Funções

Funções

▼ Definir função (**def**)

- Utiliza-se **def** para criar uma função.
  - Exemplo: `def funcao ():`
- Chamada de função é realizada digitando o nome da função + parênteses.
  - Exemplo: `funcao ()`
- Exemplo de um função simples:

```
def nome(nome):
        print(f"Meu nome é {nome}")

    nome("Gustavo")
```

▼ Parâmetro nomeado

- Normalmente, os argumentos devem ser passados para a função de modo a coincidir com a ordem dos parâmetros definidos. Entretanto, é possível definir os argumentos em outra ordem, caso sejam nomeados. Exemplo:

```
def teste(x, y):
    print(x, y)
```

```
teste(1, 2)  # x=1, y=2
teste(y=1, x=2)  # x=2, y=1
```

- Não é possível passar argumentos não nomeados após argumentos nomeados

▼ Valor padrão para parâmetro

- É possível definir um valor padrão para o parâmetro, que será utilizado caso não seja passado um valor para ele. Exemplo:

```
def saudacao(nome, mensagem = "Olá"):
    print(f"{mensagem}, {nome}!")

saudacao("João", "Bem-vindo")
saudacao("Maria")  # Exibe: "Olá, Maria!"
```

- Também é possível definir um parâmetro sem valor, de modo que parâmetro = None
  - Exemplo de uso:

```
def exibir_mensagem(nome, mensagem = None):
    if mensagem is None:
        mensagem = "Olá!"
    print(f"{mensagem}, {nome}!")

exibir_mensagem("João", "Bem-vindo")  # Exibe: "Bem-vindo, João!"
exibir_mensagem("Maria")  # Exibe: "Olá, Maria!"
```

▼ Utilizar variável global em uma função (**global**)

- `global` é usado para declarar que uma variável dentro de uma função é a mesma que uma variável global (fora da função). Isso permite que a função modifique o valor da variável global diretamente, em vez de criar uma nova variável local com o mesmo nome. Exemplo:

```
contador = 0  # Variável global

def incrementar():
```

```
    global contador
    contador += 1

incrementar()
print(contador)  # Exibe: 1

incrementar()
print(contador)  # Exibe: 2
```

- Mesmo apresentando uma boa utilidade, o uso de `global` pode ser considerado como uma má prática de programação. Isso ocorre pois não é ideal a criação de outra variável dentro de uma função, com o mesmo nome de uma variável global.

▼ Verificar variáveis locais ou globais de uma função (**locals( )** e **globals( )**)

- `locals()` mostra as variáveis locais de uma função. É usada dentro do escopo da função que será verificada
- `globals()` pode ser utilizada para acessar uma variável global

```
x = 100  # Variável global

def teste():
    y = 50  # Variável local
    print("Locals:", locals())  # Mostra as variáveis locais
    print("Globals:", globals()["x"])  # Acessa uma variável global

teste()
```

▼ Função que retorna valor **None**

- Funções que não retornam valor explicitamente, ou seja, que não possuem uma instrução `return` ou que usam `return` sem especificar nada, retornam automaticamente o valor especial `None`
    - Pode-se fazer um paralelo com o tipo `void` em C, pois ambos indicam ausência de um valor significativo retornado pela função. Mas se diferenciam, já que em Python, funções sempre retornam algo (mesmo que seja implicitamente `None`), enquanto o tipo `void` em C representa ausência total de retorno

- Exemplo de função que retorna `None` :

```
def minha_funcao():
    print("Esta função não retorna nada")
resultado = minha_funcao()  # resultado será None
print(resultado)  # Exibe None
```

  - Isso ocorre pois `print()` é uma função que não retorna um valor específico, cumprindo apenas a utilidade de exibir algo na tela. Assim, não é atribuído à variável qualquer tipo de valor significativo

▼ Retorno de valores das funções (**return**)

- **Utilidade do `return`:** O `return` é usado para devolver um valor a partir de uma função, permitindo que ele seja armazenado em uma variável ou usado em outra parte do programa.
  - Diferentemente de funções que retornam `None`, aqui o `return` permite capturar e utilizar o resultado (`8`, no exemplo a seguir) para cálculos, exibições ou outras operações no programa.

```
def soma(a, b):
    return a + b

resultado = soma(3, 5)  # A função retorna 8
print(resultado)  # Exibe 8
```

▼ Parâmetro para quantidade de argumentos não nomeados (***args**)

- `args` é uma convenção para representar um parâmetro que pode aceitar um número variável de argumentos posicionais.
- O asterisco **(`*`)** antes do nome indica que a função pode receber múltiplos valores, que serão agrupados em uma tupla.
- `args` é apenas um nome convencional, podendo ser substituído por outro, mas o uso de `args` é amplamente adotado por questões de legibilidade.
- Exemplo do uso de ***args**:

```
def minha_funcao(*args):
    for arg in args:
        print(arg)

minha_funcao(1, 2, 3, 4)  # Exibe: 1, 2, 3, 4
```

- Neste exemplo, a função recebe múltiplos argumentos e os exibe um a um. O uso do `*args` facilita passar quantos argumentos forem necessários sem precisar especificá-los individualmente.

