# Instruções Condicionais

# Exemplo de if-else em **R**

x <- 10

if (x > 5) {

print("x é maior que 5")

} else {

print("x é menor ou igual a 5")

}

# Exemplo de if-else em **Python**

x = 10

if x > 5:

print("x é maior que 5")

else:

print("x é menor ou igual a 5")

# Loops

# Exemplo de loop for em **R**

for (i in 1:5) {

print(i)

}

# Iterar sobre a sequência de 1 a 5 no Python

for i in range(1, 6):

print(i)

# Exemplo de loop while em **R**

count <- 1

while (count <= 5) {

print(count)

count <- count + 1

}

# Loop while para imprimir os números de 1 a 5 no Python

count = 1

while count <= 5:

print(count)

count += 1

# Controle de Fluxo

# Exemplo de break em **R**

for (i in 1:10) {

if (i == 6) {

break

}

print(i)

}

# Exemplo de next (similar ao continue em outras linguagens) em **R**

for (i in 1:5) {

if (i %% 2 == 0) {

next

}

print(i)

}

# Exemplo de break em **Python**

for i in range(1, 11):

if i == 6:

break

print(i)

# Exemplo de continue em **Python**

for i in range(1, 6):

if i % 2 == 0:

continue

print(i)

# Funções em R

# Exemplo de escopo local e global em **R**

x <- "global"

funcaoTeste <- function() {

x <- "local"

print(paste("Dentro da função: x =", x))

}

funcaoTeste()

print(paste("Fora da função: x =", x))

# R passa argumentos por valor

alteraValor <- function(num) {

num <- num \* 10

print(paste("Dentro da função: num =", num))

}

num <- 5

alteraValor(num)

print(paste("Fora da função: num =", num))

# Funções em Python

# Exemplo de escopo local e global em Python

x = "global"

def funcao\_teste():

global x

x = "local"

print(f"Dentro da função: x = {x}")

funcao\_teste()

print(f"Fora da função: x = {x}")

# Python passa argumentos imutáveis por valor e mutáveis por referência

def altera\_valor(num):

num \*= 10

print(f"Dentro da função: num = {num}")

num = 5

altera\_valor(num)

print(f"Fora da função: num = {num}")

# Demonstração com um objeto mutável (lista)

def altera\_lista(lista):

lista.append("novo elemento")

print(f"Dentro da função: lista = {lista}")

lista = ["original"]

altera\_lista(lista)

print(f"Fora da função: lista = {lista}")

### Imutáveis:

1. Inteiros (`int`)

2. Números de ponto flutuante (`float`)

4. Strings (`str`)

5. Tuplas (`tuple`)

6. Booleanos (`bool`)

### Mutáveis:

1. Listas (`list`)

2. Dicionários (`dict`)

3. Conjuntos (`set`)

# Programação Recursiva

# Função recursiva para calcular o fatorial em R[[1]](#footnote-1)

fatorial <- function(n) {

print(paste("Chamada: fatorial(", n, ")")) # Log para visualizar a chamada

if (n == 0) {

print("Caso base alcançado: fatorial(0) = 1")

return(1)

} else {

result <- n \* fatorial(n - 1)

print(paste("Retorno: fatorial(", n, ") = ", result))

return(result)

}

}

# Exemplo de uso

fatorial(5) # Resultado: 120

# Função recursiva para calcular o Valor Presente Líquido (VPL) no **Python**

def calcular\_vpl(fluxos, taxa, periodo=0):

if periodo == len(fluxos):

return 0

else:

presente = fluxos[periodo] / ((1 + taxa)\*\*periodo)

print(f"Período {periodo}: valor presente = {presente:.2f}")

return presente + calcular\_vpl(fluxos, taxa, periodo + 1)

