Semana 10 Aula 1
Estruturas de Repetição
Atividades práticas - Criação de programas
Código da aula: [SIS]ANO1C1B2S10A1
Objetivos da Aula

- Aplicar o conceito de estrutura de repetição com enfoque no método While.
- Desenvolver sistemas computacionais utilizando ambiente de desenvolvimento;
- Resolver problemas computacionais com estratégias criativas.
- Caderno, canetas e lápis;
- Acesso ao laboratório de informática e/ou internet.
- Recursos didáticos Competências da unidade (técnicas e socioemocionais) O

Exposição

Atividades práticas - Criação de programas

Estruturas de repetição em programação: enfoque no laço while

As estruturas de repetição são fundamentais na programação, para realizar tarefas repetidas sem a necessidade de reescrever o mesmo código várias vezes. O laço while, em particular, é uma ferramenta poderosa para repetir instruções até que uma condição especificada deixe de ser verdadeira, permitindo uma maior flexibilidade em comparação com estruturas de repetição fixas.

Em programação, há três tipos principais de laços de repetição que são comumente utilizados em diversas linguagens de programação: **for, while e do-while.**

Laço for (para): *usado quando o número de iterações é conhecido antes do início do laço. É ideal para iterar sobre coleções de dados ou sequências em que o tamanho é predefinido.*

Exemplo em Python:

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

Laço while (enquanto): *apropriado quando o número de iterações não é conhecido e a decisão de sair do laço depende de condições que são avaliadas durante a execução do laço.*

Exemplo em Python

```
i = 0 while i < 5: # Executa enquanto i for menor que 5
print(i) i += 1
```

Do-While (Faça-Enquanto): realize determinada ação “Enquanto” tal condição for verdadeira.*****No python se usa o breack**

Semana 10 Aula 2
Estruturas de Repetição
Atividades práticas - Criação de programas
Código da aula: [SIS]ANO1C1B2S10A2
Objetivos da Aula

- ❖ Aplicar o conceito de estrutura de repetição com enfoque no método While.
- ❖ Desenvolver sistemas computacionais utilizando ambiente de desenvolvimento;
- ❖ Resolver problemas computacionais com estratégias criativas.
- ❖ Caderno, canetas e lápis;
- ❖ Acesso ao laboratório de informática e/ou internet.
- ❖ Recursos didáticos Competências da unidade (técnicas e socioemocionais) O

Exposição
Operadores:

- **Operadores Aritméticos:** Estes operadores são utilizados para realizar cálculos matemáticos básicos. Eles atuam sobre operandos numéricos (inteiros ou de ponto flutuante) e produzem um resultado numérico. As operações comuns incluem adição, subtração, multiplicação, divisão e outras operações relacionadas.

- **Operadores Booleanos** (ou Lógicos): Estes operadores trabalham com valores booleanos (verdadeiro ou falso) e produzem um resultado booleano. Eles são fundamentais para construir expressões lógicas e controlar o fluxo de execução em programas. Os operadores booleanos mais comuns são a conjunção (E/AND), a disjunção (OU/OR) e a negação (NÃO/NOT).

- **Operadores de Comparação (ou Relacionais):** Estes operadores são usados para comparar dois valores e determinar a relação entre eles. O resultado de uma operação de comparação é sempre um valor booleano (verdadeiro ou falso). As comparações típicas incluem igualdade, desigualdade, maior que, menor que, maior ou igual a, e menor ou igual a.

Tabela Demonstrativa dos Operadores:

Categoria	Operador	Símbolo	Exemplo (Considerando a=5\$ e \$b=3\)	Resultado do Exemplo
Aritméticos	Adição	+	a + b\$ \	\$8\
	Subtração	-	a - b\$ \	\$2\
	Multiplicação	*	a * b\$ \	\$15\
	Divisão	/	a / b\$ \	\$1.666...\
	Módulo	%	a \% b\$ \	\$2\
	Exponenciação	**	a ** b\$ \	\$125\
Booleanos	E (AND)	and	True and False	False

	OU (OR)	`or`	`True or False`	`True`
	NÃO (NOT)	`not`	`not True`	`False`
De Comparação	Igual a	`==`	a == b\	`False`
	Diferente de	`!=`	a != b\	`True`
	Maior que	`>`	a > b\	`True`
	Menor que	`<`	a < b\	`False`
	Maior ou igual a	`>=`	a >= b\	`True`
	Menor ou igual a	`<=`	a <= b\	`False`

Semana 10 Aula 3
Estruturas de Repetição
Atividades práticas - Criação de programas
Código da aula: [SIS]ANO1C1B2S10A3
Objetivos da Aula

- Aplicar o conceito de estrutura de repetição com enfoque no método While.
- Desenvolver sistemas computacionais utilizando ambiente de desenvolvimento;
- Resolver problemas computacionais com estratégias criativas.
- Caderno, canetas e lápis;
- Acesso ao laboratório de informática e/ou internet.
- Recursos didáticos Competências da unidade (técnicas e socioemocionais) O

Exposição
Aninhamento e escopo de variáveis
O aninhamento ocorre quando uma estrutura de controle é colocada dentro de outra, como um laço while dentro de outro laço while ou dentro de uma instrução if. Isso é útil quando se lida com situações que exigem várias camadas de decisão ou repetição.
O escopo de uma variável refere-se à parte do programa onde a variável é acessível. Variáveis definidas dentro de um laço têm escopo local; elas só podem ser acessadas dentro desse laço. Por outro lado, variáveis definidas fora de todos os laços têm escopo global, podendo ser acessadas de qualquer lugar do programa.

Exemplo de escopo em Python:

```
x = "global"

# Loop for para iterar duas vezes
for i in range(2):
    if i == 0:
        y = "local na primeira iteração"
    else:
        y = "local na segunda iteração"

    print(f"Iteração {i+1}:")
    print(f"    x: {x}")  # Pode acessar x
    print(f"    y: {y}")  # Pode acessar y, pois está definida dentro do loop
```

Depurador (Debugger)

Definição: Um depurador é uma ferramenta essencial em ambientes de desenvolvimento de software que permite aos programadores inspecionar e

controlar a execução de um programa passo a passo. Ele oferece a capacidade de pausar a execução em pontos específicos (breakpoints), observar o valor de variáveis em tempo real, analisar o fluxo de controle e identificar a causa de erros (bugs) no código.

Em essência, um depurador permite "olhar por dentro" do programa enquanto ele está rodando, tornando o processo de encontrar e corrigir erros muito mais eficiente do que simplesmente analisar o código estaticamente ou depender de instruções de impressão (como `print`) para entender o que está acontecendo.

Em resumo, o depurador oferece controle granular sobre a execução do código, permitindo identificar precisamente onde e por que os erros ocorrem.

Refatoração

Definição: Refatoração é o processo de reestruturar um código de computador existente sem alterar seu comportamento externo. O objetivo principal da refatoração é melhorar a qualidade interna do código, tornando-o mais fácil de entender, manter, modificar e estender no futuro.

A refatoração não corrige bugs (embora possa torná-los mais fáceis de encontrar e corrigir posteriormente). Em vez disso, ela foca em aspectos como:

- Legibilidade: Tornar o código mais claro e fácil de entender para outros desenvolvedores (e para o próprio autor no futuro).
- Simplicidade: Reduzir a complexidade desnecessária do código.
- Manutenibilidade: Facilitar a modificação e correção de bugs no futuro.
- Extensibilidade: Tornar mais fácil adicionar novas funcionalidades ao código.
- Desacoplamento: Reduzir as dependências entre diferentes partes do código.
- Remoção de Código Duplicado: Consolidar lógica repetida em um único lugar.
- Melhoria de Nomes: Escolher nomes mais descritivos para variáveis, funções e classes.

O código refatorado faz exatamente a mesma coisa que o código original, mas é mais fácil de ler, entender e modificar no futuro. Por exemplo, se a lógica de cálculo do aumento precisar ser alterada, ela estará em um único lugar.

Em resumo, um depurador ajuda a encontrar e corrigir erros durante a execução do programa, enquanto a refatoração melhora a estrutura interna e a qualidade do código existente sem alterar seu comportamento externo. Ambas são práticas cruciais para o desenvolvimento de software robusto e sustentável.

Semana 10 Aula 4
Estruturas de Repetição
Atividades práticas - Criação de programas
Código da aula: [SIS]ANO1C1B2S10A4
Objetivos da Aula

- ❖ Aplicar o conceito de estrutura de repetição com enfoque no método While.
- ❖ Desenvolver sistemas computacionais utilizando ambiente de desenvolvimento;
- ❖ Resolver problemas computacionais com estratégias criativas.
- ❖ Caderno, canetas e lápis;
- ❖ Acesso ao laboratório de informática e/ou internet.
- ❖ Recursos didáticos Competências da unidade (técnicas e socioemocionais) O

Exposição

Definição de Laços e Decisões

Estruturas de Decisão (ou Condicionais): São construções de programação que permitem que um programa execute diferentes blocos de código com base na avaliação de uma condição (que resulta em verdadeiro ou falso). Elas controlam o fluxo de execução, permitindo que certas partes do código sejam ignoradas ou executadas dependendo do resultado dessa avaliação. As estruturas de decisão mais comuns são o `if` (se), `else` (senão) e `elif` (senão se).

