Semana 19 - Lógica e Linguagem - Estruturas de Repetição

Tópico Principal da Aula: Estruturas de repetição - Criação de programas: prática

Subtítulo/Tema Específico: Conceitos Fundamentais e Aplicação Prática em Python (Aula 1)

Código da aula: [SIS]ANO1C1B3S19A1

Objetivos da Aula:

- Conhecer os conceitos sobre o desenvolvimento e execução prática de programas computacionais utilizando estruturas de repetição.

Competências Técnicas:

- Desenvolver sistemas computacionais utilizando ambiente de desenvolvimento.
- Migrar sistemas, implementando rotinas e estruturas de dados mais eficazes.

Competências Socioemocionais:

- Resolver problemas computacionais com estratégias criativas.

Recursos Adicionais (Sugestão, pode ser adaptado):

- Recurso audiovisual para exibição de vídeos e imagens.
- Caderno para anotações.
- Acesso ao laboratório e computadores para resolução das atividades.
- Duração da aula: 50 minutos.

Exposição do Conteúdo:

Referência do Slide: Slide 06 - Vamos relembrar o que são as estruturas de repetição?

- **Definição:** Estruturas de repetição, também conhecidas como laços ou loops, são construções de programação que permitem executar um bloco de código várias vezes. Em Python, o laço
  `for` é utilizado para iterar sobre uma sequência (como listas, tuplas, dicionários, conjuntos ou strings). Já o laço `while` é empregado para repetir um bloco de código enquanto uma condição específica for verdadeira..

Variável

Uma **variável** é um espaço na memória do computador reservado para armazenar um valor. Ela possui um nome (identificador) que você utiliza para se referir a esse valor ao longo do seu código. Em Python, você não precisa declarar o tipo de uma variável; o interpretador o infere automaticamente com base no valor que você atribui a ela.

Características:

- **Nome:** Um nome único para identificar o valor.
- **Valor:** A informação que a variável guarda (pode ser um número, um texto, uma lista, etc.).
- **Dinamicamente Tipada:** O tipo da variável pode mudar durante a execução do programa se você atribuir um novo valor de um tipo diferente.

Exemplo:

Python

```
# Atribuindo um número inteiro a uma variável
idade = 30

# Atribuindo um texto (string) a uma variável
nome = "João da Silva"

# Atribuindo um número decimal (float)
altura = 1.75

# Atribuindo um valor booleano (verdadeiro ou falso)
e_maior_de_idade = True

# Imprimindo os valores das variáveis
print(f"Nome: {nome}")
print(f"Idade: {idade}")
print(f"Altura: {altura}")
print(f"É maior de idade? {e_maior_de_idade}")
```

Lista (list)

Uma **lista** é uma coleção ordenada e *mutável* de itens. "Ordenada" significa que os itens mantêm uma posição definida, e "mutável" significa que você pode alterar seu conteúdo (adicionar, remover ou modificar itens) após sua criação. As listas são definidas por colchetes `[]` e os itens são separados por vírgulas.

Características:

- **Ordenada:** Os elementos têm um índice, começando em 0.
- **Mutável:** Pode ser alterada após a criação.
- **Heterogênea:** Pode conter itens de diferentes tipos (números, textos, outras listas, etc.).

Exemplo:

Python

```
# Criando uma lista de frutas
frutas = ["maçã", "banana", "laranja"]
print(f"Lista original: {frutas}")

# Acessando um item pelo índice
primeira_fruta = frutas[0]
print(f"A primeira fruta é: {primeira_fruta}")

# Modificando um item
```

```
frutas[1] = "morango"
print(f"Lista após modificação: {frutas}")

# Adicionando um item ao final da lista
frutas.append("uva")
print(f"Lista após adicionar 'uva': {frutas}")

# Removendo um item
frutas.remove("laranja")
print(f"Lista após remover 'laranja': {frutas}")
```

Tupla (tuple)

Uma **tupla** é uma coleção ordenada e *imutável* de itens. Assim como as listas, os itens são ordenados e acessados por um índice. A principal diferença é que, uma vez criada, uma tupla não pode ser alterada. Tuplas são definidas por parênteses `()` com os itens separados por vírgulas.

Características:

- **Ordenada:** Os elementos têm um índice.
- **Imutável:** Não pode ser alterada após a criação.
- **Desempenho:** Geralmente, são mais rápidas que as listas.

