Capítulo 8

Listas

8.1 Uma lista é uma sequência

Como uma string, uma *lista* é uma sequência de valores. Em uma string, os valores são caracteres; em uma lista, eles podem ser de qualquer tipo. Os valores em lista são chamados *elementos* ou às vezes *itens*.

index {element} index {sequence} index {item}

Existem várias maneiras de criar uma nova lista; o mais simples é incluir os elementos entre colchetes ("[" e "]"):

```
[10, 20, 30, 40]
['cururu crocante', 'estômago de carneiro', 'vômito de cotovia']
```

O primeiro exemplo é uma lista de quatro inteiros. O segundo é uma lista de três `strings`. Os elementos de uma lista não precisam ser do mesmo tipo. A lista a seguir contém uma `string`, um `float`, um inteiro e (lo!) outra lista:

```
['spam', 2.0, 5, [10, 20]]
```

Encaixando uma lista dentro de outra.

Uma lista que não contém elementos é chamada de lista vazia; você pode criar uma com colchetes vazios, `[]`.

Como você poderia esperar, você pode atribuir valores de lista a variáveis:

```
>>> queijos = ['Cheddar', 'Edam', 'Gouda']
>>> numbers = [17, 123]
>>> empty = []
>>> print (queijos, números, vazios)
['Cheddar', 'Edam', 'Gouda'] [17, 123] []
```

8.2 Listas são mutáveis

A sintaxe para acessar os elementos de uma lista é a mesma usada para acessar os caracteres de uma `string`, o operador colchete. A expressão dentro dos colchetes especifica o índice. Lembre-se que os índices começam em 0:

```
>>> print(queijos[0])
Cheddar
```

Ao contrário das `strings`, as listas são mutáveis porque você pode alterar a ordem dos itens ou atribuir outro valor a um item. Quando o operador colchete aparece no lado esquerdo de uma atribuição, ele identifica o elemento da a lista que será modificado.

```
>>> numeros = [17, 123]
>>> numeros[1] = 5
>>> print(numeros)
[17, 5]
```

O último elemento de `numeros`, que costumava ser 123, agora é 5.

Você pode pensar em uma lista como uma relação entre índices e elementos. Essa relação é chamada de *mapeamento*; cada index "representa" um dos elementos.

Os índices de lista funcionam da mesma maneira que os índices de uma `string`:

- Qualquer expressão inteira pode ser usada como um index.
- Se você tentar ler ou escrever um elemento que não existe, você terá um "IndexError".

exception!IndexError IndexError

- Se um índice tiver um valor negativo, ele será contado ao contrário iniciando do final da lista.

O operador `in` também funciona em listas.

```
>>> queijos = ['Cheddar', 'Edam', 'Gouda']
>>> 'Edam' in queijos
Verdade
>>> 'Brie' in queijos
Falso
```

8.3 Percorrendo uma lista

A maneira mais comum de percorrer os elementos de uma lista é com um laço de repetição `for`. A sintaxe é a mesma que para strings:

```
for queijo in queijos:
    print (queijo)
```

Isso funciona bem se você precisar ler apenas os elementos da lista. Mas se você quiser escrever ou atualizar os elementos, você precisa dos índices. Uma maneira comum de fazer isso é combinar as funções `range` e `len`:

```
for i in range(len(numeros)):
    numeros[i] = numeros[i] * 2
```

Essa repetição percorre a lista e atualiza cada elemento. `len` retorna o número de elementos na lista. `range` retorna uma lista de índices de 0 a $n - 1$, onde n é o tamanho da lista. Cada vez que passa pela repetição, o `i` recebe o índice do próximo elemento. A declaração de atribuição no corpo usa `i` para ler o valor antigo do elemento e atribui o novo valor.

