



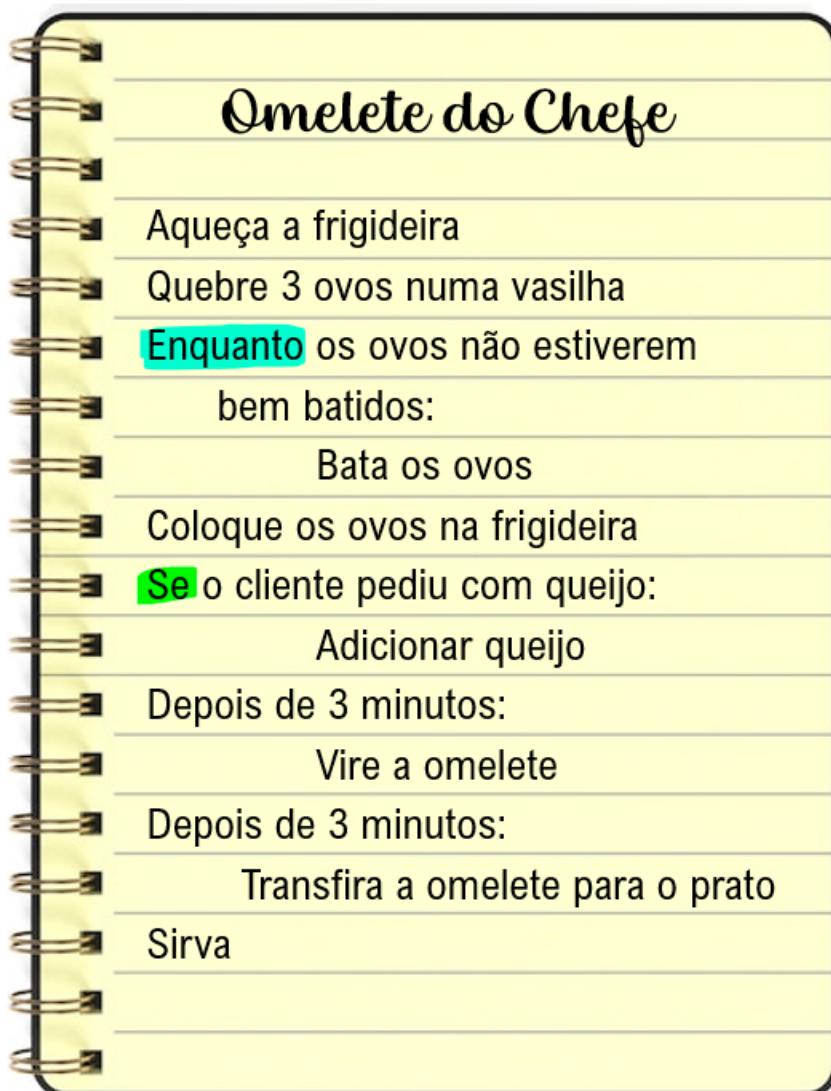
COMPUTAÇÃO 1 – AULA 6

Estrutura de repetição while

Prof. Cesar Raitz

1. Introdução

- Lembra do algoritmo da omelete?



- O **Se** sugere uma **estrutura condicional**, que vimos na Aula 4.
- Agora veremos o **Enquanto**, que é uma **estrutura de repetição**, também sujeita a uma condição.

2. Definição

- **Estrutura de repetição** é um conjunto instruções que são repetidas enquanto uma condição for satisfeita.
- Como exemplo, vamos olhar o algoritmo abaixo:

Algoritmo 1.	Terminal
1. Escrever "Bom dia!" 2. Escrever "Bom dia!" 3. Escrever "Bom dia!" 4. Escrever "Bom dia!" 5. Escrever "Bom dia!"	Bom dia! Bom dia! Bom dia! Bom dia! Bom dia!

- Nada espetacular certo? Mas é suficiente para entendermos a estrutura de repetição que será usada aqui:

Algoritmo 2.
1. Enquanto não escreveu 5 vezes: a. Escrever "Bom dia!"

- Agora temos uma **condição** para continuar repetindo.
- *Importante:* Se o computador ler a linha 1a *apenas uma vez*, escreverá "Bom dia!" *apenas uma vez*.
- Para funcionar, o computador deve ler a linha 1a mais 4 vezes!
- Vamos fazer um passo-a-passo pra entender melhor isso?

- Começamos na linha 1. Como ainda não há nada escrito, prosseguimos para 1a.

Algoritmo 3.

1. Enquanto **não escreveu 5 vezes**:
- a. Escrever "Bom dia!"
2. Escrever "Acabou :("

Terminal



- Na 1a, escrevemos "Bom dia!" no papel (ou Terminal) **mas ainda não pulamos para a linha 2**, não acabou! 😳
Voltamos para a linha 1.

Algoritmo 3.

1. Enquanto **não escreveu 5 vezes**:
- a. Escrever "Bom dia!"
2. Escrever "Acabou :("

Terminal

Bom dia!

