Estrutura de Dados: Dicionários

Módulo 1: O Essencial: A Estrutura por Trás da Velocidade

1.1 O Que São Dicionários? Uma Analogia Simples

Imagine uma **caixa de arquivos** em um escritório. Cada documento está em uma pasta com uma **etiqueta** com um nome único. Para encontrar um documento, você não precisa revirar a caixa inteira; basta procurar pela etiqueta.

Os dicionários em Python funcionam exatamente assim. Eles armazenam dados em pares de **chave** e **valor**.

- A **chave** é a etiqueta. Ela é um identificador único para cada item.
- O **valor** é o conteúdo da pasta. Pode ser qualquer tipo de informação.

A grande vantagem dos dicionários é que eles permitem um acesso **direto e instantâneo** a qualquer valor, simplesmente usando sua chave.

1.2 Dicionários vs. Listas: A Diferença que Muda o Jogo

Você já conhece as **listas**, que organizam dados por **posição** (índice numérico).

Python

```
frutas = ["maçã", "banana", "uva"]
print(frutas[1])  # Saída: banana (acesso por posição)
```

Em um dicionário, a ordem não é a principal preocupação. O que importa é a **relação** entre a

chave e o valor.

Python

```
precos = {"maçã": 2.50, "banana": 1.75, "uva": 3.00}
print(precos["banana"])  # Saída: 1.75 (acesso por chave)
```

Para um dicionário com milhões de itens, buscar o preço da "banana" leva o mesmo tempo que buscar em um dicionário de três itens.

1.3 Por Que Dicionários São Tão Rápidos? O Conceito de Hashing

O segredo da velocidade está em uma técnica chamada **hashing**. Quando você cria uma chave, o Python a processa com uma função que gera um número único. Esse número é usado para calcular a posição exata na memória onde o valor será armazenado.

É por isso que as chaves devem ser **imutáveis** (não podem ser alteradas). Se uma chave pudesse mudar, seu número único também mudaria, e o Python não saberia mais onde encontrar o valor.

Módulo 2: Criação e a Regra de Ouro das Chaves

2.1 O Símbolo {} e a Ambiguidade (Dicionário vs. Conjunto)

O símbolo {} é usado para criar tanto dicionários quanto **conjuntos (sets)**.

- **Dicionário:** Contém pares chave: valor.
  Python
  ```
  dicionario = {"cor": "azul", "tamanho": "M"}
  ```

- **Conjunto:** Contém apenas valores. O Python remove duplicações.

```python
cores = {"vermelho", "azul", "vermelho"}
print(cores)  # Saída: {'vermelho', 'azul'}
```

Importante: Um {} vazio é sempre um dicionário. Para criar um conjunto vazio, use set().

2.2 A Regra de Ouro: O Que Pode e o Que Não Pode Ser Chave?

Qualquer objeto **hashable** (imutável) pode ser uma chave.

- **Sim, podem ser chaves:** strings, números (int, float), e tuplas.
- **Não podem ser chaves:** listas, dicionários e conjuntos, pois são mutáveis.

```python
dicionario_valido = {
  1: "int",
  3.14: "float",
  "chave": "string",
  (1, 2, 3): "tupla"
}

# Isso causaria um erro (TypeError):
# dicionario_invalido = { ["lista"]: "erro" }
```

Módulo 3: Manipulação Completa de Dicionários

3.1 Acessando Valores com Segurança

Já vimos que dicionario[chave] gera um erro se a chave não existe. O método **.get()** resolve

isso:

- dicionario.get(chave, valor_padrao): Retorna o valor se a chave existe. Caso contrário, retorna valor_padrao (ou None se não especificado).

3.2 Adicionando e Atualizando Pares

Uma única sintaxe serve para adicionar um novo item ou atualizar um existente.

Python

```
livro = {"titulo": "1984"}
livro["autor"] = "George Orwell"      # Adiciona "autor"
livro["autor"] = "Eric Arthur Blair"  # Atualiza o valor de "autor"
```

3.3 Removendo Itens: del, .pop() e .popitem()

- **del**: Remove o par chave: valor permanentemente.
- **.pop(chave)**: Remove o par e **retorna o valor** do item removido.
- **.popitem()**: Remove e retorna o **último** par (chave, valor) adicionado.
- **.clear()**: Remove todos os itens do dicionário.

