

# CAPÍTULO 6

## Listas

Listas são um tipo de variável que permite o armazenamento de vários valores, acessados por um índice. Uma lista pode conter zero ou mais elementos de um mesmo tipo ou de tipos diversos, podendo inclusive conter outras listas. O tamanho de uma lista é igual à quantidade de elementos que ela contém.

Podemos imaginar uma lista como um edifício de apartamentos, onde o térreo é o andar zero, o primeiro andar é o andar 1 e assim por diante. O índice é utilizado para especificarmos o “apartamento” onde guardaremos nossos dados.

Em um prédio de seis andares, teremos números de andar variando entre 0 e 5. Se chamarmos nosso prédio de P, teremos P[0] como o endereço do térreo, P[1] como endereço do primeiro andar, continuando assim até P[5]. Em Python, P seria o nome da lista; e o número entre colchetes, o índice.

Listas são mais flexíveis que prédios e podem crescer ou diminuir com o tempo. Vejamos como criar uma lista em Python na listagem 6.1.

### ► Listagem 6.1 – Uma lista vazia

```
L=[]
```

Essa linha cria uma lista chamada L com zero elemento, ou seja, uma lista vazia. Os colchetes ([ ]) após o símbolo de igualdade servem para indicar que L é uma lista. Vejamos agora como criar uma lista z, com 3 elementos na listagem 6.2.

### ► Listagem 6.2 – Uma lista com três elementos

```
Z=[ 15, 8, 9 ]
```

A lista Z foi criada com três elementos: 15, 8 e 9. Dizemos que o tamanho da lista é 3. Como o primeiro elemento tem índice 0, temos que o último elemento é Z[2]. Veja o resultado de testes com z na listagem 6.3.

## Capítulo 6 ■ Listas

### ► Listagem 6.3 – Acesso a uma lista

```
>>> Z=[15,8,9]
>>> Z[0]
15
>>> Z[1]
8
>>> Z[2]
9
```

Utilizando o nome da lista e um índice, podemos mudar o conteúdo de um elemento. Observe os testes no interpretador, apresentados na listagem 6.4.

### ► Listagem 6.4 – Modificação de uma lista

```
>>> Z=[15,8,9]
>>> Z[0]
15
>>> Z[0]=7
>>> Z[0]
7
>>> Z
[7, 8, 9]
```

Quando criamos a lista Z, o primeiro elemento era o número 15. Por isso, Z[0] era 15. Quando executamos Z[0]=7, alteramos o conteúdo do primeiro elemento para 7. Isso pode ser verificado quando pedimos para exibir Z, agora com 7, 8 e 9 como elementos.

Vejamos um exemplo onde um aluno tem cinco notas e no qual desejamos calcular sua média aritmética. Veja o programa na listagem 6.5.

### ► Listagem 6.5 – Cálculo da média

```
notas=[6,7,5,8,9] ❶
soma=0
x=0
while x<5: ❷
    soma += notas[x] ❸
    x+=1 ❹
print("Média: %5.2f" % (soma/x))
```

Criamos a lista de notas em ①. Em ②, criamos a estrutura de repetição para variar o valor de *x* e continuar enquanto este for menor que 5. Lembre-se de que uma lista de cinco elementos contém índices de 0 a 4. Por isso inicializamos *x=0* na linha anterior. Em ③, adicionamos o valor de *notas[0]* à soma e depois *notas[1]*, *notas[2]*, *notas[3]* e *notas[4]*, um elemento a cada repetição. Para isso, utilizamos o valor de *x* como índice e o incrementamos de 1 em ④. Uma grande vantagem desse programa foi que não precisamos declarar cinco variáveis para guardar as cinco notas. Todas as notas foram armazenadas na lista, utilizando um índice para identificar ou acessar cada valor.

Vejamos uma modificação desse exemplo, mas, dessa vez, vamos ler as notas uma a uma. O programa modificado é apresentado na listagem 6.6.

#### ► Listagem 6.6 – Cálculo da média com notas digitadas

```
notas=[0,0,0,0,0] ①
soma=0
x=0

while x<5:
    notas[x]=float(input("Nota %d:" % x)) ②
    soma += notas[x]
    x+=1
x=0 ③

while x<5: ④
    print("Nota %d: %.2f" % (x, notas[x]))
    x+=1
print("Média: %.2f" % (soma/x))
```

Em ① criamos a lista de notas com cinco elementos, todos zero. Em ②, utilizamos a repetição para ler as notas do aluno e armazená-las na lista de notas. Veja que adicionamos *nota[x]* à *soma* já na linha seguinte. Terminada a primeira repetição, teremos a lista de notas preenchidas. Para imprimir a lista de notas, reinicializamos o valor da variável *x* para 0 ③ e criamos outra estrutura de repetição ④.

**Exercício 6.1** Modifique o programa da listagem 6.6 para ler 7 notas em vez de 5.

## 6.1 Trabalhando com índices

Vejamos outro exemplo: um programa que lê cinco números, armazena-os em uma lista e depois solicita que o usuário escolha um número a mostrar. O objetivo é, por exemplo, ler 15, 12, 5, 7 e 9 e armazená-los na lista. Depois, se o usuário digitar 2, ele imprimirá o segundo número digitado, 3, o terceiro, e assim sucessivamente. Observe que o índice do primeiro número é 0, e não 1: essa pequena conversão será feita no programa da listagem 6.7.

#### ► Listagem 6.7 – Apresentação de números

```
números=[0,0,0,0,0]
x=0

while x<5:
    números[x]=int(input("Número %d:" % (x+1))) ①
    x+=1

while True:
    escolhido=int(input("Que posição você quer imprimir (0 para sair): "))
    if escolhido == 0:
        break
    print("Você escolheu o número: %d" % (números[escolhido-1])) ②
```

Execute o programa da listagem 6.7 e experimente alguns valores. Observe que em ① adicionamos 1 a *x* para que possamos imprimir Número 1..5 e não a partir de 0. Isso é importante porque começar a contar de 0 não é natural para a maioria das pessoas. Veja que mesmo imprimindo *x+1* para o usuário, a atribuição é feita para *números[x]* porque nossas listas começam em 0. Em ②, fizemos a operação inversa. Quando o usuário escolhe o número a imprimir, ele faz uma escolha entre 1 e 5. Como 1 é o elemento 0, 2, o elemento 1, e assim por diante. Diminuímos o valor da escolha de um para obtermos o índice de notas.

## 6.2 Cópia e fatiamento de listas

Embora listas em Python sejam um recurso muito poderoso, todo poder traz responsabilidades. Um dos efeitos colaterais de listas aparece quando tentamos fazer cópias. Vejamos um teste no interpretador na listagem 6.8.

### ► Listagem 6.8 – Tentativa de copiar listas

```
>>> L=[1,2,3,4,5]
>>> V=L
>>> L
[1, 2, 3, 4, 5]
>>> V
[1, 2, 3, 4, 5]
>>> V[0]=6
>>> V
[6, 2, 3, 4, 5]
>>> L
[6, 2, 3, 4, 5]
```

Veja que, ao modificarmos `V`, modificamos também o conteúdo de `L`. Isso porque uma lista em Python é um objeto e, quando atribuímos um objeto a outro, estamos apenas copiando a mesma referência da lista, e não seus dados em si. Nesse caso, `V` funciona como um apelido de `L`, ou seja, `V` e `L` são a mesma lista. Vejamos o que acontece no gráfico da figura 6.1.

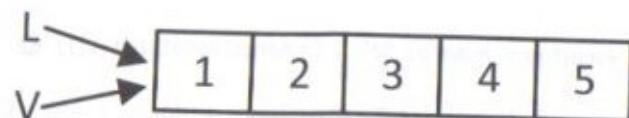


Figura 6.1 – Duas variáveis referenciando a mesma lista.

Quando modificamos `V[0]`, estamos modificando o mesmo valor de `L[0]`, pois ambos são referências, ou apelidos para a mesma lista na memória.

