Capítulo 12

Listas e similares

Uma lista de compras, uma lista de tarefas a realizar, uma lista de presença. Todos nós estamos habituados a esses objetos. Em Python, uma lista lista é mais ou menos isso: uma sequência de elementos, que podem ser acessados individualmente. E sobre a qual podemos realizar algumas operações como adicionar ou remover elementos, procurar um elemento ou modificar um elemento.

No caso de Python, os elementos são acessados pela posição que ocupam dentro da lista. O esquema é o mesmo que utilizamos para acessar os caracteres de um string. Vejamos alguns exemplos, inicialmente de como podemos criar listas em Python.

```
>>> q1 = []
>>> q2 = list()
>>> q3 = [1,2,3]
>>> q4 = ['Cerveja', 'Carne', 'Carvão']
>>> q5 = list(q4)
```

Os dois primeiros comandos criam listas vazias, ou seja, que não contêm nenhum elemento. O terceiro cria uma lista com três elementos, cada um deles um número inteiro. O quarto comando cria, também, uma lista com três elementos, mas cada um deles é um string. O último comando cria uma outra lista, que é igual à lista armazenada em `q4`, ou seja, com os mesmos strings.

Apesar de termos usado todos os elementos do mesmo tipo, isso não é necessário. Podemos misturar em uma lista valores de tipos diferentes. Por exemplo, a seguir construímos uma lista que tem números inteiros, `float`s e um string. Note, também, que essa lista contém elementos repetidos, ou seja, o número 3,14 aparece duas vezes dentro da lista.

```
>>> q = [2.3, 3, 3.14, 'Carvão', 7, 3.14]
>>> q
[2.3, 3, 3.14, 'Carvão', 7, 3.14]
```

Como dissemos, podemos acessar os elementos da lista usando um índice, que indica a sua posição. Assim como os caracteres de um string, as posições iniciam em zero e são incrementadas, de um em um. Ou seja, o primeiro elemento tem índice zero, o segundo tem índice um e assim por diante.

```
>>> q = [2.3, 3, 3.14, 'Carvão', 7, 3.14]
>>> q[0]
2.3
>>> q[3]
'Carvão'
>>> q[-1]
3.14
```

Assim como nos strings, um índice negativo representa a posição à partir do final da lista. Porém, um índice que não existe como seis ou sete negativo gera um erro, indicado pelo interpretador.

```
>>> q = [2.3, 3, 3.14, 'Carvão', 7, 3.14]
>>> q[6]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
```

Podemos criar novas listas, também, a partir de pedaços de uma lista existente. Usamos a mesma notação dos strings, com o ":". Lembramos que a notação `[a:b]` inclui todos os elementos que estejam entre `a` e `b-1`. Ou seja, o valor na posição `b`, não entra na sublista. É importante notar que essa notação cria uma nova lista com os mesmos elementos da lista original.

```
>>> q = [2.3, 3, 3.14, 'Carvão', 7, 3.14]
>>> q[1:4]
[3, 3.14, 'Carvão']
>>> q[-4:4]
[3.14, 'Carvão']
```

Qualquer um dos valores na notação pode ser omitido. Se o primeiro for omitido, significa "a partir do início da lista". Se o segundo for omitido, significa "até o fim da lista". A seguir, alguns exemplos.

```
>>> q = [2.3, 3, 3.14, 'Carvão', 7, 3.14]
>>> q[:]
[2.3, 3, 3.14, 'Carvão', 7, 3.14]
>>> q[:4]
[2.3, 3, 3.14, 'Carvão']
>>> q[-4:]
[3.14, 'Carvão', 7, 3.14]
```

Ao contrário do que acontece no acesso a elementos individuais, na notação de sublista não ocorre um erro se usarmos um índice que não existe na lista. Nos dois exemplos abaixo, a lista criada é delimitada pelo início e pelo fim da lista original, no caso em que o índice usado está "abaixo do início" ou "acima do final", respectivamente.

```
>>> q = [2.3, 3, 3.14, 'Carvão', 7, 3.14]
>>> q[:17]
[2.3, 3, 3.14, 'Carvão', 7, 3.14]
>>> q[-30:17]
[2.3, 3, 3.14, 'Carvão', 7, 3.14]
```

