

Tipos Estruturados Sequenciais em Python Objetivos

Strings

Strings são comuns em todas as linguagens de programação e são utilizados para armazenar cadeias de caracteres. Nos capítulos anteriores, os strings já foram utilizados, e o objetivo aqui é formalizar o conceito, bem como apresentar aspectos que ainda não foram vistos.

Um string em Python é uma sequência composta por quaisquer caracteres delimitada por aspas simples ou duplas. No Exemplo 4.1 são definidos os strings S e D, cada um utilizando um tipo diferente de aspas. Ao utilizar o comando *type* com essas variáveis, verifica-se que ambas são da classe “str”, ou seja, string.

A função *len* permite descobrir o tamanho do string, pois retorna a quantidade de caracteres de seu conteúdo.

Exemplo 4.1 Primeiros usos de strings

```
>>> S = 'Cadeia de texto definido com aspas simples'
```

```
>>> D = "Cadeia de texto definido com aspas duplas"
```

```
>>> print(S)
```

```
Cadeia de texto definido com aspas simples
```

```
>>> print(D)
```

```
Cadeia de texto definido com aspas duplas
```

```
>>> type(D)
```

```
<class 'str'>
```

```
>>> type(S)
```



```
<class 'str'>
```

```
>>> len(S) 42
>>> S[0] # Primeiro elemento do string S 'C'
>>> S[1] # Segundo elemento do string S 'a'
>>> S[41] # Último elemento do string S 'S'
>>> S[42] # Elemento inexistente no string S. Gera erro.
```

Traceback (most recent call last):

```
File "<pyshell#7>", line 1, in <module> print(S[42])
IndexError: string index out of range
```

```
>>>
```

Como também pode ser visto nesse exemplo, o uso de um índice entre colchetes que permite acesso individual aos caracteres que o compõem. Uma vez que o índice do primeiro caractere é zero, o índice do último será $len - 1$. No caso do string S do exemplo S[41] é o último caractere, visto que tem dimensão 42. Caso seja utilizado um índice além do limite, o interpretador gera uma mensagem de erro.

Todos os tipos sequenciais em Python também aceitam indexadores negativos, os quais são interpretados como contagem da direita para a esquerda, em que o índice -1 é o do último elemento, -2 do penúltimo, e assim sucessivamente, conforme mostrado no Exemplo 4.2.

Exemplo 4.2 Uso de indexação negativa em tipos sequenciais

```
>>> X = 'ABCD'

>>> X[-1] 'D'
>>> X[-2]
'C'
```



```
>>> X[-3] 'B'
>>> X[-4] 'A'
>>>
```

Manipulação de strings

4.1.1.1 Atribuição de valor aos strings

A qualquer momento é possível alterar o conteúdo de um objeto do tipo string. Porém, não é possível alterar individualmente um caractere que o compõe. Isso ocorre porque em Python os strings são objetos imutáveis. Isso significa que cada operação de atribuição de valor a um string, na verdade, produz um novo objeto. Isso pode ser comprovado com o uso da função *id*, vista no Capítulo 2. A cada atribuição de valor o *id* do objeto se altera, indicando ser um outro objeto.

Observe e repita as operações feitas no Exemplo 4.3. Percebe-se que, ao tentar atribuir um caractere ao elemento `V[0]`, ocorre uma mensagem de erro indicando a impossibilidade de completar o comando.

Exemplo 4.3 Manipulação de strings

```
>>> V = "" # String vazio

>>> len(V)
0

>>> type(V)

<class 'str'> # De fato V é string, mas está vazio

>>> id(V)

1742944 # id de V

>>> V = 'Novo'
```



```
>>> id(V)
```

```
49492576 # Novo id de V
```

```
>>> V = 'Outro'
```

```
>>> id(V)
```

```
49492320 # mais um
```

```
>>> V[0] = 'P'
```

Traceback (most recent call last):

File "<pyshell#81>", line 1, in <module> V[0] = 'P'

TypeError: 'str' object does not support item assignment

4.1.1.2 Concatenação e multiplicação de strings

O Exemplo 4.4 também mostra que é possível utilizar os operadores concatenação e "+" e multiplicação "*" com objetos string.

