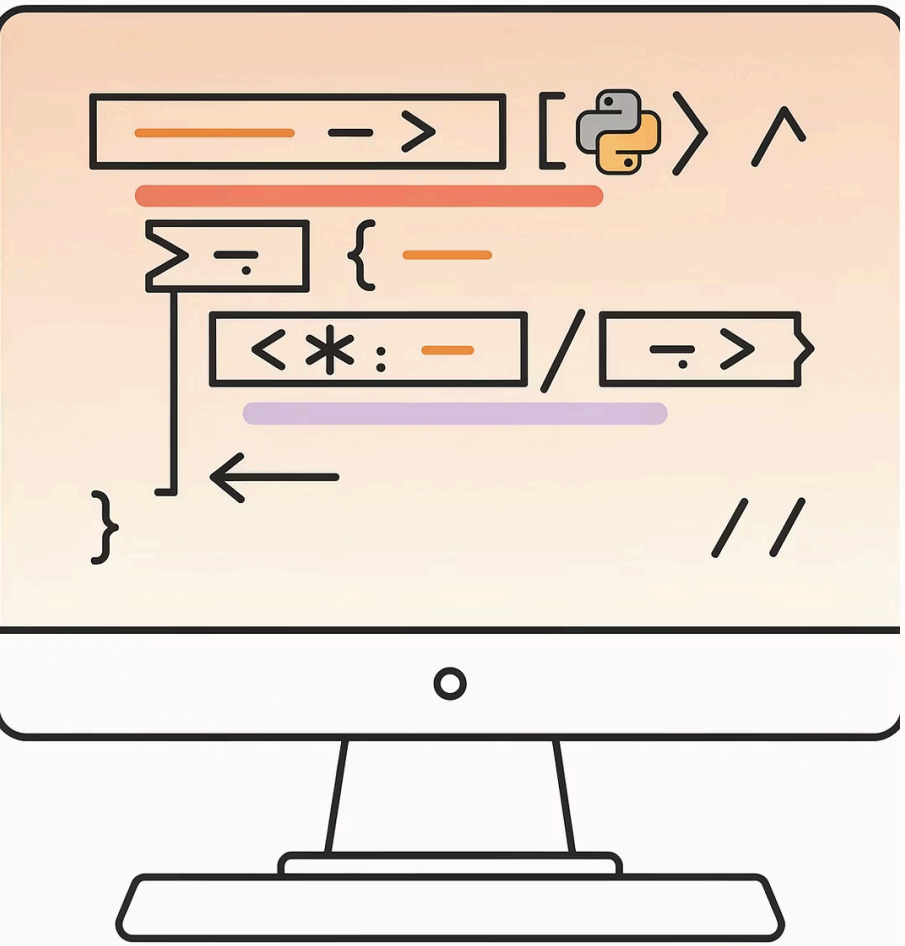


CEFET/RJ

Estrutura de Repetição em Python

Estruturas de repetição permitem automatizar tarefas e processar dados de forma eficiente em Python. Elas evitam repetir manualmente o mesmo comando várias vezes. Loops são essenciais quando trabalhamos com muitos valores, tornando o código mais limpo e poderoso.

Eduardo Ogasawara
eduardo.ogasawara@cefet-rj.br
https://eic.cefet-rj.br/~eogasawara

CONCEITOS BÁSICOS

Controle de Fluxo: Tomada de Decisão

O que é controle de fluxo

Programas não executam sempre as mesmas instruções. Condições e repetições permitem que o código se adapte a situações diferentes. O programa decide o que executar e quantas vezes.

Exemplo

```
x = 10
if x > 5:
    print("Maior que 5")
else:
    print("5 ou menos")
```

Saída

```
Maior que 5
```

DADOS ESCALARES

Trabalhando com Dados Simples

Valores individuais

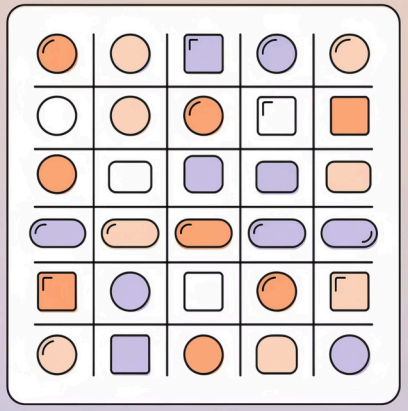
Podemos representar uma única pessoa usando variáveis escalares. Cada variável guarda um valor único na memória.

