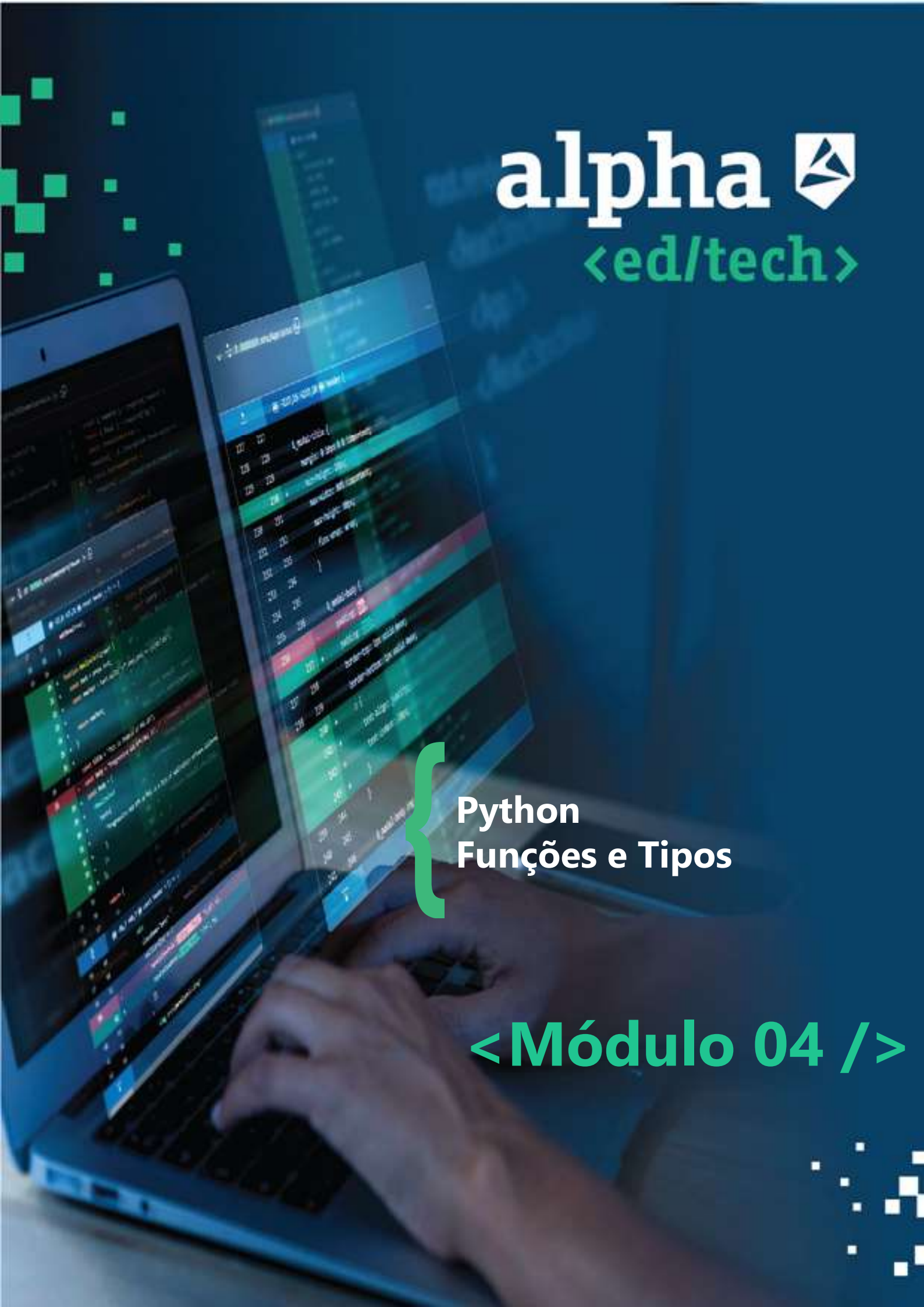
alpha
<ed/tech>
Python
Funções e Tipos
<Módulo 04 />

Funções e Tipos

Introdução

Bem-vindos ao capítulo sobre "Funções e Tipos" em Python. Este capítulo é fundamental para aprofundar seus conhecimentos, pois abordaremos dois aspectos essenciais: as **funções**, que são blocos de código reutilizáveis e organizados, e os **tipos de dados**, que formam a base de como a informação é armazenada e manipulada em Python. Além disso, abordaremos os *Type Hints* e a ferramenta *MyPy* para garantir que o código seja não apenas funcional, mas também tipado de maneira correta.