Podemos, ainda, adicionar um terceiro valor nessa notação de lista. Esse terceiro valor indica de quantos em quantos elementos desejamos pegar da lista. No exemplo a seguir, usamos `[1:5:2]` para indicar que desejamos os elementos que estão entre a posição um até a posição cinco (essa não é incluída), de dois em dois, ou seja, as posições 1 e 3. No segundo exemplo no quadro a seguir criamos uma lista com elementos inteiros de 100 a 119 e em seguida criamos uma nova lista com os elementos entre as posições 3 e 18, de 3 em três.

```
>>> q = [2.3, 3, 3.14, 'Carvão', 7, 3.14]
>>> q1 = q[1:5:2]
>>> q1
[3, 'Carvão']
>>> t = list(range(100,120))
>>> t[3:18:3]
[103, 106, 109, 112, 115]
```

Os componentes da lista podem ser modificados, depois que a lista foi criada.

Por isso em Python diz-se que é uma estrutura mutável. Para trocar o valor armazenado em uma posição da lista basta fazer uma atribuição àquela posição. Mas não podemos atribuir um valor a uma posição que não existe ainda. Em particular, não podemos incluir um novo elemento na lista atribuindo um valor para a próxima posição "livre".

```
>>> q = [2.3, 3, 3.14, 'Carvão', 7, 3.14]
>>> q[2] = 'Carne'
>>> q
[2.3, 3, 'Carne', 'Carvão', 7, 3.14]
>>> q[-1] = 0
>>> q
[2.3, 3, 'Carne', 'Carvão', 7, 0]
>>> q[6] = 'Gelo'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list assignment index out of range
```

12.1 Inclusão e exclusão

Para incluir novos elementos na lista precisamos usar alguma função. A função `append` inclui um elemento no final da lista, que passa ter um elemento a mais. A função `insert` recebe um parâmetro a mais, que indica em que posição o novo elemento deve ser inserido. O elemento que estava naquela posição continua na lista, mas passa a ocupar a posição seguinte, o mesmo acontecendo com os elementos que estavam nas posições sucessivas. Ou seja, os elementos são deslocados, uma posição para frente. Note que a forma correta de chamar essas funções é usando a notação de ponto, que vimos anteriormente. Repare também que usar na função `insert` uma posição que não existe na lista não provoca um erro. O elemento novo é inserido no final ou no início da lista.

```
>>> q = [2.3, 3, 3.14, 'Carvão', 7, 3.14]
>>> q.append('Gelo')
>>> q
[2.3, 3, 3.14, 'Carvão', 7, 3.14, 'Gelo']
>>> q.insert(2,33)
>>> q
[2.3, 3, 33, 3.14, 'Carvão', 7, 3.14, 'Gelo']
>>> q.insert(10,'Linguiça')
>>> q
[2.3, 3, 33, 3.14, 'Carvão', 7, 3.14, 'Gelo', 'Linguiça']
>>> q.insert(-10, 8)
>>> q
[8, 2.3, 3, 33, 3.14, 'Carvão', 7, 3.14, 'Gelo', 'Linguiça']
```

Da mesma forma, para remover elementos da lista usamos funções. A função `pop` não requer nenhum parâmetro e remove o último elemento da lista. Além de remover, a chamada à função também retorna um valor, que é o valor que foi removido.

```
>>> q = [2.3, 3, 3.14, 'Carvão', 7, 3.14]
>>> x = 2 * q.pop()
>>> q
[2.3, 3, 3.14, 'Carvão', 7]
>>> x
6.28
```

Podemos, também, passar um parâmetro para a função `pop` que indica qual é a posição que queremos remover da lista. Note no quadro a seguir que vamos remover a primeira ocorrência de 3.14 da lista, e não a última como fizemos anteriormente. Se a posição indicada for inválida, o interpretador aponta um erro no comando.

```
>>> q = [2.3, 3, 3.14, 'Carvão', 7, 3.14]
>>> x = 2 * q.pop(2)
>>> q
[2.3, 3, 'Carvão', 7, 3.14]
>>> x
6.28
```

Em vez de especificar qual é a posição que queremos eliminar, podemos indicar qual é o valor a ser excluído. Caso o valor apareça mais do que uma vez na lista, apenas a primeira ocorrência será eliminada. E se o valor indicado não

estiver na lista, o interpretador aponta um erro na execução do comando.

```
>>> q = [2.3, 3, 3.14, 'Carvão', 7, 3.14]
>>> q.remove(3.14)
>>> q
[2.3, 3, 'Carvão', 7, 3.14]
>>> q.remove('Gelo')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: list.remove(x): x not in list
```

