Introdução aos vetores em *Python*

Como estudado no texto a respeito do conceito de vetores, um vetor é uma sequência de dados ocupando posições consecutivas de memória e por isso existe uma ordem natural (o primeiro elemento, o segundo e assim por diante). A grande vantagem de usar vetor é poder trabalhar com um grande número de variáveis utilizando um único nome (para a variável). Para isso existe uma sintaxe especial para pegar elementos do vetor em posições específicas (como o primeiro elemento ou o décimo elemento).

A linguagem *Python* encapsula bastante o conceito de vetor, agregando funcionalidades e por isso geralmente o material didático sobre *vetores* em *Python* utiliza o nome *lista*.

Neste texto: entradas e saídas; definir/imprimir vetor; definir "string"; referência a listas; definir/imprimir matriz.

1. Como ler vários dados de uma só vez

Eventualmente é interessante implementar um código que possibilite a entrada de vários valores, com um único `ENTER`. Por exemplo, se estiver copiando uma sequência de dados de uma tabela.

Para fazer isso em *Python* é necessário algum "truque" para, ler tudo como uma *cadeia de caracteres* (*string*), depois quebrá-la e transformar nos valores adequados. Além disso, existe uma grande distinção se estiver usando *Python 2* ou *Python 3*, no *Python 2* é preciso usar uma função especial, a `raw_input`, que no *Python 3* sumiu! No *3*, o `input()` passou a ler tudo como caracteres.

O truque básico é ler como *string*, quebrar em itens (usando os separadores, geralmente espaço em branco) e por fim converter para o tipo desejado. Por exemplo, para ler dois valores, o primeiro sendo inteiro e o seguindo flutuante, podemos fazer:

Tab. 1. Exemplos de técnicas para ler vários dados com um único ENTER em Python.

Python 2	*Python 3*

`linha = raw_input();`
`itens = linha.split();`
`n, val = int(itens[0]),`
`float(itens[1]);`
`print("n=%d, val=%f" % (n,val));`
	`linha = input();`
	`itens = linha.split();`
	`n, val = int(itens[0]),`
	`float(itens[1]);`
	`print("n=%d, val=%f" % (n,val));`

Note que usamos um formatador especial `%d` para valores inteiros e `%f` para ponto flutuante.

No exemplo acima precisamos converter o primeiro item para inteiro e o segundo para *flutuante*. Entretanto se todos itens forem de mesmo tipo, existe uma função que realiza automática o laço percorrendo todos os elementos e convertendo um a uma, é o mapeamento `map`, como ilustrado abaixo. Experimente o código com um única linha de entrada de dados: `0.0 1.0 1.5 2.0 3.0 3.5 <ENTER>`.

Tab. 2. Exemplos de técnicas para ler vários dados "reais" com um único ENTER em Python.

Python 2	*Python 3*

`linha = raw_input();`
`itens = linha.split();`
`vetor = map(float, itens); # Erro se tentar: map(int, itens)`
`tam = len(vetor);`
`for i in range(tam) :`
`  print("%2d | %4.1f" % (i,vetor[i]));`
	`linha = input();`
	`itens = linha.split();`
	`vetor = list(map(float, itens)); # Erro se tentar: map(int, itens)`
	`tam = len(vetor);`
	`for i in range(tam) :`
	`  print("%2d | %4.1f" % (i,vetor[i]));`

O comando `linha.split()` (método) percorre os dados da *cadeia de caracteres*, quebrando-o em itens (separados por espaço em branco).
O comando `map(float, itens)` percorre os dados do *vetor/lista* `itens`, tentando transformar seus elementos em *flutuante* (se algum deles não puder tornar-se *flutuante*, um erro é lançado).
O comando `vet = list(.)` gera uma *lista* (tem comportamement semelhante aos "velhos" *vetores* de outras linguagens), que portanto poder ser indexada (como em `vet[0]`).

Note que os formatadores `%d` e `%f` foram usado com parâmetros, `%2d` e `%4.1f`. O primeiro indica que devemos ajutar 2 posições à direita e o segundo ajustar 4 posições à direito e usar apenas 1 casa decimal. Estes formatadores permitem saídas mais interessantes, em particular a construção de tabelas.

Se desejar examinar um código com várias opções sobre entrada de valores em uma única linha, pegue esta versão para *Python 3* ou se ainda usar o *Python 2*, pegue este código.
Esperimente realizar alterações no código, examine os formatadores, troque "`float`" por "`int`", mude a forma de digitar os dados, examine e procure compreender as mensagens de erro, acerte-a.
Teste até estar seguro de ter entendido o `split`, o `map` e os formatadores `%d` e `%f`.

