

3. Criação de uma progressão aritmética

Escreva um programa que leia três dados de entrada: o primeiro termo, a razão e a quantidade de termos de uma P.A., todos números inteiros. O programa deve calcular todos os termos, colocando-os em uma lista, e exibi-la no final.

Listas vinculadas

A seguir é abordada uma característica das listas. Em Python é possível utilizar o operador de atribuição “=” para atribuir uma lista existente a outro identificador. No Exemplo 4.17 foi criada a lista L com cinco elementos e, em seguida, foi utilizado o operador de atribuição para vincular V a L: `V = L`. De fato, o termo correto aqui é **vincular** e não **copiar**. Observe as linhas seguintes do exemplo: o primeiro elemento de V foi alterado de 2 para 15 e a lista L também foi alterada.

Isso ocorre porque, ao utilizar o operador de atribuição “=”, o efeito produzido foi fazer que o novo identificador V aponte para os mesmos dados contidos em L, ou seja, após o comando `V = L`, os dois identificadores passam a ter o mesmo id, portanto, fazendo referência ao mesmo local da memória onde está armazenado o conteúdo da lista.

Caso precise de uma cópia da lista original, deve utilizar o método `copy`, como mostrado no final deste exemplo.

Exemplo 4.17 Listas vinculadas

```
>>> L = [2, 4, 6, 8, 10]

>>> V = L # cria o vínculo entre V e L

>>> V

[2, 4, 6, 8, 10]

>>> V[0] 2
```



```
>>> V[0] = 15 # de modo que ao alterar V altera-se
```

```
>>> V # L e vice-versa [15, 4, 6, 8, 10]
```

```
>>> L
```

```
[15, 4, 6, 8, 10]
```

```
>>> id(L) # |
```

```
48849904 # | Verifique que L e V
```

```
>>> id(V) # | tem o mesmo id 48849904 # |
```

```
>>> C = L.copy() # cria a lista C como uma cópia de L
```

```
>>> C
```

```
[15, 4, 6, 8, 10]
```

```
>>> id(C) # C tem outro id 48889048
```

```
>>> C[0] = 2 # alterações em C não alteram L
```

```
>>> C # e vice-versa [2, 4, 6, 8, 10]
```

```
>>> L
```

```
[15, 4, 6, 8, 10]
```

Listas aninhadas

Foi dito anteriormente que uma lista pode conter qualquer tipo de objeto disponível em Python. No entanto, até o momento foram apresentados exemplos de listas contendo números e strings. Aqui, o objetivo é mostrar outras opções, tais como listas e tuplas dentro de listas. O termo “listas aninhadas” (em inglês, *nesting*) é utilizado para a situação em que há listas dentro de uma lista.

No Exemplo 4.18 a lista L é uma lista aninhada que contém, inicialmente,



duas sublistas (termo informal que aqui será utilizado para referenciar as listas que estão contidas em outra lista).

Para se ter acesso a um elemento de uma lista aninhada, é preciso utilizar dois índices entre colchetes e posicionados lado a lado da seguinte maneira: `L[i][j]`. O primeiro índice, `i`, seleciona a sublista `i` e o segundo índice, `j`, seleciona o elemento `j` dentro da sublista `i`. Com esse recurso, é possível implementar matrizes em programas escritos em Python.

Exemplo 4.18 Listas aninhadas

```
>>> L = [[1, 2, 3], [4, 5, 6]] # L é uma lista aninhada
```

```
>>> L[0] [1, 2, 3]
```

```
>>> type(L[0]) # L[0] é o primeiro elemento de L
```

```
<class 'list'> # e é uma lista
```

```
>>> L[1] [4, 5, 6]
```

```
>>> type(L[1]) # L[1] é o segundo elemento de L
```

```
<class 'list'> # e também é uma lista
```

```
>>> L[0][0] # primeiro elemento da sublista 0 1
```

```
>>> L[1][0] # primeiro elemento da sublista 1
```

```
>>> L[1][2] # terceiro elemento da sublista 1 6
```

```
>>> A = [7, 8, 9, 10] # seja a matriz A com 4 elementos
```

```
>>> L.append(A) # pode-se usar append e incluir a em L
```

```
>>> L # L fica assim
```

```
[[1, 2, 3], [4, 5, 6], [7, 8, 9, 10]]
```



É possível utilizar o método *append* de uma lista para acrescentar a ela uma nova lista, como está exemplificado no final do Exemplo 4.18.

