

Python: Listas

Rodrigo D. Malara
(adaptado de Claudio Esperança)

Estruturas de dados

- Maneira de organizar dados de maneira a facilitar seu acesso
- Algumas formas são clássicas:
 - Listas
 - Arrays (vetores e matrizes)
 - Tuplas (registros)
 - Árvores
- Linguagens freqüentemente possuem primitivas para construção dessas E.D.
 - Estruturas de dados *embutidas*
- Outras E.D. mais complexas podem ser construídas combinando as E.D. clássicas

Estrutura de dados abstrata

- É uma especificação matemática que define uma coleção de dados e uma série de operações sobre ela
- É *abstrata* porque não especifica como as operações são feitas mas somente os dados de entrada e o resultado
- Numa linguagem de programação, essa coleção de operações é chamada de *interface* ou API (*Application Programming Interface*)
- Usuários da e.d.a devem se preocupar com a *interface* e não com a implementação, que pode mudar com o tempo
- A *implementação* de uma e.d.a. requer cuidados quanto à correção e a eficiência da mesma

Listas

- São arranjos seqüenciais de informações mais simples
- Caracterizam-se por permitir o acesso eficiente aos seus elementos em ordem seqüencial
- A definição clássica de uma lista como estrutura de dados abstrata compreende:
 - Operação de construção de uma lista vazia
 - Operação que testa se uma dada lista é vazia
 - Operação para obter o primeiro elemento de uma lista
 - Uma operação para adicionar um novo elemento no início de uma lista
 - Operação para retirar o elemento inicial de uma lista

Listas em Python

- A estrutura conhecida como *lista* (*list*, em inglês) em Python é bastante mais geral do que e.d.a. *lista* clássica
- Na verdade, pode ser vista como uma implementação tanto de *listas* como de *arrays*
 - Além de acesso seqüencial, suportam também acesso direto através de índices
- Listas são variedades de seqüências assim como strings e portanto têm APIs semelhantes
 - Podem ser indexadas e fatiadas
 - Podem ser concatenadas (+) e repetidas

Listas em Python

- Entretanto, há diferenças importantes entre listas e strings
 - Seqüência genérica X de seqüência de caracteres
 - Elementos de listas podem ser alterados individualmente mas os de strings, não
- Listas constituem o tipo de agregação de dados mais versátil e comum da linguagem Python
 - Podem ser usadas para implementar estruturas de dados mais complexas como matrizes e árvores, por exemplo

Listas: constantes e índices

- Uma constante do tipo lista é escrita entre colchetes com os elementos separados por vírgula:
 - `[]` # lista vazia
 - `[1,2]` # lista com 2 elementos
- Os elementos de uma lista podem ser de qualquer tipo, inclusive listas. Ex.:
 - `lista = [1, 'a', 2+3j, ['ab', 'CD']]`
- Os elementos de uma lista podem ser acessados por índices como strings
 - O primeiro elemento tem índice 0
 - O último elemento pode ser acessado pelo índice -1

Listas: constantes e índices

```
>>> lista = [1, 'a', 2+3j, ['ab', 'CD']]
```

```
>>> lista [0]
```

```
1
```

```
>>> lista [2]
```

```
(2+3j)
```

```
>>> lista [3]
```

```
['ab', 'CD']
```

```
>>> lista [-1]
```

```
['ab', 'CD']
```

```
>>> lista [0] = 2
```

```
>>> lista
```

```
[2, 'a', (2+3j), ['ab', 'CD']]
```


Listas: Concatenação e Repetição

- O operador + pode ser usado para concatenação e o operador * para repetição

```
>>> lista = [0]*4
```

```
>>> lista
```

```
[0, 0, 0, 0]
```

```
>>> lista = lista + [1]*3
```

```
>>> lista
```

```
[0, 0, 0, 0, 1, 1, 1]
```