2. Defininido e imprimindo uma lista

Pode-se construir uma lista em *Python* fornecendo todos seus elementos, como em: `vet = [21, 22, 23, 24];` (o finalizador `;` só é obrigatório se tiver mais de um comando na mesma linha).

Pode-se também construir uma lista em *Python* de modo interativo, agregando novo elemento a cada passo usando a função comando `append(.)`, como em: `vet.append(10+i);`, que anexa o elemento de valor `10+i` à lista.

No exemplo a seguir construímos uma lista com *n* (usando o valor fixo *10* no código) elementos, iniciando no *10+i* e seguindo até o *10+n-1*. Nesse exemplo ilustramos o uso de iterador (*in*) e o anexador de novo elemento em lista (*append*).

Exemplo 1. Construir uma lista e imprimir seus elementos.
```
# Python2 para 'print' sem pular linha : print("*" , end = "");
from __future__ import print_function
```

```
n = 10;

# Definir vetor/lista com: (10,11,12,13,...10+n-1)
vet = []; # define um vetor vazio
for i in range(0, n, 1) : # ou mais simples neste caso "for i in range(n)"
  vet.append(10+i); # anexe mais um elemento `a lista
# Imprime o vetor/lista de uma so' vez (na mesma linha)
print("\nO vetor: ", end=""); # informe o que vira impresso a seguir (na
mesma linha devido ao parametro end=""
print(vet);

# Imprimir vet em linha unica na forma "( 0,  0) ( 1,   1)..."
print("\nNovamente o vetor impresso de 2 modos: "); # informe o que vira
impresso a seguir (e "quebre" a linha)
i = 0;
for item in vet : # "item in vet" e' um iterador, item comeca em vet[0],
depois vet[1] e assim por diante
  print("(%2d, %2d) " % (item, vet[i]), end="");
  i += 1;
print(); # quebre a linha
```

3. Defininido uma *cadeia de caracteres* ("string")

Em *Python* é possível definir um vetor com caracteres, formando uma palavra, ou seja, uma *cadeia de caracteres*, que abreviadamente é denominada "*string*". Para isso pode-se fazer uma atribuição como constante ou ler os dados como palavra. Isso é ilustrado no exemplo abaixo.

Exemplo 2. Trabalhando com "strings". Uma "string" fixa e digitar uma frase, imprimindo o número de caracteres nela.

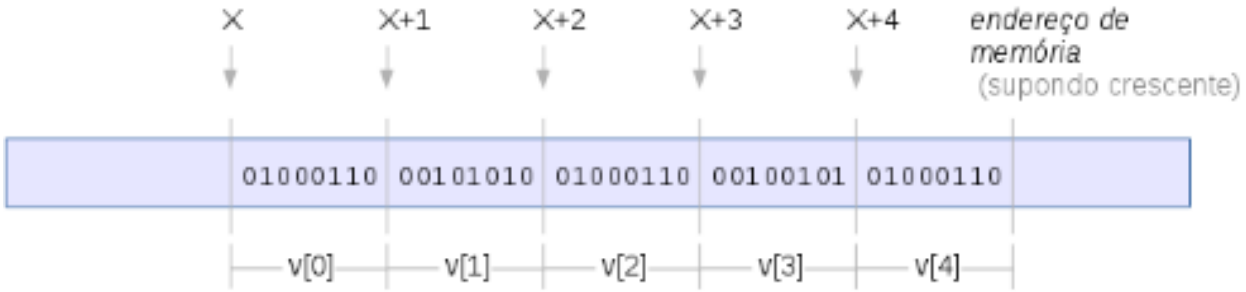
```
# Definir "string"
string = "0123456789abcdefghih"; # (1)
# Imprimir cada caractere da string com seu codigo ASCII
print("\nImprime caracteres e seus codigos ASCII na forma de tabela\nColuna 1
=> caractere; coluna 2 => seu codigo ASCII");
for char in string : # iterador
   print("%20c : %3d" % (char, ord(char))); # funcao ord(.) devolve codigo
ASCII do caractere parametro
print(); # quebre a linha

# Agora um entrada de dados via teclado e pegando uma frase como "string" (o
ENTER finaliza a frase)
print("Digite uma frase e tecle ENTER/CR");
string = input(); # se usar Python 2 precisa trocar 'input' por 'raw_input'
print("Foi digitada uma frase com %d caracteres: %s" % (len(string),
string));

# Observacao: se definiu um vetor como "string" como em (1) acima, NAO e'
possivel alterar a "string", por exemplo,
# o comando abaixo resulta erro "TypeError: 'str' object does not support
item assignment"
# string[2] = '*'; # isso resultaria erro! experimente
```

4. Sobre copiar ou referenciar um vetor/lista

Ao declarar uma *lista/tupla*, significa que associamos um nome de variável à uma sequência de posições de memória, sendo que cada item pode ter um tamanho distinto dos demais, como em `l = [1,[2],"tres"];`. Uma diferença essencial entre *lista* (como `l = [1,2,3];`) e *tupla* (como `l = (1,2,3);`) é que uma *lista* é *mutável* (*que pode ser alterada*) e *tupla* é *imutável* (impossível de alterar algum elemento).