A concatenação aplica-se a dois operandos do tipo string e produz a junção dos dois. Por outro lado, se o programador tentar utilizar esse operador misturando string com outro tipo de objeto, ocorrerá um erro, uma vez que tais combinações não são suportadas.

Exemplo 4.4 Concatenação de strings

```
>>> S = 'Festa' # carrega S com algum texto
```

```
>>> T = ' na Vila' # faz o mesmo com T
```

```
>>> U = S + T # o operador '+' está definido em Python
```



```
>>> U # para efetuar a concatenação de strings 'Festa na Vila'
```

```
>>> U = 'Hoje tem ' + U
```

```
>>> U
```

```
'Hoje tem Festa na Vila'
```

```
>>> S = 'Festa' + 1000 # tenta concatenar string com outro tipo Traceback (most recent call last):
```

```
File "<pyshell#115>", line 1, in <module> S = 'Festa' + 1000
```

```
TypeError: must be str, not int
```

O operador multiplicação de string está definido para uso com um operador string e outro numérico inteiro, sem importar a ordem em que ambos aparecem na expressão. Dados um string S e um número N, ao utilizar esse operador com ambos, será gerado um string resultante em que S ocorre N vezes.

Exemplo 4.5 Multiplicação de strings

```
>>> S = "repete."
```

```
>>> T = S * 3
```

```
>>> T
```

```
'repete.repete.repete.'
```

Fatiamento

Fatiamento, ou *slicing*, é um recurso disponível nos tipos sequenciais existentes em Python, strings, listas e tuplas. O fatiamento é utilizado para selecionar partes específicas de um tipo sequencial e trata-se de um recurso mais poderoso do que muitos programadores imaginam. Um pouco desse poder será mostrado no Capítulo 5, no qual serão resolvidos exercícios



utilizando funções recursivas que operam com listas fatiadas.

Seja um string S, o fatiamento utiliza a notação S[início:final] para selecionar o substring de S, que começa na posição dada pelo indexador início e termina na posição dada pelo indexador final – 1.

O Exemplo 4.6 inicializa o string S com as quinze primeiras letras minúsculas do alfabeto. Em seguida, é realizado o fatiamento S[3:10], que seleciona desde o caractere cujo índice é 3 até o caractere cujo índice é 9 (10

– 1). Assim sendo, a parte selecionada será “defghij”, conforme mostrado.

Caso os valores definidos para o par indexador dados por [início:final] sejam incoerentes, o fatiamento retornará um string vazio. Para que esse par seja coerente, é necessário que ocorra: início < final. Assim, S[3:4] seleciona o substring “d”.

O retorno produzido pelo fatiamento também pode ser atribuído a um novo objeto, como foi feito no exemplo com o objeto P.

Os índices de fatiamento também podem ser fornecidos por meio de objetos do tipo número inteiro em substituição aos valores numéricos fixos, conforme exemplificado a seguir com os objetos i e f.

Exemplo 4.6 Fatiamento de strings

```
>>> S = 'abcdefghijklmno' # Define o string S com 15 caracteres

>>> print(S) abcdefghijklmno
>>> len(S) 15
>>> S[3:10] # Substring de S das posições 3 a 9 'defghij'
>>> S[0:5] # Substring de S das posições 0 a 4 'abcde'
>>> P = S[3:10] # Atribui a P o substring de S de 3 a 9
```



```
>>> print(P) defghij
>>> len(P) 7
>>> i = 3

>>> f = 10

>>> S[i:f] # Uso de objetos como índices 'defghij'
>>> i = 0

>>> f = 5

>>> S[i:f]

'abcde'
```

O fatiamento também pode omitir um dos índices da faixa de seleção. Quando isso acontece, a interpretação é feita segundo as opções contidas no Quadro 4.1.

