INE5404 - Programação Orientada a Objetos II - Listas e Dicionários

Operações básicas e exemplos executáveis

Prof. Jônata Tyska Carvalho

Lembre-se que este é um documento interativo. Todos os exemplos de código nele contidos podem e devem ser executados (botão 'play' ao lado de cada código ou shift+enter) para que você possa ver os conceitos apresentados funcionando na prática.

Sinta-se motivado a executar os exemplos múltiplas vezes, modificá-los adicionando e testando novas funcionalidades. Tente criar e testar hipóteses prevendo qual será o resultado da execução dos códigos com as suas alterações. Caso suas alterações causem erros, tente entender os erros causados e como corrigí-los. Considere este documento não apenas como uma aula, mas também como um ambiente interativo de experimentação e descoberta.

Lembre-se que aprender a programar requer prática e dedicação. Discuta problemas e ideias com os colegas através do fórum do Moodle, e não deixe de nos contatar caso tenha dúvidas ou dificuldades. Bons estudos!

Listas

Declaração de listas

Listas são declaradas no momento da atribuição de valores, assim como as demais variáveis em Python. É possível declarar uma lista vazia, sem elementos, ou uma lista que começa com um certo número de elementos. O símbolo utilizado para representar listas é o colchete. Veja abaixo alguns exemplos de declaração de listas.

In [ ]:
```
#declarando uma lista vazia
lista = []
print(lista)
```

In [ ]:
```
#declarando uma lista com inteiros
lista= [3,9,2,1,22]
print(lista)
```

In [ ]:
```
#declarando uma lista com floats
lista=[1.1,2.0,3,9,15.2]
```

```
print(lista)
```

In [ ]:
```
#declarando uma lista com inteiros, floats e caracteres
lista=[7.2,"a",5,9.7,"casa"]
print(lista)
```

Perceba como nos primeiros exemplos, apesar de muitos valores terem sido colocados na lista, todos eles pertenciam a um mesmo tipo de dado (inteiros ou floats).

Entretanto o último exemplo mostra como as listas em Python são flexíveis (heterogêneas), permitindo o armazenamento de múltiplos tipos dados em uma mesma variável.

Note que as listas podem armazenar qualquer tipo de dado, incluindo outras listas. Matrizes em Python são nativamente representadas desta forma, usando listas de listas. Por exemplo, uma matriz bidimensional de 3 linhas por 3 colunas, é representada em Python através de uma lista que armazena 3 listas, tendo cada uma destas listas 3 elementos. Cada lista armazenada é uma linha da matriz, enquanto cada elemento destas listas é uma coluna daquela linha. Veja o exemplo abaixo.

In [ ]:
```
#representação de matriz usando listas - perceba que as quebras de linha foram i
matriz=[
        [1,2,3],
        [4,5,6],
        [7,8,9]
       ]
print(matriz)
```

Acessando elementos de uma lista

O elementos de uma lista são acessados pelo nome da variável, seguido de um ou mais índices (no caso de listas de listas) entre colchetes. Ver a tabela e o exemplo abaixo.

Para o exemplo abaixo temos:

```
lista=[7.2,"a",5,9.7,"casa","100"]
```

Índice:	**0**	**1**	**2**	**3**	**4**	**5**
valor:	7.2	"a"	5	9.7	"casa"	100

In [1]:
```
lista=[7.2,"a",5,9.7,"casa","100"]
print(lista[2])
print(lista[3])
print(lista[4])
```

```
5
9.7
casa
```

Fique atento para que seus programas não tentem acessar índices inexistentes em uma lista, isso irá gerar um erro como mostrado no exemplo abaixo.

In [ ]:
```
lista=[10,22,35]
print(f"Primeiro elemento {lista[0]}")
print(f"Segundo elemento {lista[1]}")
print(f"Terceiro elemento {lista[2]}")
print(lista[3])#!!!! índice inexistente, irá gerar um erro
```

