LINGUAGEM DE PROGRAMAÇÃO

Linguagem de programação: Estrutura de dados em Python

Prof. Me. Wesley Viana

- Unidade de Ensino: 02
- Competência da Unidade: Conhecer a linguagem de programação Python
- Resumo: Saber utilizar modelos de estrutura de dados na linguagem Python.
- Palavras-chave: Algoritmos; Python; Ordenação; Estrutura de dados; Busca.
- Título da Teleaula: Linguagem de programação: Estrutura de dados em Python
- Teleaula nº: 02

Contextualização

- Linguagem de programação: Estrutura de dados em Python
- Estrutura de dados
- Algoritmos de busca
- Algoritmos de ordenação

Linguagem de programação: Estrutura de dados em Python

Estrutura de dados em Python

Em Python existem objetos em que podemos armazenar mais de um valor, aos quais damos o nome de estruturas de dados.

Tudo em Python é um objeto. Já conhecemos alguns tipos de objetos em Python, tais como o int (inteiro), o str (string), o float (ponto flutuante).Os tipos de estruturas de dados: listas, tuplas, conjuntos, dicionário e matriz.

Vamos estudar esses objetos na seguinte ordem:

- Objetos do tipo sequência: texto, listas e tuplas.
- Objetos do tipo set (conjunto).
- Objetos do tipo mapping (dicionário).
- Objetos do tipo array NumPy.

Estrutura de dados em Python

Objetos do tipo sequência

Essas estruturas de dados representam sequências finitas indexadas por números não negativos. O primeiro elemento de uma sequência ocupa o índice 0; o segundo, 1; o último elemento, a posição n - 1, em que n é capacidade de armazenamento da sequência.

Sequência de caracteres

Um texto é um objeto da classe str (strings), que é um tipo de sequência. Os objetos da classe str possuem todas as operações, mas são objetos imutáveis, razão pela qual não é possível atribuir um novo valor a uma posição específica.

Estrutura de dados em Python

```
texto = "Aprendendo Python na disciplina de linguagem de programação."

print(f"Tamanho do texto = {len(texto)}")
print(f"Python in texto = {'Python' in texto}")
print(f"Quantidade de y no texto = {texto.count('y')}")
print(f"As 5 primeiras letras são: {texto[0:6]}")
```

Na entrada 1, usamos algumas operações das sequências. A operação len() permite saber o tamanho da sequência. O operador 'in', por sua vez, permite saber se um determinado valor está ou não na sequência.

O operador count permite contar a quantidade de ocorrências de um valor. E a notação com colchetes permite fatiar a sequência, exibindo somente partes dela. Na linha 6, pedimos para exibir da posição 0 até a 5, pois o valor 6 não é incluído

Estrutura de dados em Python

Vamos falar agora sobre a função split(), usada para "cortar" um texto e transformá-lo em uma lista. Essa função pode ser usada sem nenhum parâmetro: texto.split(). Nesse caso, a string será cortada a cada espaço em branco que for encontrado. Caso seja passado um parâmetro: texto.split(","), então o corte será feito no parâmetro especificado.

```
texto = "Aprendendo Python na disciplina de linguagem de programação."
print(f"texto = {texto}")
print(f"Tamanho do texto = {len(texto)}\n")

palavras = texto.split()
print(f"palavras = {palavras}")
print(f"Tamanho de palavras = {len(palavras)}")
```

Estrutura de dados em Python

Listas

É uma estrutura de dados do tipo sequencial que possui como principal característica ser mutável. Ou seja, novos valores podem ser adicionados ou removidos da sequência. Em Python, as listas podem ser construídas de várias maneiras:

- Usando um par de colchetes para denotar uma lista vazia: lista1 = []
- Usando um par de colchetes e elementos separados por vírgulas: lista2 = ['a', 'b', 'c']
- Usando uma "list comprehension": [x for x in iterable]
- Usando o construtor de tipo: list()

Estrutura de dados em Python

Por meio da estrutura de repetição, imprimimos cada elemento da lista juntamente com seu índice. Veja que a sequência possui a função index, que retorna a posição de um valor na sequência.

```
vogais = ['a', 'e', 'i', 'o', 'u'] # também poderia ter sido criada usando aspas duplas

for vogal in vogais:
    print (f'Posição = {vogais.index(vogal)}, valor = {vogal}')
```

As listas possuem diversas funções, além das operações já mencionadas. Na documentação oficial (PSF, 2020b) você encontra uma lista completa com todas as operações possíveis.

Estrutura de dados em Python

List comprehension (Compreensões de lista)

A list comprehension, também chamada de listcomp.

Esse tipo de técnica é utilizada quando, dada uma sequência, deseja-se criar uma nova sequência, porém com as informações originais transformadas ou filtradas por um critério.

