Tópico 08

Introdução à programação de computadores

Introdução à estrutura de dados heterogênea – list

1. Introdução

Aluna(o), é importante que entenda a necessidade das estruturas de dados heterogêneas para seu código Python utilizando listas!

Frequentemente, em outras linguagens de programação, precisamos criar pilhas diferentes pelo simples fato de termos dados de tipos diferentes (STRING, INT, FLOAT E BOOL). Em Python, podemos colocar todos estes dados em uma mesma pilha, aqui conhecida como lista.

Em linguagens de baixo nível (como o C), os vetores são difíceis de trabalhar, sendo necessário pensar muito bem sobre a melhor forma de interagir. Em Python e outras linguagens de alto nível (como MATLAB, R etc.), estas operações são facilitadas e intuitivas.

Em Python, podemos contar com diversas funções e módulos, a fim de fazer operações em listas com diferentes tipos de valores, ações que, em outras linguagens, faríamos com muitas linhas de programação e correndo risco de ter um erro futuro por não termos analisado casos particulares.

Vamos, então, aprender e analisar as diversas funcionalidades das listas Python para que você possa tirar vantagem completa desta linguagem.

2. Estruturas heterogêneas de dados: listas

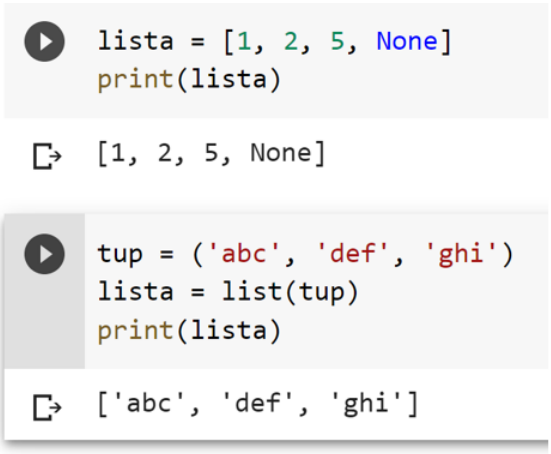
Listas são os tipos de dados que melhor representam a versatilidade da linguagem Python.

"LISTAS têm comprimento variável e seu conteúdo pode ser modificado no local. Você pode defini-las usando colchetes [ ] ou usando a função LIST."
(MCKINNEY, 2019, p. 54).

Trazemos aqui como sugestão o vídeo a seguir. Ele contém uma forma resumida de aula do assunto listas, o qual vai te ajudar bastante na compreensão do assunto.

Aula resumida de listas.
LISTAS em PYTHON

Podemos observar as formas de inicialização de uma lista a partir da imagem a seguir:



Declarando LISTAS.

Podemos também transformar STRING em LISTA:

```
string = "frase"
lista = list(string)
print(lista)

# [OUT]:
# ['f', 'r', 'a', 's', 'e']
```

O caminho de volta é feito com JOIN:

```
lista = ["f", "r", "a", "s", "e"]
string = "".join(lista)
print(string)

# [OUT]:
# frase
```

```
gen = [0, 5, 7, None, True, "casa"]
for i in gen:
  print(i)

# [OUT]:
# 0
# 5
# 7
# None
# True
# casa
```

Frequentemente, usamos listas, a fim de gerar uma iteração a partir do FOR:

```
lista = [1, 2]
lista.append(3)
lista

# [OUT]:
# [1, 2, 3]
```

ADICIONANDO OU REMOVENDO ELEMENTOS

Podemos adicionar elementos em uma lista usando APPEND, ou EXTEND.

```
lista = [1, 2]
lista.extend([3])
lista

# [OUT]:
# [1, 2, 3]
```

Usando INSERT, podemos especificar a localização de inserção de dados em uma lista:

```
lista = [1, "amarelo", [1, 2], (3,4)]
lista.insert(1, "vermelho")
lista

# [OUT]
# [1, 'vermelho', 'amarelo', [1, 2], (3, 4)]
```

> *"INSERT é computacionalmente caro em comparação com APPEND, porque as referências aos elementos subsequentes*

> *devem ser deslocadas internamente para abrir espaço para o novo elemento" (MCKINNEY, 2019, p. 55).*

A operação inversa de INSERT é POP. Com POP, podemos escolher qual elemento retirar da lista, selecionando seu índice dentro do POP:

```
lista = ["PYTHON", 2020, "UVV-ON", "EAD"]
lista.pop(1)
lista

# [OUT]:
# ['PYTHON', 'UVV-ON', 'EAD']
```