Quando temos listas dentro de listas como no exemplo abaixo, temos diversas possibilidades de acesso. Se utilizarmos apenas o nome da variável, sem índices, acessamos todos os valores simultamente. Ao utilizar apenas um índice, acessamos todos os valores de uma determinada linha. E por fim, ao utilizar dois índices, o primeiro se refere a uma determinada linha, e o segundo a um elemento dela, como mostra o exemplo abaixo.

```
matriz=[[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

Índice:	**0**	**1**	**2**
valor:	[1, 2, 3]	[4, 5, 6]	[7, 8, 9]

In [2]:
```
matriz=[
        [1,2,3],
        [4,5,6],
        [7,8,9]
       ]
#mostrando todos os elementos
print(matriz)
#mostrando uma linha por vez
print(f"Linha 1={matriz[0]}")
print(f"Linha 2={matriz[1]}")
print(f"Linha 3={matriz[2]}")
#mostrando o segundo elemento terceira linha
print(f'Este é o segundo elemento da terceira linha: {matriz[2][1]}')
```

```
[[1, 2, 3], [4, 5, 6], [7, 8, 9]]
Linha 1=[1, 2, 3]
Linha 2=[4, 5, 6]
Linha 3=[7, 8, 9]
Este é o segundo elemento da terceira linha: 8
```

In [3]:
```
#exemplo que percorre e imprime a matriz com dois whiles - não esqueça de execut
i=0
while i<3:
    j=0
    while j<3:
        print(f"{matriz[i][j]} ",end="") #dica: ao utilizar o parâmetro end="" r
        j=j+1
    print() #este print serve para realizar a quebra de linha a cada linha da ma
    i=i+1
```

```
1 2 3
4 5 6
7 8 9
```

In [4]:
```
#exemplo que percorre e imprime a matriz utilizando dois laços de repetição for
for linha in matriz:
    for valorColuna in linha:
```

```
        print(f"{valorColuna} ",end="")
    print()
```

```
1 2 3
4 5 6
7 8 9
```

Modificando valores em uma lista

Você percebeu que acessamos aos elementos de uma lista da mesma forma que acessamos os caracteres de uma string? Isso mesmo, usamos índices para acessar tantos os elementos de uma lista, quanto os caracteres de uma string. Com uma lista porém, diferente das strings que não nos permitem modificar caracteres individualmente, é possível modificar os elementos da lista fazendo uma atribuição a posição desejada, conforme mostra o exemplo abaixo.

Exemplo 1. Recebe 5 valores informados pelo usuário, e ao final mostra a soma e a média dos valores

In [ ]:
```
contador=0
soma=0
listaValor=[0,0,0,0,0] #inicializamos a lista com valores 0
while contador<5:
    valor=float(input(f"Digite um valor numérico ({contador+1} de 5): "))
    listaValor[contador]=valor #modificamos o valor atribuído inicialmente pelo
    soma=soma+valor
    contador=contador+1
print(f"O usuário informou os seguintes valores: {listaValor}")
print(f"A soma desta valores é {soma} e a média é {soma/contador}") #perceba com
```

Fatiamento (slicing) de listas

Assim como as strings, as listas em Python permitem a realização do fatiamento ou *slicing*, permitindo o acesso a partes específicas das listas. Através da especificação de um índice inicial e final, separados por dois pontos, podemos acessar parte da lista contida no intervalo informado, conforme mostra o exemplo abaixo.

In [5]:
```
lista=[1,2,3,4,5,6,7,8,9]
print(lista[2:5]) #lembre-se que o fatiamento considera o intervalo [inicio,fim)
```

```
[3, 4, 5]
```

O fatiamento de listas funciona exatamente da mesma forma como nas strings, com a diferença que podemos modificar valores específicos da lista usando o fatiamento, veja o exemplo abaixo.

In [6]:
```
lista=[1,2,3,4,5,6,7,8,9]
lista[2:5]=[30,40,50]
print(lista)
```

```
[1, 2, 30, 40, 50, 6, 7, 8, 9]
```

Cópia de listas

É muito importante entender o comportamento da cópia de listas em Python. Ao copiar uma lista atribuindo uma determinada variável a outra, como mostra o exemplo abaixo, estamos na verdade apontando as duas variáveis para o mesmo endereço de memória. Assim, ao alterarmos o valor de uma variável, estaremos alterando também o valor da outra, pois as duas apontam para o mesmo endereço da memória.

In [ ]:
```
lista1=[1,2,3,4,5]

print(f"A variável lista1 começa com estes valores {lista1}")

