UND 2

NF

NÃO PODE FALTAR

ESTRUTURAS DE DADOS EM PYTHON

Vanessa Cadan Scheffer

COMO DEFINIMOS ESTRUTURAS 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.

Fonte: Shutterstock.

Deseja ouvir este material?

INTRODUÇÃO

"Todos os dados em um programa Python são representados por objetos ou pela relação entre objetos." (PSF, p. 1, 2020a). Tudo em Python é um objeto. Mas, afinal, o que são objetos? Embora tenhamos uma seção inteira dedicada a esse tema, por ora basta entendermos que um objeto é uma estrutura que possui certas características e ações.

Já conhecemos alguns tipos de objetos em Python, tais como o int (inteiro), o str (string), o float (ponto flutuante), tipos que nos lembram das variáveis primitivas de outras linguagens, como C ou Java.

Um objeto pode ser mais complexo que um tipo primitivo. Por exemplo, o tipo primitivo int, na linguagem C, ocupa no máximo 4 bytes (32 bits) e pode armazenar valores entre -2.147.483.648 e 2.147.483.647 (https://bit.ly/2NRspI9). Já o objeto do tipo int, na linguagem Python, não possui um limite definido, uma vez que fica limitado apenas à memória RAM (random access memory) disponível no ambiente (https://bit.ly/3g6S0J1).

Outro exemplo interessante é a classe str (string). Em linguagem como a C, uma string é um vetor do tipo primitivo *char*. Por sua vez, em Python, esse objeto, além de tamanho ilimitado, possui vários métodos para manipulação de textos, como o split(), o replace(), dentre outros.

O tipo do objeto determina os valores que ele pode armazenar e as operações que podem ser feitas com tal estrutura. Nesta seção, portanto, nosso objetivo é aprender novos tipos de objetos suportados pela linguagem Python. Vamos, mais especificamente, aprender sobre os tipos de estruturas de dados: listas, tuplas, conjuntos, dicionário e matriz.

ESTRUTURAS DE DADOS EM PYTHON

Em Python, existem objetos que são usados para armazenar mais de um valor, também chamados de estruturas de dados. Cada objeto é capaz de armazenar um tipo de estrutura de dados, particularidade essa que habilita funções diferentes para cada tipo. Portanto, a escolha da estrutura depende do problema a ser resolvido. Para nosso estudo, vamos dividir as estruturas de dados em objetos do tipo *sequência*, do tipo *set*, do tipo *mapping* e do tipo *array NumPy*. Cada grupo pode possuir mais de um objeto. Vamos estudar esses objetos na seguinte ordem:

1. Objetos do tipo *sequência*: texto, listas e tuplas.
2. Objetos do tipo *set* (conjunto).
3. Objetos do tipo *mapping* (dicionário).
4. Objetos do tipo *array NumPy*.

OBJETOS DO TIPO *SEQUÊNCIA*

Os objetos do tipo *sequência* são estruturas de dados capazes de armazenar mais de um valor. 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. As estruturas de dados desse grupo possuem algumas operações em comum, apresentadas no Quadro 2.1.

Quadro 2.1 | Operações em comum dos objetos do tipo sequência

Operação	Resultado	
x in s	True caso um item de s seja igual a x, senão False	True caso um item de s seja igu
s + t	Concatenação de s e t	Concatena (junta) duas sequên
n * s	Adiciona s a si mesmo n vezes	Multiplica a sequência n vezes
s[i]	Acessa o valor guardado na posição i da sequência	O primeiro valor ocupa a posiç
s[i:j]	Acessa os valores da posição i até j	O valor j não está incluído
s[i:j:k]	Acessa os valores da posição i até j, com passo k	O valor j não está incluído
len(s)	Comprimento de s	Função built-in usada para sab
min(s)	Menor valor de s	Função built-in usada para sab
max(s)	Maior valor de s	Função built-in usada para sab
s.count(x)	Número total de ocorrência de x em s	Conta quantas vezes x foi enco

Fonte: adaptado de PSF (2020c).

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 apresentadas no Quadro 2.1, mas são objetos imutáveis, razão pela qual não é possível atribuir um novo valor a uma posição específica. Vamos testar algumas operações. Observe o código a seguir.

In [1]:
```
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]}")
Tamanho do texto = 60
Python in texto = True
Quantidade de y no texto = 1
As 5 primeiras letras são: Aprend
```

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.

Além das operações disponíveis no Quadro 2.1, a classe *str* possui vários outros métodos. A lista completa das funções para esse objeto pode ser encontrada na documentação oficial (PSF, 2020c). Podemos usar a função *lower()* para tornar um objeto *str* com letras minúsculas, ou, então, a função *upper()*, que transforma para maiúsculo. A função *replace()* pode ser usada para substiuir um caractere por outro. Observe o código a seguir.

