

[Mostrar código](#)

# Curso IA Aplicada aos Desafios Socioambientais da Amazônia



## Bloco 3 :: Encontro 10

### Sobre o curso

O curso Inteligência Artificial Aplicada aos Desafios Socioambientais da Amazônia, promovido pelo Instituto de Inteligência Artificial Aplicada (I2A2), é uma iniciativa pioneira voltada para capacitar moradores do Pará e da região Norte. Com duração de seis meses, combina aulas online, atividades assíncronas, workshops práticos, encontros com especialistas e mentoria contínua. O objetivo é aplicar a IA de forma prática em problemas ambientais reais, em alinhamento com as diretrizes e temas da COP30.

Ao longo do curso, os participantes exploram desde fundamentos de Machine Learning até IA Generativa, trabalhando com dados ambientais da Amazônia para enfrentar questões como desmatamento, queimadas, qualidade da água e riscos climáticos. Cada grupo finaliza sua jornada com um projeto integrador, apresentando soluções sustentáveis e socialmente viáveis. A formação busca fortalecer competências técnicas e o protagonismo de lideranças locais, articulando ciência, tecnologia e justiça ambiental, consolidando a Amazônia como polo de inovação para o desenvolvimento sustentável.



Linguagem Python

### ≡ Trilha do Conhecimento

- Criação e chamada de funções simples.
- Organização de dados em listas.
- Atividade: calcular média móvel de área desmatada.
- Desafio 5

### ≡ FUNÇÕES

Funções são blocos de código reutilizáveis que só são executados quando chamados. Elas permitem que o programador organize melhor seu código, evitando repetições e facilitando a manutenção. Uma função pode receber dados de entrada, **chamados de parâmetros**, que são utilizados para realizar alguma operação interna. **Ao final da execução, ela pode retornar um resultado**

Para exemplificar e refletir, podemos comparar uma função a um pequeno agente que recebe dados do ambiente – como temperatura, umidade ou imagens de satélite – e entrega respostas importantes para ações ambientais. Por exemplo, uma função pode ser responsável por analisar dados de monitoramento florestal e retornar alertas de risco de desmatamento. Dessa forma, as funções são fundamentais para transformar dados em decisões, contribuindo com soluções tecnológicas para a preservação da Amazônia.

```
def nome_da_funcao(param1, param2):  
    # bloco de código  
    resultado = param1 + param2  
    return resultado
```

#### Componentes da estrutura:

def: palavra-chave que define uma função.

nome\_da\_funcao: nome que você escolhe para identificar a função.

(param1, param2): os parâmetros que a função recebe (podem ser zero ou mais).

:: indica o início do corpo da função.

return: (opcional) indica o valor que será retornado pela função.

## ❯ :: Criando uma Função

Em Python, uma função é definida usando a **palavra-chave def**:

```
1 def alerta_calor(temperatura):  
2     if temperatura > 35:  
3         print("⚠️ Alerta: temperatura elevada! Risco de estresse térmico na fauna e flora.")  
4     else:  
5         print("✅ Temperatura dentro do normal para a região.")
```

[+ Código](#)[+ Texto](#)

## ❯ :: Chamando uma Função

Para chamar uma função, use o nome da função seguido por parênteses:

```
1 alerta_calor(38)
```

```
⚠️ Alerta: temperatura elevada! Risco de estresse térmico na fauna e flora.
```

## ❯ :: Argumentos

Argumentos são informações que enviamos para dentro de uma função quando ela é chamada. Eles são colocados entre parênteses logo após o nome da função. Se uma função precisa usar algum dado para realizar uma tarefa — como um número, um nome ou uma medida ambiental — esse dado é passado como argumento. Podemos usar um ou vários argumentos, separados por vírgulas, dependendo do que a função precisa para funcionar corretamente.

Por exemplo, se quisermos criar uma função que receba o nome de uma pessoa envolvida em um projeto socioambiental, podemos passar esse nome como argumento e usá-lo dentro da função para personalizar uma mensagem. Isso ajuda a tornar os sistemas mais dinâmicos e úteis. No contexto do nosso curso, os argumentos são como os dados coletados do meio ambiente (como temperatura, umidade ou localização) que são enviados para as funções, permitindo que a IA analise, interprete e ofereça soluções para os desafios da Amazônia.

```
def nome_da_funcao(argumento1, argumento2, ...):  
    # bloco de código que usa os argumentos  
    return resultado # (opcional)  
  
1 def mostrar_alerta(nome_projeto):  
2     print("Monitoramento iniciado para o projeto:", nome_projeto)
```

```
1 mostrar_alerta("Pé-de-Pincha")
```

```
⚡ Monitoramento iniciado para o projeto: Pé-de-Pincha
```