Python

```
itens = {"a": 1, "b": 2, "c": 3}
valor_removido = itens.pop("a")
print(f"Valor removido com pop: {valor_removido}") # Saída: 1
print(f"Dicionário agora: {itens}") # Saída: {'b': 2, 'c': 3}

# Removendo o último item
ultima_remocao = itens.popitem()
print(f"Último item removido: {ultima_remocao}") # Saída: ('c', 3)
```

Módulo 4: Iteração e as Vistas de Dicionário

4.1 Loops Simples e Explícitos

Você pode iterar diretamente sobre o dicionário para obter as chaves.

Python

```
estoque = {"lápis": 50, "caneta": 120}
for item in estoque:
    print(item)  # Saída: lápis, caneta
```

O uso de .keys() faz a mesma coisa, mas deixa o código mais claro sobre a sua intenção.

4.2 O Poder de .items()

Para acessar a chave e o valor em um único loop, o método .items() é a melhor escolha.

Python

```
for item, quantidade in estoque.items():
    print(f"O produto {item} tem {quantidade} unidades.")
```

4.3 O Que São "Vistas" de Dicionário?

Os métodos .keys(), .values() e .items() retornam objetos especiais chamados **vistas**. Uma vista é como uma "janela" para o dicionário. Ela **não é uma cópia**, o que economiza muita memória para dicionários grandes. As vistas se atualizam automaticamente se o dicionário original mudar.

Python

```
estoque = {"lápis": 50}
visao_itens = estoque.items()
print(f"Vista inicial: {visao_itens}")

estoque["caneta"] = 120
print(f"Vista após adição: {visao_itens}")
# A vista foi atualizada automaticamente!
```

Módulo 5: Dicionários e Outras Estruturas (Integração)

5.1 Dicionários Aninhados

Para representar dados complexos, você pode ter dicionários dentro de dicionários.

Python

```
# Dados de um produto com detalhes e avaliações
produto_detalhado = {
    "id": "PROD-001",
    "nome": "Notebook X-Pro",
    "especificacoes": {
```

```
        "processador": "Intel Core i7",
        "ram_gb": 16,
        "ssd_gb": 512
    },
    "avaliacoes": [
        {"usuario": "Ana", "nota": 5},
        {"usuario": "Carlos", "nota": 4}
    ]
}

# Acessando o valor de RAM
print(produto_detalhado["especificacoes"]["ram_gb"]) # Saída: 16

# Acessando a nota da primeira avaliação
print(produto_detalhado["avaliacoes"][0]["nota"]) # Saída: 5
```

5.2 Listas de Dicionários

É um padrão muito comum para representar coleções de "objetos" com a mesma estrutura. Pense em uma lista de registros de funcionários.

Python

```
funcionarios = [
    {"id": 1, "nome": "Júlia", "cargo": "Analista"},
    {"id": 2, "nome": "Pedro", "cargo": "Gerente"},
    {"id": 3, "nome": "Lucas", "cargo": "Desenvolvedor"}
]

# Acessando o nome do segundo funcionário
print(funcionarios[1]["nome"]) # Saída: Pedro

# Iterando sobre a lista e acessando os dados de cada funcionário
for f in funcionarios:
    print(f"Nome: {f['nome']} - Cargo: {f['cargo']}")
```

5.3 Tuplas como Chaves

Tuplas são imutáveis e podem ser usadas como chaves. Isso é perfeito para dados compostos, como coordenadas ou nomes.

Python

```
# A chave é uma tupla (cidade, estado)
populacao_br = {
    ("São Paulo", "SP"): 12396372,
    ("Rio de Janeiro", "RJ"): 6775561
}

# Acessando a população usando a tupla como chave
print(populacao_br[("São Paulo", "SP")])
# Saída: 12396372
```

Módulo 6: Desafios e Soluções Detalhadas

Desafio 1: Análise de Texto Simples

O Desafio: Você tem um texto e precisa contar a frequência de cada palavra e, depois, imprimir apenas as que aparecem mais de uma vez.