▼ Argumentos nomeados / keyword arguments (**`**kwargs`**)

- `kwargs` é uma convenção para passar argumentos nomeados variáveis para uma função. A palavra "kwargs" significa "keyword arguments" (argumentos por palavra-chave). Quando se usa `**kwargs`, a função pode aceitar um número variável de argumentos nomeados **(chave=valor)**.
- Exemplo:

```
def saudacao(**kwargs):
    for chave, valor in kwargs.items():
        print(chave, valor)

saudacao(nome='João', idade=25)
# Exibe:
nome João
idade 25
```

  - Nesse exemplo, `**kwargs` permite passar quantos parâmetros nomeados(par chave-valor) forem necessários. Dentro da função, os argumentos são tratados como um dicionário, onde as chaves são os nomes dos parâmetros e os valores são os valores passados.

- Observação: O nome `kwargs` não é obrigatório. Pode-se usar qualquer nome, mas a convenção é usar `kwargs`.

▼ Positional-Only Parameters (**/**) e Keyword-Only Arguments (*****)

- Positional-Only Parameters é basicamente o uso de `/` na definição da função, sendo utilizado para indicar que os parâmetros à esquerda dele só podem ser passados por posição, e não por nome
    - Em outras palavras, tudo que vem antes da barra nos parâmetros de uma função devem ser argumentos não nomeados (passados por posição)
    - Exemplo:

```
def soma(a, b, /):
    return a + b

print(soma(3, 5))   # Correto
print(soma(a=3, b=5))  # Erro! "a" e "b" só podem ser passados po
```

- Keyword-Only Arguments é basicamente o uso de `*` na definição da função, sendo utilizado para indicar que todos os parâmetros à direita dele só podem ser passados por nome (keyword arguments).
    - Em outras palavras, tudo que vem depois do asterisco nos parâmetros de uma função devem ser argumentos nomeados (passados por posição)
    - Exemplo:

```
def multiplicacao(x, *, y):
    return x * y

print(multiplicacao(4, y=3))   # Correto
print(multiplicacao(4, 3))  # Erro! "y" deve ser passado por nome
```

▼ First-Class Functions e Higher Order Functions (Primeira Classe e Ordem Superior)

- **Funções de Primeira Classe (First-Class Functions):** são funções que podem ser tratadas como valores. Isso permite que funções sejam atribuídas a variáveis, passadas como **argumentos** para outras funções e **retornadas** por outras funções.
    - De forma simples, são funções que são tratadas como outros tipos de dados comuns (strings, inteiros, etc...).

- Exemplo:

```
def saudacao():
    print("Olá!")

minha_funcao = saudacao
minha_funcao()  # Exibe: Olá!
```

- **Funções de Ordem Superior (Higher Order Functions):** São funções que **recebem outras funções como argumentos** ou **retornam funções**. Elas são uma aplicação do conceito de funções de primeira classe, permitindo maior flexibilidade e abstração no código.
  - De forma simples, são funções que podem receber e/ou retornar outras funções.
  - Exemplo:

```
def aplicar_funcao(func, valor):
    return func(valor)

def quadrado(x):
    return x ** 2

resultado = aplicar_funcao(quadrado, 4)  # resultado será 16
```

- Exemplo envolvendo as duas:

```
# Função de primeira classe (First-Class Function), pois é atribuída como
def saudacao(msg, nome):
    return f"{msg}, {nome}!"

# Função de ordem superior (Higher Order Function), pois recebe uma fur
def executa(funcao, *args):
    return funcao(*args)

# Exemplos de Higher Order Function chamando uma First-Class Function
```

```
print(executa(saudacao, "Bom dia", "Luiz"))
print(executa(saudacao, "Bom noite", "Maria"))
```

▼ Closure

- Uma **closure** em Python é uma função que "lembra" o ambiente no qual foi criada, mesmo depois que esse ambiente não existe mais. Em outras palavras, é uma função que captura e utiliza variáveis do escopo em que foi definida, mesmo que esse escopo já tenha sido encerrado.
    - Isso ocorre porque a função "interna" pode acessar as variáveis locais da função "externa" onde foi criada, mesmo após a execução dessa função externa.
- Estrutura básica de uma closure:

1. Há uma função externa que define um escopo.
2. Dentro dessa função externa, há uma função interna que utiliza variáveis do escopo externo.
3. A função externa retorna a função interna, que mantém o acesso ao escopo onde foi definida.

