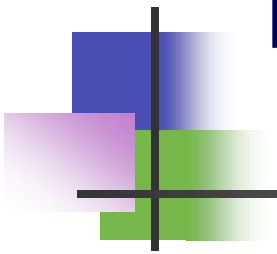
Escola do Futuro do Estado de Goiás – EFG – Paulo Renato Souza

Professor: Roberto Carlos

Python – Revisão



- Em Python, os tipos primitivos, também conhecidos como tipos básicos, são os blocos de construção fundamentais para armazenar e manipular dados. São cinco tipos:

1. Números inteiros (int):

- Representam números sem casas decimais, como 1, 2, -3, 100.
- São usados para contar, calcular e armazenar valores discretos.
- Exemplos: idade de uma pessoa, quantidade de produtos em estoque.

1. Números inteiros (int):

- Declaração simples de um número inteiro:

```
num_inteiro = 10
```

- Operações aritméticas com inteiros:

```
a = 5
b = 3
soma = a + b
subtracao = a - b
multiplicacao = a * b
divisao = a / b
```

1. Números inteiros (int):

- Conversão de string para inteiro:

```
numero_string = "20"
numero_inteiro = int(numero_string)
```

- Operações com entrada do usuário (input) convertida para inteiro:

```
entrada = input("Digite um número inteiro: ")
numero = int(entrada)
```

1. Números inteiros (int):

- Uso de inteiros em estruturas de controle:

```
idade = 25
if idade >= 18:
    print("Você é maior de idade.")
else:
    print("Você é menor de idade.")
```

- Iterando com inteiros em loops:

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

1. Números inteiros (int):

- Indexação de listas com inteiros:

```
lista = [10, 20, 30, 40]
print(lista[2]) # imprime o terceiro elemento da lista, que é 30
```

2. Números de ponto flutuante (float):

- Representam números com casas decimais, como 3.14, 5.50, -8.9.
- São usados para medidas, valores monetários e cálculos científicos.
- Exemplos: temperatura, preço de um produto, área de um terreno.

2. Números de ponto flutuante (float):

- Declaração simples de um número de ponto flutuante:

```
num_float = 3.14
```

- Operações aritméticas com números de ponto flutuante:

```
a = 3.5
b = 1.2
soma = a + b
subtracao = a - b
multiplicacao = a * b
divisao = a / b
```

2. Números de ponto flutuante (float):

- Conversão de string para ponto flutuante:

```
numero_string = "5.75"
numero_float = float(numero_string)
```

- Operações com entrada do usuário (input) convertida para ponto flutuante:

```
entrada = input("Digite um número de ponto flutuante: ")
numero = float(entrada)
```

2. Números de ponto flutuante (float):

- Uso de números de ponto flutuante em estruturas de controle:

```
altura = 1.75
if altura >= 1.8:
    print("Você é alto.")
else:
    print("Você não é tão alto.")
```

- Iterando com números de ponto flutuante em loops:

```
for i in range(1, 5):
    print(1.5 * i)
```

2. Números de ponto flutuante (float):

- Uso de números de ponto flutuante em cálculos matemáticos avançados:

```
import math


raio = 2.5
area_circulo = math.pi * raio ** 2
```

3. Strings (str):

- São sequências de caracteres, como letras, números, símbolos e espaços em branco.
- São usadas para armazenar textos, nomes, frases e mensagens.
- Exemplos: nome de um cliente, título de um livro, conteúdo de um email.

3. Strings (str):

- Declaração simples de uma string:

```
nome = "Python"
```

- Concatenação de strings:

```
saudacao = "Olá, " + nome + "!"
```

3. Strings (str):

- Acesso aos caracteres de uma string:

```
primeira_letra = nome[0]
```

- Iterando sobre os caracteres de uma string:

```
for letra in nome:
    print(letra)
```

3. Strings (str):

- Métodos de string:

```
mensagem = "python é uma linguagem de programação"
print(mensagem.upper())  # Converte para maiúsculas
print(mensagem.capitalize())  # Capitaliza a primeira letra
print(mensagem.split())  # Divide a string em uma lista de palavras
```

- Formatação de strings:

```
idade = 25
altura = 1.75
frase = "Eu tenho {} anos e minha altura é {:.2f} metros".format(idade, altura)
```

3. Strings (str):

- Fatiamento de strings:

```
sobrenome = "Doe"
primeiro_nome = "João"
nome_completo = primeiro_nome + " " + sobrenome
print(nome_completo[0:4])  # Imprime "João"
```

- Verificação de substrings:

```
texto = "Python é uma linguagem de programação"
if "Python" in texto:
    print("Python encontrado!")
```

3. Strings (str):

- Escapando caracteres especiais:

```
texto_com_aspas = "Python é uma linguagem de \"scripting\""
```

Exercício:

```python
def main():
    # Solicita ao usuário que insira seu nome, idade e altura
    nome = input("Digite seu nome: ")
    idade = int(input("Digite sua idade: "))
    altura = float(input("Digite sua altura em metros: "))

    # Exibe as informações de volta para o usuário
    print("\nInformações inseridas:")
    print("Nome:", nome)
    print("Idade:", idade, "anos")
    print("Altura:", altura, "metros")

if __name__ == "__main__":
    main()
```

4. Booleanos (bool):

- Representam valores lógicos, **True (verdadeiro) ou False (falso)**.
- São usados para tomar decisões, verificar condições e realizar comparações.
- Exemplos: um usuário está logado no sistema? Uma compra foi aprovada?

