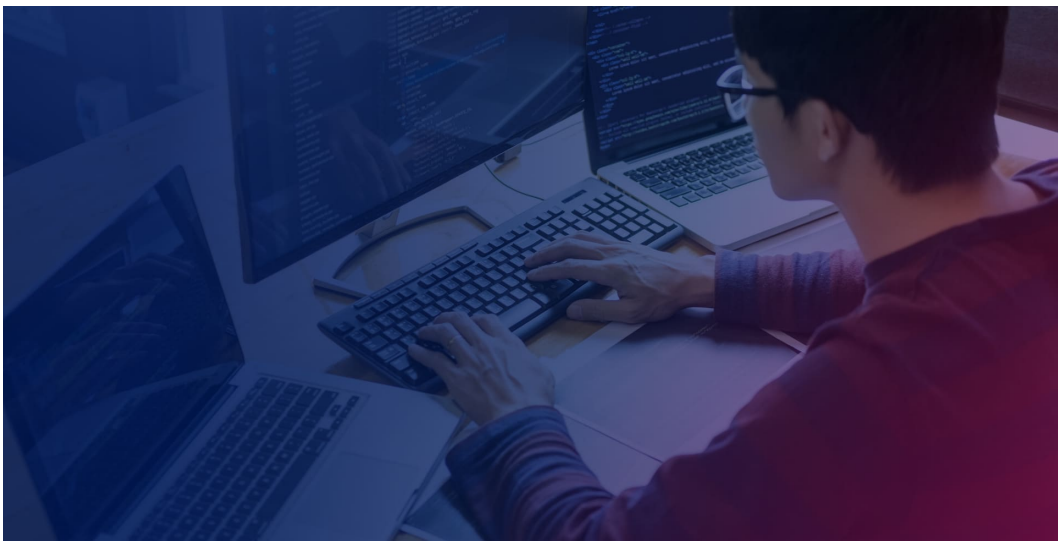


Estruturas de dados básicas do Python

Prof. Frederico Tosta de Oliveira

Prof. Kleber de Aguiar

Descrição

Formas de armazenar e organizar os dados em um programa de computador utilizando a estrutura de dados.

Propósito

Explicar a importância das estruturas de dados. Discutir o funcionamento e as características de algumas dessas estruturas em Python: Vetor, Matriz, Lista, Pilha, Fila, Tuplas e Conjunto.

Preparação

Antes de iniciar a leitura deste conteúdo, sugerimos que tenha uma versão do software Python e a biblioteca NumPy instalados em seu computador.

Para entrar no shell, independentemente do sistema operacional, abra, após sua instalação, o prompt de comando, digite **python** e pressione a tecla [ENTER] ou [RETURN]. Deverá aparecer uma tela informando a versão do Python instalada.

Nesse momento, o interpretador aguardará a digitação do programa. O código digitado fica ao lado do marcador >>>. Os textos sem o marcador são os retornos da execução do código.

Objetivos

Módulo 1

Vetores, matrizes e classificação das estruturas de dados

Reconhecer as classificações de estrutura de dados e as estruturas de dados Vetor e Matriz.

Módulo 2

Listas

Identificar as estruturas de dados Lista.

Módulo 3

Pilhas, Filas, Tuplas e Conjuntos

Reconhecer as estruturas de dados Pilha, Fila, Tupla e Conjunto.

Introdução

Suponha que você deseje assistir a um filme em determinada plataforma de streaming. Já imaginou quão difícil e lenta seria a busca de um título se ele não estivesse organizado obedecendo a uma forma bem definida?

Os dados e as informações em um computador, portanto, precisam ser armazenados e organizados em uma forma bem definida para permitir que a busca de ambos seja feita de forma eficiente. A forma como eles são organizados e armazenados no computador é caracterizado por estrutura de dados.

No vídeo a seguir, contextualizaremos um pouco mais o conceito de estrutura de dados e apresentaremos um resumo dos próximos módulos:

Para assistir a um vídeo sobre o assunto, acesse a versão online deste conteúdo.

1 - Vetores, matrizes e classificação das estruturas de dados

Ao final deste módulo, você será capaz de reconhecer as classificações de estrutura de dados e as estruturas de dados Vetor e Matriz.

Classificação das estruturas de dados

Definição

Trata-se de uma forma de armazenar e organizar informações para facilitar o acesso e sua modificação. Uma única estrutura de dados não funciona perfeitamente para todos os casos; por isso, é necessário conhecer seu funcionamento.

As estruturas de dados são classificadas segundo as características descritas adiante.

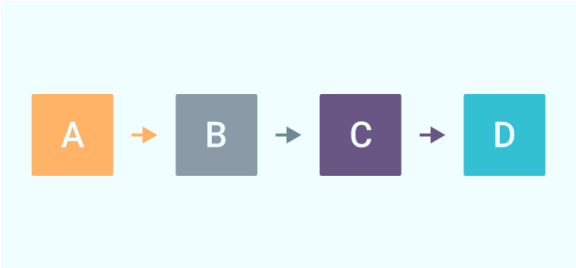
Características

As estruturas de dados podem ser classificadas de acordo com as seguintes características: lineares ou não lineares, homogêneas ou não homogêneas, estáticas ou dinâmicas. Vejamos cada umas delas.

Lineares ou não lineares

Estruturas de dados lineares contêm seus dados organizados de forma cronológica ou sequencial, isto é, em que um item é adicionado de forma adjacente ao outro. As estruturas de dados não lineares, por sua vez, não são organizadas sequencialmente.

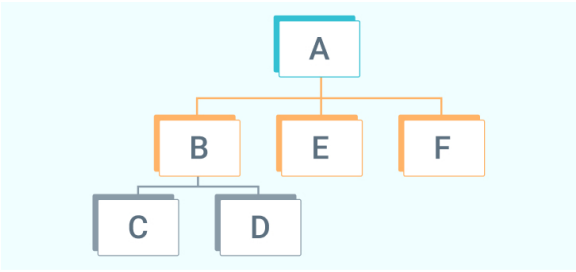
Os itens de uma estrutura não linear são capazes de estar conectados a diversos outros, podendo significar um relacionamento entre eles. Veja as imagens que correspondem a cada uma dessas estruturas:



Estruturas de dados lineares.

Lineares

Cada item está conectado apenas aos seus vizinhos: o item A só está conectado ao item B (sucessor); o item B; conectado aos itens A (antecessor) e C (sucessor) - e assim por diante.



Estruturas de dados não lineares.

Não lineares

O primeiro item (A) está conectado simultaneamente a três outros itens: B, E e F. Nesse tipo de estrutura de dados, não é possível determinar se há uma sequência entre os itens nem quem vem antes ou depois. Concorda?

Homogêneas ou não homogêneas

Elas podem ser de:

Mesmo tipo

Estruturas de dados homogêneas contêm todos os dados do **mesmo tipo** (inteiro, decimal, caractere etc.).

Tipos diferentes

Estruturas de dados não homogêneas podem conter dados de **tipos diferentes**.