Uma repetição `for` sobre uma lista vazia nunca executa o corpo:

```
for x in empty:
    print('Isso nunca acontece.')
```

Embora uma lista possa conter outra lista, a lista encaixada ainda conta como um único elemento. O comprimento dessa lista é quatro:

```
['spam', 1, ['Brie', 'Roquefort', 'Pol le Veq'], [1, 2, 3]]
```

8.4 Operações com Listas

O operador `+` concatena listas:

```
>>> a = [1, 2, 3]
>>> b = [4, 5, 6]
>>> c = a + b
>>> print(c)
[1, 2, 3, 4, 5, 6]
```

Similarmente, o operador `*` repete uma lista um dado número de vezes:

```
>>> [0] * 4
[0, 0, 0, 0]
>>> [1, 2, 3] * 3
[1, 2, 3, 1, 2, 3, 1, 2, 3]
```

O primeiro exemplo repete quatro vezes. O segundo exemplo repete a lista três vezes.

8.5 Fatiamento de listas

O operador de fatiamento também funciona em listas:

```
>>> t = ['a', 'b', 'c', 'd', 'e', 'f']
>>> t[1:3]
['b', 'c']
>>> t[:4]
['a', 'b', 'c', 'd']
>>> t[3:]
['d', 'e', 'f']
```

Se você omitir o primeiro índice, a fatia começará do início. Se você omitir o segundo, a fatia vai até o final da lista. Então se você omitir as duas, a fatia é uma copia da lista inteira.

```
>>> t[:]
['a', 'b', 'c', 'd', 'e', 'f']
```

Como as listas são mutáveis, geralmente é útil fazer uma cópia antes de realizar operações que dobram, perfuram ou mutilam listas.

O operador de fatiamento no lado esquerdo de uma atribuição pode atualizar múltiplos elementos:

```
>>> t = ['a', 'b', 'c', 'd', 'e', 'f']
>>> t[1:3] = ['x', 'y']
>>> print(t)
['a', 'x', 'y', 'd', 'e', 'f']
```

8.6 Métodos para listas

Python fornece métodos que operam em listas. Por exemplo, `append` adiciona um novo elemento ao final de uma lista:

```
>>> t = ['a', 'b', 'c']
>>> t.append('d')
>>> print(t)
['a', 'b', 'c', 'd']
```

`extend` leva uma lista como um argumento para o método `append` e adiciona todos os elementos o final de uma lista:

```
>>> t1 = ['a', 'b', 'c']
>>> t2 = ['d', 'e']
>>> t1.extend(t2)
>>> print(t1)
['a', 'b', 'c', 'd', 'e']
```

Este exemplo deixa `t2` sem modificações.

`sort` ordena os elementos da lista do menor para o maior:

```
>>> t = ['d', 'c', 'e', 'b', 'a']
>>> t.sort()
>>> print(t)
['a', 'b', 'c', 'd', 'e']
```

A maioria dos métodos para lista é do tipo `void`; eles modificam a lista e retornam `None`. Se você acidentalmente escrever `t = t.sort()`, você se desapontará com o resultado.

8.7 Apagando elementos

Existem várias maneiras de excluir elementos de uma lista. Se você sabe o índice do elemento que você quer eliminar, você pode usar `pop`:

```
>>> t = ['a', 'b', 'c']
>>> x = t.pop(1)
>>> print(t)
['a', 'c']
>>> print(x)
b
```

`pop` modifica a lista e retorna o elemento que foi removido. Se você não fornecer um índice, ele deleta e retorna o último elemento da lista.

Se você não precisar do valor removido, você pode usar o operador `del`:

```
>>> t = ['a', 'b', 'c']
>>> del t[1]
>>> print(t)
['a', 'c']
```

Se você sabe o elemento que quer remover (mas não o índice), você pode usar `remove`:

```
>>> t = ['a', 'b', 'c']
>>> t.remove('b')
>>> print(t)
['a', 'c']
```

O valor retornado por `remove` é `None`.

Para remover mais de um elemento, você pode usar `del` com um índice de fatia:

```
>>> t = ['a', 'b', 'c', 'd', 'e', 'f']
>>> del t[1:5]
>>> print(t)
['a', 'f']
```

Como sempre, a fatia seleciona todos os elementos, mas não inclui o segundo índice.

