```
1 from django.shortcuts import render
2 from .forms import ContatoForm
3
4 def contato_view(request):
5     form = ContatoForm()
6     return render(request, 'blog/contato.html', {'form':
7     form})
```

Criar um template para exibir o formulário:

```
1 <!-- estrutura html -->
2 <h1>Formulário de Contato</h1>
3 <form method="post">
4     {% csrf_token %}
5     {{ form.as_p }}
6     <button type="submit">Enviar</button>
7 </form>
8 <!-- estrutura html -->
```

Manipulação de dados do formulário

Processando dados do formulário na View:

- Verificar se o método de solicitação é POST e validar o formulário:

```
1 def contato_view(request):
2     if request.method == 'POST':
3         form = ContatoForm(request.POST)
4         if form.is_valid():
5             # Processar os dados
6             nome = form.cleaned_data['nome']
7             email = form.cleaned_data['email']
8             mensagem = form.cleaned_data['mensagem']
9             # Aqui você pode salvar os dados, enviar e-mail,
10            etc.
11        else:
12            form = ContatoForm()
13    return render(request, 'blog/contato.html', {'form': form})
```

- `is_valid()`: Verifica se os dados do formulário atendem aos critérios de validação. `cleaned_data`: Retorna os dados do formulário já validados e limpos.



Introdução ao ModelForm

O que é um ModelForm?

- Um ModelForm é uma classe que cria um formulário baseado em um modelo do Django, facilitando a criação de formulários para inserção e atualização de dados no banco de dados.

Exemplo de ModelForm para o modelo post:

- Em forms.py:

```
1 from django import forms
2 from .models import Post
3
4 class PostForm(forms.ModelForm):
5     class Meta:
6         model = Post
7         fields = ['titulo', 'conteudo', 'autor']
```

Vantagens do uso de ModelForm:

- Redução de código repetitivo.
- Integração direta com a validação do modelo.
- Manutenção simplificada e melhor organização.

Usando ModelForm em uma View

Passos para usar ModelForm:

- Criar uma view para gerenciar o formulário de criação de um post:

```
1 from django.shortcuts import render, redirect
2 from .forms import PostForm
3
4 def criar_post_view(request):
5     if request.method == 'POST':
6         form = PostForm(request.POST)
7         if form.is_valid():
8             form.save()
9             return redirect('index')
10    else:
11        form = PostForm()
12    return render(request, 'blog/criar_post.html', {'form': form})
```

Continua...



- Criar um template para renderizar o formulário:

```
1 <!DOCTYPE html>
2 <html>
3 <head>
4     <title>Criar Post</title>
5 </head>
6 <body>
7     <h1>Criar um Novo Post</h1>
8     <form method="post">
9         {% csrf_token %}
10        {{ form.as_p }}
11        <button type="submit">Salvar</button>
12    </form>
13 </body>
14 </html>
```

Validação de dados em formulários

Validação automática em formulários Django:

- Campos como EmailField e CharField vêm com validação embutida.

Validação personalizada:

- Adicionar métodos `clean_<campo>()` para validação personalizada:

```
1 class ContatoForm(forms.Form):
2     email = forms.EmailField()
3
4     def clean_email(self):
5         email = self.cleaned_data.get('email')
6         if not email.endswith('@dominio.com'):
7             raise forms.ValidationError("O e-mail deve
8             terminar com '@dominio.com'")
9         return email
```

Validação em ModelForm:

- Validar campos de um ModelForm similarmente, sobrescrevendo o método `clean()`.

Estilizando formulários com widgets

Uso de widgets para personalizar formulários:

- Widgets permitem personalizar a renderização dos campos.
- Exemplo:

```
1 class ContatoForm(forms.Form):  
2     nome = forms.CharField(widget=forms.TextInput(attrs={  
3         'class': 'form-control'}))  
4     mensagem = forms.CharField(widget=forms.Textarea(attrs={  
5         'rows': 5, 'class': 'form-control'}))
```

Tipos comuns de widgets:

- TextInput: Campo de texto simples.
- Textarea: Área de texto para entradas longas.
- Select: Campo de seleção suspensa.

Aplicação prática:

- Adicionar classes CSS e outros atributos HTML para melhorar a aparência e a usabilidade dos formulários.



Revisão:

- Criação e manipulação de formulários em Django usando classes Form e ModelForm.
- Validação de entrada de dados do usuário para garantir a integridade.
- Personalização de formulários com widgets e atributos de estilo.
- Integração de formulários com o banco de dados usando ModelForm.

Nota: Usar sempre a validação embutida e personalizada para garantir que os dados coletados sejam seguros e válidos.