Dependendo da aplicação, esse efeito pode ser desejado ou não. Para criar uma cópia independente de uma lista, utilizaremos outra sintaxe. Vejamos o resultado das operações da listagem 6.9.

### ► Listagem 6.9 – Cópia de listas

```
>>> L=[1,2,3,4,5]
>>> V=L[:]
>>> V[0]=6
>>> L
[1, 2, 3, 4, 5]
>>> V
[6, 2, 3, 4, 5]
```

Ao escrevermos `L[:]`, estamos nos referindo a uma nova cópia de `L`. Assim `L` e `V` se referem a áreas diferentes na memória, permitindo alterá-las de forma independente, como na figura 6.2.



Figura 6.2 – Duas variáveis referenciando duas listas.

Podemos também fatiar uma lista, da mesma forma que fizemos com strings no capítulo 3. Vejamos alguns exemplos no interpretador na listagem 6.10.

### ► Listagem 6.10 – Fatiamento de listas

```
>>> L=[1,2,3,4,5]
>>> L[0:5]
[1, 2, 3, 4, 5]
>>> L[:5]
[1, 2, 3, 4, 5]
>>> L[:-1]
[1, 2, 3, 4]
>>> L[1:3]
[2, 3]
>>> L[1:4]
[2, 3, 4]
>>> L[3:]
[4, 5]
>>> L[:3]
[1, 2, 3]
>>> L[-1]
5
>>> L[-2]
4
```

Veja que índices negativos também funcionam. Um índice negativo começa a contar do último elemento, mas observe que começamos de `-1`. Assim `L[0]` representa o primeiro elemento; `L[-1]`, o último; `L[-2]`, o penúltimo, e assim por diante.

## 6.3 Tamanho de listas

Podemos usar a função `len` com listas. O valor retornado é igual ao número de elementos da lista. Veja alguns testes na listagem 6.11.

### ► Listagem 6.11 – Tamanho de listas

```
>>> L=[12,9,5]
>>> len(L)
3
>>> V=[]
>>> len(V)
0
```

A função `len` pode ser utilizada em repetições para controlar o limite dos índices. Veja o exemplo na listagem 6.12.

### ► Listagem 6.12 – Repetição com tamanho fixo da lista

```
L=[1,2,3]
x=0

while x < 3:
    print(L[x])
    x+=1
```

Isso pode ser reescrito como na listagem 6.13.

### ► Listagem 6.13 – Repetição com tamanho da lista usando `len`

```
L=[1,2,3]
x=0

while x < len(L):
    print(L[x])
    x+=1
```

A vantagem é que se trocarmos `L` para:

```
L=[7,8,9,10,11,12]
```

o resto do programa continuaria funcionando, pois utilizamos a função `len` para calcular o tamanho da lista. Observe que o valor retornado pela função `len` é um número que não pode ser utilizado como índice, mas que é perfeito para testarmos os

limites de uma lista, como fizemos na listagem 6.13. Isso acontece porque `len` retorna a quantidade de elementos na lista e nossos índices começam a ser numerados de 0 (zero). Assim, os índices válidos de uma lista (`L`) variam de 0 até o valor de `len(L)-1`.

## 6.4 Adição de elementos

Uma das principais vantagens de trabalharmos com listas é poder adicionar novos elementos durante a execução do programa. Vejamos um teste no interpretador (Listagem 6.14).

Para adicionar um elemento ao fim da lista, utilizaremos o método `append`. Em Python, chamamos um método escrevendo o nome dele após o nome do objeto. Como listas são objetos, sendo `L` a lista, teremos `L.append(valor)`. Métodos são recursos de orientação a objetos, suportados e muito usados em Python. Você pode imaginar um método como uma função do objeto. Quando invocado, ele já sabe a que objeto estamos nos referindo, pois o informamos à esquerda do ponto. Falaremos mais sobre métodos no capítulo 10; por enquanto, observe como os utilizamos e saiba que são diferentes de funções.

### ► Listagem 6.14 – Adição de elementos à lista

```
>>> L=[]
>>> L.append("a")
>>> L
['a']
>>> L.append("b")
>>> L
['a', 'b']
>>> L.append("c")
>>> L
['a', 'b', 'c']
>>> len(L)
3
```

Vejamos um programa que lê números até que 0 seja digitado. Esse programa depois os imprimirá na mesma ordem em que foram digitados (Listagem 6.15).

### ► Listagem 6.15 – Adição de elementos à lista

```
L=[]
while True:
    n=int(input("Digite um número (0 sai):"))
    if n == 0:
        break
    L.append(n)
x=0
while x < len(L):
    print(L[x])
    x=x+1
```

Esse simples programa é capaz de ler e imprimir um número inicialmente indeterminado de valores. Isso é possível porque adicionamos elementos à lista `L`, conforme necessário.

Outra forma de adicionarmos elementos a uma lista é com adição de listas (Listagem 6.16).

Quando adicionamos apenas um elemento, tanto `L.append(1)` quanto `L+[1]` produzem o mesmo resultado. Perceba que em `L+[1]` escrevemos o elemento a adicionar dentro de uma lista (`[1]`), e que, em `append`, apenas `L`. Isso porque, quando adicionamos uma lista a outra, o interpretador executa um método chamado `extend` que adiciona os elementos de uma lista a outra. Vejamos alguns exemplos na listagem 6.17.

### ► Listagem 6.16 – Adição de listas

```
>>> L=[]
>>> L=L+[1]
>>> L
[1]
>>> L+=[2]
>>> L
[1, 2]
>>> L+=[3,4,5]
>>> L
[1, 2, 3, 4, 5]
```

### Capítulo 6 ■ Listas

### ► Listagem 6.17 – Adição de elementos e listas

```
>>> L=["a"]
>>> L.append("b")
>>> L
['a', 'b']
>>> L.extend(["c"])
>>> L
['a', 'b', 'c']
>>> L.append(["d","e"])
>>> L
['a', 'b', 'c', ['d', 'e']]
>>> L.extend(["f","g","h"])
>>> L
['a', 'b', 'c', ['d', 'e'], 'f', 'g', 'h']
```

O método `extend` sequer aceita parâmetros que não sejam listas. Se você utilizar o método `append` com uma lista como parâmetro, em vez de adicionar os elementos no fim da lista, `append` adicionará a lista inteira, mas como apenas um novo elemento. Teremos então listas dentro de listas (Listagem 6.18).

### ► Listagem 6.18 – Adição de elementos e listas com append

```
>>> L=["a"]
>>> L.append(["b"])
>>> L.append(["c","d"])
>>> len(L)
3
>>> L[1]
['b']
>>> L[2]
['c', 'd']
>>> len(L[2])
2
>>> L[2][1]
'd'
```

Esse conceito é interessante, pois permite a utilização de estruturas de dados mais complexas, como matrizes, árvores e registros. Por enquanto, vamos utilizar esse recurso para armazenar múltiplos valores por elemento.

**Exercício 6.2** Faça um programa que leia duas listas e que gere uma terceira com os elementos das duas primeiras.

**Exercício 6.3** Faça um programa que percorra duas listas e gere uma terceira sem elementos repetidos.

## 6.5 Remoção de elementos da lista

Como o tamanho da lista pode variar, permitindo a adição de novos elementos, podemos também retirar alguns elementos da lista, ou mesmo todos eles. Para isso, utilizaremos a instrução `del` (Listagem 6.19).

### ► Listagem 6.19 – Remoção de elementos

```
>>> L=["a","b","c"]
>>> del L[1]
>>> L
['a', 'c']
>>> del L[0]
>>> L
['c']
```

É importante notar que o elemento excluído não ocupa mais lugar na lista, fazendo com que os índices sejam reorganizados, ou melhor, que passem a ser calculados sem esse elemento.

Podemos também apagar fatias inteiras de uma só vez (Listagem 6.20).

