Relatório 2 – Prática: Linguagem de Programação Python (I)

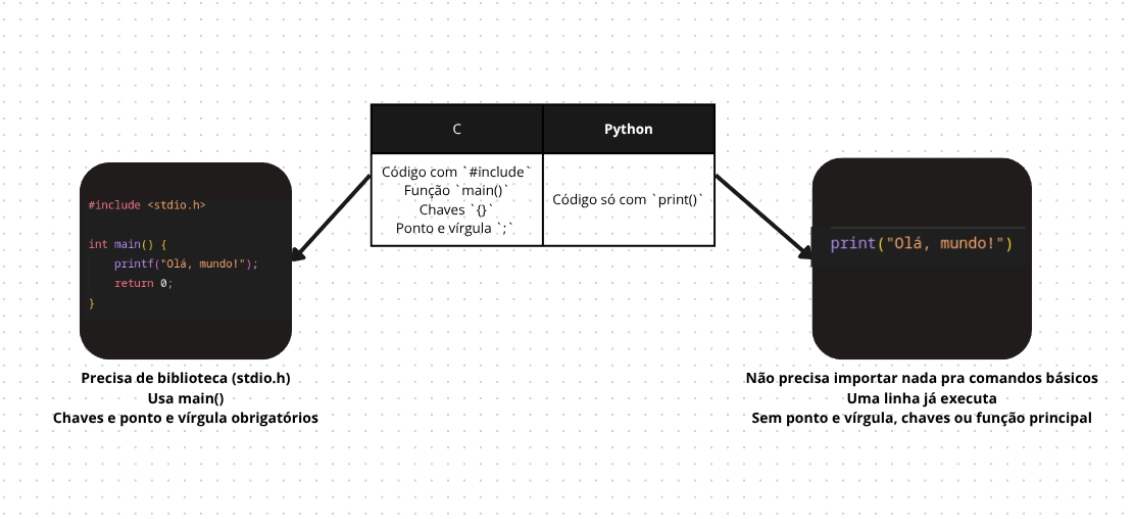
Ronny Gabryel Colatino de Souza

Descrição da atividade

Sintaxe e Primeiros Comandos

O que mais me chamou atenção logo de cara foi como a sintaxe do Python é limpa. Vindo do C, ver que um print pode ser feito com uma linha sem ponto e vírgula já mostra a diferença de proposta entre as linguagens.

Em C, só pra mostrar isso na tela, é necessário biblioteca, main, ponto e vírgula. No Python, é direto. Outra coisa é que Python não usa {}, usa **indentação obrigatória**, o que deixa o código mais legível, mas exige cuidado com os espaços.