In [2]:

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

print(texto.upper())
print(texto.replace("i", 'XX'))
APRENDENDO PYTHON NA DISCIPLINA DE LINGUAGEM DE PROGRAMAÇÃO.
Aprendendo Python na dXXscXXplXXna de lXXnguagem de programação.
```

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. Observe o código a seguir.

In [3]:

```
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)}")
texto = Aprendendo Python na disciplina de linguagem de
programação.
Tamanho do texto = 60

palavras = ['Aprendendo', 'Python', 'na', 'disciplina', 'de',
'linguagem', 'de', 'programação.']
Tamanho de palavras = 8
```

Na linha 5 da entrada 3 (In [3]), usamos a função split() para cortar o texto e guardamos o resultado em uma variável chamada "palavras". Veja no resultado que o texto tem tamanho 60, ou seja, possui uma sequência de 60 caracteres. Já o tamanho da variável "palavras" é 8, pois, ao cortar o texto, criamos uma lista com as palavras (em breve estudaremos as listas). A função *split()*, usada dessa forma, corta um texto em cada espaço em branco.

EXEMPLIFICANDO

Vamos usar tudo que aprendemos até o momento para fazer um exercício de análise de texto. Dado um texto sobre strings em Python, queremos saber quantas vezes o autor menciona a palavra *string* ou *strings*. Esse tipo de raciocínio é a base para uma área de pesquisa e aplicação dedicada a criar algoritmos e técnicas para fazer análise de sentimentos em textos. Veja no código a seguir, como é simples fazer essa contagem, usando as operações que as listas possuem.

In [4]:

```
texto = """Operadores de String
Python oferece operadores para processar texto (ou seja, valores
de string).
```

```
Assim como os números, as strings podem ser comparadas usando
operadores de comparação:
==, !=, <, > e assim por diante.
O operador ==, por exemplo, retorna True se as strings nos dois
lados do operador tiverem o mesmo valor (Perkovic, p. 23, 2016).
"""
print(f"Tamanho do texto = {len(texto)}")
texto = texto.lower()
texto = texto.replace(",", "").replace(".", "").replace("(",
"").replace(")", "").replace("\n", " ")
lista_palavras = texto.split()
print(f"Tamanho da lista de palavras = {len(lista_palavras)}")

total = lista_palavras.count("string") +
lista_palavras.count("strings")

print(f"Quantidade de vezes que string ou strings aparecem =
{total}")
Tamanho do texto = 348
Tamanho da lista de palavras = 58
Quantidade de vezes que string ou strings aparecem = 4
```

Na entrada 4 (In [4]), começamos criando uma variável chamada "texto" que recebe uma citação do livro: PERKOVIC, Ljubomir. **Introdução à computação usando Python**: um foco no desenvolvimento de aplicações. Rio de Janeiro: LTC, 2016 (disponível na biblioteca virtual).

Na linha 8, aplicamos a função *lower()* a essa string para que todo o texto fique com letras minúsculas e guardamos o resultado dessa transformação, dentro da própria variável, sobrescrevendo, assim, o texto original.

Na linha 9, fazemos uma série encadeada de transformações usando a função *replace()*, que age substituindo o primeiro parâmetro pelo segundo. No primeiro replace(",", ""), substituímos todas as vírgulas por nada. Então, na verdade, estamos apagando as vírgulas do texto sem colocar algo no lugar. No último replace("\n", " "), substituímos as quebras de linhas por um espaço em branco.

Na linha 10 criamos uma lista ao aplicar a função *split()* ao texto. Nesse caso, sempre que houver um espaço em branco, a string será cortada, criando um novo elemento na lista. Veja no resultado que criamos uma lista com 58 elementos.

Na linha 13 está a "mágica": usamos a função *count()* para contar tanto a palavra "string" no singular quanto o seu plural "strings". Uma vez que a função retorna um número inteiro, somamos os resultados e os exibimos na linha 15.

Essa foi uma amostra de como as estruturas de dados, funcionando como objetos na linguagem Python, permite implementar diversas soluções com poucas linhas de código, já que cada objeto já possui uma série de operações implementadas, prontas para serem usadas. Utilize o emulador a seguir para testar o código e explorar novas possibilidades.

LISTAS

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()

Os objetos do tipo *sequência* são indexados, o que significa que cada elemento ocupa uma posição que começa em 0. Portanto, um objeto com 5 elementos terá índices que variam entre 0 e 4. O primeiro elemento ocupa a posição 0; o segundo, a posição 1; o terceiro, a posição 2; o quarto, a posição 3; e o quinto, a posição 4. Para facilitar a compreensão, pense na sequência como um prédio com o andar térreo. Embora, ao olhar de fora um prédio de 5 andares, contemos 5 divisões, ao adentrar no elevador, teremos as opções: T, 1, 2, 3, 4.

Observe, no código a seguir, a criação de uma lista chamada de "vogais". Por meio da estrutura de repetição, imprimos 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.

In [5]:
```
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}')
Posição = 0, valor = a
Posição = 1, valor = e
Posição = 2, valor = i
Posição = 3, valor = o
Posição = 4, valor = u
```
Que tal testar o código da entrada 5 na ferramenta Python Tutor? Aproveite a ferramenta e explore novas possibilidades.
In [6]:
```
# Mesmo resultado.