Exemplo:

Python

```
# Criando uma tupla de coordenadas
coordenadas = (10, 20)
print(f"Coordenadas: {coordenadas}")

# Acessando um item pelo índice
eixo_x = coordenadas[0]
print(f"Valor do eixo X: {eixo_x}")

# Tentativa de modificar uma tupla (resultará em erro)
try:
    coordenadas[0] = 30
except TypeError as e:
    print(f"Erro ao tentar modificar a tupla: {e}")

# Tuplas são úteis para dados que não devem mudar, como dias da semana
dias_semana = ("Segunda", "Terça", "Quarta", "Quinta", "Sexta", "Sábado", "Domingo")
```

Dicionário (dict)

Um **dicionário** é uma coleção não ordenada de pares *chave-valor*. Em vez de usar índices numéricos, você acessa os valores através de uma "chave" única. Dicionários são mutáveis e são ideais para armazenar dados que possuem uma relação clara entre si. São definidos por chaves `{}`.

Características:

- **Não Ordenado (em versões antigas do Python):** A ordem de inserção não era garantida. A partir do Python 3.7, os dicionários mantêm a ordem de inserção.
- **Mutável:** Você pode adicionar, remover e modificar pares chave-valor.
- **Indexado por Chaves:** As chaves devem ser únicas e de tipos imutáveis (como strings, números ou tuplas).

Exemplo:

Python

```
# Criando um dicionário para armazenar informações de um aluno
aluno = {
    "nome": "Maria",
    "idade": 21,
    "curso": "Engenharia de Software",
    "ativo": True
}
print(f"Dicionário do aluno: {aluno}")

# Acessando um valor pela sua chave
nome_aluno = aluno["nome"]
print(f"Nome do aluno: {nome_aluno}")

# Modificando um valor
aluno["idade"] = 22
print(f"Dicionário após atualizar a idade: {aluno}")

# Adicionando um novo par chave-valor
aluno["cidade"] = "São Paulo"
print(f"Dicionário após adicionar a cidade: {aluno}")

# Removendo um par chave-valor
del aluno["ativo"]
print(f"Dicionário após remover a chave 'ativo': {aluno}")
```

Vetor e Matriz

Em Python, os termos **vetor** (array unidimensional) e **matriz** (array bidimensional) não são tipos de dados nativos como listas ou dicionários. No entanto, eles podem ser facilmente representados usando listas. Para aplicações que exigem alto desempenho computacional (como em ciência de dados e aprendizado de máquina), é comum usar a biblioteca **NumPy**, que oferece estruturas de dados `array` muito mais eficientes.

1. Usando Listas Nativas

- **Vetor:** Um vetor pode ser representado por uma simples lista.
- **Matriz:** Uma matriz pode ser representada por uma lista de listas, onde cada lista interna representa uma linha da matriz.

Exemplo com Listas:

Python

```
# Vetor (array de 1 dimensão)
vetor_temperaturas = [25.5, 26.1, 24.8, 27.0]
print(f"Vetor de temperaturas: {vetor_temperaturas}")
print(f"Temperatura na segunda medição: {vetor_temperaturas[1]}")

# Matriz (array de 2 dimensões) 2x3
matriz_numeros = [
    [1, 2, 3],  # Linha 0
    [4, 5, 6]   # Linha 1
]
print(f"\nMatriz de números:\n{matriz_numeros}")

# Acessando o elemento na linha 1, coluna 2 (valor 6)
elemento = matriz_numeros[1][2]
print(f"Elemento na linha 1, coluna 2: {elemento}")
```

2. Usando a Biblioteca NumPy

A biblioteca NumPy é a forma padrão e mais eficiente para se trabalhar com vetores e matrizes em Python para fins numéricos.

