Revisão de Python: Coleções de Dados

> Coleções de dados são usadas para armazenar múltiplos valores em uma única variável.

- **Listas:** São usadas para armazenar uma coleção ordenada de itens.
- **Tuplas:** São usadas para armazenar uma coleção ordenada e imutável de itens.
- **Conjuntos:** São usados para armazenar uma coleção não ordenada e sem duplicatas de itens.
- **Dicionários:** São usados para armazenar uma coleção de pares chave-valor.

Listas

> Listas são usadas para armazenar uma coleção ordenada de itens. Estes itens podem ser de diferentes tipos.

- **Criação de Listas:** As listas podem ser criadas usando colchetes `[]` ou a função `list()`.

```
cores = ["vermelho", "verde", "azul"]
numeros = list((1, 2, 3, 4, 5))
```

- **Acesso a Itens:** Os itens de uma lista podem ser acessados por índice.

```
cores = ["vermelho", "verde", "azul"]
print(cores[0])
print(cores[1])
print(cores[2])
```

- **Alteração de Itens:** Os itens de uma lista podem ser alterados por índice.

```
cores = ["vermelho", "verde", "azul"]
cores[0] = "amarelo"
print(cores)
```

- **Adição de Itens:** Há vários métodos para adicionar itens a uma lista.
  - `append()`: Adiciona um item ao final da lista.
  - `insert()`: Adiciona um item em uma posição específica da lista.
  - `extend()`: Adiciona os itens de uma lista a outra lista.
  - `+=`: Adiciona os itens de uma lista a outra lista.
  - `+`: Concatena duas listas.
  - `*`: Repete os itens de uma lista.

```
cores = ["vermelho", "verde", "azul"]
cores.append("amarelo")
cores.insert(1, "laranja")
cores.extend(["roxo", "rosa"])
cores += ["preto", "branco"]
cores = cores + ["cinza", "marrom"]
cores *= 2
print(cores)
```

- **Remoção de Itens:** Há vários métodos para remover itens de uma lista.
  - `remove()`: Remove o primeiro item com um valor específico.
  - `pop()`: Remove um item em uma posição específica da lista.
  - `del`: Remove um item em uma posição específica da lista.
  - `clear()`: Remove todos os itens da lista.

```
cores = ["vermelho", "verde", "azul"]
cores.remove("verde")
cores.pop(1)
del cores[0]
cores.clear()
print(cores)
```

- `sort()`: Ordena os itens de uma lista em ordem crescente.
- `reverse()`: Inverte a ordem dos itens de uma lista.

```
numeros = [3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5]
numeros.sort()
numeros.reverse()
print(numeros)
```

- `index()`: Retorna a posição de um item específico.
- `count()`: Retorna o número de vezes que um item aparece na lista.
- `len()`: Retorna o número de itens na lista.

```
numeros = [3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5]
print(numeros.index(5))
print(numeros.count(5))
print(len(numeros))
```

Slice

> O slice é usado para acessar um subconjunto de itens de uma lista. Possui lógica semelhante à função `range()`.

- **Sintaxe:** `lista[início:fim:passo]`
- **Início:** Índice de início do slice.
- **Fim:** Índice de fim do slice.
- **Passo:** Tamanho do passo do slice.

```
numeros = [3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5]
print(numeros[2:5])
print(numeros[:5])
print(numeros[5:])
print(numeros[::2])
print(numeros[::-1])
```

- É possível usar slice para alterar itens de uma lista.

```
numeros = [3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5]
numeros[2:5] = [0, 0, 0]
print(numeros)
```

- Outras estruturas de dados também suportam slice, exemplo:
  - Tuplas
  - Strings
  - Arrays
  - Numpy Arrays
  - Pandas Series
  - Pandas DataFrames
  - Dask DataFrames
  - Spark DataFrames
  - etc.

Indexação Negativa

- A indexação negativa é usada para acessar itens de uma lista a partir do final.
- O índice `-1` refere-se ao último item da lista.
- O índice `-2` refere-se ao penúltimo item da lista.
- E assim por diante.

```
numeros = [3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5]
print(numeros[-1])
print(numeros[-2])
print(numeros[-3])
```

> é muito usado para acessar o último item de uma lista.