lista2=lista1 #ao fazer isso estamos apontando as duas variáveis, lista1 e lista

lista2[0]=100 #desta forma, ao alterarmos o primeiro valor de lista2, estaremos

print(f"Os valores de lista2 após a alteração são {lista2}")
print(f"Os valores de lista1 agora são {lista1}")
```

É muito importante que você se lembre deste comportamento ao escrever os seus programas. Você deve estar se perguntando "e como eu faço para efetivamente copiar uma lista?", não é mesmo? Uma das possibilidades é utilizar o fatiamento para realizar tal operação. Lembre-se que se não especificarmos o primeiro ou o último elemento do fatiamento estamos nos referindo ao primeiro e ao último elemento da lista. Assim, ao não informar ambos, estamos acessando todos os valores de uma lista, que são assim devidamente copiados para outra variável como mostra o exemplo abaixo.

In [ ]:
```
lista1=[1,2,3,4,5]

print(f"A variável lista1 começa com estes valores {lista1}")

lista2=lista1[:] #perceba que estamos utilizando o fatiamento sem especificar o

lista2[0]=100 #nesse caso estamos efetivamente alterando a cópia da lista1, e nã

print(f"Os valores de lista2 após a alteração são {lista2}")
print(f"Os valores de lista1 continuam sendo {lista1}")
```

Tamanho de listas

Novamente, assim como a função *len* retorna o tamanho de uma string, essa mesma função retorna o tamanho de uma lista, como mostram os exemplos abaixo.

In [ ]:
```
lista=[50,40,30,20,10]
tamanhoLista=len(lista)
print(f"O tamanho da lista é {tamanhoLista}")
```

A função len é frequentemente utilizada ao utilizar um while para percorrer uma lista da qual não sabemos o tamanho, assim garantimos que não vamos tentar acessar um índice que não existe na lista. Veja o exemplo do código abaixo.

In [ ]:
```
lista=[33,66,99,132] #note que você pode remover ou adicionar elementos da lista
pos=0
while pos<len(lista):
```

```
    print(f"Elemento {pos+1} = {lista[pos]}")
    pos=pos+1
```

Adição de elementos em uma lista

Muitas vezes é necessário adicionar novos elementos a uma lista, e isso é possível em Python através da função append.

In [ ]:
```
lista=[]#declara uma lista vazia
lista.append(5)
lista.append(18)
print(lista)
```

In [ ]:
```
#programa que lê um número de valores indefinidos informados pelo usuário até qu
listaValores=[] #declara uma lista vazia
soma=0
while True:
    valor=float(input("Digite um valor numérico (digite 0 para encerrar): "))
    if valor==0:
        break; #lembre-se que a instrução break interrompe explicitamente um laç
    listaValores.append(valor) #adiciona o valor informado à lista
    soma=soma+valor
numeroValores=len(listaValores) #usa a função len para verificar o número de val
print(f"O usuário informou {numeroValores} valor(es), a soma deste(s) é {soma:.1
print(f"Os valores informados pelo usuário foram: {listaValores}")
```

É possível adicionar elementos em uma lista através do operador + (soma). Note que neste caso não estamos realizando operações matemáticas, quando os operandos do operador + são listas, a operação realizada é similar à concatenação de strings, porém estamos concatenando listas. Veja exemplos a seguir.

In [ ]:
```
lista=["a",1,"b",2,"c"]
print(f"Lista com valores iniciais {lista}")
lista=lista+[3]
print(f"Lista após a adição de elemento {lista}")
```

In [ ]:
```
#exemplo de concatenação de listas
lista1=[1,2,3]
lista2=[4,5,6]
lista3=lista1+lista2
print(f"Listas concatenadas {lista3}")
```

Remoção de elementos de uma lista

Por vezes pode ser necessária a remoção de elementos de uma lista. Da mesma forma que podemos inserir novos elementos a uma lista, podemos também apagar elementos existentes. A sintaxe é um pouco diferente da adição de elementos, para apagar elementos usamos a instrução *del* (alternativa para a função *remove*), que precede o elemento que desejamos remover. Sempre que um elemento da lista é apagado, os índices são reorganizados. Veja abaixo alguns exemplos.

In [ ]:
```
lista=["ABC",4.7,10,22,"novo",33,44]
print(f"Os valores iniciais da lista são {lista}")
```