4. Booleanos (bool):

- Declaração de booleanos simples:

```
verdadeiro = True
falso = False
```

- Operações de comparação que retornam booleanos

```
a = 5
b = 3
resultado = a > b  # Isso resultará em True
```

4. Booleanos (bool):

- Uso de booleanos em estruturas de controle

```
idade = 18
if idade >= 18:
    print("Você é maior de idade.")
else:
    print("Você é menor de idade.")
```

- Operações lógicas com booleanos

```
tem_carteira_de_motorista = True
tem_carro = False
pode_dirigir = tem_carteira_de_motorista and tem_carro  # Isso resultará em False
```

4. Booleanos (bool):

- Verificação de pertencimento em sequências (retorna booleano)

```
lista = [1, 2, 3, 4, 5]
existe_numero_3 = 3 in lista  # Isso resultará em True
```

- Uso de operadores de comparação em cadeia (retorna booleano)

```
x = 10
resultado = 1 < x < 20  # Isso resultará em True
```

4. Booleanos (bool):

- Uso de funções que retornam booleanos

```
texto = "Python é uma linguagem de programação"

contem_palavra_python = texto.startswith("Python")  # Isso resultará em True
```

4. Exercício

- Verificação de elegibilidade para votar
- Escreva um programa que solicite a idade de um usuário e verifique se ele é elegível para votar. O programa deve imprimir uma mensagem indicando se o usuário pode ou não votar com base na idade inserida.

4. Exercício

```python
def verificar_elegibilidade(idade):
    if idade >= 18:
        return True
    else:
        return False

def main():
    idade = int(input("Digite sua idade: "))
    elegivel = verificar_elegibilidade(idade)

    if elegivel:
        print("Você é elegível para votar.")
    else:
        print("Você não é elegível para votar.")

if __name__ == "__main__":
    main()
```

5. Complexos (complex):

- Representam números compostos por uma parte real e uma parte imaginária. (j)
- São usados em matemática avançada, física e engenharia.
- Exemplos: raízes quadradas de números negativos, correntes elétricas.

5. Complexos (complex):

- Declaração simples de um número complexo

```
num_complexo = 3 + 2j
```

- Operações aritméticas com números complexos

```
a = 1 + 2j
b = 2 - 1j
soma = a + b
subtracao = a - b
multiplicacao = a * b
divisao = a / b
```

5. Complexos (complex):

- Acesso às partes real e imaginária de um número complexo

```
num = 4 + 3j
parte_real = num.real
parte_imaginaria = num.imag
```

- Conversão de string para número complexo

```
num_string = "2+5j"
num_complexo = complex(num_string)
```

5. Complexos (complex):

- Operações com entrada do usuário (input) convertida para número complexo

```
entrada = input("Digite um número complexo no formato a+bj: ")
num_complexo = complex(entrada)
```

5. Complexos (complex):

- Uso de números complexos em estruturas de controle:

```
z = 2 + 3j
if z.real > 0:
    print("Parte real positiva.")
else:
    print("Parte real não é positiva.")
```

5. Complexos (complex):

- Iterando com números complexos em loops:

```
for i in range(3):
    num = complex(i, i + 1)
    print(num)
```

5. Exercícios

- Exercício: Cálculo da raiz quadrada de um número complexo

5. Exercícios

```python
def calcular_raiz_quadrada_complexa(num_complexo):
    raiz_quadrada = num_complexo ** 0.5
    return raiz_quadrada

def main():
    # Solicita ao usuário que insira as partes real e imaginária do número complexo
    parte_real = float(input("Digite a parte real do número complexo: "))
    parte_imaginaria = float(input("Digite a parte imaginária do número complexo: "))

    # Cria o número complexo com as partes real e imaginária fornecidas
    num_complexo = complex(parte_real, parte_imaginaria)

    # Calcula a raiz quadrada do número complexo
    raiz_quadrada = calcular_raiz_quadrada_complexa(num_complexo)

    # Exibe o resultado
    print("A raiz quadrada do número complexo", num_complexo, "é", raiz_quadrada)

if __name__ == "__main__":
    main()
```

6. Operadores em Python

- Os operadores em Python são símbolos que permitem realizar operações em valores, variáveis e expressões. Eles são essenciais para a construção de programas e a execução de tarefas complexas.
- Python possui diversos tipos de operadores, cada um com sua função específica:

6.1 Tipos de Operadores:

- **1. Operadores Aritméticos:**
- **Adição (+)**: Soma dois valores. Exemplo: 2 + 3 = 5.
- **Subtração (-)**: Subtrai um valor do outro. Exemplo: 5 - 2 = 3.
- **Multiplicação (*)**: Multiplica dois valores. Exemplo: 2 * 3 = 6.

6.1 Tipos de Operadores:

- **1. Operadores Aritméticos:**
- **Divisão (/):** Divide um valor pelo outro. Exemplo: 6 / 2 = 3.
- **Potenciação ():**** Eleva um valor a outro. Exemplo: 2 ** 3 = 8.
- **Resto da divisão (%):** Retorna o resto da divisão de um valor pelo outro. Exemplo: 10 % 3 = 1.
- **Divisão inteira (//):** Realiza a divisão inteira de um valor pelo outro. Exemplo: 10 // 3 = 3.

