Dicionários

Dicionários

▼ Definição

> Os dicionários são estruturas de dados que armazenam informações na forma de pares chave-valor, permitindo acessar valores de forma rápida e organizada por meio de suas respectivas chaves. Eles servem para representar dados estruturados de maneira intuitiva, como informações de um cadastro, configurações de um programa ou até associações mais complexas, como contagens de ocorrências ou mapeamentos entre diferentes elementos. Por serem mutáveis e dinâmicos, os dicionários são amplamente usados quando é necessário adicionar, alterar ou remover informações durante a execução do programa.

- Os dicionários armazenam dados como pares chave-valor

- Os valores podem ser de qualquer tipo, como números, strings, listas, outros dicionários ou até mesmo funções. Portanto, os valores podem ser de tipos mutáveis e imutáveis
- É um tipo de dado mutável, assim como o tipo list, podendo ser modificados após sua criação. Isso significa que é possível adicionar novas chaves, alterar valores existentes ou remover pares chave-valor
- São ideais para organizar dados complexos ou hierárquicos
- Pode-se fazer um paralelo com as structs em linguagem C

▼ Sintaxe

- Segue a lógica: variável = { chave: valor }
- Valor são separados entre si por meio de vírgulas
- O nome da chave é escrita entre aspas e é utilizado (:) para atribuir seu valor
- Exemplo:

```
pessoa = {
   "nome": "Gustavo",
   "sobrenome": "Silva",
   "idade": 25,
   "altura": 1.75
   }
```

- Definição em uma linha:

```
pessoa = {"nome": "Gustavo", "idade": 25}
```

- Caso sejam criadas duas chaves com o mesmo nome, apenas o valor da última será considerado. Exemplo:

```
pessoa = {
   "nome": "Gustavo",
   "sobrenome": "Silva",
      "sobrenome": "Santos",  # Esse será o valor correspondente à chave
   }
```

▼ Criação de dicionário utilizando **dict( )**

- É menos utilizado do que a criação de dicionários por meio de chaves
- Exemplo:

```
produto = dict(nome="Computador", cor="Cinza", codigo=12345)
```

▼ Acessar valores (**[ ]**, **get( )** e **setdefault( )**)

- Para acessar os valores armazenados em um dicionário, utiliza-se colchetes `[]`, com o nome da chave entre eles. Cada chave funciona como um identificador exclusivo para o valor associado. Exemplo:

```
dicionario = {"nome": "Gustavo", "idade": 25, "cidade": "São Paulo"}
print(dicionario["nome"])  # Exibe: Gustavo
```

- A função `get()` é usada em dicionários para acessar o valor associado a uma chave, mas com uma diferença importante: ela permite definir um **valor padrão** caso a chave não exista. Isso evita o erro `KeyError`, que ocorre ao tentar acessar uma chave inexistente usando colchetes `[]`
  - Chave existe: retorna o valor da chave
  - Chave inexistente: retorna None por padrão
  - Chave inexistente: com valor padrão definido: retorna o valor definido: dicionario.get(chave, **valor_padrao**)
  - Exemplo com os três casos:

```
dicionario = {"nome": "Gustavo", "idade": 25}

# Chave existente
print(dicionario.setdefault("nome", "Ana"))  # Exibe: Gustavo
print(dicionario)
# Exibe: {'nome': 'Gustavo', 'idade': 25}

# Chave inexistente (adiciona com valor padrão)
print(dicionario.setdefault("cidade", "São Paulo"))  # Exibe: São Paulo
print(dicionario)
# Exibe: {'nome': 'Gustavo', 'idade': 25, 'cidade': 'São Paulo'}
```

- Exemplo com if e is:

```
dicionario = {"nome": "Gustavo", "idade": 25}

if pessoa.get("altura") is None:
    print("Não existe")
else:
    print(pessoa["altura"])
```

- O método `setdefault()` é usado para obter o valor de uma chave e, caso a chave **não exista**, ele **adiciona a chave ao dicionário com um valor padrão**. Se a chave já existir, ele simplesmente retorna o valor associado a ela sem fazer nenhuma alteração no dicionário
  - Sintaxe: `dicionario.setdefault(chave, valor_padrao)`
  - Exemplo:

```
dicionario = {"nome": "Gustavo", "idade": 25}

# Chave existente
print(dicionario.setdefault("nome", "Ana"))  # Exibe: Gustavo
print(dicionario)
# Exibe: {'nome': 'Gustavo', 'idade': 25}

# Chave inexistente (adiciona com valor padrão)
print(dicionario.setdefault("cidade", "São Paulo"))  # Exibe: São Paulo
print(dicionario)
# Exibe: {'nome': 'Gustavo', 'idade': 25, 'cidade': 'São Paulo'}
```

