Escola Nacional de Ciências Estatísticas

Apostila de Introdução à Programação
Prof. Eduardo Corrêa

Capítulo VII – Listas (parte 1)

Sumário

VII. Listas

As **coleções** – também chamadas de **estruturas de dados** (EDs) ou *containers* – são utilizadas nas linguagens de programação para organizar e armazenar conjuntos de dados relacionados, com o intuito de permitir com que estes sejam processados de forma eficiente por diferentes algoritmos. No papel de cientista de dados, você frequentemente precisará trabalhar com coleções para conseguir estudar ou transformar bases de dados.
Conforme o nome indica, uma coleção serve para armazenar **coleções de valores relacionados**, organizando-os sempre de alguma maneira "esperta" em diferentes células. Neste capítulo, você aprenderá a trabalhar com a **lista**, a mais conhecida e popular coleção da linguagem Python, que se destaca por sua flexibilidade e grande aplicabilidade prática. Este capítulo apresenta os conceitos básicos sobre listas, tais como criação, carga via teclado, acesso aos elementos, fatiamento, métodos para modificação de dados e outas operações.

VII.1 Muito prazer, me chamo lista!

No Capitulo IV aprendemos que uma **variável** pode armazenar apenas **um valor de cada vez**. Levando esta informação em consideração, imagine que você precisasse fazer um programa que inicialmente lê o nome, a idade e o bairro de 1000 clientes de uma loja. Considere que esse programa tem um menu de opções: se o usuário teclar 1 o programa emite um relatório dos clientes ordenados por bairro; se teclar 2 o relatório é emitido com a ordenação por nome. Caso este programa fosse implementado apenas com o uso de variáveis, seria preciso declarar nada menos do que 3000 delas (1000 para os nomes, 1000 para as idades e 1000 para os bairros), o que é completamente inviável. Esta é a típica situação em que se torna necessária a utilização de um recurso importante da linguagem Python: as **listas**.

Enquanto uma variável pode armazenar apenas um valor de cada vez, as **listas** são estruturas capazes de armazenar **diversos valores de uma vez só**. Esses valores são também chamados de **elementos** da lista. A Figura 1 ilustra uma lista chamada "notas" com 8 elementos. Isso significa que ela está armazenando 8 informações de uma só vez (neste caso, as notas de 8 alunos).

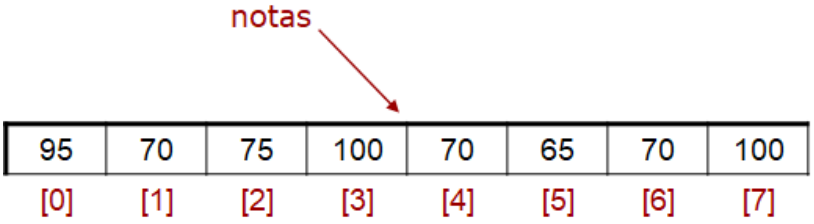

Figura 1. Lista com 8 elementos.

Mais formalmente, uma lista pode ser definida como uma coleção de *n* elementos armazenados em sequência. Cada elemento está associado a uma **posição** ou **índice** da lista, que consiste no número que indica a posição do elemento na sequência. O primeiro índice é *0* e o último *n-1*. Em "notas", os índices são os números 0 a 7, exibidos abaixo dos valores armazenados nesta lista.

As listas podem ser criadas usando uma atribuição, de forma similar ao que fazemos com variáveis. Porém, devemos especificar a sequência de valores entre colchetes, onde os

valores devem estar separados por vírgula. O programa a seguir mostra como criar e imprimir a lista da Figura 1.

```
notas = [95, 70, 75, 100, 70, 65, 70, 100]

print(notas)
```

```
>>>
[95, 70, 75, 100, 70, 65, 70, 100]
```

VII.2 Propriedades Básicas das Listas

Agora que sabemos criar uma lista, vamos conhecer as suas propriedades básicas e aprender como podemos modificar e recuperar informações nela armazenadas. Para isso, considere novamente a lista "notas", criada no programa anterior. Esta lista possui 8 elementos. A Figura 2 ilustra que essa lista (assim como qualquer outra) possui dois componentes:

- **ÍNDICE, SUBSCRITO, CÉLULA ou POSIÇÃO**: identifica uma posição da lista.
  - *** * ATENÇÃO**: o **primeiro** índice da lista é sempre o **índice 0** (e não 1).
  - Já o **último** índice da lista é sempre ***n-1***, onde *n* corresponde ao número de elementos da lista. Se a lista tem 8 elementos, o último estará armazenado no índice 7 (e não 8), como é o caso da lista da Figura 2.