Exemplo 4.7 Fatiamento de strings com omissão de início ou final

```
>>> S = 'abcdefghijklmno' # Define o string S com 15 caracteres

>>> S[:5] # Substring de S das posições 0 a 4
'abcde'

>>> S[5:] # Substring de S das posições 5 ao final 'fghijklmno'

>>>
```

Por fim, o fatiamento pode apresentar um terceiro parâmetro, que é o passo. Para compreender como esse terceiro parâmetro é utilizado, suponha-se que ele tenha o valor P. Assim sendo, o string será dividido em substrings de tamanho P e apenas será selecionado o primeiro caractere de cada subdivisão. No Exemplo 4.8 isso é demonstrado.



Exemplo 4.8 Fatiamento de strings com terceiro parâmetro

```
>>> S = 'abcdefghijklmno' # Define o string S com 15 caracteres
```

```
>>> S[0:15:4] # Do início ao fim seleciona 1 a cada 4 'aeim'
```

```
>>> T = '9pula8pula7pula6pula5'
```

```
>>> T[0:21:5] # Do início ao fim seleciona 1 a cada 5 '98765' # Quando se quer trabalhar com todo o string
```

```
>>> T[::5] # é possível omitir os delimitadores '98765' # e se produz o mesmo resultado
```

Quadro 4.1 Opções de fatiamento de tipos sequenciais.

Forma de fatiamento	Interpretação
$S[\text{ini}:\text{fim}]$	O fatiamento tem início e final, então, seleciona-se o substring desde a posição ini até a posição fim -1.
$S[:\text{fim}]$	Neste caso, foi emitido o índice inicial da faixa. O interpretador assume que deve selecionar o string a partir do primeiro caractere até a posição especificada, ou seja, de 0 a fim -1.
$S[\text{ini}:]$	Nesta caso, foi emitido o índice final da faixa. O interpretador assume que deve selecionar o string a partir da posição especificada até o final dele, ou seja, de ini até o final de S.
$S[\text{ini}:\text{fim}:\text{p}]$	Nesta opção estão colocados três parâmetros para fatiamento. Os dois primeiros definem o início e o final do substring. O terceiro define o passo, ou seja especifica que de cada p caracteres toma-se apenas o primeiro para compor o substring.

Métodos da classe “str”

Em termos simples e iniciais, pode-se dizer que **métodos** são funções



específicas contidas em uma classe de objetos. Em adição aos operadores vistos anteriormente, a classe string tem um conjunto de métodos úteis ao programador. Um desses métodos – o format – já foi visto no Item 2.5, em que foi explicado o comando print, utilizado para exibição de dados em tela.

Não há espaço aqui para explicar em detalhes cada um dos métodos disponíveis, porém, no Exemplo 4.9, serão mostrados os usos de alguns deles.

	Para os exemplos a seguir seja S = "abc_123_XYZ"	
Método	Descrição	Retorno
S.lower	Retorna um string com todas as letras minúsculas e não afeta os demais caracteres.	abc_123_xyz
S.upper	Retorna um string com todas as letras maiúsculas e não afeta os demais caracteres.	ABC_123_XYZ
	Retorna um string invertendo as letras	
S.swapcase	maiúsculas e minúsculas e não afeta os demais caracteres.	ABC_123_xyz
S.title	Retorna um string com a primeira letra maiúscula e as demais minúsculas, para cada sequência de letras.	Abc_123_xyz
S.capitalize	Retorna um string com a primeira letra maiúscula e as demais minúsculas.	Abc_123_xyz
S.find("123")	Pesquisa um substring em S e retorna um número inteiro, indicando a posição se o encontrar, ou -1 caso não encontre.	4
S.find("m")		-1
S.count["_"]	Conta quantas vezes um substring está contido em S.	2