- Estamos na 1, e "não escreveu 5 vezes" é verdade.
O que nos faz ir novamente para 1a.

Algoritmo 3.

1. Enquanto **não escreveu 5 vezes**:
- a. Escrever "Bom dia!" = **TRUE**
2. Escrever "Acabou :("

Terminal

Bom dia!

- Na linha 1a, escrevemos o segundo "Bom dia!" e voltamos para testar a condição na linha 1.

Algoritmo 3.

1. Enquanto **não escreveu 5 vezes**:
- a. Escrever "Bom dia!"
2. Escrever "Acabou :("

Terminal

Bom dia!

Bom dia!

- Já entendeu a dinâmica?

Agora, imagine que já tenha cinco "Bom dia!" escritos e voltamos para a linha 1. Desta vez "não escreveu 5 vezes" é **falso**, então vamos *sair do bloco de repetição*, seguindo para a linha 2.

Algoritmo 3.

1. Enquanto não escreveu 5 vezes:
 - a. Escrever "Bom dia!"
2. Escrever "Acabou :("

Terminal

```
Bom dia!
Bom dia!
Bom dia!
Bom dia!
Bom dia!
```

- Na linha 2, escrevemos "Acabou :(" e encerra-se o algoritmo.

Algoritmo 3.

1. Enquanto não escreveu 5 vezes:
 - a. Escrever "Bom dia!"
2. Escrever "Acabou :("

FIM

Terminal

```
Bom dia!
Bom dia!
Bom dia!
Bom dia!
Bom dia!
Acabou :(
```

- Ler um algoritmo não é como ler um texto literário!

- A sequência de instruções foi:

1, 1a, 1, 1a, 1, 1a, 1, 1a, 1, 1a, 1, 2, FIM.

1

2

3

4

5

Exercício 1. O que esse algoritmo faz?

O que aparece no Terminal? Qual é a sequência de instruções?

Algoritmo 4.	Terminal
<ol style="list-style-type: none">1. Começar com $x = 0$2. Enquanto x for menor que 3:<ol style="list-style-type: none">a. Escrever xb. Somar 1 a x3. Escrever "Fim"	

Contadores e incrementos

- Até agora, o papel das variáveis foi guardar *valores fixos* (constantes, resultados de `input` ou contas).
- No Algoritmo 4, x tem um papel diferente, pois fica mudando na linha 2a. Essa variável é chamada de **contador**.
- Os contadores *guardam o número de repetições realizadas* mas, geralmente, também são usados para alguma outra coisa.
- No jargão da computação, quando se soma um a x , estamos **incrementando**. O contrário (subtração) é chamado de **decrementar**.
- No lugar da linha 2b, você encontraria: `Incrementar x`
- Mas como fica o algoritmo acima na linguagem Python?

```
1. x = 0          # inicia o contador
2. while x < 3:
3.     print(x)    # mostra x na tela
4.     x = x + 1   # incrementa x
5. print("Fim")
```



- Entendeu? Então se liga na **sintaxe**:

```
<instruções antes>
while <condição para continuar>:
    <instruções dentro do while>
<instruções depois>
```

- <condição para continuar> é uma expressão booleana, enquanto **for True**, as <instruções dentro do while> serão repetidas.

Laços e iterações

- É mais comum chamar estruturas de repetição de **laços** (ou *loops*).
- Cada repetição no laço é uma **iteração**.

3. Alguns exemplos

Aproveitando os contadores

- Quase sempre, o contador será usado para algo mais além de contar o número de iterações. Um exemplo seria, mostrar as potências de 2 com expoente de 1 a 8:

```
1. p = 1
2. while p <= 8:
3.     valor = 2**p
4.     print(f"2^{p} = {valor}")
5.     p = p + 1
```

Exercício 2. Imprima a tabuada de 5.

Escreva um código para imprimir a tabuada de 5 usando um laço **while**. A saída no Terminal deve ser:

```
5 x 1 = 5
5 x 2 = 10
5 x 3 = 15
<...>
5 x 10 = 50
```

Generalize o exercício para imprimir qualquer tabuada.

Explorando Listas

- Suponha que você queira imprimir cada item de uma lista, enumerando-os:

```
1. tarefas = [
2.     "Jantar no bandejão",
3.     "Estudar Comp.1",
4.     "Lavar a louça"
5. ]
6. i = 0
7. while i < len(tarefas):
8.     uma_tarefa = tarefas[i]
9.     print(f"{i+1}. {uma_tarefa}")
10.    i = i + 1
```

- Suponha que você queira imprimir somente os números pares de uma lista:

```
1. numeros = [0, 3, 4, 6, 10]
2. j = 0
3. while j < len(numeros):
4.     if numeros[j] % 2 == 0:
5.         print(numeros[j])
6.     j = j + 1
```