Exemplo com NumPy:

Python

```
# É necessário instalar a biblioteca primeiro: pip install numpy
import numpy as np

# Vetor com NumPy
vetor_np = np.array([10, 20, 30, 40])
print(f"Vetor NumPy: {vetor_np}")
print(f"Tipo do objeto: {type(vetor_np)}")

# Matriz com NumPy 3x3
matriz_np = np.array([
    [9, 8, 7],
    [6, 5, 4],
    [3, 2, 1]
])
print(f"\nMatriz NumPy:\n{matriz_np}")

```

- # Operações matemáticas são muito mais fáceis com NumPy
- print(f"\nMultiplicando todo o vetor por 2: {vetor_np * 2}")
- **Aprofundamento/Complemento (se necessário):** Compreender e dominar as estruturas de repetição é crucial para qualquer programador, pois elas otimizam o código, reduzem a redundância e permitem a automação de tarefas repetitivas. A escolha entre for e while depende da necessidade: **for é ideal quando o número de iterações é conhecido ou quando se itera sobre elementos de uma coleção, enquanto while é mais adequado para situações onde a repetição continua até que uma condição seja satisfeita.**
- **Exemplo Prático:**
  - **Laço for:** Para iterar sobre uma lista de números:
  - Python

```
for numero in [1, 2, 3, 4, 5]:
    print(numero)
```

  -
  -
  - **Laço while:** Para contar de 1 a 5:
  - Python

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

Referência do Slide: Slide 07 - Exemplos práticos – Laço for

- **Definição:** O laço for permite a criação concisa de listas, conhecida como "compreensão de lista", e a iteração sobre múltiplas sequências simultaneamente usando a função zip(). Geradores são uma forma eficiente de iteração para grandes conjuntos de dados, pois geram itens sob demanda, economizando memória.
- **Aprofundamento/Complemento (se necessário):** A compreensão de lista é uma característica poderosa do Python que permite criar listas de forma mais legível e eficiente do que com um loop for tradicional em muitos casos. A função zip() é fundamental para combinar iteráveis, permitindo processar elementos correspondentes de diferentes sequências em uma única iteração. Geradores, por sua vez, são especialmente úteis para lidar com grandes volumes de dados que não cabem inteiramente na memória, como fluxos de arquivos ou resultados de consultas a bancos de dados.
- **Exemplo Prático:**
  - **Compreensão de Lista:** Criar uma lista de quadrados dos números de 0 a 9.
  - Python

```
quadrados = [x**2 for x in range(10)]
print(quadrados) # Saída: [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

  -
  -
  - **Iterar sobre múltiplas sequências com zip():**
  - Python

```
nomes = ["Alice", "Bob", "Carlos"]
idades = [25, 30, 22]
for nome, idade in zip(nomes, idades):
    print(f"{nome} tem {idade} anos")
```

Referência do Slide: Slide 08 - Estruturas de repetição com Python - Laço while

- **Definição:** O laço while executa um bloco de código repetidamente enquanto uma condição especificada permanece verdadeira. Uma característica avançada é o uso do bloco
  else, que é executado após a conclusão normal do laço while (ou seja, quando a condição se torna falsa), a menos que o laço seja interrompido por um break. É importante ter cuidado com laços infinitos, que ocorrem quando a condição de parada nunca é atingida.

- **Aprofundamento/Complemento (se necessário):** A principal diferença entre for e while é que o for é geralmente usado quando o número de iterações é conhecido ou pode ser determinado a partir de uma sequência, enquanto o while é usado quando o número de iterações é desconhecido e a repetição depende de uma condição. O bloco else com while é útil para executar um código final somente se o laço while for concluído sem interrupção.
- **Exemplo Prático:**
  - **Uso de else com while:**
  - Python

```
contador = 0
while contador < 3:
    print(f"Contador: {contador}")
    contador += 1
else:
    print("Loop concluído normalmente.")
```

Referência do Slide: Slide 09 - Diferenças entre laço For e laço While

- **Definição:**
  - **Laço for:** Começa definindo o valor inicial de uma variável de controle. Antes de cada iteração, verifica se a condição especificada é verdadeira; se falsa, o laço encerra. Se verdadeira, o bloco de código é executado, e a variável de controle é atualizada (geralmente incrementada ou decrementada). O fluxo retorna à condição até que ela se torne falsa.
  - **Laço while:** Verifica se a condição inicial é verdadeira; se falsa, o laço encerra. Se verdadeira, o bloco de código é executado. Após a execução do bloco, o fluxo retorna à condição para verificar se ela ainda é verdadeira antes da próxima iteração, formando um ciclo até que a condição seja falsa.
- **Aprofundamento/Complemento (se necessário):** A principal distinção reside no controle do fluxo. O for é iterativo, projetado para percorrer elementos de uma sequência. O while é condicional, projetado para repetir enquanto uma condição específica for atendida. A escolha entre eles depende da clareza e da eficiência para a tarefa em questão.
- **Exemplo Prático:**
  - **Laço for (com range() para um número definido de repetições):**
  - Python