# Exemplo de uso: uma série de fluxos de caixa futuros

fluxos\_de\_caixa = [1000, 1500, 2000, 2500, 3000] # fluxos próximos 5 anos

taxa\_de\_desconto = 0.05 # taxa de desconto anual de 5%

vpl = calcular\_vpl(fluxos\_de\_caixa, taxa\_de\_desconto)

print(f"Valor Presente Líquido total = {vpl:.2f}")

# Programação em Paralelo

Exemplo Canônico: Cálculo de Números Primos

Vou mostrar como calcular números primos até um limite superior específico, tanto de forma serial quanto paralela, e medir o tempo de execução para cada abordagem.

library(parallel)

# Função para verificar se um número é primo

is\_prime <- function(n) {

if (n <= 1) return(FALSE)

for (i in 2:sqrt(n)) {

if (n %% i == 0) return(FALSE)

}

return(TRUE)

}

# Limites para o cálculo de números primos

n <- 1000000 # um milhão

# Configurando o cluster de núcleos

nucleos <- detectCores() - 1

cl <- makeCluster(nucleos)

clusterExport(cl, "is\_prime")

# Medindo o tempo de execução paralela

tempo\_paralelo <- system.time({

resultados\_primos\_paralelo <- parLapply(cl, 1:n, is\_prime)

})

stopCluster(cl)

print(paste("Tempo total de execução em paralelo: ", tempo\_paralelo["elapsed"], "segundos"))

# Medindo o tempo de execução serial

tempo\_serial <- system.time({

resultados\_primos\_serial <- sapply(1:n, is\_prime)

})

print(paste("Tempo total de execução serial: ", tempo\_serial["elapsed"], "segundos"))

**Explicação**

* **Função is\_prime**: Esta função verifica se um número é primo, sendo uma operação ideal para paralelismo porque cada número pode ser verificado independentemente dos outros.
* **Cálculo Paralelo vs. Serial**: No código paralelo, a função parLapply distribui a tarefa de verificar cada número entre os núcleos disponíveis, enquanto no serial, sapply processa cada número sequencialmente em um único núcleo.
* **Medição de Tempo**: Utilizamos system.time() para capturar o tempo total necessário para processar todos os números tanto no modo serial quanto no paralelo.

Este exemplo deve mostrar uma diferença significativa nos tempos de execução, com o paralelo sendo mais rápido que o serial para limites maiores de n, especialmente quando o cálculo é intensivo como o teste de primalidade. Isso ajuda a ilustrar claramente as vantagens do processamento paralelo para tarefas apropriadas.

**OBS**: O operador %% no R é conhecido como o operador de módulo ou resto da divisão. Ele retorna o resto de uma divisão de um número pelo outro. É uma ferramenta útil em muitos contextos de programação, especialmente em algoritmos que envolvem divisibilidade, ciclos ou padrões repetitivos.

1. Quando você executa uma função recursiva, como a função de cálculo do fatorial, a sequência de chamadas recursivas continua até que o caso base seja alcançado. O caso base é o ponto de parada da recursão, após o qual não há mais chamadas recursivas. Aqui está o que acontece, passo a passo:

   **Chamadas Recursivas:** Cada chamada à função fatorial faz uma nova chamada a si mesma com n - 1 até que n seja 0 (o caso base). Cada chamada é empilhada na pilha de execução do programa.

   **Caso Base:** Quando o caso base é alcançado (n == 0), a função retorna 1. É nesse ponto que você vê o log "Caso base alcançado: fatorial(0) = 1". Até este momento, nenhum retorno da função recursiva aconteceu, apenas chamadas estão sendo empilhadas.

   **Desempilhamento e Retornos:** Após o caso base, as funções começam a "desempilhar". Cada função que foi chamada agora completa sua execução e retorna seu valor. Isto é, a função que foi chamada com n = 1 espera pelo retorno de fatorial(0), calcula seu resultado (1 \* 1), e então retorna 1. Esse retorno é então usado pela função chamada com n = 2, que calcula 2 \* 1 e retorna 2, e assim por diante. [↑](#footnote-ref-1)