vogais = []
print(f"Tipo do objeto vogais = {type(vogais)}")

vogais.append('a')
vogais.append('e')
vogais.append('i')
vogais.append('o')
vogais.append('u')

for p, x in enumerate(vogais):
```

```
    print(f"Posição = {p}, valor = {x}")
Tipo do objeto vogais = <class 'list'>
Posição = 0, valor = a
Posição = 1, valor = e
Posição = 2, valor = i
Posição = 3, valor = o
Posição = 4, valor = u
```

Veja, na entrada 6 (In [6]), que repetimos o exemplo com algumas alterações, a primeira das quais foi criar uma lista vazia na linha 3. Observe que, mesmo estando vazia, ao imprimirmos o tipo do objeto, o resultado é "class list", pois o colchete é a sintaxe para a construção de listas. Outra novidade foi usar a função *append()*, que adiciona um novo valor ao final da lista. Na sintaxe usamos `vogais.append(valor)`, notação que significa que a função *append()* é do objeto *lista*. Também substituímos o contador manual *("p")* pela função *enumerate()*, que é usada para percorrer um objeto iterável retornando a posição e o valor. Por isso, na estrutura de repetição precisamos usar as variáves *p* e *x*. A primeira guarda a posição e a segunda guarda o valor. Usamos o nome *x* propositalmente para que você perceba que o nome da variável é de livre escolha.

Agora que já sabemos criar uma lista, vamos testar algumas das operações mencionadas no Quadro 2.1. Observe o código a seguir:

In [7]:

```
frutas = ["maça", "banana", "uva", "mamão", "maça"]
notas = [8.7, 5.2, 10, 3.5]

print("maça" in frutas) # True
print("abacate" in frutas) # False
print("abacate" not in frutas) # True
print(min(frutas)) # banana
print(max(notas)) # 10
print(frutas.count("maça")) # 2
print(frutas + notas)
print(2 * frutas)
True
False
True
banana
10
2
['maça', 'banana', 'uva', 'mamão', 'maça', 8.7, 5.2, 10, 3.5]
['maça', 'banana', 'uva', 'mamão', 'maça', 'maça', 'banana',
'uva', 'mamão', 'maça']
```

Na entrada 7 (In [7]), definimos duas listas, e, da linha 4 à linha 11, exploramos algumas operações com esse tipo de estrutura de dados. Nas linhas 4 e 5 testamos, respectivamente, se os valores "maça" e "abacate" estavam na lista, e os resultados foram True e False. Na linha 6, testamos se a palavra "abacate" não está na lista, e obtivemos True, uma vez que isso é verdade. Nas linhas 7 e 8 usamos as funções *mínimo* e *máximo* para saber o menor e o maior valor de cada lista. O mínimo de uma lista de palavras é feito sobre a ordem alfabética. Na linha 9, contamos quantas

vezes a palavra "maça" aparece na lista. Na linha 10, concatenamos as duas listas e, na linha 11, multiplicamos por 2 a lista de frutas – veja no resultado que uma "cópia" da própria da lista foi criada e adicionada ao final.

Até o momento, quando olhamos para a sintaxe de construção da lista, encontramos semelhanças com a construção de arrays. No entanto, a lista é um objeto muito versátil, pois sua criação suporta a mistura de vários tipos de dados, conforme mostra o código a seguir. Além dessa característica, o fatiamento (slice) de estruturas sequenciais é uma operação muito valiosa. Observe o mencionado código:

In [8]:
```
lista = ['Python', 30.61, "Java", 51 , ['a', 'b', 20], "maça"]

print(f"Tamanho da lista = {len(lista)}")

for i, item in enumerate(lista):
    print(f"Posição = {i},\t valor = {item} ----------------->
tipo individual = {type(item)}")

print("\nExemplos de slices:\n")