Estáticas e dinâmicas

Elas podem ser de:

Tamanho fixo

Estruturas de dados estáticas têm seu tamanho fixo, que é definido durante a escrita do código. Uma vez alocada a memória para a estrutura, ela não pode ser alterada durante a execução do programa, porém, quando ele é iniciado, o espaço alocado para ela é garantido.

Tamanho variado

Estruturas de dados dinâmicas podem variar de tamanho durante a execução do programa, sendo capazes de aumentar ou diminuir. Apesar de tais estruturas serem mais flexíveis, a alocação de memória não é garantida; além disso, o aumento descontrolado dessa estrutura pode “estourar” a memória do seu computador, fazendo com que seu programa pare.

Vetores e matrizes

Vetor

Os vetores e matrizes são estruturas de dados:

Homogêneas

Contêm todos os seus elementos do mesmo tipo (inteiro, decimal, caracteres etc.).

Estáticas

Têm seu tamanho delimitado ao longo de sua declaração e continuam fixas durante a execução do programa.

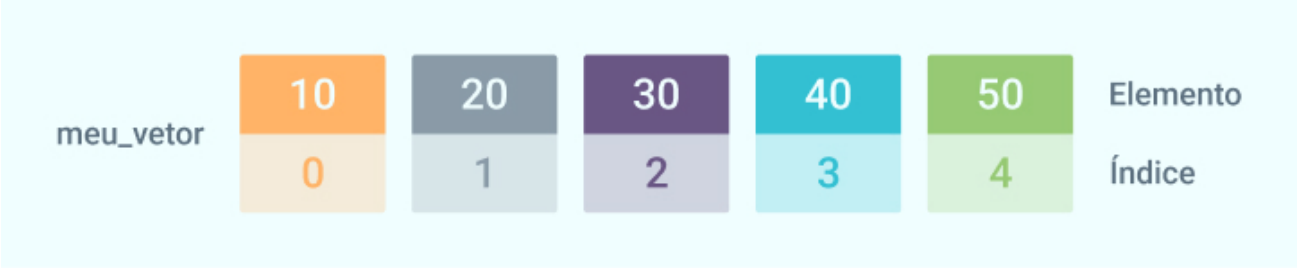
Lineares

Têm seus dados organizados de forma sequencial.

A matriz é uma sequência de elementos do mesmo tipo que ocupa um lugar contínuo de memória. Cada elemento de uma matriz pode ser referenciado pelo seu **índice** (ou **posição**). Normalmente, os índices são números inteiros que começam em zero.

> **As matrizes podem ter várias dimensões. Quando a matriz só possui uma dimensão, ela ganha o nome de vetor.**

A imagem a seguir mostra um vetor de números inteiros denominado **meu_vetor** composto por cinco elementos: 10, 20, 30, 40 e 50.



Vetor de números inteiros.

Observe que cada elemento está associado a um índice. Para acessar o número 10, utilizamos o índice 0; para acessar o 20, o índice 1 - e assim por diante. As matrizes são extremamente eficientes na recuperação de seus dados. Dessa forma, conseguimos armazenar diversos valores sem a necessidade de criar muitas variáveis.

O Python disponibiliza duas formas de trabalhar com matrizes sem a necessidade de importar outras bibliotecas. A primeira forma utiliza a classe **list**; a outra, a classe **array** do pacote **array**.

Em nenhum dos casos, existe uma experiência pura de trabalhar com matrizes, principalmente no que diz respeito à dinamicidade. Tanto os objetos do tipo **list** quanto os do **array** são dinâmicos. Por isso, vamos utilizar a biblioteca NumPy, uma das principais bibliotecas para computação científica em Python.

Definindo um vetor

Para definirmos um vetor em NumPy, utilizamos a seguinte sintaxe:

```python
nome = array([ elementos ], dtype=tipo)
```

Nesse caso, **nome** é o nome da variável; **elementos**, os itens separados por vírgula que vão compor o vetor; e tipo, o tipo do elemento que será armazenado no vetor como int, float e str. O tamanho do vetor é definido automaticamente pelo número de elementos.

Para ilustrarmos isso, vamos declarar uma variável chamada **notas_real**, que é um **vetor** de **inteiros** contendo os seis possíveis valores de notas do Real: R$2,00, R$5,00, R$10,00, R$20,00, R$50,00 e R$100,00. Primeiramente, vamos importar a classe **array** do NumPy e, em seguida, declarar o vetor.

```python
>>> from numpy import array
>>> notas_real = array([2, 5, 10, 20, 50, 100], dtype=int)
```

Acessando os dados do vetor

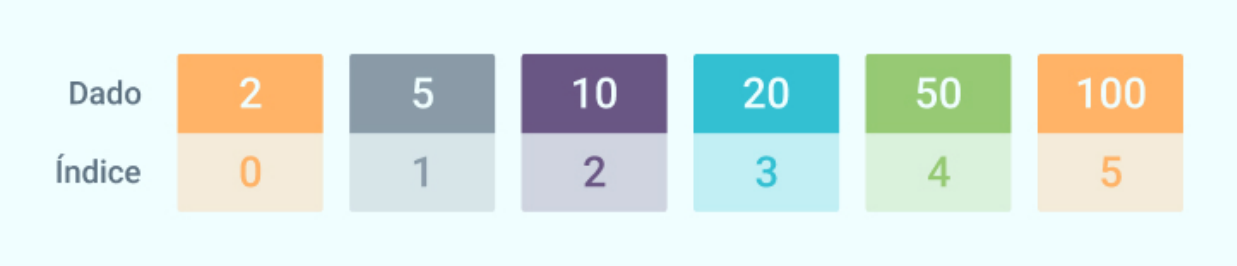
Depois de definida a variável, precisamos encontrar uma forma de acessar seus elementos. Os vetores são extremamente otimizados na recuperação de um dado armazenado, pois eles permitem recuperá-lo diretamente pela sua posição dentro do vetor, ou seja, pelo seu índice.

Em NumPy, podemos acessar um elemento de um vetor da seguinte forma:

```python
nome[indice]
```

Nesse caso, **indice** é um número inteiro que representa a posição do elemento no vetor. Em NumPy, o índice começa em 0 (zero).

Uma ilustração dos índices do vetor **notas_real** está representada na imagem a seguir. O elemento 2 é armazenado no índice 0; assim, ele pode ser acessado por **notas_real[0]**. Já o elemento 5, que está armazenado no índice 1, pode ser acessado por **notas_real[1]** - e assim por diante.



Índices do vetor notas_real.

Este código imprime os dados armazenados nos índices 0 e 5:

Python

```
>>> notas_real = array([2, 5, 10, 20, 50, 100], dtype=int)
>>> print(notas_real[0])
2
>>> print(notas_real[5])
100
```

Para alterar um elemento do vetor, basta atribuir o novo valor ao seu índice. Vamos praticar isso um pouco.

No campo Input do emulador de código a seguir, informe o índice do elemento que você deseja substituir e, na linha seguinte, o novo valor a ser inserido nesse elemento do vetor.