Além disso, as sublistas podem ser de variados tamanhos e conteúdos. Tudo pode ser aninhado em Python, conforme a vontade e a necessidade do programador.

Outra característica é que o grau de aninhamento não tem limite de profundidade, ou seja, é possível ter uma lista, dentro de outra lista, dentro de outra maior ainda, e assim por diante, tantos níveis quantos forem necessários, para resolver algum problema computacional. O Exemplo 4.19 mostra isso, com a lista L, que apresenta grau de profundidade 4.

Exemplo 4.19 Listas aninhadas com profundidade 4

```
>>> L = [[1, 2, ['3.1', '3.2', ['3.3.1', ['3.3.2.1', '3.3.2.2'], '3.3.3']]], [4,5, 6]]

>>> L[0]
[1, 2, ['3.1', '3.2', ['3.3.1', ['3.3.2.1', '3.3.2.2'], '3.3.3']]]

>>> L[0][2]

['3.1', '3.2', ['3.3.1', ['3.3.2.1', '3.3.2.2'], '3.3.3']]

>>> L[0][2][2]

['3.3.1', ['3.3.2.1', '3.3.2.2'], '3.3.3']

>>> L[0][2][2][1] # a lista mais aninhada tem 4 índices ['3.3.2.1', '3.3.2.2']

>>>
```

6. Programa para criar e exibir uma matriz

Escreva um programa que leia dois números inteiros Lin e Col, que representam, respectivamente, a quantidade de linhas e colunas em uma matriz.



Utilizando listas aninhadas, crie uma representação para essa matriz, utilizando a função *randint* para gerar números para cada posição da matriz. Apresente-a na tela com uma aparência matricial.

Tuplas

A tupla é um tipo sequencial em muitos aspectos semelhante à lista, porém, imutável como um string. As tuplas são definidas, atribuindo uma lista de dados separados por vírgulas ou, como é mais frequente, encapsulando os dados em parênteses. Uma vez criada, a tupla não pode ser modificada.

Tuplas são capazes de conter quaisquer outros tipos definidos em Python, números, strings, listas, outras tuplas etc. Como são sequenciais, o acesso aos elementos se dá por meio de índices, para os quais valem as mesmas regras de strings e listas. Aceitam os operadores de concatenação “+” e multiplicativo “*” e aplicam-se a elas as operações de fatiamento.

As tuplas são muito utilizadas para encapsular o retorno de funções. Esse assunto será visto em detalhes no Capítulo 5.

Operações básicas com tuplas

O Exemplo 4.21 ilustra as operações básicas frequentemente realizadas com tuplas.

Exemplo 4.21 Operações básicas em tuplas

```
>>> V = () # define uma tupla vazia
```

```
>>> V
```

```
()
```

```
>>> len(V) # o tamanho de V é zero 0
```

```
>>> P = 3, 6, 9 # define uma tupla
```

```
>>> P (3, 6, 9)
```

```
>>> T = (17, 3, 'txt', 3.8) # define uma tupla
```



```
>>> type(T)
```

```
<class 'tuple'>
```

```
>>> len(T) # contém 4 elementos 4
```

```
>>> T[0] 17
```

```
>>> T[2]
```

```
'txt'
```

```
>>> T[3] 3.8
```

```
>>> T = T + (15, 16) # concatenação com outra tupla
```

```
>>> T
```

```
(17, 3, 'txt', 3.8, 15, 16)
```

```
>>> P = (0, 1)
```

```
>>> P = P * 6 # uso do operador multiplicativo '*'
```

```
>>> P
```

```
(0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1)
```

É necessário que o programador tenha atenção e tome cuidado ao definir uma tupla que contenha um único elemento numérico. O Exemplo 4.22 mostra que, ao se tentar definir uma tupla com um único valor, define-se um objeto de tipo "int". Isso se deve ao fato de que os parênteses também são utilizados para expressões aritméticas como $(2 + X) * 2$. O Python, haja vista seu esquema de prioridades, interpreta a expressão (14) como uma expressão aritmética, e não como uma tupla. Para se definir corretamente uma tupla com um único objeto, deve-se usar um caractere vírgula ",",



dentro dos parênteses e após o objeto, como mostrado.