Ainda para remover elementos da lista podemos utilizar o comando `del`. A vantagem é que podemos usar a mesma notação que usamos para acessar os elementos da lista, ou seja, a lista seguida do índice que desejamos remover.

```
>>> q = [2.3, 3, 3.14, 'Carvão', 7, 3.14]
>>> del q[2]
>>> q
[2.3, 3, 'Carvão', 7, 3.14]
>>> del q[5]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list assignment index out of range
```

Com o comando `del` podemos ainda remover vários elementos usando a notação de sublistas que vimos anteriormente. Por exemplo, no quadro a seguir vamos criar uma lista com os números inteiros de 100 até 119. Depois, vamos remover todos os elementos entre as posições 3 e 18, de 3 em 3. Ou seja, 103, 106, 109, 112 e 115.

```
>>> t = list(range(100, 120))
>>> del t[3:18:3]
>>> t
[100, 101, 102, 104, 105, 107, 108, 110, 111, 113, 114, 116,
117, 118, 119]
```

12.2 Outras operações

Nesta seção apresentamos algumas outras operações que são muito utilizadas com as listas. Ainda não mencionamos, mas podemos obter o número de elementos de uma lista com a função, que já conhecemos, `len`. Para verificar se um valor faz parte de uma lista utilizamos o operador `in`. Vejamos alguns exemplos:

```
>>> q = [2.3, 3, 3.14, 'Carvão', 7, 3.14]
>>> len(q)
6
>>> len(q[1:5:2])
2
>>> 'Carvão' in q
True
>>> 'Gelo' in q
False
```

A aplicação do operador `in` produz um valor booleano. Em um programa, se quisermos inserir um elemento na lista, apenas se ele não estiver presente, ou seja, não queremos elementos repetidos na lista, podemos ter o seguinte trecho de código:

Programa 12.1 Verificando se elemento existe em uma lista, antes de inserir

```
1 if not 'Carvão' in q:
2     q.append('Carvão')
```

Podemos comparar listas utilizando os operadores relacionais que vimos anteriormente. Duas listas são iguais quando elas são do mesmo tamanho e todos os seus elementos são iguais. Para comparar se uma lista é maior ou menor que outra, o interpretador compara o primeiro elemento de cada uma delas e obtém o resultado dessa comparação. Se os primeiros elementos das duas listas forem iguais, são comparados os segundos de cada uma delas, e assim por diante, até que se encontre uma diferença ou que uma das listas termine.

```
>>> q = [2.3, 3, 3.14, 'Carvão', 7, 3.14]
>>> q == [2.3, 3, 3.14, 'Carvão', 7, 3.14]
True
>>> q < [3, 3.14, 'Carvão', 7, 3.14]
True
>>> q > [2, 3, 3.14, 'Carvão', 7, 3.14]
True
>>> q < [2.3, 3, 3.14, 'Carvão', 7, 3.14, 'Gelo']
True
```

Duas lista podem ser somadas. Essa operação resulta em uma nova lista que é a concatenação das duas listas originais.

```
>>> q = [2.3, 3, 3.14, 'Carvão', 7, 3.14]
>>> q + q
[2.3, 3, 3.14, 'Carvão', 7, 3.14, 2.3, 3, 3.14, 'Carvão', 7,
3.14]
>>> q + ['Gelo']
[2.3, 3, 3.14, 'Carvão', 7, 3.14, 'Gelo']
>>> q[1:5:2] + q
[3, 'Carvão', 2.3, 3, 3.14, 'Carvão', 7, 3.14]
>>> [] + q + []
[2.3, 3, 3.14, 'Carvão', 7, 3.14]
```

E podemos até multiplicar uma lista por um número. O número indica quantas vezes a lista original deve ser repetida, para formar uma nova lista.

```
>>> q = [2.3, 3, 3.14, 'Carvão', 7, 3.14]
>>> 2 * q
[2.3, 3, 3.14, 'Carvão', 7, 3.14, 2.3, 3, 3.14, 'Carvão', 7,
3.14]
>>> ['Gelo'] * 3
['Gelo', 'Gelo', 'Gelo']
>>> q[1:5:2] * 4
[3, 'Carvão', 3, 'Carvão', 3, 'Carvão', 3, 'Carvão']
>>> 1024 * []
[]
```

12.2.1 Exercícios

Como a manipulação de listas e de strings é similar em vários aspectos, aproveitamos essa seção para praticar o uso desses dois tipos de dados.