### ► Listagem 6.20 – Remoção de fatias

```
>>> L=list(range(101))
>>> del L[1:99]
>>> L
[0, 99, 100]
```

## 6.6 Usando listas como filas

Uma lista pode ser utilizada como fila se obedecermos a certas regras de inclusão e eliminação de elementos. Em uma fila, a inclusão é sempre realizada no fim, e as remoções são feitas no início. Dizemos que o primeiro a chegar é o primeiro a sair (*FIFO - First In First Out*).

É mais simples de entender se imaginarmos uma fila de banco. Quando a agência abre pela manhã, a fila está vazia. Quando os clientes começam a chegar, eles vão diretamente para o fim da fila. Os caixas então começam a atender esses clientes por ordem de chegada, ou seja, o cliente que chegou primeiro será atendido primeiro. Uma vez que o cliente é atendido, ele sai da fila. Então, um novo cliente passa a ser o primeiro da fila e o próximo a ser atendido.

Para escrevermos algo similar em Python, imaginaremos uma lista de clientes, representando a fila, onde o valor de cada elemento é igual à ordem de chegada do cliente. Vamos imaginar uma lista inicial com 10 clientes. Se outro cliente chegar, realizaremos um `append` para que ele seja inserido no fim da fila (`fila.append(último)`). Para retirarmos um cliente da fila e atendê-lo, poderíamos fazer `del fila[0]`, porém, isso apagaria o cliente da fila. Se quisermos retirá-lo da fila e, ao mesmo tempo, obter o elemento retirado, podemos utilizar o método `pop` (`fila.pop(0)`). O método `pop` retorna o valor do elemento e o exclui da fila. Passamos 0 como parâmetro para indicar que queremos excluir o primeiro elemento. Veja o programa completo na listagem 6.21.

**Exercício 6.4** O que acontece quando não verificamos se a lista está vazia antes de chamarmos o método `pop`?

**Exercício 6.5** Altere o programa da listagem 6.21 de forma a poder trabalhar com vários comandos digitados de uma só vez. Atualmente, apenas um comando pode ser inserido por vez. Altere-o de forma a considerar operação como uma string.

Exemplo: FFFAAAS significaria três chegadas de novos clientes, três atendimentos e, finalmente, a saída do programa.

**Exercício 6.6** Modifique o programa para trabalhar com duas filas. Para facilitar seu trabalho, considere o comando A para atendimento da fila 1; e B, para atendimento da fila 2. O mesmo para a chegada de clientes: F para fila 1; e G, para fila 2.

#### ► Listagem 6.21 – Simulação de uma fila de banco

```

último = 10
fila = list(range(1,último+1))
while True:
    print("\nExistem %d clientes na fila" % len(fila))
    print("Fila atual:", fila)
    print("Digite F para adicionar um cliente ao fim da fila,")
    print("ou A para realizar o atendimento. S para sair.")
    operação = input("Operação (F, A ou S):")
    if operação == "A":
        if(len(fila))>0:
            atendido = fila.pop(0)
            print("Cliente %d atendido" % atendido)
        else:
            print("Fila vazia! Ninguém para atender.")
    elif operação == "F":
        último+=1 # Incrementa o ticket do novo cliente
        fila.append(último)
    elif operação == "S":
        break
    else:
        print("Operação inválida! Digite apenas F, A ou S!")

```

## 6.7 Uso de listas como pilhas

Uma pilha tem uma política de acesso bem definida: novos elementos são adicionados ao topo. A retirada de elementos também é feita pelo topo.

Imagine uma pilha de pratos para lavar. Retiramos o prato que está no topo da pilha para lavar e, se mais alguns pratos chegarem, serão também adicionados ou empilhados ao topo, ou seja, um sobre o outro. Na pilha, o último elemento a chegar é o primeiro a sair (*LIFO - Last In First Out*). Por enquanto, vamos nos concentrar nas pilhas de prato. Veja o programa da listagem 6.22, que simula uma pia de cozinha cheia de pratos.

Você deve ter percebido que o exemplo da listagem 6.22 é muito parecido com o programa da listagem 6.21. Isso porque a grande diferença entre pilhas e filas é o

elemento que escolhemos para retirar. Em uma fila, o primeiro elemento é retirado primeiro. Já em pilhas, retira-se a partir do último elemento. A única mudança em Python é o valor que passamos para o método `pop`. No caso de uma pilha, como retiramos o último elemento, passamos `-1` a `pop`.

#### ► Listagem 6.22 – Pilha de pratos

```

prato = 5
pilha = list(range(1,prato+1))
while True:
    print("\nExistem %d pratos na pilha" % len(pilha))
    print("Pilha atual:", pilha)
    print("Digite E para empilhar um novo prato,")
    print("ou D para desempilhar. S para sair.")
    operação = input("Operação (E, D ou S):")
    if operação == "D":
        if(len(pilha))>0:
            lavado = pilha.pop(-1)
            print("Prato %d lavado" % lavado)
        else:
            print("Pilha vazia! Nada para lavar.")
    elif operação == "E":
        prato+=1 # Novo prato
        pilha.append(prato)
    elif operação == "S":
        break
    else:
        print("Operação inválida! Digite apenas E, D ou S!")

```

Você pode também observar o uso de pilhas com seu browser de internet. Abra algumas páginas, clicando em seus links. Observe como o histórico de navegação é modificado. Cada novo link adiciona uma página a seu histórico de navegação. Se você clicar em voltar uma página, o browser utilizará a última página que entrou no histórico. Nesse caso, o histórico do browser funciona como uma pilha. Na realidade, ele é um pouco mais complexo, pois permite que voltemos várias vezes e que depois possamos voltar no outro sentido se necessário (avançar). Agora faça outro teste: volte duas ou três vezes e depois visite um novo link. Dessa vez, o browser deve ter desativado a função de avanço, uma vez que o histórico mudou

(você mudou de direção). Tente entender esse processo como duas pilhas, uma à esquerda e outra à direta. Quando você escolhe voltar, desempilhamos um elemento da pilha à esquerda. Ao visitar um novo link, adicionamos um novo elemento a essa mesma pilha. Para simular a opção de avançar, imagine que, ao retirarmos um elemento da pilha à esquerda, devemos adicioná-lo à da direita. Se escolhermos um endereço novo, apagamos a pilha da direita.

Essas mesmas operações podem ser simuladas com outras estruturas, como uma lista simples e uma variável contendo onde estamos no histórico de navegação. A cada novo endereço, adicionariamos um novo elemento à lista e atualizariamos nossa posição. Ao voltarmos, decrementaríamos a posição e a incrementaríamos no caso de avanço. Se mudássemos de direção, bastaria apagar os elementos entre a posição atual e o fim da lista antes de adicionar o novo elemento.

A importância de aprender a manipular listas como filas ou pilhas é entender algoritmos mais complexos no futuro.

**Exercício 6.7** Faça um programa que leia uma expressão com parênteses. Usando pilhas, verifique se os parênteses foram abertos e fechados na ordem correta. Exemplo:

(( ))	OK
((())())	OK
(( ))	Erro

Você pode adicionar elementos à pilha sempre que encontrar abre parênteses e desempilhá-la a cada fecha parênteses. Ao desempilhar, verifique se o topo da pilha é um abre parênteses. Se a expressão estiver correta, sua pilha estará vazia no final.

## 6.8 Pesquisa

Podemos pesquisar se um elemento está ou não em uma lista, verificando do primeiro ao último elemento se o valor procurado estiver presente. Vejamos a listagem 6.23.