Exemplo 4.22 Detalhes sobre as tuplas

```
>>> T = (14) # essa sintaxe define um 'int' e não
```

```
>>> type(T) # uma tupla
```

```
<class 'int'>
```

```
>>> T = (14,) # desta forma se define uma tupla
```

```
>>> type(T) # contendo um único objeto
```

```
<class 'tuple'>
```

Atribuições envolvendo tuplas

O operador de atribuição tem sido largamente utilizado neste livro, sem que em qualquer momento tenha sido explicado de maneira formal. Trata-se de uma operação um tanto óbvia e intuitiva, e por esse motivo tal falta de formalização pode ter passada despercebida.

Quando se escreve $X = 10$, está se atribuindo o conteúdo 10 ao identificador X. Quando se escreve $Y = (2 + X) / 3$, o resultado da expressão aritmética está sendo calculado e atribuído ao identificador Y. E, ao escrever $L = [9, 18, 27]$, está sendo criada uma lista com três elementos e atribuída ao identificador L. Em expressões assim, deve começar sua leitura pelo lado direito, pois é ele que gera o objeto resultante em memória. Em seguida, o identificador explicitado do lado esquerdo é aplicado a esse objeto.

É utilizando esse mecanismo que as atribuições são processadas pelo interpretador Python. O mesmo ocorre quando se trata de tuplas, porém, com elas há uma característica que difere dos demais tipos de objetos da linguagem.

As tuplas podem estar **dos dois lados** da operação de atribuição. A primeira atribuição feita no Exemplo 4.23 mostra esse conceito. Seu lado direito contém uma tupla de números inteiros definida sem o uso de parênteses. E o lado esquerdo contém



uma tupla de identificadores. A associação é feita segundo a sequência com que os identificadores e os valores aparecem na expressão. Assim, o identificador a recebe o valor 10, b recebe 15 e c recebe 20. Para que essa atribuição múltipla funcione, é necessário que em ambos os lados da expressão haja o mesmo número de identificadores e valores.

O Exemplo 4.23 mostra outras situações. Observe-as com atenção. O último caso, em particular, mostra como fazer a inversão de dados entre duas variáveis, algo muito comum em programas. A expressão `a, b = b, a` faz que o valor contido em b seja colocado em a, e vice-versa.

Exemplo 4.23 Expressões de atribuição múltipla

```
>>> a, b, c = 10, 15, 20 # atribuição múltipla

>>> a 10
>>> b 15
>>> c 20
>>> d, e = a*2, b*3 # do lado esquerdo são permitidas

>>> d # expressões aritméticas quaisquer 20 # segundo as regras válidas para
>>> e # esse tipo de expressão 45

>>>

>>> x = 5 # x contém o valor 5
>>> x, y = 10, x*3 # neste caso x receberá 10 e

>>> x # y receberá 5 multiplicado por 3 10 # ou seja, é usado o valor anterior
>>> y # de x, e não o novo valor 15
>>> L, m = [3, 6, 9], 14 # L recebe uma lista e m recebe o

>>> L # número inteiro 14 [3, 6, 9]
>>> m 14
>>> a, b, c = 2, 4, 6, 8 # gera erro, quantidades são diferentes Traceback (most
```



recent call last):

```
File "<pyshell#238>", line 1, in <module> a, b, c = 2, 4, 6, 8
```

```
ValueError: too many values to unpack (expected 3)
```

```
>>> a, b = 17, -9 # a recebe 17 e b recebe -9
```

```
>>> a, b = b, a # inverte os valores de a e b
```

```
>>> a
```

```
-9
```

```
>>> b 17
```

Tuplas são imutáveis

No entanto, ao se tentar alterar um objeto escrevendo uma expressão do tipo `T[0] = 29`, será gerado um erro.

```
>>> T[0] = 29 # erro ao tentar alterar o elemento
Traceback (most recent call last):
```

```
File "<pyshell#148>", line 1, in <module> T[0] = 29
```

```
TypeError: 'tuple' object does not support item assignment
```

Dada essa característica de imutabilidade, muitos programadores que estão aprendendo Python questionam sua utilidade. Na realidade, elas não são tão utilizadas como as listas, porém, sua característica de imutabilidade tem sua importância. É muito comum nos programas que dados sejam passados de um módulo para outro. Se isso for feito com listas, dado que são mutáveis, elas podem acabar sendo alteradas em algum ponto. Ao fazer essas mesmas transferências utilizando tuplas, sua imutabilidade garante uma integridade e consequente consistência. Essa garantia de integridade é bastante conveniente nos programas em geral e, em particular, naqueles que são muito grandes e contém muitos módulos.

