Introdução às Listas: Armazenamento com Tuplas

Listas – Objetivos da aula

- Reconhecer comportamento esperado de programas simples com base em requisitos.
- Armazenar dados compostos em listas e tuplas.
- Manipular e acessar informações com indexação.
- Testar integridade de dados usando assert.
- Aplicar TDD no desenvolvimento de pequenas funcionalidades.

Por que usar listas e tuplas Listas

- **Listas**: coleções mutáveis de itens.
- **Tuplas**: coleções imutáveis, ideais para pares de informações como (nome, valor).
- Permitem organizar dados de forma estruturada.

Diferença entre listas e tuplas

- Listas podem ser alteradas, tuplas não.
- Escolher depende da necessidade de segurança vs. flexibilidade.

```
lista=[1, 2, 3]   # mutável
tupla=(1, 2, 3)   # imutável
```

Diferença entre listas e tuplas

- Listas e tuplas são estruturas de dados em Python que armazenam coleções de elementos, mas diferem em mutabilidade e uso.

- As listas, definidas com colchetes [ ], são mutáveis, permitindo adicionar, remover ou alterar elementos, enquanto as tuplas, definidas com parênteses ( ), são imutáveis, garantindo que os dados permaneçam constantes.

Criando uma lista de tuplas

- Cada tupla armazena nome e valor do produto.
- Mantém informação agrupada e consistente.

```
compras = []
compras.append(("Caneta", 10))
compras.append(("Caderno", 20))
```

Acessando elementos

- Podemos desempacotar tuplas para acessar valores.
- Facilita leitura e manipulação dos dados.

```
nome, valor = compras[0]
assert nome == "Caneta"
assert valor == 10
```

Indexação

- Listas suportam indexação direta.
- Permite pegar qualquer produto sem percorrer todos os elementos.

```
assert compras[1][0] == "Caderno"
assert compras[1][1] == 20
```

Loop while para inserir dados

- Permite inserção contínua de produtos.
- Controlado por sentinela (s/n).

```
compras = []
continuar = "s"
while continuar == "s":
    produto = input("Nome do produto: ")
    valor = int(input("Valor: "))
    compras.append((produto, valor))
    continuar = input("Deseja continuar? (s/n) ")
```

Testando a inserção com assert

- Testa que os dados inseridos estão corretos.
- Primeira verificação de integridade de dados.

```
compras = [("Caneta", 10)]
assert compras[0] == ("Caneta", 10)
```

Inserção múltipla simulada

- Simulando uma inserção de múltiplas entradas.
- Permite testes automatizados com **assert**.

```python
def test_adicionar_compras():
    compras = []
    entradas = [("Caneta", 10), ("Caderno", 20)]
    compras = adicionar_compras(compras, entradas)
    assert len(compras) == 2
    assert compras[0] == ("Caneta", 10)
    assert compras[1] == ("Caderno", 20)
```

```python
def adicionar_compras(compras, entradas):
    i = 0
    while i < len(entradas):
        compras.append(entradas[i])
        i += 1
    return compras
```

Validando tipos de dados

- Garante que cada tupla mantém tipo correto.
- Evita erros posteriores no programa.

```
nome, valor = compras[0]
assert isinstance(nome, str)
assert isinstance(valor, int)
```

Estrutura de decisão ao inserir

- Permite filtrar entradas inválidas antes de armazenar.
- Pode ser testado com **with** pytest.raises(ValueError).

```python
def adicionar_compra(compras, produto, valor):
    if valor < 0:
        raise ValueError("Valor inválido")
    else:
        compras.append((produto, valor))
    return compras
```

Estrutura de decisão ao inserir

```python
import pytest

def test_adicionar_compra():
    compras = []

    # Valor positivo adiciona normalmente
    compras = adicionar_compra( compras, "Caneta", 10)
    assert compras[-1] == ("Caneta", 10)

    # Valor negativo gera ValueError
    with pytest.raises(ValueError):
        adicionar_compra(compras, "Caderno", -5)
```

Acessando o último item

- Indexação negativa pega o último elemento da lista.

```
ultimo = compras[-1]
assert ultimo == ("Caderno", 20)
```

Atualizando valor

- Listas são mutáveis, então podemos atualizar tuplas inteiras.
- Mantém integridade do par (nome, valor).

```
produto, valor = compras[0]
compras[0] = (produto, 15)
assert compras[0][1] == 15
```

Verificando existência

- Podemos testar se um item existe na lista.
- Útil para validação de dados antes de operar.

```
assert ( "Caneta", 15 ) in compras
```

Contando itens

- **len()** retorna quantos produtos estão cadastrados.
- Pode ser usado para validar inserções.

```
total = len(compras)
assert total == 2
```

Soma dos valores

- Calcula total de pontos usando while.

```python
def calcular_total(compras):
    total_valor = 0
    i = 0
    while i < len(compras):
        total_valor += compras[i][1]
        i += 1
    return total_valor
```

```python
def test_calcular_total():
    compras = [("Caneta", 10), ("Caderno", 25)]
    assert calcular_total(compras) == 35
```

Decisão condicional sobre valores

- Permite categorizar produtos com if/else.
- Funciona dentro de tuplas e listas.

```
nome, valor = compras[0]
if valor > 20:
    mensagem = "Compra alta"
else:
    mensagem = "Compra baixa"
assert mensagem == "Compra baixa"
```

Extraindo nomes

- Podemos criar listas auxiliares somente com nomes.
- Útil para exibição ou filtros.

```
nomes = []
i = 0
while i < len(compras):
    nomes.append(compras[i][0])
    i += 1
assert nomes == ["Caneta", "Caderno"]
```

Extraindo valores

- Similar ao slide anterior, mas com valores.

```
valores = []
i = 0
while i < len(compras):
    valores.append(compras[i][1])
    i += 1
assert valores == [15, 20]
```

Função de teste de integridade

- O teste abaixo verifica a integridade de cada tupla na lista de compras.
- Garante que os tipos de dados estejam corretos antes de outras operações.

```python
def test_integridade_compras():
    compras = [("Caneta", 10), ("Caderno", 25)]
    i = 0
    while i < len(compras):
        nome, valor = compras[i]
        assert isinstance(nome, str)   # garante que o nome é string
        assert isinstance(valor, int)  # garante que o valor é inteiro
        i += 1
```

Mensagem dinâmica

- Tupla e formatação de strings:

```
nome, valor = compras[0]
mensagem = f"O produto {nome} vale {valor} pontos"
assert mensagem == "O produto Caneta vale 15 pontos"
```

While + Sentinela

- Repetição controlada combinando while e sentinela:

```
compras = []
continuar = "s"
i = 0
while continuar == "s" and i < 2:
    compras.append(("Produto", i+1))
    i += 1
    continuar = "n"
assert len(compras) == 2
```

Evitando duplicatas

- Mantém integridade evitando repetições:

```
produto_novo = ( "Caneta" , 10)
if produto_novo not in compras:
    compras.append(produto_novo)
assert compras.count(produto_novo) == 1
```

Atualizando produto existente

- Permite modificar valores de produtos já cadastrados.

```
produto, valor = "Caneta", 20
i = 0
while i < len(compras):
    if compras[i][0] == produto:
        compras[i] = (produto, valor)
    i += 1
assert compras[0][1] == 20
```

Exibição de produtos

- Mostra como percorrer a lista de tuplas com while.
- Soma pontos de todas as compras e exibe.

```
i = 0
while i < len(compras):
    print(f"{compras[i][0]} - {compras[i][1]} pontos")
    i += 1
```

```
total = 0
i = 0
while i < len(compras):
    total += compras[i][1]
    i += 1
print(f"Total de pontos: {total}")
```

Encerramento

- Você aprendeu:
  - Armazenar produtos e valores em lista de tuplas
  - Usar **.append()** para inserir dados dinamicamente
  - Acessar dados via indexação
  - Controlar repetição de entradas com **while** e **sentinela**
  - Validar dados com **assert** e **with** pytest.raises
  - Aplicar **TDD** no cadastro de produtos do AraCoins

PUC Minas