### ► Listagem 6.23 – Pesquisa sequencial

```
L=[15,7,27,39]
p=int(input("Digite o valor a procurar:"))
achou=False ①
```

```
x=0
while x<len(L):
    if L[x]==p:
        achou=True ②
        break ③
    x+=1
if achou: ④
    print("%d achado na posição %d" % (p,x))
else:
    print("%d não encontrado" % p)
```

A pesquisa simplesmente compara todos os elementos da lista com o valor procurado, interrompendo a repetição ao encontrar o primeiro elemento cujo valor é igual ao procurado. É importante saber que podemos ou não encontrar o que procuramos, daí a utilização da variável `achou` em ①. Essa variável do tipo lógico (booleano) será utilizada para verificar se saímos da repetição por termos achado o que procurávamos ou simplesmente porque visitamos todos os elementos sem encontrar o valor procurado. Veja que `achou` é marcada como `True` em ②, mas dentro de `if` com a condição de pesquisa, e antes do `break` em ③. Dessa forma, `achou` só será `True` se algum elemento for igual ao valor procurado. Em ④, verificamos o valor de `achou` para decidir o que vamos imprimir.

**Exercício 6.8** Modifique o exemplo (Listagem 6.23) de forma a realizar a mesma tarefa, mas sem utilizar a variável `achou`. Dica: observe a condição de saída do `while`.

**Exercício 6.9** Modifique o exemplo para pesquisar dois valores. Em vez de apenas `p`, leia outro valor `v` que também será procurado. Na impressão, indique qual dos dois valores foi achado primeiro.

**Exercício 6.10** Modifique o programa do exercício 6.9 de forma a pesquisar `p` e `v` em toda a lista e informando o usuário a posição onde `p` e a posição onde `v` foram encontrados.

## 6.9 Usando for

Python apresenta uma estrutura de repetição especialmente projetada para percorrer listas. A instrução **for** funciona de forma parecida a **while**, mas a cada repetição utiliza um elemento diferente da lista.

A cada repetição, o próximo elemento da lista é utilizado, o que se repete até o fim da lista. Vamos escrever um programa que utilize **for** para imprimir todos os elementos de uma lista (Listagem 6.24).

### ► Listagem 6.24 – Impressão de todos os elementos da lista com for

```
L=[8,9,15]
for e in L: ❶
    print(e) ❷
```

Quando começamos a executar o **for** em ❶, temos **e** igual ao primeiro elemento da lista, no caso, 8, ou seja, `L[0]`. Em ❷ imprimimos 8, e a execução do programa volta para ❶, onde **e** passa a valer 9, ou seja, `L[1]`. Na próxima repetição **e** valerá 15, ou seja, `L[2]`. Depois de imprimir o último número, a repetição é concluída, pois não temos mais elementos a substituir. Se tivéssemos que fazer a mesma tarefa com **while**, teríamos que escrever um programa como o da listagem 6.25.

### ► Listagem 6.25 – Impressão de todos os elementos da lista com while

```
L=[8,9,15]
x=0
while x<len(L):
    e=L[x]
    print(e)
    x+=1
```

Embora a instrução **for** facilite nosso trabalho, ela não substitui completamente **while**. Dependendo do problema, utilizaremos **for** ou **while**. Normalmente utilizaremos **for** quando quisermos processar os elementos de uma lista, um a um. **while** é indicado para repetições nas quais não sabemos ainda quantas vezes vamos repetir ou onde manipulamos os índices de forma não sequencial.

Vale lembrar que a instrução **break** também interrompe o **for**. Vejamos a pesquisa, escrita usando **for**, na listagem 6.26.

### ► Listagem 6.26 – Pesquisa usando for

```
L=[7,9,10,12]
p=int(input("Digite um número a pesquisar:"))
for e in L:
    if e == p:
        print("Elemento encontrado!")
        break ❶
    else: ❷
        print("Elemento não encontrado.")
```

Utilizamos a instrução **break** para interromper a busca depois de encontrarmos o primeiro elemento em ❶. Em ❷ utilizamos um **else**, parecido com o da instrução **if**, para imprimir a mensagem informando que o elemento não foi encontrado. O **else** deve ser escrito na mesma coluna de **for** e só será executado se todos os elementos da lista forem visitados, ou seja, se não utilizarmos a instrução **break**, deixando **for** terminar normalmente.

**Exercício 6.11** Modifique o programa da listagem 6.15 usando **for**. Explique por que nem todos os **while** podem ser transformados em **for**.

## 6.10 Range

Podemos utilizar a função **range** para gerar listas simples. A função **range** não retorna uma lista propriamente dita, mas um gerador ou *generator*. Por enquanto, basta entender como podemos usá-la. Imagine um programa simples que imprime de 0 a 9 na tela (Listagem 6.27).

### ► Listagem 6.27 – Uso da função range

```
for v in range(10):
    print(v)
```

Execute o programa e veja o resultado. A função **range** gerou números de 0 a 9 porque passamos 10 como parâmetro. Ela normalmente gera valores a partir de 0, logo, ao especificarmos apenas 10, estamos apenas informando onde parar. Experimente mudar de 10 para 20. O programa deve imprimir de 0 a 19. Agora

experimente 20000. A vantagem de utilizarmos a função `range` é gerar listas eficientemente, como mostrado no exemplo, sem precisar escrever os 20.000 valores no programa.

Com a mesma função `range`, podemos também indicar qual é o primeiro número a gerar. Para isso, utilizaremos dois parâmetros: `início` e `fim` (Listagem 6.28).

#### ► Listagem 6.28 – Uso da função range com intervalos

```
for v in range(5, 8):
    print(v)
```

Usando 5 como `início` e 8 como `fim`, vamos imprimir os números 5, 6 e 7. A notação aqui para o `fim` é a mesma utilizada com fatias, ou seja, o `fim` é um intervalo aberto, isto é, não incluso na faixa de valores.

Se acrescentarmos um terceiro parâmetro à função `range`, teremos como saltar entre os valores gerados, por exemplo, `range(0, 10, 2)` gera os pares entre 0 e 10, pois começa de 0 e adiciona 2 a cada elemento. Vejamos um exemplo onde geramos os 10 primeiros múltiplos de 3 (Listagem 6.29).

#### ► Listagem 6.29 – Uso da função range com saltos

```
for t in range(3, 33, 3):
    print(t, end=" ")
print()
```

Observe que um gerador como o retornado pela função `range` não é exatamente uma lista. Embora seja usado de forma parecida, é, na realidade, um objeto de outro tipo. Para transformar um gerador em lista, utilize a função `list` (Listagem 6.30).

#### ► Listagem 6.30 – Transformação do resultado de range em uma lista

```
L=list(range(100,1100,50))
print(L)
```

Voltando à listagem 6.29, observe que utilizamos uma construção especial com a função `print`, onde `t` é o valor que queremos imprimir, mas com `end=" "`, que indica a função para não pular de linha após a impressão. `end` é, na realidade, um parâmetro opcional da função `print`. Veremos mais sobre isso quando estudarmos funções no capítulo 8. Veja também que, para saltar a linha no final do programa, fizemos uma chamada a `print()` sem qualquer parâmetro.

## 6.11 Enumerate

Com a função `enumerate` podemos ampliar as funcionalidades de `for` facilmente. Vejamos como imprimir uma lista, onde teremos o índice entre colchetes e o valor à sua direita (Listagem 6.31).

#### ► Listagem 6.31 – Impressão de índices sem usar a função enumerate

```
L=[5,9,13]
x=0
for e in L:
    print("[%d] %d" % (x,e))
    x+=1
```

Veja o mesmo programa, mas utilizando a função `enumerate` na listagem (Listagem 6.32).

#### ► Listagem 6.32 – Impressão de índices usando a função enumerate

```
L=[5,9,13]
for x, e in enumerate(L):
    print("[%d] %d" % (x,e))
```

A função `enumerate` gera uma tupla em que o primeiro valor é o índice e o segundo é o elemento da lista sendo enumerada. Ao utilizarmos `x`, `e` em `for`, indicamos que o primeiro valor da tupla deve ser colocado em `x`, e o segundo, em `e`. Assim, na primeira iteração teremos a tupla (0,5), onde `x=0` e `e=5`. Isso é possível porque Python permite o desempacotamento de valores de uma tupla, atribuindo um elemento da tupla a cada variável em `for`. O que temos a cada iteração de `for` é equivalente a `x, e = (0,5)`, em que o gerador `enumerate` retorna cada vez uma nova tupla. Os próximos valores retornados são (1,9) e (2,13), respectivamente. Experimente substituir `x, e` na listagem 6.30 por `z`. Antes de `print`, faça `x, e = z`. Adicione mais um `print` para exibir também o valor de `z`.