- Exemplo:

```
def criar_multiplicador(multiplicador):
    def multiplicar(numero):
        return numero * multiplicador
    return multiplicar

# Criamos uma função "dobrar" que lembra do multiplicador = 2
dobrar = criar_multiplicador(2)

# Criamos outra função "triplicar" que lembra do multiplicador = 3
triplicar = criar_multiplicador(3)

# Usando as funções
print(dobrar(5))      # Exibe: 10
print(triplicar(5))   # Exibe: 15
```

- A função `criar_multiplicador` é a função **externa** e possui a variável `multiplicador`.
- A função `multiplicar` é a função **interna**, que usa a variável `multiplicador` do escopo da função externa.
- Quando `criar_multiplicador` retorna a função `multiplicar`, essa função ainda "lembra" do valor de `multiplicador`, mesmo que `criar_multiplicador` já tenha terminado sua execução.
- Assim, a função interna (que forma a closure) não é executada imediatamente quando a função externa é chamada, mas fica "guardada" com o contexto do escopo em que foi criada.
- A execução só acontece quando a função retornada (a closure) é chamada posteriormente.

▼ Função **lambda** (definir função em uma linha)

> ❓ As funções **lambda** são funções sem nome criadas em uma única linha, geralmente usadas para tarefas simples. Elas permitem definir funções de forma concisa usando a palavra-chave `lambda`

- **Estrutura**: lambda *argumentos*: expressão
- As funções lambda são úteis para simplificar código em situações onde funções tradicionais seriam desnecessariamente verbosas
- Exemplo simples (somar dois número):

```
soma = lambda x, y: x + y
print(soma(2, 3))  # Exibe: 5
```

- Exemplo simples (dobrar um número):

```
dobrar = lambda x: x * 2
print(dobrar(4))  # Exibe: 8
```

- Exemplo em ordenação de uma lista de tuplas:

```
tuplas = [(1, 'b'), (3, 'a'), (2, 'c')]
tuplas.sort(key=lambda x: x[1])
print(tuplas)  # Exibe: [(3, 'a'), (1, 'b'), (2, 'c')]
```

> ‼️ Ao utilizar a função **lambda**, deve-se evitar usá-la para lógicas complexas, pois isso prejudica a legibilidade e dificulta a depuração. Lambdas são mais adequados para funções simples e descartáveis, como em `map`, `filter` ou `sorted`. Ademais, deve-se usar `def` se a função for reutilizada ou precisar de nomeação. Lambdas suportam apenas expressões simples, não múltiplas linhas ou comandos complexos

▼ Variáveis livres + variáveis não locais (**nonlocal**)

- Uma **variável livre** é uma variável usada dentro de uma função, mas que não é definida dentro dela. Em vez disso, ela vem de um escopo externo. Isso geralmente acontece dentro de **closures**. Exemplo:

```
def externa():
    x = 10  # Variável definida no escopo externo

    def interna():
        print(x)  # x é uma variável livre aqui

    return interna

funcao = externa()  # Cria a closure
funcao()  # Exibe: 10
```

  - `x` é uma variável livre dentro de `interna()`, pois ela **não é definida dentro de** `interna()`, mas sim na função `externa()`.

- A palavra-chave `nonlocal` permite modificar variáveis **não locais**, ou seja, variáveis que pertencem ao escopo de uma função externa (mas não ao escopo global). Exemplo:

```
def contador():
    count = 0  # Variável no escopo da função externa

    def incrementar():
        nonlocal count  # Permite modificar a variável count
        count += 1
        return count
    return incrementar

c = contador()
print(c())  # Exibe: 1
print(c())  # Exibe: 2
```

- Sem `nonlocal`, a variável `count` seria tratada como uma **variável livre apenas para leitura**, e tentar modificar seu valor causaria um erro (`UnboundLocalError: cannot access local variable 'count' where it is not associated with a value`).

▼ Funções recursivas (**return função( )**)

> ❓ Uma função recursiva é uma função que se chama dentro de si mesma. Esse processo de repetição continua até atingir uma condição chamada de caso base, que interrompe a recursão. A recursão é uma técnica muito útil quando o problema pode ser dividido em subproblemas menores que têm a mesma estrutura do problema original.