6.1 Tipos de Operadores:

- **1. Operadores Aritméticos:**
- **Divisão (/):** Divide um valor pelo outro. Exemplo: 6 / 2 = 3.
- **Potenciação ():**** Eleva um valor a outro. Exemplo: 2 ** 3 = 8.
- **Resto da divisão (%):** Retorna o resto da divisão de um valor pelo outro. Exemplo: 10 % 3 = 1.
- **Divisão inteira (//):** Realiza a divisão inteira de um valor pelo outro. Exemplo: 10 // 3 = 3.

6.1 Exercícios: Calculadora

```python
# Entrada de dados
num1 = float(input("Digite o primeiro número: "))
num2 = float(input("Digite o segundo número: "))

# Operações matemáticas
soma = num1 + num2
subtracao = num1 - num2
multiplicacao = num1 * num2
divisao = num1 / num2
divisao_inteira = num1 // num2
modulo = num1 % num2
exponenciacao = num1 ** num2

# Saída de dados
print(f"Soma: {soma}")
print(f"Subtração: {subtracao}")
print(f"Multiplicação: {multiplicacao}")
print(f"Divisão: {divisao}")
print(f"Divisão inteira: {divisao_inteira}")
print(f"Módulo: {modulo}")
print(f"Exponenciação: {exponenciacao}")
```

6. 2. Operadores de Comparação:

- **Igual a (==):** Verifica se dois valores são iguais. Exemplo: 2 == 2 retorna True.
- **Diferente de (!=):** Verifica se dois valores são diferentes. Exemplo: 2 != 3 retorna True.
- **Maior que (>):** Verifica se um valor é maior que outro. Exemplo: 3 > 2 retorna True.
- **Menor que (<):** Verifica se um valor é menor que outro. Exemplo: 2 < 3 retorna True.

6. 2. Operadores de Comparação:

- **Maior ou igual a (>=):** Verifica se um valor é maior ou igual a outro. Exemplo: 3 >= 3 retorna True.
- **Menor ou igual a (<=):** Verifica se um valor é menor ou igual a outro. Exemplo: 2 <= 3 retorna True.

6. 2. Exercício: Comparação de Números:

```python
# Entrada de dados
num1 = float(input("Digite o primeiro número: "))
num2 = float(input("Digite o segundo número: "))

# Operadores de comparação
igual = num1 == num2
diferente = num1 != num2
maior = num1 > num2
menor = num1 < num2
maior_ou_igual = num1 >= num2
menor_ou_igual = num1 <= num2

# Saída de dados
print(f"{num1} é igual a {num2}: {igual}")
print(f"{num1} é diferente de {num2}: {diferente}")
print(f"{num1} é maior que {num2}: {maior}")
print(f"{num1} é menor que {num2}: {menor}")
print(f"{num1} é maior ou igual a {num2}: {maior_ou_igual}")
print(f"{num1} é menor ou igual a {num2}: {menor_ou_igual}")
```

6. 3. Operadores Lógicos:

- **E (and):** Retorna True, se ambas as expressões forem verdadeiras. Exemplo: (2 > 1) and (2 < 3) retorna True.
- **Ou (or):** Retorna True, se pelo menos uma das expressões for verdadeira. Exemplo: (2 > 1) or (2 > 3) retorna True.
- **Não (not):** Inverte o valor da expressão. Exemplo: not (2 > 3) retorna True.

6. 3. Exercício: Verificação de Acesso:

```python
# Entrada de dados
idade = int(input("Digite sua idade: "))
eh_aluno = bool(input("Você é aluno? (sim/não) "))
tem_curso_superior = bool(input("Você tem curso superior? (sim/não) "))

# Operadores lógicos
acesso_permitido = (idade >= 18) and (eh_aluno or tem_curso_superior)

# Saída de dados
if acesso_permitido:
    print("Acesso permitido!")
else:
    print("Acesso negado.")
```

6. 4. Operadores de Atribuição:

- **Atribuição (=):** Atribui um valor a uma variável. Exemplo: x = 5.
- **Atribuição de soma (+=):** Soma um valor a uma variável. Exemplo: x += 2.
- **Atribuição de subtração (-=):** Subtrai um valor de uma variável. Exemplo: x -= 2.
- **Atribuição de multiplicação (*=):** Multiplica um valor por uma variável. Exemplo: x *= 2.
- **Atribuição de divisão (/=):** Divide um valor por uma variável. Exemplo: x /= 2.

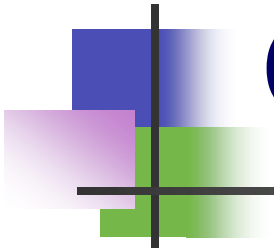
6. 4. Exercício: Calculadora de Salário

```python
# Entrada de dados
salario_bruto = float(input("Digite o salário bruto: "))
numero_dependentes = int(input("Digite o número de dependentes: "))

# Operadores de atribuição
desconto_inss = salario_bruto * 0.1
desconto_ir = salario_bruto * 0.2
valor_dependente = 100 * numero_dependentes
deducoes = desconto_inss + desconto_ir + valor_dependente
salario_liquido = salario_bruto - deducoes

# Saída de dados
print(f"Salário bruto: R${salario_bruto:.2f}")
print(f"Desconto INSS: R${desconto_inss:.2f}")
print(f"Desconto IR: R${desconto_ir:.2f}")
print(f"Valor dependentes: R${valor_dependente:.2f}")
print(f"Salário líquido: R${salario_liquido:.2f}")
```

6. 5. Operadores de Identidade:

- **É (is):** Verifica se dois objetos são o mesmo objeto na memória. Exemplo: x is y ***retorna True*** se x e y são o mesmo objeto.
- **Não é (is not):** Verifica se dois objetos não são o mesmo objeto na memória. Exemplo: x is not y ***retorna True*** se x e y não são o mesmo objeto.