Cálculo de IMC

Vamos calcular o IMC de um indivíduo usando valores simples.

```
weight = 60
height = 1.75
subject = "A"
healthy = True
bmi = weight / height**2
bmi
```

Resultado: 19.59183673469388



Evoluindo para Múltiplos Valores

Usando vetores (arrays)

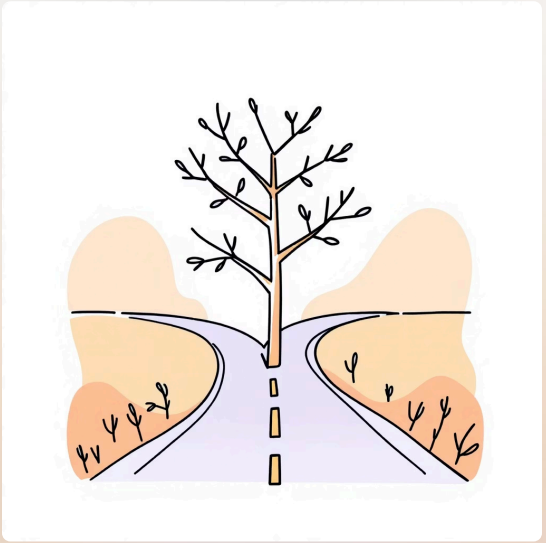
Quando temos várias pessoas, usamos estruturas que guardam múltiplos valores. Em Python usamos arrays do NumPy para isso. Agora precisamos de estruturas de repetição.

```
import numpy as np

weight = np.array([60, 72, 57, 90, 95, 72])
height = np.array([1.75, 1.80, 1.65, 1.90, 1.74, 1.91])
subject = np.array(["A", "B", "C", "D", "E", "F"])

weight
```

Saída: array([60, 72, 57, 90, 95, 72])

CONDICIONAIS

Estruturas Condicionais (if / else)



Tomada de decisão

Estruturas condicionais permitem escolher entre caminhos diferentes.

Avaliação

Uma condição é avaliada como verdadeira ou falsa.

Execução

Apenas um bloco de código é executado com base no resultado.

```
nota = 7
if nota >= 6:
   resultado = "aprovado"
else:
   resultado = "reprovado"
resultado
```

Exemplo de Decisão com if

Condição avaliada

O if avalia uma única condição lógica: `nota >= 6`

1

Decisão tomada

O fluxo do programa depende do valor atual da variável.

2

Resultado final

Se verdadeira, um bloco executa; caso contrário, o else executa.

3

```
nota = 7
if nota >= 6:
    resultado = "aprovado"
else:
    resultado = "reprovado"
resultado
```

Saída: 'aprovado'

DIFERENÇA IMPORTANTE

Condições em Vetores: if vs condicional vetorizada

if tradicional

Espera um único valor booleano (True ou False). Não funciona diretamente com vetores.

np.where

Para aplicar uma condição a cada elemento de um vetor, usamos operações vetorizadas no NumPy.

```
import numpy as np

alturas = np.array([1.65, 1.80, 1.55])
classe = np.where(alturas < 1.70, "baixa", "alta")
classe
```

Saída: array(['baixa', 'alta', 'baixa'], dtype='

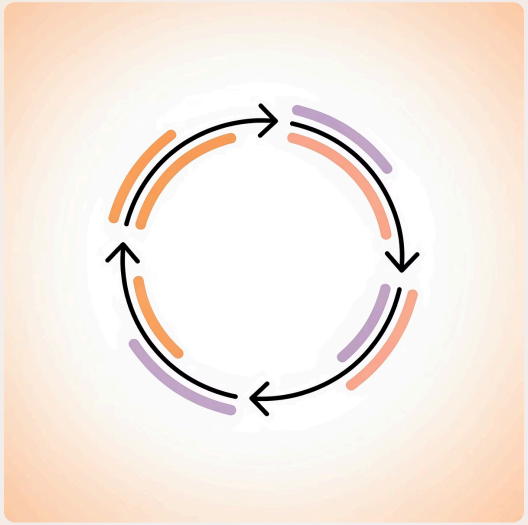
Condicional Vetorizada com np.where

Elemento a elemento

- A condição é avaliada para cada posição do vetor
- O resultado é um novo vetor com os valores escolhidos
- Isso permite transformar dados em lote

```
alturas = np.array([1.65, 1.80, 1.55])
classe = np.where(alturas < 1.70, "baixa", "alta")
classe
```

Altura	Condição	Classe
1.65	< 1.70 ✓	baixa
1.80	< 1.70 ×	alta
1.55	< 1.70 ✓	baixa



LOOPS

Repetição: Quando e Por Que Usar

01

Quando repetir

Repetição aplica a mesma operação várias vezes de forma automática.