- A indexação negativa também pode ser usada com slice.

```
numeros = [3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5]
print(numeros[-5:])
print(numeros[:-5])
print(numeros[-5:-2])
print(numeros[::-1])
```

List Comprehension

> É uma maneira eficiente de criar listas sem usar loops.

```
numeros = [1, 2, 3, 4, 5]
quadrados = [x ** 2 for x in numeros]
print(quadrados)
```

- List Comprehension pode ser usado para filtrar itens de uma lista.

```
numeros = [1, 2, 3, 4, 5]
pares = [x for x in numeros if x % 2 == 0]
print(pares)
```

Exercícios

1. Crie uma lista com os números de 1 a 10.
2. Inverta a ordem dos itens da lista.
3. Adicione o número 11 ao final da lista.
4. Remova o número 3 da lista.
5. Adicione o número 3 ao início da lista.

Tuplas

> Tuplas são usadas para armazenar uma coleção ordenada e imutável de itens.

- **Criação de Tuplas:** As tuplas podem ser criadas usando parênteses `()` ou a função `tuple()`.

```
cores = ("vermelho", "verde", "azul")
numeros = tuple([1, 2, 3, 4, 5])
```

- **Acesso a Itens:** Os itens de uma tupla podem ser acessados por índice.

```
cores = ("vermelho", "verde", "azul")
print(cores[0])
print(cores[1])
print(cores[2])
```

- **Alteração de Itens:** Os itens de uma tupla **não** podem ser alterados após a criação.

```
cores = ("vermelho", "verde", "azul")
cores[0] = "amarelo" # Erro!
```

Por que usar tuplas?

- **Imutabilidade:** As tuplas são imutáveis, o que significa que seus itens não podem ser alterados após a criação.
- **Desempacotamento:** As tuplas podem ser desempacotadas em variáveis individuais.
- **Retorno Múltiplo:** As funções podem retornar múltiplos valores como uma tupla.
- **Iteração Eficiente:** As tuplas são mais eficientes para iteração do que listas.
- **Chaves de Dicionários:** As tuplas podem ser usadas como chaves de dicionários.
- **Segurança:** As tuplas são mais seguras do que listas em ambientes concorrentes.

Desempacotamento de Tuplas

- O desempacotamento de tuplas é usado para atribuir os itens de uma tupla a variáveis individuais.

```
cores = ("vermelho", "verde", "azul")
r, g, b = cores
print(r, g, b)
```

- Troca de valores de variáveis (swap).

```
a = 1
b = 2
a, b = b, a
print(a, b)
```

- Atribuição de valores a múltiplas variáveis.

```
a, b, c = 1, 2, 3
print(a, b, c)
```

- Retorno múltiplo de funções.

```
def divisao_e_resto(a, b):
    return a // b, a % b
div, res = divisao_e_resto(10, 3)
print(div, res)
```

- Ignorar valores de uma tupla.

```
a, _, c = (1, 2, 3)
print(a, c)
```

- Trocar valores de uma lista ou array.

```
numeros = [1, 2, 3, 4, 5]
numeros[0], numeros[-1] = numeros[-1], numeros[0]
print(numeros)
```

> Compreeensão de Tuplas não é suportada em Python.

```
numeros = [1, 2, 3, 4, 5]
quadrados = (x ** 2 for x in numeros) # è um gerador, nao uma tupla!
quad = tuple(x ** 2 for x in numeros) # Tupla de quadrados
```

Exercícios

1. Compare o desempenho de iteração entre listas e tuplas.
2. Compare o desempenho das atribuições abaixo:

```
a, b, c, d = 1, 2, 3, 4
```

```
a = 1
b = 2
c = 3
d = 4
```

Conjuntos

> Conjuntos são usados para armazenar uma coleção não ordenada e sem duplicatas de itens.

- **Criação de Conjuntos:** Os conjuntos podem ser criados usando chaves `{}` ou a função `set()`.

```
cores = {"vermelho", "verde", "azul"}
numeros = set([1, 2, 3, 4, 5])
vazio = {} # Dicionário, não conjunto!
valido = set() # Conjunto vazio
```

- **Acesso a Itens:** Os itens de um conjunto **não** podem ser acessados por índice.

```
cores = {"vermelho", "verde", "azul"}
print(cores[0]) # Erro!
```

- **Compreeensão de Conjuntos:** A compreensão de conjuntos é usada para criar conjuntos de maneira eficiente.

```
quadrados = {x ** 2 for x in range(1, 6)}
print(quadrados)
```