Podemos remover elementos pelo seu conteúdo ao invés de removê-los pelo índice (como é feito no POP). Para tal, utilizamos a função REMOVE que buscará o primeiro elemento da lista que possui tal conteúdo e retirá-lo-á da lista:

```
lista = [1, 2, "três"]
lista.remove("três")
lista

# [OUT]:
# [1, 2]
```

Para verificar se existe um determinado conteúdo na lista, podemos utilizar a palavra-chave IN:

```
lista = [1, 2, "três", [4, 5]]
2 in lista

# [OUT]:
# True
```

Ou podemos utilizar a palavra-chave NOT, a fim de negar a palavra-chave IN:

```
lista = [1, 2, "três", [4, 5]]
2 not in lista

# [OUT]:
# False
```

Se um determinado conteúdo não existe na lista e tentamos removê-lo, teremos um erro no retorno:

```
lista = [1, 2, "três", [4, 5]]
lista.remove(3)

# [OUT]:
# ---------------------------------------------------------------
-# ---------
# ValueError                                Traceback (most recent
# # call last)
# <ipython-input-120-428617b72660> in <module>
#       1 lista = [1, 2, "três", [4, 5]]
# ----> 2 lista.remove(3)

# ValueError: list.remove(x): x not in list
```

Podemos combinar REMOVE e IN, com o intuito de evitar o erro acima:

```
lista = [1, 2, "três", [4, 5]]
if 3 in lista:
  lista.remove(3)
lista

# [OUT]:
# [1, 2, 'três', [4, 5]]
```

CONCATENANDO E COMBINANDO LISTAS

Podemos usar o sinal de mais (+), a fim de concatenar duas listas.

```
lista1 = [1, "a", 2, "b"]
lista2 = [3, "c", 4, "d"]
lista3 = lista1 + lista2
lista3

# [OUT]:
# [1, 'a', 2, 'b', 3, 'c', 4, 'd']
```

Podemos editar uma lista adicionando múltiplos elementos a ela utilizando EXTEND:

```
lista1 = [1, "a", 2, "b"]
lista1.extend([3, "c", 4, "d"])
lista1
```

```
# [OUT]:
# [1, 'a', 2, 'b', 3, 'c', 4, 'd']
```

> *“Observe que a concatenação de lista por adição é uma operação comparativamente custosa, pois uma nova lista deve ser criada e os objetos copiados. Usar EXTEND para acrescentar elementos a uma lista existente, especialmente se você estiver construindo uma lista grande, é geralmente preferível”*
>
> *(MCKINNEY, 2019, p. 56).*

CÓPIA DE LISTAS

Existem dois tipos de cópias de listas, DEEP (profunda) e SHALLOW (sombra).

Quando usamos o método DEEP, fazemos uma cópia e temos uma nova lista:

```
import copy
lista1 = [1, 2, 3, 4, 5]
lista2 = copy.deepcopy(lista1)
lista2[1] = 10
print(lista1)
print(lista2)

# [OUT]:
# [1, 2, 3, 4, 5]
# [1, 10, 3, 4, 5]
```

Podemos perceber que as modificações aportadas à lista1 não afetaram a lista2.

No método SHALLOW (sombra), nós criamos uma outra lista a partir de uma já existente usando a atribuição direta. Ela funciona como espelho e não como cópia, assim todas as modificações aportadas ao espelho refletirão na lista fonte:

```
lista1 = [1, 2, 3, 4, 5]
lista2 = lista1
lista2[1] = 10
print(lista1)
print(lista2)
```

```
# [OUT]:
# [1, 10, 3, 4, 5]
# [1, 10, 3, 4, 5]
```

ORDENAÇÃO

Para ordenar uma lista, sem criar uma nova, utilizamos a função SORT:

```
lista = [1, 5, 7, 3, 6, 2, 4]
lista.sort()
lista

# [OUT]
# [1, 2, 3, 4, 5, 6, 7]

lista = ["c", "a", "d", "b", "f"]
lista.sort()
lista

# [OUT]:
# ['a', 'b', 'c', 'd', 'f']
```

Podemos utilizar funções, para fazer a ordenação, como, por exemplo, utilizar a ordenação por tamanho das palavras (LEN):

```
lista = ["março", "janeiro", "abril", "dezembro", "agosto"]
lista.sort(key=len)
lista

# [OUT]:
# ['março', 'abril', 'agosto', 'janeiro', 'dezembro']
```

BUSCA BINÁRIA E MANTENDO UMA LISTA ORDENADA

O módulo BISECT é utilizado para busca e inserção binária. BISECT.BISECT implementa busca que indica onde um elemento deveria ser adicionado, para manter a lista ordenada.

```
import bisect
lista = [1, 1, 1, 3, 7, 15, 16, 17]
bisect.bisect(lista, 14)

# [OUT]:
# 5
```