Por exemplo, ao declarar a *tupla* com `v = (1,2,3);`, além do espaço com informações sobre a *tupla*, são reservadas 3 posições de memória, uma para cada elemento da *tupla*. Deve-se destacar que usar um comando do tipo `aux = v;`, seja *v* uma lista ou uma *tupla*, *não implica em copiar v*, mas sim ter uma nova referência para *v*. Assim, no caso de *lista*, que portanto pode ser alterada, ao usar os comandos

```
v = [1,2,3]; aux = v; aux[2] = -1; print(v);
```

será impresso na tela: `[1, 2, -1]`, ou seja, o elemento da posição 2 de *v* foi alterado para *-1*.

Mas existe uma função em uma biblioteca *Python* que copia listas genéricas (`copy`). No exemplo a seguir ilustramos como fazer copia de *listas* ou *tuplas*.

Exemplo 3. Construindo um vetor e imprimindo seus elementos via um apontador.

```
import copy # para funcao 'copy(.)'
def copiar (lst_tpl) : # funcao para copiar elementos de lista/tupla gerando
lista nova
  lista = [];
  for item in lst_tpl : lista.append(item);
  return lista;

lista = [1,2,[3]]; # define lista com 3 elementos, sendo o terceiro uma lista
com 1 elemento
aux = lista;
aux[2] = -1; # alterou lista[2]
print("Lista: lista = %s" % lista);

lista = [1,2,[3]]; # lista
print("Lista: lista = %s" % lista);
aux1 = copy.copy(lista); # copiar com funcao "copy(.)" de biblioteca "copy"
aux2 = list(lista);      # copiar com gerar de lista "list(.)"
aux3 = copiar(lista);    # copiar com funcao local "copiar(.)"

aux1[2] = -1; # NAO alterou lista[2]
aux2[2] = -1; # NAO alterou lista[2]
aux3[2] = -1; # NAO alterou lista[2]
print("aux1=%s, lista=%s" % (aux1, lista));
print("aux2=%s, lista=%s" % (aux2, lista));
```

```
print("aux3=%s, lista=%s" % (aux3, lista));

tupla = [1,2,[3]]; # tupla
print("Tupla: tupla = %s" % tupla);
aux1 = copy.copy(tupla); # usar funcao 'copy' de biblioteca 'copy'
aux2 = list(tupla);
aux3 = copiar(tupla);
print("aux1=%s, tupla=%s" % (aux1, tupla));
print("aux2=%s, tupla=%s" % (aux2, tupla));
print("aux3=%s, tupla=%s" % (aux3, tupla));
```

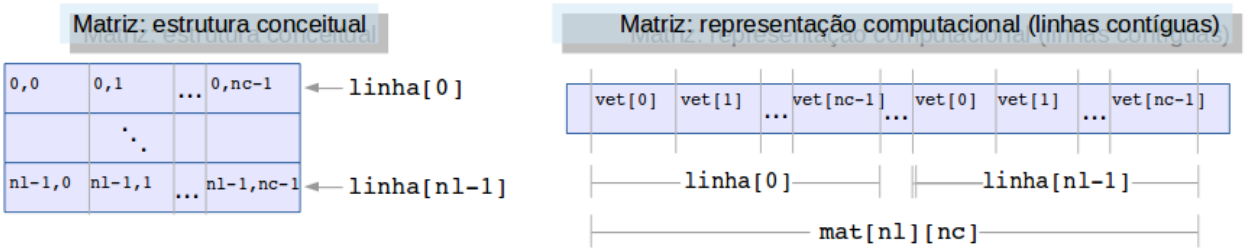
Experimente! Esse código produz como saída as seguintes linhas:

Saídas do exemplo 3.

```
Lista: lista = [1, 2, -1]
Lista: lista = [1, 2, [3]]
aux1=[1, 2, -1], lista=[1, 2, [3]]
aux2=[1, 2, -1], lista=[1, 2, [3]]
aux3=[1, 2, -1], lista=[1, 2, [3]]
Tupla: tupla = [1, 2, [3]]
aux1=[1, 2, [3]], tupla=[1, 2, [3]]
aux2=[1, 2, [3]], tupla=[1, 2, [3]]
aux3=[1, 2, [3]], tupla=[1, 2, [3]]
```