## ❯ :: Parâmetros ou Argumentos?

Parâmetros e argumentos são palavras que muitas vezes significam a mesma coisa: **informações passadas para uma função**. Mas há uma pequena diferença. Quando estamos **definindo a função**, os nomes que colocamos entre parênteses são chamados de **parâmetros** — eles funcionam como “caixinhas” prontas para receber os dados. Já quando **chamamos a função**, os valores reais que enviamos para essas caixinhas são chamados de **argumentos**. Por exemplo, se criamos uma função que espera um nome (nome é o parâmetro), e depois chamamos essa função com "Projeto Amazônia" (esse é o argumento), estamos fazendo a mágica acontecer com dados reais do nosso contexto ambiental.

## ❯ :: Número de Argumentos?

Por padrão, uma função precisa ser chamada com a **quantidade exata de argumentos que ela espera**. Isso significa que, se a função foi definida para receber dois argumentos, você deve chamá-la com exatamente dois valores — **nem mais, nem menos** — caso contrário, o Python exibirá um erro.

```
1 def alerta_queimada(regiao, nivel_alerta):
2     print("="*80)
3     print("🔥 ALERTA DE QUEIMADA!".center(80))
4     print("="*80)
5     print("Região monitorada:", regiao.upper())
6     print("Nível do alerta:", nivel_alerta.upper())
```

```
1 alerta_queimada("Alto Xingu", "Crítico")
```

```
=====
                        🔥 ALERTA DE QUEIMADA!
=====
Região monitorada: ALTO XINGU
Nível do alerta: CRÍTICO
```

## ⌵ :: Argumentos Arbitrários, \*args

Pode haver situações em que **não sabemos quantas informações serão recebidas por uma função** — por exemplo, dados vindos de sensores em diferentes regiões. Nesses casos, usamos um `*` antes do nome do parâmetro na definição da função. Isso permite que a função receba uma **quantidade variável de argumentos**, organizados como uma tupla, que pode ser percorrida e analisada conforme necessário. Assim, conseguimos criar funções mais flexíveis para lidar com diferentes tipos de entradas ambientais.

```
1 import time
2
3 def registrar_alertas(*regioes):
4     print("="*80)
5     print("Iniciando monitoramento nas seguintes regiões".center(80))
6     print("="*80)
7     for i, regiao in enumerate(regioes, start=1):
8         print(f"📍 Região {i}: {regiao} iniciado monitoramento!")
9         time.sleep(1)
```

```
1 registrar_alertas("Marajó", "Xingu", "Tapajós", "Javari")
```

```
=====
                        Iniciando monitoramento nas seguintes regiões
=====
📍 Região 1: Marajó iniciado monitoramento!
📍 Região 2: Xingu iniciado monitoramento!
📍 Região 3: Tapajós iniciado monitoramento!
📍 Região 4: Javari iniciado monitoramento!
```

## ⌵ :: Argumentos de Palavras-Chave

Também é possível enviar argumentos para uma função usando a **sintaxe de chave = valor** (por exemplo, `regiao="Marajó"`). Nesse formato, cada argumento é identificado pelo nome do parâmetro definido na função, permitindo que você **não se preocupe com a ordem** em que os argumentos são passados. Isso torna o código mais claro e flexível — especialmente útil em aplicações com muitos dados ambientais, como no nosso curso, onde uma função pode receber temperatura, umidade, nome da região e nível de alerta, independentemente da sequência em que esses dados são enviados.

```
1 def gerar_alerta(regiao, tipo_alerta, nivel):
2     print("-" * 40)
3     print(f"📍 Região: {regiao}")
4     print(f"🌿 Tipo de alerta: {tipo_alerta}")
5     print(f"🔥 Nível: {nivel}")
6     print("-" * 40)
```

```
1 gerar_alerta(nivel="Crítico", tipo_alerta="Queimada", regiao="Xingu")
```

```
-----
📍 Região: Xingu
🌿 Tipo de alerta: Queimada
🔥 Nível: Crítico
-----
```