## 6.12 Operações com listas

Podemos percorrer uma lista de forma a verificar o menor e o maior valor (Listagem 6.33).

### ► Listagem 6.33 – Verificação do maior valor

```
L=[1,7,2,4]
máximo=L[0] ①
for e in L:
    if e > máximo:
        máximo = e
print(máximo)
```

Em ①, utilizamos um pequeno truque, inicializando o máximo com o valor do primeiro elemento. Precisamos de um valor para máximo antes de utilizá-lo na comparação com `if`. Se usássemos 0, não teríamos problema, desde que nossa lista não tenha valores negativos.

**Exercício 6.12** Altere o programa da listagem 6.33 de forma a imprimir o menor elemento da lista.

**Exercício 6.13** A lista de temperaturas de Mons, na Bélgica, foi armazenada na lista `T = [-10, -8, 0, 1, 2, 5, -2, -4]`. Faça um programa que imprima a menor e a maior temperatura, assim como a temperatura média.

## 6.13 Aplicações

Vejamos uma situação na qual temos que selecionar os elementos de uma lista de forma a copiá-los para outras duas listas. Para simplificar o problema, imagine que os valores estejam inicialmente na lista `V`, mas que devam ser copiados para `P`, se forem pares; ou para a `I`, se forem ímpares. Veja o programa que resolve esse problema, na listagem 6.34.

### ► Listagem 6.34 – Cópia de elementos para outras listas

```
V=[9,8,7,12,0,13,21]
P=[]
I=[]
for e in V:
    if e % 2 == 0:
        P.append(e)
    else:
```

```
    I.append(e)
print("Pares: ", P)
print("Ímpares: ", I)
```

Vejamos agora um programa que controla a utilização das salas de um cinema. Imagine que a lista `Salas = [10,2,1,3,0]` contenha o número de lugares vagos nas salas 1, 2, 3, 4 e 5, respectivamente. Esse programa lerá o número da sala e a quantidade de lugares solicitados. Ele deve informar se é possível vender o número de lugares solicitados, ou seja, se ainda há lugares livres. Caso seja possível vender os bilhetes, atualizará o número de lugares livres. A saída ocorre quando se digita 0 no número da sala (Listagem 6.35).

### ► Listagem 6.35 – Controle da utilização de salas de um cinema

```
lugares_vagos=[10,2,1,3,0]
while True:
    sala=int(input("Sala (0 sai): "))
    if sala == 0:
        print("Fim")
        break
    if sala>len(lugares_vagos) or sala<1:
        print("Sala inválida")
    elif lugares_vagos[sala-1]==0:
        print("Desculpe, sala lotada!")
    else:
        lugares =int(input("Quantos lugares você deseja (%d vagos): "% lugares_vagos[sala-1]))
        if lugares > lugares_vagos[sala-1]:
            print("Esse número de lugares não está disponível.")
        elif lugares < 0:
            print("Número inválido")
        else:
            lugares_vagos[sala-1]-=lugares
            print("%d lugares vendidos" % lugares)
print("Utilização das salas")
for x,l in enumerate(lugares_vagos):
    print("Sala %d - %d lugar(es) vazio(s)" % (x+1, l))
```

## 6.14 Listas com strings

No capítulo 3 vimos que strings podem ser indexadas ou acessadas letra por letra. Listas em Python funcionam da mesma forma, permitindo o acesso a vários valores e se tornando uma das principais estruturas de programação da linguagem.

Vejamos agora outro exemplo de listas, mas utilizando strings, na listagem 6.36. Nesse caso, S é uma lista, e cada elemento, uma string.

### ► Listagem 6.36 – Listas com strings

```
>>> S=["maçãs", "peras", "kiwis"]
>>> print(len(S))
3
>>> print(S[0])
maçãs
>>> print(S[1])
peras
>>> print(S[2])
kiwis
```

O programa da listagem 6.37 lê e imprime uma lista de compras até que seja digitado fim.

### ► Listagem 6.37 – Lendo e imprimindo uma lista de compras

```
compras=[]
while True:
    produto=input("Produto:")
    if produto == "fim":
        break
    compras.append(produto)
for p in compras:
    print(p)
```

## 6.15 Listas dentro de listas

Um fator interessante é que podemos acessar as strings dentro da lista, letra por letra, usando um segundo índice. Vejamos as listagens 6.38 e 6.39.

### ► Listagem 6.38 – Listas com strings, acessando letras

```
>>> S=["maçãs", "peras", "kiwis"]
>>> print(S[0][0])
m
>>> print(S[0][1])
a
>>> print(S[1][1])
e
>>> print(S[2][2])
w
```

### ► Listagem 6.39 – Impressão de uma lista de strings, letra a letra

```
L=["maçãs", "peras", "kiwis"]
for s in L:
    for letra in s:
        print(letra)
```

Isso nos leva a outra vantagem das listas em Python: listas dentro de listas. Como bônus, temos também que os elementos de uma lista não precisam ser do mesmo tipo. Vejamos um exemplo onde teríamos uma lista de compras na listagem 6.40. O primeiro elemento seria o nome do produto; o segundo, a quantidade; e o terceiro, seu preço.

### ► Listagem 6.40 – Listas com elementos de tipos diferentes

```
produto1 = [ "maçã", 10, 0.30]
produto2 = [ "pera", 5, 0.75]
produto3 = [ "kiwi", 4, 0.98]
```

Assim, produto1, produto2, produto3 seriam três listas com três elementos cada uma. Observe que o primeiro elemento é do tipo string; o segundo, do tipo inteiro; e o terceiro, do tipo ponto flutuante (*float*)!

Vejamos agora outra lista na listagem 6.41.

### ► Listagem 6.41 – Listas de listas

```
produto1 = [ "maçã", 10, 0.30]
produto2 = [ "pera", 5, 0.75]
produto3 = [ "kiwi", 4, 0.98]
```

```
compras = [ produto1, produto2, produto3]
print(compras)
```

Agora temos uma lista chamada `compras`, também com três elementos, mas cada elemento é uma lista à parte. Para imprimir essa lista, teríamos o programa da listagem 6.42.

#### ► Listagem 6.42 – Impressão das compras

```
produto1 = [ "maçã", 10, 0.30]
produto2 = [ "pera", 5, 0.75]
produto3 = [ "kiwi", 4, 0.98]
compras = [ produto1, produto2, produto3]
for e in compras:
    print("Produto: %s" % e[0])
    print("Quantidade: %d" % e[1])
    print("Preço: %.2f" % e[2])
```

Da mesma forma, poderíamos ter acessado o preço do segundo elemento com `compras[1][2]`. Vejamos agora um programa completo, capaz de perguntar nome do produto, quantidade e preço e, no final, imprimir uma lista de compras completa.

#### ► Listagem 6.43 – Criação e impressão da lista de compras

```
compras = []
while True:
    produto = input("Produto: ")
    if produto == "fim":
        break
    quantidade = int(input("Quantidade: "))
    preço = float(input("Preço: "))
    compras.append([produto, quantidade, preço])
soma = 0.0
for e in compras:
    print("%20s %5d %.2f %.2f" % (e[0],
                                    e[1], e[2],
                                    e[1] * e[2]))
    soma += e[1] * e[2]
print("Total: %.2f" % soma)
```

## 6.16 Ordenação

Até agora, os elementos de nossas listas apresentam a mesma ordem em que foram digitados, sem qualquer ordenação. Para ordenar uma lista, realizaremos uma operação semelhante à da pesquisa, mas trocando a ordem dos elementos quando necessário. Um algoritmo muito simples de ordenação é o “Bubble Sort”, ou método de bolhas, fácil de entender e aprender. Por ser lento, você não deve utilizá-lo com listas grandes.