Listas aninhadas em tuplas

Por serem capazes de conter objetos de qualquer outro tipo, as tuplas podem



contar com uma ou mais listas aninhadas. Quando isso ocorre, embora a tupla seja imutável, a lista aninhada continua sendo mutável. O Exemplo 4.24 ilustra esse aspecto. Nele foi definida a tupla P, cujo terceiro elemento – P[2] – é uma lista. Pode-se alterar os elementos da lista aninhada, porém, não é possível remover a lista de dentro da tupla.

Exemplo 4.24 Listas aninhadas em Tuplas

```
>>> P = (14, 26, [0, 0, 0], 31)

>>> P[2] [0, 0, 0]
>>> type(P[2]) # o terceiro elemento de P é uma lista

<class 'list'>

>>> P[2][1] = 39 # então é possível alterar um elemento

>>> P # da lista P[2] (14, 26, [0, 39, 0], 31)
>>> P[2].append(16) # é possível aumentar a lista P[2]

>>> P

(14, 26, [0, 39, 0, 16], 31)
>>> P[2].remove(0) # é possível diminuí-la

>>> P

(14, 26, [39, 0, 16], 31)

>>> P[2].clear() # e até mesmo limpá-la

>>> P
```



(14, 26, [], 31)

porém não é possível alterar o objeto

```
>>> P[2] = 'outra coisa' # P[2] da tupla P para outro tipo
Traceback (most recent call last):
```

```
File "<pyshell#198>", line 1, in <module> P[2] = 'outra coisa'
```

```
TypeError: 'tuple' object does not support item assignment
```

Tuplas como registros (records) que contêm dados

Outro uso para as tuplas é tratá-las como um repositório de dados inter-relacionados. Neste livro, será empregado o termo “registro” para fazer referência a esse tipo de construção.

Por hipótese, imagine-se um conjunto de dados inter-relacionado, ou seja, imagine-se um registro que contenha o código de um produto, seu nome, a quantidade em estoque e o preço unitário de compra. Esse conjunto pode ser representado pela tupla exibida na primeira linha do Exemplo 4.25. Esse conjunto pode ser utilizado de muitas maneiras: pode ser passado para uma função, pode retornar de uma função, pode ser incluído em uma lista, pode ser gravado em ou lido de um arquivo etc. São muitas as possibilidades. Vejam as linhas seguintes do exemplo, em que são montados mais dois conjuntos, e todos são inseridos em uma lista L. Ao fazer isso, a lista está se tornando um banco de dados de produtos. E fica claro que, assim como é possível inserir esses registros em uma lista, também é possível usar outra tupla no lugar de tal lista, caso se queira.

Exemplo 4.25 Uso de tupla como registro

```
>>> P = (12336, 'Sabão', 1337, 1.37)
```

```
>>> L = []
```

```
>>> L.append(P)
```



```
>>> P = (13446, 'Arroz 1kg', 3554, 2.65)
```

```
>>> L.append(P)
```

```
>>> P = (13956, 'Fubá 500g', 439, 1.19)
```

```
>>> L.append(P)
```

```
>>> L
```

```
[(12336, 'Sabão', 1337, 1.37), (13446, 'Arroz 1kg', 3554, 2.65), (13956,  
'Fubá 500g', 439, 1.19)]
```

Uma vez construído esse conjunto de dados contido na lista, podem-se recuperá-los e usá-los de várias maneiras. O Exemplo 4.26 mostra uma possibilidade dentre muitas.