8.8 Listas e funções

Há várias funções internas que podem ser usadas em listas que permitem que você examine rapidamente uma lista sem escrever seus próprios laços:

```
>>> nums = [3, 41, 12, 9, 74, 15]
>>> print(len(nums))
6
>>> print(max(nums))
74
>>> print(min(nums))
3
>>> print(sum(nums))
154
>>> print(sum(nums)/len(nums))
25
```

A função `sum()` só funciona quando os elementos da lista são números. As outras funções (`max()`, `len()`,etc.) funcionam com listas de strings e outros tipos semelhantes.

Poderíamos reescrever um programa anterior que calculava a média de uma lista de números inseridos pelo usuário usando uma lista.

Primeiramente, o programa calcula a média sem uma lista:

```
total = 0
count = 0
while (True):
    inp = input('Enter a number: ')
    if inp == 'done': break
    value = float(inp)
    total = total + value
    count = count + 1

average = total / count
print('Average:', average)

# Code: http://www.py4e.com/code3/avenum.py
```

Nesse programa, nós temos as variáveis `count` e `total` para guardar o número e a quantidade total dos números do usuário, conforme solicitamos repetidamente ao usuário por um número.

Nós podemos simplesmente lembrar cada número quando o usuário o inserisse e usar funções internas para calcular a soma e calcular no final.

```
numlist = list()
while (True):
    inp = input('Enter a number: ')
    if inp == 'done': break
```

```
    value = float(inp)
    numlist.append(value)

average = sum(numlist) / len(numlist)
print('Average:', average)

# Code: http://www.py4e.com/code3/avelist.py
```

Fazemos uma lista vazia antes do inicio do laço, e a cada vez que temos um número, o acrescentamos na lista. No final do programa, nós simplesmente calculamos a soma dos números na lista e os dividimos pela contagem dos números na lista para chegar à média.

8.9 Listas e strings

Uma string é uma sequência de caracteres e uma lista é uma sequência de valores, mas uma lista de caracteres não é o mesmo que uma string. Para converter de uma string para uma lista de caracteres, você pode usar `list`:

```
>>> s = 'spam'
>>> t = list(s)
>>> print(t)
['s', 'p', 'a', 'm']
```

Como `list` é o nome de uma função interna, você deve evitar usá-la como nome de uma variável. Eu além disso evito usar "I" porque parece muito com o número "1". Então é por isso que eu uso a letra "t".

A função `list` quebra uma string em letras individuais. Se você quer quebrar uma string em palavras, você pode usar o método `split`:

```
>>> s = 'sentindo falta dos fiordes'
>>> t = s.split()
>>> print(t)
['sentindo', 'falta', 'dos', 'fiordes']
>>> print(t[2])
the
```

Depois de você ter usado `split` para dividir a string em uma lista de palavras, você pode usar o operador de índices (colchete) para ver uma palavra particular da lista. Você pode chamar `split` com um argumento opcional chamado um *delimiter* que especifica qual caractere usar como limite de palavras.

```
>>> s = 'spam-spam-spam'
>>> delimiter = '-'
>>> s.split(delimiter)
['spam', 'spam', 'spam']
```

`join` é o inverso de `split`. Ele pega uma lista de strings e concatena os elementos. `join` é um método de string, então você tem que invocar isso no delimitador e passar a lista como um parâmetro.

```
>>> t = ['sentindo', 'falta', 'dos', 'fiordes']
>>> delimiter = ' '
>>> delimiter.join(t)
'sentindo falta dos fiordes'
```

Nesse caso, o delimitador é um caractere de espaço, então `join` coloca um espaço entre as palavras. Para concatenar strings sem espaços, você pode usar a string vazia, "", como delimitador.

8.10 Linhas aliadas

Normalmente, quando estamos lendo um arquivo, queremos fazer algo com as linhas, além de apenas imprimir a linha inteira. Muitas vezes queremos encontrar as "linhas interessantes" e depois analisá-las para encontrar a parte que seja interessante. E se quiséssemos imprimir o dia da semana a partir das linhas que começam com "De"?

```
De stephen.marquard@uct.ac.za Sáb Jan  5 09:14:16 2008
```

O método `split` é muito eficaz quando confrontado com este tipo de problema. Podemos escrever um pequeno programa que procura linhas que começam com "De", divide essas linhas e depois imprime as três primeiras letras:

```
fhand = open('mbox-short.txt')
for line in fhand:
    line = line.rstrip()
    if not line.startswith('From '): continue
    words = line.split()
    print(words[2])

# Code: http://www.py4e.com/code3/search5.py
```

O programa produz a seguinte saída:

```
Sáb
Sex
Sex
Sex
...
```

Mais tarde, aprenderemos técnicas cada vez mais sofisticadas para escolher as linhas para trabalhar e como separá-las para encontrar a informação exata que estamos procurando.