<code>S.replace("_", "**")]</code>	Este método recebe dois parâmetros do tipo string. O primeiro é procurado dentro de S e, caso seja encontrado, é substituído pelo segundo substring.	<code>abc*123*XYZ</code>
<code>S.isalnum</code>	Retorna True caso o string contenha apenas letras e números. Caso contrário, retorna False.	
<code>S.isalpha</code>	Retorna True caso o string contenha apenas letras. Caso contrário, retorna False.	
<code>S.isnumeric</code>	Retorna True caso o string contenha apenas números. Caso contrário, retorna False. Útil para testar a entrada de dados numéricos digitados no teclado.	
<code>S.partition("_")]</code>	Pesquisa S em busca do substring passado e, caso o encontre, retorna três strings: a parte antes do substring, o próprio substring e o resto de S. Caso não o encontre, retorna o próprio S mais dois strings vazios.	
<code>S.split("_"]</code>	Retorna uma lista de strings separados a partir de S, utilizando o substring passado como parâmetro como delimitador da separação. Se ele não for fornecido, o espaço em branco é utilizado como delimitador.	<code>["abc", "123, "XYZ"]</code>

Quadro 4.2 Alguns métodos disponíveis na classe “str”.

Exemplo 4.9 Métodos da classe “str”

```
>>> S = 'abc_123_XYZ'
```

```
>>> S.lower() # retorna todas as letras em minúsculas 'abc_123_xyz'
```

```
>>> S.upper()
```

```
'ABC_123_XYZ'
```

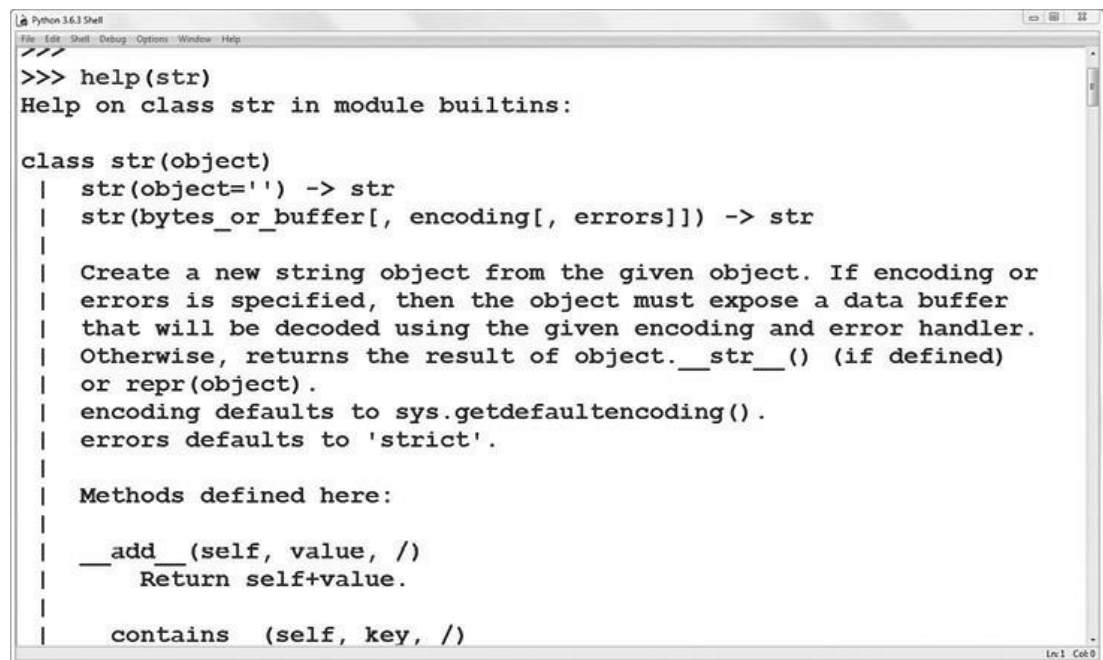


```
>>> S.title() 'Abc_123_Xyz'
>>> S.swapcase() 'ABC_123_xyz'
>>> S.find('123') 4
>>> S.find('m')
```