BISECT.INSORT implementa a inserção do elemento desejado mantendo a lista ordenada:

```
import bisect
lista = [1, 1, 1, 3, 7, 15, 16, 17]
bisect.insort(lista, 14)
lista

# [OUT]:
# [1, 1, 1, 3, 7, 14, 15, 16, 17]
```

O módulo BISECT não procura saber se a lista está ordenada, tampouco a ordena, pois tais atividades são computacionalmente custosas. Isto por um lado é um problema, já que podemos assim manter uma lista desordenada. Por outro lado, utilizar BISECT é computacionalmente mais rápido do que utilizar SORT, por isso é melhor usar BISECT, quando temos certeza que a lista já está ordenada e queremos adicionar um elemento.

FATIAMENTO

Podemos utilizar a notação de SLICE, a fim de retornar pedaços da lista. A notação é **lista[inicio:parada:passo]**. Onde inicio é incluído, parada não é incluído e passo apresenta de quantos em quantos elementos serão retornados.

- **Considerando lista = [1, 1, 1, 3, 7, 14, 15, 16, 17]:**
- **lista[0:6:2]- retorna uma lista com os 6 primeiros elementos de lista tomados de 2 em 2 (3 elementos, no máximo): [1, 1, 7];**
- **lista[:6:2]- retorna o mesmo que o exemplo anterior já que 0 é o default de início;**
- **lista[8:0:-1]- retorna a lista desde o índice 8 até o índice 0 tomados da direita para a esquerda. O problema é que 0 é excluído;**
- **lista[::-1]- Neste caso, o índice 0 não é excluído;**
- **lista[5:9]- podemos não inserir o passo caso ele seja 1: [14, 15, 16, 17];**
- **lista[5:]- o default de parada é o fim da lista: [14, 15, 16, 17];**

- **lista[-1:]- -1 é o índice do fim da lista: [17].**

Segue uma imagem que retrata como podemos pensar os índices dentro da lista:

Ilustrando a convenção de fatiamento em PYTHON.

ENUMERATE

Quando estamos iterando sobre uma lista, comumente queremos utilizar também o índice dos elementos. Por exemplo, se quisermos a ordem dos elementos de uma lista:

```
lista = [1, 1, 1, 3, 7, 14, 15, 16, 17]
i = 0
for value in lista:
  print(value, ":", i)
  i += 1

# [OUT]:
# 1 : 0
# 1 : 1
# 1 : 2
# 3 : 3
# 7 : 4
# 14 : 5
# 15 : 6
```

```
# 16 : 7
# 17 : 8
```

Podemos utilizar ENUMERATE para retornar i:

```
lista = [1, 1, 1, 3, 7, 14, 15, 16, 17]
for i, value in enumerate(lista):
  print(value, ":", i)

# [OUT]:
# 1 : 0
# 1 : 1
# 1 : 2
# 3 : 3
# 7 : 4
# 14 : 5
# 15 : 6
# 16 : 7
# 17 : 8
```

SORTED

Ela retorna uma nova lista de elementos ordenados de qualquer sequência. Em LISTAS, mas também em STRINGS, por exemplo.

```
sorted([4, 1, 5, 3, 2])

# [OUT]:
# [1, 2, 3, 4, 5]

sorted("inspira, expira")

# [OUT]:
# [' ', ',', 'a', 'a', 'e', 'i', 'i', 'i', 'n', 'p', 'p', 'r', 'r',
's', 'x']
```

ZIP

Podemos utilizar ZIP, a fim de gerar “pares” entre listas em suas ordens.

```
lista1 = ["amor", "bahia", "casa"]
lista2 = [1, 2, 3]
zipped = zip(lista1, lista2)
list(zipped)
```

```
# [OUT]:
# [('amor', 1), ('bahia', 2), ('casa', 3)]
```

Podemos utilizar mais de duas sequências, a fim de fazer o zip. No entanto, a sequência com menor número de elementos será usada como referência:

```
lista1 = ["amor", "bahia", "casa"]
lista2 = [1, 2, 3]
lista3 = ["amizade", "pernambuco"]
zipped = zip(lista1, lista2, lista3)
list(zipped)

# [OUT]:
# [('amor', 1, 'amizade'), ('bahia', 2, 'pernambuco')]
```