Funções Básicas

Uma função é um bloco de código que só é executado quando chamado. Funções são definidas usando a palavra-chave **'def'** seguida pelo **nome da função** e **parênteses**, que podem conter **parâmetros**.

```
def minha_funcao():
    print("Olá, mundo!")
```

Para chamar esta função, simplesmente use seu nome seguido de parênteses

```
minha_funcao()   # Saída: Olá mundo!
```

Parâmetros e Argumentos

Uma função é um bloco de código que só é executado quando chamado. Funções são definidas usando a palavra-chave **def** seguida pelo **nome da função** e **parênteses**, que podem conter **parâmetros**.

```
def saudacao(nome):
    print(f"Olá, {nome}!")

saudacao("José")  # Saída: Olá, José!
```

Se você quiser que um parâmetro tenha um valor padrão, basta passar na declaração da função o valor para este parâmetro. Caso a função não receba um valor para este parâmetro será utilizado o valor padrão informado.

```
def saudacao(nome="José"):
    print(f"Olá, {nome}!")

saudacao()  # Saída: Olá, José!
```

Retorno de Valores

Uma função pode retornar um valor como resultado de sua execução, usando a palavra-chave **return**.

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

resultado = soma(5, 3)
print(resultado)  # Saída: 8
```

Você pode ainda retornar vários valores em uma só função, separando os mesmos por vírgula. Cada valor será atribuído à uma variável diferente no retorno.

```
def inversor(a, b):
    return b, a

valora, valorb = repetidor(5, 3)
print(f"{valora}, {valorb}")  # Saída: 3, 5
```

Variáveis Locais e Globais

Variáveis definidas dentro de uma função são locais àquela função, enquanto variáveis fora são globais.

```
def minha_funcao():
    variavel_local = "local"
    print(variavel_local)

minha_funcao()  # Saída: local
```

```
def minha_funcao(variável):
    print(variavel)

variavel_global = "global"
minha_funcao(variavel_global)  # Saída: global
```

Atente-se para os nomes das variáveis e parâmetros das funções, para não correr o risco de que um valor global seja alterado para um valor local por utilizar o mesmo nome de variáveis

Funções Básicas

Exercício Resolvido 1

O aluno após se cadastrar deve receber uma mensagem de boas-vindas, então você deveria criar uma função que retorne a mensagem específica.

O uso desta função pode permitir que a solução de cadastro não repita a impressão de mensagem em vários lugares no código, melhorando a manutenção do código futuramente. Pense no caso onde você precisaria atualizar a mensagem de boas-vindas, com um função você altera em somente um local, e não em vários caso a impressão estivesse espalhada pelo código.

```
def boasvindas():
    return “Bem-vindo(a) ao curso de Python!”

boasvindas()  # Saída: “Bem-vindo(a) ao curso de Python!”
```

Exercício Resolvido 2

Neste caso você deve considerar que o parâmetro idade deve ter um valor padrão 18, para que caso ela não seja informada assuma-se que a idade do aluno(a) será a mínima necessária para o curso. Aqui não estamos entrando no mérito de validação cadastral, e sim só estamos realizando o cadastro, então neste caso podemos assumir que quem irá se cadastrar terá ao menos a idade mínima aceita pelo curso.

```
def cadastro(nome, idade=18):
    return “Bem-vindo(a) ao curso de Python {nome} ({idade} anos)!”

cadastro(“José da Silva”,18)
# Saída: “Bem-vindo(a) ao curso de Python José da Silva (18 anos)!”

cadastro(“Ana Moreira”)
# Saída: “Bem-vindo(a) ao curso de Python Ana Moreira (18 anos)!”
```

Exercício Resolvido 3

Neste caso devemos utilizar o retorno múltiplo de resultados em uma função, retornando as informações de matrícula e turma.

```
def cadastro(nome, idade=18, endereço, cep):
    matricula = 12345
    turma = “PY-2024-001”
    return matricula, turma

nome = “José da Silva”
m, t = cadastro(nome, 18, ”Avenida Brasil, 100”, “12345-678”)
print(f“Olá {nome}, sua matrícula é {m} na turma {t}”)

# Saída: “Olá José da Silva, sua matrícula é 12345 na turma PY-2024-001”
```

Funções Puras e Impuras

Conceitos de Pureza

- **Funções Puras:** são **determinísticas**, significa que para um mesmo conjunto de dados de entrada a função sempre retornará o mesmo resultado. Deve ser uma função **sem efeitos colaterais**, significa que nenhum estado fora da função será modificado (não modifica variáveis globais, não escreve em arquivos, não altera banco de dados etc.).

- **Funções Impuras:** são funções que não seguem uma ou ambas propriedades citadas acima. Elas podem produzir efeitos colaterais e/ou ter resultados diferentes para o mesmo conjunto de dados de entrada.

Benefícios e Limitações

- **Funções Puras**

  Testabilidade: são mais fáceis de testar devido à previsibilidade

  Reusabilidade: podem ser reutilizadas em diferentes contextos

  Limitações: nem sempre é prático ou possível evitar efeitos colaterais, especialmente em operações de E/S (entrada / saída)

- **Funções Impuras**

  Praticidade: em muitos casos, é necessário interagir com o mundo externo (E/S, operações de rede etc.).

  Complexidade em Testes: requerem mais esforço para testar, devido aos efeitos colaterais e dependências externas.

Vamos ver agora alguns exemplos e casos de uso das funções puras e impuras.

Função Pura