Exemplo 4.26 Uso de tupla como registro

```
>>> L # seja a lista L carregada no Exemplo 4.25
```

```
[(12336, 'Sabão', 1337, 1.37), (13446, 'Arroz 1kg', 3554, 2.65), (13956,  
'Fubá 500g', 439, 1.19)]
```

deseja-se recuperar os dados do segundo elemento de L (L[1]) # fazendo que variáveis separadas recebam os valores contidos # na tupla. Então, tem-se:

```
>>>Codigo, Nome, Qtde, PcUnit = L[1]
```

```
>>>Codigo 13446
```

```
>>>Nome 'Arroz 1kg'
```

```
>>>Qtde 3554
```

```
>>>PcUnit 2.65
```



O tipo *range*

Muitos textos sobre a linguagem Python fazem referência a *range* como uma função, quando, na verdade, segundo a documentação oficial – Seção 4.6.6 do Capítulo 4 da The Python Standard Library – trata-se de um tipo sequencial imutável. Por esse motivo, neste livro será utilizada a expressão tipo *range* em vez de função *range*.

Basicamente, o tipo *range* produz uma sequência imutável contendo números inteiros, sendo comumente empregada em laços implementados com o comando *for*, que será visto a seguir. Pode-se afirmar que ele é um gerador de números inteiros que seguem uma regra de progressão aritmética definida pelos parâmetros utilizados. A sintaxe para uso desse tipo tem duas opções, em que a diferença são os parâmetros necessários.

1. `class range(stop)`
2. `class range(start, stop, [step])`

Na opção 1 é fornecido apenas o limite final da progressão aritmética, assumindo-se que o valor inicial é 0 e o incremento é 1.

Na opção 2 são fornecidos o valor inicial e o limite final. Opcionalmente, pode ser fornecido também um terceiro parâmetro: o passo.

Em ambos os casos, os parâmetros devem, obrigatoriamente, ser números inteiros. Quanto a terminologia utilizada, há uma diferença entre os termos valor e limite empregados nas definições anteriores. O valor inicial sempre estará incluído na sequência gerada, ao passo que o limite final nunca estará incluído.

No Exemplo 4.27 são mostrados vários casos de uso do *range*. Ao utilizá-lo no IDLE, é preciso converter seu retorno para uma lista utilizando a sintaxe: `list(range(...))`

Exemplo 4.27 Uso do tipo *range*

```
>>> list(range(10)) # forma do caso 1 [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> list(range(5, 15)) # forma do caso 2, omitindo o passo [5, 6, 7, 8, 9, 10, 11, 12, 13, 14]
>>> list(range(-3, 4)) # os parâmetros podem ser negativos [-3, -2, -1, 0, 1, 2, 3]
```



```
>>> list(range(5, 15, 3)) # forma do caso 2, com os 3 parâmetros
```

```
[5, 8, 11, 14]
```

```
>>> list(range(10, 0, -1)) # o passo pode ser negativo [10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
```

```
>>> list(range(6, -1, -2))
```

```
[6, 4, 2, 0]
```

```
>>> list(range(10, 3, 1)) # dados inconsistentes geram uma [] # sequência vazia
```

O comando for

Na introdução deste capítulo, foi dada a definição do conceito de objeto iterável. Strings, listas e tuplas são iteráveis, e agora é necessário mostrar como fazer uso deste conceito ao mesmo tempo simples e poderoso.

No Capítulo 3 foi apresentado o comando de laço while, e até aqui muitos programas exemplo foram escritos utilizando-o. Agora, será apresentado o comando for, que é uma segunda opção disponível em Python para a construção de laços de repetição.

Esse comando vale-se fortemente do conceito de iterável, pois é utilizado para a iteração sobre os tipos sequenciais. O Exemplo 4.27 mostra o uso do comando for iterando com a lista L. Os comandos subordinados ao for serão repetidos cinco vezes, pois a lista L contém cinco elementos. A cada repetição, o objeto x recebe um valor dentre os contidos em L, segundo a sequência da lista. Desse modo, na primeira vez x recebe 2, na segunda vez recebe 4, e assim sucessivamente, até o término do laço.

Isso é um laço iterador, muito utilizado na programação com Python. No lugar da lista L seria possível utilizar uma tupla, um string ou um conjunto (que ainda será visto).