-1

```
>>> S.count('_') 2
>>> S.replace('_', '*') 'abc*123*XYZ'
>>> S.replace('123', 'xpto') 'abc_xpto_XYZ'
>>> S.partition('_') ('abc', '_', '123_XYZ')
>>> S.partition('*') ('abc_123_XYZ', '', '')
>>> S.split('_') ['abc', '123', 'XYZ']
```

Para obter um resumo de todos os métodos disponíveis na classe string, utilize os comandos **dir** e **help** no IDLE ou consulte a documentação da linguagem.



```
Python 3.6.3 Shell
File Edit Shell Debug Options Window Help
>>> help(str)
Help on class str in module builtins:

class str(object)
| str(object='') -> str
| str(bytes_or_buffer[, encoding[, errors]]) -> str
|
| Create a new string object from the given object. If encoding or
| errors is specified, then the object must expose a data buffer
| that will be decoded using the given encoding and error handler.
| Otherwise, returns the result of object.__str__() (if defined)
| or repr(object).
| encoding defaults to sys.getdefaultencoding().
| errors defaults to 'strict'.
|
| Methods defined here:
|
| __add__(self, value, /)
|     Return self+value.
|
| contains (self, key, /)
```

Exemplo de uso do comando help no IDLE.

Listas

Durante o desenvolvimento de software, independentemente de plataforma e linguagem, é comum a necessidade de criar, manter e manipular conjuntos de dados.



Tais conjuntos são muito variados, tanto quanto à natureza dos dados como com relação às quantidades envolvidas.

Na linguagem Python, o tipo lista é a ferramenta disponível para atender a essa demanda e representa o mais genérico, versátil e poderoso tipo sequencial. Por exemplo, as listas podem ser empregadas para armazenar coleções de números inteiros ou reais, palavras, nomes ou quaisquer outras informações necessárias à solução de algum problema computacional. Muitas vezes, tais conjuntos são muito grandes, e é preciso mantê-los e manipulá-los de maneira segura e eficiente na memória, bem como gravá-los em disco ou enviá-los de um computador para outro em uma rede.

Assim como o string, é um tipo sequencial composto por elementos organizados de modo linear, na qual cada um pode ser acessado a partir de um índice que representa sua posição na coleção, iniciando em zero. Em função disso, a lista suporta muitas das mesmas operações que já foram vistas para o tipo string no Item 4.1. Então, tem-se que as listas apresentam os mesmos mecanismos de indexação e fatiamento, suportam os operadores de concatenação “+” e multiplicação “*” e têm comprimento variável, que pode ser descoberto com o uso da função *len*.

Por outro lado, algumas de suas principais características são completamente opostas às dos strings. Quanto ao conteúdo, os elementos de uma lista podem ser qualquer tipo de objeto. Além disso, as listas são mutáveis, de modo que seus elementos podem ser alterados livremente e a qualquer momento pelo programador.

Operações básicas com listas

O Exemplo 4.10 mostra as operações básicas possíveis de ser efetuadas com as listas. É possível criar uma lista atribuindo-se um conjunto de dados entre colchetes [] a um identificador de objeto. Se não houver dados entre os colchetes, cria-se uma lista vazia.

O acesso individual aos elementos da lista é feito por meio de seu índice e cada elemento é mutável, podendo, portanto, ser alterado. Além disso, esses elementos podem ser de quaisquer tipos, compondo uma lista heterogênea.

Caso se queira excluir um elemento da lista, basta utilizar o comando *del*, passando como parâmetro o elemento a ser excluído.

Exemplo 4.10 Operações básicas com listas



```
>>> L = [] # cria uma lista vazia
>>> type(L) # mostra o tipo do objeto L

<class 'list'>

>>> print(L) # exibe a lista L []
>>> L = [10, 15, 20, 25, 30] # L passa a conter 4 elementos

>>> print(L) # Exibe a lista. No IDLE pode-se [10, 15, 20, 25, 30] # omitir o print
>>> L[0] # primeiro elemento: índice 0 10
>>> L[4] # último elemento: índice 4 30
>>> len(L) # comprimento de L 5
>>> L[0] = 8 # L é mutável

>>> L

[8, 15, 20, 25, 30] # L[0] foi alterado para 8

>>> A = [3, 7.5, 'txt'] # Nova lista elementos heterogêneos
>>> A

[3, 7.5, 'txt']

>>> type(A)

<class 'list'>

>>> type(A[0]) # o primeiro elemento é 'int'

<class 'int'>

>>> type(A[1]) # o segundo elemento é 'float'
```



```
<class 'float'>
```

```
>>> type(A[2]) # o terceiro elemento é 'str'
```

```
<class 'str'>
```

```
>>> del(A[1]) # Exclui o segundo elemento de A
```

```
>>> A # no caso, o valor 7.5 [3, 'txt']
```

No Exemplo 4.11, deve-se ter o cuidado de interpretar o resultado produzido pelo operador de adição "+". Quando os operadores envolvidos forem elementos da lista, a operação será definida em função dos tipos desses elementos. No caso do exemplo, trata-se de tipos numéricos, e o resultado é a adição dos valores neles contidos. Se o mesmo operador for aplicado a duas listas, então, o resultado será uma nova lista concatenando-as, gerando uma terceira.

Exemplo 4.11 Uso do operador aditivo "+" com listas

```
>>> X = L[0] + A[1] # Soma o primeiro elemento de L com
```

```
>>> X # o segundo elemento de A, ou seja,
```

```
15.5 # 8 + 7.5 = 15.5
```

```
>>> X = L + A # Cuidado: aqui é diferente. Como
```

```
>>> X # não foram usados os índices [8, 15, 20, 25, 30, 3, 7.5, 'txt']
```

```
>>> type(X) # o resultado foi a concatenação
```

```
<class 'list'> # das duas listas L e A.
```

O Exemplo 4.12 ilustra o fatiamento de listas, que segue o conceito já visto para



strings, porém, neste caso, produzindo como resultado uma nova lista. Utiliza-se a mesma notação `L[início:final]` para selecionar o sublista de `L` que começa na posição dada pelo indexador início e termina na posição dada pelo indexador final – 1. Também é possível utilizar a notação que inclui o passo: `L[início:final:passo]`, que, dentro do intervalo `[início:final]`, seleciona o primeiro elemento de cada subintervalo dado pelo valor contido no passo.

Exemplo 4.12 Fatiamento de listas

```
>>> L = [1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23]

>>> L[0:3] # elementos de 0 a 2 [1, 3, 5]
>>> L[4:10] # elementos de 4 a 9 [9, 11, 13, 15, 17, 19]
>>> L[:5] # elementos de 0 a 4 [1, 3, 5, 7, 9]
>>> L[5:] # elementos de 5 ao último [11, 13, 15, 17, 19, 21, 23]
>>> L[0:8:3] # elementos de 0 a 7 e [1, 7, 13] # retorna o primeiro a cada 3
>>> L[::4] # considera a lista toda e [1, 9, 17] # retorna o primeiro a cada 4
```

Quando aplicado a listas, o operador multiplicativo “*” necessita de uma lista de origem e um número inteiro e produz uma nova lista com diversas repetições da lista original. Assim, conforme mostrado no Exemplo 4.13, se a lista for `[3, 7]` e o número inteiro for 3, será produzida a lista `[3, 7, 3, 7, 3, 7]`.