## ⌵ :: Argumentos de Palavras-Chave Arbitrárias, \*\*kwargs

Quando não sabemos quantos argumentos nomeados (chave = valor) serão passados para uma função, podemos usar **\*\* antes do nome do parâmetro na definição da função — geralmente chamado de \*\*kwargs**. Isso faz com que a função receba esses argumentos como um **dicionário**, onde cada chave representa o nome do parâmetro e cada valor é o dado correspondente. Essa abordagem torna a função mais flexível, permitindo lidar com diferentes tipos e quantidades de informações — algo muito útil no nosso curso, onde os dados ambientais podem variar bastante entre sensores ou regiões monitoradas.



```
1 def registrar_dados(**kwargs):
2     print("📊 Dados recebidos da estação de monitoramento:")
```

```

3     for chave, valor in kwargs.items():
4         print(f"- {chave.capitalize()}: {valor}")

1 registrar_dados(regiao="Marajó", temperatura="32°C", umidade="88%", co2="Alto")

```

  Dados recebidos da estação de monitoramento:

- Regiao: Marajó
- Temperatura: 32°C
- Umidade: 88%
- Co2: Alto

## ⌵ :: Valor do Parâmetro Padrão


É possível definir um valor padrão para um parâmetro. Assim, se a função for chamada sem esse argumento, ela usará o **valor definido como padrão**. Isso torna a função mais flexível, ideal quando nem sempre temos todos os dados disponíveis em aplicações ambientais.

```

1 def alerta_chuva(regiao="Região Desconhecida"):
2     print(f"☁️ Alerta de chuvas intensas para: {regiao}")

```

```
1 alerta_chuva("Tapajós")
```

 ☁️ Alerta de chuvas intensas para: Tapajós

```
1 alerta_chuva()
```

 ☁️ Alerta de chuvas intensas para: Região Desconhecida

## ⌵ :: Valores de Retorno

Para que uma função devolva um resultado, usamos a instrução **return**. Ela encerra a função e envia um valor de volta para quem a chamou, permitindo que esse resultado seja usado em outras partes do código — como cálculos, relatórios ou decisões automatizadas.

```

def nome_da_funcao(parâmetros):
    # bloco de código
    resultado = operação_com_os_parâmetros
    return resultado

```

```

1 def calcular_nivel_alerta(temperatura):
2     if temperatura > 35:
3         return "Crítico"
4     elif temperatura > 30:
5         return "Moderado"
6     else:
7         return "Normal"

```

```

1 nivel = calcular_nivel_alerta(33)
2 print("🔍 Nível de alerta:", nivel)

```

 🔍 Nível de alerta: Moderado

## ⌵ :: A Declaração "pass"

**Funções não podem ficar vazias**, mas se você ainda não definiu o que ela fará, use a instrução **pass**. Isso evita erros e permite que você construa a estrutura do código aos poucos.

```

1 def analisar_dados_sensor():
2     pass # Função ainda será implementada futuramente

```

## ⌵ :: Argumentos Somente Posicionais

Você pode definir que certos argumentos de uma função devem ser passados somente por posição, e não por nome. Para isso, basta adicionar `/` após esses parâmetros na definição da função. Isso ajuda a **evitar erros ou confusões no uso da função**.

```

1 def registrar_temperatura(regiao, temperatura, /):
2     print(f"🔥 Temperatura registrada na região {regiao}: {temperatura}°C")

```

```

1 # chamada correta (por posição):
2 registrar_temperatura("Xingu", 34)

```

 🔥 Temperatura registrada na região Xingu: 34°C

```

1 # chamada incorreta (por palavra-chave)
2 registrar_temperatura(regiao="Xingu", temperatura=34) #Vai gerar erro!

```

## ⌵ :: Argumentos Somente de Palavras-Chave

Para definir que certos argumentos de uma função devem ser passados somente por **palavra-chave (usando nome=valor)**, basta adicionar um `*` antes desses parâmetros na definição da função. Isso aumenta a clareza e evita erros na ordem dos dados.

```
1 def registrar_qualidade_ar(*, regioao, co2, umidade):
2     print(f"📍 Região: {regiao}")
3     print(f"🌫️ Nível de CO2: {co2}")
4     print(f"💧 Umidade do ar: {umidade}")

1 # chamada correta
2 registrar_qualidade_ar(regiao="Tapajós", co2="Alto", umidade="72%")
```

```
📄
📍 Região: Tapajós
🌫️ Nível de CO2: Alto
💧 Umidade do ar: 72%
```

```
1 # chamada incorreta
2 registrar_qualidade_ar("Tapajós", "Alto", "72%")
```

**Atenção:** O uso de `*` obriga o uso de nomes nos argumentos, deixando claro o que cada dado representa. Isso é essencial em sistemas de IA que lidam com diversos parâmetros ambientais, onde confundir a ordem pode comprometer a análise.