print("lista[1] = ", lista[1])
print("lista[0:2] = ", lista[0:2])
print("lista[:2] = ", lista[:2])
print("lista[3:5] = ", lista[3:5])
print("lista[3:6] = ", lista[3:6])
print("lista[3:] = ", lista[3:])
print("lista[-2] = ", lista[-2])
print("lista[-1] = ", lista[-1])
print("lista[4][1] = ", lista[4][1])
Tamanho da lista = 6
Posição = 0,   valor = Python -----------------> tipo individual
= <class 'str'>
Posição = 1,   valor = 30.61 -----------------> tipo individual
= <class 'float'>
Posição = 2,   valor = Java -----------------> tipo individual =
<class 'str'>
Posição = 3,   valor = 51 -----------------> tipo individual =
<class 'int'>
Posição = 4,   valor = ['a', 'b', 20] -----------------> tipo
individual = <class 'list'>
Posição = 5,   valor = maça -----------------> tipo individual =
<class 'str'>

Exemplos de slices:

lista[1] =  30.61
lista[0:2] =  ['Python', 30.61]
lista[:2] =  ['Python', 30.61]
lista[3:5] =  [51, ['a', 'b', 20]]
```

```
lista[3:6] =  [51, ['a', 'b', 20], 'maça']
lista[3:] =  [51, ['a', 'b', 20], 'maça']
lista[-2] =  ['a', 'b', 20]
lista[-1] =  maça
lista[4][1] =  b
```

Na entrada 8 (In [8]), criamos uma lista que contém: texto, número (float e int) e lista! Isto mesmo: uma lista dentro de outra lista. Em Python, conseguimos colocar uma estrutura de dados dentro de outra sem ser necessário ser do mesmo tipo. Por exemplo, podemos colocar um dicionário dentro de uma lista. Para auxiliar a explicação do código, criamos a estrutura de repetição com enumerate, que indica o que tem em cada posição da lista. Veja que cada valor da lista pode ser um objeto diferente, sem necessidade de serem todos do mesmo tipo, como acontece em um array em C, por exemplo. Da linha 10 à linha 18 criamos alguns exemplos de slices. Vejamos a explicação de cada um:

- lista[1]: acessa o valor da posição 1 da lista.
- lista[0:2]: acessa os valores que estão entre as posições 0 e 2. Lembre-se de que o limite superior não é incluído. Ou seja, nesse caso serão acessados os valores das posição 0 e 1.
- lista[:2]: quando um dos limites não é informado, o interpretador considera o limite máximo. Como não foi informado o limite inferior, então o slice será dos índices 0 a 2 (2 não incluído).
- lista[3:5]: acessa os valores que estão entre as posições 3 (incluído) e 5 (não incluído). O limite inferior sempre é incluído.
- lista[3:6]: os indíces da lista variam entre 0 e 5. Quando queremos pegar todos os elementos, incluindo o último, devemos fazer o slice com o limite superior do tamanho da lista. Nesse caso, é 6, pois o limite superior 6 não será incluído.
- lista[3:]: outra forma de pegar todos os elementos até o último é não informar o limite superior.
- lista[-2]: o slice usando índice negativo é interpretado de trás para frente, ou seja, índice -2 quer dizer o penúltimo elemento da lista.
- lista[-1]: o índice -1 representa o último elemento da lista.
- lista[4][1]: no índice 5 da lista há uma outra lista, razão pela qual usamos mais um colchete para conseguir acessar um elemento específico dessa lista interna.

Esses exemplos nos dão uma noção do poder que as listas têm. Utilize o emulador a seguir para criar e testar novos fatiamentos em listas. O que acontece se tentarmos acessar um índice que não existe? Por exemplo: tentar imprimir `print(lista[6])`. Aproveite a oportunidade e explore-a!
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. No decorrer do nosso estudo vamos explorar várias delas.

LIST COMPREHENSION (COMPREENSÕES DE LISTA)

A list comprehension, também chamada de listcomp, é uma forma pythônica de criar uma lista com base em um objeto iterável. 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. Para entender essa técnica, vamos

supor que tenhamos uma lista de palavras e desejamos padronizá-las para minúsculo. Observe o código a seguir.

In [9]:
```
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)
Antes da listcomp =  ['Python', 'Java', 'JavaScript', 'C', 'C#',
'C++', 'Swift', 'Go', 'Kotlin']

Depois da listcomp =  ['python', 'java', 'javascript', 'c',
'c#', 'c++', 'swift', 'go', 'kotlin']
```
Na entrada 9 (In [9]), 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". Na saída fizemos a impressão antes e depois da listcomp. Veja a diferença.

A listcomp é uma forma pythônica de escrever um *for*. O código da entrada 9 poderia ser escrito conforme mostrado a seguir, e o resultado é o mesmo.

In [10]:
```
linguagens = '''Python Java JavaScript C C# C++ Swift Go
Kotlin'''.split()
print("Antes da listcomp = ", linguagens)

for i, item in enumerate(linguagens):
    linguagens[i] = item.lower()

print("\nDepois da listcomp = ", linguagens)
Antes da listcomp =  ['Python', 'Java', 'JavaScript', 'C', 'C#',
'C++', 'Swift', 'Go', 'Kotlin']