6. 5. Exercício: Comparação de Objetos:



Vou ficar devendo esse!

6. 6. Operadores de Membros:

- **Em (in):** Verifica se um valor está presente em uma sequência. Exemplo: 2 in [1, 2, 3] retorna True.
- **Não está em (not in):** Verifica se um valor não está presente em uma sequência. Exemplo: 2 not in [1, 2, 3] retorna False.

6. 6. Exercício: Comparação de Objetos:

Vou ficar devendo esse também!

6. 7. Operadores Ternários:

- **Condicional (condicao if resultado_verdadeiro else resultado_falso):**"expressão condicional", permite realizar uma operação condicional em uma única linha de código. Ele é composto por três partes:

1. Condição: Uma expressão que pode ser avaliada como True ou False.

2. Resultado Verdadeiro: O valor a ser retornado se a condição for True.

3. Resultado Falso: O valor a ser retornado se a condição for False.

6. 7. Operadores Ternários:

```
resultado = condicao if expressao_verdadeira else expressao_falsa
```

```
idade = 18

maior_de_idade = "Maior de idade" if idade >= 18 else "Menor de idade"

print(maior_de_idade)  # Saída: "Maior de idade"
```

6. 7. Operadores Ternários:

- **Vantagens:**
- **Concisão:** Permite escrever código mais conciso e legível.
- **Eficiência:** Evita a necessidade de usar instruções if e else em casos simples.

6. 7. Operadores Ternários:

- **Desvantagens:**
- **Complexidade:** Pode ser difícil de ler e entender em casos mais complexos.
- **Aninhamento:** O aninhamento de operadores ternários pode tornar o código ilegível.

6. 7. Operadores Ternários:

- **Dicas para usar operadores ternários:**
- Use-os para operações condicionais simples.
- Evite aninhar operadores ternários.
- Use comentários para explicar o código se necessário.

6. 7. Operadores Ternários:

- **Exemplos de uso:**
- Verificar se um número é par ou ímpar.
- Definir o valor padrão de uma variável.
- Atribuir valores diferentes a uma variável com base em uma condição.
- Operadores ternários são uma ferramenta poderosa que pode tornar seu código Python mais conciso e eficiente. No entanto, é importante usá-los com cuidado para evitar que o código se torne ilegível.

6. 7. Operadores Ternários:

- **Exercícios: Verificação de Paridade:**

```
numero = int(input("Digite um número: "))

paridade = "Par" if numero % 2 == 0 else "Ímpar"

print(f"O número {numero} é {paridade}.")
```

6. 7. Operadores Ternários:

- **Exercícios: Cálculo de Maior Valor**

```
num1 = int(input("Digite o primeiro número: "))
num2 = int(input("Digite o segundo número: "))

maior = num1 if num1 > num2 else num2

print(f"O maior valor entre {num1} e {num2} é {maior}.")
```

6. 7. Operadores Ternários:

- **Exercícios: Atribuição de Faixa Etária:**

```
idade = int(input("Digite sua idade: "))

faixa_etaria = "Criança" if idade < 12 else "Adolescente" if idade < 18 else "Adulto"

print(f"Com {idade} anos, você é considerado(a) {faixa_etaria}.")
```

6. 7. Operadores Ternários:

- **Exercícios: Cálculo de Desconto:**

```
valor_produto = float(input("Digite o valor do produto: "))

desconto = 0.1 if valor_produto > 100 else 0.05

valor_final = valor_produto - (valor_produto * desconto)

print(f"O valor final do produto com desconto é R${valor_final:.2f}.")
```

6. 7. Operadores Ternários:

- **Exercícios: Verificação de Aprovação:**

```
nota1 = float(input("Digite a primeira nota: "))
nota2 = float(input("Digite a segunda nota: "))

media = (nota1 + nota2) / 2

resultado = "Aprovado(a)" if media >= 6 else "Reprovado(a)"

print(f"Com média {media:.2f}, você está {resultado}.")
```

6. 7. Operadores Ternários:

- **Exercícios: Classificação de IMC**

```
peso = float(input("Digite seu peso em kg: "))
altura = float(input("Digite sua altura em metros: "))

imc = peso / (altura ** 2)

classificacao = "Abaixo do Peso" if imc < 18.5 else "Normal" if imc < 25 else "Sobrepeso" if imc < 30 else "Obesidade"

print(f"Com IMC de {imc:.2f}, você está classificado(a) como {classificacao}.")
```

6. 7. Operadores Ternários:

- **Exercícios: Classificação de IMC**

```
escolha_jogador1 = input("Digite sua escolha (pedra, papel ou tesoura): ")
escolha_jogador2 = input("Digite a escolha do adversário (pedra, papel ou tesoura): ")

resultado = "Empate"

if escolha_jogador1 == "pedra":
    resultado = "Jogador 1 vence!" if escolha_jogador2 == "tesoura" else "Jogador 2 vence!"
elif escolha_jogador1 == "papel":
    resultado = "Jogador 1 vence!" if escolha_jogador2 == "pedra" else "Jogador 2 vence!"
elif escolha_jogador1 == "tesoura":
    resultado = "Jogador 1 vence!" if escolha_jogador2 == "papel" else "Jogador 2 vence!"

print(f"Resultado: {resultado}")
```

6. 7. Operadores Ternários:

- **Lembre-se:**
- Os operadores ternários são uma ferramenta poderosa para simplificar seu código Python.
- Use-os com cuidado para evitar que o código se torne ilegível.
- Pratique com os exercícios acima para aprimorar seu domínio dos operadores ternários.