A ordenação pelo método de bolhas consiste em comparar dois elementos a cada vez. Se o valor do primeiro elemento for maior que o do segundo, eles trocarão de posição. Essa operação é então repetida para o próximo elemento até o fim da lista. O método de bolhas exige que percorramos a lista várias vezes. Por isso, utilizaremos um marcador para saber se chegamos ao fim da lista trocando ou não algum elemento. Esse método tem outra propriedade, que é posicionar o maior elemento na última posição da lista, ou seja, sua posição correta. Isso permite eliminar um elemento do fim da lista a cada passagem completa. Vejamos o programa da listagem 6.44.

#### ► Listagem 6.44 – Ordenação pelo método de bolhas

```
L=[7,4,3,12,8]
fim=5 ❶
while fim > 1: ❷
    trocou=False ❸
    x=0 ❹
    while x<(fim-1): ❺
        if L[x] > L[x+1]: ❻
            trocou=True ❽
            temp=L[x] ❾
            L[x]=L[x+1]
            L[x+1]=temp
        x+=1
    if not trocou: ❿
        break
    fim-=1 ❾
for e in L:
    print(e)
```

Execute o programa e tente entender como ele funciona. Em ❶, utilizamos a variável `fim` para marcar a quantidade de elementos da lista. Precisamos marcar o fim ou a posição do último elemento porque no “Bubble Sort” não precisamos verificar o último elemento após uma passagem completa. Em ❷, verificamos se `fim > 1`, pois como compararmos o elemento atual(`L[x]`) com o seguinte(`L[x+1]`), precisamos de, no mínimo, dois elementos. Utilizamos a variável `trocou` em ❸ para indicar que não realizamos nenhuma troca. O valor de `trocou` será utilizado mais tarde para verificar se já temos a lista ordenada ou não. A variável `x` ❹ será utilizada como índice, começando da posição 0. Nossa condição do segundo `while` ❺ é especial, pois temos que garantir um próximo elemento para comparar com o atual. Isso faz com que a condição de saída desse `while` seja `x < (fim - 1)`, ou melhor, que `x` seja anterior ao último elemento. Em ❻ temos a comparação em si, onde `x` é nossa posição atual, e o próximo elemento é o de índice `x+1`. Como estamos ordenando em ordem crescente, desejamos que o próximo elemento sempre seja maior que o atual, dessa forma garantindo a ordenação da lista. Como compararmos apenas dois elementos de cada vez, temos que visitar a lista várias vezes, até que todos os elementos estejam em suas posições. Se a condição de ❻ for verdadeira, esses elementos estão fora de ordem. Nesse caso, devemos inverter ou trocar a posição dos dois elementos. Em ❼ marcamos que efetuamos uma troca e, em ❽, utilizamos uma variável auxiliar ou temporária para trocar os dois valores de posição.

A troca de valores entre duas variáveis é mostrada nas figuras 6.3, 6.4 e 6.5. Como cada variável só guarda um valor de cada vez, utilizaremos uma terceira variável temporária (`temp`) para armazenar o valor de uma delas durante a troca.

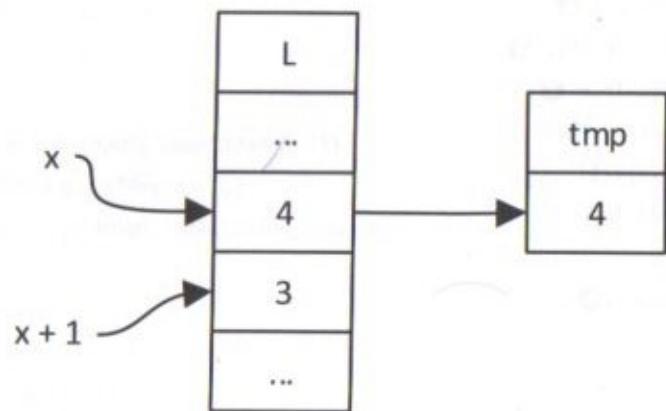


Figura 6.3 – Troca passo a passo. Primeira etapa.

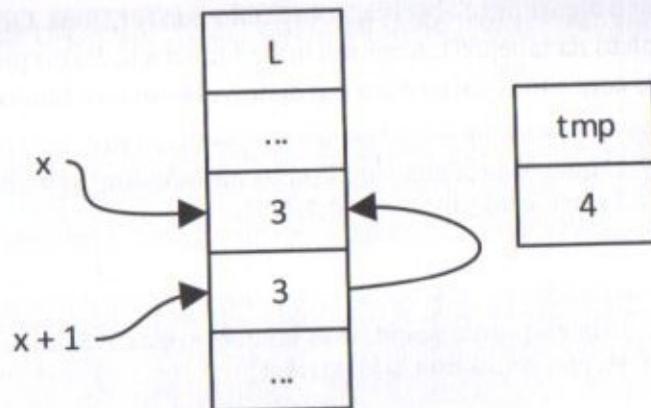


Figura 6.4 – Troca passo a passo. Segunda etapa.

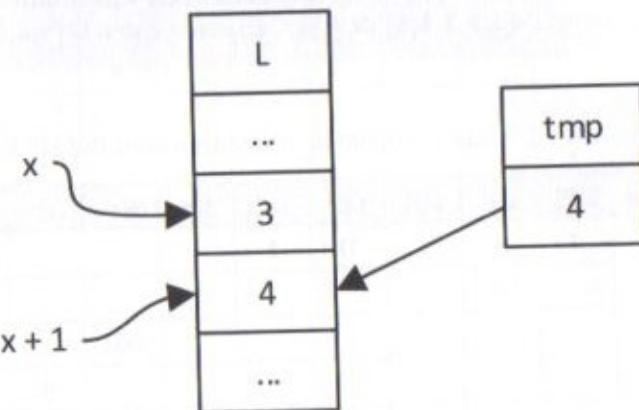


Figura 6.5 – Troca passo a passo. Terceira etapa.

Você pode entender melhor o processo de troca de valores utilizando um problema do dia a dia: imagine que você tenha duas xícaras, uma com leite e outra com café. O problema consiste em passar o leite para a xícara que contém o café, e o café para a xícara que contém o leite, ou seja, você deve trocar o conteúdo das xícaras. Nesse caso, resolveríamos o problema utilizando uma terceira xícara para facilitar a operação. Fizemos o mesmo durante a ordenação, onde chamamos nossa terceira variável de `temp`.

Quando a repetição iniciada em ❺ termina, o maior elemento está posicionado na última posição da lista, ou seja, em sua posição correta. Em ❾, verificamos se algo foi trocado na repetição anterior. Se não trocamos nada de lugar, nossa lista está ordenada, e não precisamos executar a repetição outra vez, por isso o `break`. Caso contrário, como a última posição já está com o elemento correto, diminuiremos o valor de `fim` para que não precisemos mais verificá-lo.

Vamos rastrear o algoritmo e observar como tudo isso funciona. Observe o rastreamento completo na tabela 6.1. A coluna linha indica a linha do programa sendo executada: não confunda com os números dentro dos círculos brancos da listagem.

**Exercício 6.14** O que acontece quando a lista já está ordenada? Rastreie o programa da listagem 6.44, mas com a lista  $L=[1, 2, 3, 4, 5]$ .

**Exercício 6.15** O que acontece quando dois valores são iguais? Rastreie o programa da listagem 6.44, mas com a lista  $L=[3, 3, 1, 5, 4]$ .

**Exercício 6.16** Modifique o programa da listagem 6.44 para ordenar a lista em ordem decrescente.  $L=[1, 2, 3, 4, 5]$  deve ser ordenada como  $L=[5, 4, 3, 2, 1]$ .