```
for i in range(5): # Repete 5 vezes (de 0 a 4)
    print(f"Iteração for: {i}")
```

-
-
- **Laço while (com condição explícita):**
- Python

```
i = 0
while i < 5: # Repete enquanto 'i' for menor que 5
    print(f"Iteração while: {i}")
    i += 1
```

Subtítulo/Tema Específico: Tratamento de Exceções e Boas Práticas (Aula 2)

Código da aula: [SIS]ANO1C1B3S19A2

Objetivos da Aula:

- Conhecer os conceitos sobre o desenvolvimento e execução prática de programas computacionais utilizando estruturas de repetição.

Competências Técnicas:

- Desenvolver sistemas computacionais utilizando ambiente de desenvolvimento.
- Migrar sistemas, implementando rotinas e estruturas de dados mais eficazes.

Competências Socioemocionais:

- Resolver problemas computacionais com estratégias criativas.

Recursos Adicionais (Sugestão, pode ser adaptado):

- Recurso audiovisual para exibição de vídeos e imagens.
- Caderno para anotações.
- Acesso ao laboratório e computadores para resolução das atividades.
- Duração da aula: 50 minutos.

Exposição do Conteúdo:

Referência do Slide: Slide 06 - Tratamento de exceções

- **Definição:** No contexto dos laços de repetição, o tratamento de exceções é fundamental para lidar com erros que podem ocorrer durante a execução do código, como divisões por zero ou acesso a índices inválidos. Em Python, isso é feito principalmente com os blocos
  try e except.
- **Aprofundamento/Complemento (se necessário):** O tratamento de exceções permite que um programa continue sua execução mesmo após a ocorrência de um erro, evitando que ele "trave". É uma prática essencial para criar software robusto e tolerante a falhas, especialmente em cenários onde a entrada de dados ou as condições de execução são imprevisíveis.
- **Exemplo Prático:**
- Python

```
numeros = [10, 0, 5, 2]
for num in numeros:
    try:
        resultado = 100 / num
        print(f"100 dividido por {num} é {resultado}")
    except ZeroDivisionError:
        print(f"Erro: Não é possível dividir por zero quando o número é {num}.")
```

- **Links de Vídeos:**
    - ALURA. Phyton: crie a sua primeira aplicação. 02 Try except: ALURA. Phyton: crie a sua primeira aplicação. 02 Try except
    -

Referência do Slide: Slide 07 - Boas práticas e dicas no uso de estruturas de repetição

- **Definição:** Ao trabalhar com laços de repetição, é importante seguir algumas boas práticas para garantir a eficiência e a robustez do código:
    - **Evitar laços infinitos acidentais:** Sempre verifique se as condições de parada do laço podem ser atingidas.
    - **Uso eficiente de iteradores:** Em Python, é preferível usar iteradores e geradores para laços, pois são mais eficientes em termos de memória.
    - **Compreensões de lista vs. laços:** Compreensões de lista podem ser mais rápidas e legíveis do que um laço `for` para criar listas.
- **Aprofundamento/Complemento (se necessário):** A otimização do uso de laços passa por entender quando e como aplicar cada tipo de estrutura. Laços infinitos, embora por vezes intencionais (como em servidores que escutam requisições continuamente), devem ser controlados com mecanismos de interrupção. O uso de iteradores e geradores é um conceito avançado que melhora o desempenho e o consumo de memória, especialmente em grandes conjuntos de dados, pois eles produzem elementos sob demanda. Compreensões de lista oferecem uma sintaxe concisa e muitas vezes mais performática para a criação de listas transformadas ou filtradas.
- **Exemplo Prático:**
    - **Evitando laço infinito:**
    - Python

```
contador = 0
while contador < 5:
    print(contador)
    contador += 1 # Garante que a condição 'contador < 5' se tornará falsa eventualmente
```

    -
    -
    - **Uso de iteradores (exemplo simples com `range`):**
    - Python

```
for i in range(1000000): # 'range' é um iterador e não cria a lista completa em memória
    pass
```

    -

    -
    - **Compreensão de lista:**
    - Python