6. 7. Operadores Ternários:

- **Exercícios Bônus:**
- Escreva um programa que verifica se um ano é bissexto.
- Escreva um programa que calcula a área de um triângulo, considerando os diferentes tipos de triângulos.
- Escreva um programa que converte uma temperatura de Celsius para Fahrenheit e vice-versa.

7. Estruturas de Controle:

- As estruturas de controle são elementos fundamentais em Python, permitindo que você dite o fluxo de execução do seu código, definindo quais partes serão executadas e em qual ordem.
- Elas se dividem em dois tipos principais:

7. Estruturas de Controle:

Estruturas Condicionais:

- **if:** Permite executar um bloco de código se uma condição for verdadeira.
- **elif:** Usado em conjunto com if para verificar outras condições caso a primeira seja falsa.
- **else:** Executa um bloco de código caso todas as condições anteriores sejam falsas.

7. Estruturas de Controle:

Estruturas Condicionais:

Exemplo:

```
numero = 10

if numero > 0:
    print("O número é positivo")
elif numero == 0:
    print("O número é zero")
else:
    print("O número é negativo")
```

7. Estruturas de Controle:

Estruturas de Repetição:

- **while:** Executa um bloco de código enquanto uma condição for verdadeira.
- **for:** Executa um bloco de código um número determinado de vezes ou para cada item em uma sequência.

7. Estruturas de Controle:

Outras Estruturas de Controle:

- **break:** Sai de um loop while ou for antes do final.
- **continue:** Pula para a próxima iteração de um loop while ou for.
- **pass:** Faz nada, útil para manter a estrutura do código.

7.1. if (Se):

- A instrução if é uma estrutura condicional que permite executar um bloco de código se uma condição específica for verdadeira.
- É uma ferramenta fundamental para controlar o fluxo de execução do seu programa e tomar decisões com base em diferentes cenários.

7.1. if (Se):

```
numero = 10

if numero > 0:
    print("O número é positivo")
```

Neste exemplo, a condição numero > 0 é verdadeira, então o bloco de código dentro do if será executado, imprimindo a mensagem "O número é positivo".

7.1. if (Se):

Operadores de comparação:

A condição dentro do if pode ser formada por diversos operadores de comparação, como:

- **==**: igual a
- **!=**: diferente de
- **<**: menor que
- **<=**: menor ou igual a
- **>**: maior que
- **>=**: maior ou igual a

7.1. if (Se):

Combinando operadores:

É possível combinar operadores de comparação usando **and** e **or** para criar condições mais complexas.

```
idade = 20
nacionalidade = "brasileira"

if idade >= 18 and nacionalidade == "brasileira":
    print("Você pode votar")
```

7.1. if (Se):

Estruturas aninhadas:

Você pode usar if dentro de outro if para criar estruturas aninhadas e lidar com diferentes cenários.

```
nota = 8

if nota >= 7:
    print("Aprovado")
    if nota >= 9:
        print("Com distinção!")
else:
    print("Reprovado")
```

7.2. elif (“E esse pode ser?”):

ELIF é uma palavra-chave em Python que significa "else if" **(ELSE + IF).** É usada em conjunto com a instrução if para verificar condições adicionais após a condição principal

```
if condição1:
    # Bloco de código a ser executado se a condição1 for verdadeira
elif condição2:
    # Bloco de código a ser executado se a condição1 for falsa e a condição2 for verdadeira
...
else:
    # Bloco de código a ser executado se todas as condições anteriores forem falsas
```

7.2. elif (“E esse pode ser?”):

Exemplo:

```
numero = 10

if numero > 0:
    print("O número é positivo")
elif numero == 0:
    print("O número é zero")
else:
    print("O número é negativo")
```

7.2. elif (“E esse pode ser?”):

Vantagens do elif:

- Permite verificar várias condições em sequência.
- Evita a repetição de código e torna o código mais conciso.
- Melhora a legibilidade do código.

7.3. else (se não):

É usada em conjunto com a instrução if e/ou elif para fornecer um bloco de código alternativo caso todas as condições anteriores sejam falsas. O else é opcional.

7.3. else (se não):

É usada em conjunto com a instrução if e/ou elif para fornecer um bloco de código alternativo caso todas as condições anteriores sejam falsas. O else é opcional.

```
if condição1:
    # Bloco de código a ser executado se a condição1 for verdadeira
elif condição2:
    # Bloco de código a ser executado se a condição1 for falsa e a condição2 for verdadeira
...
else:
    # Bloco de código a ser executado se todas as condições anteriores forem falsas
```

7.3. else (se não):

Exemplo:

```
numero = 10

if numero > 0:
    print("O número é positivo")
elif numero == 0:
    print("O número é zero")
else:
    print("O número é negativo")
```

7.3. else (se não):

Vantagens do else:

- Permite fornecer um comportamento alternativo caso todas as condições anteriores sejam falsas.
- Torna o código mais completo e robusto.
- Melhora a legibilidade do código.

7.3. else (se não):

Exemplos de uso do else:

- Validar a entrada de um usuário.
- Exibir uma mensagem de erro caso uma operação falhe.
- Executar um conjunto de ações padrão caso nenhuma condição específica seja atendida.