Solução e Lógica Passo a Passo:

Python

```python
texto = "Python é uma linguagem de programação poderosa e Python é fácil de aprender"
texto_limpo = texto.lower().replace("é", "e").replace(",", "").replace(".", "")
palavras = texto_limpo.split()

# Passo 1: Criamos um dicionário vazio para a contagem
frequencia = {}

# Passo 2: Percorremos a lista de palavras e contamos
for palavra in palavras:
    # Usamos o .get() para evitar o KeyError
    contagem_atual = frequencia.get(palavra, 0)
    frequencia[palavra] = contagem_atual + 1

# Passo 3: Percorremos o dicionário de frequência
print("--- Palavras repetidas ---")
for palavra, contagem in frequencia.items():
    if contagem > 1:
        print(f"'{palavra}' aparece {contagem} vezes.")
```

Explicação da Lógica: A chave aqui é a linha frequencia.get(palavra, 0). Ela busca a contagem atual da palavra. Se a palavra não está no dicionário (é a primeira vez que a encontramos), get retorna o valor padrão 0, e nós adicionamos 1 a ela, criando a entrada no dicionário.

Desafio 2: Gerenciamento de Estoque de Loja

O Desafio: Crie um sistema que gerencie um estoque. Você deve:

1. Começar com um estoque inicial.
2. Atualizar a quantidade de um item.
3. Adicionar um novo item.
4. Remover um item.
5. Listar os itens em falta (com quantidade 0).

Solução e Lógica Passo a Passo:

Python

```
estoque = {
    "lápis": 50,
    "caneta": 120,
    "caderno": 75,
    "borracha": 0
}

print("Estoque inicial:", estoque)

# --- Exemplo de atualização de item ---
item_para_atualizar = "borracha"
nova_quantidade = 30
if item_para_atualizar in estoque:
    estoque[item_para_atualizar] = nova_quantidade
    print(f"Estoque de '{item_para_atualizar}' atualizado para {nova_quantidade}.")
else:
    print(f"Erro: Item '{item_para_atualizar}' não encontrado para atualização.")

# --- Exemplo de adição de novo item ---
item_novo = "régua"
quantidade_novo = 15
# Usamos 'not in' para garantir que não vamos sobrescrever um item existente
if item_novo not in estoque:
    estoque[item_novo] = quantidade_novo
    print(f"Item '{item_novo}' adicionado com quantidade {quantidade_novo}.")
else:
    print(f"Erro: Item '{item_novo}' já existe no estoque.")

# --- Exemplo de remoção de item ---
item_para_remover = "caneta"
if item_para_remover in estoque:
    del estoque[item_para_remover]
    print(f"Item '{item_para_remover}' removido do estoque.")

# --- Exemplo de listagem de itens em falta ---
print("\n--- Itens em falta ---")
encontrou_falta = False
for item, quantidade in estoque.items():
    if quantidade == 0:
        print(f"- {item}")
        encontrou_falta = True
if not encontrou_falta:
```

```
    print("Nenhum item em falta no momento.")

print("\nEstoque final:", estoque)
```

Explicação da Lógica: A solução usa **blocos if/else** para controlar o fluxo do programa de forma segura. Os loops com .items() são usados para percorrer os dados e a lógica de verificação (in estoque ou not in estoque) garante que as operações sejam feitas apenas quando faz sentido.

Módulo 7: Outras Operações Úteis

7.1 Juntando Dicionários

- **.update()**: Adiciona ou atualiza itens de um dicionário a outro.
- **Desempacotamento ****: Cria um novo dicionário combinando dois ou mais.

Python

```
dados_pessoais = {"nome": "Lucas", "idade": 28}
novos_dados = {"cidade": "BH", "idade": 29} # "idade" será sobrescrita

# Opção 1: Usando update (modifica o dicionário original)
dados_pessoais.update(novos_dados)
print(dados_pessoais)
# Saída: {'nome': 'Lucas', 'idade': 29, 'cidade': 'BH'}

# Opção 2: Usando desempacotamento (cria um novo dicionário)
dados_completos = {**dados_pessoais, **novos_dados}
print(dados_completos)
# Saída: {'nome': 'Lucas', 'idade': 29, 'cidade': 'BH'}
```

7.2 Copiando Dicionários

Se você fizer dict2 = dict1, ambos os nomes se referem ao **mesmo** dicionário. Para fazer uma cópia independente, use o método .copy().