5. Defininido matriz como lista de lista

Uma vez que um vetor/lista é armazenado consecutivamente na memória, então podemos armazenar vários vetores também consecutivamente para obter algo que represente (e seja tratado como) *matriz*. Como na imagem abaixo, em que cada alocamos *nl* vetores consecutivos na memória, cada vetor com *nc* posições de memória, desde `vet[0]` até `vet[nc-1]`. Assim, podemos ver este agregado de dados como uma matriz `mat[][]`, cuja primeira linha é o primeiro vetor e assim por diante. Isso está ilustrado na imagem abaixo.



Desse modo obtemos uma estrutura com dois índices que pode ser manipulada como uma matriz (como em *mat[i][j]*). No exemplo abaixo ilustramos isso de dois modos, imprimindo as linhas da matriz e mostrando que pode-se passar como parâmetro uma linha de matriz para uma função que tenha como *parâmetro formal* um vetor.

Exemplo 4. Construindo matriz como vetor de vetor e ilustrando que cada linha dela é um vetor, podendo portanto ser usado para chamar uma função que tem como parâmetro um vetor.

```
# Vetor de vetor ou matriz
# Funcao para somar elementos de um vetor e devolver essa soma
def somalinha (vet) :
  tam = len(vet); soma = 0;
```

```
  for i in range(tam) : soma += vet[i];
  return soma;

# Definir vetor/lista com: (0,1,2,3,...n-1)
nl = 3; nc = 4;
mat = []; # define um vetor vazio
for i in range(0, nl, 1) : # ou mais simples neste caso "for i in range(n)"
  linha = []; # define um vetor vazio
  for j in range(0, nc, 1) : # ou mais simples neste caso "for i in range(n)"
    linha.append(i*nc+j);
  mat.append(linha); # anexe nova linha `a matriz 'mat'
# Imprime o vetor/lista de uma so' vez (na mesma linha)
print("\nA matriz: ", end=""); # informe o que vira impresso a seguir (na
mesma linha devido ao parametro end=""
print(mat);

# Pegando o primeiro e segundo elemento da terceira linha de mat: mat[2][1]
prim = mat[2][0]; seg = mat[2][1]; # pegar primeiro e segundo elemento da
linha mat[2]
print("O primeiro e segundo elemento da terceira linha da matriz: %d e %d" %
(prim, seg));

print("Linhas da matriz: 0=%s ; 1=%s ; 2=%s" % (mat[0], mat[1], mat[2])); #
cada linha e' um vetor!
# print(mat[0]); print(mat[1]); print(mat[2]);

# Verifique que cada linha da matriz e' de fato um vetor
for i in range(0, nl) :
  print("Soma da linha %2d = %3d" % (i, somalinha(mat[i])));
```

6. Cuidados com referências (ou "apelidos")

Vocês devem tomar muito cuidado com atribuições envolvendo listas, pois na verdade **não** ocorre uma cópia, mas comentado na seção 4. Por exemplo, estude e teste em seu computador a sequência de comandos a seguir.

```
>>> a = [1, 2, 3];
>>> b = a;
>>> b[2] = -9;
>>> print(a)
[1, 2, -9]
```

Cód. 1. Exemplo de uso de atribuição com listas, equivalente a referência, não cópia.

Entretanto o Python oferece um mecanismo que efetivamente realiza uma cópia de cada um dos elementos, portanto envolve um laço de tamanho igual ao número de elementos da lista. Isso é feito usando algo como `[inicio:fim]`, sendo inicio a posição iniciar a partir da qual deseja copiar e fim o último elemento.

Atenção ao fim, não é posição, mas sim número do elemento, então se fim=3, irá copia até o terceiro elemento. Por exemplo, algo como `b = a[4:3];` resultaria em lista vazia, pois o terceiro elemento está antes do elemento da posição 4.

```
>>> a = [0, "um", 2, 3, 4, 5, 6, 7, 8];
>>> v1 = a[4:3]; v2 = a[0:3]; print(v1, v2);
([], [0, 'um', 2])
```

Cód. 2. Exemplo truque para cópiar vários elementos.
Leônidas de Oliveira Brandão
http://line.ime.usp.br

Alterações 

***Alterações**:*
2021/06/27: acertos número do exemplo 3 (para 4) e no seu código; acertos de cedilha; outras pequenas correções