7.4. while (enquanto):

É usada para criar um loop que repete um bloco de código enquanto uma condição específica for verdadeira.

7.4. while (enquanto):

Vantagens do while:

- Permite executar um bloco de código várias vezes sem precisar repetir o código.
- É útil para tarefas que precisam ser repetidas um número indeterminado de vezes.
- Torna o código mais conciso e eficiente.

7.4. while (enquanto):

Exemplo:

```
numero = 1

while numero <= 10:
    print(numero)
    numero += 1
```

7.4. while (enquanto):

- A condição dentro do while deve ser sempre verificada. Se for sempre verdadeira, o loop será executado infinitamente.
- Use um incrementador ou decrementador dentro do loop para atualizar a variável da condição.
- É possível usar o break para sair do loop antes da condição se tornar falsa.
- As estruturas de controle podem ser aninhadas para criar lógica mais complexa.

7.4. while (enquanto):

Exemplos de uso do while:

- Ler dados de um usuário até que ele digite um valor válido.
- Simular um jogo que termina quando o jogador perde todas as vidas.
- Executar uma tarefa de forma contínua, como monitorar um sensor.

7.5. for (para):

É usada para criar um loop que repete um bloco de código para cada item em uma sequência.

```
for <variável> in <sequência>:
    # Bloco de código a ser executado para cada item da sequência
```

7.5. for (para):

Vantagens do for:

- Permite executar um bloco de código para cada item em uma sequência sem precisar repetir o código.
- É útil para tarefas que precisam ser repetidas um número determinado de vezes.
- Torna o código mais conciso e eficiente.

7.5. for (para):

Exemplos de uso do for:

- Ler os nomes de um arquivo e imprimir cada um em uma nova linha.
- Calcular a média de uma lista de números.
- Gerar uma lista de números pares de 1 a 10.

7.5. for (para):

Exemplos de uso do for:

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

soma = 0

for numero in numeros:
    soma += numero

print(f"A soma dos números é: {soma}")
```

7.5. for (para):

Exemplos de uso do for:

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

soma = 0

for numero in numeros:
    soma += numero

print(f"A soma dos números é: {soma}")
```

7.5. for (para):

Exemplos de uso do for: Este código itera sobre o dicionário pessoa e imprime a chave e o valor de cada parâmetro em uma nova linha.

```
pessoa = {
    "nome": "João",
    "idade": 25,
    "cidade": "São Paulo"
}

for chave, valor in pessoa.items():
    print(f"{chave}: {valor}")
```

7.5. for (para):

Exercícios: Tabuada usando while

```
numero = int(input("Digite um número: "))

multiplicador = 1

while multiplicador <= 10:
    resultado = numero * multiplicador
    print(f"{numero} x {multiplicador} = {resultado}")
    multiplicador += 1
```

7.5. for (para):

Exercícios: Tabuada usando for

```
numero = int(input("Digite um número: "))

for multiplicador in range(1, 11):
    resultado = numero * multiplicador
    print(f"{numero} x {multiplicador} = {resultado}")
```

7.5.1 Range:

Em Python, range é uma função que retorna uma sequência imutável de números em um intervalo especificado. É comumente utilizado em loops, principalmente em conjunto com a estrutura for, para iterar sobre uma sequência de números.

A sintaxe básica do range é:

```
range(start, stop[, step])
```

7.5.1 Range:

- **start:** O número inicial da sequência (opcional). Se não especificado, por padrão é assumido como 0.
- **stop:** O número de parada da sequência. O range gera números até, mas não incluindo, este número.
- **step:** O tamanho do passo entre os números (opcional). Por padrão, é assumido como 1.

```
range(start, stop[, step])
```

7.5.1 Range:

- **Exemplo:**

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

7.5. for (para):

Exercícios: fatorial com while

```
numero = int(input("Digite um número: "))

fatorial = 1

while numero > 1:
    fatorial *= numero
    numero -= 1

print(f"O fatorial de {numero} é {fatorial}")
```

7.5. for (para):

Exercícios: fatorial com for

```
numero = int(input("Digite um número: "))

fatorial = 1

for i in range(2, numero + 1):
    fatorial *= i

print(f"O fatorial de {numero} é {fatorial}")
```

7.5. for (para):

Exercícios: fatorial com for

```
numero = int(input("Digite um número: "))

fatorial = 1

for i in range(2, numero + 1):
    fatorial *= i

print(f"O fatorial de {numero} é {fatorial}")
```

7.5. for (para):

Exercícios: média de uma lista

```
numeros = []

while True:
    numero = input("Digite um número (ou 'sair' para finalizar): ")
    if numero == "sair":
        break
    numeros.append(float(numero))

media = sum(numeros) / len(numeros)

print(f"A média dos números digitados é {media}")
```

7.5. for (para):

Exercícios: média de uma lista

```
numeros = []

for i in range(5):
    numero = float(input("Digite um número: "))
    numeros.append(numero)

media = sum(numeros) / len(numeros)

print(f"A média dos números digitados é {media}")
```

7.5. for (para):

Exercícios: jogo de adivinhação

```
import random

numero_secreto = random.randint(1, 100)

palpite = 0

while palpite != numero_secreto:
    palpite = int(input("Digite um palpite: "))
    if palpite < numero_secreto:
        print("Seu palpite é menor que o número secreto.")
    elif palpite > numero_secreto:
        print("Seu palpite é maior que o número secreto.")

print("Você acertou o número secreto!")
```

Randon:

- O **random.randint(a, b)** é uma função do módulo random. Ele gera um número inteiro aleatório entre os valores de a e b, inclusive.
- A linha **numero_secreto = random.randint(1, 100)** é onde é gerado um número aleatório entre 1 e 100 (inclusive) e é atribuído à variável **numero_secreto**. Esta é a variável que o usuário tentará adivinhar no decorrer do jogo.