```
print(f"Valor no índice 5 = {lista[5]}")
del lista[4]
print(f"Elemento de índice 4 apagado, os valores da lista são {lista}")
print(f"Valor no índice 5 = {lista[5]}")
#tenha cuidado para não tentar acessar índices inexistentes após a remoção de el
```

Perceba que a função *del* pode ser usado também para apagar variáveis simples, isto é, liberar a memória utilizada por ela, e também apagar elementos de outras estruturas de dados, como os dicionários que serão apresentados a seguir.

Dicionários

Recapitulando, as listas em Python são estruturas de dados compostas, isto é, permitem o armazenamento de múltiplos valores, heterogêneas, os valores podem ser de diferentes tipos de dados. Para acessar cada elemento de uma lista usamos um índice numérico, mais especificamente um valor inteiro.

A linguagem Python oferece um tipo de dado importante chamado de **dicionário**. Os dicionários têm um comportamento similar ao das listas, entretanto além da utilização de índices númericos, eles permitem a utilização de strings (e até mesmo outros tipos de dados (sempre imutáveis)) para indexar valores, criando assim uma estrutura de chaves-valores. Tal estrutura é muito útil em diversas ocasiões, por um exemplo quando queremos armazenar de forma intuitiva os dados da tabela de preços abaixo.

Produto	Preço
Banana	2.50
Alface	1.20
Tomate	3.70

Os dicionários são representados e declarados utilizando chaves ({}), chaves e valores são separados por dois-pontos (:), e cada par chave-valor é separado por vírgulas (,). Veja a sintaxe abaixo para a declaração de um dicionário.

```
<variável>={<chave1>:<valo1>,<chave2>:<valor2>, ... ,
<chaveN:valorN>}
```

Veja a seguir um exemplo do código em que a tabela de preços acima é representada utilizando um dicionário.

In [ ]:
```
tabela={"Banana":2.50 ,"Alface":1.20,"Tomate":3.70}
print(f'O preço do tomate é R$ {tabela["Tomate"]:.2f}')
```

Como vimos no exemplo acima, o acesso aos elementos de um dicionário é feito de maneira semelhante ao acesso aos elementos de uma lista, com a diferença que utilizamos as chaves como índices, e tais chaves podem ser do tipo string (e outros). Na realidade em um mesmo dicionário podemos ter chaves de tipos diferentes como strings, inteiros e mesmo floats como mostra o exempo abaixo.

In [ ]:
```
multiChaves={7.2:"chave do tipo float","segundo valor":"chave do tipo string",9:
print(f"Primeiro elemento: {multiChaves[7.2]}")
print(f"Segundo elemento: {multiChaves['segundo valor']}")
print(f"Terceiro elemento: {multiChaves[9]}")
```

Adição de elementos em um dicionário

Ao atribuir um valor a um índice (chave) existente de um dicionário, substituímos o valor armazenado anteriormente. Quando atribuimos um valor a um índice inexistente, estamos na realidade adicionando um novo valor ao dicionário, e o índice antes inexistente, passa a referenciar o valor recém atribuído. Veja o exemplo a seguir.

In [ ]:
```
tabela={"Banana":2.50 ,"Alface":1.20,"Tomate":3.70}
print(f"Tabela inicial = {tabela}")
tabela["Feijão"]=5.20 #adição de elemento
tabela["Alface"]=0.95 #alteração de valor
print(f'Nova tabela = {tabela}')
```

O operador *in* permite verificar a existência de uma chave em um dicionário. Este operador é bastante útil para a utilização de dicionários como contadores. Veja o exemplo abaixo.

In [ ]:
```
#conta o número de vezes que o usuário digitou uma palavra, até que ele digite 0
contador={}
while True:
    palavra=input("Digite uma palavra ou 0 para encerrar o programa: ")
    if palavra=="0":
        break;
    else:
        if palavra in contador: #se a palavra já for uma chave do dicionário, in
            contador[palavra]=contador[palavra]+1
        else: #se a palavra informada não for uma chave, atribua o valor 1 a ela
            contador[palavra]=1
print(f"A contagem de palavras foi a seguinte: {contador}")
```

