



Decoradores

Decoradores

▼ Funções decoradoras

- Uma **função decoradora** (ou simplesmente *decorator*) é uma função que **recebe outra função como argumento**, adiciona algum comportamento a ela e retorna uma nova função com esse comportamento **modificado**, sem alterar o código original da função decorada.
- Os decoradores são muito usados para modificar o comportamento de funções, como adicionar logs, verificar permissões, medir tempo de execução, entre outros. Podem ser usadas para adicionar, remover, restringir, alterar, etc.
- Elas seguem o conceito de **Higher Order Functions**.

▼ Decoradores (**@funcao_decoradora**)

- **Decoradores** são a aplicação prática das funções decoradoras usando a **sintaxe especial com @**. Ou seja, um decorador é o uso de uma função decoradora com **@nome_do_decorador** antes da definição de outra função.
- Os decoradores utilizam um **açúcar sintático** (*syntax sugar*), que é uma forma mais elegante e legível de escrever um código que poderia ser implementado de maneira mais verbosa.
- Exemplo de decorador que exibe mensagem antes e depois de executar uma função:

```

# Criando uma função decoradora que modifica o comportamento da
def meu_decorador(func): # Isso é uma função decoradora
    def interna(): # Função que envolve a original
        print("Antes de executar a função")
        func() # Chama a função original
        print("Depois de executar a função")
    return interna # Retorna a função modificada

# Aplicando o decorador com @
@meu_decorador # Isso é um decorador (aplicação da função decoradora)
def dizer_ola():
    print("Olá, mundo!")

# Chamando a função decorada
dizer_ola()

```

- Cria a **função decoradora** `meu_decorador (func)`, que:
 - Envolve (`interna ()`) a função original (`func ()`).
 - **Adiciona** mensagens antes e depois da execução da função.
 - **Retorna** `interna ()`, que **será chamada no lugar da original**.
- Aplicamos `@meu_decorador` à função `dizer_ola ()`, **modificando seu comportamento**.
- Ao chamar `dizer_ola ()`, **em vez de rodar diretamente, ele passa por** `interna ()`, imprimindo mensagens antes e depois.

▼ Exemplo detalhado de código com decoradores

- Exemplo de uso de decorador:

```

# Criando um decorador para validar se todos os argumentos são números
def validar_numeros(func):
    def interna(*args, **kwargs):
        print("Validando os argumentos...")

        # Verifica se todos os argumentos são números (int ou float)
        for arg in args:
            if not isinstance(arg, (int, float)):

```

```

        raise TypeError(f"O argumento {arg} não é um número!")

    # Chama a função original e armazena o resultado
    resultado = func(*args, **kwargs)

    print(f"Resultado calculado: {resultado}")
    print("Validação concluída.")

    return resultado

return interna # Retorna a função modificada

# Aplicando o decorador usando @
@validar_numeros
def soma(a, b):
    print(f"Executando {soma.__name__}...") # Exibe interna, pois a função
    return a + b

# Testando a função decorada
resultado = soma(10, 5)
print(resultado)

# Testando com argumento inválido
# resultado = soma(10, "abc") # Isso vai gerar um erro de TypeError

```

1. Criamos o decorador `validar_numeros`

- Ele **recebe uma função** (`func`) como argumento.
- Dentro dele, a função `interna` verifica se os argumentos são números.
- Se tudo estiver correto, **chama a função original** e imprime o resultado.

2. Aplicamos o decorador com `@validar_numeros`

- **O Python passa a executar** `validar_numeros (soma)`, ou seja, `soma` agora **está decorada**.
- Sempre que `soma` for chamada, ela **passará primeiro pela verificação de tipos** antes de executar a soma.

3. Testamos a função decorada

- `soma (10, 5)` funciona normalmente.
- `soma (10 , "abc")` gera um erro porque `"abc"` não é um número.



Esse exemplo **demonstra como decoradores ajudam a adicionar funcionalidades** (como validação) sem modificar diretamente a função original.

▼ Decoradores com parâmetros



- Os parâmetros em um decorador permitem personalizar seu comportamento de acordo com a necessidade. Em vez de criar vários decoradores diferentes para modificar funções de maneiras específicas, podemos usar parâmetros para definir como a modificação será aplicada. Por exemplo, um decorador pode aceitar um número como argumento para repetir a execução de uma função múltiplas vezes ou receber um texto para exibir uma mensagem antes de executar a função original. Assim, é possível adaptar o mesmo decorador para diferentes situações sem precisar reescrevê-lo.

- Decoradores podem **receber parâmetros** para tornar seu comportamento mais flexível. Para isso, são necessárias três camadas de funções:
 - **Uma função externa** que **recebe o parâmetro** do decorador.
 - **Uma função interna** que **recebe a função** a ser decorada.
 - **Uma função mais interna** que **modifica o comportamento** da função original.
- Estrutura básica:

```
def decorador_com_parametros(parametro): # Camada 1
    def decorador(func): # Camada 2
        def interna(*args, **kwargs): # Camada 3
```

```
        print(f"Executando com parâmetro: {parametro}")
        return func(*argss, **kwargs)
    return interna
return decorador
```

- Exemplo de decorador que repete a execução da função:

```
# Criando o decorador com parâmetro
def repetir(n):
    def decorador(func):
        def interna(*args, **kwargs):
            for _ in range(n): # Executa a função 'n' vezes
                func(*args, **kwargs)
            return interna
        return decorador # Retorna o decorador

# Aplicando o decorador com parâmetro
@repetir(3)
def saudacao():
    print("Olá, mundo!")

# Chamando a função decorada
saudacao()

# Exibe:
Olá, mundo!
Olá, mundo!
Olá, mundo!
```

- **Explicação passo a passo**

1. `repetir (n)` (Camada 1)
 - Define um decorador que recebe `n` como **argumento**.
2. `decorador (func)` (Camada 2)
 - **Recebe a função original** e **retorna a função modificada**.
3. `interna (* args , ** kwargs)` (Camada 3)
 - Executa a função original `n` vezes.

4. Quando chamamos `saudacao ()`, o decorador faz com que a mensagem seja exibida 3 vezes.

▼ Ordem de aplicação dos decoradores

- Quando várias funções decoradoras são aplicadas a uma única função, elas são executadas de cima para baixo, **mas a aplicação do decorador ocorre de baixo para cima.**
 - Isso significa que:
 1. O **decorador mais próximo** da função é aplicado **primeiro**.
 2. O **decorador mais distante** da função é aplicado por **último**.
 3. Durante a execução da função, a ordem é do primeiro decorador aplicado até o último.
- Exemplo:

```
def parametros_decorador(nome):
    def decorador(func):
        print("Decorador:", nome)

        def sua_nova_funcao(*args, **kwargs):
            res = func(*args, **kwargs)
            final = f"{res} {nome}"
            return final
        return sua_nova_funcao
    return decorador

@parametros_decorador(nome="5")
@parametros_decorador(nome="4")
@parametros_decorador(nome="3")
@parametros_decorador(nome="2")
@parametros_decorador(nome="1")
def soma(a, b):
    return a + b

dez_mais_cinco = soma(10, 5)
print(dez_mais_cinco)
```