- **VALOR ou ELEMENTO**: corresponde ao conteúdo da lista

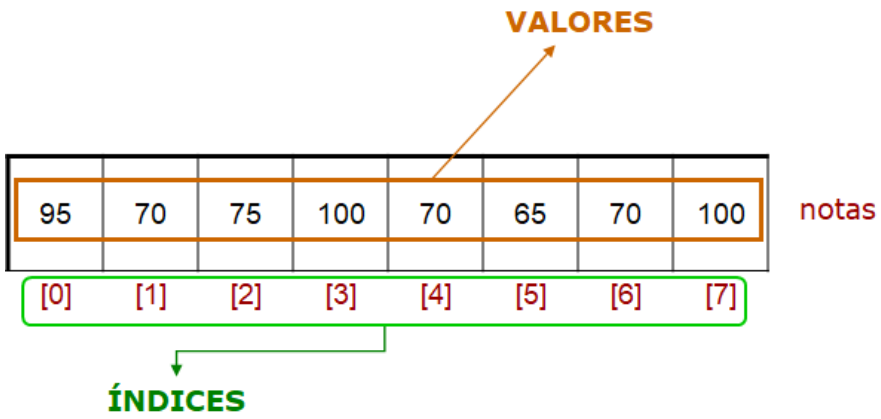

Figura 2. Índices e valores

Na figura acima, tem-se que o índice (ou subscrito ou posição) 0 da lista possui o valor 95 armazenado. O índice 1, por sua vez, possui o valor 70. Já o índice 2 possui o valor 75 e assim por diante. Note ainda que a lista pode ter valores repetidos: o valor 70 aparece nas posições 1, 4 e 6 e o valor 100 nas posições 3 e 7.

Para **armazenar** um valor em uma determinada posição **de uma lista já existente**, basta referenciar a posição em questão entre colchetes [] e utilizar normalmente o operador de atribuição (=). Exemplos:

```
notas[7] = 92    # armazena o valor 92 no índice 7 de "notas"
                 # (o valor anterior será substituído)

notas[1] = notas[5]  # torna o conteúdo do índice 1 da lista
                     # igual ao conteúdo do índice 5 - os dois
                     # índices passam a armazenar o valor 65
```

De maneira análoga, se desejamos ler o valor de uma posição da lista, basta especificá-lo entre colchetes. Veja alguns exemplos:

```
print(notas[2])        # imprime 75 na tela

if notas[5] > 60:      # resulta em True, pois notas[5]
                       # armazena o valor 65

notas[3] <= notas[0]   # retorna False, pois notas[3]
                       # armazena 100 e notas[0] armazena 95
```

Em resumo, uma lista funciona da mesma forma que uma variável. No entanto, a variável é uma "caixinha" ou "gaveta" que só pode armazenar um valor de cada vez. Uma lista pode guardar muitos valores de uma vez porque é um "conjunto de gavetas" (cada uma dessas gavetas pode ser acessada através de seu índice).

A seguir um exemplo básico de programa contendo a criação de uma lista, e depois o acesso e modificação de alguns de seus elementos.

```
#cria a lista "a" com 10 elementos
a = [-45, 6, 0, 72, 1543, -89, 0, 62, 20, -1]
print('lista original =', a)

#impressão dos elementos de índice 2 e 6
print('a[2] = ', a[2])
print('a[6] = ', a[6])

#novas atribuições
a[2] = a[2] + 1;   #agora a[2] vale 1}
a[6] = a[4];       #agora a[6] vale 1543}

#impressão dos elementos de índice 2 e 6
print('** a[2] e a[6] foram modificados:')
print('a[2] = ', a[2])
print('a[6] = ', a[6])

#impressão da lista inteira
```

```
print('lista após modificações =', a)
```

```
>>>
lista original = [-45, 6, 0, 72, 1543, -89, 0, 62, 20, -1]
a[2] =  0
a[6] =  0
** a[2] e a[6] foram modificados:
a[2] =  1
a[6] =  1543
lista após modificações = [-45, 6, 1, 72, 1543, -89, 1543, 62, 20, -1]
```

O programa anterior cria uma lista de 10 elementos chamada "a" (linha 2), imprimindo-a em seguida (linha 3). Depois imprime na tela o conteúdo armazenado nos subscritos 2 e 6 (linhas 6 e 7). A seguir, nas linhas 10 e 11, o conteúdo armazenado nos mesmos subscritos 2 e 6 é modificado. Esses valores modificados são impressos nas linhas 15 e 16. Por fim, a última linha do programa imprime toda a lista de uma vez.

Listas e Vetores

Na linguagem Python a estrutura de dados lista desempenha o **mesmo papel** que é desempenhado pela estrutura conhecida como **vetor** (***array unidimensional***) em outras linguagens de programação.

Assim como toda variável possui um tipo (`int`, `float`, `bool` ou `str`), toda lista do Python é um objeto tipo `list`. Podemos verificar isso utilizando a função `type()`, que é usada no Python para informar o tipo de qualquer objeto. Toda lista possui um **tamanho**, que pode ser obtido com o uso da função **len()**. Esta é uma função do Python que recebe como entrada uma coleção, e como saída retorna o número de elementos dessa coleção.

```
nome = 'Jane Austen'
idade = 41
livros = ['Orgulho e Preconceito', 'Razão e Sensibilidade', 'Emma']

print(type(nome))
print(type(idade))
print(type(livros))

print('núm. de elementos da lista "livros" = ', len(livros))
```

```
>>>
<class 'str'>
<class 'int'>
<class 'list'>
núm. de elementos da lista "livros" =  3
```

VII.3 Tipos de Lista

Na linguagem Python é possível criar 3 tipos de lista:

- **Lista simples**: todos os elementos possuem o mesmo tipo de dado. A lista "notas", apresentada nas Figuras 1 e 2 é um exemplo de lista simples, uma vez que todos os seus elementos são do tipo inteiro. A Figura 3 mostra outro exemplo de lista simples, em que todos os elementos são do tipo string.