Exemplo:

5
20

Clique em Executar e observe o retorno do script no campo Console do emulador:

Exercício 1 — TUTORIAL — COPIAR

Python3

```
from numpy import array

def main():
    idades = array([10, 30, 45, 62, 74], dtype=int)
    print(f"vetor antes da insercao: {idades}")
```

null

null

Iterando o vetor

Para percorrer um vetor, utiliza-se a sintaxe:

```python
for elemento in vetor:
....(código)
```

Nesse caso, **elemento** é o nome da variável que conterá o valor de cada elemento; **vetor**, o nome da variável que desejamos percorrer; e (**código**), o trecho de código que será repetido pelo laço.

Veja o exemplo a seguir. Nele, vamos imprimir todos os elementos do vetor notas_real.

```python
>>> for elemento in notas_real:
...   print(elemento, end=' - ')
2 - 5 - 10 - 20 - 50 - 100
```

Matrizes

Definindo uma matriz

A declaração de matrizes é muito semelhante à de um vetor. Declaramos uma matriz de duas dimensões da seguinte forma:

```python
nome =array([ [vetor1], [vetor2], [vetor3], ... ], dtype=tipo)
```

Nesse caso, **nome** é o nome da variável, **vetor1**, **vetor2** e **vetor3** representam as linhas da matriz e **tipo** é o tipo do elemento que será armazenado na matriz como int, float e str. O número de colunas será determinado pelo número de elementos dos vetores.

Para ilustrarmos isso, vamos imaginar uma tabela contendo as cadeiras (ou matérias) para os **dois** semestres de um ano. Cada semestre contém **três** cadeiras. No primeiro semestre, temos as cadeiras Português, Matemática e Química; no segundo semestre, História, Geografia e Física.

Veja a tabela adiante. Na primeira linha, há as cadeiras do primeiro semestre; na segunda linha, as cadeiras do segundo semestre. Observe que existem duas linhas e três colunas:

1 semestre	Português	Matemática	Química

2 semestre	História	Geografia	Física

Tabela: Tabela de cadeiras.
Frederico Tosta de Oliveira.

Para representarmos essa tabela como uma matriz, vamos declarar uma variável chamada **cadeiras**, que é uma matriz com elementos do tipo **str** (caracteres). Teremos dois elementos na primeira dimensão (linha) e três na segunda dimensão (coluna).

Observe a seguir:

```python
cadeiras = array([[“Português”, “Matemática”, “Química”], [“História”, “Geografia”, “Física”]], dtype=str)
```

Acessando os dados da matriz

Assim como nos vetores, os dados de uma matriz podem ser acessados diretamente pelos seus índices. Em uma matriz de duas dimensões, pode-se entender o índice como coordenadas de linha e coluna.

Em NumPy, podemos acessar um dado de uma matriz da seguinte forma:

```python
nome[indice_1][indice_2]
```

Nesse caso, **indice_1** é um número inteiro que representa a posição do item na primeira dimensão (linha); **indice_2**, um número inteiro que representa a posição do item na segunda dimensão (coluna). Assim como os vetores, os índices da matriz começam em 0 (zero).

Uma ilustração dos índices da matriz **cadeiras** está representada na imagem a seguir. A cadeira Português é armazenada no índice 00, de forma que ela pode ser acessada por **cadeiras[0][0]**, enquanto a cadeira Matemática pode ser acessada por **cadeiras[0][1]** - e assim por diante.

Índices da matriz cadeiras.

Este código imprime os dados armazenados nos índices 00 e 12:

```python
```

```
>>> cadeiras = array([[" Português", "Matemática" , "Química" ], ["História", "Geografia" , "Física"
]], dtype=str)>>> print(cadeiras[0][0])
Português
>>> print(cadeiras[1][2])
Física
```

Para alterar um elemento de uma matriz, basta atribuir o novo valor ao seu índice. No exemplo a seguir, vamos substituir o valor **Física**, que está no índice [1][2], por **Cálculo**.

Python

```
>>> cadeiras = array([["Português", "Matemática", "Química"], ["História", "Geografia", "Física"]], dtype=str)
>>> cadeiras[1][2] = 'Cálculo'
>>> print(cadeiras[1][2])
Cálculo
```

Iterando a matriz

Para percorrer os elementos de uma matriz de duas dimensões, utiliza-se a sintaxe:

Python

```
for linha in matriz:
    for elemento in linha:
        ....(código)
```

Veja o exemplo a seguir. A função itera_matriz imprime todos os elementos da matriz recebida por parâmetro.

No exemplo, vamos percorrer a matriz cadeira. Clique em Executar no seguinte emulador:

Exercício 2 TUTORIAL COPIAR

Python3

```
from numpy import array

def itera_matriz(matriz):
    for linha in matriz:
        print('Início de uma nova linha')
```

null

null

Note que, no script do emulador anterior, há uma função altera_matriz (linhas 9 a 11), que tem quatro argumentos: uma **matriz**, uma variável **elemento** e os índices **i** e **j**, que indicarão o elemento da matriz recebida por parâmetro a ser alterada.

Dica

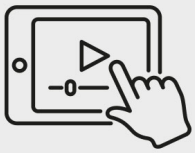
Que tal colocar a mão na massa e praticar a alteração de elemento em uma matriz?

Desfaça o o comentário da linha 15 (deletando o caracter #) e informe os parâmetros na chamada da função altera_matriz: substitua o primeiro **None** pelo valor que você deseja inserir na matriz; o segundo **None**, pelo índice referente à linha a ser modificada; e o terceiro **None**, pelo índice referente à coluna. Não se esqueça de clicar em Executar.

Multiplicação de matrizes

Veja neste vídeo como percorrer e manipular os dados de uma matriz, além de um exemplo da multiplicação de matrizes.

Para assistir a um vídeo sobre o assunto, acesse a versão online deste conteúdo.



A biblioteca NumPy oferece uma gama de funcionalidades referentes a matrizes, desde as operações, como a multiplicação de matrizes e o cálculo de determinante, até a leitura de planilhas. Um bom domínio dessa biblioteca pode ajudar muito no desenvolvimento de sistemas científicos e de engenharia.

Falta pouco para atingir seus objetivos.

Vamos praticar alguns conceitos?

Questão 1

Considere o código a seguir:

Python

```
>>> from numpy import array
>>> vetor1 = array([1, 2, 3, 4], dtype=int)
>>> vetor2 = array([0, 0, 0, 0], dtype=int)
>>> aux = 0
>>> for elemento in vetor1:
...  vetor2[aux] = elemento*10
```

Quais seriam os valores impressos pelo programa?

A [0, 0, 0, 0]

B [11, 12, 13, 14]

C [10, 20, 30, 40]

D [1, 2, 3, 4]

E [20, 30, 40, 50]

Parabéns! A alternativa C está correta.

Foram trocados os valores do vetor2 pelos valores do vetor1 multiplicados por 10. Portanto, os valores impressos pelo programa são [10, 20, 30, 40].

Questão 2

Considere o código a seguir:

Python

```
>>> from numpy import array
>>> matriz = array([[1, 2, 3], [4, 5, 6], [7, 8, 9]], dtype=int)
>>> aux_linha = 0
>>> aux_coluna = 0
>>> for linha in matriz:
>>>     for elemento in linha:
```

Quais seriam os valores impressos pelo programa?