Estrutura de dados em Python

```
linguagens = ["Python", "Java", "JavaScript", "C", "C#", "C++", "Swift", "Go", "Kotlin"]

#linguagens = '''Python Java JavaScript C C# C++ Swift Go Kotlin'''.split() # Essa sintaxe produz o mesmo resultado que a linha 1


print("Antes da listcomp = ", linguagens)


linguagens = [item.lower() for item in linguagens]


print("\nDepois da listcomp = ", linguagens)
```

Criamos uma lista chamada "linguagens" que contém algumas linguagens de programação. Na linha 2, deixamos comentado outra forma de criar uma lista com base em um texto com utilização da função split(). Na linha 6, criamos nossa primeira list comprehension.

Observação: como se trata da criação de uma lista, usam-se colchetes! Dentro do colchetes há uma variável chamada "item" que representará cada valor da lista original. Veja que usamos item.lower() for item in linguagens, e o resultado foi guardado dentro da mesma variável original, ou seja, sobrescrevemos o valor de "linguagens".

Estrutura de dados em Python

Funções map() e filter()

Funções built-in que são usadas por esse tipo de estrutura de dados: map() e filter().

A função map() é utilizada para aplicar uma determinada função em cada item de um objeto iterável.

Para que essa transformação seja feita, a função map() exige que sejam passados dois parâmetros: a função e o objeto iterável.

Estrutura de dados em Python

Criamos uma lista na linha 3; e, na linha 5, aplicamos a função map() para transformar cada palavra da lista em letras minúsculas. Veja que, como o primeiro parâmetro da função map() precisa ser uma função, optamos por usar uma função lambda. Na linha 6, imprimimos a nova lista.

A função map retorna um objeto iterável. Para que possamos ver o resultado, precisamos transformar esse objeto em uma lista. Fazemos isso na linha 8 ao aplicar o construtor list() para fazer a conversão. Por fim, na linha 9, fazemos a impressão da nova lista e, portanto, conseguimos ver o resultado.

```
# Exemplo

print("Exemplo")

linguagens = '''Python Java JavaScript C C#
C++ Swift Go Kotlin'''.split()


nova_lista = map(lambda x: x.lower(),
linguagens)

print(f"A nova lista é = {nova_lista}\n")


nova_lista = list(nova_lista)

print(f"Agora sim, a nova lista é =
{nova_lista}")
```

Estrutura de dados em Python

A função range() cria um objeto numérico iterável. Então usamos o construtor list() para transformá-lo em uma lista com números, que variam de 0 a 20.

Lembre-se de que o limite superior do argumento da função range() não é incluído. Na linha 3, criamos uma nova lista com a função filter, que, com a utilização da expressão lambda, retorna somente os valores pares.

```
numeros  = list(range(0, 21))


numeros_pares = list(filter(lambda x: x % 2
== 0, numeros))


print(numeros_pares)
```

Estrutura de dados em Python

Tuplas

A grande diferença entre listas e tuplas é que as primeiras são mutáveis, razão pela qual, com elas, conseguimos fazer atribuições a posições específicas: por exemplo, lista[2] = 'maça'. Por sua vez, nas tuplas isso não é possível, uma vez que são objetos imutáveis.

Em Python, as tuplas podem ser construídas de três maneiras:

Usando um par de parênteses para denotar uma tupla vazia: tupla1 = ()

Usando um par de parênteses e elementos separados por vírgulas: tupla2 = ('a', 'b', 'c')

Usando o construtor de tipo: tuple()

Estrutura de dados em Python

"Não vi diferença nenhuma entre usar lista e usar tupla". Em alguns casos, mais de uma estrutura realmente pode resolver o problema, mas em outros não.

Como a tupla é imutável, sua utilização ocorre em casos nos quais a ordem dos elementos é importante e não pode ser alterada, já que o objeto tuple garante essa característica. A função enumerate(), que normalmente usamos nas estruturas de repetição, retorna uma tupla cujo primeiro elemento é sempre o índice da posição e cujo segundo elemento é o valor em si.

```
vogais = ('a', 'e', 'i', 'o', 'u')
print(f"Tipo do objeto vogais = {type(vogais)}")

for p, x in enumerate(vogais):
    print(f"Posição = {p}, valor = {x}")
```

Estrutura de dados em Python

Objetos do tipo Set

A tradução "conjunto" para set nos leva diretamente à essência desse tipo de estrutura de dados em Python.

Um objeto do tipo set habilita operações matemáticas de conjuntos, tais como: união, intersecção, diferença, etc. Esse tipo de estrutura pode ser usado, portanto, em testes de associação e remoção de valores duplicados de uma sequência (PSF, 2020c).

Das operações que já conhecemos sobre sequências, conseguimos usar nessa nova estrutura:

len(s)

x in s

x not in s

Estrutura de dados em Python

Além dessas operações, podemos adicionar um novo elemento a um conjunto com a função add(valor). Também podemos remover com remove(valor).

Em Python, os objetos do tipo set podem ser construídos destas maneiras:

Usando um par de chaves e elementos separados por vírgulas: set1 = {'a', 'b', 'c'}

Usando o construtor de tipo: set(iterable)

Estrutura de dados em Python

Não é possível criar um set vazio, com set = {}, pois essa é a forma de construção de um dicionário.

Para construir com utilização da função set(iterable), obrigatoriamente temos de passar um objeto iterável para ser transformado em conjunto.

Esse objeto pode ser uma lista, uma tupla ou até mesmo uma string (que é um tipo de sequência).

Estrutura de dados em Python

Criamos 4 exemplos de construção de objetos set. Com exceção do primeiro, no qual não usamos o construtor set(), os demais resultam na mesma estrutura. Veja que, no exemplo 2, passamos como parâmetro uma sequência de caracteres 'aeiouaaaaaa' e, propositalmente, repetimos a vogal a. O construtor interpreta como um iterável e cria um conjunto em que cada caractere é um elemento, eliminando valores duplicados. Veja, na linha 13, o exemplo no qual transformamos a palvra 'banana' em um set, cujo resultado é a eliminação de caracteres duplicados.

```
vogais_1 = {'aeiou'} # sem uso do construtor

print(type(vogais_1), vogais_1)

vogais_2 = set('aeiouaaaaaa') # construtor com string

print(type(vogais_2), vogais_2)

vogais_3 = set(['a', 'e', 'i', 'o', 'u']) # construtor com lista

print(type(vogais_3), vogais_3)

vogais_4 = set(('a', 'e', 'i', 'o', 'u')) # construtor com tupla

print(type(vogais_4), vogais_4)


print(set('banana'))
```

Estrutura de dados em Python

Objetos do tipo mapping

As estruturas de dados que possuem um mapeamento entre uma chave e um valor são consideradas objetos do tipo mapping. Em Python, o objeto que possui essa propriedade é o dict (dicionário). Uma vez que esse objeto é mutável, conseguimos atribuir um novo valor a uma chave já existente.

Podemos construir dicionários em Python das seguintes maneiras:

Usando um par de chaves para denotar um dict vazio: dicionario1 = {}

Usando um par de elementos na forma, chave : valor separados por vírgulas: dicionario2 = {'one': 1, 'two': 2, 'three': 3}

Usando o construtor de tipo: dict()

Estrutura de dados em Python

```
# Exemplo 1 - Criação de dicionário vazio, com atribuição posterior de chave e valor

dici_1 = {}

dici_1['nome'] = "João"

dici_1['idade'] = 30


# Exemplo 2 - Criação de dicionário usando um par elementos na forma, chave : valor

dici_2 = {'nome': 'João', 'idade': 30}


# Exemplo 3 - Criação de dicionário com uma lista de tuplas. Cada tupla representa um par chave : valor

dici_3 = dict([('nome', "João"), ('idade', 30)])
```

Usamos 4 sintaxes distintas para criar e atribuir valores a um dicionário. Da linha 2 à linha 4, criamos um dicionário vazio e, em seguida, criamos as chaves e atribuímos valores.

Na linha 7, já criamos o dicionário com as chaves e os valores. Na linha 10, usamos o construtor dict() para criar, passando como parâmetro uma lista de tuplas: dict([(tupla 1), (tupla 2)]).

Cada tupla deve ser uma combinação de chave e valor. Na linha 13, também usamos o construtor dict(), mas agora combinado com a função built-in zip().

Estrutura de dados em Python

Exemplo 4 - Criação de dicionário com a função built-in zip() e duas listas, uma com as chaves, outra com os valores.

dici_4 = dict(zip(['nome', 'idade'], ["João", 30]))

print(dici_1 == dici_2 == dici_3 == dici_4) # Testando se as diferentes construções resultamo em objetos iguais.

A função zip() é usada para combinar valores de diferentes sequências e retorna um iterável de tuplas, em que o primeiro elemento é referente ao primeiro elemento da sequência 1, e assim por diante.

Na linha 16, testamos se as diferentes construções produzem o mesmo objeto. O resultado True para o teste indica que são iguais.

Estrutura de dados em Python

Objetos do tipo array NumPy

O caso da biblioteca NumPy, criada especificamente para a computação científica com Python. O NumPy contém, entre outras coisas:

- Um poderoso objeto de matriz (array) N-dimensional.
- Funções sofisticadas.
- Ferramentas para integrar código C/C++ e Fortran.
- Recursos úteis de álgebra linear, transformação de Fourier e números aleatórios.

Sem dúvida, o NumPy é a biblioteca mais poderosa para trabalhar com dados tabulares (matrizes), além de ser um recurso essencial para os desenvolvedores científicos, como os que desenvolvem soluções de inteligência artificial para imagens.

Estrutura de dados em Python

Criamos várias formas de matrizes com a biblioteca NumPy. Veja que, na linha 1, importamos a biblioteca para que pudéssemos usar seus objetos e funções. Para criar uma matriz, usamos numpy.array(forma), em que forma são listas que representam as linhas e colunas.

Veja que, nas linhas 5 e 6, criamos matrizes, respectivamente, com 3 linhas e 2 colunas e 2 linhas e 3 colunas. O que mudou de uma construção para a outra é que, para construir 3 linhas com 2 colunas, usamos três listas internas com dois valores, já para construir 2 linhas com 3 colunas, usamos duas listas com três valores cada.

```
import numpy

matriz_1_1 = numpy.array([1, 2, 3]) # Cria matriz 1 linha e 1 coluna

matriz_2_2 = numpy.array([[1, 2], [3, 4]]) # Cria matriz 2 linhas e 2 colunas

matriz_3_2 = numpy.array([[1, 2], [3, 4], [5, 6]]) # Cria matriz 3 linhas e 2 colunas

matriz_2_3 = numpy.array([[1, 2, 3], [4, 5, 6]]) # Cria matriz 2 linhas e 3 colunas

print(type(matriz_1_1))

print('\n matriz_1_1 = ', matriz_1_1)

print('\n matriz_2_2 = \n', matriz_2_2)

print('\n matriz_3_2 = \n', matriz_3_2)

print('\n matriz_2_3 = \n', matriz_2_3)
```

ALGORITMOS DE BUSCA: Analisando os comandos dos mecanismos de buscas

Algoritmos de busca

Esse universo, como o nome sugere, os algoritmos resolvem problemas relacionados ao encontro de valores em uma estrutura de dados. Em Python, temos a operação "in" ou "not in" usada para verificar se um valor está em uma sequência.

```
nomes = 'João Marcela Sonia Daryl Vernon Eder Mechelle Edan Igor Ethan Reed Travis Hoyt'.split()

print('Marcela' in nomes)
print('Roberto' in nomes)
```

Usamos o operador in para verificar se dois nomes constavam na lista. No primeiro, obtivemos True; e no segundo, False.

Algoritmos de busca

Busca linear (ou Busca Sequencial)

Percorre os elementos da sequência procurando aquele de destino, começa por uma das extremidades da sequência e vai percorrendo até encontrar (ou não) o valor desejado. Pesquisa linear examina todos os elementos da sequência até encontrar o de destino, o que pode ser muito custoso computacionalmente.

Para implementar a busca linear, vamos precisar de uma estrutura de repetição (for) para percorrer a sequência, e uma estrutura de decisão (if) para verificar se o valor em uma determinada posição é o que procuramos.

Algoritmos de busca

```
def executar_busca_linear(lista, valor):
    for elemento in lista:
        if valor == elemento:
            return True
    return False
```

Criamos a função "executar_busca_linear", que recebe uma lista e um valor a ser localizado. Na linha 2, criamos a estrutura de repetição, que percorrerá cada elemento da lista pela comparação com o valor buscado (linha 3). Caso este seja localizado, então a função retorna o valor booleano True; caso não seja encontrado, então retorna False.

Algoritmos de busca

Nossa função é capaz de determinar se um valor está ou não presente em uma sequência, certo? E se, no entanto, quiséssemos também saber sua posição na sequência? Em Python, as estruturas de dados do tipo sequência possuem a função index(), que é usada da seguinte forma: sequencia.index(valor). A função index() espera como parâmetro o valor a ser procurado na sequência.

```
vogais = 'aeiou'


resultado = vogais.index('e')
print(resultado)
```

Algoritmos de busca

Complexidade

Em termos computacionais, um algoritmo é considerado melhor que o outro quando, para a mesma entrada, utiliza menos recursos computacionais em termos de memória e processamento. Estudo da viabilidade de um algoritmo, em termos de espaço e tempo de processamento, é chamado de análise da complexidade do algoritmo.

Análise da complexidade é feita em duas dimensões: espaço e tempo.

Podemos, então, concluir que a análise da complexidade de um algoritmo tem como um dos grandes objetivos encontrar o comportamento do algoritmo (a função matemática) em relação ao tempo de execução para o pior caso, ao que chamamos de complexidade assintótica.

Algoritmos de busca

Busca binária

Outro algoritmo usado para buscar um valor em uma sequência é o de busca binária. A primeira grande diferença entre o algoritmo de busca linear e o algoritmo de busca binária é que, com este último, os valores precisam estar ordenados. A lógica é a seguinte:

- Encontra o item no meio da sequência (meio da lista).
- Se o valor procurado for igual ao item do meio, a busca se encerra.
- Se não for, verifica-se se o valor buscado é maior ou menor que o valor central.
- Se for maior, então a busca acontecerá na metade superior da sequência (a inferior é descartada); se não for, a busca acontecerá na metada inferior da sequência (a superior é descartada).

Algoritmos de busca

Veja que o algoritmo, ao encontrar o valor central de uma sequência, a divide em duas partes, o que justifica o nome de busca binária

Suponha que tenhamos uma lista com 1024 elementos. Na primeira iteração do loop, ao encontrar o meio e excluir uma parte, a lista a ser buscada já é diminuída para 512. Na segunda iteração, novamente ao encontrar o meio e excluir uma parte, restam 256 elementos. Na terceira iteração, restam 128. Na quarta, restam 64. Na quinta, restam 32. Na sexta, restam 16. Na sétima 8. Na oitava 4. Na nona 2. Na décima iteração resta apenas 1 elemento. Ou seja, para 1024 elementos, no pior caso, o loop será executado apenas 10 vezes, diferentemente da busca linear, na qual a iteração aconteceria 1024 vezes.

Algoritmos de busca

Veja que o algoritmo, ao encontrar o valor central de uma sequência, a divide em duas partes, o que justifica o nome de busca binária

Suponha que tenhamos uma lista com 1024 elementos. Na primeira iteração do loop, ao encontrar o meio e excluir uma parte, a lista a ser buscada já é diminuída para 512. Na segunda iteração, novamente ao encontrar o meio e excluir uma parte, restam 256 elementos. Na terceira iteração, restam 128. Na quarta, restam 64. Na quinta, restam 32. Na sexta, restam 16. Na sétima 8. Na oitava 4. Na nona 2. Na décima iteração resta apenas 1 elemento. Ou seja, para 1024 elementos, no pior caso, o loop será executado apenas 10 vezes, diferentemente da busca linear, na qual a iteração aconteceria 1024 vezes.

Algoritmos de busca

Nas linhas 2 e 3, inicializamos as variáveis que contêm o primeiro e o último indíce da lista. No começo da execução, esses valores são o índice 0 para o mínimo e o último índice, que é o tamanho da lista menos 1. Essas variáveis serão atualizadas dentro do loop, conforme condição.

Na linha 4 usamos o while como estrutura de repetição, pois não sabemos quantas vezes a repetição deverá ser executada. Esse while fará com que a execução seja feita para todos os casos binários.

Na linha 6, usamos uma equação matemática (a média estatística) para encontrar o meio da lista.

```
def executar_busca_binaria(lista, valor):
    minimo = 0
    maximo = len(lista) - 1
    while minimo <= maximo:
        # Encontra o elemento que divide a lista ao meio
        meio = (minimo + maximo) // 2
        # Verifica se o valor procurado está a esquerda ou direita
do valor central
```

Algoritmos de busca

Na linha 8, checamos se o valor que estamos buscando é menor que o valor encontrado no meio da lista.

Caso seja, então vamos para a linha 9, na qual atualizamos o índice máximo. Nesse cenário, vamos excluir a metade superior da lista original.

Caso o valor não seja menor que o meio da lista, então vamos para a linha 10, na qual testamos se ele é maior. Se for, então atualizamos o menor índice, excluindo assim a metade inferior.

Se o valor procurando não for nem menor nem maior e ainda estivermos dentro do loop, então ele é igual, e o valor True é retornado pelo comando na linha 13.

```
if valor < lista[meio]:
        maximo = meio - 1
    elif valor > lista[meio]:
        minimo = meio + 1
    else:
        return True # Se o valor for encontrado para aqui
return False # Se chegar até aqui, significa que o valor não foi encontrado
```

Porém, se já fizemos todos os testes e não encontramos o valor, então é retornado False na linha 14.

Pense como podemos melhorar o desenvolvimento de um algoritmo

Exercício: Quais as vantagens e limitações da busca sequencial?

Quais as vantagens e limitações da busca sequencial?

Busca Sequencial, é a forma mais simples de busca, percorre-se registro por registro em busca da chave.

Na melhor das hipóteses, a chave de busca estará na posição 0. Portanto, teremos um único acesso em lista[0]. Possui resultados melhores para quantidades pequena e média de buscas.

Na pior das hipóteses, a chave é o último elemento ou não pertence à lista e, portanto, acessamos todos os n elementos da lista. Perda de eficiência para os outros registros, o método é mais "caro".

Algoritmos de Ordenação

Algoritmos de Ordenação

A essência dos algoritmos de ordenação consiste em comparar dois valores, verificar qual é menor e colocar na posição correta.

O que vai mudar, neste caso, é como e quando a comparação é feita. Para que possamos começar a entender a essência dos algoritmos de ordenação.

Em Python, existem duas formas já programadas que nos permitem ordenar uma sequência: a função built-in sorted() e o método sort(), presente nos objetos da classe list.

Algoritmos de Ordenação

A função built-in sorted() cria uma nova lista para ordenar e a retorna, razão pela qual, como resultado da linha 7, há uma lista ordenada, guardada dentro da variável lista_ordenada1. Por sua vez, o método sort(), da classe list, não cria uma nova lista, uma vez que faz a ordenação "inplace" (ou seja, dentro da própria lista passada como parâmetro).

Isso justifica o resultado obtido pela linha 8, que mostra que, dentro da variável lista_ordenada2, está o valor None e, também, o resultado da linha 9, em que pedimos para imprimir a lista que agora está ordenada.

```
lista = [10, 4, 1, 15, -3]


lista_ordenada1 = sorted(lista)

lista_ordenada2 = lista.sort()


print('lista = ', lista, '\n')

print('lista_ordenada1 = ', lista_ordenada1)

print('lista_ordenada2 = ', lista_ordenada2)

print('lista = ', lista)
```

Logo, concluímos que: 1) a função built-in sorted() não altera a lista original e faz a ordenação em uma nova lista; e 2) o método sort() faz a ordenação na lista original com retorno None.

Algoritmos de Ordenação

Na linha 3, que a ordenação consiste em comparar um valor e seu vizinho. Caso o valor da posição mais à frente seja menor, então deve ser feita a troca de posições.

Para fazer a troca, usamos uma forma mais clássica, na qual uma variável auxiliar é criada para guardar, temporariamente, um dos valores (no nosso caso estamos guardando o menor).

Na linha 5, colocamos o valor maior na posição da frente e, na linha 6, resgatamos o valor da variável auxiliar colocando-a na posição correta.

```
lista = [7, 4]


if lista[0] > lista[1]:
    aux = lista[1]
    lista[1] = lista[0]
    lista[0] = aux


print(lista)
```

Algoritmos de Ordenação

Podemos fazer a troca usando a atribuição múltipla.

Nesse caso, a atribuição é feita de maneira posicional: o primeiro valor após o sinal de igualdade vai para a primeira variável, e assim por diante.

```
lista = [5, -1]


if lista[0] > lista[1]:
    lista[0], lista[1] = lista[1], lista[0]


print(lista)
```

Algoritmos de Ordenação

Selection sort (Ordenação por seleção)

O algoritmo selection sort recebe esse nome, porque faz a ordenação sempre escolhendo o menor valor para ocupar uma determinada posição.

A lógica do algoritmo é a seguinte:

- Iteração 1: percorre toda a lista, procurando o menor valor para ocupar a posição 0.
- Iteração 2: a partir da posição 1, percorre toda a lista, procurando o menor valor para ocupar a posição 1.
- Iteração 3: a partir da posição 2, percorre toda a lista, procurando o menor valor para ocupar a posição 2.
- Esse processo é repetido N-1 vezes, sendo N o tamanho da lista.

Algoritmos de Ordenação

Temos uma variável que guarda o tamanho da lista (n). Precisamos de duas estruturas de controle para iterar, tanto para ir atualizando a posição de inserção quanto para achar o menor valor da lista.

Usamos a variável i para controlar a posição de inserção e a variável j para iterar sobre os valores da lista, procurando o menor valor. A busca pelo menor valor é feita com o auxílio de uma variável com a qual, quando o menor valor for encontrado, a variável index_menor guardará a posição para a troca dos valores.

Quando o valor na posição i já for o menor, então index_menor não se atualiza pelo j. Na linha 9, usamos a atribuição múltipla para fazer a troca dos valores, quando necessário.

```
def executar_selection_sort(lista):
    n = len(lista)
    for i in range(0, n):
        index_menor = i
        for j in range(i+1, n):
            if lista[j] < lista[index_menor]:
                index_menor = j
        lista[i], lista[index_menor] = lista[index_menor], lista[i]
    return lista
lista = [10, 9, 5, 8, 11, -1, 3]
executar_selection_sort(lista)
```

Algoritmos de Ordenação

Bubble sort (Ordenação por "bolha")

O algoritmo bubble sort (algoritmo da bolha) faz a ordenação sempre a partir do ínicio da lista, comparando um valor com seu vizinho. Esse processo é repetido até que todas as pessoas estejam na posição correta. A lógica do algoritmo é a seguinte:

Iteração 1: seleciona o valor na posição 0 e o compara com seu vizinho – se for menor, há troca; se não for, seleciona o próximo e compara, repetindo o processo.

Iteração 2: seleciona o valor na posição 0 e compara ele com seu vizinho, se for menor troca, senão seleciona o próximo e compara, repetindo o processo.

Iteração N - 1: seleciona o valor na posição 0 e o compara com seu vizinho – se for menor, há troca; se não for, seleciona o próximo e compara, repetindo o processo.

Algoritmos de Ordenação

Temos uma variável que guarda o tamanho da lista (n). Precisamos de duas estruturas de controle para iterar, tanto para controlar a bolha, que é a quantidade de vezes que a comparação voltará para o início, quanto para controlar as comparações entre vizinhos.

Usamos a variável i para o controle da bolha e a j para fazer as comparações entre vizinhos. Veja que, na linha 5, fazemos a comparação – caso o valor antecessor seja maior que o de seu sucessor, então é feita a troca de posições na linha 6; caso contrário, o valor de j é incremento, e uma nova comparação é feita, até que tenham sido comparados todos valores.

```
def executar_bubble_sort(lista):
    n = len(lista)
    for i in range(n-1):
        for j in range(n-1):
            if lista[j] > lista[j + 1]:
                lista[j], lista[j + 1] = lista[j + 1], lista[j]
    return lista


lista = [10, 9, 5, 8, 11, -1, 3]
executar_bubble_sort(lista)
```

Algoritmos de Ordenação

Insertion sort (Ordenação por inserção)

Faz a ordenação pela simulação da inserção de novos valores na lista. A lógica do algoritmo é a seguinte:

- Início: parte-se do princípio de que a lista possui um único valor e, consequentemente, está ordenada.
- Iteração 1: parte-se do princípio de que um novo valor precisa ser inserido na lista; nesse caso, ele é comparado com o valor já existente para saber se precisa ser feita uma troca de posição.
- Iteração 2: parte-se do princípio de que um novo valor precisa ser inserido na lista; nesse caso, ele é comparado com os valores já existentes para saber se precisam ser feitas trocas de posição.
- Iteração N: parte-se do princípio de que um novo valor precisa ser inserido na lista; nesse caso, ele é comparado com todos os valores já existentes (desde o início) para saber se precisam ser feitas trocas de posição.

Algoritmos de Ordenação

Temos uma variável que guarda o tamanho da lista (n). Precisamos de duas estruturas de controle para iterar. Na primeira estrutura (linha 3), usamos o for para controlar a variável i, que representa a posição do valor a ser inserido. Uma vez que sabemos exatamente quantas vezes iterar, o for pode ser usado.

Observe que o for começa na posição 1, pois o algoritmo parte do princípio de que a lista possui um valor e um novo precisa ser inserido. Na linha 5, inicializamos a variável j, com a posição anterior ao valor a ser inserido.

```
def executar_insertion_sort(lista):
    n = len(lista)
    for i in range(1, n):
        valor_inserir = lista[i]
        j = i - 1
```

Algoritmos de Ordenação

Na linha 7, criamos a segunda estrutura de repetição com while, pois não sabemos quantas casas vamos ter de percorrer até encontrar a posição correta de inserção.

Veja que o loop acontecerá enquanto houver elementos para comparar (j >= 0) e o valor da posição anterior (lista[j]) for maior que o valor a ser inserido. Enquanto essas condições acontecerem, os valores já existentes vão sendo "passados para frente" (linha 8) e j vai decrementando (linha 9).

Quando a posição for encontrada, o valor é inserido (linha 10).

```
while j >= 0 and lista[j] > valor_inserir:
        lista[j + 1] = lista[j]
        j -= 1
    lista[j + 1] = valor_inserir


return lista



lista = [10, 9, 5, 8, 11, -1, 3]
executar_insertion_sort(lista)
```

Algoritmos de Ordenação

Merge sort (Ordenação por junção)

O algoritmo merge sort recebe esse nome porque faz a ordenação em duas etapa:

(i) divide a lista em sublistas;

e (ii) junta (merge) as sublistas já ordenadas.

O paradigma de dividir e conquistar envolve três etapas em cada nível da recursão:

(i) dividir o problema em vários subproblemas;

(ii) conquistar os subproblemas, resolvendo-os recursivamente – se os tamanhos dos subproblemas forem pequenos o suficiente, apenas resolva os subproblemas de maneira direta;

(iii) combine as soluções dos subproblemas na solução do problema original.

Algoritmos de Ordenação

Etapa de divisão:

- Com base na lista original, encontre o meio e separe-a em duas listas: esquerda_1 e direita_2.

- Com base na sublista esquerda_1, se a quantidade de elementos for maior que 1, encontre o meio e separe-a em duas listas: esquerda_1_1 e direta_1_1.

- Com base na sublista esquerda_1_1, se a quantidade de elementos for maior que 1, encontre o meio e separe-a em duas listas: esquerda_1_2 e direta_1_2.

- Repita o processo até encontrar uma lista com tamanho 1.

- Chame a etapa de merge.

- Repita o processo para todas as sublistas.

Algoritmos de Ordenação

Esse algoritmo possui mais complexidade de código.

Primeiro ponto: vamos precisar de duas funções, uma que divide e outra que junta.

Segundo ponto: a divisão é feita de maneira lógica, ou seja, as sublistas são "fatiamentos" da lista original. O algoritmo de merge vai sempre receber a lista inteira, mas tratará de posições específicas.

Terceiro ponto: na etapa de divisão, serão feitas sucessivas subdivisões aplicando-se a mesma regra, tarefa para cuja realização vamos usar a técnica de recursão, fazendo chamadas recursivas a função de divisão.

Algoritmos de Ordenação

```
def executar_merge_sort(lista):
    if len(lista) <= 1: return lista
    else:
        meio = len(lista) // 2
        esquerda = executar_merge_sort(lista[:meio]
)
        direita = executar_merge_sort(lista[meio:])
        return executar_merge(esquerda, direita)
def executar_merge(esquerda, direita):
    sub_lista_ordenada = []
    topo_esquerda, topo_direita = 0, 0
    while topo_esquerda < len(esquerda) and topo_
direita < len(direita):
```

Criamos nossa função de ordenação por merge sort, que, na verdade, são duas funções, uma para dividir as listas (executar_merge_sort) e outra para fazer o merge (executar_merge).

A função que faz a divisão recebe como parâmetro a lista a ser ordenada. Na linha 2, se o tamanho da lista é menor ou igual 1, isso significa que a sublista só tem 1 valor e está ordenada, razão pela qual seu valor é retornado; caso não seja, então é encontrado o meio da lista e feita a divisão entre sublistas da direita e da esquerda.

Esse processo é feito recursivamente até que se tenha sublistas de tamanho 1.

Algoritmos de Ordenação

```
        if esquerda[topo_esquerda] <= direita[topo_di
reita]:
            sub_lista_ordenada.append(esquerda[topo_
esquerda])
            topo_esquerda += 1
        else:
            sub_lista_ordenada.append(direita[topo_dir
eita])
            topo_direita += 1
    sub_lista_ordenada += esquerda[topo_esquerda:
]
    sub_lista_ordenada += direita[topo_direita:]
    return sub_lista_ordenada
lista = [10, 9, 5, 8, 11, -1, 3]
executar_merge_sort(lista)
```

A função de junção, ao receber duas listas, percorre cada uma delas pelo while na linha 13, e, considerando cada valor, o que for menor é adicionado à sublista ordenada.

Algoritmos de Ordenação

Quicksort (Ordenação rápida)

Dado um valor em uma lista ordenada, à direita desse número existem somente números maiores que ele; e à esquerda, somente os menores.

Esse valor, chamado de pivô, é a estratégia central no algoritmo quicksort.

O algoritmo quicksort também trabalha com a estratégia de dividir para conquistar, pois, a partir do pivô, quebrará uma lista em sublistas (direita e esquerda) – a cada escolha do pivô e a cada quebra da lista, o processo de ordenação vai acontecendo.

Algoritmos de Ordenação

A lógica é a seguinte:

- Primeira iteração: a lista original será quebrada através de um valor chamado de pivô. Após a quebra, os valores que são menores que o pivô devem ficar à sua esquerda e os maiores à sua direita. O pivô é inserido no local adequado, trocando a posição com o valor atual.
- Segunda iteração: agora há duas listas, a da direita e a da esquerda do pivô. Novamente são escolhidos dois novos pivôs e é feito o mesmo processo, de colocar à direita os menores e à esquerda os maiores. Ao final os novos pivôs ocupam suas posições corretas.
- Terceira iteração: olhando para as duas novas sublistas (direita e esquerda), repete-se o processo de escolha dos pivôs e separação.
- Na última iteração, a lista estará ordenada, como resultado dos passos anteriores.

Algoritmos de Ordenação

```
def executar_quicksort(lista, inicio, fim):
    if inicio < fim:
        pivo = executar_particao(lista, inicio, fim)
        executar_quicksort(lista, inicio, pivo-1)
        executar_quicksort(lista, pivo+1, fim)
    return lista
```

Implementamos nosso quicksort em duas funções: executar_quicksort e executar_particao. A função executar_quicksort é responsável por criar as sublistas, cada uma das quais, no entanto, deve ser criada com base em um pivô.

Por isso, caso a posição de início da lista seja menor que o fim (temos mais que 1 elemento), então é chamada a função executar_particao, que de fato faz a comparação e, quando necessário, troca os valores de posição, além de retornar o índice correto para o pivô.

Algoritmos de Ordenação

```
def executar_particao(lista, inicio, fim):
    pivo = lista[fim]
    esquerda = inicio
    for direita in range(inicio, fim):
        if lista[direita] <= pivo:
            lista[direita], lista[esquerda] =
lista[esquerda], lista[direita]
            esquerda += 1
    lista[esquerda], lista[fim] = lista[fim],
lista[esquerda]
    return esquerda

lista = [10, 9, 5, 8, 11, -1, 3]
executar_quicksort(lista, inicio=0, fim=len(lista)-1)
```

Na linha 10, fazemos a definição do pivô como o último valor da lista (e mesmo da sublista). Na linha 11, criamos a variável que controla a separação da lista da esquerda, ou seja, a lista que guardará os valores menores que o pivô. Usamos a estrutura de repetição para ir comparando o pivô com todos os valores da lista à direita.

A cada vez que um valor menor que o pivo é encontrado (linha 13), é feita a troca dos valores pelas posições (linha 14), e a delimitação da lista dos menores (esquerda) é atualizada (linha 15). Na linha 16, o pivô é colocado na sua posição (limite da lista esquerda), fazendo a troca com o valor que ali está. Por fim, a função retorna o índice do pivô.

Qual modelo de algoritmo de ordenação apresenta mais performance na de linguagem de programação Python?

Recapitulando

Recapitulando

- Estrutura de dados em Python
- Algoritmos de busca
- Algoritmos de Ordenação