```
quadrados = [x**2 for x in range(5)]
print(quadrados) # Saída: [0, 1, 4, 9, 16]
```

    -
    -

- **Links de Vídeos:**
    - Para aprofundar em iteradores e geradores: Pesquise por "Python iterators and generators tutorial".
    - Para mais sobre compreensões de lista: Pesquise por "Python list comprehensions best practices".

Subtítulo/Tema Específico: Aplicação de Listas e Laços for (Aula 3)

Código da aula: [SIS]ANO1C1B3S19A3

Objetivos da Aula:

- Conhecer os conceitos sobre o desenvolvimento e execução prática de programas computacionais utilizando estruturas de repetição.

Competências Técnicas:

- Desenvolver sistemas computacionais utilizando ambiente de desenvolvimento.
- Migrar sistemas, implementando rotinas e estruturas de dados mais eficazes.

Competências Socioemocionais:

- Resolver problemas computacionais com estratégias criativas.

Recursos Adicionais (Sugestão, pode ser adaptado):

- Recurso audiovisual para exibição de vídeos e imagens.
- Caderno para anotações.
- Acesso ao laboratório e computadores para resolução das atividades.
- Duração da aula: 50 minutos.

Exposição do Conteúdo:

Referência do Slide: Slide 06 - Aplicação de listas

- **Definição:** A aplicação de listas é um conceito importante ao trabalhar com laços de repetição. Listas são coleções ordenadas e mutáveis de itens, e os laços de repetição (especialmente o `for`) são frequentemente usados para processar ou iterar sobre os elementos de uma lista.
- **Aprofundamento/Complemento (se necessário):** Listas em Python são estruturas de dados muito versáteis que podem armazenar diferentes tipos de dados. A

iteração sobre listas com laços for é uma das operações mais comuns, permitindo realizar ações em cada elemento, como modificá-los, exibi-los ou usá-los em cálculos.

- **Exemplo Prático:**
- Python

```
frutas = ["maçã", "banana", "cereja"]
for fruta in frutas:
    print(f"Eu gosto de {fruta}")
```

-
-
- **Links de Vídeos:**
  - ALURA. Phyton: crie a sua primeira aplicação. 03 Listas: ALURA. Phyton: crie a sua primeira aplicação. 03 Listas
    51

Referência do Slide: Slide 07 - Aplicação dos laços de repetição for

- **Definição:** A aplicação prática dos laços de repetição for no desenvolvimento de software em Python é vasta, sendo essencial para processar coleções de dados, realizar repetições com número definido de iterações e muito mais.
- **Aprofundamento/Complemento (se necessário):** O laço for em Python é amplamente utilizado em cenários como:
  - Iteração sobre elementos de listas, tuplas, strings e outros iteráveis.
  - Execução de um bloco de código um número específico de vezes (usando range()).
  - Processamento de dados de arquivos linha por linha.
  - Criação de compreensões de lista e dicionário.
- **Exemplo Prático:**
- Python

```
# Iterando sobre uma string
for letra in "Python":
    print(letra)

# Contando de 0 a 4 usando range()
for i in range(5):
    print(i)
```

-
-
- **Links de Vídeos:**
  - ALURA. Phyton: crie a sua primeira aplicação. 05 Laço de repetição for: ALURA. Phyton: crie a sua primeira aplicação. 05 Laço de repetição for
    54
  - Quer aprimorar suas habilidades com a estrutura de repetição FOR no Python? Assista ao vídeo a seguir. Nele você irá descobrir como utilizá-la de forma prática e eficiente: HASHTAG PROGRAMAÇÃO. Estrutura de Repetição FOR no Python - Criando um Loop
    55

Referência do Slide: Saiba mais - Break e Continue

- **Definição:** As ferramentas break e continue são comandos úteis para gerenciar o fluxo de execução dentro dos laços de repetição em Python.
    - break: Interrompe completamente o laço atual e transfere o controle para a instrução imediatamente após o laço.
    - continue: Pula o restante do código dentro da iteração atual do laço e passa para a próxima iteração.
- **Aprofundamento/Complemento (se necessário):** break é frequentemente usado quando uma condição específica é atendida e não há mais necessidade de continuar as iterações do laço. continue é útil quando se deseja pular uma iteração específica que não atende a certas condições, sem interromper todo o laço.
- **Exemplo Prático:**
    - **Uso de break:**
    - Python