02

Laços (for, while)

Controlam a execução passo a passo, iterando sobre elementos.

03

Operações vetorizadas

São mais rápidas quando possíveis, processando todos os dados de uma vez.

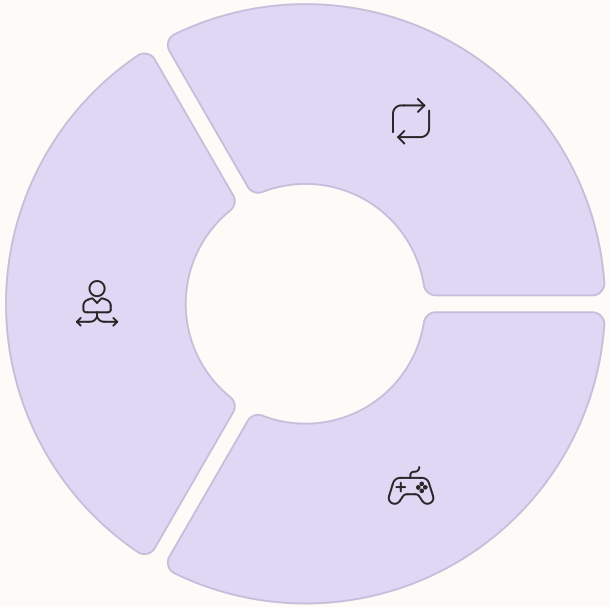
```
import numpy as np

x = np.array([1, 2, 3, 4])
y = x**2
y
```

Saída: array([1, 4, 9, 16])

Relação entre Condição e Repetição

Condições decidem

Determinam se algo deve ou não executar



Repetições executam

Realizam operações várias vezes

Combinação poderosa

Condições dentro do loop controlam quando agir

```
i = 1
while i <= 5:
    if i % 2 == 0:
        print(i, "é par")
    i += 1
```

Saída: 2 é par e 4 é par

∞ LOOP FOR

Estrutura de Repetição: for

O que é o for

Usamos **for** quando sabemos quantas vezes o código deve ser executado. Ele percorre sequências como listas, arrays ou intervalos. É uma das estruturas mais usadas em Python.

Características principais

- Iteração sobre sequências
- Número definido de repetições
- Sintaxe limpa e legível

```
for i in range(1, 6):
    print(i)
```

Saída

```
1
2
3
4
5
```

Calculando IMC com for (forma idiomática)

Percorrendo vetores em paralelo

A função `zip` permite iterar sobre dois vetores ao mesmo tempo.

Evitando índices manuais

Não precisamos gerenciar índices explicitamente.

Código mais seguro

O código fica mais simples, limpo e menos propenso a erros.

```
import numpy as np

weight = np.array([60, 72, 57, 90, 95, 72])
height = np.array([1.75, 1.80, 1.65, 1.90, 1.74, 1.91])

bmi = []
for w, h in zip(weight, height):
    bmi.append(w / h**2)

bmi = np.array(bmi)
bmi
```

Resultado: array([19.59, 22.22, 20.94, 24.93, 31.38, 19.74])

DEPURAÇÃO

Inspecionando os Cálculos

Acompanhando o loop

A função `print()` dentro do loop mostra cada passo da execução. Isso ajuda a entender como o vetor é preenchido progressivamente.

É uma técnica simples mas poderosa de depuração que revela a lógica interna do programa.

```
import numpy as np

bmi = np.zeros(len(weight))
for i in range(len(weight)):
    bmi[i] = weight[i] / height[i]**2
    print(bmi)
```

Saída progressiva: O vetor começa com zeros e cada posição é calculada uma por vez, mostrando a evolução do preenchimento.

Depurando no VS Code

Breakpoints

Permitem parar o programa em pontos específicos para análise detalhada.

Observando variáveis

Podemos ver os valores de todas as variáveis em tempo real a cada iteração.

Facilitando correções

Isso facilita muito o entendimento de erros e da lógica do programa.

```
for w, h in zip(weight, height):
    bmi_val = w / h**2   # breakpoint aqui
    print(w, h, bmi_val)
```

Cada iteração mostra os valores de peso, altura e IMC calculado, permitindo verificar a correção dos cálculos.

BOAS PRÁTICAS

Escopo, Nomes e Reprodutibilidade

Escopo de variáveis

Variáveis existem dentro de blocos e arquivos específicos. Entender o escopo evita conflitos.