```
def multiplicar(a, b):
    return a * b

Print(multiplicar(2, 3))  # Saída: 6
```

Função que sempre retorna o mesmo resultado para os mesmos argumentos de entrada recebidos (sempre retornará 6 quando receber 2 e 3 respectivamente como entrada) e não possui efeitos colaterais.

Função Impura – Altera Estado Global

```
contador = 0

def incrementar_contador():
    global contador
    contador += 1

incrementar_contador()
print(contador)  # Saída: 1
```

A função **incrementar_contador** é impura porque altera o valor da variável global **contador**.

Função Impura – Efeitos Colaterais de E/S

```
def salvar_arquivo(texto, nome_arquivo):
    with open(nome_arquivo, 'w') as arquivo:
        arquivo.write(texto)
    return f"Arquivo {nome_arquivo} salvo com sucesso."

print(salvar_arquivo("Olá, mundo!", "exemplo.txt"))
# Saída: Arquivo, exemplo.txt salvo com sucesso.
```

Esta função é impura, pois realiza uma operação de escrita em arquivo, que é um efeito colateral

Função Impura – Não Determinística

```
import Random

def lancar_dado():
    return Random.randint(1, 6)

print(lancar_dado())  # Pode retornar qualquer número entre 1 e 6
```

A função **lancar_dado** é impura pois seu retorno é não determinístico, ou seja, dependendo do estado interno do gerador é retornado um número aleatório.

Funções Puras e Impuras

Exercício Resolvido 1

Para resolver esta questão vamos trabalhar com uma operação de soma, e criar duas funções:

FUNÇÃO PURA

```
def soma_pura(a, b):
    return a + b

resultado = soma_pura(5, 3)
print(“Resultado da função pura: “, resultado)
# Saída: “Resultado da função pura: 8”
```

Não importa quantas vezes a função **soma_pura** for chamada, passando os mesmos valores **5** e **3** respectivamente como entrada, ela sempre retornará o valor **8**. Além disso ela não modifica nenhum estado externo ou variável global.

FUNÇÃO IMPURA

```
resultado = 0

def soma_impura(a, b):
    global resultado
    resultado = a + b

soma_impura(5, 3)
print(“Resultado da função pura: “, resultado)
# Saída: “Resultado da função pura: 8”
```

Já a função **soma_impura** é impura pois altera o estado da variável global resultado, gerando um efeito colateral pois a função está fazendo algo além de retornar um valor, está modificando um estado externo a seu escopo.

Exercício Resolvido 2

A função original incrementa um contador global, o que é um efeito colateral que precisará ser removido da função para que ela seja transformada em uma função pura. Vamos fazer com que a função receba o valor atual do contador como argumento e retorne o valor incrementado, sem alterar nenhuma variável externa.

```
def incrementar_contador_puro(contador_atual):
    return contador_atual + 1

# Exemplo de uso
contador_atual = 0
novo_contador = incrementar_contador_puro(contador_atual)
print("Contador atual:", contador_atual)  # Saída: 0
print("Novo contador:", novo_contador)    # Saída: 1
```

Exercício Resolvido 3

A função **modificar_contador_global** é impura porque ela tem um efeito colateral. Antes de chamar a função a variável global possui o valor **10**. Após a chamada da função ela recebe então o valor **20** demonstrando a impureza da função devido a mudança da variável global.

```
contador = 10

def modificar_contador_global(valor):
    global contador
    contador = valor

print("Valor original do contador: ", contador)
# Saída: "Valor original do contador: 10"

modificar_contador_global(20)

print("Valor do contador após função: ", contador)
# Saída: "Valor do contador após função: 20"
```

Funções com Callbacks

Callback

Um callback é uma função que é passada como argumento para outra função e é invocada dentro dessa função. Em Python, como as funções são elementos de primeira classe, elas podem ser passadas como parâmetros, retornadas por outras funções e atribuídas a variáveis.

Uso de Callbacks

Callbacks são úteis em muitos cenários, como em operações assíncronas, manipulação de eventos, ou quando você quer permitir que o usuário de sua função forneça algum código personalizado que deve ser executado em um determinado ponto da execução da função.