Figura 3. Exemplo de lista simples

- **Lista mista**: este tipo de lista possui elementos de diferentes tipos. A Figura 4 apresenta um exemplo de lista mista. Veja que os elementos nas posições 0 e 2 são do tipo string, o elemento da posição 1 é do tipo inteiro e o da posição 3 do tipo float.

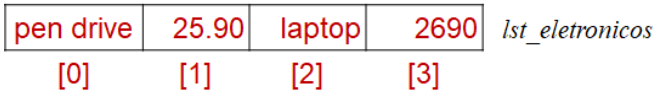

Figura 4. Exemplo de lista mista

- **Lista bidimensional**: trata-se de uma lista de listas, ou seja, uma lista cujos elementos são outras listas. As listas bidimensionais são muito importantes no Python, pois são utilizadas para representar **matrizes**. Elas serão abordadas no capítulo 8 de nossa apostila, onde trataremos de tópicos avançados sobre listas.

A seguir um programa que cria 4 listas. As duas primeiras são listas simples, a terceira uma lista vazia e quarta é uma lista mista.

```
lst_familia = ['John', 'Yoko', 'Julian', 'Sean']

sequencia_fibonacci = [0,1, 1, 2, 3, 5, 8, 13, 21, 34, 55]

lista_vazia = []

lst_eletronicos = ['Pen Drive', 25.90, 'Laptop', 2690]
```

VII.4 Indexação

A operação que consiste em acessar um elemento de uma lista, seja para consulta-lo ou modifica-lo, é conhecida como **indexação**. Como vimos anteriormente, para indexar um dado elemento, devemos especificar o nome da lista e o índice do elemento desejado entre colchetes. Veja alguns exemplos considerando a lista "carros", exibida na Figura 5.

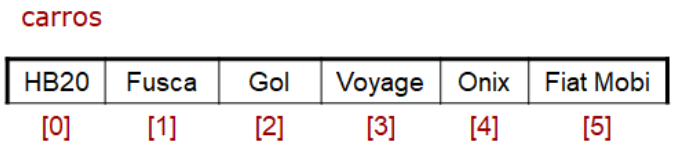

Figura 5. Lista "carros"

Veja alguns exemplos de operação de indexação sobre essa lista:

- `carros[0]`: indexa o primeiro elemento da lista, que, conforme ilustra a figura é "HB20".

- `carros[4]`: indexa o quinto elemento da lista – cujo valor é igual a "Onix".

- `carros[6]`: esta operação de indexação resulta em um **erro**, mais precisamente um erro do tipo **IndexError**. O motivo é simples de entender: a lista tem apenas 6 elementos, portanto os índices da lista estão na faixa entre 0 e 5. Logo, não existe a posição 6, não existe elemento de índice 6 (nem qualquer índice com valor acima de 5).

O Python também permite a indexação utilizando **índices negativos**. Neste caso, -1 serve para indexar o último índice da lista, -2 o penúltimo e assim sucessivamente. Veja:

- `carros[-1]`: indexa o último elemento da lista, cujo valor é "Fiat Mobi". Também é possível indexar esse elemento fazendo `carros[5]`.

- `carros[-2]`: indexa o penúltimo elemento da lista, cujo valor é igual a "Onix". Também é possível indexar esse elemento por `carros[4]`.

- `carros[-7]`: resulta em um **erro**. A lista tem apenas 6 elementos, e o primeiro é indexado por `carros[-6]` ou `carros[0]`.

O operador **in** pode ser utilizado para verificar se um valor pertence à lista (ou seja, se alguma posição da lista armazena o valor). Basta indicar o valor e depois escrever **in** e o nome da lista. É como se estivéssemos perguntando "O valor *x* está nessa lista?". A operação resultará em True se o valor estiver na lista. Caso contrário, resultará em False.

- `'Fusca' in carros` retorna True

- `'BMW' in carros` retorna False

Já **not in** faz o contrário. Neste caso, a pergunta é "o valor *x* não está na lista?"

- `'Fusca' not in carros` retorna False

- `'BMW' not in carros` retorna True

Tipo do Elemento Indexado
Uma operação de indexação retorna um elemento do tipo que estiver armazenado na posição indexada. Neste exemplo, a lista é simples, com todos os elementos sendo nomes de carros. Logo a operação de indexação sobre a lista "carros" sempre retornará um elemento do tipo `str`.

VII.5 Iteração

Iteração é a operação que consiste em **percorrer** todos ou parte dos elementos de uma lista, **um por um**, em sequência. Esta operação é implementada através da construção de um laço com o comando `for` (também é possível usar o `while`, porém é mais simples iterar com o `for` na maioria da situações práticas).

A forma mais simples de iterar por todos os elementos de uma lista é utilizando a estrutura **`for-coleção`**, para realizar a iteração diretamente sobre os elementos. A sintaxe é apresentada na Figura 6.

```
for v in coleção:
    comando1
    comando2
    …
    comandon
```

Figura 6. Modelo de utilização da estrutura *for-in*.

Nesta sintaxe:

- O comando `for` vai percorrer sequencialmente cada elemento da lista que for especificada como "*coleção*"
- A cada iteração, a variável "*v*" receberá como valor um dos elementos da coleção.