## ⌵ :: Combine Somente Posicional e Somente Palavra-Chave

Você pode combinar os dois tipos de argumentos na mesma função. Os parâmetros que vêm antes da `/` devem ser passados **somente por posição**, e os que vêm depois do `*` devem ser passados **somente por palavra-chave**. Isso permite mais controle e clareza no uso da função.

```
1 def registrar_dados_ambientais(regiao, /, *, temperatura, umidade):
2     print(f"📍 Região monitorada: {regiao}")
3     print(f"🌡️ Temperatura: {temperatura}°C")
4     print(f"💧 Umidade: {umidade}%")

1 # chamada correta
2 registrar_dados_ambientais("Xingu", temperatura=33, umidade=80)
```

```
📄
📍 Região monitorada: Xingu
🌡️ Temperatura: 33°C
💧 Umidade: 80%
```

```
1 # chamada incorreta
2 registrar_dados_ambientais(regiao="Xingu", temperatura=33, umidade=80)
```

## ⌵ :: Recursão

Recursão é quando uma função chama a si mesma dentro de sua própria definição. É um conceito comum em matemática e programação, usado para resolver problemas de forma repetitiva e elegante, como cálculos ou percursos em estruturas de dados.

Em Python, isso é possível e pode ser útil para automatizar tarefas que se repetem com pequenas variações. No entanto, é preciso ter cuidado: se a função não tiver uma **condição de parada**, ela pode entrar em um loop infinito, consumindo muita memória ou travando o programa. Com uma lógica bem definida, a recursão pode ser uma solução poderosa — por isso, é ideal testá-la e ajustar conforme necessário para entender bem seu funcionamento.

```
1 def contagem_regressiva(n):
2     if n <= 0:
3         print("🚀 Lançamento!")
4     else:
5         print(f"🕒 Contando: {n}")
6         contagem_regressiva(n - 1)
```

```
1 contagem_regressiva(5)
```

```
📄
🕒 Contando: 5
🕒 Contando: 4
🕒 Contando: 3
🕒 Contando: 2
🕒 Contando: 1
🚀 Lançamento!
```

```
1 def contar_ate(n):
2     if n == 0:
3         return
4     contar_ate(n - 1)
5     print(n)
```

```
1 contar_ate(5)
```

```
1  
2  
3  
4  
5
```

## LISTAS

Em Python, listas são usadas para armazenar múltiplos itens em uma única variável. Elas fazem parte dos quatro principais tipos de dados para coleções em Python — junto com tuplas, conjuntos (sets) e dicionários (dicts) — cada um com características e finalidades diferentes.

As listas são criadas utilizando colchetes [], e permitem armazenar elementos de qualquer tipo (números, textos, até outras listas), podendo ser modificadas a qualquer momento. São ideais quando precisamos trabalhar com conjuntos de dados ambientais variáveis, como temperaturas diárias, nomes de regiões monitoradas ou registros de alertas.

```
nome_da_lista = [item1, item2, item3, ...]
```

```
1 regioes_monitoradas = ["Marajó", "Xingu", "Tapajós", "Javari"]
```

```
1 type(regioes_monitoradas)
```

```
list
```

```
1 print(regioes_monitoradas)
```

```
['Marajó', 'Xingu', 'Tapajós', 'Javari']
```

## :: Itens da Lista

Os itens de uma lista em Python são ordenados, ou seja, mantêm a sequência em que foram inseridos. Eles também são modificáveis, permitindo que sejam alterados após a criação da lista, e aceitam valores repetidos.

Cada item na lista possui uma posição específica, chamada de índice. O primeiro item tem o índice [0], o segundo [1], e assim por diante. Isso permite acessar e manipular facilmente qualquer dado, como se estivéssemos consultando uma tabela com posições fixas — ideal para organizar dados ambientais no nosso curso.

```
1 regioes = ["Xingu", "Marajó", "Tapajós", "Xingu"]
```

```
1 # acessando pelo índice  
2 print(regioes[0]) # Saída: Xingu
```

```
Xingu
```

```
1 # acessando pelo índice  
2 print(regioes[2]) # Saída: Tapajós
```

```
Tapajós
```

## :: Mutável

**Listas em Python são mutáveis**, ou seja, após serem criadas, é possível **alterar, adicionar ou remover itens** livremente. Essa flexibilidade é muito útil quando trabalhamos com dados dinâmicos — como informações ambientais que mudam constantemente durante o monitoramento da Amazônia.