Depois da listcomp =  ['python', 'java', 'javascript', 'c',
'c#', 'c++', 'swift', 'go', 'kotlin']
```
Agora vamos usar a listcomp para construir uma lista que contém somente as linguagens que possuem "Java" no texto. Veja o código a seguir.

In [11]:
```
linguagens = '''Python Java JavaScript C C# C++ Swift Go
Kotlin'''.split()

linguagens_java = [item for item in linguagens if "Java" in
item]

print(linguagens_java)
['Java', 'JavaScript']
```
Na entrada 11 (In [11]), a listcomp é construída com uma estrutura de decisão (if) a fim de criar um filtro. Veja que a variável *item* será considerada somente se ela tiver "Java" no texto. Portanto, dois itens da lista original atendem a esse requisito e são adicionados à nova lista. Vale ressaltar que a operação `x in s` significa "x está em s?"; portanto, Java está em JavaScript. Se precisarmos criar um filtro que retorne somente palavras idênticas, precisamos fazer: `linguagens_java = [item for item in linguagens if "Java" == item]`. A listcomp da entrada 6 poderia ser escrita conforme o código a seguir. Veja que precisaríamos digitar muito mais instruções.

In [12]:
```
linguagens = '''Python Java JavaScript C C# C++ Swift Go
Kotlin'''.split()
linguagens_java = []

for item in linguagens:
    if "Java" in item:
        linguagens_java.append(item)

print(linguagens_java)
['Java', 'JavaScript']
```

FUNÇÕES *MAP()* E *FILTER()*

Não poderíamos falar de listas sem mencionar duas 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. Observe os dois exemplos a seguir.

In [13]:
```
# Exemplo 1
print("Exemplo 1")
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}")


# Exemplo 2
print("\n\nExemplo 2")
numeros = [0, 1, 2, 3, 4, 5]

quadrados = list(map(lambda x: x*x, numeros))
print(f"Lista com o número elevado a ele mesmo = {quadrados}\n")
Exemplo 1
A nova lista é = <map object at 0x0000022E6F7CD0B8>

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


Exemplo 2
Lista com o número elevado a ele mesmo = [0, 1, 4, 9, 16, 25]
```

No exemplo 1 da entrada 13 (In [13]), 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: se você olhar o resultado, verá: `A nova lista é = map object at 0x00000237EB0EF320 .`
No entanto, onde está 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.

No exemplo 2 da entrada 11 (In [11]), criamos uma lista numérica na linha 14; e, na linha 16, usamos a função *lambda* para elevar cada número da lista a ele mesmo (quadrado de um número). Na própria linha 16, já fazemos a conversão do resultado da função *map()* para o tipo *list*.

A função *filter()* tem as mesmas características da função *map()*, mas, em vez de usarmos uma função para transformar os valores da lista, nós a usamos para filtrar. Veja o código a seguir.

In [14]:
```
numeros  = list(range(0, 21))

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

print(numeros_pares)
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20]
```
Na entrada 14 (In [14]), 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.

TUPLAS

As tuplas também são estruturas de dados do grupo de objetos do tipo sequência. 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()

Observe o codigo a seguir, no qual criamos uma tupla chamada de "vogais" e, por meio da estrutura de repetição, imprimos cada elemento da tupla. Usamos, ainda, uma variável auxiliar *p* para indicar a posição que o elemento ocupa na tupla.

In [15]:
```
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
```
Com o exemplo apresentado na entrada 15 (In [15]), você pode estar pensando: "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. Voltamos à discussão inicial da nossa seção: objetos podem ter operações em comum entre si, mas cada objeto possui operações próprias. Por exemplo, é possível usar as operações do Quadro 2.1 nas tuplas. No entanto, veja o que acontece se tentarmos usar a função *append()* em uma tupla.

In [16]:
```
vogais = ()

vogais.append('a')
------------------------------------------------------------------
-----------
```

```
AttributeError                            Traceback (most recent
call last)
<ipython-input-16-df6bfea0326b> in <module>
      1 vogais = ()
      2
----> 3 vogais.append('a')

AttributeError: 'tuple' object has no attribute 'append'
```

In [17]:

```
vogais = ('a', 'e', 'i', 'o', 'u')

for item in enumerate(vogais):
    print(item)