Exemplo 4.28 Uso do comando for print("Inicio do Programa")

```
L = [2, 4, 6, 8, 10]
```

```
i = 0
```



for x in L:

print("Elemento {0} = {1}".format(i, x))

x = 0 # foi colocado 0 em x, mas isso não afeta o laço i+=1

print("Fim do Programa")



```
Python 3.6.3 Shell
File Edit Shell Debug Options Window Help
Início do Programa
Elemento 0 = 2
Elemento 1 = 4
Elemento 2 = 6
Elemento 3 = 8
Elemento 4 = 10
Fim do Programa
>>>
```

Suponha-se, agora, que por algum motivo o valor do objeto x tenha sido alterado durante a iteração. Por exemplo, como mostrado no Exemplo 4.28, após o print foi feito x = 0. Isso não afeta nem o laço nem a lista L. O valor de x pode, sim, ser alterado dessa maneira, porém, na próxima iteração x receberá o próximo valor contido em L.

Formato geral do comando for

Os laços em Python iniciam uma linha de cabeçalho na qual se especifica um objeto iterador (objctrl) que receberá, um a um, os valores contidos em um objeto de tipo sequencial (objseq), que será denominado **sequência iterável**. Para cada valor atribuído ao objeto de controle, os comandos contidos no <bloco 1> serão executados. Desse modo, o laço será executado um certo número de vezes, que depende exclusivamente da quantidade de elementos contidos em objseq.

Assim como o comando while, o for também suporta a opcional cláusula else, que funciona exatamente da mesma maneira, como já foi visto para o comando while. Se o laço terminar normalmente, sem que uma instrução break seja executada, então, o <bloco 2> de comandos será executado.

```
for objctrl in objseq:
    <bloco 1> else:
    <bloco 2>
```



Os comandos `break` e `continue` podem ser utilizados aqui do mesmo modo como visto para o `while`: `break` que termina o laço imediatamente e `continue` que termina a iteração atual e segue para a próxima.

Com relação aos motivos de existir a cláusula `else` em um comando de laço, valem exatamente as mesmas considerações feitas quando foi visto o comando `while`.

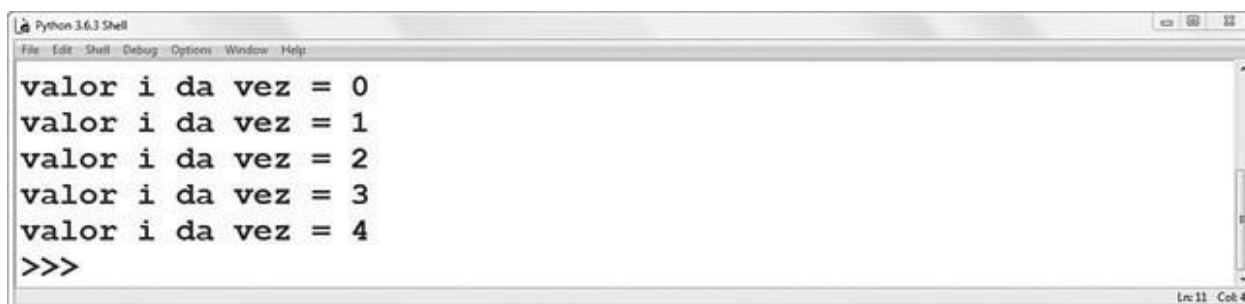
Exemplos de uso

A seguir serão vistos diversos exemplos de uso do comando `for`. O primeiro é o Exemplo 4.29, no qual o tipo `range` é utilizado para gerar a sequência iterável. Neste caso, o objeto iterador recebe os valores de 0 a 4 gerados por `range(5)`.

Exemplo 4.29 Uso do comando `for` em conjunto com `range`

```
for i in range(5):
```

```
    print("valor i da vez = {}".format(i))
```



O Exemplo 4.30 mostra a execução de um `for` que utiliza um string como sequência iterável. Para cada caractere contido no string é feita uma saída em tela, seguida de um caractere hífen “-”

Exemplo 4.30 Comando `for` em conjunto com string

```
S = 'Programe em Python' for x in S:
```

```
    print(x, end='-')
```

```
# Este código produz a saída
```

```
P-r-o-g-r-a-m-e- -e-m- -P-y-t-h-o-n-
```



O uso do comando `for` com tuplas seria semelhante ao já visto com listas, *range* e string. Porém, o uso de tuplas ou listas contendo outras tuplas abre possibilidades mais amplas. O Exemplo 4.31 contém uma tupla `T` constituída de três elementos que também são tuplas, cada uma com dois números. O comando `for` foi construído de modo que o iterador (`objctrl`) é uma tupla de objetos (`a`, `b`). Isso faz com que, a cada tupla contida em `T`, o objeto `a` receba um valor e `b` receba o outro.