Deletando elementos

- O operador *del* pode ser usado para remover elementos de uma lista

■ Ex.:

```
>>> lista  
[1, 2, 3, ['ab', 'CD']]  
>>> del lista [2]  
>>> lista  
[1, 2, ['ab', 'CD']]  
>>> del lista [2][1]  
>>> lista  
[1, 2, ['ab']]
```

Listas: fatias (slices)

- A notação de fatias também pode ser usada, inclusive para atribuição:

```
>>> lista = [1, 'a', 2+3j, ['ab', 'CD']]
```

```
>>> lista [1:]
```

```
['a', (2+3j), ['ab', 'CD']]
```

```
>>> lista [:1]
```

```
[1]
```

```
>>> lista [1:2]
```

```
['a']
```

```
>>> lista [0:-1]
```

```
[1, 'a', (2+3j)]
```

Listas: atribuição a fatias

- A atribuição a uma fatia requer que o valor atribuído seja uma seqüência (uma lista ou uma string, por exemplo)
- A atribuição substitui os elementos da fatia pelos da seqüência

```
>>> lista = [1, 'y', ['ab', 'CD']]
```

```
>>> lista [1:1] = ['z']
```

```
>>> lista
```

```
[1, 'z', 'y', ['ab', 'CD']]
```

```
>>> lista [1:3] = [['x']]
```

```
>>> lista
```

```
[1, ['x'], ['ab', 'CD']]
```

```
>>> lista [1:-1] = [2,3,4]
```

```
>>> lista
```

```
[1, 2, 3, 4, ['x']]
```

```
>>> lista [:2] = 'xyz'
```

```
>>> lista
```

```
['x', 'y', 'z', 3, 4, ['ab', 'CD']]
```

Incrementos em Fatias

- É possível usar um terceiro número na notação de fatias designando o incremento
 - Default é 1 , ou seja, toma os elementos de um em um do menor para o maior índice
 - Pode-se usar qualquer número inteiro diferente de 0
 - `a[0:10:2]` retorna uma lista com os 10 primeiros elementos de `a` tomados de 2 em 2 (5 elementos, no máximo)
 - `a[5:0:-1]` retorna uma lista com os 5 primeiros elementos de `a` tomados da esquerda para a direita
- Obs.: Esta notação só existe nas versões de Python a partir da 2.3

Incrementos em Fatias

■ Exemplo

```
>>> a = ['a', 2, 3, 'd', 'x']
```

```
>>> a[:3:2]
```

```
['a', 3]
```

```
>>> a[::-1]
```

```
['x', 'd', 3, 2, 'a']
```

Incrementos em Fatias

- Se um incremento de fatia é diferente de 1, uma atribuição à fatia deve ter o mesmo número de elementos:

```
>>> l = [1,2,3,4,5]
>>> l [0::2] = ['x','y','z']
>>> l
['x', 2, 'y', 4, 'z']
>>> l [0::2] = [6,7]
```

Traceback (most recent call last):

File "<pyshell#17>", line 1, in -toplevel-

l [0::2] = [6,7]

ValueError: attempt to assign sequence of size 2 to extended slice of size 3

Operador “in”

- Permite saber se um elemento pertence a uma lista
- Serve também para strings

■ Ex.:

```
>>> lista = [1, 'a', 'bc']
```

```
>>> 1 in lista
```

```
True
```

```
>>> 2 in lista
```

```
False
```

```
>>> 'b' in lista
```

```
False
```

```
>>> 'b' in lista[2]
```

```
True
```

```
>>> 'bc' in 'abcd'
```

```
True
```


Inicializando listas

- Não é possível atribuir a uma posição inexistente de uma lista

```
>>> vetor = []
```

```
>>> vetor[0] = 1
```

Traceback (most recent call last):

File "<pyshell#21>", line 1, in -toplevel-

vetor[0] = 1

IndexError: list assignment index out of range

- Se uma lista vai ser usada como um array, isto é, vai conter um número predeterminado de elementos, é conveniente iniciá-la

```
>>> vetor = [0]*10
```

```
>>> vetor[0] = 3
```

```
>>> vetor
```

```
[3, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

Usando *None*

- No uso de estruturas de dados, às vezes é importante preencher uma posição com um valor “não válido”
- A melhor opção para esse uso é empregar o valor especial `None`
 - Não faz parte de tipo nenhum
 - É melhor que usar 0, `[]` ou uma string vazia
- Útil para criar uma lista “vazia” mas com um número conhecido de posições. Ex.:

```
>>> lista = [None]*5
>>> lista
[None, None, None, None, None]
```

Len, min e max

- `len (lista)` retorna o número de elementos de *lista*
- `min (lista)` e `max (lista)` retornam o menor/maior elemento de *lista*

■ Ex.:

```
>>> lista = [1, 2, 9, 3, 4]
```

```
>>> min (lista)
```

```
1
```

```
>>> len (lista)
```

```
5
```

```
>>> max (lista)
```

```
9
```

```
>>> max (['a', 'b', 'c'])
```

```
'c'
```

min e max

- Na verdade, min e max podem ser usados também com vários argumentos ao invés de uma lista

■ Ex.:

```
>>> min (1,2,3,4)
```

```
1
```

```
>>> max (3,4,5)
```

```
5
```

```
>>> max ([],[1],['a'])
```

```
['a']
```

A função *list*

- Pode ser usada para converter uma string numa lista
- É útil pois uma lista pode ser modificada, mas uma string, não
- Para fazer a transformação inversa, pode-se usar o *método* join (veremos métodos mais tarde)

■ Ex.:

```
■ >>> lista = list('alo')
```

```
■ >>> lista
```

```
■ ['a', 'l', 'o']
```

```
■ >>> lista[1]='xx'
```

```
■ >>> lista
```

```
■ ['a', 'xx', 'o']
```

```
■ >>> ''.join(lista)
```

```
■ 'axxo'
```

A função *range*

- Retorna uma progressão aritmética de inteiros numa lista
- Forma geral: `range(início, parada, incremento)`
 - *início* (opcional) é o primeiro valor a ser gerado (default: 0)
 - *parada* é o limite da progressão: a progressão termina no último valor antes de parada
 - *incremento* (opcional) é o passo da progressão (default:1)
- Ex.:

```
>>> list(range(3))  
[0, 1, 2]  
>>> list(range(2,5,2))  
[2, 4]  
>>> list(range(5,2,-2))  
[5, 3]
```

Comando *for*

- Permite iterar sobre os elementos de uma lista
- Forma geral: `for var in lista : comandos`
 - Os *comandos* são repetidos para cada valor de *lista*
 - Durante a repetição, *var* possui o valor corrente da *lista*
- Uma grande utilidade da função `range` é construir a lista de iteração
- Ex.:

```
>>>for i in range(1,7): print (i)
```

1 2 3 4 5 6

Comparando listas

- Listas são comparadas lexicograficamente
 - Se duas listas são iguais até os k -ésimos elementos, o resultado da comparação depende da comparação entre os $(k+1)$ -ésimos elementos
 - Se alguma das listas tem somente k elementos, então esta é a menor
 - Duas listas são iguais se e somente se têm o mesmo comprimento e todos os elementos de mesma posição são iguais
- Uma lista é maior que um número mas menor que uma string
 - Não me pergunte por quê!

Comparando listas

```
>>> [1,2] < [2, 3]
```

```
True
```

```
>>> [1,2] < [1, 2, 3]
```

```
True
```

```
>>> [1,2] != [1,2]
```

```
False
```

```
>>> min([[1],[2,3],[3,4],[]])
```

```
[]
```

```
>>> max([[1],[2,3],[3,4],[]])
```

```
[3, 4]
```

```
>>> min(0,[],"")
```

```
0
```

```
>>> max(0,[],"")
```

```
"
```

Variáveis do tipo *list*

- Uma variável do tipo lista na verdade *contém uma referência* para um valor do tipo lista
 - Atribuir uma variável a outra, cria uma nova referência mas não uma nova lista
 - Para se criar um novo valor, pode-se usar uma expressão que retorne o valor desejado
 - Para saber se duas variáveis se referem ao mesmo valor pode-se usar o operador **is**

Variáveis do tipo *list*