Veja o exemplo a seguir:

```
lst = ['John', 'Yoko', 'Julian', 'Sean']

for pessoa in lst:
    print(pessoa)
```

```
>>>
John
Yoko
Julian
Sean
```

Uma segunda maneira de iterar por uma lista e fazer não diretamente sobre os seus elementos, mas sim **com base nos índices** da lista. Isto pode ser implementado como o auxílio das funções `range()` e `len()`. Veja o exemplo a seguir:

```
lst = ['John', 'Yoko', 'Julian', 'Sean']

for k in range(len(lst)):
    print("elemento", k, "=", lst[k])
```

```
>>>
elemento 0 = John
elemento 1 = Yoko
elemento 2 = Julian
elemento 3 = Sean
```

Pode parecer difícil, mas não há nenhum mistério aqui. Veja a explicação:

- Como sabemos, a função `len()` serve para retornar o número de elementos de uma lista. Logo, no exemplo apresentado, **`len`**`(lst)` retorna 4.

- Já a função `range()` gera uma sequência de números. Então, neste código **`range`**`(`**`len`**`(lst))` resulta em **`range`**`(4)`, gerando a sequência {0, 1, 2, 3}

- Então temos um `for` que vai percorrer a sequência {0, 1, 2, 3}. Com isso, podemos na primeira iteração *k* valerá *0* e poderemos acessar o primeiro elemento da lista. Na segunda iteração *k* valerá *1*, o que nos permitirá acessar o segundo elemento. E assim sucessivamente.

VII.6 Métodos de Listas

Quando você cria uma lista em um programa, você "ganha de presente" uma série de **métodos** que podem ser utilizados para executar operações sobre a lista. Na realidade, toda lista é uma coleção que contém dados (valores dos elementos armazenados) e métodos (funções presentes "dentro" da lista e que sempre estão disponíveis para serem utilizadas pelo programador). Assim como ocorre com as funções normais (como `len()`, `type()` etc.), um método pode receber zero ou mais argumentos e retornará um valor. A diferença é que, para chamar um método, devemos adicionar um ponto (".") e o nome do método ao final do nome de uma lista – isto é chamado de "invocar" o método (um nome um pouco feioso, mas não tão terrível quanto "indentação"... na verdade invocar significa simplesmente executar"). A sintaxe para invocar um método é resumida na Figura 7.

```
lista.método(parâmetros)
```

Figura 7. Sintaxe para executar um método

Nesta sintaxe, "lista" corresponde ao nome da lista, "método" ao nome do método que desejamos executar e "parâmetros" são os parâmetros exigidos por tal método. Cada método executa uma ação diferente, portanto exige parâmetros diferentes. Alguns métodos servem para **modificar** uma lista (ex.: inserir ou remover elementos) outros para **explorá-la** de alguma forma (ex.: contar o número de ocorrências de um determinado valor). Neste capítulo, apresentaremos apenas os métodos básicos para alterar listas, deixando os demais para o próximo capítulo. Estes métodos são relacionados a seguir:

- lista.**append(*x*)**: insere elemento *x* no final da lista. Esse é um método muito importante, pois é simples, eficiente e muito utilizado na prática.

- lista.**insert(*i, x*)**: insere elemento *x* na posição *i*.

- lista.**pop*(i)***: remove um elemento no final (caso *i* não seja especificado) ou da posição *i* da lista (caso *i* seja especificado). O método retorna o elemento removido.

- lista.**remove(*x*)**: remove o primeiro elemento que tiver o valor *x*.

- lista.**clear()**: esvazia a lista

- lista.**extend(*lista2*)**: concatena os elementos da *lista2* ao final de *lista*. Também pode-se usar *lista* += *lista2*.

O programa a seguir exemplifica a utilização prática dos métodos acime. Analise-o com calma. Para facilitar a sua compreensão, colocamos comentários ao lado de cada linha que apresentam o conteúdo da lista após a invocação do método.

```
numeros = [5, 10]

numeros.append(20)      # insere 20 no final: [5, 10, 20]
numeros.append(10)      # insere 10 no final: [5, 10, 20, 10]
numeros.insert(2,15)    # insere 15 na posição 2: [5, 10, 15, 20, 10]
numeros.insert(0,10)    # insere 10 na posição 0: [10, 5, 10, 15, 20, 10]

numeros.pop()           # remove o último elemento: [10, 5, 10, 15, 20]
numeros.pop(3)          # remove o quarto elemento: [10, 5, 10, 20]
numeros.remove(10)      # remove o primeiro 10: [5, 10, 20]

numeros.extend([40, 50])  # estende a lista: [5, 10, 20, 40, 50]

numeros.clear()           # esvazia a lista: [ ]
```

- *** * IMPORTANTE**: os métodos modificam a lista sem que precisemos realizar uma atribuição com o operador "="

VII.7 Carregando uma Lista via Teclado (Tamanho Fixo)

Nos exemplos apresentados até agora, nossas listas foram definidas diretamente dentro do código do programa, como no exemplo a seguir:

```
lst = ['John', 'Yoko', 'Julian', 'Sean']
```

Porém, na maioria dos programas os dados de entrada são especificados pelo usuário. Sendo assim, essa seção apresenta a receita básica para que você possa carregar listas a partir da digitação do usuário. Esta receita é resumida na Figura 8.