Exemplo 4.31 Comando *for* em conjunto com uma tupla de tuplas `print("Início do Programa\n")`

```
T = ((3, 6), (5, 11), (7, 16))
```

```
for (a, b) in T: print(a, b)
```

```
print("\nFim do Progrma")
```



O Exemplo 4.32 mostra um caso semelhante ao 4.31, agora, ilustrando-o com a lista de produtos gerada no Exemplo 4.25, visto anteriormente. É possível notar o poder dessas iterações, uma vez que a lista constituída de tuplas fornece, a cada vez, um conjunto completo de dados sobre um produto. Dentro do bloco de comandos do laço é possível efetuar qualquer operação que seja necessária com esses dados. No caso, foi feita uma simples exibição em tela e foi calculado o valor total gasto com cada produto em estoque.

Os dados foram carregados de modo literal nas primeiras linhas desse programa. Essas linhas podem ser substituídas por uma leitura de arquivo em disco ou acesso a um banco de dados, que forneceriam os dados para o algoritmo.

Exemplo 4.32 Comando *for* em conjunto com uma tupla listas de tuplas `L = []`

```
P = (12336, 'Sabão', 1337, 1.37)
```

```
L.append(P)
```



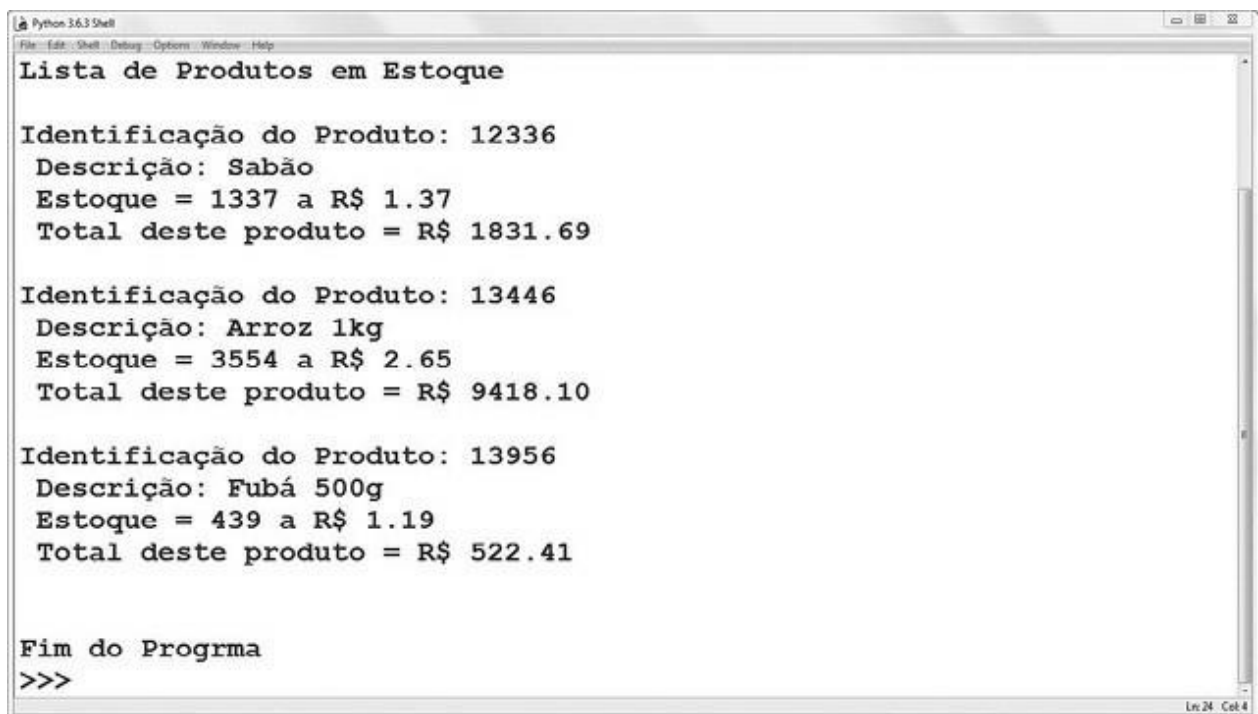
```
P = (13446, 'Arroz 1kg', 3554, 2.65)
```

```
L.append(P)
```

```
P = (13956, 'Fubá 500g', 439, 1.19)
```

```
L.append(P)
```

```
print("Lista de Produtos em Estoque\n") for (Cod, Nome, Qtde, PcUnit) in L:  
    print("Identificação do Produto:", Cod) print(" Descrição:", Nome)  
    print(" Estoque = {0} a R$ {1:.2f}".format(Qtde, PcUnit)) print(" Total deste  
produto = R$ {0:.2f}".format(Qtde*PcUnit)) print()  
print("\nFim do Progrma")
```



```
Python 3.6.3 Shell  
File Edit Shell Debug Options Window Help  
Lista de Produtos em Estoque  
  
Identificação do Produto: 12336  
Descrição: Sabão  
Estoque = 1337 a R$ 1.37  
Total deste produto = R$ 1831.69  
  
Identificação do Produto: 13446  
Descrição: Arroz 1kg  
Estoque = 3554 a R$ 2.65  
Total deste produto = R$ 9418.10  
  
Identificação do Produto: 13956  
Descrição: Fubá 500g  
Estoque = 439 a R$ 1.19  
Total deste produto = R$ 522.41  
  
Fim do Progrma  
>>>
```