Exemplo 4.13 Uso do operador multiplicativo “*” com listas

```
>>> A = [3, 7] * 3

>>> A

[3, 7, 3, 7, 3, 7]

>>> L = [0] * 10
>>> L

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



Outra operação que pode ser útil em muitos casos é a conversão de um string em uma lista. Anteriormente, foi mostrado o uso do método `split()` pertencente à classe “str”, que é capaz de separar um string em elementos. É possível também separar um string fazendo que cada caractere seja um elemento em uma lista resultante, conforme mostrado no Exemplo 4.14.

Exemplo 4.14 Conversão de string em lista

```
>>> S = 'Um texto.'
```

```
>>> L = list(S) # uso da função list para separar
```

```
>>> L # um string
```

```
['U', 'm', ' ', 't', 'e', 'x', 't', 'o', '.']
```

```
>>> S = '5 7 8.8 12' # string com espaços em branco
```

```
>>> L = S.split() # separa S usando espaço em branco
```

```
>>> L # como delimitador ['5', '7', '8.8', '12']
```

```
>>> S = '5;7;8.8;12' # string com o caractere ';'
```

```
>>> L = S.split(';') # separa S usando ';'
```

```
>>> L # como delimitador
```

```
['5', '7', '8.8', '12']
```

```
>>>
```

Operador in

O operador *in* permite ao programador verificar se um valor está presente em



uma lista. Ou, então, pode-se utilizá-lo em conjunto com o operador lógico *not* (*not in*) para verificar o contrário. Em ambos os casos, o resultado produzido é *True* (verdadeiro) ou *False* (falso).

Esse operador não se aplica apenas às listas, pelo contrário, ele pode ser utilizado em todos os tipos estruturados existentes em Python. O Exemplo

4.15 ilustra alguns de seus usos. Mais adiante, será visto que há outras formas de utilizá-lo.

Exemplo 4.15 Operadores *in* e *not in*

```
>>> L = [3, 6, 9]

>>> 9 in L # 9 está em L True
>>> 5 in L # 5 não está em L False
>>> 5 not in L # como 5 não está em L o operador True # not in resulta em True
>>> Caes = ['Labrador', 'Poodle', 'Terrier']
>>> a = 'Collie' # testar se 'Collie' está na lista

>>> if a in Caes:

... print('Boa escolha')

... else:

... print('Não temos essa raça')

Não temos essa raça # este é o resultado do if-else

>>>
```

Métodos da classe “list”

A classe “list” apresenta um conjunto de métodos que podem ser utilizados pelos programadores para executar tarefas típicas envolvendo listas. O Quadro 4.3



apresenta todos esses métodos e os descreve. O Exemplo 4.16 ilustra seu uso.

Para obter um resumo de todos os métodos disponíveis na classe “list”, use os comandos *dir(list)* e *help(list)* ou consulte a documentação da linguagem.

	Considere-se disponível a lista L
Método	Descrição
L.append(object)	Acrescenta um objeto à lista. Exemplo: L.append(5)
L.clear()	Limpa a lista, removendo todos seus elementos. Exemplo: L.clear()
L.copy()	Produz uma cópia da lista L. Exemplo: Nova - L.copy()
L.count(object)	Retorna o número de ocorrências de um objeto dentro da lista Exemplo: Qtde - L.count(5)
L.extend(iterable)	Expande a lista L, acrescentando a ela todos os elementos contidos no objeto iterável passado como parâmetro. Exemplo: L.extend(OutraLista)
L.index(value, [start, stop])	Retorna o índice da primeira ocorrência do valor “value” dentro da lista. Se start e stop (opcionais) forem fornecidos, o método considera apenas seu intervalo. Caso “value” não esteja na lista, é gerado um erro. Exemplo: posição = L.index(5)
L.insert(index, object)	Insere o objeto fornecido na posição dada por “index”, deslocando todos os demais para a direita. Exemplo: L.insert(2, 30)
L.pop(index)	Retorna o elemento que está na posição dada por “index” e o remove da lista. Exemplo: L.pop(0)
L.remove(value)	Remove da lista a primeira ocorrência do valor “value”. Se o valor não estiver na lista, gera um erro. Exemplo: L.remove(5)



L.reverse()	<p>Inverte a posição dos elementos dentro da lista: o primeiro valor passa a ser o último, o segundo passa a penúltimo, e assim por diante. Não retorna nada, pois inverte a própria lista.</p> <p>Exemplo: L.reverse()</p>
L.sort(...)	<p>Ordena a lista, colocando-a em ordem crescente ou decrescente. Não retorna nada, pois ordena a própria lista.</p> <p>Exemplo:</p> <p>L.sort() # ordena em ordem crescente L.sort(reverse=True) # ordena em ordem decrescente</p> <p>Observações:</p> <ol style="list-style-type: none">1. Se desejar preservar a lista L e gerar uma cópia ordenada dela, utilize a função sorted em vez deste método.2. O método sort não funciona com listas heterogêneas que misturem números com strings.

Quadro 4.3 Métodos da classe “list” disponíveis ao programador.

Exemplo 4.16 Métodos da classe “list”