8.11 Objetos e valores

Se nós executarmos estas atribuições:

```
a = 'banana'
b = 'banana'
```

Sabemos que ambos `a` e `b` se referem a uma string, mas não sabemos se eles se referem à *mesma* string. Existem dois estados possíveis:

Figure 8.1: Variables and Objects

Em um caso, `a` e `b` referem-se a dois objetos diferentes que possuem o mesmo valor. No segundo caso, eles se referem ao mesmo objeto.

Para verificar se duas variáveis se referem ao mesmo objeto, você pode usar o operador `is`.

```
>>> a = 'banana'
>>> b = 'banana'
>>> a is b
True
```

Neste exemplo, o Python criou apenas uma string objeto, e tanto `a` quanto `b` se referem a ele.

Mas quando você cria duas listas, você obtém dois objetos:

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

Nesse caso, diríamos que as duas listas são equivalentes, porque elas têm os mesmos elementos, mas não são idênticas, porque não são o mesmo objeto. Se dois objetos são idênticos, eles também são *equivalentes*, mas se forem equivalentes, eles não são necessariamente *idênticos*.

Até agora, temos usado "objeto" e "valor" de forma intercambiável, mas é mais preciso dizer que um objeto tem um valor. Se você executar `a = [1,2,3]`, onde `a`refere-se a uma lista de objetos cujo valor é uma sequência particular de elementos. Se outra lista tiver os mesmos elementos, diremos que ela tem o mesmo valor.

8.12 Aliados

Se a refere-se a um objeto e você atribui b = a, então ambas as variáveis referem-se ao mesmo objeto:

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

A associação de uma variável a um objeto é chamada de *referência*. Neste exemplo, existem duas referências ao mesmo objeto.

Um objeto com mais de uma referência possui mais de um nome, então dizemos que o objeto é *aliado*.

Se o objeto aliado for mutável, as alterações feitas com um pseudônimo afetam o outro:

```
>>> b[0] = 17
>>> print(a)
[17, 2, 3]
```

Embora esse comportamento possa ser útil, é propenso a erros. Em geral, é mais seguro evitar aliados quando você está trabalhando com objetos mutáveis.

Para objetos imutáveis como strings, aliados não são um problema tão grande. Neste exemplo:

```
a = 'banana'
b = 'banana'
```

quase nunca tem diferença se a e b irão se referir à mesma string ou não.

8.13 Listas como argumento

Quando você passa uma lista como argumento de uma função, a função recebe uma referência a essa lista. Se a função modificar o parâmetro desta lista, o chamador verá a alteração. Por exemplo, remove_primeiro_elemento remove o primeiro elemento de uma lista:

```
def remove_primeiro_elemento(t)
    del t[0]
```

Abaixo, é mostrado como ela é usada:

```
>>> letras = ['a', 'b', 'c']
>>> remove_primeiro_elemento(letras)
>>> print(letras)
['b', 'c']
```

O parâmetro `t` e a variável `letras` são aliases para o mesmo objeto.

É importante distinguir entre operações que modificam listas e operações que criam novas listas. Por exemplo, o método `append` modifica a lista, porém o operador `+` cria uma nova lista:

```
>>> t1 = [1, 2]
>>> t2 = t1.append(3)
>>> print(t1)
[1, 2, 3]
>>> print(t2)
None

>>> t3 = t1 + [3]
>>> print(t3)
[1, 2, 3]
>>> t2 is t3
False
```

A diferença é importante quando você escreve funções que supostamente modificam listas. Por exemplo, a função *não* remove o início de uma lista:

```
def errado_remove_primeiro_elemento(t)
    t = t[1:]                #ERRADO!
```

O operador 'slice' cria uma nova lista e a atribuição faz com que `t` se refira a essa, mas nenhuma dessas atribuições tem qualquer efeito sobre a lista passada como argumento.