- **Adição de Itens:** Há vários métodos para adicionar itens a um conjunto.
  - `add()`: Adiciona um item ao conjunto.
  - `update()`: Adiciona os itens de um conjunto a outro conjunto.
  - `|`: Adiciona os itens de um conjunto a outro conjunto.
  - `union()`: Adiciona os itens de um conjunto a outro conjunto.

```
cores = {"vermelho", "verde", "azul"}
cores.add("amarelo")
cores.update(["roxo", "rosa"])
cores |= {"preto", "branco"}
cores = cores.union(["cinza", "marrom"])
print(cores)
```

- **Remoção de Itens:** Há vários métodos para remover itens de um conjunto.
  - `remove()`: Remove um item do conjunto.
  - `discard()`: Remove um item do conjunto.
  - `pop()`: Remove um item do conjunto.
  - `clear()`: Remove todos os itens do conjunto.

```
cores = {"vermelho", "verde", "azul"}
cores.remove("verde")
cores.discard("verde")
cores.pop()
cores.clear()
print(cores)
```

- **Operações de Conjuntos:** Há vários operadores para realizar operações de conjuntos.
  - `|` : União
  - `&` : Interseção
  - `-` : Diferença
  - `^` : Diferença Simétrica

```
pares = {2, 4, 6, 8, 10}
primos = {2, 3, 5, 7, 11}
print(pares | primos)
print(pares & primos)
print(pares - primos)
print(pares ^ primos)
```

- Conjuntos não podem conter itens mutáveis, como listas, conjuntos ou dicionários.

```
conjunto = {1, [2, 3]} # Erro!
```

- Tuplas e strings podem ser usadas como itens de um conjunto, pois são imutáveis.

```
conjunto = {1, (2, 3)} # OK!
```

Exercícios

1. Escreva um programa que leia uma lista de números inteiros e imprima a lista sem duplicatas.
2. Sejam A e B conjuntos, implemente as funções que calculam:
   - $A \cup B$
   - $A \cap B$
   - $A - B$
   - $B - A$
   - $A \Delta B$ (Diferença Simétrica)
   - $\frac{|A \cap B|}{|A \cup B|}$ (Indice de Jaccard, Obs. levante uma exceção se $A \cup B = \emptyset$)

Dicionários

> Dicionários são usados para armazenar uma coleção de pares chave-valor.
> As chaves de um dicionário devem ser únicas e imutáveis.

- **Criação de Dicionários:** Os dicionários podem ser criados usando chaves `{}` ou a função `dict()`.

```
cores = {"r": "vermelho", "g": "verde", "b": "azul"}
numeros = dict([(1, "um"), (2, "dois"), (3, "três")])
```

- **Acesso a Itens:** Os itens de um dicionário podem ser acessados por chave.

```
cores = {"r": "vermelho", "g": "verde", "b": "azul"}
print(cores["r"])
print(cores["g"])
print(cores["b"])
```

- **Adição de Itens:** Os itens de um dicionário podem ser adicionados por chave.

```
cores = {"r": "vermelho", "g": "verde", "b": "azul"}
cores["y"] = "amarelo"
print(cores)
```

- **Remoção de Itens:** Há vários métodos para remover itens de um dicionário.
  - `pop()` : Remove um item do dicionário por chave.
  - `popitem()` : Remove o último item do dicionário.
  - `del` : Remove um item do dicionário por chave.
  - `clear()` : Remove todos os itens do dicionário.

```
cores = {"r": "vermelho", "g": "verde", "b": "azul"}
cores.pop("g")
cores.popitem()
del cores["r"]
cores.clear()
print(cores)
```

- **Métodos de Dicionários:** Há vários métodos para manipular dicionários.
  - `keys()`: Retorna uma lista de chaves do dicionário.
  - `values()`: Retorna uma lista de valores do dicionário.
  - `items()`: Retorna uma lista de pares chave-valor do dicionário.
  - `get()`: Retorna o valor de uma chave específica.
  - `update()`: Atualiza o dicionário com outro dicionário.

```
cores = {"r": "vermelho", "g": "verde", "b": "azul"}
print(cores.keys())
print(cores.values())
print(cores.items())
print(cores.get("g"))
cores.update({"y": "amarelo", "p": "roxo"})
print(cores)
```

> Tentar acessar uma chave que não existe em um dicionário levanta uma exceção.

```
cores = {"r": "vermelho", "g": "verde", "b": "azul"}
print(cores["y"]) # Erro!
```