Estruturas de Repetição (ou Laços): São construções de programação que permitem que um bloco de código seja executado repetidamente até que uma determinada condição seja satisfeita ou um número específico de iterações seja alcançado. Elas automatizam tarefas repetitivas, evitando a necessidade de escrever o mesmo código várias vezes. Os laços mais comuns são `for` (para) e `while` (enquanto), e em algumas linguagens, `do-while` (faça-enquanto).

Relação entre Laços e Decisões

A relação entre laços e decisões é fundamental para criar programas com lógica complexa e dinâmica. Dentro de um laço, frequentemente encontramos estruturas de decisão que controlam o que acontece em cada iteração. Da mesma forma, as condições que governam a execução de um laço podem envolver múltiplas decisões lógicas.

Em essência:

- Decisões dentro de Laços: Permitem executar diferentes ações dentro de cada repetição do laço, dependendo de certas condições. Isso torna os laços mais flexíveis e capazes de lidar com diferentes situações durante a iteração.

- Decisões controlando Laços: As condições que determinam se um laço continua ou termina são geralmente expressões booleanas que podem envolver múltiplas comparações e operadores lógicos (ou seja, decisões).

Essa combinação permite criar algoritmos que não apenas repetem ações, mas também adaptam seu comportamento em cada repetição com base em avaliações lógicas.

Exemplos

Exemplo 1: Decisão dentro de um Laço (`for`)

Imagine que você tem uma lista de números e quer imprimir apenas os números pares:

```
Python

numeros = [1, 2, 3, 4, 5, 6]

for numero in numeros:
    if numero % 2 == 0:  # Decisão: verificar se o número é par
        print(f"{numero} é par")
    else:
        print(f"{numero} é ímpar")
```

Neste exemplo, o laço `for` itera por cada número da lista. Dentro do laço, a estrutura de decisão `if/else` verifica se o número atual é par (resto da divisão por 2 é 0). Dependendo do resultado dessa decisão, uma mensagem diferente é impressa.

Exemplo 2: Decisão controlando um Laço (`while`)

Imagine um programa que pede ao usuário para digitar um número positivo e continua pedindo até que ele digite um número válido:

```
Python

numero = 0
while numero <= 0:  # Laço continua enquanto a decisão for verdadeira (número não é positivo)
    entrada = input("Digite um número positivo: ")
    try:
        numero = int(entrada)
        if numero <= 0:
            print("Por favor, digite um número maior que zero.")
    except ValueError:
        print("Entrada inválida. Digite um número inteiro.")

print(f"Você digitou o número positivo: {numero}")
```

Aqui, o laço `while` continua executando enquanto a condição `numero <= 0` for verdadeira (uma decisão). Dentro do laço, há outra decisão (`if numero <= 0:`) para exibir uma mensagem de erro específica se o número digitado não for positivo, mesmo após a conversão para inteiro.

Exemplo 3: Múltiplas Decisões controlando um Laço (`while`)

Considere um jogo simples onde o jogador tem um certo número de tentativas para adivinhar um número secreto:

```
Python

numero_secreto = 42
tentativas_restantes = 3
acertou = False

while tentativas_restantes > 0 and not acertou:  # Laço continua se houver tentativas e o jogador não acertou (duas decisões)
    palpite = int(input(f"Tentativa {3 - tentativas_restantes + 1}: Digite seu palpite: "))
    tentativas_restantes -= 1

    if palpite == numero_secreto:  # Decisão 1: verificar se acertou
        print("Parabéns! Você acertou!")
        acertou = True
    elif palpite < numero_secreto:  # Decisão 2: verificar se o palpite é menor
        print("O número secreto é maior.")
    else:  # Decisão 3: se não for igual nem menor, é maior
        print("O número secreto é menor.")

if not acertou:
    print(f"Você perdeu! O número secreto era {numero_secreto}.")
```

Neste caso, o laço `while` continua enquanto duas condições (decisões) forem verdadeiras: ainda há tentativas restantes e o jogador não acertou. Dentro do laço, várias estruturas `if/elif/else` tomam decisões sobre a relação entre o palpite do jogador e o número secreto.

Esses exemplos ilustram como laços e decisões trabalham em conjunto para criar programas com fluxo de controle dinâmico e adaptável.