Uma alternativa é escrever uma função que cria e retorna uma nova lista. Por exemplo, `calda` retorna todos, menos o primeiro elemento da lista:

```
def calda(t):
    return t[1:]
```

A função não modifica a lista original. Aqui é mostrado como é usada:

```
>>> letras = ['a', 'b', 'c']
>>> resto = tail(letras)
>>> print(resto)
['b', 'c']
```

Exercício 1: Escreva uma função chamada `corte` que recebe uma lista e a modifica, removendo o primeiro e o último elemento, e retorna `None`. Depois escreva uma função chamada `meio` que recebe uma lista e retorna uma nova lista que contém todos, menos o primeiro e o último elemento.

8.14 Debugging

Descuido com o uso de listas (e outros objetos mutáveis) pode levar a longa horas de debugging. Abaixo mostra algumas deslizes comuns e formas de evitá-los:

1. Não esqueça que a maioria do métodos de listas modifica o argumento e retorna `None`. Isso é o oposto dos métodos das strings, que retornam uma nova string e não modificam a string original.

Se você está acostumado a escrever código de string como este:

```
~~~~ {.python}
palavra = palavra.strip()
~~~~
```

É tentador escrever um código de lista desta forma:

```
~~~~ {.python}
t = t.sort()            # ERRADO!
~~~~

\index{sort método}
\index{método!sort}
```

Como sort retorna `None`, a próxima operação que você executar com `t` provavelmente irá falhar.

Antes de usar métodos de listas e operadores, você deve ler a documentação com cuidado e depois testar elas em um modo interativo. Os métodos e os operadores que as listas compartilham com outras sequência (como strings) são documentadas em:

docs.python.org/library/stdtypes.html#common-sequence-operations

Os métodos e operadores que são apenas aplicadas em sequências mutáveis são documentadas em:

docs.python.org/library/stdtypes.html#mutable-sequence-types

2. Escolha um "idioma" e fique com ele.

   Parte dos problemas, envolvendo listas, é que existem muitas maneiras para fazer coisas. Por exemplo, para remover um elemento de uma lista, você pode usar o método `pop`, `remove`, `del`, ou simplesmente, utilizando uma instrução `slice`.

   Para adicionar um elemento, você pode usar o método `append` ou o operador `+`. Porém, não esqueça que estas são as formas corretas de se fazer estes procedimentos:

   ```
   t.append(x)
   t = t + [x]
   ```

E essas, são as erradas:

```
t.append([x])           # ERRADO!
t = t.append(x)         # ERRADO!
t + [x]                 # ERRADO!
t = t + x               # ERRADO!
```

Tente implementar esses exemplos em um modo interativo para ter certeza que você entende o que eles fazem. Note que apenas o último causa erro na compilação, os outros três são permitidos, porém não faz aquilo que é desejado.

3. Faça cópias para evitar aliados.

   Se você quer usar o método `sort` que modifica o argumento, porém também precisa manter a lista original intacta, você pode criar uma cópia.

   ```
   orig = t[:]
   t.sort()
   ```

   Nesse exemplo, você também pode usar uma função existente chamada `sorted`, que retorna uma nova lista organizada e também mantém a original intacta. Porém, nesse caso, você deve evitar usar `sorted` como um nome da variável!

4. Listas, `split`, e arquivos

   Quando nós lemos e analisamos um arquivo, existem várias ocasiões em que a entrada pode causar falhas em nosso programa, então é uma boa ideia revisitar o padrão *guardião* quando se trata de escrever programas que lêem um arquivo e procuram por uma 'agulha no palheiro'.