Podemos utilizar ZIP, com o intuito de iterar sobre múltiplas sequências ao mesmo tempo:

```
lista1 = ["amor", "bahia", "casa"]
lista2 = ["amizade", "pernambuco", "rua"]
for i, (a, b) in enumerate(zip(lista1, lista2)):
  print("{0}: {1}, {2}".format(i, a, b))

# [OUT]:
# 0: amor, amizade
# 1: bahia, pernambuco
# 2: casa, rua
```

Se pretendemos criar uma sequência tautológica utilizando apenas a operação lógica OU:

```
p = [True, False, False, True]
q = [True, True, True, False]
PorQ = []
for (a,b) in zip(p, q):
  PorQ.append(a or b)
PorQ

# [OUT]:
# [True, True, True, True]
```

Zip requer mais de um argumento, mas, quando o que temos é um único argumento (uma lista, cujos elementos também são

listas, ou outro tipo de sequência), o * em uma chamada de função “desempacota” uma lista (ou outro iterável), tornando cada um de seus elementos um argumento separado.

Por exemplo, vamos criar uma lista já “zippada” que contenha os nomes e sobrenomes de atrizes e queremos retornar duas listas uma com os nomes e outra com os sobrenomes.

Utilizaremos o operador * , para fazer a operação inversa de zip:

```
atrizes = [("Anne", "Hathaway"), ("Lupita", "Nyongo"), ("Natalie",
"Portman")]
nome, sobrenome = zip(*atrizes)
print(nome)
print(sobrenome)

# [OUT]:
# ('Anne', 'Lupita', 'Natalie')
# ('Hathaway', 'Nyongo', 'Portman')
```

COMPREENSÕES DE LISTAS

Python tem uma ferramenta conhecida como List Comprehensions que torna a formação de algumas listas a partir de outras listas um processo mais simples.

> ***[<expressão> for <item> in <sequência/iterador>]***
>
> *“Esta instrução é avaliada em uma lista cujos itens são obtidos aplicando <expression>, uma expressão Python tipicamente envolvendo a variável <item>, a cada item do contêiner iterável <seqüência / iterador>.”*
>
> *(PERKOVIC, 2016, p. 446).*

Por exemplo, leve em consideração o código abaixo:

```
nomes = ["Alberto", "Amanda", "lindsey", "Marina", "augusto",
"Ana", "anne", "Luzia"]
resultado = []
for x in nomes:
  if x[0].lower() == "a":
    resultado.append(x.upper())
```

```
resultado

# [OUT]:
# ['ALBERTO', 'AMANDA', 'AUGUSTO', 'ANA', 'ANNE']
```

Repare que, no caso acima, temos uma lista preexistente (nomes), da qual vamos retirar os dados, para formar nossa outra lista (resultado). Também temos uma condição (a primeira letra do item é “a”), que selecionará os itens que serão adicionados à nova lista. Por fim, os itens escolhidos para a nova lista são editados antes de serem adicionados (são transformados em letras maiúsculas).

Como estas tarefas são usuais em Python e pensando na praticidade de programação, foi desenvolvida uma linha de código que substitui toda a criação desta nova lista do jeito tradicional. A forma a seguir pode ser utilizada:

```
nomes = ["Alberto", "Amanda", "lindsey", "Marina", "augusto",
"Ana", "anne", "Luzia"]
resultado = [x.upper()
for x in nomes
  if x[0].lower()=="a"]
resultado

# [OUT]:
# ['ALBERTO', 'AMANDA', 'AUGUSTO', 'ANA', 'ANNE']
```

As duas formas anteriores têm o mesmo funcionamento, no entanto a segunda é mais simples. Enquanto que na primeira precisamos explicitamente adicionar (append) os itens na nova lista, na segunda, supõe-se que essa é a operação desejada e, portanto, ela é omitida.

Se no primeiro caso temos a ordem: FOR; IF; EDIÇÃO. No segundo caso, a ordem é: EDIÇÃO; FOR; IF. Ainda no segundo caso, fazemos toda esta operação em uma única linha em entre colchetes [ ].

Sintaxe (padrão) Python para LIST COMPREHENSIONS.

