# Documentação Técnica – Códigos (Legado vs Refatorado)

Escopo: documentação dos trechos destacados na interface (pares legado/refatorado), enfatizando o que cada trecho executa e por que foi refatorado.

## Sumário



## 1. Trecho: Soma de números (Java → Python)

Função executada: somar os elementos de uma coleção numérica e emitir o total.

### *Código legado (Java):*

```
public class SumExample {
  public static void main(String[] args) {
    int[] numbers = {1, 2, 3, 4, 5};
    int sum = 0;
    for (int num : numbers) {
      sum += num;
    }
    System.out.println("The sum of the numbers is: " + sum);
  }
}
```

### *Trecho refatorado (Python):*

```
# This program calculates the sum of a list of numbers
numbers = [1, 2, 3, 4, 5]
sum_value = 0
for num in numbers:
    sum_value += num
print(f"The sum of the numbers is: {sum_value}")
```

### *Descrição & Explicação*

• Estruturas: em Java, uso de array primitivo e laço for■each; em Python, lista e laço for. • Acumulação: variável sum/sum_value é incrementada a cada iteração. • E/S: Java escreve em System.out; Python usa print.

### *Motivo da refatoração*

• Reduzir verbosidade e aproximar do ecossistema usado no projeto atual. • Facilitar evolução (ex.: troca de lista para geradores, uso de sum()).

### *Complexidade*

• Tempo O(n), espaço O(1) adicional. Mesma semântica em ambas as versões.

## 2. Função total de valores (Java → Python)

Função executada: calcular o total (soma) de um vetor/lista de valores numéricos.

### *Código legado (Java):*

```
public static double total(double[] values) {
  double t = 0;
  for (int i = 0; i < values.length; i++) {
    t += values[i];
  }
  return t;
}
```

### *Trecho refatorado (Python):*

```
def total(values: list[float]) -> float:
    return sum(values)
```

### *Descrição & Explicação*

• Java itera por índice e acumula manualmente. • Python delega a soma para a função embutida sum(), que é otimizada e mais legível. • Assinatura explícita com anotação de tipos (list[float] → float).

### *Motivo da refatoração*

• Remover boilerplate de laço/índice. • Melhorar legibilidade e potencial de otimização interna.

### *Complexidade*

• Tempo O(n), espaço O(1) adicional. Comportamento idêntico ao legado.

## 3. Diretrizes de Refatoração aplicadas

| # | Diretriz |
|---|---|
| 1 | Preferir construções nativas de alto nível (sum, map, comprehensions). |
| 2 | Eliminar laços e contadores desnecessários quando houver APIs da linguagem para o propósito. |
| 3 | Explicitar tipos onde útil para manutenção/testes (anotações em Python). |
| 4 | Manter equivalência semântica e cobertura de testes ao migrar (mesmos casos e limites). |

## 4. Anexos: Como validar e testar

• Testes de unidade sugeridos: validar soma vazia (0), positivos, negativos, mistos, valores grandes. • Exemplo em Python:

```
def test_total_basic():
    assert total([1,2,3,4,5]) == 15

def test_total_empty():
    assert total([]) == 0

def test_total_negative():
    assert total([-2, 5, -3]) == 0
```

• Observação: Se houver necessidade de alta performance em grandes coleções, considerar iteradores e evitar materializações desnecessárias.