Python

```
original = {"a": 1, "b": 2}
copia = original.copy()

copia["c"] = 3

print(original) # Saída: {'a': 1, 'b': 2}
print(copia)    # Saída: {'a': 1, 'b': 2, 'c': 3}
```

Módulo 8: Desafio Final: Sistema de Farmácia com Interação do Usuário

Este módulo combina tudo o que você aprendeu em um único programa interativo. Usaremos um loop while para manter o sistema rodando e o try-except para lidar com erros de entrada do usuário.

- **while True**: Cria um loop infinito. O programa só para quando uma condição break for acionada.
- **try-except**: Permite "testar" um bloco de código (try) e, se um erro acontecer, executar outro bloco (except) sem que o programa trave.

Python

```
estoque_farmacia = {
    "paracetamol": {"preco": 5.50, "quantidade": 25},
    "ibuprofeno": {"preco": 8.90, "quantidade": 15},
    "vitamina C": {"preco": 12.00, "quantidade": 50},
```

```
}

while True:
    print("\n--- Sistema da Farmácia ---")
    print("1. Vender produto")
    print("2. Adicionar/atualizar produto")
    print("3. Listar estoque")
    print("4. Sair")

    try:
        escolha = input("Digite sua escolha: ")
        escolha = int(escolha) # Tenta converter a entrada para número
    except ValueError:
        print("Entrada inválida. Digite um número de 1 a 4.")
        continue # Pula o resto do loop e reinicia

    if escolha == 1:
        print("\n--- Venda ---")
        try:
            produto = input("Nome do produto: ").lower()
            quantidade_venda = int(input("Quantidade: "))

            if estoque_farmacia[produto]["quantidade"] >= quantidade_venda:
                estoque_farmacia[produto]["quantidade"] -= quantidade_venda
                print(f"Venda de {quantidade_venda} {produto} realizada.")
            else:
                print("Estoque insuficiente.")
        except KeyError:
            print("Produto não encontrado no estoque.")
        except ValueError:
            print("Quantidade inválida. Por favor, use números.")

    elif escolha == 2:
        print("\n--- Adicionar/Atualizar ---")
        try:
            produto = input("Nome do produto: ").lower()
            preco = float(input("Preço: "))
            quantidade = int(input("Quantidade: "))

            if produto in estoque_farmacia:
                estoque_farmacia[produto]["quantidade"] += quantidade
                print(f"Estoque de {produto} atualizado.")
            else:
```

```
            estoque_farmacia[produto] = {"preco": preco, "quantidade": quantidade}
            print(f"Produto {produto} adicionado.")
        except ValueError:
            print("Entrada inválida. Preço e quantidade devem ser números.")

    elif escolha == 3:
        print("\n--- Estoque Atual ---")
        for produto, dados in estoque_farmacia.items():
            print(f"Produto: {produto} | Preço: R${dados['preco']:.2f} | Quantidade: {dados['quantidade']}")

    elif escolha == 4:
        print("Saindo do sistema. Até logo!")
        break # Sai do loop while e encerra o programa

    else:
        print("Opção não existe. Tente novamente.")
```

Explicação da Lógica:

- O loop while True faz com que o programa execute repetidamente.
- A primeira try-except lida com a entrada do menu. Se o usuário digitar algo que não seja um número, o programa não trava; ele avisa e continua (continue).
- As try-except dentro das opções de venda e adição lidam com erros específicos:
  - KeyError se o usuário digitar o nome de um produto que não existe.
  - ValueError se o usuário digitar uma quantidade ou preço que não é um número.
- A lógica para adicionar/atualizar itens verifica se o produto já existe usando in estoque_farmacia. Se existir, ele apenas atualiza a quantidade. Se não, ele cria um novo item no dicionário.
- A opção "4" usa break para sair do loop, encerrando o programa de forma controlada.

Módulo 9: Conclusão

Os dicionários são uma das estruturas de dados mais importantes e versáteis do Python. Dominá-los é um passo crucial para escrever códigos eficientes e elegantes.

Com este guia, você tem em mãos uma base sólida para trabalhar com dicionários e pode começar a construir programas interativos. Continue praticando com os exemplos e desafios. A melhor forma de dominar é aplicando o que você aprendeu.