1. criar uma lista vazia
2. criar um laço para receber cada informação
2.1 dentro do laço, receber cada informação via input()
2.2 dentro do laço, inserir cada informação na lista usando o método append

Figura 8. Receita para carregar uma lista via teclado

O programa a seguir mostra como usar a receita na prática. Ele cria serve para criar uma lista contendo 5 nomes de filmes lidos via teclado. Cada nome lido é inserido no fim da lista com o uso do método `append()`. Digite o programa e teste-o com diferentes entradas.

```
print('Digite os nomes dos 5 filmes que você mais gosta:')

lst_filmes_favoritos = []  #1. Cria a lista vazia

for i in range(5):     #2. Cria um laço para receber as informações
    titulo = input()   #2.1 Recebe uma informação
    lst_filmes_favoritos.append(titulo) #2.2 Insere a informação na lista

print('Os seus 5 filmes favoritos são:')
print(lst_filmes_favoritos)
```

A lista é criada vazia, mas ao final do processamento terá 5 elementos. Veja um exemplo de simulação de execução:

```
>>>
Digite os nomes dos 5 filmes que você mais gosta:
Edukators
Casablanca
O Filho da Noiva
O Discreto Charme da Burguesia
Orgulho e Preconceito

Os seus 5 filmes favoritos são:
['Edukators', 'Casablanca', 'O Filho da Noiva', 'O Discreto Charme da
Burguesia', 'Orgulho e Preconceito']
```

Também seria possível fazer usar o `input()` dentro do `append()`, sem carregar os dados primeiros para a variável "titulo": `lst_filmes_favoritos.append(input())`.

VII.8 Problemas resolvidos

EXEMPLO VII.1 Criar um programa que armazene seis nomes em uma lista. Depois solicite um número entre 0 e 5 e imprima o nome da pessoa que está armazenado no índice cujo valor é igual a esse número. Caso o usuário digite um valor fora da faixa entre 0 e 5, imprima uma mensagem de erro na tela.

```
MAX_NOMES = 6

lst_nomes = []

for x in range(MAX_NOMES):
    print("digite o nome ", x, ": ")
    nome = input()
    lst_nomes.append(nome)


print('Digite um numero entre 0 e ', MAX_NOMES - 1, ': ')
n = int(input())

if (n > 0) and (n < MAX_NOMES):
     print(lst_nomes[n])
else:
     print('Número inválido')
```

Neste exemplo, declaramos a variável MAX_NOMES com o valor 6 (número de elementos que serão inseridos na lista) no início do programa e a utilizamos em diferentes linhas do código. Conforme apresentado no capítulo anterior, esse é um recurso bastante prático que podemos utilizar em programas, para que seja possível alterar apenas uma linha de código para fazer com que, automaticamente, o programa leia um número maior ou menor de dados via teclado.

EXEMPLO VII.2 Faça um programa que armazene 15 matrículas em uma lista "matriculas" e 15 médias finais em uma lista "medias". Em seguida imprima um relatório contendo apenas a matrícula dos alunos com média igual ou superior a 7.

```
#parte 1 - lê os dados
TAM = 15

matriculas = []; medias = []

for i in range(TAM):
    print('Digite a matricula ', i + 1, ': ');
    mat = input()
    matriculas.append(mat)

    print('Digite média final ', i + 1, ': ');
    media = float(input())
    medias.append(media)
```

```
#parte 2 - imprime o relatório
print()
print('RELATORIO DE APROVADOS:')

for i in range(TAM):
    if (medias[i] >= 7.0):
         print(matriculas[i])
```

Veja que na parte 1 do programa carregamos duas diferentes listas via teclado, uma com as matrículas (elementos do tipo string) e outra com as médias finais (elementos do tipo float). Para a interação com o usuário ficar mais bonita, utilizarmos `i + 1` nos `prints` que solicitam a entrada de dados. Com isso, o programa na hora de pedir a 1ª matrícula (e também a 1ª média), o programa exibirá "Digite a matrícula 1" em vez de "Digite a matrícula 0".

Na parte 2 do programa, realizamos uma iteração sobre a lista de médias e para cada posição testamos se um valor acima de 7.0 está armazenado. Se isto ocorrer, vamos até a lista de matrículas e imprimimos a matrícula referente a tal média (que estará armazenada na mesma posição).

Embora simples, o programa representa o esquema típico dos programas que futuramente você criará para realizar a análise de dados: primeiro é feita a entrada dos dados e uma vez que estes dados tenham sido carregados em alguma coleção, realiza-se a análise destes dados.

EXEMPLO VII.3 Faça um programa que leia duas listas de números reais "a" e "b", cada uma com 10 elementos. A partir destas duas listas, gere um terceira chamada "c" da seguinte forma: o primeiro elemento de "c" deve ser obtido a partir a soma do primeiro elemento de "a" com o último (décimo) de "b". O segundo elemento de "c" deve representar a soma do segundo elemento de "a" com o penúltimo de "b", e assim por diante.

```
TAM = 10

#parte 1(a) - lê os dados da lista "a"
a = []
for i in range(TAM):
    print('a[', i, ']: ');
    valor = float(input())
    a.append(valor)

#parte 1(b) - lê os dados da lista "b"
b = []
for i in range(TAM):
    print('b[', i, ']: ');
    valor = float(input())
    b.append(valor)

#parte 3 - gera "c"
c = []
for i in range(TAM):
    c.append(a[i] + b[TAM - (i+1)])

print(c) #parte 4 - imprime "c"
```