Tabela 6.1 – Rastreamento da ordenação com Bubble Sort

Linha	L[0]	L[1]	L[2]	L[3]	L[4]	fim	trocou	x	temp
1	7	4	3	12	8				
2						5			
4							False		
5								0	
8							True		
9									7
10	4								
11		7							
12						1			
8							True		
9									7
10		3							
11			7						
12							2		
12								3	

Linha	L[0]	L[1]	L[2]	L[3]	L[4]	fim	trocou	x	temp
8									True
9									12
10						8			
11								12	
12									4
15									4
4									False
5									0
8									True
9									4
10			3						
11				4					1
12									2
12									3
12							3		
15									False
4									0
5									1
12									2
12									

## 6.17 Dicionários

Dicionários consistem em uma estrutura de dados similar às listas, mas com propriedades de acesso diferentes. Um dicionário é composto por um conjunto de chaves e valores. O dicionário em si consiste em relacionar uma chave a um valor específico.

Em Python, criamos dicionários utilizando chaves ({}). Cada elemento do dicionário é uma combinação de chave e valor. Vejamos um exemplo onde os preços de mercadorias sejam como os da tabela 6.2.

Tabela 6.2 – Preços de mercadorias

Produto	Preço
Alface	R\$ 0,45
Batata	R\$ 1,20
Tomate	R\$ 2,30
Feijão	R\$ 1,50

A tabela 6.2 pode ser vista como um dicionário, onde chave seria o produto; e valor, seu preço. Vejamos como criar esse dicionário em Python na listagem 6.45.

#### ► Listagem 6.45 – Criação de um dicionário

```
tabela = { "Alface": 0.45,
           "Batata": 1.20,
           "Tomate": 2.30,
           "Feijão": 1.50 }
```

Um dicionário é acessado por suas chaves. Para obter o preço da alface, digite no interpretador, depois de ter criado a tabela, `tabela["Alface"]`, onde `tabela` é o nome da variável do tipo dicionário, e “Alface” é nossa chave. O valor retornado é o mesmo que associamos na tabela, ou seja, 0,45.

Diferentemente de listas, onde o índice é um número, dicionários utilizam suas chaves como índice. Quando atribuímos um valor a uma chave, duas coisas podem ocorrer:

1. Se a chave já existe: o valor associado é alterado para o novo valor.
2. Se a chave não existe: a nova chave será adicionada ao dicionário.

Observe a listagem 6.46. Em ①, acessamos o valor associado à chave “Tomate”. Em ②, alteramos o valor associado à chave “Tomate” para um novo valor. Observe que o valor anterior foi perdido. Em ③, criamos uma nova chave, “Cebola”, que é adicionada ao dicionário. Veja também como Python imprime o dicionário. Outra diferença entre dicionários e listas é que, ao utilizarmos dicionários, perdemos a noção de ordem. Observe que, durante a manipulação do dicionário, a ordem das chaves foi alterada.

#### ► Listagem 6.46 – Funcionamento do dicionário

```
>>> tabela = { "Alface": 0.45,
...             "Batata": 1.20,
...             "Tomate": 2.30,
...             "Feijão": 1.50 }
>>> print(tabela["Tomate"]) ①
2.3
>>> print(tabela)
{'Batata': 1.2, 'Alface': 0.45, 'Tomate': 2.3, 'Feijão': 1.5}
>>> tabela["Tomate"] = 2.50 ②
>>> print(tabela["Tomate"])
2.5
>>> tabela["Cebola"] = 1.20 ③
>>> print(tabela)
{'Batata': 1.2, 'Alface': 0.45, 'Tomate': 2.5, 'Cebola': 1.2, 'Feijão': 1.5}
```

Quanto ao acesso aos dados, temos que verificar se uma chave existe, antes de acessá-la (Listagem 6.47).

#### ► Listagem 6.47 – Acesso a uma chave inexistente

```
>>> tabela = { "Alface": 0.45,
...             "Batata": 1.20,
...             "Tomate": 2.30,
...             "Feijão": 1.50 }
>>> print(tabela["Manga"])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'Manga'
```

Se a chave não existir, uma exceção do tipo `KeyError` será ativada. Para verificar se uma chave pertence ao dicionário, podemos usar o operador `in` (Listagem 6.48).

#### ► Listagem 6.48 – Verificação da existência de uma chave

```
>>> tabela = { "Alface": 0.45,
...             "Batata": 1.20,
...             "Tomate": 2.30,
...             "Feijão": 1.50 }
```

```
>>> print("Manga" in tabela)
False
>>> print("Batata" in tabela)
True
```

Podemos também obter uma lista com as chaves do dicionário, ou mesmo uma lista dos valores associados (Listagem 6.49).

#### ► Listagem 6.49 – Obtenção de uma lista de chaves e valores

```
>>> tabela = { "Alface": 0.45,
...             "Batata": 1.20,
...             "Tomate": 2.30,
...             "Feijão": 1.50 }
>>> print(tabela.keys())
dict_keys(['Batata', 'Alface', 'Tomate', 'Feijão'])
>>> print(tabela.values())
dict_values([1.2, 0.45, 2.3, 1.5])
```

Observe que os métodos `keys()` e `values()` retornam geradores. Você pode utilizá-los diretamente dentro de um `for` ou transformá-los em lista usando a função `list`. Vejamos um programa que utiliza dicionários para exibir o preço de um produto na listagem 6.50.

#### ► Listagem 6.50 – Obtenção do preço com um dicionário

```
tabela = { "Alface": 0.45,
           "Batata": 1.20,
           "Tomate": 2.30,
           "Feijão": 1.50 }

while True:
    produto=input("Digite o nome do produto, fim para terminar:")
    if produto == "fim":
        break
    if produto in tabela: ❶
        print("Preço %.2f" % tabela[produto]) ❷
    else:
        print("Produto não encontrado!")
```

Em ❶ verificamos se o dicionário contém a chave procurada. Em caso afirmativo, imprimimos o preço associado à chave, ou haverá uma mensagem de erro.

Para apagar uma chave, utilizaremos a instrução `del` (Listagem 6.51).

#### ► Listagem 6.51 – Exclusão de uma associação do dicionário

```
>>> tabela = { "Alface": 0.45,
...             "Batata": 1.20,
...             "Tomate": 2.30,
...             "Feijão": 1.50 }
>>> del tabela["Tomate"]
>>> print(tabela)
{'Batata': 1.2, 'Alface': 0.45, 'Feijão': 1.5}
```

Você pode estar-se perguntando quando utilizar listas e quando utilizar dicionários. Tudo depende do que você deseja realizar. Se seus dados são facilmente acessados por suas chaves, quase nunca você precisa acessá-los de uma só vez: um dicionário é mais interessante. Além disso, você pode acessar os valores associados a uma chave rapidamente sem pesquisar. A implementação interna de dicionários também garante uma boa velocidade de acesso quando temos muitas chaves. Porém, um dicionário não organiza suas chaves, ou seja, as primeiras chaves inseridas nem sempre serão as primeiras na lista de chaves. Se seus dados precisam preservar a ordem de inserção (como em filas ou pilhas, continue a usar listas), dicionários não serão uma opção.

## 6.18 Dicionários com listas

Em Python, podemos ter dicionários nos quais as chaves são associadas a listas ou mesmo a outros dicionários. Imagine uma relação de estoque de mercadorias onde teríamos, além do preço, a quantidade em estoque (Listagem 6.52).

#### ► Listagem 6.52 – Dicionário com listas

```
estoque = { "tomate": [ 1000, 2.30 ],
            "alface": [ 500, 0.45 ],
            "batata": [ 2001, 1.20 ],
            "feijão": [ 100, 1.50 ] }
```

Nesse caso, o nome do produto é a chave, e a lista consiste nos valores associados, uma lista por chave. O primeiro elemento da lista é a quantidade disponível; e o segundo, o preço do produto.

Uma aplicação seria processarmos uma lista de operações e calcular o preço total de venda, atualizando também a quantidade em estoque.