# print(tuple(enumerate(vogais)))
# print(list(enumerate(vogais)))
(0, 'a')
(1, 'e')
(2, 'i')
(3, 'o')
(4, 'u')
```

A entrada 16 (In [16]) resulta no erro "AttributeError: 'tuple' object has no attribute 'append'", uma vez que a tupla não possui a operação de *append*.

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 elemtno é o valor em si. Observe o código a seguir e veja que, para cada item de um enumerate(), é impressa uma tupla. Deixamos comentados duas outras formas de ver o resultado da função *enumerate()*. No primeiro comentário usamos o construtor *tuple()* para transformar o resultado em uma tupla, caso no qual temos uma tupla de tuplas. No segundo comentário, usamos o contrutor *list()*, caso no qual temos uma lista de tuplas. Não deixe de testar os códigos e explorar novas possibilidades.

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, diferenteça, 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

Além dessas operações, podemos adicionar um novo elemento a um conjunto com a função add(valor). Também podemos remover com remove(valor). Veja uma lista completa de funções no endereço https://bit.ly/2NF7eIT.

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)

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). Veja os exemplos de construção a seguir.

In [18]:
```
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'))
<class 'set'> {'aeiou'}
<class 'set'> {'u', 'o', 'i', 'e', 'a'}
<class 'set'> {'u', 'o', 'i', 'e', 'a'}
<class 'set'> {'u', 'o', 'i', 'e', 'a'}
{'n', 'a', 'b'}
```

Na entrada 18 (In [18]), 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.

EXEMPLIFICANDO

O poder do objeto *set* está em suas operações matemáticas de conjuntos. Vejamos um exemplo: uma loja de informática recebeu componentes usados de um computador para avaliar se estão com defeito. As peças que não estiverem com defeito podem ser colocadas à venda. Como, então, podemos criar uma solução em Python para resolver

esse problema? A resposta é simples: usando objetos do tipo set. Observe o código a seguir.

In [19]:

```
def create_report():
    componentes_verificados = set(['caixas de som', 'cooler',
'dissipador de calor', 'cpu', 'hd', 'estabilizador', 'gabinete',
'hub', 'impressora', 'joystick', 'memória ram', 'microfone',
'modem', 'monitor', 'mouse', 'no-break', 'placa de captura',
'placa de som', 'placa de vídeo', 'placa mãe', 'scanner',
'teclado', 'webcam'])
    componentes_com_defeito = set(['hd', 'monitor', 'placa de
som', 'scanner'])

    qtde_componentes_verificados = len(componentes_verificados)
    qtde_componentes_com_defeito = len(componentes_com_defeito)

    componentes_ok =
componentes_verificados.difference(componentes_com_defeito)

    print(f"Foram verificados {qtde_componentes_verificados}
componentes.\n")
    print(f"{qtde_componentes_com_defeito} componentes
apresentaram defeito.\n")

    print("Os componentes que podem ser vendidos são:")
    for item in componentes_ok:
        print(item)


create_report()
```

```
Foram verificados 23 componentes.

4 componentes apresentaram defeito.

Os componentes que podem ser vendidos são:
placa mãe
no-break
cpu
dissipador de calor
estabilizador
mouse
placa de vídeo
hub
teclado
microfone
modem
caixas de som
memória ram
```

```
gabinete
webcam
cooler
placa de captura
impressora
joystick
```
Na entrada 19 (In [19]), criamos uma função que gera o relatório das peças aptas a serem vendidas. Nessa função, são criados dois objetos *set*: "componentes_verificados" e "componentes_com_defeito". Nas linhas 5 e 6, usamos a função *len()* para saber quantos itens há em cada conjunto. Na linha 8, fazemos a "mágica"! Usamos a função *difference()* para obter os itens que estão em *componentes_verificados*, mas não em *componentes_com_defeito*. Essa operação também poderia ser feita com o sinal de subtração: `componentes_ok = componentes_verificados - componentes_com_defeito`. Com uma única operação conseguimos extrair uma importante informação!

Aproveite o emulador a seguir para testar o exemplo e explorar novas possibilidades. Teste as funções *union()* e *intersection()* em novos conjuntos e veja o resultado.

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()

Observe os exemplos a seguir com maneiras de construir o dicionário.

In [20]:
```
# 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)])
```