- **Verificação de Chave:** O operador `in` pode ser usado para verificar se uma chave existe em um dicionário.

```
if "y" in cores:
    print(cores["y"])
```

- **Valor Padrão:** O método `get()` pode retornar um valor padrão se a chave não existir.

```
print(cores.get("y", "não encontrado"))
```

Compreensão de Dicionários

> A compreensão de dicionários é usada para criar dicionários de maneira eficiente.

```
numeros = [1, 2, 3, 4, 5]
quadrados = {x: x ** 2 for x in numeros}
print(quadrados)
```

Comparação de Coleções

- **Listas:** São usadas para armazenar uma coleção ordenada de itens.
- **Tuplas:** São usadas para armazenar uma coleção ordenada e **imutável** de itens.
- **Conjuntos:** São usados para armazenar uma coleção **não ordenada** e sem duplicatas de itens.
- **Dicionários:** São usados para armazenar uma coleção de pares **chave-valor**.

Iteração sobre Coleções

> Iteração é o processo de acessar itens de uma coleção de dados.

- **For Loop:** É usado para iterar sobre uma sequência (como uma lista, tupla, conjunto ou string) ou outros objetos iteráveis.

```
cores = ["vermelho", "verde", "azul"]
for cor in cores:
    print(cor)
```

- **Iteração com Índice:** A função `enumerate()` pode ser usada para iterar sobre uma sequência com índices.

```
cores = ["vermelho", "verde", "azul"]
for i, cor in enumerate(cores):
    print(i, cor)
```

- **Iteração em Dicionários:** Os dicionários podem ser iterados por chave, valor ou ambos.

```
cores = {"r": "vermelho", "g": "verde", "b": "azul"}
for chave in cores:
    print(chave, cores[chave])
for chave, valor in cores.items():
    print(chave, valor)
```

> Observe que a ordem de iteração em um dicionário é arbitrária.
> No segundo exemplo, a função `items()` retorna uma lista de pares chave-valor que é desempacotado.

Operações sobre Coleções

- **max, min:** Retorna o maior e o menor valor de uma coleção.
- **sum:** Retorna a soma dos valores de uma coleção.
- **len:** Retorna o número de itens de uma coleção.

- **all:** Retorna `True` se todos os itens de uma coleção são verdadeiros.

```
print(all(i**3 % 2 == 0 for i in range(100)))
```

- **any:** Retorna `True` se pelo menos um item de uma coleção é verdadeiro.

```
print(any(i**3 % 2 == 0 for i in range(100)))
```

> Obs.: São considerados verdadeiros os valores diferentes de zero, `None`, `False`, `[]`, `()`, `{}`, `""` e `0`.

- Exemplo de função para verificar se um número é primo.

```
def eh_primo(x):
    if x < 2:
        return False
    for i in range(2, int(x ** 0.5) + 1):
        if x % i == 0:
            return False
    return True
```

```
def eh_primo2(x):
    return x > 1 and all(x % i != 0 for i in range(2, int(x ** 0.5) + 1))
```

> Compare a performance das funções `eh_primo` e `eh_primo2` .

- **sorted**: Retorna uma lista ordenada de uma coleção.

```
numeros = [3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5]
print(sorted(numeros))
```

- **reversed**: Retorna uma lista invertida de uma coleção.

```
numeros = [3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5]
print(list(reversed(numeros)))
```

- **zip**: Retorna gerador de tuplas com itens de coleções correspondentes.

```
roupas = ["camisa", "calça", "sapato"]
cores = ["vermelho", "verde", "azul"]
tamanhos = ["P", "M", "G"]
#usando indices
for i in range(len(roupas)):
    print(roupas[i], cores[i], tamanhos[i])

#usando zip
for roupa, cor, tamanho in zip(roupas, cores, tamanhos):
    print(roupa, cor, tamanho)
```

- descompactar uma lista de tuplas (unzip)

```
pares = [(2, 4), (6, 8), (10, 12)]
a, b = zip(*pares)
print(a, b)
```

- **enumerate:** Retorna gerador de tuplas com índices e itens de uma coleção.

```
cores = {"vermelho", "verde", "azul"}
for i, cor in enumerate(cores):
    print(i, cor)
```

> Observem que, mesmo um conjunto não possuindo índices, a função `enumerate` associa um contador como variável de controle do loop.