#### ► Listagem 6.53 – Exemplo de dicionário com estoque e operações de venda

```
estoque = { "tomate": [ 1000, 2.30],
            "alface": [ 500, 0.45],
            "batata": [ 2001, 1.20],
            "feijão": [ 100, 1.50] }

venda = [ ["tomate", 5], ["batata", 10], ["alface", 5] ]
total = 0

print("Vendas:\n")
for operação in venda:
    produto, quantidade = operação ①
    preço = estoque[produto][1] ②
    custo = preço * quantidade
    print("%12s: %3d x %6.2f = %6.2f" %
          (produto, quantidade, preço, custo))
    estoque[produto][0] -= quantidade ③
    total += custo

print("Custo total: %21.2f\n" % total)
print("Estoque:\n")
for chave, dados in estoque.items(): ④
    print("Descrição: ", chave)
    print("Quantidade: ", dados[0])
    print("Preço: %6.2f\n" % dados[1])
```

Em ① utilizamos uma operação de desempacotamento, como já fizemos com `for` e `enumerate`. Como `operação` é uma lista com dois elementos, ao escrevermos `produto, quantidade` temos o primeiro elemento de `operação` atribuído a `produto`; e o segundo, a `quantidade`. Dessa forma, a construção é equivalente a:

```
produto = operação[0]
quantidade = operação[1]
```

Em ②, utilizamos o conteúdo de `produto` como chave no dicionário `estoque`. Como nossos dados são uma lista, escolhemos o segundo elemento, que armazena o preço do referido produto. Observe que atribuir nomes a cada um desses componentes facilita a leitura do programa.

Imagine escrever:

```
preço = estoque[operação[0]][1]
```

Em ③, atualizamos a quantidade em estoque subtraindo a quantidade vendida do estoque atual.

Já em ④ utilizamos o método `items` do objeto dicionário. O método `items` retorna uma tupla contendo a chave e o valor de cada item armazenado no dicionário. Usando um `for` com duas variáveis, `chave` e `dados`, efetuamos o desempacotamento desses valores em uma só passagem. Para entender melhor como isso acontece, experimente alterar o programa para exibir o valor de `chave` e `dados` a cada iteração.

**Exercício 6.17** Altere o programa da listagem 6.53 de forma a solicitar ao usuário o produto e a quantidade vendida. Verifique se o nome do produto digitado existe no dicionário, e só então efetue a baixa em estoque.

**Exercício 6.18** Escreva um programa que gere um dicionário, onde cada chave seja um caractere, e seu valor seja o número desse caractere encontrado em uma frase lida.

Exemplo: O rato -> { "O":1, "r":1, "a":1, "t":1, "o":1}

## 6.19 Tuplas

Tuplas podem ser vistas como listas em Python, com a grande diferença de serem imutáveis. Tuplas são ideais para representar listas de valores constantes e também para realizar operações de empacotamento e desempacotamento de valores. Primeiramente, vejamos como criar uma tupla.

Tuplas são criadas de forma semelhante às listas, mas utilizamos parênteses em vez de colchetes. Por exemplo:

```
>>> tupla = ("a", "b", "c")
>>> tupla
('a', 'b', 'c')
```

Tuplas suportam a maior parte das operações de lista, como fatiamento e indexação.

```
>>> tupla[0]
'a'
>>> tupla[2]
```

```
'c'
>>> tupla[1:]
('b', 'c')
>>> tupla * 2
('a', 'b', 'c', 'a', 'b', 'c')
>>> len(tupla)
3
```

Mas tuplas não podem ter seus elementos alterados. Veja o que acontece se tentarmos alterar uma tupla:

```
>>> tupla[0] = "A"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

Várias funções utilizam ou geram tuplas em Python. Tuplas podem ser utilizadas com `for`:

```
>>> for elemento in tupla:
...     print(elemento)
...
a
b
c
```

Python também permite criar tuplas usando valores separados por vírgula, independente de usarmos parênteses:

```
>>> tupla = 100, 200, 300
>>> tupla
(100, 200, 300)
```

No caso, 100, 200 e 300 foram convertidos em uma tupla com três elementos. Esse tipo de operação é chamado de empacotamento.

Tuplas também podem ser utilizadas para desempacotar valores, por exemplo:

```
>>> a, b = 10, 20
>>> a
10
>>> b
20
```

Onde o primeiro valor, 10, foi atribuído à primeira variável `a` e 20 a segunda, `b`. Esse tipo de construção é interessante para distribuirmos o valor de uma tupla em várias variáveis.

Também podemos trocar rapidamente os valores de variáveis com construções do tipo:

```
>>> a, b = 10, 20
>>> a, b = b, a
>>> a
20
>>> b
10
```

Onde a tupla da esquerda foi usada para atribuir os valores à direita. Nesse caso, as atribuições: `a=b` e `b = a` foram realizadas imediatamente, sem precisar utilizarmos uma variável intermediária para a troca.

A sintaxe do Python é um tanto especial quando precisamos criar tuplas com apenas um elemento. Como os valores são escritos entre parênteses, quando apenas um valor estiver presente, devemos acrescentar uma vírgula para indicar que o valor é uma tupla com apenas um elemento. Veja o que acontece, usando e não usando a vírgula:

```
>>> t1 = (1)
>>> t1
1
>>> t2 = (1,)
>>> t2
(1,)
>>> t3 = 1,
>>> t3
(1,)
```

Veja que em `t1` não utilizamos a vírgula, e o código foi interpretado como um número inteiro entre parênteses. Já em `t2`, utilizamos a vírgula, e nossa tupla foi corretamente construída. Em `t3`, criamos outra tupla, mas nesse caso nem precisamos usar parênteses.

Podemos também criar tuplas vazias, escrevendo apenas os parênteses:

```
>>> t4=()
>>> t4
()
>>> len(t4)
0
```

Tuplas também podem ser criadas a partir de listas, utilizando-se a função `tuple`:

```
>>> L=[1,2,3]
>>> T=tuple(L)
>>> T
(1, 2, 3)
```

Embora não possamos alterar uma tupla depois de sua criação, podemos concatená-las, gerando novas tuplas:

```
>>> t1=(1,2,3)
>>> t2=(4,5,6)
>>> t1+t2
(1, 2, 3, 4, 5, 6)
```

Observe que se uma tupla contiver uma lista ou outro objeto que pode ser alterado, este continuará a funcionar normalmente. Veja o exemplo de uma tupla que contém uma lista:

```
>>> tupla=("a", ["b", "c", "d"])
>>> tupla
('a', ['b', 'c', 'd'])
>>> len(tupla)
2
>>> tupla[1]
['b', 'c', 'd']
>>> tupla[1].append("e")
>>> tupla
('a', ['b', 'c', 'd', 'e'])
```

Neste caso, nada mudou na tupla em si, mas na lista que é seu segundo elemento. Ou seja, a tupla não foi alterada, mas a lista que ela continha, sim.

## CAPÍTULO 7

# Trabalhando com strings

No capítulo 3, vimos que podemos acessar strings como listas, mas também falamos que strings são imutáveis em Python. Vejamos o que acontece na listagem 7.1.

### ► Listagem 7.1 – Alteração de uma string

```
>>> S="Alô mundo"
>>> print(S[0])
A
>>> S[0]="a"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```

Se quisermos trabalhar caractere a caractere com uma string, alterando seu valor, teremos que primeiramente transformá-la em uma lista (Listagem 7.2).

### ► Listagem 7.2 – Convertendo uma string em lista

```
>>> L=list("Alô Mundo")
>>> L[0]="a"
>>> print(L)
['a', 'l', 'ô', ' ', 'M', 'u', 'n', 'd', 'o']
>>> s="".join(L)
>>> print(s)
alô Mundo
```

A função `list` transforma cada caractere da string em um elemento da lista retornada. Já o método `join` faz a operação inversa, transformando os elementos da lista em string.