```
for i in range(10):
    if i == 5:
        break # Sai do loop quando i é 5
    print(i)
```

    -
    -
    - **Uso de continue:**
    - Python

```
for i in range(10):
    if i % 2 == 0: # Pula números pares
        continue
    print(i)
```

    -
    -
- **Links de Vídeos:**
    - Explore o potencial das ferramentas break e continue no Python com este vídeo explicativo! Aprenda a usar estes comandos para gerenciar loops de forma precisa e eficiente: HASHTAG PROGRAMAÇÃO. Break e Continue no Python - Ferramentas da Estrutura de Repetição
      57

Subtítulo/Tema Específico: Refatoração de Código e Dicionários (Aula 4)

Código da aula: [SIS]ANO1C1B3S19A4

Objetivos da Aula:

- Conhecer os conceitos sobre o desenvolvimento e execução prática de programas computacionais utilizando estruturas de repetição.

Competências Técnicas:

- Desenvolver sistemas computacionais utilizando ambiente de desenvolvimento.
- Migrar sistemas, implementando rotinas e estruturas de dados mais eficazes.

Competências Socioemocionais:

- Resolver problemas computacionais com estratégias criativas.

Recursos Adicionais (Sugestão, pode ser adaptado):

- Recurso audiovisual para exibição de vídeos e imagens.
- Caderno para anotações.
- Acesso ao laboratório e computadores para resolução das atividades.
- Duração da aula: 50 minutos.

Exposição do Conteúdo:

Referência do Slide: Slide 06 - Melhoria contínua - Refatoração de códigos

- **Definição:** A melhoria contínua no desenvolvimento de software inclui o conceito de refatoração de códigos. **Refatorar um código significa reestruturá-lo sem alterar seu comportamento externo, com o objetivo de melhorar sua legibilidade, manutenibilidade e eficiência.** Isso é crucial para aprimorar os processos de desenvolvimento.
- **Aprofundamento/Complemento (se necessário):** A refatoração é uma prática comum e essencial em engenharia de software. Ela visa tornar o código mais limpo, fácil de entender e de dar manutenção, o que é fundamental para projetos de longo prazo. Embora não adicione novas funcionalidades, a refatoração prepara o terreno para futuras expansões e reduz a probabilidade de bugs.
- **Exemplo Prático:**
    - **Antes da refatoração:**
    - Python

```
def calcular_total_pedido(preco1, qtd1, preco2, qtd2):
    total1 = preco1 * qtd1
    total2 = preco2 * qtd2
    return total1 + total2
```

    -
    -
    - **Depois da refatoração (tornando mais genérico e legível):**
    - Python

```
def calcular_item(preco, quantidade):
    return preco * quantidade

def calcular_total_pedido(itens):
    total = 0
    for item in itens:
        total += calcular_item(item['preco'], item['quantidade'])
    return total
```

    -
    -

- **Links de Vídeos:**
  - ALURA. Phyton: crie a sua primeira aplicação. 07 Refatorando o código: ALURA. Phyton: crie a sua primeira aplicação. 07 Refatorando o código 69696969

Referência do Slide: Slide 07 - Dicionários

- **Definição:** Dicionários são outra estrutura de dados fundamental em Python, especialmente úteis quando se avança na compreensão de listas e laços de repetição. Eles armazenam dados em pares de chave-valor, onde cada chave é única e mapeia para um valor.
- **Aprofundamento/Complemento (se necessário):** Dicionários são ideais para representar dados que têm uma relação chave-valor, como informações de contato (nome: telefone), configurações (opção: valor) ou registros de bancos de dados. Eles permitem acesso rápido aos valores através de suas chaves, e são mutáveis, o que significa que podem ser modificados após sua criação.
- **Exemplo Prático:**
- Python

```
# Criando um dicionário
aluno = {
    "nome": "Maria",
    "idade": 16,
    "curso": "Desenvolvimento de Sistemas"
}

# Acessando valores
print(f"Nome do aluno: {aluno['nome']}")
print(f"Idade do aluno: {aluno['idade']}")

# Iterando sobre um dicionário
for chave, valor in aluno.items():
    print(f"{chave}: {valor}")
```

-
-
- **Links de Vídeos:**
  - ALURA. Phyton: crie a sua primeira aplicação. 02 Dicionários: ALURA. Phyton: crie a sua primeira aplicação. 02 Dicionários.