A 1, 2, 3

B 1, 4, 7

C 3, 6, 9

D 1, 5, 9

E 1, 6, 9

Parabéns! A alternativa D está correta.

Esse é um clássico programa de imprimir a diagonal principal. O laço se inicia percorrendo as linhas da matriz. Para cada linha, é percorrido cada elemento. Mantém-se uma variável auxiliar para saber em qual linha e coluna se está a cada iteração do laço (aux_linha e aux_coluna). Quando o número da linha igual ao da coluna (aux_linha == aux_coluna) é encontrado, isso indica que se está na diagonal principal da matriz. Nesse momento, o valor do elemento da matriz é impresso.

2 - Listas

Ao final deste módulo, você será capaz de identificar as estruturas de dados Lista.

Visão geral

Introdução

Neste módulo, você será apresentado às estruturas de dados Lista e como elas são implementadas em Python. Além disso, conhecerá a classe **list** e seus principais métodos e características.

> **Algumas estruturas de dados são muito semelhantes, diferindo principalmente na forma como são empregadas. Esse é o caso das listas, pilhas e filas. Neste módulo, falaremos sobre as listas.**

Quanto às suas características, as listas são estruturas de dados que podem ser:

Lineares

Têm seus dados organizados de forma sequencial.

Não homogêneas

Podem conter elementos de diferentes tipos (inteiro, decimal, caracteres etc.).

Dinâmicas

Seus tamanhos podem variar durante a execução do programa quando inserimos e removemos elementos.

Vejamos de forma mais detalhada as listas em Python.

Conceitos de lista

Conceitos

Em Python, a classe responsável por implementar as listas se chama **list**. O **list** é um dos tipos sequenciais básicos do Python, assim como **tuple** e **range**. Ele é um tipo de dado interno ao Python e não precisa ser importado.

As listas possuem diversos métodos que podem ser utilizados para realizar diversas ações, como inserir ou buscar elemento, além de inverter e ordenar, entre outros, como veremos a seguir.

Definindo uma lista

Para declararmos um objeto do tipo **list**, podemos proceder de diversas formas:

Utilizar apenas os colchetes para declarar uma lista vazia

lista = []

Utilizar os colchetes com os elementos separados por vírgula

lista = [a, b, c]

Utilizar o construtor list(), que aceita como parâmetro outras coleções

lista = list()
lista = list([1, 2, 3])

Os elementos de uma lista em Python podem ser de diferentes tipos, incluindo outra lista. No exemplo adiante, definiremos uma lista que contém um número inteiro, uma string e outra lista.

Python

```
>>> lista = [1, 'Hello', [1, 2]]
```

É importante considerar que as listas são sequências **mutáveis**.

> **Elas podem ser alteradas durante a execução do programa.**

Acessando os dados da lista

A forma de acessar um elemento da lista é muito similar à das matrizes. Precisa-se apenas do índice do elemento.

No exemplo a seguir, vamos imprimir o valor do elemento na posição 1, lembrando que o **primeiro elemento da lista tem índice 0**.

```python
>>> lista = ['a', 'b', 'c', 'b']
>>> item = lista[1]
>>> print(item)
'b'
```

Durante a programação, é comum não saber o índice do elemento que se deseja acessar. Para esse tipo de situação, pode-se utilizar o método **index**, que retorna o índice do elemento procurado.

Observe que o método **index** retorna o índice da primeira ocorrência do elemento.

```python
>>> lista = ['a', 'b', 'c', 'b']
>>> indice_c = lista.index('c')
>>> print(indice_c)
2
>>> indice_b = lista.index('b')
>>> print(indice_b)
```

Assim como nas matrizes, podemos utilizar o índice de um elemento para alterá-lo.

No exemplo a seguir, recuperaremos o índice da palavra **carro** e o alteraremos para **moto**:

```python
>>> lista = ['avião', 'helicóptero', 'carro', 'navio']
>>> indice_carro = lista.index('carro')
>>> print(indice_carro)
2
>>> lista[indice_carro] = 'moto'     ← Alteração do elemento na posição de índice 2 da lista
>>> print(lista)
```

Iterando a lista

Em Python, percorrer uma lista é algo feito de forma muito rápida e direta. Por ser iterável (*iterable*), ou seja, capaz de retornar seus elementos um de cada vez, é possível percorrer a lista utilizando o laço **for** diretamente, sem a necessidade de criar variáveis de controle de *loop*.

Para isso, empregaremos a seguinte sintaxe:

Python

```
for elemento in lista:
....(código)
```

Nesse caso, **elemento** é o nome da variável que conterá o valor de cada elemento dentro do laço; **lista**, o nome da variável do tipo lista que desejamos percorrer; e (**código**), o trecho de código a ser repetido pelo laço.

No exemplo adiante, vamos iterar a lista chamada **minha_lista** e imprimir seu valor:

Python

```
>>> minha_lista = [1, 'moto', [1, 2, 3]]
>>> for elemento in minha_lista:
>>>     print(elemento)
1
moto
[1, 2, 3]
```

Observe que listas dentro de listas ocupam apenas uma posição; por isso, a lista [1, 2, 3] foi retornada inteiramente.

Comentário

Também é possível percorrer uma lista acessando seus elementos por meio de seus índices:

for i in range(0, len(Lista)):
**   print(Lista[i])**

Nesse caso, o i é a variável que conterá o valor de cada número do intervalo gerado pela função nativa **range**, que retorna uma sequência de números inteiros, iniciando pelo seu primeiro argumento e terminando no número anterior ao seu segundo argumento. Há ainda um terceiro argumento, o incremento (“passo”), o qual, quando não informado, tem como valor padrão o número **0**.

Como a função nativa **len** retorna a quantidade de elementos de **Lista** (3), os valores assumidos por i durante a execução do laço **for** serão **0, 1, 2** (os índices de Lista).

Alterando a lista

Por ser dinâmico (ou mutável, no linguajar Python), o objeto do tipo **list** contém um conjunto de métodos que permite alterar seus elementos. Veremos a seguir esses principais métodos.

Quando precisamos inserir um novo elemento na lista, podemos utilizar os métodos **append** ou **insert**. Esses métodos executam as inserções da seguinte forma:

Append

O método **append** insere o elemento passado como parâmetro no final da lista

Insert

O método insert **insere** o elemento na posição indicada no parâmetro.

Confira o código adiante para verificar como são utilizados tais métodos:

```python
>>> lista = list()
>>> lista.append('carro')
>>> print(lista)
['carro']
>>> lista.append('moto')
>>> lista.append('avião')
>>> print(lista)
['carro', 'moto', 'avião']
>>> lista.insert(1, 'bicicleta')     ← Inserimos o elemento bicicleta na posição de índice 1
>>> print(lista)
['carro', 'bicicleta', 'moto', 'avião']
```

Observe que adicionamos três elementos em sequência utilizando o método **append**, carro, moto e avião; depois, utilizamos o método **insert** para inserir o elemento bicicleta na posição de **índice 1**, ou seja, como segundo elemento da lista.