Reprodutibilidade

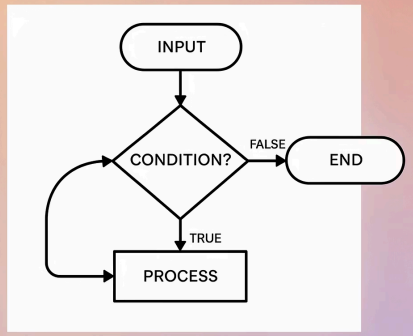
Rodar o script do início ao fim garante resultados consistentes e previsíveis.

Evitando erros

Isso evita resultados errados causados por resíduos de memória de execuções anteriores.

```
def compute():
    x = 10
    return x * 2

compute()
```

Saída: `20`



LOOP WHILE

Estrutura de Repetição: while

1

O que é o while

Executa o código enquanto uma condição for verdadeira

2

Repetições indefinidas

O número de repetições não é conhecido previamente

3

Cuidado essencial

A condição deve mudar para evitar loops infinitos

```
i = 1
while i <= 5:
    print(i)
    i += 1
```

Saída

```
1
2
3
4
5
```

Calculando IMC com while

1

Inicialização

Precisamos inicializar a variável de controle antes do loop

2

Condição

A condição decide se o loop continua ou para

3

Atualização

O índice é atualizado manualmente a cada iteração

```
import numpy as np

i = 0
bmi = np.zeros(len(weight))
while i < len(weight):
    bmi[i] = weight[i] / height[i]**2
    i += 1

bmi
```

Saída: array([19.59, 22.22, 20.94, 24.93, 31.38, 19.74])

FUNÇÕES

Encapsulando o Cálculo em uma Função

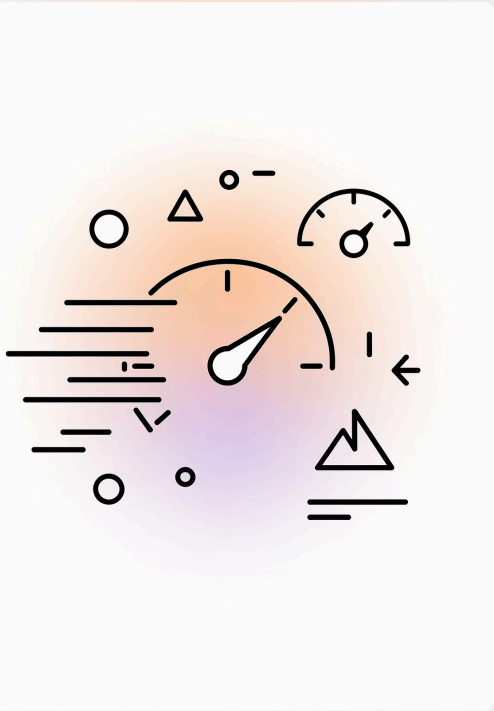
Organização e reutilização

Funções agrupam código que realiza uma tarefa específica. Isso torna o programa mais organizado, legível e reutilizável em diferentes contextos.

Vamos calcular o IMC usando uma função com **while**.

```
import numpy as np

def compute_bmi(weight, height):
    i = 0
    bmi = np.zeros(len(weight))
    while i < len(weight):
        bmi[i] = weight[i] / height[i]**2
        i += 1
    return bmi

bmi = compute_bmi(weight, height)
bmi
```

Resultado: array([19.59, 22.22, 20.94, 24.93, 31.38, 19.74])