EXEMPLO VII.4 Faça um programa que, inicialmente, leia 25 números inteiros e armazene em uma lista "lst". Em seguida, o programa deve determinar e imprimir o maior e o menor dos 25 números e a posição em que cada um deles está armazenado.

```
TAM = 25;

#parte 1 - lê os dados para a lista lst
lst = []
for i in range(TAM):
    print('lst[', i, ']: ', end=" ");
    numero = int(input())
    lst.append(numero)

# parte 2 - determina o maior e o menor
# inicia com a hipótese de que o maior e o menor estão na posição 0
pos_maior = 0
pos_menor = 0
maior = lst[0]
menor = lst[0]

# analisa todas as demais posições (1 em diante), fazendo o seguinte:

# atualiza maior e pos_maior quando acha alguém maior que
# o que foi achado anteriormente

# atualiza menor e pos_menor quando acha alguém menor que
# o que foi achado anteriormente

for i in range(1,TAM):
    if lst[i] > maior:
         maior = lst[i]
         pos_maior = i

    if lst[i] < menor:
         menor = lst[i]
         pos_menor = i

#parte 3 - imprime resultado
print('O maior número é', maior, "na posição", pos_maior)
print('O menor número é , menor, "na posição", pos_menor)
```

EXEMPLO VII.5 Criar um programa que leia uma lista "w" de 10 valores inteiros e construa outra lista "u" da seguinte forma.

Lista w	3	8	4	2	...	16
Lista u	9	4	12	1	...	8

```
#parte 1 - lê os dados da lista w e, ao mesmo tempo, gera u}
N = 10

w = []
u = []

for j in range(N):
    print('A[' ,j, ']: ', end=" ")
    x = int(input())
    w.append(x)

    if j % 2 == 0:
        u.append(w[j] * 3)
    else:
        u.append(w[j] // 2)

#parte 2 - imprime w e u
print(w)
print(u)
```

VII.9 Carregando uma Lista via Teclado (Tamanho Indeterminado)

Imagine que você precisa fazer um programa com as seguintes características: ele deve receber o nome de uma disciplina da ENCE e um conjunto contendo as notas finais de todos os alunos. Ao final da digitação, o programa deverá imprimir um relatório que listará todas as notas e indicará quais delas estão acima da média da turma. O programa deve ser "genérico" o suficiente para atender a qualquer disciplina da universidade, considerando que nenhuma disciplina pode ter mais de 100 alunos matriculados (suponha que esta seja uma regra da instituição).

Neste problema, será necessário usar uma lista para o armazenamento de dados (as notas finais). No entanto, dependendo da disciplina a lista será carregada com uma quantidade diferente de dados (ex.: para a disciplina Introdução à Programação a lista normalmente precisará receber a nota de 60 alunos, enquanto que para a disciplina Bases de Dados esse número poucas vezes passa de 40 alunos). Em resumo, o que ocorre é que **conhecemos o limite máximo** de informações que poderão ser armazenadas na lista, mas esta quantidade de informações muda para cada disciplina e é sempre em um valor menor ou igual a 100.

A resolução de um problema deste tipo pode ser feita de duas maneiras. A primeira delas é permitir com que o usuário utilize um ***flag*** para sinalizar que deseja terminar o processo de digitação (no caso deste problema, basta escolher um valor impossível de nota final para servir como *flag*, como –1 ou –999). A segunda maneira (mais simples) é perguntar a quantidade de informações que serão digitadas, antes de começar a receber os dados. Os exemplos VII.6 e VII.7 ilustram as duas formas de resolver o problema.

PROBLEMA: Construir um programa receba o nome de uma disciplina e a quantidade de notas finais (a quantidade máxima de informações que pode ser digitada é 100).

Em seguida gere um relatório que liste todas as notas e indique as que estão acima da média.

EXEMPLO VII.6 Resolução com o uso de FLAG

```
print('Programa Notas Finais')
print('====================')
print;

# ------------------------------------------------
# PASSO 1: recebe o nome da disciplina
# ------------------------------------------------
disciplina = input('Qual o nome da disciplina? ')

# ------------------------------------------------
# PASSO 2: recebe notas finais até o usuário desejar encerrar
# ------------------------------------------------
soma = 0      # receberá a soma dos valores das notas
x = 0         # receberá cada nota que será inserida na lista
lst_notas = []

while (x != -999):
    print('Digite uma nota ou -999 para encerrar: ')
    x = float(input()) # armazena inicialmente a nota em "x"

    if (x >= 0) and (x <= 10):      # coloca a nota na lista apenas
                                    # se ela tiver valor entre 0 e 10

        lst_notas.append(x) # coloca a nota na lista

        soma += x      # incrementa o somatório dos valores das notas -
                       # necessário para o cálculo da média

# ------------------------------------------------
# PASSO 3: calcula a média e imprime o relatório
# ------------------------------------------------
if (len(lst_notas) > 0):
    media = soma / len(lst_notas)

    print('* * DISCIPLINA: ', disciplina,' - MEDIA FINAL = ',  round(media,2))

    for i in range(len(lst_notas)):
        print('Nota ', i+1, ' = ', lst_notas[i], end=" ")
        if (lst_notas[i] > media):
           print(' ** ACIMA DA MEDIA ', end=" ");

        print()
```