Da mesma forma se precisamos iterar sobre listas que estão dentro de listas, ou mesmo tuplas que estão dentro de listas, usualmente faríamos isto com um FOR dentro de outro, exemplo:

```
nomes = [["Alberto", "augusto", "Hygor"],
["Amanda", "lindsey", "Marina", "Ana", "anne", "Liniker"]]
sem_letra_a = []
for pessoas in nomes:
  for x in pessoas:
    if x[0].lower() != "a":
      sem_letra_a.append(x)

sem_letra_a

# [OUT]:
# ['Hygor', 'lindsey', 'Marina', 'Liniker']
```

No exemplo acima, como temos listas dentro de uma lista, precisamos iterar dentro da lista externa for pessoas in nomes, e, assim, encontraremos duas listas. Então, iteramos dentro das listas internas for x in pessoas. Podemos, desta forma, usar a mesma lógica nesta mesma ordem, com a finalidade de usar a list comprehensions:

```
nomes = [["Alberto", "augusto", "Hygor"],
["Amanda", "lindsey", "Marina", "Ana", "anne", "Liniker"]]
sem_letra_a = [x for pessoas in nomes for x in pessoas if
x[0].lower() != "a"]
sem_letra_a

# [OUT]:
# ['Hygor', 'lindsey', 'Marina', 'Liniker']
```

Imaginemos, então, que temos listas dentro de listas dentro de uma lista:

```
lista_interna_1 = ["vitória","Montevideo"]
lista_interna_2 = ["Argentina","Cuba"]
lista_interna_3 = ["Lille","londres"]
lista_interna_4 = ["Ucrânia","Irlanda"]
lista_do_meio_1 = [lista_interna_1,lista_interna_2]
lista_do_meio_2 = [lista_interna_3, lista_interna_4]
lista_externa = [lista_do_meio_1,lista_do_meio_2]
lista_externa

# [OUT]:
```

```
# [[['vitória', 'Montevideo'], ['Argentina', 'Cuba']],
#  [['Lille', 'londres'], ['Ucrânia', 'Irlanda']]]
```

Claro, poderíamos declarar diretamente lista_externa como uma lista de listas de listas:

```
lista_externa = [[["vitória","Montevideo"],["Argentina","Cuba"]] ,
                 [["Lille","londres"],["Ucrânia","Irlanda"]]]
```

Mas declaramos separadamente, a fim de mostrar que, no caso acima, temos três camadas de listas e precisamos iterar sobre todos estes elementos, para fazer uma certa seleção de dados e criar uma outra lista. Teríamos então:

```
lista_externa = [[["vitória","Montevideo"],["Argentina","Cuba"]],
[["Lille","londres"],["Ucrânia","Irlanda"]]]
lista_add = []
for lista_do_meio in lista_externa:
  for lista_interna in lista_do_meio:
    for x in lista_interna:
      if len(x) > 6:
        lista_add.append(x)
lista_add

# [OUT]:
# ['vitória', 'Montevideo', 'Argentina', 'londres', 'Ucrânia',
'Irlanda']
```

No código acima, selecionamos todas as palavras que tenham mais de 6 letras e as adicionamos em lista_add. Seguindo a lógica do list comprehensions, o código ficaria:

```
lista_externa = [[["vitória","Montevideo"],["Argentina","Cuba"]],
[["Lille","londres"],["Ucrânia","Irlanda"]]]
lista_add = [x for lista_do_meio in lista_externa
             for lista_interna in lista_do_meio
             for x in lista_interna if len(x)>6]
lista_add

# [OUT]:
# ['vitória', 'Montevideo', 'Argentina', 'londres', 'Ucrânia',
'Irlanda']
```

INTRODUÇÃO ÀS ESTRUTURAS DE DADOS HETEROGÊNEA – LIST

– Parte 01

INTRODUÇÃO ÀS ESTRUTURAS DE DADOS HETEROGÊNEA – LIST

– Parte 02

INTRODUÇÃO ÀS ESTRUTURAS DE DADOS HETEROGÊNEA – LIST – Parte 03

Conclusão

3. Conclusão

Este tópico procurou mostrar a importância das estruturas de dados heterogêneas, que representam uma grande vantagem da linguagem PYTHON.

Nesse viés, muitas vantagens podemos tirar das listas. Podemos fazer cálculos sobre a lista, ordená-la, encontrar conteúdo, inserir informação onde quisermos, guardar tipos diferentes de informação, entre muitas outras funcionalidades.

É importante ter em mente as capacidades das listas. Elas certamente serão úteis em todos os códigos de complexidade média, ou alta em Python. Logo, essa estrutura vem para trazer facilidades imensas, mesmo que, por outro lado, seja computacionalmente custoso lidar com tipos variados de dados.

4. Referências

MCKINNEY, Wes. Python for Data Analysis. United States of America: O'Reilly, 2019.

PERKOVIC, Ljubomir. Introdução à computação usando Python: um foco no desenvolvimento de aplicações. Rio de Janeiro, RJ: LTC, 2016. 489 p. ISBN 9788521630814.

Parabéns, esta aula foi concluída!

O que achou do conteúdo estudado?

Péssimo Ruim Normal Bom Excelente

Deixe aqui seu comentário

Mínimo de caracteres: 0/150

Enviar