- Você lembra que a função **sum()** soma os números de uma lista? Podemos usar uma repetição para fazer a mesma coisa, mas vamos precisar de uma variável para guardar a soma.

```
1. numeros = [0, 3, 4, 6, 10]
2. soma = 0
3. j = 0
```

```

4. while j < len(numeros):
5.     soma = soma + numeros[j]
6.     j = j + 1
7. print("A soma é", soma)

```

- Variáveis como `soma` são chamadas de **acumuladores**, pois vão acumulando o resultado que será obtido ao final do laço.
- Acumuladores podem ser usados para uma diversidade de coisas, inclusive contar caracteres:

```

1. # Conta caracteres r na string
2. frase = "O rato roeu a roupa"
3. soma_r = 0
4. i = 0
5. while i < len(frase):
6.     caractere = str.lower(frase[i])
7.     if caractere == "r":
8.         soma_r = soma_r + 1
9.     i = i + 1
10. print(f"Há {soma_r} letras r.")

```

- Note que todo caractere da frase é passado para minúscula, antes de compará-lo com "r". Assim contamos tanto o "R" como o "r".

Construindo listas

- Usando **list.append()** dentro do laço, podemos ir aumentando a lista, em cada iteração. Um exemplo comum é construir uma sequência numérica:

```
1. sequencia = [0]
2. j = 0
3. while j < 9:
4.     x = sequencia[-1]
5.     list.append(sequencia, 2*x)
6.     j = j + 1
7. print(sequencia)
```

- Outra aplicação comum é construir uma lista de dados obtidos com **input()**.

```
1. nomes = []
2. print("Digite o nome de cada aluno:")
3. j = 0
4. while j < 10:
5.     aluno = input("? ")
6.     list.append(nomes, aluno)
7.     j = j + 1
```

- Porém, listas digitadas pelo usuário nunca tem um número fixo de entradas. Ao invés de usar um contador para continuar, vamos comparar as entradas com uma string pré-definida, como "fim".

- Caso a entrada seja "fim", usamos **break** para interromper o laço:

```
1. nomes = []
2. print("Digite o nome de cada aluno")
3. print("ou fim para terminar:")
4. while True:
5.     aluno = input("? ")
6.     if aluno == "fim":
7.         break
8.     list.append(nomes, aluno)
9. print("Obrigado professor!")
```

Construindo strings

- Certos problemas pedem a *modificação de strings*.
- Só que strings são **imutáveis** e não podemos alterar a original 🤔.
- No entanto, podemos olhar os caracteres individuais com indexação
- Pense no Jogo da Forca: você escolhe algumas letras, como "d" e "a", só que a palavra escondida só tem "a", então só os "a" da palavra serão revelados. Se a palavra for "abacate", o jogo mostraria "a_a_a__" certo?
- Assim, temos que pegar a palavra original e verificar cada letra, substituindo as letras não reveladas por "_":

```
1. # Jogo da Forca
2. adivinhar = "abacate"
3. reveladas = "da"
4. mostrar = ""
5. i = 0
6. while i < len(adivinhar):
7.     letra = adivinhar[i]
8.     if letra in reveladas:
9.         mostrar = mostrar + letra
10.    else:
11.        mostrar = mostrar + "_"
12.    i = i + 1
13. print(adivinhar)
14. print(mostrar)
```

- `adivinhar` e `reveladas` têm papel de **valor fixo**.
- `mostrar` é uma variável **acumuladora** (caracteres são acrescentados a cada iteração)
- A única variável com papel de **contador** (de repetições) é a `i`.

4. Auto-atribuição

- Nesta aula, você reparou que, frequentemente, as variáveis são incrementadas com um ou outro valor?
 - mostrar = mostrar + letra
 - soma = soma + numeros[j]
 - i = i + 1
- Ou seja, ocorre uma soma (+) com o valor guardado, o resultado é guardado na própria variável com atribuição (=).
- Para encurtar o código, as linguagens de programação têm os operadores de **auto-atribuição**.
- Para soma, `x = x + 1` se torna `x += 1`. Você pode ler da seguinte forma:

Original	<code>i = i + 1</code>
Equivalente	<code>i += 1</code>
Leia-se	Soma 1 ao valor de i e guarda em i.

Original	<code>mostrar = mostrar + letra</code>
Equivalente	<code>mostrar += letra</code>
Leia-se	Soma letra à string mostrar e guarda em mostrar.

Original	<code>soma = soma + numeros[j]</code>
Equivalente	<code>soma += numeros[j]</code>
Leia-se	Soma o número da posição j da lista à variável soma e guarda em soma.

- Também há operadores de auto-atribuição para soma, multiplicação, etc.

Original	Equivalente
$x = x + y$	$x += y$
$x = x * y$	$x *= y$
$x = x - y$	$x -= y$
$x = x / y$	$x /= y$



Mas cuidado! Não misture operadores de atribuição com auto-atribuição. *Provavelmente não é o que você quer!*

$x = x + 1$ não é igual a

- $x += x + 1$
- $x + 1 += 1$

mas sim

- $x += 1$

Note também a sintaxe correta: $+=$ e não $=+$!