```
>>> L = [3, 6, 9]
```

```
>>> L.append(5) # insere novo objeto no final de L
```

```
>>> L
```

```
[3, 6, 9, 5]
```

```
>>> L.append(2)
```

```
>>> L
```

```
[3, 6, 9, 5, 2]
```

```
>>> L.insert(2, 15) # insere novo objeto na posição 2
```



```
>>> L
```

```
[3, 6, 15, 9, 5, 2]
```

```
>>> L.insert(99, 21) # insere novo objeto na posição 99,
```

```
>>> L # porém a lista não tem tais posições [3, 6, 15, 9, 5, 2, 21] # então insere  
no final
```

```
>>> L.append(6)
```

```
[3, 6, 15, 9, 5, 2, 21, 6]
```

```
>>> L.count(6) # conta as ocorrências do valor 6
```

```
2
```

```
>>> L.count(45) # conta as ocorrências do valor 45 0
```

```
>>> L.index(15) # retorna o índice de 15 2
```

```
>>> L.index(6) # retorna o índice da primeira 1 # ocorrência de 6
```

```
>>> L.index(45) # 45 não está na lista, gera erro Traceback (most recent call  
last):
```

```
File "<pyshell#76>", line 1, in <module> L.index(45)
```

```
ValueError: 45 is not in list
```

```
>>> L.pop(3) # retorna o elemento de índice 3 e o 9 # remove da lista
```

```
>>> L
```

```
[3, 6, 15, 5, 2, 21, 6]
```

```
>>> L.remove(6) # remove a primeira ocorrência de 6
```

```
>>> L
```

```
[3, 15, 5, 2, 21, 6]
```

```
>>> A = [22, 32, 42]
```



```
>>> L.extend(A) # acrescenta a lista A em na lista L
```

```
>>> L
```

```
[3, 15, 5, 2, 21, 6, 22, 32, 42]
```

```
>>> L.reverse() # inverte a lista
```

```
>>> L
```

```
[42, 32, 22, 6, 21, 2, 5, 15, 3]
```

```
>>> L.reverse() # inverte novamente
```

```
>>> L
```

```
[3, 15, 5, 2, 21, 6, 22, 32, 42]
```

```
>>> L.sort() # ordena em ordem crescente
```

```
>>> L
```

```
[2, 3, 5, 6, 15, 21, 22, 32, 42]
```

```
>>> L.sort(reverse=True) # ordena em ordem decrescente
```

```
>>> L
```

```
[42, 32, 22, 21, 15, 6, 5, 3, 2]
```

```
>>> L.clear() # limpa a lista, deixando-a vazia
```

```
>>> L = ['dado', 'uva', 'caixa', 'lata', 'casa']
```



```
>>> L.sort() # ordenação de lista com strings
```

```
>>> L
```

```
['caixa', 'casa', 'dado', 'lata', 'uva']
```

Não é possível usar o método sort com a lista heterogênea abaixo

```
>>> L = [23, 7.7, 3.9, 3, 35, 'txt', '3em1']
```

```
>>> L.sort() # se usar ocorre erro Traceback (most recent call last):
```

```
File "<pyshell#28>", line 1, in <module> L.sort()
```

```
TypeError: '<' not supported between instances of 'str' and 'float'4.2.3
```