Randon:

- Portanto, o **random.randint()** está lá para gerar um número aleatório que o jogador deve tentar adivinhar. Isso adiciona um elemento de aleatoriedade e desafio ao jogo, pois o número secreto não é conhecido de antemão e pode mudar em cada execução do programa.

8. Tuplas e Listas:

- Tuplas e Listas são estruturas de dados fundamentais para armazenar e organizar coleções de dados.
- Apesar de algumas similaridades, como a capacidade de armazenar diferentes tipos de dados e serem indexadas, elas possuem diferenças importantes em termos de mutabilidade, desempenho e uso ideal.

8. 1.Tuplas:

- **Definição:** Uma tupla é uma coleção ordenada e imutável de elementos. Ela é criada usando parênteses e seus elementos são separados por vírgulas.
- **Exemplos:**

```
# Tupla vazia
minha_tupla = ()

# Tupla com um elemento
tupla_um_elemento = ("banana",)

# Tupla com múltiplos elementos
nomes = ("Ana", "João", "Maria")
coordenadas = (10.23, -54.32)
```

8. 1.Tuplas:

Características:

- **Imutabilidade:** Uma vez criada, uma tupla não pode ser alterada. Isso significa que você não pode adicionar, remover ou modificar seus elementos.
- **Acesso por índice:** Como as listas, as tuplas podem ser acessadas por índice para recuperar elementos específicos.

8. 1.Tuplas:

Características:

- **Desempenho:** Tuplas geralmente são mais eficientes que listas em termos de tempo de acesso e memória.
- **Uso ideal:** As tuplas são ideais para armazenar dados que não precisam ser modificados após a criação, como coordenadas, datas e informações de configuração.

8. 1.Tuplas:

Características:

```
# Criando uma tupla simples
tupla1 = (1, 2, 3, 4, 5)
```

8. 1.Tuplas:

Características:

```
# Acessando elementos da tupla
print("Elemento na posição 0:", tupla1[0])  # Saída: 1
print("Elemento na posição 3:", tupla1[3])  # Saída: 4
```

8. 1.Tuplas:

Características:

```
# Iterando sobre os elementos de uma tupla
print("Iterando sobre os elementos da tupla:")
for elemento in tupla1:
    print(elemento)
```

8. 1.Tuplas:

Características:

```
# Concatenando tuplas
tupla2 = ('a', 'b', 'c')
tupla_concatenada = tupla1 + tupla2
print("Tupla concatenada:", tupla_concatenada)
```

8. 1.Tuplas:

Características:

```
# Desempacotando uma tupla
x, y, z = tupla2
print("Desempacotando tupla:", x, y, z)
```

8. 1.Tuplas:

Características:

```
# Verificando se um elemento está presente na tupla
if 3 in tupla1:
    print("O elemento 3 está presente na tupla.")
else:
    print("O elemento 3 não está presente na tupla.")
```

8. 1.Tuplas:

Características:

```
# Obtendo o comprimento de uma tupla
print("Comprimento da tupla1:", len(tupla1))
```

8. 1.Tuplas:

Características:

```
# Utilizando tuplas como chaves em dicionários
dicionario = {(1, 2): 'valor1', (3, 4): 'valor2'}
print("Valor associado à chave (1, 2):", dicionario[(1, 2)])
```

8. 1.Tuplas:

Características:

```python
# Retornando múltiplos valores de uma função como tupla
def retorna_valores():
    return 1, 2, 3

valores = retorna_valores()
print("Valores retornados pela função:", valores)
```

8. 1.Tuplas:

Use tuplas:

- Quando você precisa armazenar dados que não precisam ser modificados.
- Quando você precisa de um alto desempenho em termos de tempo de acesso e memória.
- Quando você precisa de uma estrutura de dados imutável para garantir a segurança e a integridade dos dados.

8. 2.Listas:

Definição: Uma lista é uma coleção **ordenada** e **mutável** de elementos.

Ela é criada usando colchetes e seus elementos são separados por vírgulas.

8. 2.Listas:

```
# Lista vazia
minha_lista = []

# Lista com um elemento
lista_um_elemento = ["banana"]

# Lista com múltiplos elementos
frutas = ["maçã", "laranja", "uva"]
numeros = [1, 2, 3, 4]
```

8. 2.Listas:

Características:

- **Mutabilidade:** As listas podem ser facilmente modificadas após a criação. Você pode adicionar, remover e modificar seus elementos.
- **Acesso por índice:** As listas podem ser acessadas por índice para recuperar elementos específicos.

8. 2.Listas:

Características:

- **Desempenho:** As listas podem ser menos eficientes que tuplas em termos de tempo de acesso e memória, especialmente para operações de inserção e remoção.
- **Uso ideal:** As listas são ideais para armazenar dados que precisam ser modificados com frequência, como listas de compras, resultados de experimentos e conjuntos de dados dinâmicos.

8. 2.Listas:

Características:

- **Desempenho:** As listas podem ser menos eficientes que tuplas em termos de tempo de acesso e memória, especialmente para operações de inserção e remoção.
- **Uso ideal:** As listas são ideais para armazenar dados que precisam ser modificados com frequência, como listas de compras, resultados de experimentos e conjuntos de dados dinâmicos.