- **Caso Base**: O ponto onde a recursão deve parar. É uma condição simples que não chama mais a função recursiva. Sem o caso base, a função entraria em um ciclo infinito e causaria um erro chamado **stack overflow**.
- **Passo Recursivo**: Aqui, a função se chama novamente, mas com um valor alterado, geralmente menor ou mais simples do que o valor original. O objetivo é reduzir o problema até atingir o caso base.
- **Pilha de execução:** Quando uma função recursiva é chamada, o estado atual da função é armazenado na pilha de execução. Cada chamada

subsequente cria um novo nível na pilha, com uma nova instância da função, até que o caso base seja atingido. Quando isso acontece, as chamadas começam a ser resolvidas na ordem inversa, retornando os valores e desempilhando as funções uma a uma.

- Exemplo simples de recursão para contagem regressiva:

```
def contagem_regressiva(n):
    if n <= 0:  # Caso base
        print("Fim!")
        return
    print(n)
    contagem_regressiva(n - 1)  # Chamada recursiva

contagem_regressiva(5)
```

- Exemplo de recursão em função fatorial:

```
def fatorial(n):
    if n <= 1:  # Caso base
        return n
    return n * fatorial(n - 1)  # Chamada recursiva

print(fatorial(5))  # Exibe: 120
```

- Exemplo para a sequeência de Fibonacci:

```
def fatorial(n):
    if n <= 1:  # Caso base
        return n
    return n * fatorial(n - 1)  # Chamada recursiva

print(fatorial(5))  # Exibe: 120
```

▼ Problema dos parâmetros utáveis

❓ O problema dos parâmetros mutáveis em Python ocorre quando se usa um objeto mutável (como listas ou dicionários) como valor padrão de um parâmetro em uma função. Isso pode levar a comportamentos inesperados porque o objeto é criado uma única vez, no momento da definição da função, e não recriado a cada chamada.

- Exemplo do problema:

```
def adiciona_clientes(nome, lista=[]):
    lista.append(nome)
    return lista


cliente1 = adiciona_clientes('luiz')
adiciona_clientes('Joana', cliente1)
print(cliente1)  # ['luiz', 'Joana']

cliente2 = adiciona_clientes('Helena')
adiciona_clientes('Maria', cliente2)

print(cliente2)  # ['luiz', 'Joana', 'Helena', 'Maria']
```

  - No exemplo acima, a lista usada como valor padrão não é recriada a cada chamada da função, então os itens continuam sendo adicionados à mesma lista.

- Exemplo solucionado:

```
def adiciona_clientes(nome, lista=None):
    if lista is None:
        lista = []
    lista.append(nome)
    return lista
```

```
cliente1 = adiciona_clientes('luiz')
adiciona_clientes('Joana', cliente1)
adiciona_clientes('Fernando', cliente1)
cliente1.append('Edu')

cliente2 = adiciona_clientes('Helena')
adiciona_clientes('Maria', cliente2)

cliente3 = adiciona_clientes('Moreira')
adiciona_clientes('Vivi', cliente3)

print(cliente1)  # ['luiz', 'Joana', 'Fernando', 'Edu']
print(cliente2)  # ['Helena', 'Maria']
print(cliente3)  # ['Moreira', 'Vivi']
```

- Para evitar esse problema, usa-se um valor imutável como `None` e cria-se o objeto dentro da função
- Agora, uma nova lista será criada toda vez que `lista = None`, evitando o problema

▼ Evitar indentação em funções - Guard Clause

> ❓ O conceito de Guard Clause é uma prática que melhora a legibilidade do código ao evitar indentação excessiva (espaços ou tabulações à esquerda, como muitos `if`). Ele consiste em verificar condições logo no início de uma função e retornar ou lançar uma exceção imediatamente caso essas condições não sejam atendidas. Isso evita a necessidade de aninhamento excessivo de blocos `if` dentro da função.

- Desse modo, o uso de Guard Clause pode ser útil para:
  - Remover a necessidade de `else`, tornando o código mais direto
  - Reduzir a indentação, tornando o código mais fácil de ler
  - Torna o fluxo da função mais claro, pois os casos inválidos são tratados logo no início

- Exemplo SEM Guard Clause:

```
def dobrar_natural(n):
    if isinstance(n, int) and n >= 0:
        resultado = n * 2
        return resultado
    else:
        return "Número inválido"
```

  - Aqui, o fluxo principal do código fica dentro do `if`, aumentando a indentação desnecessariamente

- Exemplo COM Guard Clause:

```
def dobrar_natural(n):
    if not isinstance(n, int) or n < 0:
            return "Número inválido"  # Retorna cedo se a condição não fo
    return n * 2  # Fluxo principal fica menos indentado
```

  - Aqui, o fluxo principal do código fica dentro do `if`, aumentando a indentação desnecessariamente