▼ Adicionar ou alterar valores

- Para adicionar um novo par chave-valor, basta usar uma nova chave. Se a chave já existir, o valor será atualizado. Exemplo:

```
d = {"nome": "Maria"}
d["idade"] = 30       # Adiciona a chave "idade"
d["nome"] = "João"    # Atualiza o valor da chave "nome"
```

- É possível adicionar uma chave dinamicamente. Exemplo:

```
d = {}
chave = "idade"
d[chave] = 30
print(d)  # Exibe: {'idade': 30}
```

▼ Remover Valores (**del**, **pop( )** ou **popitem( )**)

- Método de remoção `del` :

```
dicionario = {"nome": "Gustavo", "idade": 25}
del dicionario["idade"]  # Remove a chave "idade"
```

  - Desse modo, a chave idade foi removida, restando apenas a chave nome.

- Método de remoção `pop()` :

```
dicionario = {"nome": "Gustavo", "idade": 25}
nome = dicionario.pop("nome")  # Remove e retorna o valor da chave
```

  - Removeu a chave nome, restando apenas a chave idade. A variável recebeu o valor que estava associado à chave removida, portanto `nome = "Gustavo"`, sendo do tipo `str`

- Método para remover o último par chave-valor `popitem()` :

```
dicionario = {"nome": "Gustavo", "idade": 25}
ultimo_item = dicionario.popitem()
print(ultimo_item)  # Exibe: ('idade', 25)
print(dicionario)  # Exibe: {'nome': 'Gustavo'}
```

  - Não é possível passar uma chave como parâmetro nesse método

▼ Quantidade de chaves (**len( )**)

- É utilizada a função `len()` para calcular a quantidade de chaves em um dicionário

```
pessoa = {
   "nome": "Gustavo",
```

```
"sobrenome": "Silva",
"idade": 25,
"altura": 1.75
}
print(len(pessoa))  # Exibe: 4
```

▼ Iterar sobre um Dicionário + Modificando valores durante a iteração

- A iteração sobre um dicionário é uma técnica para percorrer seus elementos, permitindo acessar as chaves, valores ou os pares chave-valor. Isso é útil para realizar alguma operação para cada elemento do dicionário, como imprimir os dados, realizar cálculos ou buscar informações específicas.

```
dicionario = {"nome": "Gustavo", "idade": 25, "cidade": "São Paulo"}
for chave in dicionario:
    print(chave)  # Exibe: nome, idade, cidade
```

  - Por padrão, um loop `for` em um dicionário percorre suas chaves.

- É possível modificar os um dicionário durante uma iteração. Exemplo:

```
numeros = {"a": 1, "b": 2, "c": 3}
for chave in numeros:
        numeros[chave] *= 2
print(numeros)  # {'a': 2, 'b': 4, 'c': 6}
```

!!
- Deve-se ter cuidado ao tentar modificar os valores diretamente durante a iteração. O ideal é criar uma cópia das chaves ou usar métodos específicos.

▼ Iterar apenas sobre chaves (**keys( )**)

- Além da utilização de um `for` , que já percorre as chaves de um dicionário por padrão, também pode ser utilizado o método `keys()` . Exemplo:

```
dicionario = {"nome": "Gustavo", "idade": 25, "cidade": "São Paulo"}
for chave in dicionario.keys():
    print(chave)  # Exibe: nome, idade, cidade
```

▼ Iterar apenas sobre valores (**values( )**)

- Para percorrer apenas os valores de um dicionário, ao invés das chaves, utiliza-se o método `values()`

```
dicionario = {"nome": "Gustavo", "idade": 25, "cidade": "São Paulo"}
for valor in dicionario.values():
    print(valor)  # Exibe: Gustavo, 25, São Paulo
```

▼ Iterar sobre pares chave-valor (**items( )**)

- Para acessar tanto as chaves quanto os valores ao mesmo tempo, é utilizado o método items(). Ele retorna cada par como uma tupla (chave, valor)
- Exemplo com o par:

```
dicionario = {"nome": "Gustavo", "idade": 25, "cidade": "São Paulo"}
for par in dicionario.items():
    print(par)  # Exibe: ('nome', 'Gustavo') ('idade', 25) ('cidade', 'São Paulo
```

- Exemplo com duas variáveis:

```
dicionario = {"nome": "Gustavo", "idade": 25, "cidade": "São Paulo"}
for chave, valor in dicionario.items():
    print(f"Chave: {chave}, Valor: {valor}")
# Exibe:
# Chave: nome, Valor: Gustavo
# Chave: idade, Valor: 25
# Chave: cidade, Valor: São Paulo
```