## :: Permitir Duplicatas

Como as listas são indexadas, elas podem conter itens repetidos. Isso significa que é possível ter vários elementos com o mesmo valor, cada um ocupando uma posição diferente na lista — algo útil quando queremos registrar eventos que ocorrem mais de uma vez, como alertas recorrentes em uma mesma região.

```
1 alertas = ["Xingu", "Tapajós", "Xingu", "Marajó", "Xingu"]
```

```
1 print(alertas)
```

```
['Xingu', 'Tapajós', 'Xingu', 'Marajó', 'Xingu']
```

## :: Tamanho da Lista

Para saber quantos itens existem em uma lista, utilizamos a função `len()`. Ela retorna o tamanho total da lista, ou seja, a quantidade de elementos armazenados — útil para análises e controle de dados em monitoramentos ambientais.

```
1 regioes = ["Xingu", "Marajó", "Tapajós", "Javari"]
2 quantidade = len(regioes)
3 print(f"🌐 Total de regiões monitoradas: {quantidade}")
```

```
🔄 🌐 Total de regiões monitoradas: 4
```

## 📄 :: O Construtor `list()`

Também é possível criar uma nova lista utilizando o construtor `list()`. Essa forma é especialmente útil para transformar outros tipos de dados — como tuplas ou strings — em listas, tornando o conteúdo modificável e pronto para análise ou processamento.

```
1 regioes_tuple = ("Xingu", "Marajó", "Tapajós")
2 regioes_lista = list(regioes_tuple)
3
4 print("Lista de regiões monitoradas:", regioes_lista)
```

```
🔄 Lista de regiões monitoradas: ['Xingu', 'Marajó', 'Tapajós']
```

## 📄 :: Acesso aos Itens

```
1 # acesso por índices
2 regioes = ["Xingu", "Marajó", "Tapajós", "Javari"]
3
4 print("Primeira região:", regioes[0])
5 print("Segunda região:", regioes[1])
6 print("Última região:", regioes[3])
```

```
🔄 Primeira região: Xingu
Segunda região: Marajó
Última região: Javari
```

```
1 # acesso por índice negativo
2 regioes = ["Xingu", "Marajó", "Tapajós", "Javari"]
3
4 print("Última região:", regioes[-1])
5 print("Penúltima região:", regioes[-2])
```

```
🔄 Última região: Javari
Penúltima região: Tapajós
```

```
1 # por faixa de índices
2 regioes = ["Xingu", "Marajó", "Tapajós", "Javari"]
3
4 print("Regiões da Marajó até Javari:", regioes[1:4])
```

```
🔄 Regiões da Marajó até Javari: ['Marajó', 'Tapajós', 'Javari']
```

```
1 # por faixa de índices
2 regioes = ["Xingu", "Marajó", "Tapajós", "Javari"]
3
4 print("Regiões da Marajó até Javari:", regioes[1:])
```

```
🔄 Regiões da Marajó até Javari: ['Marajó', 'Tapajós', 'Javari']
```

## 📄 :: Verifica Se Item Existe

```
1 regioes = ["Xingu", "Marajó", "Tapajós", "Javari"]
2
3 if "Xingu" in regioes:
4     print("✅ Região Xingu está sendo monitorada.")
5 else:
6     print("⚠️ Região Xingu não está na lista de monitoramento.")
```

```
🔄 ✅ Região Xingu está sendo monitorada.
```

## 📄 :: Altera Valor do Item

```
1 # Altera pelo valor do índice
2
3 # Lista original de regiões monitoradas
4 regioes = ["Xingu", "Marajó", "Tapajós", "Javari"]
5
6 # Atualizando o nome da segunda região (índice 1)
7 regioes[1] = "Baixo Amazonas"
8
9 # Exibindo a lista atualizada
10 print("📌 Lista atualizada de regiões:", regioes)
```

```
🔄 📌 Lista atualizada de regiões: ['Xingu', 'Baixo Amazonas', 'Tapajós', 'Javari']
```

```
1 # Altera pelo intervalo de índices
2
```

```
3 # Lista original de regiões monitoradas
4 regioes = ["Xingu", "Marajó", "Tapajós", "Javari", "Purús"]
5
6 # Substituindo os itens nos índices 1 a 3 (sem incluir o 4)
7 regioes[1:4] = ["Baixo Amazonas", "Madeira", "Alto Solimões"]
8
9 # Exibindo a lista atualizada
10 print("🔴 Lista atualizada de regiões:", regioes)
```

🔴 Lista atualizada de regiões: ['Xingu', 'Baixo Amazonas', 'Madeira', 'Alto Solimões', 'Purús']