Considerações finais sobre o comando for

As várias formas de uso observadas do comando for, não esgotam todas as possibilidades existentes, embora tenham sido abordadas as principais e mais frequentes.

A implementação desse comando é especialmente otimizada para efetuar as



iterações utilizando o iterador e a sequência iterável. Como consequência direta, tem-se que os laços for rodam mais rápido que laços construídos com o comando while. Muitas fontes de consulta disponíveis na bibliografia, por exemplo, Lutz (2009), recomendam que o programador privilegie o uso do comando for em detrimento do while, sempre que isso for possível.

Além disso, o código escrito com o comando for resulta em ser mais simples, de mais fácil leitura e menos sujeito a erros do programador, quando comparado ao código escrito utilizando-se while. Por exemplo, não é necessário que o programador tenha de controlar contadores de laço nem com a inicialização dos objetos com os quais tal contagem será feita. Na prática, esses contadores nem estarão no código se a escolha recair sobre o comando for.

Quando a sequência iterável for uma lista, o programador deve lembrar-se de que a lista é um objeto mutável e tomar cuidado com o que ocorre com ela durante as repetições do laço. Não é nada recomendável que a lista seja alterada durante o laço. Se isso ocorrer, podem ser verificados resultados incoerentes no processamento das repetições. Basta pensar que o laço está construído tendo por base uma lista que é alterada a cada iteração, o que leva a uma condição imprevisível e instável. Essa situação é indesejada e deve ser evitada sempre.

Essas considerações dizem respeito exclusivamente a Python. Em outras linguagens, os comandos de laço têm outras formas de implementação, de modo que cada caso é diferente do outro e precisa ser devidamente estudado pelo programador para que este descubra quais opções trazem as maiores vantagens.

Programa que gera uma lista com a sequência de Fibonacci

Busca sequencial de um valor em uma lista

Escreva um programa que leia um número inteiro N e gere uma lista com números pares de 2 até N. Se N for par, deve estar incluído na lista. Em seguida, inicie um laço que deve permanecer em execução enquanto x for diferente de zero. Para cada valor de x fornecido, o programa deve informar se x está ou não na lista.

A solução desse problema tem duas partes: a geração da lista e a pesquisa de x.



A primeira parte tem esta solução: `L = list(range(2, N+1, 2))`

Isso mesmo! Uma única linha de código é capaz de criar a lista pedida, pois se vale dos recursos disponíveis no tipo *range*.

A segunda parte será resolvida de duas maneiras:

a) Privilegiando o uso dos recursos de Python:

