

---

## ✓ Fundamentos de Programação em Python

**Pontifícia Universidade Católica de Campinas**

**Prof. Dr. Denis Mayr Lima Martins**

---

### Aula 3: Estruturas de Controle e Laços de Repetição

Bem-vindo a mais uma aula de **Fundamentos de Programação em Python!**

Vamos mergulhar em uma das partes mais importantes da programação em Python: as estruturas de controle e laços de repetição!

Usamos esses conceitos para controlar o fluxo de execução do nosso programa.

#### O que veremos hoje?

- Estruturas de controle: `if` (condição), `else` (caso contrário) e `elif` (caso específico).
- Laços de repetição `for` (laço de iteração) e o `while` (laço de condição).
- Operadores de atribuição, comparação e de lógica Booleana.

#### Objetivos de Aprendizagem

Ao final desta aula, você será capaz de: Aqui está a lista resumida em **quatro bullet points**:

- Compreender estruturas condicionais (`if`, `elif`, `else`) e seu papel na tomada de decisões.
  - Explorar laços de repetição (`for` e `while` para automatizar tarefas repetitivas).
  - Diferenciar o uso de `for` e `while`, evitando loops infinitos e aplicando boas práticas.
  - Explorar expressões booleanas e operadores lógicos (`and`, `or`, `not`).
  - **Aplicar esses conceitos na resolução de problemas práticos**, como validação de entrada e contagem de valores.
- 

## ✓ Operadores de relacionais ou de comparação

Os operadores relacionais ou de comparação em Python são utilizados para comparar dois valores e retornam um resultado booleano (`True` ou `False`). Os principais operadores de comparação são: igualdade (`==`), diferença (`!=`), maior (`>`), menor (`<`), maior ou igual (`>=`) e menor ou igual (`<=`). Esses operadores são amplamente utilizados em verificações lógicas, filtragem de dados e controle de fluxo dentro do código

```
1 42 == 42
```

```
1 3 == 4
```

```
1 2 != 3
```

```
1 "dog" != "cat"
```

```
1 "hello" == "hello"
```

```
1 "hello" == "Hello" # Case sensitive
```

```
1 42 > 0
```

```
1 42 < 0
```

```
1 42 >= 40
```

```
1 42 <= 43
```

```
1 num_participantes = 100  
2  
3 num_participantes >= 50
```

## ✓ Operadores Lógicos

Em Python, existem três operadores lógicos fundamentais que nos ajudam a conectar duas expressões e obter um resultado booleano. Esses operadores são semelhantes aos operadores aritméticos, mas com uma diferença importante: em vez de retornar números ou até mesmo strings, eles apenas oferecem dois resultados possíveis - verdadeiro (True) ou falso (False). São eles:

- `and` (e lógico)
- `or` (ou lógico)
- `not` (não)

```
1 # Avalie o resultado da expressão abaixo  
2 (4 < 5) and (5 < 6)
```

Vamos ver no passo-a-passo:

1. `(4 < 5) and (5 < 6)`
2. `True and (5 < 6)`
3. `True and True`
4. `True`

```
1 (2 < 5) and (9 < 6)
```

```
1 (1 == 3) or (2 == 2)
```

```
1 idade = int(input("Digite sua idade: "))
2 eh_maior_de_idade = idade >= 18
3 print("Idade do usuário: ", idade)
4 print("Maior de idade? ", eh_maior_de_idade)
```

### Tabela Verdade para Operadores `and`, `or`:

	A	B	A and B	A or B
0	False	False	False	False
1	False	True	False	True
2	True	False	False	True
3	True	True	True	True

```
1 not (1 == 2)
```

```
1 not (1 == 1)
```

```
1 arquivo_existe = False
2 print("O arquivo existe? ", not arquivo_existe)
```

### ✓ Precedência de operadores lógicos

Nos operadores aritméticos seguimos a ordem: parênteses, potenciação, multiplicação/divisão/módulo, adição/subtração da esquerda para a direita. Enquanto isso, nos operadores lógicos seguimos a seguinte ordem:

1. Parênteses;
2. `not`;
3. `and`;
4. `or`.

### Exercício 1: avalie as expressões abaixo.

- `(10 % 2 == 0) and (2 > 1)`
- `2 + 2 == 4 and not 2 + 2 == 5 and 2 + 2 == 2 + 2`

- `2 + 2 == 4 and not 2 + 2 == 5 and 2 * 2 == 2 + 2`

```
1 # Adicione suas respostas aqui
```

```
1 # Adicione suas respostas aqui
```

## Resumo de operadores de comparação e lógicos em Python

- **Operadores de igualdade:**

- `==` (igual a)
- `!=` (não igual a)

- **Operadores de desigualdade:**

- `<` (menor que)
- `>` (maior que)
- `<=` (menor ou igual a)
- `>=` (maior ou igual a)

- **Operadores lógicos:**

- `and` (e, lógico e AND)
- `or` (ou, lógico OU)
- `not` (não, lógico NOT)

## ✓ Estruturas condicionais


O comando `if` em Python é utilizado para criar estruturas condicionais, permitindo que um bloco de código seja executado apenas se uma determinada condição for verdadeira. Ele é essencial para a tomada de decisões dentro de um programa.

Sintaxe: uma expressão **lógica** seguida de dois pontos (`:`), e o bloco de código associado deve ser **indentado** corretamente. Em alguns casos, podem ser utilizados os comandos `elif` (para testar condições adicionais) e `else` (para definir uma ação caso nenhuma das condições anteriores seja atendida).

```
1 if 10 % 2 == 0:  
2     print("10 é par")
```


```
1 hoje_e_domingo = False  
2  
3 if hoje_e_domingo == True:  
4     print('Hoje é domingo')  
5 else:  
6     print('Hoje não é domingo')
```

```
6 print('Hoje nao e domingo')
```

 **Dica:** Uma maneira melhor de escrever o código acima é omitir a parte `== True`:

```
1 hoje_e_domingo = False
2
3 if hoje_e_domingo:
4     print('Hoje é domingo')
5 else:
6     print('Hoje nao é domingo')
```

```
1 temperatura = 30
2 prob_chuva = 0.9
3 print("O dia será: ")
4
5 if temperatura >= 30:
6     print("Quente")
7
8 if prob_chuva > 0.7:
9     print("Com pancadas de chuva")
10
11 if temperatura >= 30 and prob_chuva > 0.8:
12     print("Muito abafado")
```


 **Importante:** Na prática, não importa o que seja colocado dentro da área de condições, desde que o resultado seja um booleano (`True` ou `False`).

```
1 idade = int(input("Digite sua idade: "))
2
3 if idade >= 18:
4     print("Você é maior de idade.") # Executado se a condição for verdadeira
5 else:
6     print("Você é menor de idade.") # Executado se a condição for falsa
7
8 print("Idade do usuário: ", idade)
```

**Exemplo de controle de acesso:** Neste exemplo, vamos implementar uma verificação simples de permissão de usuário. Dependendo do tipo do usuário, (administrador, técnico, ou operador normal), diferentes permissões serão exibidas.

```
1 # Definimos o perfil do usuário logado
2 perfil_usuario = 'tecnico'
3
4 # Verificamos o perfil do usuário
5 if perfil_usuario == 'admin':
6     print('Usuário: Administrador')
7     print('Você tem acesso total ao sistema de produção')
8 elif perfil_usuario == 'tecnico': # Se for técnico
```

```
9     print('Usuário: Técnico de Manutenção')
10    print('Você tem acesso para realizar manutenções')
11 elif perfil_usuario == "operador_normal": # Se for um operador normal
12     print('Usuário: Operador Normal')
13     print('Você tem acesso para operar as máquinas')
14 else: # Se não for nenhum dos casos acima
15     print('Usuário: Visitante')
16     print('Você está sem acesso')
```

 **Exercício 2:** Implemente um algoritmo que dada a idade de um nadador (lida do teclado) imprime a categoria na qual ele está:

- infantil A = 5 - 7 anos
- infantil B = 8-10 anos
- juvenil A = 11-13 anos
- juvenil B = 14-17 anos
- adulto = maiores de 18 anos

```
1 # Adicione suas respostas aqui
```

## ✓ Estruturas de laço de repetição

Os **laços de repetição** em Python são estruturas fundamentais para a automação de tarefas repetitivas dentro de um programa. Eles permitem a execução de um bloco de código múltiplas vezes, reduzindo a necessidade de comandos redundantes.

Python oferece dois tipos principais de laços: o `for`, utilizado para percorrer sequências como listas, strings e intervalos numéricos, e o `while`, que executa um bloco de código enquanto uma determinada condição lógica permanecer verdadeira.

### ✓ While loop

O comando `while` em Python é uma estrutura de repetição que executa um bloco de código enquanto uma determinada condição for verdadeira. Ele é amplamente utilizado quando o número exato de iterações não é conhecido previamente, permitindo a repetição baseada em uma expressão lógica.

```
1 contador = 10
2 while contador > 0:
3     print(contador)
4     contador = contador - 1
```

### ✓ Operadores de atribuição

Os **operadores de atribuição** permitem realizar operações matemáticas e atualizar variáveis de maneira mais concisa, como `+=`, `-=`, `*=`, `/=`, `//=`, `%=` e `**=`. Esses operadores são amplamente empregados em laços de repetição e cálculos acumulativos, tornando a escrita do programa mais sucinta.

```
1 contador = 10
2 while contador > 0:
3     print(contador)
4     contador -= 1
```

```
1 # Novo exemplo
2 continuar = True # Variável booleana para controlar o loop
3
4 # Loop que continua enquanto 'continuar' for True
5 while continuar:
6     resposta = input("Deseja continuar? Digite s (para sim) ou n(para nao):")
7
8     if resposta == "n":
9         continuar = False # A condição de parada é atingida, o loop vai par
10        print("Loop encerrado.")
11    elif resposta == "s":
12        print("Você escolheu continuar.")
13    else:
14        print("Não reconheci sua resposta. Tente novamente.")
```

Double-click (or enter) to edit

## ✓ Loop infinito

Ao utilizar o comando **while** em Python, é fundamental adotar boas práticas para evitar a ocorrência de **loops infinitos**, que podem comprometer a execução do programa, causando a sua paralisação. Um **loop infinito** ocorre quando a condição de parada nunca se torna **False**, resultando em uma execução contínua sem término.

Para evitar esse problema, é essencial garantir que a variável de controle seja devidamente atualizada dentro do bloco de repetição. Além disso, é recomendável estabelecer critérios de saída adicionais, como contadores auxiliares ou verificações condicionais, assegurando que o laço não permaneça em execução indefinidamente.

```
1 # Cuidado com esse código! Loop infinito
2 while True:
3     print("Loop")
```

```
1 continuar = True # Variável booleana para controlar o loop
2 contador = 10
```

```

3
4 # Loop que continua enquanto 'continuar' for True ou até a 11a interação
5 while continuar and contador >= 0:
6     resposta = input("Deseja continuar? Digite s (para sim) ou n(para nao):
7
8     if resposta == "n":
9         continuar = False # A condição de parada é atingida, o loop vai par
10        contador = 0 # Zera o contador
11        print("Loop encerrado pelo usuário.")
12    elif resposta == "s":
13        print("Você escolheu continuar.")
14    else:
15        print("Não reconheci sua resposta. Tente novamente.")
16
17    print("Tentativas restantes: ", contador)
18    contador -= 1 # Decrementa o contador
19
20
21 print("Saiu do loop")
22

```

## ✓ Comando break

O comando **break** é uma instrução de controle de fluxo utilizada para **interromper a execução de um laço de repetição**, encerrando o loop imediatamente, independentemente da condição de parada definida.

Seu uso é particularmente relevante em situações onde um critério de saída específico deve ser atendido, permitindo maior flexibilidade na estrutura do programa.

```

1 # Novas palavras-chave: break
2 count = 10
3 while count > 0:
4     count = count - 1
5     print("O valor é", count)
6     if count == 5:
7         break

```

```

1 # Para fazermos juntos
2 valor = 2 # Mostrar no quadro: count / condição / operacoes
3
4 while valor <= 12:
5     print('O número par é ', valor)
6     valor = valor + 2
7     if valor == 10:
8         break

```

```

1 # Remodelando o exemplo previamente visto
2
3 contador = 10

```



```

4
5 # Loop que continua enquanto 'continuar' for True ou até a 10a interação
6 while True:
7     print("Tentativas restantes: ", contador)
8
9     if contador <= 0:
10        break
11
12    contador -= 1 # Decrementa o contador
13
14    resposta = input("Deseja continuar? Digite s (para sim) ou n(para nao):
15
16    if resposta == "n":
17        print("Loop encerrado pelo usuário.")
18        break
19    elif resposta == "s":
20        print("Você escolheu continuar.")
21    else:
22        print("Não reconheci sua resposta. Tente novamente.")
23
24
25 print("Saiu do loop")
26

```

## ✓ Comando `continue`

O comando `continue` é uma instrução de controle de fluxo utilizada dentro de laços de repetição, como o `while`, para **pular a iteração atual** e continuar a execução do loop a partir da próxima iteração. Diferentemente do comando `break`, que interrompe completamente a repetição, o `continue` permite que o laço prossiga sem executar o restante do código dentro da iteração corrente.

Esse comando é útil em situações onde certas condições precisam ser ignoradas temporariamente, como na filtragem de dados ou no controle de validações dentro de um loop. No entanto, seu uso deve ser planejado com cautela para garantir a clareza e previsibilidade do código.

```

1 # Usando o comando continue
2 contador = 0
3 while contador < 10:
4     contador += 1
5     if contador % 2 == 0:
6         continue # Pula números pares e continua a próxima iteração
7     print("Número ímpar:", contador)

```

## ✓ Função `len()`

A função `len()` é utilizada para determinar o comprimento de uma sequência, como

**strings, listas e tuplas.** Quando aplicada a uma **string**, ela retorna o número total de caracteres, incluindo espaços e pontuações. Em conjunto com estruturas de repetição, a função `len()` permite percorrer cada caractere da string de forma controlada, possibilitando a realização de diversas operações, como contagem de letras e filtragem de elementos específicos.

```
1 palavra = "Python"
2 len(palavra) # Número de caracteres na string
```

```
1 palavra = "papaia"
2 indice = 0
3
4 while indice < len(palavra):
5     letra = palavra[indice]
6     indice += 1
7
8     if letra == "a":
9         continue
10
11     print(letra)
```

### Exercício 3:

1. Crie um programa que pergunte ao usuário por um número e, em seguida, peça se ele deseja multiplicar esse número por uma sequência de números inteiros a partir de 1. Por exemplo, se o usuário digita 5, o programa deve imprimir  $5 + 10 + 15 + 20$ .
2. Crie um programa que solicita um número inteiro positivo ao usuário e exibe esse número invertido. Dicas:
  - Utilize um loop `while` para extrair os dígitos do número.
  - Utilize operações matemáticas (`%` e `//`) para manipular os dígitos.

## ✓ For loop

O comando `for` é uma estrutura de repetição amplamente utilizada para percorrer elementos de uma sequência, como **listas, strings, tuplas e intervalos numéricos**. Diferente do `while`, que depende de uma condição lógica para execução, o `for` itera sobre cada elemento da sequência de forma **controlada e previsível**, tornando-o ideal para manipulação de conjuntos de dados.

Esse laço é amplamente aplicado em algoritmos que envolvem **iterações finitas**, como processamento de textos, varredura de listas e cálculos computacionais.

```
1 for letra in "Python":
```

```
2     print(letra)
```

```
1 for i in [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]:  
2     print(i)
```

```
1 for i in [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]:  
2     if i % 2 == 0: # Apenas números pares  
3         print(i)
```

## ▼ Função `range()`

A função `range()` é utilizada em conjunto com o laço `for` para gerar sequências numéricas. Sua principal finalidade é permitir a iteração controlada sobre um conjunto de valores inteiros, sem a necessidade de criar listas manualmente.

A função aceita até três parâmetros: **início**, **fim** (não inclusivo) e **passo**, proporcionando flexibilidade na definição dos intervalos numéricos. Essa característica torna o `range()` essencial para a construção de laços que exigem um número específico de iterações.

```
1 for i in range(0, 11):  
2     print(i)
```

```
1 # Itera de 0 até 10, com passo 2  
2 for i in range(0, 11, 2): # range(start, stop, step)  
3     print(i)
```

```
1 soma = 0  
2 for numero in range(11): # 0 zero pode ser omitido no início da função range  
3     soma += numero  
4     print(numero)  
5 print("Soma: ", soma)
```

```
1 # Imprimir números de 0 a 20 usando um loop  
2 # ...
```

```
1 # Gere os números de 10 até 1 (contagem regressiva).  
2 for i in range(10, 0, -1):  
3     print(i)
```

```
1 # Números de 0 a 20 com passo 5  
2 # ...
```

```
1 # Fatorial do número 5  
2 #...
```



## Exercícios para Praticar:

**Nível Fácil:** Escreva um programa que solicita ao usuário uma palavra e conta quantas vogais e consoantes ela possui.

### Requisitos:

- Utilize um loop `for` para percorrer a string.
- Utilize condições (`if`) para verificar se um caractere é uma vogal ou consoante.
- Considere apenas letras (ignore espaços, números e caracteres especiais).

**Nível Médio:** Crie um programa que solicita um número inteiro positivo ao usuário e calcula a **soma de seus dígitos**.

### Requisitos:

- Utilize um **loop while** para extrair os dígitos do número.
- Utilize um operador matemático para obter o último dígito (`%`).

### Exemplo de Entrada e Saída:

```
Digite um número: 1234
A soma dos dígitos é: 10 # (1 + 2 + 3 + 4)
```

```
1 # Adicione suas respostas aqui
```

## Conclusão

Parabéns por concluir esta aula!

## O que aprendemos hoje?

Nesta aula, exploramos conceitos fundamentais para controlar o fluxo de um programa em **Python**:

- Como utilizar **estruturas condicionais** (`if`, `elif`, `else`) para tomar decisões.
- O uso de **laços de repetição** (`for` e `while`) para executar comandos de forma automática.
- A diferença entre **iteração controlada** (`for`) e **iteração condicional** (`while`).

## Próximos Passos

- Resolva os problemas na seção "Exercícios para Praticar".
- Caso tenha dúvidas, revise os exemplos e experimente modificá-los. A programação se aprende na prática!

Conteúdo adicional

 [Documentação Oficial do Python – Estruturas de Controle](#)

Parabéns pela dedicação! Nos vemos na próxima aula!