   Vamos revisitar nosso programa que procura pelo dia da semana nas linhas do nosso arquivo:

   ```
   De stephen.marquard@uct.ac.za Sáb Jan  5 09:14:16 2008
   ```

   Como estamos quebrando essa linha em palavras, nós podemos dispensar o uso do método `startswith` e simplesmente procurar pela primeira palavra que determina se estamos interessados, de fato, nessa linha. Nós podemos usar `continue` para pular linhas que não têm "De" como sua primeira palavra:

   ```
   fhand = open('mbox-short.txt')
   for linha in fhand:
       palavras = line.split()
       if palavras[0] != 'De' : continue
       print(palavras[2])
   ```

   Isso parece ser bem mais simples, além que não precisamos do método `rstrip` para remover a quebra de linha no final do arquivo. Porém, qual o melhor?

```
python search8.py
Sat
Traceback (most recent call last):
    File "search8.py", line 5, in <module>
        if palavras[0] != 'De' : continue
IndexError: list index out of range
```

Isso funciona e vemos o dia a partir da primeira linha (Sáb), mas o programa falha com `traceback error`. Então, o que deu errado? Por quê o nosso elegante, inteligente e "Pythonic" programa falhou?

Você poderia encarar seu programa por um longo tempo, quebrar a cabeça tentando decifrá-lo, ou simplesmente pode pedir a ajuda de alguém. Porém, a abordagem mais rápida e inteligente é adicionar uma linha de código `print`. O melhor lugar para adicionar essa instrução é logo antes da linha em que o programa falhou e pôr para imprimir na tela o dado que está causando falha.

Essa abordagem pode gerar várias linha na saída, pelo menos, você terá, imediatamente, várias pistas sobre o problema em questão. Então, adicionamos um `print` da variável `letras` logo depois da linha cinco. Nós até adicionamos o prefixo "Debugging:" na linha, para podermos manter nossa saída regular separadas de nossa saída de debugging.

```
for linha in fhand:
    palavras = linha.split()
    print('Debugging:', palavras)
    if palavras[0] != 'De' : continue
    print(palavras[2])
```

Quando compilamos o programa, várias saídas são impressas na tela, porém, no final, vemos nossa saída de debug e a referência para sabermos o que aconteceu logo antes da mensagem de erro.

```
Debug: ['X-DSPAM-Confidence:', '0.8475']
Debug: ['X-DSPAM-Probability:', '0.0000']
Debug: []
Traceback (most recent call last):
  File "search9.py", line 6, in <module>
    if palavras[0] != 'From' : continue
IndexError: list index out of range
```

Cada linha de depuração imprime uma lista de palavras que obtemos quando dividimos a linha em palavras. Se o programa der erro, a lista de palavras é vazia `[]`. Se abrimos o arquivo em um editor de texto e olharmos o mesmo, ele será exibido da seguinte maneira:

```
X-DSPAM-Result: Innocent
X-DSPAM-Processed: Sat Jan  5 09:14:16 2008
X-DSPAM-Confidence: 0.8475
X-DSPAM-Probability: 0.0000

Details: http://source.sakaiproject.org/viewsvn/?view=rev&rev=39772
```

O erro ocorre quando nosso programa encontra uma linha em branco. Claro que não existem palavras em uma linha em branco. Por quê não pensamos nisso quando estávamos escrevendo o código? Quando o código procura pela primeira palavra (`palavras[0]`) para checar se bate com "From", nós temos um erro de índice fora de alcance ("index out of range").

Obviamente, esse é um local perfeito para adicionarmos algum código *guardião* para evitar a checagem da primeira palavra caso a mesma não esteja lá. Existem várias formas de proteger nosso código; nós iremos escolher checar o número de palavras que temos antes de olhar para a primeira palavra:

```
fhand = open('mbox-short.txt')
contador = 0
for linha in fhand:
    palavras = line.split()
    # print 'Debug:', palavras
    if len(palavras) == 0 : continue
    if palavras[0] != 'De' : continue
    print(palavras[2])
```

Primeiro, comentamos as linhas que imprimem o debug, em vez de removê-las (caso nossa modificação falhe, e precisemos do debuggingz novamente). Depois, adicionamos uma declaração guardiã que checa se temos zero palavras, se sim, nós usamos "continue" para pular para próxima linha do arquivo.

