Linguagem de Programação

Estruturas de dados em Python

Prof.ª Elisa Antolli

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

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

- 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

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.

Estrutura de dados em Python

Objetos do tipo sequência

Ex.: 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

Objetos do tipo sequência

Ex.: Sequência de caracteres.

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

```
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]}")
```

Fonte: elaborado pelo autor.

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.

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

```
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}')
```

Fonte: elaborado pelo autor.

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)}")
```

Fonte: elaborado pelo autor.

Estrutura de dados em Python

List comprehension (Compreensões de lista)

A list comprehension, também pode ser 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

Lista

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

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

Fonte: elaborado pelo autor.

```
C:\Users\Usuario\Desktop                                    >python teste4.py
Antes da listcomp =  ['Python', 'Java', 'JavaScript', 'C', 'C#', 'C++', 'Swift', 'Go', 'Kotlin']

Depois da listcomp =  ['python', 'java', 'javascript', 'c', 'c#', 'c++', 'swift', 'go', 'kotlin']
```

Fonte: elaborado pelo autor.

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

```
# 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}")
```

Fonte: elaborado pelo autor.

```
Exemplo
A nova lista é = <map object at 0x000001BFCF968F10>

Agora sim, a nova lista é = ['python', 'java', 'javascript', 'c', 'c#', 'c++', 'swift', 'go', 'kotlin']
```

Fonte: elaborado pelo autor.

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.

```
1  numeros = list(range(0, 21))
2
3  numeros_pares = list(filter(lambda x: x % 2 == 0, numeros))
4
5  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.

Estrutura de dados em Python

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}")
```

```
Tipo do objeto vogais = <class 'tuple'>
Posição = 0, valor = a
Posição = 1, valor = e
Posição = 2, valor = i
Posição = 3, valor = o
Posição = 4, valor = u
```

Fonte: elaborado pelo autor.

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).

Estrutura de dados em Python

Objetos do tipo Set

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

```
len(s)
x in s
x not in s
```

Fonte: elaborado pelo autor.

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

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.

Estrutura de dados em Python

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

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

```
# 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)])
```

Fonte: elaborado pelo autor.

Estrutura de dados em Python

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.

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.

Estrutura de dados em Python

```
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: parte 1

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 Rechelle Edan Igor Ethan Reed Travis Hoyt'.split()

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

Fonte: elaborado pelo autor.

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
```

Fonte: elaborado pelo autor.

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.

Algoritmos de busca

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

Fonte: elaborado pelo autor.

Algoritmos de busca: parte 2

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

Algoritmos de busca

Busca binária

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 metade 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

```
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
        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
```

Fonte: elaborado pelo autor.

Algoritmos de busca

Quais as vantagens e limitações da busca sequencial?

Busca Sequencial, é a forma mais simples de busca, percorresse 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.

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

```
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)
```

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

lista_ordenada1 =  [-3, 1, 4, 10, 15]
lista_ordenada2 =  None
lista =  [-3, 1, 4, 10, 15]
```

Fonte: elaborado pelo autor.

Algoritmos de Ordenação

```
lista = [7, 4]

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

print(lista)
```

```
[4, 7]
```

Fonte: elaborado pelo autor.

Algoritmos de Ordenação

```
lista = [5, -1]

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

print(lista)
```

```
[-1, 5]
```

Fonte: elaborado pelo autor.

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.

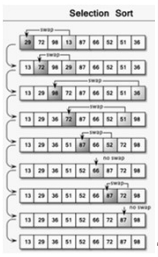
Algoritmos de Ordenação

Selection sort (Ordenação por seleção)

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

Selection sort



Fonte: wikiresource.com.

Algoritmos de Ordenação

Selection sort

```
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, 3]
print(executar_selection_sort(lista))
```

```
[3, 5, 8, 9, 10, 11]
```

Fonte: elaborado pelo autor.

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.

Algoritmos de Ordenação

Bubble sort (Ordenação por "bolha")

A lógica do algoritmo é a seguinte:

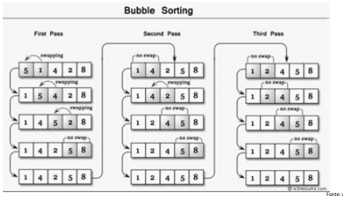
Iteração 1: seleciona o valor na posição 0 e o compara com seu vizinho – se for menor, há troca; se não for, seleciona o próximo e compara, repetindo o processo.

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

Bubble sort (Ordenação por "bolha")



Fonte: wikiresource.com.

Algoritmos de Ordenação

Bubble sort

```
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)
```

```
[-1, 3, 5, 8, 9, 10, 11]
```

Fonte: elaborado pelo autor.

Algoritmos de Ordenação

Merge sort (Ordenação por junção)

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

1. divide a lista em sublistas;
2. e junta (merge) as sublistas já ordenadas.

Algoritmos de Ordenação

Merge sort (Ordenação por junção)

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. combinar as soluções dos subproblemas na solução do problema original.

Algoritmos de Ordenação

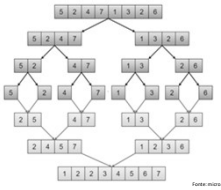
Merge sort (Pseudo-algoritmo)

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

Merge sort (Pseudo-algoritmo)



Fonte: micro.medium.

Recapitulando

Recapitulando

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