OTIMIZAÇÃO

Implementando do Jeito Mais Eficiente

Sem loops necessários

Não precisamos de loops para este cálculo específico

Operação vetorizada

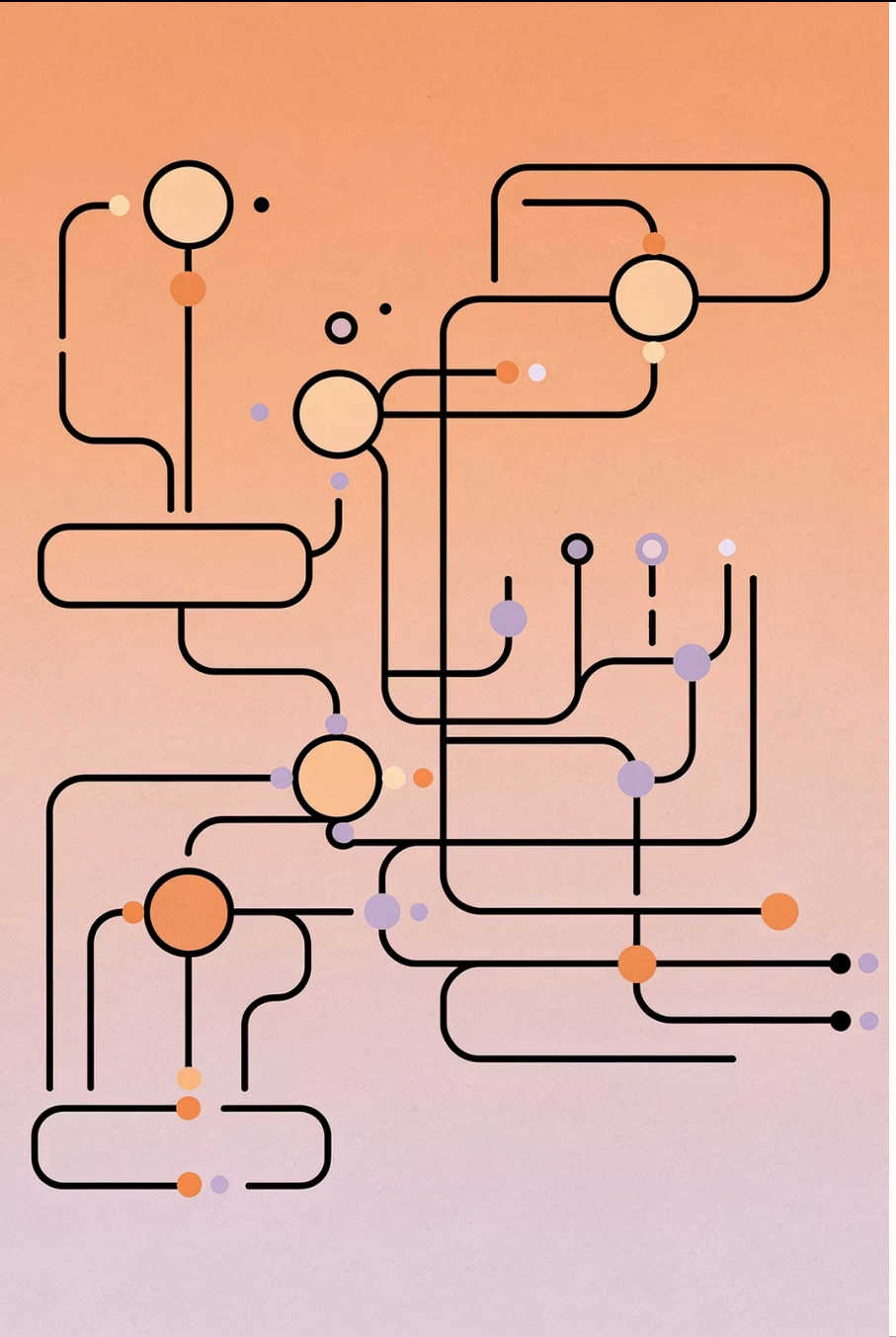
O NumPy opera em vetores inteiros de uma só vez

Código otimizado

O código fica mais rápido, limpo e pythônico

```
def compute_bmi(weight, height):
    return weight / height**2

bmi = compute_bmi(weight, height)
bmi
```

Saída: array([19.59, 22.22, 20.94, 24.93, 31.38, 19.74])



Usando a Função com Escalares e Vetores

1

Função única

Uma mesma implementação serve para diferentes tipos de dados

2

Aplicação automática

O NumPy aplica automaticamente a operação ao tipo correto

0

Loops desnecessários

Evitamos escrever loops manuais, simplificando o código

Com escalar (um valor)

```
compute_bmi(80, 1.79)
```

Saída: `24.97`

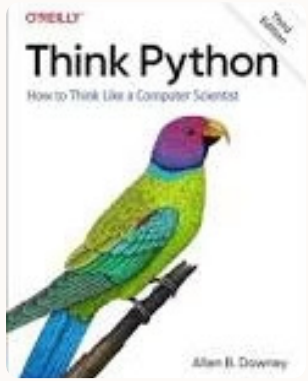
Com vetores (múltiplos valores)

```
compute_bmi(weight, height)
```

Saída: `array([19.59, 22.22, 20.94, 24.93, 31.38, 19.74])`

Flexibilidade total: A mesma função funciona perfeitamente tanto com valores individuais quanto com arrays, demonstrando o poder da vetorização do NumPy.

Referências



Think Python

Downey, A. *Think Python: How to Think Like a Computer Scientist.* O'Reilly Media.

An essential introduction to programming fundamentals and computational thinking using Python.

Python Data Science Handbook

VanderPlas, J. *Python Data Science Handbook.* O'Reilly Media.

A comprehensive guide to essential tools for working with data in Python, including NumPy, Pandas, and visualization libraries.

Data Science from Scratch

Grus, J. *Data Science from Scratch.* O'Reilly Media.

Learn data science fundamentals by building algorithms and tools from the ground up using Python.