8. 2.Listas:

Características:

```
# Criando uma lista
lista_numeros = [1, 2, 3, 4, 5]
```

8. 2.Listas:

Características:

```
# Acessando elementos da lista
print("Elemento na posição 0:", lista_numeros[0])  # Saída: 1
print("Elemento na posição 3:", lista_numeros[3])  # Saída: 4
```

8. 2.Listas:

Características:

```
# Iterando sobre os elementos de uma lista
print("Iterando sobre os elementos da lista:")
for numero in lista_numeros:
    print(numero)
```

8. 2.Listas:

Características:

```
# Adicionando elementos à lista
lista_numeros.append(6)
print("Lista após adicionar o número 6:", lista_numeros)
```

8. 2.Listas:

Características:

```
# Removendo elementos da lista
lista_numeros.remove(3)
print("Lista após remover o número 3:", lista_numeros)
```

8. 2.Listas:

Características:

```
# Verificando se um elemento está presente na lista
if 4 in lista_numeros:
    print("O número 4 está presente na lista.")
else:
    print("O número 4 não está presente na lista.")
```

8. 2.Listas:

Características:

```
# Obtendo o comprimento da lista
print("Comprimento da lista:", len(lista_numeros))
```

8. 2.Listas:

Características:

```
# Ordenando uma lista
lista_desordenada = [4, 2, 1, 3, 5]
lista_ordenada = sorted(lista_desordenada)
print("Lista ordenada:", lista_ordenada)
```

8. 2.Listas:

Características:

```
# Utilizando listas como pilhas (LIFO)
pilha = []
pilha.append(1)
pilha.append(2)
pilha.append(3)
print("Pilha:", pilha)
elemento_removido = pilha.pop()
print("Elemento removido da pilha:", elemento_removido)
```

8. 2.Listas:

Características:

```
# Utilizando listas como filas (FIFO)
from collections import deque
fila = deque()
fila.append(1)
fila.append(2)
fila.append(3)
print("Fila:", fila)
elemento_removido = fila.popleft()
print("Elemento removido da fila:", elemento_removido)
```

8. 2.Listas:

Características:

```
# Compreensão de lista (list comprehension)
quadrados = [x ** 2 for x in range(1, 6)]
print("Quadrados dos números de 1 a 5:", quadrados)
```

8. 2.Listas:

Use listas:

- Quando você precisa armazenar dados que precisam ser modificados com frequência.
- Quando você precisa de uma estrutura de dados flexível que pode ser facilmente modificada.
- Quando você precisa de uma estrutura de dados que pode crescer dinamicamente.

9.Modularização:

A modularização é uma técnica fundamental para organizar e estruturar programas maiores e mais complexos. Ela consiste em dividir o código em unidades menores e independentes, chamadas de módulos.

9.Modularização:

Benefícios da Modularização:

- **Reutilização de código:** Módulos podem ser facilmente importados e reutilizados em outros programas, evitando duplicação de código e economizando tempo e esforço.
- Os Módulos tornam o código mais organizado e fácil de entender, facilitando a manutenção e correção de erros.

9.Modularização:

Benefícios da Modularização:

- **Legibilidade:** Módulos podem ser documentados de forma individual, tornando o código mais legível para outros desenvolvedores.
- **Testes:** Módulos podem ser testados de forma independente, facilitando a identificação de erros e bugs.

9.Modularização:

Como Modularizar o Código em Python:

- **Definição de Módulos:** Um módulo é um arquivo Python que contém um conjunto de funções, classes e variáveis.
- **Importando Módulos:** Módulos podem ser importados em outros programas usando a palavra-chave import.
- **Organização do Código:** Módulos podem ser organizados em pastas e subpastas para facilitar a navegação e o gerenciamento do código.

9.Modularização:

Pacotes Python:

Pacotes Python são coleções de módulos relacionados que podem ser instalados e utilizados em seus projetos.

O Python Package Index (PyPI) é um repositório online que contém milhares de pacotes disponíveis para download e instalação.

9.Modularização:

Ferramentas para Modularização:

Ferramentas de formatação de código: Ferramentas como o Black e o YAPF podem ser utilizadas para formatar automaticamente o código e garantir a consistência do estilo.

9.Modularização:

Ferramentas para Modularização:

Linters: Ferramentas como o Pylint e o Flake8 podem ser utilizadas para verificar o código em busca de erros e potenciais problemas.

9.Modularização:

```python
# modulo_matematica.py

def soma(a, b):
    """Soma dois números."""
    return a + b

def subtracao(a, b):
    """Subtrai um número do outro."""
    return a - b

def multiplicacao(a, b):
    """Multiplica dois números."""
    return a * b

def divisao(a, b):
    """Divide um número pelo outro."""
    return a / b
```

9.Modularização:

```python
# main.py

import modulo_matematica

resultado_soma = modulo_matematica.soma(10, 5)
resultado_subtracao = modulo_matematica.subtracao(10, 5)
resultado_multiplicacao = modulo_matematica.multiplicacao(10, 5)
resultado_divisao = modulo_matematica.divisao(10, 5)

print(f"Soma: {resultado_soma}")
print(f"Subtração: {resultado_subtracao}")
print(f"Multiplicação: {resultado_multiplicacao}")
print(f"Divisão: {resultado_divisao}")
```