## 🔴 :: Inserir Item

Para adicionar um novo item em uma lista sem substituir os valores existentes, utilizamos o **método insert()**. Esse método permite inserir um item em uma posição específica, movendo os demais elementos para a frente, sem apagá-los. Isso é útil quando queremos organizar melhor os dados na lista.

```
1 # Lista original de regiões monitoradas
2 regioes = ["Marajó", "Tapajós", "Javari"]
3
4 # Inserindo "Xingu" na posição 0 (início da lista)
5 regioes.insert(0, "Xingu")
6
7 # Exibindo a lista atualizada
8 print("🔴 Regiões monitoradas:", regioes)
```

🔴 Regiões monitoradas: ['Xingu', 'Marajó', 'Tapajós', 'Javari']

## 🔴 :: Adiciona Itens na Lista

Para adicionar um item no final de uma lista, usamos o **método append()**. Ele é simples e direto, ideal quando queremos incluir novos dados ao final da lista existente, como uma nova região monitorada ou um alerta recente.

```
1 # Lista inicial de regiões monitoradas
2 regioes = ["Xingu", "Marajó", "Tapajós"]
3
4 # Adicionando uma nova região ao final da lista
5 regioes.append("Javari")
6
7 # Exibindo a lista atualizada
8 print("🔴 Regiões monitoradas:", regioes)
```

🔴 Regiões monitoradas: ['Xingu', 'Marajó', 'Tapajós', 'Javari']

## 🔴 :: Estende uma Lista

Para adicionar os elementos de outra lista à lista atual, utilizamos o **método extend()**. Ele permite unir duas listas, acrescentando todos os itens da segunda lista ao final da primeira, sem criar uma nova estrutura. Ideal para combinar dados ambientais de diferentes fontes.

```
1 # Lista principal de regiões já monitoradas
2 regioes_principais = ["Xingu", "Marajó"]
3
4 # Lista com novas regiões a serem adicionadas
5 novas_regioes = ["Tapajós", "Javari"]
6
7 # Unindo as listas
8 regioes_principais.extend(novas_regioes)
9
10 # Exibindo a lista final
11 print("🔴 Regiões monitoradas:", regioes_principais)
```

🔴 Regiões monitoradas: ['Xingu', 'Marajó', 'Tapajós', 'Javari']

## 🔴 :: Remove Item Específico da Lista

Para remover um item específico de uma lista, usamos o **método remove()**. Ele localiza o primeiro valor correspondente e o elimina da lista. Esse método é útil quando queremos excluir um dado exato – como uma região que não está mais sob monitoramento.

```
1 # Lista de regiões monitoradas
2 regioes = ["Xingu", "Marajó", "Tapajós", "Javari"]
3
4 # Removendo a região "Tapajós"
5 regioes.remove("Tapajós")
6
7 # Exibindo a lista atualizada
8 print("🔴 Regiões monitoradas:", regioes)
```

🔴 Regiões monitoradas: ['Xingu', 'Marajó', 'Javari']



## ⌵ :: Remove Índice Especificado

O **método pop()** é usado para remover um item de uma lista com base no índice informado. Ele também retorna o valor removido, o que permite usá-lo em outras partes do código — útil quando queremos registrar ou analisar o que foi excluído.

```
1 # Lista de regiões monitoradas
2 regioes = ["Xingu", "Marajó", "Tapajós", "Javari"]
3
4 # Removendo a região na posição 2 (índice 2)
5 regioao_removida = regioes.pop(2)
6
7 # Exibindo o item removido e a lista atualizada
8 print(f"⚠️ Região removida: {regiao_removida}")
9 print("📍 Npvas regiões monitoradas:", regioes)
```

```
📄 ⚠️ Região removida: Tapajós
📍 Npvas regiões monitoradas: ['Xingu', 'Marajó', 'Javari']
```

```
1 # remove usando o método del()
2
3 # Lista de regiões monitoradas
4 regioes = ["Xingu", "Marajó", "Tapajós", "Javari"]
5
6 # Removendo a segunda região (índice 1)
7 del regioes[1]
8
9 print("📍 Lista após remoção:", regioes)
```

```
📄 📍 Lista após remoção: ['Xingu', 'Tapajós', 'Javari']
```

## ⌵ :: Limpe uma Lista

O **método clear()** é usado para esvaziar completamente uma lista, removendo todos os seus itens. A lista continua existindo na memória, mas fica vazia, pronta para receber novos dados, se necessário. Ideal para reiniciar o monitoramento sem apagar a estrutura.

```
1 # Lista de regiões monitoradas durante a operação
2 regioes = ["Xingu", "Marajó", "Tapajós", "Javari"]
3
4 # Encerrando operação e limpando a lista
5 regioes.clear()
6
7 # Exibindo o resultado
8 print("📍 Regiões monitoradas após limpeza:", regioes)
```

```
📄 📍 Regiões monitoradas após limpeza: []
```