A Figura 9 apresenta uma tela com um exemplo de possível execução para o programa.

```
Programa Notas Finais
=====================
Qual o nome da disciplina? Bases de Dados
Digite uma nota ou -999 para encerrar:
10
Digite uma nota ou -999 para encerrar:
7.3
Digite uma nota ou -999 para encerrar:
7.8
Digite uma nota ou -999 para encerrar:
5.1
Digite uma nota ou -999 para encerrar:
1.5
Digite uma nota ou -999 para encerrar:
8.5
Digite uma nota ou -999 para encerrar:
-999
* * DISCIPLINA:  Bases de Dados  - MEDIA FINAL =  6.7
Nota  1  =  10.0  ** ACIMA DA MEDIA
Nota  2  =  7.3  ** ACIMA DA MEDIA
Nota  3  =  7.8  ** ACIMA DA MEDIA
Nota  4  =  5.1
Nota  5  =  1.5
Nota  6  =  8.5  ** ACIMA DA MEDIA
```

Figura 9– Execução do programa das médias resolvido com *flag*

O principal "segredo" da resolução do programa está no uso de um laço **while** (PASSO 2) para receber os dados via teclado enquanto o usuário não digitar o flag de saída (neste caso, -999).

Uma vez que o usuário tenha digitado todas as notas, elas estarão armazenadas em "lst_notas". Como sabemos, o tamanho de qualquer lista pode ser obtido com o uso da função `len()`. Com isto, basta utilizar essa função para saber quantas notas o usuário digitou. O total de notas digitados é necessário tanto para realizar o cálculo da média, como para imprimir o relatório de aprovados fazendo um `for` com base nos índices, ações realizadas no PASSO 3 do programa.

EXEMPLO VII.7 Resolução com o uso de PERGUNTA AO USUÁRIO

```
print('Programa Notas Finais')
print('=====================')
print;

# ------------------------------------------------
# PASSO 1: recebe o nome da disciplina
#          e o total de notas a serem digitadas
# ------------------------------------------------
disciplina = input('Qual o nome da disciplina? ')

ficar = True
while (ficar): # deixa o usuário preso no laço até digitar valor coerente
    total_notas = int(input('Quantas notas serão digitadas? '))
    if total_notas > 0: ficar = False
```

```
# ------------------------------------------------
# PASSO 2: recebe as notas até atingir o total
#          especificado pelo usuário
# ------------------------------------------------
soma = 0       # receberá a soma dos valores das notas
x = 0          # receberá cada nota que será inserida na lista
digitadas = 0 # contabiliza o total de notas digitadas até o momento

lst_notas = []

while (digitadas < total_notas):
    print('Digite a nota', digitadas + 1, ':')
    x = float(input()) # armazena inicialmente a nota em "x"

    if (x >= 0) and (x <= 10):      # coloca a nota na lista apenas
                                    # se ela tiver valor entre 0 e 10

        lst_notas.append(x) # coloca a nota na lista

        digitadas += 1 # incrementa o total de digitadas

        soma += x      # incrementa o somatório dos valores das notas -
                       # necessário para o cálculo da média

# ------------------------------------------------
# PASSO 3: calcula a média e imprime o relatório
# ------------------------------------------------
if (len(lst_notas) > 0):
    media = soma / len(lst_notas)

    print('* * DISCIPLINA: ', disciplina,' - MEDIA FINAL = ',  round(media,2))

    for i in range(len(lst_notas)):
        print('Nota ', i+1, ' = ', lst_notas[i], end=" ")
        if (lst_notas[i] > media):
           print(' ** ACIMA DA MEDIA ', end=" ");

        print()
```

No caso desse programa, a ideia principal foi a de garantir com que o usuário digitasse um valor correto (acima de 0) para a quantidade de notas no PASSO 1. Esse valor é armazenado na variável "total_notas".
Depois disso o problema pôde ser resolvido normalmente, de uma forma muito similar à utilizada no Exemplo VII.6. A principal diferença é que agora precisamos de uma variável chamada "digitadas" para controlar o laço do **while** do PASSO 2. Esse laço é mantido enquanto o valor de "digitadas" for inferior ao de "total_notas", para garantir a digitação do número de notas desejado pelo usuário. A Figura 10 apresenta uma tela com um exemplo de possível execução para o programa.

```
Qual o nome da disciplina? Estrutura de Dados
Quantas notas serão digitadas? 4
Digite a nota 1 :
10
Digite a nota 2 :
4.7
Digite a nota 3 :
5.5
Digite a nota 4 :
8.1
* * DISCIPLINA:  Estrutura de Dados  - MEDIA FINAL =  7.07
Nota  1  =  10.0  ** ACIMA DA MEDIA
Nota  2  =  4.7
Nota  3  =  5.5
Nota  4  =  8.1  ** ACIMA DA MEDIA
```

Figura 10. Execução do programa das médias resolvido com pergunta ao usuário.

Exercícios propostos

(1) Faça um programa que carregue uma lista com 10 valores inteiros digitados pelo usuário. Depois de carregar a lista, calcule e imprima a quantidade de números negativos e a soma dos números positivos dessa lista.

(2) Faça um programa que leia uma lista com 15 números reais. Em seguida, calcule e mostre a soma dos quadrados dos elementos dessa lista.

(3) Faça um programa que leia 7 letras do teclado, uma de cada vez, e armazene as letras digitadas em uma lista.