Podemos pensar nas duas declarações `continue` como um jeito de nos ajudar a refinar o conjunto de linhas que são interessantes para nós, e quais que queremos processar um pouco mais. Uma linha que não tem "De" como sua primeira palavra não é interessante para nós, então a pulamos.

O programa modificado é executado com sucesso, então, talvez ele esteja correto. Nossa declaração guardiã nos dá a certeza que a `palavras[0]` nunca irá falhar, porém, talvez isso não seja suficiente. Quando estamos programando, nós devemos sempre estar pensando, "O que pode acontecer de errado?"

Exercício 2: Descubra qual linha do programa acima não está devidamente protegida. Veja se você pode construir um arquivo de texto que causa falha no programa e depois modifique o programa para que a linha seja propriamente protegida e por fim, teste o programa para ter certeza que ele manipula corretamente o novo arquivo de texto.

Exercício 3: Reescreva o código guardião nos exemplos acima sem duas declarações `if`. Em vez disso, use uma expressão lógica composta usando o operador lógico `or`, com uma única declaração `if`.

8.15 Glossário

Aliados Circunstância onde duas ou mais variáveis se referem ao mesmo objeto.

delimitador Caractere ou string usado para indicar onde a string deve ser dividida

elemento Um dos valores em uma lista (ou outra sequência); também chamada de itens.
equivalente Ter o mesmo valor
índice Um valor inteiro que indica um elemento da lista
idêntico Ser o mesmo objeto (o que implica em equivalência)
lista Uma sequência de valores
lista de passagem O acesso sequencial sobre cada elemento da lista
lista aninhada Uma lista que é um elemento de uma outra lista.
objeto Algo que uma variável pode se referir. Um objeto tem um tipo e um valor.

referência Associação entre variável e seu valor

8.16 Exercícios

Exercicio 4: Baixe a copia do arquivo www.py4e.com/code3/romeo.txt. Escreva um programa para abrir o arquivo chamado romeo.txt e leia-o linha por linha. Para cada linha, separe-a em uma lista de palavras usando a função `split`. Para cada palavra, cheque se esta palavra já existe na lista. Caso não exista, adicione ela. Quando o programa terminar de verificar, ordene e imprima estas palavras em ordem alfabética.**

```
Digite o nome do arquivo: romeo.txt
['Arise', 'But', 'It', 'Juliet', 'Who', 'already',
'and', 'breaks', 'east', 'envious', 'fair', 'grief',
'is', 'kill', 'light', 'moon', 'pale', 'sick', 'soft',
'sun', 'the', 'through', 'what', 'window',
'with', 'yonder']
```

Exercício 5: Escreva um programa que leia uma caixa de email, e quando você encontrar uma linha que comece com "De", você vai separar a linha em palavras usando a função `split`. Nós estamos interessados em quem envia a mensagem, que é a segunda palavra na linha que começa com From.

```
De stephen.marquard@uct.ac.za Sáb Jan 5 09:14:16 2008
```

Você vai analisar a linha que começa com From e irá pôr para imprimir na tela a segunda palavra (para cada linha do tipo), depois o programa também deverá contar o número de linhas que começam com "De" e imprimir em tela o valor final desse contador. Esse é um bom exemplo da saída com algumas linhas removidas:

```
python fromcount.py
Digite o nome do arquivo: mbox-short.txt
stephen.marquard@uct.ac.za
louis@media.berkeley.edu
zqian@umich.edu

[...some output removed...]
```

```
ray@media.berkeley.edu
cwen@iupui.edu
cwen@iupui.edu
cwen@iupui.edu
There were 27 lines in the file with From as the first word
```

Exercício 6: Reescreva o programa que solicita o usuário uma lista de números e prints e imprime em tela o maior número e o menor número quando o usuário digitar a palavra "feito". Escreva um programa para armazenar as entradas do usuário em uma lista e use as funções `max()` e `min()` para computar o número máximo e o mínimo depois que o laço for completo.

```
Digite um número: 6
Digite um número: 2
Digite um número: 9
Digite um número: 3
Digite um número: 5
Digite um número: feito
Máximo: 9.0
Mínimo: 2.0
```