```
Digite uma palavra ou 0 para encerrar o programa: casa
Digite uma palavra ou 0 para encerrar o programa: bola
Digite uma palavra ou 0 para encerrar o programa: sorvete
Digite uma palavra ou 0 para encerrar o programa: casa
Digite uma palavra ou 0 para encerrar o programa: limão
Digite uma palavra ou 0 para encerrar o programa: bola
Digite uma palavra ou 0 para encerrar o programa: casinha
Digite uma palavra ou 0 para encerrar o programa: casa
Digite uma palavra ou 0 para encerrar o programa: 0
A contagem de palavras foi a seguinte: {'casa': 3, 'bola': 2, 'sorvete': 1, 'li
mão': 1, 'casinha': 1}
```

Acessando apenas as chaves de um dicionário

A função *keys* retorna as chaves de um determinado dicionário, podendo ser utilizada como mostram os exemplos abaixo.

In [ ]:
```
tabela={"Banana":2.50 ,"Alface":1.20,"Tomate":3.70}
print(tabela.keys())
```

Entretanto é importante destacar que a função *keys* não retorna diretamente uma lista em Python, ela retorna um tipo de dado chamado dict_keys. Por esse motivo não é possível utilizar índices para acessar as chaves retornadas. As duas principais formas de acessar tais valores são: (i) convertendo o retorno da função keys para uma lista, e (ii) usando tal retorno em conjunto com um laço de repetição for. Veja exemplos abaixo.

In [7]:
```
#convertando o retorno da função keys para uma lista
tabela={"Banana":2.50 ,"Alface":1.20,"Tomate":3.70}
listaChaves = list(tabela.keys())
print(f"Primeira chave: {listaChaves[0]}")
print(f"Primeira chave: {listaChaves[1]}")
print(f"Primeira chave: {listaChaves[2]}")
```

```
Primeira chave: Banana
Primeira chave: Alface
Primeira chave: Tomate
```

In [8]:
```
#combinando o retorno da função keys com um laço de repetição for
tabela={"Banana":2.50 ,"Alface":1.20,"Tomate":3.70}
contador=1
for chave in tabela.keys():
    print(f"Chave {contador} = {chave}")
    contador=contador+1
```

```
Chave 1 = Banana
Chave 2 = Alface
Chave 3 = Tomate
```

Acessando apenas os valores de um dicionário

De forma análoga a função *keys*, a função *values* retorna apenas os valores de um dicionário. Lembre-se, assim como a função *keys*, o retorno da função *values* não é uma lista, e sim um tipo específico chamado dict_values. Assim, para acessar cada elemento usa-se a conversão dele para listas, ou usa-se um laço de repetição for.

In [ ]:
```
#acessando apenas os valores de um dicionário
dicionario={202:"Sala de aula",101:"Secretaria",103:"Auditório"}
for valor in dicionario.values():
    print(f"Valor = {valor}")
```

Iterando chaves e valores de um dicionário

A função *items* retorna todos os pares de chave-vaor contidos em um dicionário. Ao combiná-la com um laço de repetição for, por exemplo, é possível percorrer cada elemento de um dicionário, acessando ambos chave e valor de cada elemento. Veja o exemplo abaixo.

In [ ]:
```
tabela={"Banana":2.50 ,"Alface":1.20,"Tomate":3.70}
for chave,valor in tabela.items():
    print(f"Compre {chave} por apenas R$ {valor:.2f}")
```

Cópia de dicionários

Note que da mesma forma como acontece com as listas, ao atribuirmos um dicionário a uma outra variável, estamos apenas apontando as duas variáveis para o mesmo endereço de memória. Assim para realizar a cópia de um dicionário usa-se a função *copy* (em caso de múltiplos níveis, lembre de usar *deepcopy*), veja exemplos abaixo.

In [ ]:
```
#atribuição simples faz as variáveis apontarem para o mesmo endereço de memória
dic1={202:"Sala de aula",101:"Secretaria",103:"Auditório"}
print(f"Valores em dic1 antes da mudança {dic1}")
dic2=dic1
dic2[202]="Mudança de sala - Direção"
print(f"Valores em dic2 {dic2}")
print(f"Valores em dic1 {dic1}")
```

In [ ]:
```
#cópia usando copy cria dois dicionários (duas áreas de memória) diferentes
dic1={202:"Sala de aula",101:"Secretaria",103:"Auditório"}
print(f"Valores em dic1 antes da mudança {dic1}")
dic2=dic1.copy() #atribui a dic2 uma cópia do dicionário armazenado na variável
dic2[202]="Mudança de sala - Direção"
print(f"Valores em dic2 {dic2}")
print(f"Valores em dic1 {dic1}")
```