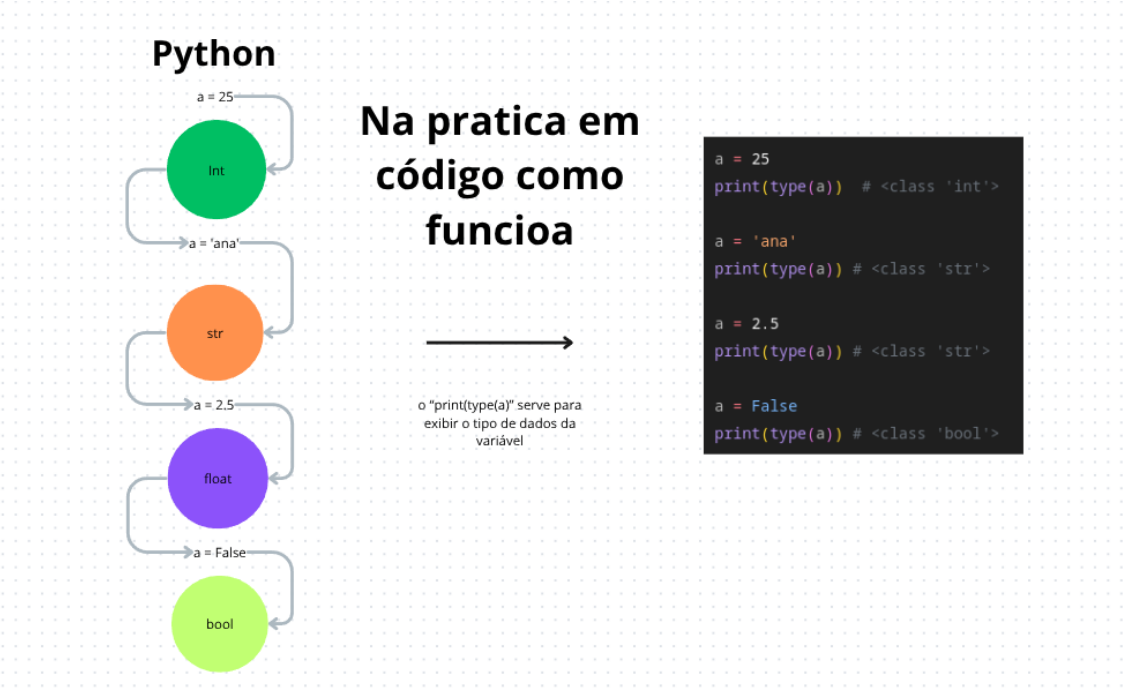


Tipos de Variáveis e Tipagem Dinâmica

Em C, você precisa declarar o tipo da variável antes de usá-la. Em Python, você só escreve:

```
idade = 25
print(type(idade))  # <class 'int'>
```

Python é dinamicamente tipado, então o tipo é definido na hora da execução.
Dá pra trocar o valor por uma string, por exemplo:



E não dá erro isso dá flexibilidade, mas pode causar confusão se não for controlado.

Coleções: listas, tuplas, Conjuntos, dicionario

Python tem várias coleções integradas. Aqui vai um resumo com exemplos:

Listas:

Listas são ordenadas, mutáveis, e aceitam tipos diferentes.

```
nomes = ["Ana", "Bruno", "Carlos"]
nomes.append("Daniel")
print(nomes[0])  # Ana
```

São úteis quando você não quer que os valores sejam alterados durante a execução. Aqui temos uma lista chamada `nomes` que armazena strings. Usamos o método `append()` para adicionar um novo nome ao final da lista. Depois, acessamos o primeiro elemento com `nomes[0]`, que imprime "Ana".

Tuplas:

```python
valores = (10, 20, 30)
print(valores[1])  # 20
```

Diferente das listas, as tuplas são imutáveis. Isso significa que, depois de criadas, seus valores não podem ser alterados. Aqui estamos acessando o segundo valor (índice 1), que é 20.

Conjuntos:

```python
numeros = {1, 2, 2, 3}
print(numeros)  # {1, 2, 3}
```

Os conjuntos (tipo `set`) eliminam valores duplicados automaticamente. Mesmo que o número 2 apareça duas vezes, ele só será armazenado uma vez. Além disso, os elementos não seguem uma ordem específica.

Dicionários:

```python
pessoa = {"nome": "Ana", "idade": 22}
print(pessoa["idade"])  # 22
```

Dicionários são estruturas que armazenam dados em pares `chave: valor`. Aqui, temos um dicionário chamado `pessoa` com duas chaves: `nome` e `idade`. Podemos acessar o valor de `idade` diretamente pela chave.

Tipo	Ordem	Mutável	Repetição	Exemplo
Lista	SIM	SIM	SIM	[1,2,3]
Tuplas	SIM	NÃO	SIM	(1,2,3)
Conjuntos	NÃO	SIM	NÃO	{1,2,3}
Dicionario	SIM	SIM	Chave única	{"chave": "valor"}

Essa tabela resume bem as principais características das coleções em Python. Ela compara listas, tuplas, conjuntos e dicionários com base em quatro critérios: se mantêm a ordem, se são mutáveis, se permitem repetições e dá um exemplo de como cada uma é declarada. É uma forma rápida de visualizar as diferenças entre elas e lembrar quando usar cada tipo.

Operadores

Aritméticos:

```python
a = 10
b = 5

print(a + b)  # Soma: resultado é 15
print(a - b)  # Subtração: resultado é 5
print(a * b)  # Multiplicação: resultado é 50
print(a / b)  # Divisão: resultado é 2.0 (float)
print(a % b)  # Módulo: resultado é 0, pois 10 é divisível por 5
print(a ** b) # Exponenciação: 10 elevado a 5 = 100000
```

Esses são os operadores aritméticos básicos em Python. Eles permitem realizar cálculos matemáticos simples e são muito utilizados no dia a dia da programação, seja em sistemas, jogos ou automações.

Relacionais:

```python
a = 10
b = 5

print(a > b)   # True, pois 10 é maior que 5
print(a < b)   # False, 10 não é menor que 5
print(a == b)  # False, os valores são diferentes
print(a != b)  # True, pois 10 é diferente de 5
print(a >= b)  # True, 10 é maior ou igual a 5
print(a <= b)  # False, 10 não é menor nem igual a 5
```

Os operadores relacionais comparam valores e retornam `True` ou `False`. Eles são usados para tomadas de decisão em estruturas como `if`.

Lógicos:

```python
x = True
y = False

print(x and y)  # False, porque só é True se os dois forem True
print(x or y)   # True, porque ao menos um é True
print(not x)    # False, inverte o valor de x (que era True)
```

Operadores lógicos combinam expressões booleanas. São fundamentais em condições mais complexas e permitem criar regras com mais de um critério.

Ternário:

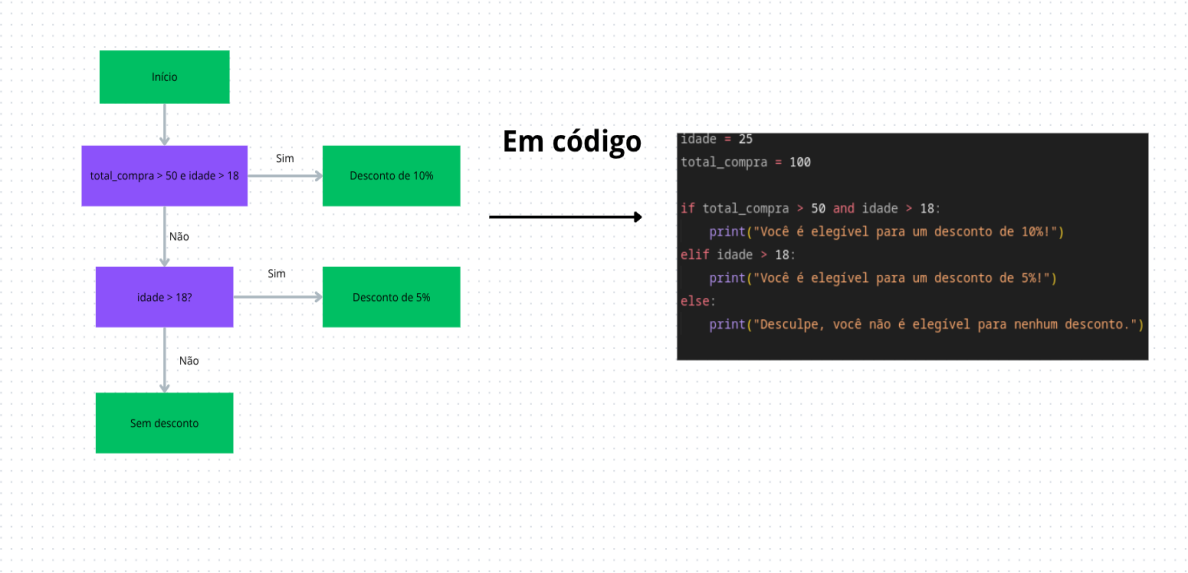
```python
idade = 18
status = "maior" if idade >= 18 else "menor"
print(status)  # maior
```

O operador ternário é uma forma curta de escrever um `if/else`. Ideal para condições simples em uma única linha. Aqui, se a idade for 18 ou mais, o status será “maior”; senão, será “menor”.

Estruturas de Controle

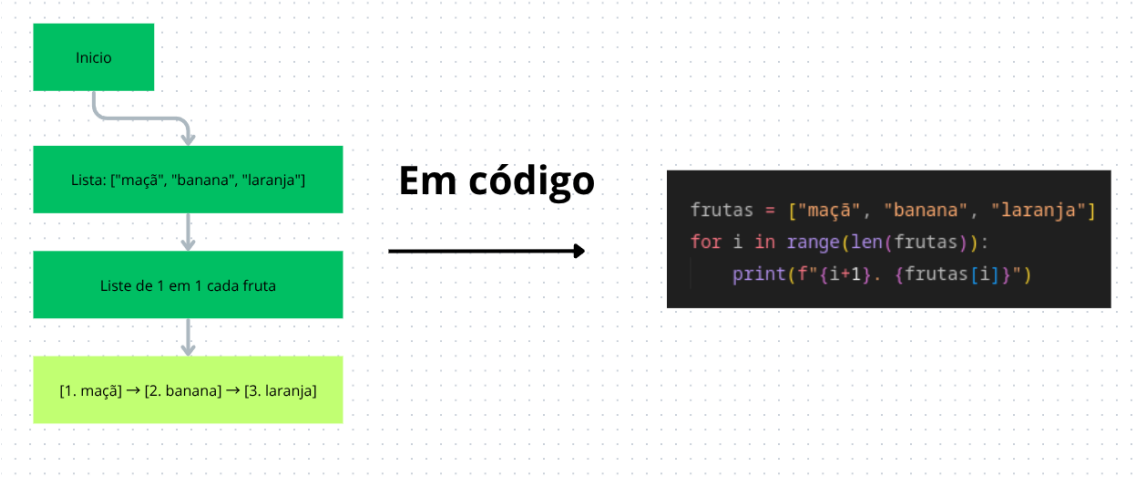
if, elif, eles:

if, elif, eles é muito útil tipo.



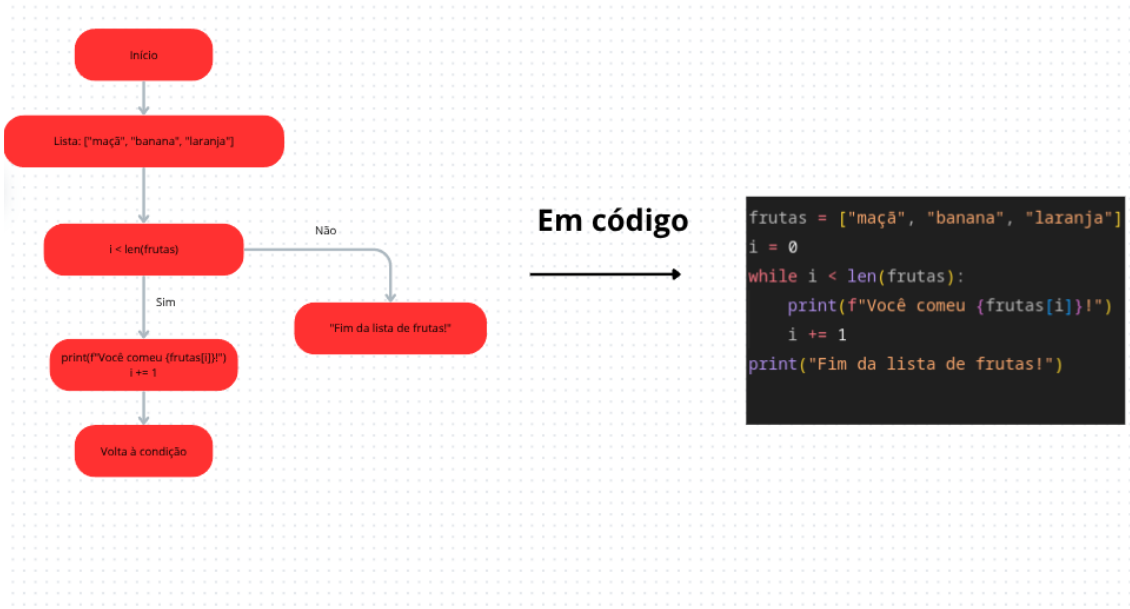
A estrutura `if, elif, else` permite criar decisões com vária possibilidades. No exemplo do fluxograma, o código simula um sistema de descontos para um cliente. Se a compra for maior que R$50 e a idade for igual a 18, o cliente ganha 10%. Se for só maior que 50, o desconto é de 5%. Caso contrário, não há desconto. Essa estrutura é útil quando temos mais de uma condição para avaliar e queremos respostas específicas para cada caso.

for:



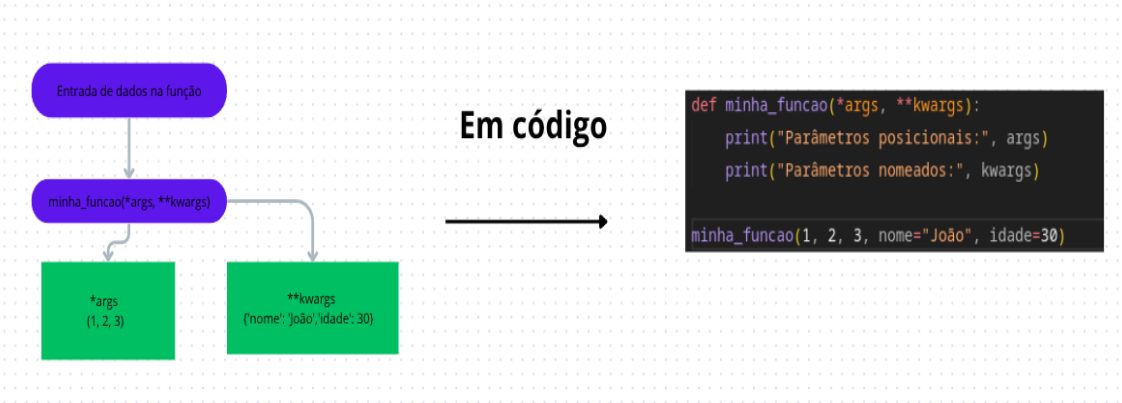
O laço `for` é usado para percorrer coleções, como listas. No exemplo, temos uma lista com frutas. O laço percorre essa lista e imprime cada fruta com um número correspondente. Isso é muito útil quando precisamos acessar ou modificar cada item de uma coleção de forma organizada e repetitiva.

while:



O laço `while` executa um bloco de código enquanto uma condição for verdadeira. No exemplo do fluxograma, ele percorre a lista de frutas usando um contador `i`. A cada iteração, imprime uma fruta e incrementa `i`. Quando `i` atinge o tamanho da lista, o laço para. Esse tipo de laço é útil quando você não sabe exatamente quantas vezes vai repetir uma ação, mas tem uma condição que controla a parada.

Funções, args, kwargs

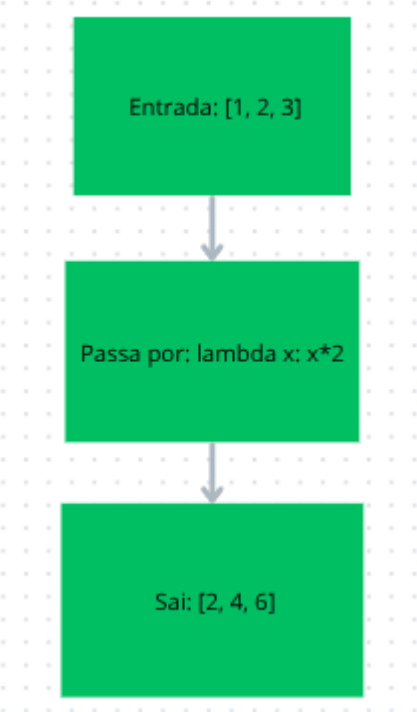


O uso de `*args` e `**kwargs` permite criar funções que recebem múltiplos argumentos. `*args` aceita uma quantidade indefinida de argumentos posicionais (como uma lista), enquanto `**kwargs` recebe argumentos nomeados (como um dicionário). No exemplo, a função imprime separadamente os valores passados por posição e por nome. Isso dá flexibilidade para funções que podem receber dados variados.

Lambda, map, reduce

```python
# lambda
soma = lambda a, b: a + b
print(soma(3, 4))  # 7

# map
nums = [1, 2, 3]
print(list(map(lambda x: x * 2, nums)))  # [2, 4, 6]
```



Em Python, usar funções lambda junto com `map` é uma forma prática e direta de lidar com listas e transformar dados. A `lambda` serve para criar pequenas funções "de uma linha", sem precisar escrever tudo com `def`. Por exemplo, `soma = lambda a, b: a + b` define uma função que simplesmente soma dois números. Se você fizer `soma(3, 4)`, o resultado será 7.

Já a função `map` entra em cena quando você quer aplicar uma mesma operação a todos os itens de uma lista, sem ter que escrever um loop manualmente. Imagine que você tem uma lista como `nums = [1, 2, 3]`. Se quiser dobrar cada número, pode usar `map(lambda x: x * 2, nums)`. Isso aplica a função `lambda x: x * 2` em cada elemento da lista, resultando em `[2, 4, 6]`.

Essa combinação de `lambda` e `map` ajuda a escrever código mais limpo e direto, especialmente em tarefas de transformação de dados.

Conclusões

As aulas me ajudaram a entender como Python é objetivo e limpo comparado ao C. Acho que o que mais me chamou atenção foram as coleções e a forma de escrever funções.

Também gostei de ver como dá pra aplicar lambda e map em listas sem complicação. Senti que agora consigo fazer códigos mais rápidos, sem ficar travado em sintaxe.

Referencias

COD3R. Curso de Python para Iniciantes – Aula 1
https://www.youtube.com/watch?v=oUrBHiT-lzo

COD3R. Curso de Python para Iniciantes – Aula 2
https://www.youtube.com/watch?v=iq7JLIH-sV0