



# PROGRAMAÇÃO FUNCIONAL

### **EQUIPE:**

Carla Vieira de Araujo Prudencio - 2313681
Cleyson de Oliveira Severiano - 2319296
Eliak Lima - 2416956
Heric Ferreira Maciel - 2223814
Raul Carvalho Teles - 2314373











### **REPOSITÓRIO GITHUB:**

https://github.com/carlaprudencio/programacao-funcional

### Responsabilidades dos Integrantes

Carla Prudencio (Código Fonte)

Cleyson de Oliveira (Código Fonte)

Eliak Lima (Casos de Testes)

Heric Ferreira (Documentação dos Requisitos)

Raul Carvalho Teles (Documentação dos Requisitos)

# DOCUMENTO DE REQUISITOS

Filtro de Números Ímpares e Elevação ao Cubo

### **Requisitos Funcionais:**

Requisito 01: O sistema deverá filtrar números ímpares.

### Implementação na linha:

numeros\_impares = [x for x in numeros if eh\_impar(x)]

### Identificação no código:

 O filtro de números ímpares é feito usando list comprehension combinada com a função lambda eh\_impar. A função lambda é responsável por definir a lógica para verificar se um número é ímpar.

Requisito 02: O sistema deverá fazer a potenciação ao cubo dos números ímpares.

### Implementação na linha:

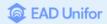
eleva\_cubo = potenciacao(3)

numeros\_elevados = aplica\_func\_numeros(eleva\_cubo, numeros\_impares)









### Identificação no código:

O sistema usa uma closure (função potenciacao) para gerar uma função que eleva os números ao cubo. Após isso, uma função de alta ordem aplica\_func\_numeros aplica a transformação a cada número ímpar.

### Requisito 03: O sistema deverá aplicar uma função a uma lista de números

### Implementação na linha:

numeros\_elevados = aplica\_func\_numeros(eleva\_cubo, numeros\_impares)

### Identificação no código:

- A função aplica\_func\_numeros é uma função de alta ordem que recebe outra função como argumento ( eleva\_cubo) e aplica a cada número da lista.

## Requisito 04: O sistema deverá testar se os números filtrados e potenciados estão corretos.

### <u>Implementado nas funções de testes:</u>

```
def test numeros impares():
```

assert numeros\_impares == [1, 3, 5, 7, 9], f"Expected [1, 3, 5, 7, 9], got {numeros\_impares}"

### def test\_numeros\_elevados\_cubo():

assert numeros\_elevados == [1, 27, 125, 343, 729], f"Expected [1, 27, 125, 343, 729], got {numeros\_elevados}"

### Identificação no código:

- Os testes automatizados garantem que os requisitos estão sendo cumpridos de forma correta.









### Requisitos Não Funcionais:

Requisito 01: O código deverá ser legível e organizado.

### Identificação no código:

O código segue boas práticas de nomenclatura e modularização:

- Funções são bem nomeadas (aplica\_func\_numeros, potenciacao, test\_numeros\_impares, etc.).
- As variáveis descritivas (numeros\_impares, numeros\_elevados).
- O código é modular, com funções específicas para cada tarefa.

### Requisito 02: O código deverá ser eficiente no processamento de listas.

### Identificação no código:

- A list comprehension é usada para filtrar os números de maneira eficiente.
- O uso de lambda e funções de alta ordem garante que o código é funcional e modular, permitindo a reutilização e a redução de código duplicado.

# Requisito 03: O código deverá ser testável com casos de teste automatizados Identificação no código:

- As funções de teste (test\_numeros\_impares, test\_numeros\_elevados\_cubo) permitem que o código seja verificado de forma automatizada, garantindo que os resultados estão corretos para a filtragem de números ímpares e a elevação ao cubo deles.









### Requisito 04: O código deverá ser modular, permitindo fácil extensão.

## Identificação no código:

- A modularidade está presente na separação das funcionalidades em funções como aplica\_func\_numeros, potenciacao. E também com os testes, tornando o código fácil de manter e de modificar no futuro, se necessário.