```
# 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.
True
```

Na entrada 20 (In [20]), 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()*. 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.

Para acessar um valor em um dicionário, basta digitar: nome_dicionario[chave]; para atribuir um novo valor *use:* nome_dicionario[chave] = novo_valor.

Que tal utilizar a ferramenta Python Tutor para explorar essa implementação e analisar o passo a passo do funcionamento do código. Aproveite a oportunidade!

Uma única chave em um dicionário pode estar associada a vários valores por meio de uma lista, tupla ou até mesmo outro dicionário. Nesse caso, também conseguimos acessar os elementos internos. No código a seguir, a função *keys()* retorna uma lista com todas as chaves de um dicionário. A função *values()* retorna uma lista com todos os valores. A função *items()* retorna uma lista de tuplas, cada uma das quais é um par chave-valor.

In [21]:
```
cadastro = {
            'nome' : ['João', 'Ana', 'Pedro', 'Marcela'],
            'cidade' : ['São Paulo', 'São Paulo', 'Rio de
Janeiro', 'Curitiba'],
            'idade' : [25, 33, 37, 18]
            }

print("len(cadastro) = ", len(cadastro))

print("\n cadastro.keys() = \n", cadastro.keys())
print("\n cadastro.values() = \n", cadastro.values())
print("\n cadastro.items() = \n", cadastro.items())

print("\n cadastro['nome'] = ", cadastro['nome'])
print("\n cadastro['nome'][2] = ", cadastro['nome'][2])
```

```
print("\n cadastro['idade'][2:] = ", cadastro['idade'][2:])
len(cadastro) =  3

 cadastro.keys() =
 dict_keys(['nome', 'cidade', 'idade'])

 cadastro.values() =
 dict_values([['João', 'Ana', 'Pedro', 'Marcela'], ['São Paulo',
'São Paulo', 'Rio de Janeiro', 'Curitiba'], [25, 33, 37, 18]])

 cadastro.items() =
 dict_items([('nome', ['João', 'Ana', 'Pedro', 'Marcela']),
('cidade', ['São Paulo', 'São Paulo', 'Rio de Janeiro',
'Curitiba']), ('idade', [25, 33, 37, 18])])

 cadastro['nome'] =  ['João', 'Ana', 'Pedro', 'Marcela']

 cadastro['nome'][2] =  Pedro

 cadastro['idade'][2:] =  [37, 18]
```

Vamos avaliar os resultados obtidos com os códigos na entrada 21 (In [21]). Primeiro, veja que criamos um dicionário em que cada chave está associada a uma lista de valores. A função *len()*, na linha 7, diz que o dicionário possui tamanho 3, o que está correto, pois len() conta quantas chaves existem no dicionário. Veja os demais comandos:

- cadastro.keys(): retorna uma lista com todas as chaves no dicionário.
- cadastro.values(): retorna uma lista com os valores. Como os valores também são listas, temos uma lista de listas.
- cadastro.items(): retorna uma lista de tuplas, cada uma das quais é composta pela chave e pelo valor.
- cadastro['nome']: acessa o valor atribuído à chave 'nome'; nesse caso, uma lista de nomes.
- cadastro['nome'][2]: acessa o valor na posição 2 da lista atribuída à chave 'nome'.
- cadastro['idade'][2:]: acessa os valores da posição 2 até o final da lista atribuída à chave 'nome'.

Como vimos, a função *len()* retorna quantas chaves um dicionário possui. No entanto, e se quiséssemos saber o total de elementos somando quantos há em cada chave? Embora não exista função que resolva esse problema diretamente, como conseguimos acessar os valores de cada chave, basta contarmos quantos eles são. Veja o código a seguir.

In [22]:

```
print(len(cadastro['nome']))
print(len(cadastro['cidade']))
print(len(cadastro['idade']))

qtde_itens = sum([len(cadastro[chave]) for chave in cadastro])
```

```
print(f"\n\nQuantidade de elementos no dicionário =
{qtde_itens}")
4
4
4


Quantidade de elementros no dicionário = 12
```

Nas três primeiras linhas da entrada 22, imprimimos a quantidade de elementos atribuídos a cada chave. Embora possamos simplesmente somar esses valores, o que faríamos se tivéssemos centenas de chaves? Para fazer essa contagem, independentemente de quantas chaves e valores existam, podemos criar uma list comprehension. Fizemos isso na linha 5. Veja que, para cada chave, usamos a função *len()*, criando assim uma lista de valores inteiros. A essa lista aplicamos a função *built-in sum()* para somar e obter a quantidade total de itens.

Como podemos ver, as estruturas de dados em Python são muito poderosas e versáteis. A combinação de diferentes estruturas permite criar soluções complexas com poucas linhas de código. Utilize o emulador para testar o código e explorar novas possibilidades.

OBJETOS DO TIPO *ARRAY NUMPY*

Quando o assunto é estrutura de dados em Python, não podemos deixar de falar dos objetos *array numpy*. Primeiramente, todas os objetos e funções que usamos até o momento fazem parte do *core* do interpretador Python, o que quer dizer que tudo está já instalado e pronto para usar. Além desses inúmeros recursos já disponíveis, podemos fazer um certo tipo de instalação e usar objetos e funções que outras pessoas/organizações desenvolveram e disponibilizaram de forma gratuita. Esse é 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.

O endereço com a documentação completa da biblioteca NumPy está disponível em https://numpy.org/. 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.

Para utilizar a biblioteca NumPy é preciso fazer a instalação com o comando `pip install numpy`. No entanto, se você estiver usando o projeto Anaconda, ou o Google Colab, esse recurso já estará instalado. Além da instalação, toda vez que for usar recusos da biblioteca, é necessário importar a biblioteca para o projeto, como o comando `import numpy`. Observe o códgio a seguir.

In [23]:
```
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)
<class 'numpy.ndarray'>