```
def processar_dados(dados, callback):
    # Processamento fictício dos dados
    resultado = dados + " processados"
    callback(resultado)

def meu_call-back(resultado):
    print("Resultado recebido no callback:", resultado)

# Uso da função com callback
processar_dados("Dados de exemplo", meu_callback)
```

Neste exemplo, a função **processar_dados** recebe uma função de callback, **meu_callback** que é chamada após o processamento dos dados.

Pode ainda utilizar callbacks em operações diversas a partir de uma função única, como demonstrado no exemplo a seguir:

```
def executar_operacao(valor, operacao):
    return operacao(valor)

# Definindo diferentes callbacks
def dobrar(x):
    return x * 2

def quadrado(x):
    return x ** 2

# Uso da função com callback
print(executar_operação(5, dobrar))  # Saída: 10
print(executar_operação(5, dobrar))  # Saída: 25
```

Ao utilizar callbacks você ganha maior flexibilidade e adaptatividade no desenvolvimento, permitindo a modificação do comportamento de uma função de forma dinâmica.

Funções com Callbacks

Exercício Resolvido 1

A função **executar_operação** é genérica e pode aplicar qualquer operação (callback) com o número fornecido. O primeiro callback **quadrado** recebe um número e retorna seu quadrado. O segundo callback **inverso** também recebe um número e retorna seu inverso, com a salvaguarda de verificar se o número não é zero para evitar a divisão por zero. As chamadas da função **executar_operação** demonstram como diferentes operações podem ser aplicadas ao mesmo número, utilizando diferentes callbacks.

```
def executar_operacao(numero, operacao):
    return operacao(numero)

# Callback para calcular o quadrado
def quadrado(num):
    return num ** 2

# Callback para calcular o inverso
def inverso(num):
    if num != 0:
        return 1 / num
    else:
        return "Divisão por zero não permitida"

# Demonstração
print("Quadrado de 4:", executar_operacao(4, quadrado))    # Saída: 16
print("Inverso de 4:", executar_operacao(4, inverso))      # Saída: 0.25
print("Inverso de 0:", executar_operacao(0, inverso))
# Saída: Divisão por zero não permitida
```

Essa solução ilustra a flexibilidade das funções em Python e como os callbacks podem ser usados para criar código mais genérico e reutilizável.

Funções com Callbacks

Exercício Resolvido 2

Para resolver esta questão, vamos criar uma função **avaliar_condição** que executa um de dos callbacks baseado em uma condição booleana. A função **avaliar_condição** verifica o valor booleano do argumento valor. Baseado nesta verificação, ela decide qual callback executar, **verdadeiro_callback** e **falso_callback** são definidos para lidar com os casos verdadeiro e falso respectivamente.

```
def avaliar_condicao(valor, callback_true, callback_false):
    if valor:
        callback_true()
    else:
        callback_false()

# Callback para quando o valor é verdadeiro
def verdadeiro_callback():
    print("O valor é verdadeiro!")

# Callback para quando o valor é falso
def falso_callback():
    print("O valor é falso!")

# Demonstração
avaliar_condicao(True, verdadeiro_callback, falso_callback)
# Saída: O valor é verdadeiro!

avaliar_condicao(False, verdadeiro_callback, falso_callback)
# Saída: O valor é falso!
```

Essa solução aborda o conceito de callbacks condicionais, mostrando como eles podem ser usados para controlar o fluxo de execução em um programa, tornando-o mais modular e adaptável a diferentes cenários.

Funções com Callbacks

Exercício Resolvido 3

Para resolver esta questão, vamos criar uma função **processar_numeros** que aceita dois números e um callback. A função **processar_números** é genérica e capaz de executar qualquer operação fornecida como um callback nos números **num1** e **num2**, e as operações **somar** e **multiplicar** são funções de callback que realizam operações matemáticas básicas nos argumentos fornecidos. As chamadas de **processar_números** com diferentes callbacks demonstram a versatilidade da função em aplicar diferentes operações matemáticas.

```
def processar_numeros(num1, num2, operacao):
    return operacao(num1, num2)

# Callback para somar os números
def somar(a, b):
    return a + b

# Callback para multiplicar os números
def multiplicar(a, b):
    return a * b

# Demonstração
print("Soma de 5 e 3:", processar_numeros(5, 3, somar))
# Saída: 8

print("Multiplicação de 5 e 3:", processar_numeros(5, 3, multiplicar))
# Saída: 15
```

Essa solução ilustra a eficácia de usar callbacks para realizar operações variadas com a mesma função base, reforçando a flexibilidade e reutilização de código em programação.