## ⌵ :: Percorre uma Lista

Você pode percorrer todos os itens de uma lista utilizando um **laço for**. Esse tipo de laço permite acessar cada elemento da lista, um por um, facilitando análises, exibições ou processamentos automatizados de dados.

```
1 # Lista de rios da região amazônica
2 rios = ["Rio Amazonas", "Rio Negro", "Rio Tapajós", "Rio Xingu"]
3
4 # Usando for para exibir cada rio
5 for rio in rios:
6     print(f"🌊 Rio monitorado: {rio}")
```

```
📄 🌊 Rio monitorado: Rio Amazonas
🌊 Rio monitorado: Rio Negro
🌊 Rio monitorado: Rio Tapajós
🌊 Rio monitorado: Rio Xingu
```

```
1 # Lista de valores de risco por região (em porcentagem)
2 riscos = [20, 55, 80, 35]
3
4 # Lista correspondente com os nomes das regiões
5 regioes = ["Xingu", "Marajó", "Tapajós", "Javari"]
6
7 # Percorrendo ambas as listas ao mesmo tempo
8 for regioao, risco in zip(regioes, riscos):
9     if risco > 70:
10         status = "🔴 Alto risco"
11     elif risco > 40:
12         status = "🟡 Risco moderado"
13     else:
14         status = "🟢 Baixo risco"
15
16     print(f"{regiao}: {status} ({risco}%)")
```

```
📄 Xingu: 🟢 Baixo risco (20%)
Marajó: 🟡 Risco moderado (55%)
Tapajós: 🔴 Alto risco (80%)
Javari: 🟢 Baixo risco (35%)
```

## ⌵ :: List Comprehension

List comprehension é uma forma mais curta e elegante de criar uma nova lista com base nos valores de uma lista existente, aplicando filtros ou transformações em uma única linha de código.

Por exemplo, se você tem uma lista de frutas e deseja criar uma nova lista contendo apenas as frutas que possuem a letra "a", normalmente usaria um for com uma condição. Com list comprehension, esse processo se torna mais direto e legível.

```
nova_lista = [expressão for item in lista_original if condição]
```

### Componentes:

expressão: o que você quer colocar na nova lista (pode ser o próprio item ou uma transformação dele).

for item in lista\_original: percorre cada item da lista original.

if condição: (opcional) filtra os itens que serão incluídos.

```
1 # Selecionar regiões da Amazônia que possuem a letra "a"
2 # Forma Tradicional
3
4 regioes = ["Xingu", "Marajó", "Tapajós", "Javari", "Purús"]
5 regioes_com_a = []
6
7 for regioao in regioes:
8     if "a" in regioao.lower():
9         regioes_com_a.append(regiao)
10
11 print("🌿 Regiões com a letra 'a':", regioes_com_a)
```

```
🔄 🌿 Regiões com a letra 'a': ['Marajó', 'Tapajós', 'Javari']
```

```
1 # Selecionar regiões da Amazônia que possuem a letra "a"
2 # Usando List Comprehension
3 regioes = ["Xingu", "Marajó", "Tapajós", "Javari", "Purús"]
4 regioes_com_a = [regiao for regioao in regioes if "a" in regioao.lower()]
5
6 print("🌿 Regiões com a letra 'a':", regioes_com_a)
```

```
🔄 🌿 Regiões com a letra 'a': ['Marajó', 'Tapajós', 'Javari']
```

## ⌵ :: Classifica Lista Alfanumericamente

As listas em Python possuem o **método sort()**, que permite classificar os itens automaticamente em ordem alfanumérica crescente (do menor para o maior, ou de A a Z, por padrão). Esse método modifica a lista original.

```
1 # Classifique a lista em ordem alfabética
2 regioes = ["Xingu", "Marajó", "Tapajós", "Javari", "Purús"]
3 regioes.sort()
4
5 print("🌿 Regiões classificadas:", regioes)
```

```
🔄 🌿 Regiões classificadas: ['Javari', 'Marajó', 'Purús', 'Tapajós', 'Xingu']
```

```
1 # Classifique a lista numericamente
2 numeros = [3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5]
3 numeros.sort()
4
5 print("🔢 Números classificados:", numeros)
```

```
🔄 🔢 Números classificados: [1, 1, 2, 3, 3, 4, 5, 5, 5, 6, 9]
```

```
1 # Ordenar em ordem decrescente
2 regioes = ["Xingu", "Marajó", "Tapajós", "Javari", "Purús"]
3 regioes.sort(reverse=True)
4
5 print("🌿 Regiões classificadas em ordem decrescente:", regioes)
```

```
🔄 🌿 Regiões classificadas em ordem decrescente: ['Xingu', 'Tapajós', 'Purús', 'Marajó', 'Javari']
```