```
>>> a = b = [1,2,3]
```

```
>>> c = a
```

```
>>> d = c[:]
```

```
>>> a is b
```

```
True
```

```
>>> c is b
```

```
True
```

```
>>> d is c
```

```
False
```

```
>>> a[1]=5
```

```
>>> b
```

```
[1, 5, 3]
```

```
>>> d
```

```
[1, 2, 3]
```

A Classe *list*

- Uma lista é na verdade um *objeto* de uma *classe* chamada `list`
 - Não vimos ainda programação OO, mas alguns pontos devem ser enfatizados
- Listas possuem *métodos* que podem ser aplicados a elas
 - Um método é semelhante a uma função, mas são invocados de forma diferente: `objeto.método(args)`
 - Ex.: `lista.reverse()` inverte a ordem dos elementos da lista
 - Para saber todos os métodos de listas, escreva `help(list)`

Alguns métodos da classe *list*

■ `append(elemento)`

- Acrescenta o elemento no fim da lista
- Observe que a operação *altera* a lista, e não simplesmente retorna uma lista modificada

■ Ex.:

```
>>> lista = [1,2]
>>> lista.append(3)
>>> lista
[1, 2, 3]
>>> lista.append([4,5])
>>> lista
[1, 2, 3, [4, 5]]
```

Alguns métodos da classe *list*

■ `count(elemento)`

- Retorna quantas vezes o elemento aparece na lista

- Ex.:

```
>>> [1,2,3,1,2,3,4].count(1)  
2
```

■ `extend(lista2)`

- Acrescenta os elementos de *lista2* ao final da lista

- OBS.: *Altera* a lista ao invés de *retornar* a lista alterada

- Ex.:

```
>>> lista=[1,2]  
>>> lista.extend([3,4])  
>>> lista  
[1, 2, 3, 4]
```

Alguns métodos da classe *list*

■ `count(elemento)`

■ Retorna quantas vezes o elemento aparece na lista

■ Ex.:

```
>>> [1,2,3,1,2,3,4].count(1)  
2
```

■ `extend(lista2)`

■ Acrescenta os elementos de *lista2* ao final da lista

■ OBS.: *Altera* a lista ao invés de *retornar* a lista alterada

■ Ex.:

```
>>> lista=[1,2]  
>>> lista.extend([3,4])  
>>> lista  
[1, 2, 3, 4]
```

Alguns métodos da classe *list*

■ `index(elemento)`

- Retorna o índice da primeira ocorrência de *elemento* na lista

- Um erro ocorre se *elemento* não consta da lista

- Ex.:

```
>>> lista = [9,8,33,12]
```

```
>>> lista.index(33)
```

```
2
```

```
>>> lista.index(7)
```

Traceback (most recent call last):

File "<pyshell#3>", line 1, in -toplevel-

lista.index(7)

ValueError: list.index(x): x not in list

Alguns métodos da classe *list*

■ `insert(indice, elemento)`

■ insere *elemento* na lista na posição indicada por *índice*

■ Ex.:

```
>>> lista = [0,1,2,3]
>>> lista.insert(1,'dois')
>>> lista
[0, 'dois', 1, 2, 3]
```

■ Como o `extend`, *altera* a lista ao invés de *retornar* a lista

■ O valor retornado é `None`!

■ Atribuições a fatias servem para a mesma finalidade mas são menos legíveis

```
>>> lista = [0,1,2,3]
>>> lista [1:1] = ['dois']
>>> lista
[0, 'dois', 1, 2, 3]
```

Alguns métodos da classe *list*

■ pop(*índice*)

- Remove da lista o elemento na posição *índice* e o retorna
- Se *índice* não for mencionado, é assumido o último

■ Ex.:

```
>>> lista = [1,2,3,4]
```

```
>>> lista.pop()
```

```
4
```

```
>>> lista
```

```
[1, 2, 3]
```

```
>>> lista.pop(1)
```

```
2
```

```
>>> lista
```

```
[1, 3]
```

Alguns métodos da classe *list*

■ `remove(elemento)`

- Remove da lista o primeiro elemento igual a *elemento*

- Se não existe tal elemento, um erro é gerado

- Ex.:

```
>>> lista = ['oi', 'alo', 'ola']
```

```
>>> lista.remove('alo')
```

```
>>> lista
```

```
['oi', 'ola']
```

```
>>> lista.remove('oba')
```

Traceback (most recent call last):

File "<pyshell#24>", line 1, in -toplevel-

lista.remove('oba')

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

Alguns métodos da classe *list*

■ reverse()

■ Inverte a ordem dos elementos da lista

■ Ex.:

```
>>> lista=[1,2,3]
```

```
>>> lista.reverse()
```

```
>>> lista
```

```
[3, 2, 1]
```

Alguns métodos da classe *list*

- `sort(cmp=None, key=None, reverse=False)`

- Ordena a lista

- Os argumentos são opcionais. Por default, a lista é ordenada crescentemente

- Ex.:

- `>>> lista = [9,8,7,1,4,2]`

- `>>> lista.sort()`

- `>>> lista`

- `[1, 2, 4, 7, 8, 9]`

Alguns métodos da classe *list*

- `sort(cmp=None, key=None, reverse=False)`

- É possível obter a ordem inversa, passando *True* para o argumento *reverse*

- Ex.:

```
>>> lista = [9,8,7,1,4,2]
>>> lista.sort(reverse=True)
>>> lista
[9, 8, 7, 4, 2, 1]
```

- OBS.: A notação acima permite passar um argumento sem especificar os anteriores, mas poderíamos ter escrito:

```
>>> lista = [9,8,7,1,4,2]
>>> lista.sort(None,None,True)
>>> lista
[9, 8, 7, 4, 2, 1]
```

Alguns métodos da classe *list*

■ `sort(cmp=None, key=None, reverse=False)`

■ O argumento *cmp* especifica uma função de comparação

■ É uma função que o sort chama para definir se um elemento é anterior ou posterior a outro

■ A função a ser passada tem a forma *comp(elem1,elem2)* e deve retornar um inteiro negativo caso *elem1* seja anterior a *elem2*, positivo caso *elem2* seja anterior a *elem1* e zero se tanto faz

■ Ex.:

```
>>> def compara(elem1,elem2):  
    return elem1%10 - elem2%10  
>>> compara(100,22)  
-2  
>>> lista=[100,22,303,104]  
>>> lista.sort(compara)  
>>> lista  
[100, 22, 303, 104]
```

Alguns métodos da classe *list*

- `sort(cmp=None, key=None, reverse=False)`

- O argumento *key* especifica uma função aplicada a cada elemento

- Se for passada uma função f , em vez de ordenar os elementos baseado em seus valores v , ordena baseado em $f(v)$

- Ex.:

```
>>> lista = ['abc','de','fghi']  
>>> lista.sort(key=len)  
>>> lista  
['de', 'abc', 'fghi']
```


Matrizes

- Listas podem ser usadas para guardar matrizes
- Por exemplo, podemos criar uma matriz-identidade de 3x3 com o código:

```
m = []  
for i in range(3):  
    m.append([0]*3)  
    m[i][i]=1
```

- Obs.: Não é boa idéia iniciar uma matriz assim:

```
m = [[0]*3]*3  
for i in range(3): m[i][i]=1  
print m
```

■ Resultado: `[[1, 1, 1], [1, 1, 1], [1, 1, 1]]`

■ Por quê? (Na realidade foram criadas referências)

Exercícios

- Escreva um programa que intercale os elementos de duas listas l1 e l2
 - Exemplo: para l1 = [1,2,3] e l2 = ['a','b','c','d','e'], o programa deve computar a lista [1,'a',2,'b',3,'c','d','e']
- Escreva um programa para computar o produto de duas matrizes m1 e m2