Em algumas ocasiões, precisamos juntar duas listas. Para isso, usamos o método **extend**, em que passamos uma lista como parâmetro. Também podemos empregar o método **append**, porém, para isso, teríamos de adicionar uma lista dentro da outra, que não é o desejado.

Observe o próximo exemplo. Nele, mostraremos a utilização dos dois métodos:

```python
>>> lista_a = [1, 2, 3]
>>> lista_b = [4, 5, 6]
>>> lista_a.append(lista_b)
>>> print(lista_a)
[1, 2, 3, [4, 5, 6]]     ← lista_b foi adicionada dentro da lista_a como um elemento único
>>> lista_a = [1, 2, 3]
>>> lista_a.extend(lista_b)
>>> print(lista_a)
[1, 2, 3, 4, 5, 6]     ← lista_a e lista_b foram concatenadas
```

Ao utilizar o método **append**, o elemento de índice 3 da lista_a passa a ser a lista_b inteira, [4, 5, 6]. Quando utilizamos o método **extend**, obtemos o resultado desejado.

Atenção!

Todos os métodos alteram a lista original: não é retornado nenhum valor.

Além dos métodos para adicionar elementos, o **list** contém aqueles usados para remover itens da lista. O método **remove** é utilizado para tirar o elemento passado como parâmetro, enquanto o **clear** remove todos os elementos da lista.

Observe, no exemplo a seguir, que, para usarmos o método **remove, não utilizaremos o índice, e sim o valor do elemento**:

Python

```
>>> lista = ['carro', 'bicicleta', 'moto', 'avião']
>>> lista.remove('bicicleta')
>>> print(lista)
['carro', 'moto', 'avião']
>>> lista.clear()
>>> print(lista)
```

Outras funções da lista

Além dos métodos apresentados anteriormente, o objeto **list** contém outros que são úteis. Dois exemplos são o método **sort**, para ordenar a lista, e o **reverse**, para inverter os elementos dela. Ambos alteram a própria lista!

No próximo exemplo, mostraremos como esses métodos são utilizados:

Python

```
>>> lista = [3, 2, 1, 6, 9]
>>> lista.sort()
>>> print(lista)
[1, 2, 3, 6, 9]
>>> lista.reverse()
>>> print(lista)
```

Algumas funções internas do Python podem ser aplicadas às listas, como **len**, **min**, **max** e **sum**, que retornam o tamanho, o elemento de valor mínimo, o elemento de valor máximo e a soma dos elementos da lista, respectivamente.

No próximo exemplo, mostraremos como essas funções são utilizadas:

Python

```
>>> lista = [3, 7, 2, 6]
>>> len(lista)
4
>>> min(lista)
2
>>> max(lista)
```

Também podemos combinar essas funções para alcançar novos objetivos. No exemplo adiante, vamos calcular a média dos números de uma lista sem percorrê-la apenas utilizando as funções **sum** e **len**:

Python

```
>>> lista = [1, 3, 5, 7, 9]
>>> media = sum(lista)/len(lista)
>>> print(media)
5.0
```

Operadores nas listas

Alguns operadores do Python, como os de pertinência (**in e not in**), equivalência (==), concatenação (+) e multiplicação (*), também podem ser aplicados aos objetos do tipo **list**.

> **Todos esses operadores retornam algum objeto, seja um booleano (True ou False), que pode ser utilizado em condicionantes if, seja uma nova lista, como acontece com o operador concatenação (+).**

No próximo exemplo, vamos mostrar o uso desses operadores:

Python

```
>>> lista1 = [1, 2, 3]
>>> lista2 = [1, 2, 3]
>>> lista3 = [4, 5, 6]
>>> print(lista1 == lista2)
True
>>> print(lista1 == lista3)
False
>>> if 3 in lista1:
...     print('Achei o 3!!!')
Achei o 3!!!
>>> nova_lista = lista1 + lista3
>>> print(nova_lista)
[1, 2, 3, 4, 5, 6]
>>> lista_repetida = lista1 * 4
>>> print(lista_repetida)
[1, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2, 3]
```

Mais sobre as listas

Uma funcionalidade muito interessante e útil que o Python disponibiliza é a capacidade de criar listas e partir de outras listas de forma simplificada utilizando a chamada **compreensão de listas** (do inglês *list comprehensions*).

A compreensão de listas permite filtrar, transformar e aplicar funções aos elementos de uma lista. Sua sintaxe é:

Python

```
[item for item in lista]
```

Nesse caso, **item** é o nome da variável que receberá o valor de cada elemento da lista. É sobre essa variável que aplicaremos os filtros e as transformações.

No exemplo a seguir, desejamos multiplicar todos os itens de uma lista por 2:

Python

```
>>> lista = [1, 2, 3]
>>> nova_lista = [item*2 for item in lista]
>>> print(nova_lista)
[2, 4, 6]
```

Para adicionarmos um filtro, colocaremos a condicionante **if** após a lista. Sua sintaxe será:

Python

```
[item for item in lista if item cond]
```

Nesse caso, **cond** é uma condicionante. Por exemplo: item > 0, item == 'a', item.startswith('b') etc.

No exemplo adiante, vamos filtrar apenas os elementos pares de uma lista, ou seja, cujo resto da divisão por dois seja zero (%2 == 0):

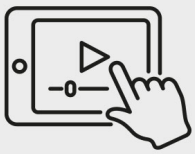
Python

```
>>> lista = [0, 1, 2, 3, 4, 5, 6]
>>> nova_lista = [item for item in lista if item%2 == 0]
>>> print(nova_lista)
[0, 2, 4, 6]
```

Criando cópias de listas (Shallow Copy versus Deep Copy)

Demonstraremos neste vídeo a forma de se copiar uma lista mostrando a diferença entre a Shallow Copy e Deep Copy.

Para assistir a um vídeo sobre o assunto, acesse a versão online deste conteúdo.



Falta pouco para atingir seus objetivos.

Vamos praticar alguns conceitos?

Questão 1

Qual é o valor correto de minha_lista?

```python
>>> minha_lista = [ 0.5*item for item in [0, 1, 2, 3] ]
```

A [0, 1, 2, 3]

B [0.0, 0.5, 1.0, 1.5]

C [0, 1, 2, 3, 4]

D [0.0, 0.5, 1.0, 1.5, 2.0]

E [0, 1, 1, 2]

Parabéns! A alternativa B está correta.

No exercício, utilizamos a compreensão de listas para criar uma lista. Vamos iterar a lista [0,1,2,3] e aplicar a transformação elemento*0.5. Com isso, obtemos [0.0, 0.5, 1.0, 1.5] como resultado.

Questão 2

Observe este programa:

```python
>>> lista = [10, 20, 30, 40]
```

```
>>> lista.append([1, 2, 3, 4])
>>> print(len(lista))
```

Qual será a saída do programa?

A 5

B 4

C 8

D 12

E 2

Parabéns! A alternativa A está correta.

O método append adiciona um único elemento à lista, mesmo que seja outra lista. A lista final será [10, 20, 30, 40, [1, 2, 3, 4]], que é composta pelos elementos 10, 20, 30, 40 e [1, 2, 3, 4], ou seja, 5 elementos. Lembre-se de que, para concatenarmos duas listas, utilizamos o método extends.

3 - Pilhas, Filas, Tuplas e Conjuntos

Ao final deste módulo, você será capaz de reconhecer as estruturas de dados Pilha, Fila, Tupla e Conjunto.

Visão geral

Introdução

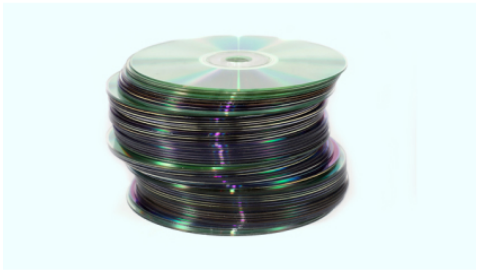
Neste módulo, você será apresentado às estruturas de dados Pilha, Fila, Tupla e Set e entenderá como elas são implementadas em Python. Além disso, conhecerá cada uma delas e seus principais métodos e características.

Algumas estruturas de dados são muito semelhantes, diferindo principalmente na forma como são utilizadas. Esse é o caso das Listas, Pilhas e Filas. Conheceremos de forma mais detalhada cada uma delas em Python.

Estrutura de dados Pilha

Pilhas

Para entender o que é a estrutura de dados **Pilha**, imagine que tenhamos um monte de CDs **empilhados**.



Para adicionarmos um novo CD à nossa pilha, precisamos colocá-lo pela parte de cima, ou seja, pelo **topo**. Para remover um CD, também precisamos fazê-lo pelo **topo**.

Não temos como retirar um CD do meio da pilha sem antes tirar todos os outros acima. Você concorda?

Computacionalmente falando, a estrutura de dados Pilha funciona exatamente como foi ilustrado anteriormente. Os elementos são inseridos ou removidos sempre a partir de um único ponto: o topo.

Assim como as listas, as pilhas são não homogêneas, dinâmicas e lineares. O princípio de funcionamento de uma pilha é o LIFO (do inglês *last in*, *first out*, ou seja, último a entrar, primeiro a sair).

As operações de inserção e remoção de um item na pilha são normalmente chamadas de ***push*** e ***pop***. Elas fazem as seguintes inserções:

ilha

A Pilha é um caso particular de lista em que só é possível adicionar e remover elementos por apenas uma das extremidades da Lista.

ɔpo

O topo de uma pilha é o local onde fica o último elemento inserido e no qual não temos como recuperar um elemento do meio da pilha sem desempilhar todos os anteriores.

Operação *push*

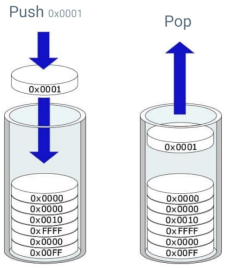
Insere um item no topo da pilha.

Operação *pop*

Remove o item do topo da pilha.

Incluindo e removendo elementos na pilha

Agora observe as seguintes imagens que ilustram o momento exato em que um elemento é inserido e removido de uma mesma pilha por meio dos movimentos de *push* e *pop*:



Movimentos do comando *push* e *pop*.

Para acessar determinado elemento, faça o seguinte:

> **Execute o comando pop até encontrar o elemento que se deseja acessar.**

O Python não contém uma classe específica para Pilha, porém a classe **list** contém dois métodos que fazem com que a lista se comporte como uma pilha. Observe cada um deles:

Append

Esse método inclui um elemento sempre ao final da lista. Ele faz o papel do **push**.

Pop

Esse método retorna e remove o último elemento da lista. Ele faz o papel do próprio pop.

Com esses dois métodos, é possível adicionar e remover elementos por apenas uma das extremidades da lista, fazendo com que ela se comporte exatamente como uma pilha. Como a nossa pilha é um objeto do tipo **list**, para inicializá-la, fazemos exatamente como nas listas.

Manipulando os dados da pilha

No exemplo adiante, vamos iniciar uma pilha vazia e utilizar os métodos **append** e **pop** para modificá-la:

Python

```
>>> pilha = []
>>> pilha.append(1)
>>> pilha.append(2)
>>> pilha.append(3)
>>> print(pilha)
[1, 2, 3]
>>> pilha.append(4)     ← último elemento inserido
>>> print(pilha)
[1, 2, 3, 4]
>>> pilha.pop()
4     ← retorna e remove o último elemento
>>> print(pilha)
[1, 2, 3]     ← o 3 passa a ser o último elemento
>>> pilha.pop()
3     ← retorna e remove o último elemento
>>> print(pilha)
[1, 2]     ← o 2 passa a ser o último elemento
```

Um exemplo prático de utilização de pilhas é o histórico de navegação. Quando entramos em um site e clicamos em menus, abas e links, estamos empilhando todas essas páginas no histórico. Ao apertarmos o botão **voltar** do navegador, estamos realizando um **pop**, removendo a última página e voltando para a anterior.

Estrutura de dados Fila

Filas

As filas estão presentes no nosso dia a dia, seja para adentrar o cinema ou para entrar no banco a fim de pagar uma conta. Em uma fila tradicional, quando uma nova pessoa entra, ela precisa ser **adicionada** na última posição, isto é, no **fim** da fila. Já a próxima pessoa a ser atendida, ou seja, **removida** da fila, é sempre a primeira, a que está no **início** da fila.

Computacionalmente falando, a estrutura de dados Fila funciona exatamente como foi ilustrado anteriormente.

Os elementos são **inseridos** sempre no **final** da fila, enquanto, para **remover** um elemento, é necessário fazê-lo pela **frente** da fila. O **final** de uma fila é aquele em que fica o último elemento inserido. Não temos como recuperar um elemento do meio da fila sem antes remover todos aqueles à sua frente.

ila

A fila é um caso particular de lista na qual adicionamos elementos por uma extremidade e os removemos pela outra da lista.

Incluindo e removendo elementos da fila

Observe a figura a seguir, que ilustra os movimentos de exclusão e inclusão dos elementos de início e fim:

Assim como as listas, as filas são:

- Não homogêneas;
- Dinâmicas;
- Lineares.

Atenção!

O princípio de funcionamento de uma fila é o FIFO. As operações de inserção e remoção de um item na fila são normalmente chamadas de enqueue e dequeue. O enqueue insere um item no final da fila, enquanto o dequeue o remove do início dela.

Para acessar determinado elemento, é preciso executar o comando ***dequeue*** até encontrar o elemento desejado.

Com esses dois métodos, é possível adicionar elementos por uma extremidade e removê-los por outra da lista, fazendo com que se comporte exatamente como uma fila. Como a nossa fila é um objeto do tipo **list**, para inicializá-la, fazemos exatamente como nas listas.

Manipulando os dados da fila

No exemplo a seguir, vamos iniciar uma fila vazia e utilizar os métodos **append** e **pop** para modificar nossa fila.

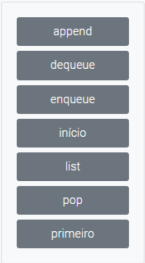
Python

```
>>> fila = []
>>> fila.append(1)
>>> fila.append(2)
>>> fila.append(3)
```

```
>>> print(fila)
[1, 2, 3]
>>> fila.append(4)     ← último elemento inserido
>>> print(fila)
[1, 2, 3, 4]
>>> fila.pop(0)
1     ← retorna e remove o primeiro elemento
>>> print(fila)
[2, 3, 4]     ← o 2 passa a ser o primeiro elemento
>>> fila.pop()
```

Atividade discursiva

Complete as frases com as palavras do quadro:



O Python não contém uma classe específica para filas, porém a classe _____ contém dois métodos que fazem com que a lista se comporte como uma fila.

O primeiro método é o _____, que já vimos anteriormente. Esse método inclui um elemento sempre ao final da fila. Ele faz o papel do _____.

O segundo método é o _____, mas com o parâmetro 0 pop(0), que retorna e remove o _____ elemento inserido, ou seja, o elemento do _____ da fila. Ele faz o papel do _____.

Digite sua resposta aqui

Exibir solução

A sequência correta é:

O Python não contém uma classe específica para filas, porém a classe **list** contém dois métodos que fazem com que a lista se comporte como uma fila.

O primeiro método é o **append**, que já vimos anteriormente. Esse método inclui um elemento sempre ao final da fila. Ele faz o papel do **enqueue**.

O segundo método é o **pop**, porém com parâmetro 0 pop(0), que retorna e remove o **primeiro** elemento inserido, ou seja, o elemento do início da fila. Ele faz o papel do **dequeue**.

Estrutura de dados Tupla

Tuplas

Em Python, a classe responsável por implementar Tuplas se chama **tuple**.

uple

É um dos tipos sequenciais básicos do Python, assim como **list** e **range**. Tipo de dado interno ao Python, ela não precisa ser importada.

> **Quanto às suas características, as Tuplas são não homogêneas, pois podem conter elementos de diferentes tipos (inteiro, decimal, caracteres etc.); estáticas, porque seus tamanhos não podem variar durante a execução do programa; e lineares, já que têm seus dados organizados de forma sequencial.**

As Tuplas são muito parecidas com a lista, como veremos a seguir:

Definindo uma tupla

Para inicializarmos um objeto do tipo tuple, podemos proceder de diversas formas:

Utilizar apenas os parênteses para iniciar uma tupla vazia

a. tupla = ()

Utilizar elementos separados por vírgula com ou sem parênteses

a. tupla = (a, b, c)
b. tupla = a, b, c

Para iniciar uma tupla de apenas um item, colocar uma vírgula após esse item

a. tupla = (a,)
b. tupla = a,

Utilizar o construtor tuple(), que aceita como parâmetro outras coleções

a. tupla = tuple()
b. tupla = tuple([1,2,3])
c. tupla = tuple((1,2,3))

Assim como nas listas, os elementos de uma tupla em Python podem ser de diferentes tipos, incluindo outra tupla, como no exemplo a seguir, em que temos como elementos um número inteiro, uma string, uma lista e uma outra tupla.

```python
tupla = (1, 'Hello', [1, 2], (3,4))
```

Diferentemente das listas, as Tuplas são sequências **imutáveis**.

> **Elas não podem ser alteradas durante a execução do programa.**

Acessando os dados da Tupla

A forma de acessar um elemento da Tupla é igual à das listas: precisamos apenas do índice do elemento.

No exemplo adiante, vamos imprimir o valor do elemento na posição 2, lembrando que o **primeiro elemento da tupla tem índice 0**:

```python
>>> tupla = ('a', 'b', 'c')
>>> item = tupla[2]
>>> print(item)
'c'
```

Assim como nas listas, podemos utilizar o método **index** para encontrar o índice de um elemento.

```python
>>> tupla = ('a', 'b', 'c')
>>> indice_a = tupla.index('a')
>>> print(indice_a)
0
```

Como as tuplas são imutáveis, se tentarmos alterar um elemento utilizando o índice, o sistema retornará um erro (TypeError).

```python
>>> tupla = ('a', 'b', 'c')
>>> tupla[2] = 'd'
Traceback (most recent call last):
File " <stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

Iterando a Tupla

As Tuplas são iteráveis (*iterable*), ou seja, capazes de retornar seus elementos um de cada vez. Para percorrermos as Tuplas, utilizaremos a mesma sintaxe das listas, como ocorre no exemplo a seguir:

```python
>>> tupla = ('a', 'b', [1, 2, 3], (1, 2, 3))
>>> for elemento in tupla:
>>>     print(elemento)
a
b
[1, 2, 3]
```

Alterar a Tupla

Como as Tuplas não podem ser alteradas, nenhum dos métodos apresentados para as listas que implicam alguma modificação está presente nas Tuplas, incluindo o **sort** e o **reverse**.

Outras funções da Tupla

As funções internas do Python aplicadas às listas também se aplicam às Tuplas: **len**, **min**, **max** e **sum**, assim como os operadores de pertinência (**in** e **not in**), equivalência (**==**), concatenação (**+**) e multiplicação (*), também podem ser aplicados aos objetos do tipo tuple. Isso se dá porque todas essas funções e operadores retornam um novo valor e não alteram a tupla.

Comentário

Por fim, vamos comparar as Tuplas com as Listas:

As Tuplas são mais eficientes que as Listas;
As Tuplas podem ser utilizadas como chave para a estrutura de dados Hash implementada em Python, como dicionário, classe **dict**;
As Tuplas podem ser incluídas em conjuntos (*sets*).

Muitos desenvolvedores veem as Tuplas apenas como listas estáticas, mas os elementos delas normalmente têm um significado. A alteração desses elementos, portanto, pode quebrar a semântica, como, por exemplo, armazenar o dia, o mês e o ano de nascimento de uma pessoa.

Dando significado às Tuplas – utilizando as Named Tuples

Veremos neste vídeo como funciona o uso das Tuplas mediante a utilização das Named Tuplas.

Para assistir a um vídeo sobre o assunto, acesse a versão online deste conteúdo.

Estrutura de dados Conjunto

Conjuntos (Set)

Em Python, a classe responsável por implementar os Conjuntos se chama Set.

et

Coleção não ordenada cujos elementos são únicos e imutáveis. Ela é um tipo de dado interno ao Python e não precisa ser importada.

> **Os Conjuntos podem ser utilizados para remover itens duplicados de outras sequências ou testes de pertinência e computar operações matemáticas, como união e interseção, entre outras.**

Quanto às suas características, os Conjuntos são:

Não homogêneos

Pois podem conter elementos de diferentes tipos (inteiro, decimal, caracteres etc.).

Dinâmicos

Já que seus tamanhos podem variar durante a execução do programa quando inserimos e removemos elementos.

Não lineares

Porque seus dados não são organizados de forma sequencial.

Definindo um conjunto

Para inicializarmos um objeto do tipo Set, podemos proceder de duas formas:

Utilizar chaves com elementos separados por vírgula

a. conjunto = {1, 2, 3}

Utilizar o construtor set(), que aceita como parâmetro outras coleções

a. conjunto = set()
b. conjunto = set((1,2,3))
c. conjunto = set([1,2,3])

Assim como nas Listas, os elementos de um conjunto podem ser de diferentes tipos, desde que o **próprio elemento e seus filhos sejam imutáveis**, como no exemplo a seguir, em que temos como elementos um número inteiro, uma string e uma tupla:

```python
conjunto = {1, 'Hello', (3,4)}
```

Apesar de os elementos de um conjunto serem imutáveis, os Conjuntos são coleções **mutáveis**, ou seja, eles podem ser alterados durante a execução do programa.

Acessando os dados do Conjunto

Como os Conjuntos não são organizados sequencialmente, eles não armazenam seus elementos em índices; com isso, não podemos acessá-los diretamente pela sua posição, como fazemos nas listas. Para recuperarmos um elemento, precisamos iterar sobre o conjunto ou utilizar o método **pop**, que será apresentado a seguir.

Iterando o Conjunto

Os Conjuntos, assim como as Listas, também são iteráveis (*iterable*), ou seja, capazes de retornar seus elementos um de cada vez. Para percorrermos os Conjuntos, utilizamos a mesma sintaxe das Listas, como se pode verificar no exemplo a seguir:

```python
>>> conjunto = {'a', 1, 2020, (28, 12), 'a'}     ← itens duplicados são automaticamente removidos
>>> for elemento in conjunto:
...  print(elemento)
1
(28, 12)
2020
```

Como os Conjuntos não armazenam seus elementos de forma sequencial, eles são retornados de forma aleatória.

Alterando o Conjunto

Por ser dinâmico, o objeto do tipo **Set** contém um conjunto de métodos que permite alterar seus elementos. Veremos adiante seus principais métodos.

Quando precisamos inserir um novo elemento no Conjunto, utilizamos o método **add**. Esse método vai inserir o elemento passado como parâmetro se ele ainda não existir no conjunto.

> **Lembre-se de que os Conjuntos não contêm objetos repetidos.**

Confira o código a seguir para verificar como se utiliza esse método:

```python
>>> conjunto = set()
>>> conjunto.add(1)
>>> conjunto.add(2)
>>> print(conjunto)
{1, 2}
>>> conjunto.add(2)     ← por não conter itens duplicados, a inserção é ignorada
```

Além do método para adicionar, o objeto do tipo **Set** contém métodos para remover elementos do conjunto. Utilizamos o método **discard** e **remove** para remover elementos do Set.

Nos dois métodos, o elemento é passado como parâmetro. A diferença é que, caso o elemento não exista, o remove **lança** um erro, enquanto o discard **ignora** o comando.

Também podemos utilizar o **método clear** para remover todos os elementos do conjunto:

```python
>>> conjunto = {1, 2, 3}
>>> conjunto.discard(1)
>>> conjunto.remove(2)
>>> print(conjunto)
{3}
>>> conjunto.discard(5)     ← o elemento 5 não existe!
```

Podemos extrair itens de forma aleatória do Conjunto utilizando o método **pop** mediante a simulação de um sorteio. Confira no exemplo a seguir:

```python
>>> conjunto = {'a', 1, 2, 3}
>>> conjunto.pop()
3
>>> print(conjunto)
{1, 'a', 2}
>>> conjunto.pop()
```

Outras funções do Conjunto

As funções internas do Python que se aplicam às Listas também se aplicam aos Conjuntos: **len**, **min**, **max** e **sum**. Os operadores de pertinência (**in** e **not in**) e equivalência (**==**) também podem ser aplicados aos objetos do tipo Set.

Os objetos do tipo Set disponibilizam alguns métodos relacionados a operações matemáticas. No exemplo a seguir, mostraremos a utilização de alguns deles: *union* (união), *intersection* (intercessão) e *difference* (diferença). Todos esses métodos retornam um novo conjunto.

Python

```
>>> conjunto_a = {1, 2, 3}
>>> conjunto_b = {4, 5, 6}
>>> conjunto_b = {3, 4, 5}
>>> uniao = conjunto_a.union(conjunto_b)
>>> print(uniao)
{1, 2, 3, 4, 5}
```

O Python permite utilizar operadores para realizar essas mesmas operações:

- operador |: união;
- operador &: interseção;
- operador -: diferença.

Falta pouco para atingir seus objetivos.

Vamos praticar alguns conceitos?

Questão 1

(FAURGS - 2018) Observe uma estrutura de dados em forma de tabela. Nela, foi realizada uma série de operações de inserção e retirada de elementos, conforme descrito e ilustrado abaixo:

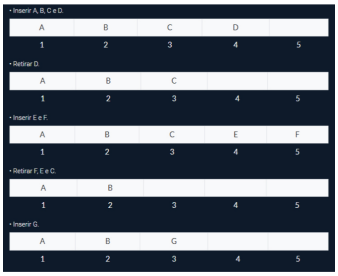


Tabela: Estrutura de dados.
Frederico Tosta de Oliveira

Pode-se deduzir, pelas operações realizadas, que tal estrutura é:

A Uma lista encadeada.

B Um vetor.

C Uma fila.

D Uma pilha.

E Um conjunto.

Parabéns! A alternativa D está correta.

Observe que os últimos elementos inseridos são os primeiros a serem retirados, obedecendo à lógica LIFO (*last in, first out*), que é a lógica da Pilha.

Questão 2

Observe os conjuntos 1 e 2 definidos a seguir:

Python

```
>>> conjunto1 = {100, 200, 300}
>>> conjunto2 = {200, 400, 500}
```

Como podemos obter os itens que estão em 1, mas não estão em 2, de forma a obter o conjunto {100, 300} como resultado?

A conjunto1.add(conjunto2)

B conjunto1.difference(conjunto2)

C conjunto1.union(conjunto2)

D conjunto1.intersection(conjunto2)

E conjunto2.intersection(conjunto1)

Parabéns! A alternativa B está correta.

Essa operação se chama diferença de conjuntos. Ela busca os elementos que estão no conjunto1 e não estão no conjunto2.

Considerações finais

Assim como na vida real, demonstramos neste conteúdo que os dados e as informações em um computador precisam ser armazenados e organizados de forma bem definida a fim de permitir que essas informações sejam encontradas e recuperadas em tempo hábil.

Identificar a melhor estrutura para armazenar os dados em um sistema é muito importante no desenvolvimento de software. Por isso, frisamos que a escolha certa da estrutura de dados, além de facilitar o desenvolvimento, pode impactar diretamente na performance do sistema.

Ouça este podcast e veja alguns aspectos importantes das estruturas de dados básicas do Python.

Para ouvir o *áudio*, acesse a versão online deste conteúdo.

Referências

CORMEN, T. H.; LEISERSON, C. E.; RIVEST, R. L.; STEIN, C. **Algoritmos**: teoria e prática. Rio de Janeiro: Campus, 2002.

PYTHON. **Python software foundation**. Consultado na internet em: 11 ago. 2020.

Explore +

Para saber mais sobre os assuntos tratados neste conteúdo, acesse duas bibliotecas:

NumPy;

Pandas (biblioteca muito utilizada em computação científica, ela é uma ferramenta extremamente otimizada para a análise e a manipulação de dados).