## ⌵ :: Personalizar a Função de Classificação

Você também pode personalizar a forma como uma lista é classificada usando o argumento **key = função**. Essa função define a lógica de ordenação, retornando um valor que será usado como base para a comparação — quanto menor o valor retornado, mais cedo o item aparecerá na lista.

```
1 # classificar regiões da Amazônia com base no número de letras no nome, do mais curto para o mais longo
2 regioes = ["Xingu", "Marajó", "Tapajós", "Javari", "Purús"]
3 regioes.sort(key=len)
4
5 print("🌿 Regiões classificadas por tamanho:", regioes)
```

```
🔄 🌿 Regiões classificadas por tamanho: ['Xingu', 'Purús', 'Marajó', 'Javari', 'Tapajós']
```

**lambda** é uma função anônima em Python — ou seja, uma função sem nome — usada quando você precisa de uma função simples e rápida, geralmente com uma única linha de código.

```
1 dobro = lambda x: x * 2
2 print(dobro(5)) # Saída: 10
```

↗ 10

```
1 # Temos uma lista de regiões com seus respectivos níveis de risco.
2 # Ordená-las do menor para o maior risco
3
4 dados = [
5     ("Xingu", 75),
6     ("Tapajós", 40),
7     ("Marajó", 90),
8     ("Javari", 55)
9 ]
10
11 # Ordenando com base no valor de risco (índice 1 da tupla)
12 dados.sort(key=lambda item: item[1])
13
14 # Exibindo resultado
15 for regioao, risco in dados:
16     print(f"{regiao}: {risco}% de risco")
```

↗ Tapajós: 40% de risco  
Javari: 55% de risco  
Xingu: 75% de risco  
Marajó: 90% de risco

## ▼ ≡ Atividade: calcular média móvel de área desmatada

Nos últimos anos, o avanço do desmatamento na Amazônia tem gerado grande preocupação entre cientistas, ambientalistas e comunidades locais. A variação no ritmo do desmatamento pode indicar pressões econômicas, ausência de fiscalização ou mudanças no uso da terra. Por isso, é essencial acompanhar os dados de forma contínua e identificar tendências com clareza. Uma das formas mais utilizadas para esse acompanhamento é o **cálculo da média móvel**, que suaviza as flutuações diárias e permite visualizar o comportamento real ao longo do tempo.

Imagine que você está atuando como parte de uma equipe de inteligência ambiental em uma ONG que monitora áreas críticas da floresta. Você recebeu dados semanais sobre a área desmatada (em km<sup>2</sup>) em uma região específica da Amazônia ao longo de 10 semanas. O desafio agora é aplicar um modelo simples de **média móvel de 3 períodos** para entender a evolução do desmatamento e identificar se há tendência de alta, estabilidade ou queda.

Essa técnica é extremamente útil para **tomada de decisão rápida e baseada em dados**, além de ser utilizada como base para acionar alertas, organizar expedições de fiscalização ou apoiar políticas públicas. Neste exercício, você vai aplicar na prática um recurso de ciência de dados que ajuda a transformar dados ambientais em inteligência acionável.

## ▼ ≡ Desafio

Você recebeu os dados semanais de desmatamento (em km<sup>2</sup>) de uma região da Amazônia durante 10 semanas. Sua missão é ajudar a equipe de monitoramento ambiental a entender se o desmatamento está aumentando, diminuindo ou se mantendo estável ao longo do tempo. Para isso, siga os passos abaixo:

1. Utilize a seguinte lista com os dados de desmatamento:

```
desmatamento_semanal = [12.4, 15.1, 13.7, 17.3, 20.5, 18.9, 22.1, 19.4, 16.8, 14.2]
```

2. Crie uma função em Python que calcule a média móvel de 3 períodos com base nesses dados.

3. Mostre:

- A lista original;
- A lista com os valores da média móvel;
- (Opcional) Indique a semana onde houve a maior queda ou o maior aumento na média.

**Lembre-se:** a média móvel é uma ferramenta importante para suavizar variações bruscas e ajudar na tomada de decisão ambiental com base em dados reais.

Mostrar código

Mostrar código

[Mostrar código](#)[Mostrar saída oculta](#)[Mostrar código](#)[Mostrar código](#)[Mostrar saída oculta](#)