Em seguida exiba numa única linha do vídeo a lista na ordem direta (ou seja, exiba primeiro o elemento 0, depois o 1, ..., até o 6) - as letras devem ser exibidas uma do lado da outra. Na linha seguinte, exiba a lista na ordem inversa (ou seja, valores do índice 6 ao 0), também com as letras uma ao lado da outra. Veja a imagem abaixo. O seu programa deverá gerar uma saída **exatamente igual** a apresentada.

```
Digite 7 letras:
N
I
T
E
R
Ó
I
Ordem direta: NITERÓI
Ordem inversa: IÓRETIN
```

(4) Faça um programa que primeiro crie uma lista "a" preenchida automaticamente com os números 1 a 10. Imprima a lista criada. Depois eleve todos os números ao quadrado, alterando o conteúdo da lista e imprimindo-a em seguida. A saída do programa deve ser exibida em uma única linha, exatamente igual a que é apresentada a seguir:

a = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10] → [1, 4, 9, 16, 25, 36, 49, 64, 81, 100]

(5) Altere o programa anterior de modo que a lista gerada tenha 30 números em vez de apenas 10.

(6) Uma escola de informática deseja saber se existem alunos cursando, simultaneamente, as disciplinas Programação Python e Programação Java.

Coloque os números das matrículas dos alunos que cursam Programação Python em uma lista (no máximo 30 alunos, podendo ser menos). Coloque os números das matrículas dos alunos que cursam Programação Java em outra lista (no máximo 30 alunos, podendo ser menos). Gere, em seguida, um relatório que mostre o número das matrículas que aparecem nas duas listas.

(7) Crie um programa que leia uma quantidade indeterminada de números inteiros e armazene-os em uma lista. Em seguida imprima um relatório que apresente o número e o índice (posição na lista) dos números que forem múltiplos de seu índice.

(8) Crie um programa que leia N números reais ($10 \leq N \leq 500$) e armazene em uma lista. Em seguida imprima os números localizados nos 5 primeiros subscritos e os números localizados nos 5 últimos subscritos da lista.

(9) Faça um programa que leia exatamente 3 números reais, os insira em uma lista e depois identifique o maior, o menor e o do meio.

(10) Crie um programa que leia 10 números inteiros e os armazene em uma lista. Em seguida imprima um relatório que exiba apenas os números que são quadrados perfeitos. Um número é quadrado perfeito quando tem um número inteiro como raiz quadrada.

DICA: alguma função do módulo math (Capítulo IV) poderá ser útil nessa resolução.

(11) Faça um Programa que leia duas listas com 10 elementos cada. Gere uma terceira lista de 20 elementos, cujos valores deverão ser compostos pelos elementos intercalados das duas outras listas.

(12) Faça um programa que receba a temperatura média de cada mês do ano e armazene-as em uma lista. Após isto, calcule a média anual das temperaturas e mostre todas as temperaturas acima da média anual, e em que mês elas ocorreram (mostrar o mês por extenso: 1 – Janeiro, 2 – Fevereiro, . . .).

(13) O valor de π pode ser obtido com o uso da série:

$\pi = 4 – (4/3) + (4/5) – (4/7) + (4/9) – (4/11) + ...$

Crie um programa que armazene o valor dos 5000 primeiros termos da série em uma lista. O primeiro índice deve conter o primeiro termo (4), o segundo índice o segundo termo (4/3), e assim por diante.

Em seguida, o programa deve calcular e exibir na tela o valor de π (mostre na tela com 20 casas decimais).

(14) Escreva um programa para inserir um elemento entre cada par de elementos consecutivos de uma lista. **IMPORTANTE**: a lista original deverá ser alterada.

elemento = 'Python'

a = ['R', 'SAS', 'Java'] → ['R', 'Python', 'SAS', 'Python', 'Java']

(15) (Crivo de Eratóstenes) Um **inteiro primo** é qualquer número inteiro que só pode ser dividido exatamente por si mesmo e por 1. O “Crivo de Eratóstenes” é um método diferente, porém muito prático de se encontrar números primos até um certo valor limite. Esse método deverá ser utilizado nesta questão, para que sejam determinados todos os números primos menores do que 1000. Para isto, escreva um programa composto pelos **3 passos** descritos a seguir:

(a) Crie uma lista de inteiros com 1001 posições (de 0 a 1000). Depois atribua o valor 1 para todas as posições desta lista, **exceto** as posições 0 e a posição 1. Ambas deverão receber o valor 0.

0	**0**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**...**	**1**	**1**
0	1	2	3	4	5	6	7	8	9	10	11	...	999	1000

Em seguida, **começando da posição 2 da lista**, sempre que for encontrado um elemento com valor igual a 1, faça um laço pelo restante da lista e defina como zero todos os elementos **cujo subscrito seja múltiplo** do subscrito com valor 1.

Exemplo: para o subscrito 2 da lista, todos os elementos posteriores à posição 2 que estiverem armazenados em subscritos múltiplos de 2 serão definidos como zero (subscritos 4, 6, 8, 10, etc.). Idem para o subscrito 3 da lista (subscritos 6, 9, 12, 15, 18, etc). E assim, para os demais subscritos.

Ao final do processamento, os elementos da lista com subscritos primos, permanecerão 1. Todos os outros elemento estarão com o valor 0, conforme ilustra a figura abaixo.

0	**0**	**1**	**1**	**0**	**1**	**0**	**1**	**0**	**0**	**0**	**1**	**...**	**0**	**0**
0	1	2	3	4	5	6	7	8	9	10	11	...	999	1000

(b) Continue o programa, criando uma rotina que percorra a lista e imprima na tela todos números primos entre 1 e 1000 (basta **imprimir** todos **os subscritos** dos elementos que contêm o valor 1).