 matriz_1_1 =  [1 2 3]

 matriz_2_2 =
 [[1 2]
 [3 4]]

 matriz_3_2 =
 [[1 2]
 [3 4]
 [5 6]]

 matriz_2_3 =
 [[1 2 3]
 [4 5 6]]
```

Na entrada 23, 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.

NumPy é uma bibliteca muito rica. Veja algumas construções de matrizes usadas em álgebra linear já prontas, com um único comando.

In [24]:
```
m1 = numpy.zeros((3, 3)) # Cria matriz 3 x 3 somente com zero
m2 = numpy.ones((2,2)) # Cria matriz 2 x 2 somente com um
m3 = numpy.eye(4) # Cria matriz 4 x 4 identidade
```

```
m4 = numpy.random.randint(low=0, high=100, size=10).reshape(2,
5) # Cria matriz 2 X 5 com números inteiros

print('\n numpy.zeros((3, 3)) = \n', numpy.zeros((3, 3)))
print('\n numpy.ones((2,2)) = \n', numpy.ones((2,2)))
print('\n numpy.eye(4) = \n', numpy.eye(4))
print('\n m4 = \n', m4)

print(f"Soma dos valores em m4 = {m4.sum()}")
print(f"Valor mínimo em m4 = {m4.min()}")
print(f"Valor máximo em m4 = {m4.max()}")
print(f"Média dos valores em m4 = {m4.mean()}")
 numpy.zeros((3, 3)) =
 [[0. 0. 0.]
 [0. 0. 0.]
 [0. 0. 0.]]

 numpy.ones((2,2)) =
 [[1. 1.]
 [1. 1.]]

 numpy.eye(4) =
 [[1. 0. 0. 0.]
 [0. 1. 0. 0.]
 [0. 0. 1. 0.]
 [0. 0. 0. 1.]]

 m4 =
 [[20 46 25 93 94]
 [ 5 12 19 48 69]]
Soma dos valores em m4 = 431
Valor mínimo em m4 = 5
Valor máximo em m4 = 94
Média dos valores em m4 = 43.1
```

Na entrada 24, criamos matrizes somente com 0, com 1 e matriz identidade (1 na diagonal principal) usando comandos específicos. Por sua vez, a matriz 'm4' foi criada usando a função que gera números inteiros aleatórios. Escolhemos o menor valor como 0 e o maior como 100, e também pedimos para serem gerados 10 números. Em seguida, usamos a função *reshape()* para transformá-los em uma matriz com 2 linhas e 5 colunas. Das linhas 11 a 14, usamos funções que extraem informações estatísticas básicas de um conjunto numérico.

Com essa introdução ao uso da biblioteca NumPy, encerramos esta seção, na qual aprendemos uma variedade de estruturas de dados em Python. Para esse desenvolvimento é importante utilizar o emulador para testar os códigos e implementar novas possibilidades. Será que conseguimos colocar um array NumPy dentro de um dicionário? Que tal testar?

REFERÊNCIAS E LINKS ÚTEIS

BANIN, S. L. **Python 3** - conceitos e aplicações: uma abordagem didática. São Paulo: Érica, 2018.

NUMPT.ORG. Getting Started. Página inicial. 2020. Disponível em: https://numpy.org/. Acesso em: 25 abr. 2020.

PERKOVIC, L. **Introdução à computação usando Python**: um foco no desenvolvimento de aplicações. Rio de Janeiro: LTC, 2016.

PSF - Python Software Fundation. Data model. 2020a. Disponível em: https://bit.ly/3iAcCL4. Acesso em: 25 abr. 2020.

PSF - Python Software Fundation. Data Structures. 2020b. Disponível em: https://bit.ly/3aqdoY3. Acesso em: 25 abr. 2020.

PSF - Python Software Fundation. Built-in Types. 2020c. Disponível em: https://bit.ly/3ixH6NE. Acesso em: 25 abr. 2020.

Bons estudos!