▼ Shallow Copy vs Deep Copy (**copy( )** e **deepcopy( )**)

- O método `copy()` cria uma **cópia rasa (shallow copy)** de um dicionário. Isso significa que ele copia as chaves e os valores do dicionário original para um novo dicionário. Valores imutáveis são copiados de forma idêntica. No entanto, se os valores do dicionário forem mutáveis (como listas ou outros dicionários), apenas a referência desses objetos será copiada, e não o conteúdo deles. Isso ocorre pois, ao copiar um valor mutável, ambos apontam para o mesmo lugar da memória. Sendo assim, quando o original é alterado, a cópia também é e vice-versa.
  - Exemplo `copy()`:

```
dicionario = {"nome": "Gustavo", "idade": 25}

copia = dicionario.copy()  # Cria uma cópia rasa
copia["nome"] = "João"

print(dicionario)  # Exibe: {"nome": "Gustavo", "idade": 25}
print(copia)  # Exibe: {"nome": "João", "idade": 25}
```

  - Exemplo de cópia de valor mutável:

```
dicionario = {"nome": "Ana", "notas": [10, 9, 8]}

copia = dicionario.copy()  # Cria uma cópia rasa
copia["notas"].append(7)

print(dicionario)  # Exibe: {"nome": "Ana", "notas": [10, 9, 8, 7]}
print(copia)  # Exibe: {"nome": "Ana", "notas": [10, 9, 8, 7]}
```

- Uma **cópia profunda (deep copy)** cria uma cópia **independente** de todos os objetos, incluindo objetos aninhados. Isso significa que alterações feitas na cópia não afetam o original
  - Existe uma biblioteca com um método específico para realizar esse tipo de cópia. Assim, para realizar uma cópia profunda, pode-se utilizar `import copy` e sua função `deepcopy()`
  - Exemplo `deepcopy()`:

```
import copy
dicionario = {"nome": "Ana", "notas": [10, 9, 8]}

copia = deepcopy(dicionario)  # Cria uma cópia profunda
copia["notas"].append(7)

print(dicionario)  # Exibe: {"nome": "Ana", "notas": [10, 9, 8]}
print(copia)  # Exibe: {"nome": "Ana", "notas": [10, 9, 8, 7]}
```

- Comparação: Shallow Copy vs Deep Copy:

Característica	Shallow Copy (`copy()`)	Deep Copy (`deepcopy()`)
Objetos Mutáveis	Copia apenas a referência	Cria uma cópia independente
Objetos Imutáveis	Copia o valor diretamente	Copia o valor diretamente
Efeito em Objetos Aninhados	Alterações na cópia podem afetar o original	Alterações na cópia não afetam o original
Complexidade	Mais rápido (copia apenas referências)	Mais lento (copia todo o conteúdo recursivamente)

▼ Atualizar dicionário (**update( )**)

- O método `update()` é usado para adicionar ou atualizar vários pares chave-valor em um dicionário de uma só vez. Ele pode receber outro dicionário ou um iterável de pares (como uma lista de tuplas) como argumento. Se as chaves já existirem, seus valores serão atualizados; caso contrário, novas chaves serão adicionadas
- Exemplo:

```
dicionario = {"nome": "Gustavo", "idade": 25}

dicionario.update({"nome": "novo valor", "altura": 1.75})

print(dicionario)  # Exibe: {'nome': 'novo valor', 'idade': 25, 'altura': 1.75}
```

- Exemplo com definição alternativa:

```
dicionario = {"nome": "Gustavo", "idade": 25}

dicionario.update({nome="novo valor",   altura=1.75})

print(dicionario)  # Exibe: {'nome': 'novo valor', 'idade': 25, 'altura': 1.75}
```

- Exemplo com tupla(também funciona para lista):

```
dicionario = {"nome": "Gustavo", "idade": 25}

tupla = (("nome", "novo valor"), ("altura", 1.75))
dicionario.update(tupla)

print(dicionario)  # Exibe: {'nome': 'novo valor', 'idade': 25, 'altura': 1.75}
```

▼ Ordenar lista de dicionários (***key***)

- O parâmetro `key` permite ordenar uma lista de dicionários com base em uma chave específica. Basta usar uma função (como uma **lambda**) para acessar o valor desejado.
  - Exemplo:

```
pessoas = [
   {"nome": "João", "idade": 25},
   {"nome": "Maria", "idade": 30},
   {"nome": "Ana", "idade": 20},
]
# Ordenar pela idade
pessoas.sort(key=lambda pessoa: pessoa["idade"])
print(pessoas)
# Exibe: [{'nome': 'Ana', 'idade': 20}, {'nome': 'João', 'idade': 25}, {'nor
```