1. Escreva um programa que coloque em uma lista os números inteiros de 1 a 100, em ordem crescente e depois mostre essa lista.

2. Escreva um programa que coloque em uma lista os números inteiros de 1 a 100, em ordem decrescente e depois mostre essa lista.

3. Faça um programa que leia dois strings. Após isso, o programa deve concatenar as informações lidas e mostrar o resultado para o usuário. Exemplo: Se o primeiro string digitado for "Bom dia, " e o segundo "moçada !", então o resultado deverá ficar: "Bom dia, moçada !".

4. Faça um programa que leia N elementos inteiros e coloque numa lista e depois leia um valor de código. Se o código for 1, o programa deve mostrar a lista na ordem direta, se o código for 2, deve mostrar a lista na ordem inversa.

5. Fazer um programa para ler um string e um caractere qualquer. Após isso, deve calcular o número de ocorrências desse caractere no string. Exemplo: Seja a string "USP - São Carlos" e o caractere "s", então o número de ocorrências é 3.

6. Escreva um programa que leia N elementos inteiros e coloque em uma lista. Após isso, seu programa deve percorrer a lista e mostrar somente os números pares.

7. Escreva um programa que leia N elementos inteiros e coloque em uma lista. Após isso, seu programa deve verificar se a sequência na lista representa uma sequência de Fibonacci. Seu programa pode considerar que qualquer sequência em que os elementos são a soma dos dois anteriores é uma sequência de Fibonacci. Por exemplo: 7 8 15 23 38.

8. Fazer um programa para ler um string e dois caracteres. Após isso, deve trocar todas as ocorrências do primeiro caractere pelo segundo. Mostre o string final na saída. Exemplo: Seja a entrada "ambiental" e os caracteres "a" e "b", então o string ficará "bmbientbl".

9. Escreva um programa que computa o valor de um polinômio em um determinado ponto. Essa programa deve receber como entradas:

   - um número inteiro que indica o grau do polinômio;
   - um lista de `float` com os coeficientes do polinômio. A posição k da lista corresponde ao coeficiente de x^k;
   - um `float` que indica o ponto no qual o polinômio deve ser calculado.

   Por exemplo, o polinômio $3x^5 - 12x^3 + 1.08x^2 - 3.9x + 8$ é representado pela lista: [8.0, -3.9, 1.08, -12.0, 0.0, 3.0].

10. Escreva um programa capaz de ler uma frase e contar o número de palavras dessa frase. Considere que as palavras estão separadas por espaços ou vírgulas.

11. Fazer um programa para ler uma frase e verificar se ela é palíndroma, isto é se ela é igual lida da esquerda para a direita e vice-versa. Exemplos: "Ana" é palíndroma, "arara" é palíndroma, "USP" não é palíndroma, "Anotaram a data da maratona" é palíndroma. (obs: os espaços em branco na frase lida devem ser desconsiderados em seu algoritmo).

12. Elabore um programa que receba uma linha de texto e conte as vogais apresentando o respectivo histograma na seguinte forma:

    Exemplo:
    Linha de texto passada: "A próxima quarta-feira é feriado."

    ```
    a : ****** (6)
    e : *** (3)
    i : *** (3)
    o : ** (2)
    u : * (1)
    ```

13. Escreva um programa que leia um número inteiro n e depois exiba as n primeiras linhas do triângulo de Pascal. Veja um exemplo abaixo. Não é difícil descobrir como essa matriz é formada. Sugestão: armazene sempre a linha corrente em uma lista.

    ```
    1
    1  1
    1  2  1
    1  3  3  1
    1  4  6  4  1
    1  5  10 10 5  1
    1  6  15 20 15 6  1
    ```

14. Elabore um programa que receba uma linha de texto e conte quantas vezes cada letra do alfabeto aparece nesse texto. Utilize uma lista para armazenar essa informação e a função `ord` para saber em que posição da lista está o contador de cada letra. Para saber se um caractere é ou não um aletra, você pode usar a função `isalpha`.

15. Implemente um programa de "criptografia" (codificação de dados visando a privacidade de acesso as informações). Dado um string seu programa deve codificar o string por meio de um processo de substituição de caracteres. Cada caractere do string tem uma representação numérica à qual você pode soma um determinado valor, alterando, assim, o conteúdo da mensagem. Para recuperar o string original, devemos tomar a mensagem criptografada e subtrair o mesmo valor. Implemente, também, o programa que recupera a mesnsagem original. Como dica, olhe as funções `ord` e `chr` da linguagem Python.

16. Escreva um programa que leia um inteiro n e depois leia duas listas, cada uma com n elementos. Em seguida, seu programa deve gerar uma terceira lista com os elementos das listas originais, intercalados.

12.3 Aplicação: estatística descritiva

Vamos usar, nesta seção, algumas operações com listas para implementar algumas das técnicas de estatística descritiva vistas na primeira parte deste livro. Inicialmente, precisamos coletar os dados que vamos usar. Para isso, por enquanto, temos que solicitar ao usuário que digite os dados, que vamos armazenar em uma lista.

Iniciamos perguntando ao usuário do nosso programa quantos são os dados que pretende fornecer. Depois, precisamos solicitar que ele digite aquela quantidade de valores e cada valor digitado adicionamos em uma lista.

Programa 12.2 Lendo os dados a serem tratados

```
1  n = int(input('Quantos valores serão digitados? '))
2  x = []
3  i = 1
4  while i <= n:
5      msg = 'Digite o valor ({}/{}): '.format(i,n)
6      r = float(input(msg))
7      x.append(r)
8      i += 1
9
10 print(x)
```

A variável `i` é utilizada para contar quantos números foram lidos. Ela é incrementada até atingir o valor de `n`. A cada iteração do comando `while`, um novo valor é lido e adicionado à lista com a função `append`. No final, mostramos a lista que foi lida, para que o usuário confira os valores. Executando esse programa com 3 valores, temos o seguinte resultado:

```
Quantos valores serão digitados? 3
Digite o valor (1/3): 5.3
Digite o valor (2/3): 3
Digite o valor (3/3): 8.1
[5.3, 3.0, 8.1]
```

Com o que aprendemos até agora, podemos melhorar um pouquinho o nosso programa. Podemos deixá-lo um pouco mais "amigável", ou seja, mais fácil para que vai usá-lo. Primeiro, em vez de perguntar quantos valores serão digitados, nosso programa deve ler e armazenar na lista todos os valores digitados, até que um valor negativo seja digitado. O segundo aperfeiçoamento é que supomos que os dados a serem digitados estão, por exemplo, no intervalo entre 0 e 100. Nesse caso, se o usuário digitar algum valor maior do que 100, vamos avisá-lo dessa restrição e o valor não é armazenado.

No programa que segue, a variável `r` é que controla a execução do comando `while`. Ele termina sua execução se o valor digitado e atribuído a `r` for negativo.

Ainda assim, mantivemos a variável `i` para que o usuário saiba quantos valores já foram digitados. Além disso, se o valor digitado for maior do que 100, uma aviso é dado ao usuário e o valor é ignorado. Se o valor for válido, ou seja, entre 0 e 100, ele é colocado na lista.

Programa 12.3 Lendo os dados e fazendo sua consistência

```
1  x = []
2  r = 0
3  i = 1
4  while r >= 0:
5      msg = 'Digite o valor ({}): '.format(i)
6      r = float(input(msg))
7      if r < 0:
8          print('Entrada de dados terminou')
9      elif r > 100:
10         print('Valor deve estar entre 0 e 100')
11     else:
12         x.append(r)
13         i += 1
14
15 print(x)
```

Executando esse programa, com os mesmos dados anteriores, temos a saída a seguir. Note que ao digitar o valor 3, o usuário enganou-se e digitou 300. Nosso programa alertou sobre o erro e ignorou o dado digitado errado.

```
Digite o valor (1): 5.3
Digite o valor (2): 300
Valor deve estar entre 0 e 100
Digite o valor (2): 3
Digite o valor (3): 8.1
Digite o valor (4): -1
Entrada de dados terminou
[5.3, 3.0, 8.1]
```

Uma vez armazenados os dados na nossa lista, podemos calcular as nossas estatísticas. Vamos iniciar com a média. Para isso precisamos somar todos os valores que estão na lista. Isso pode ser feito facilmente, já que uma lista é um tipo de objeto que podemos percorrer usando o comando `for`. No trecho de programa abaixo, a cada iteração do `for`, um valor que está na lista é atribuído à variável `r`. Então, na primeira execução, o valor que está em `x[0]` é colocado em `r`, na segunda o valor de `x[1]` e assim por diante, até o final da lista. A cada iteração o valor é somado ao valor da variável `soma`. E no final, a média é calculada e atribuída à variável `media`.

Programa 12.4 Calculando a média dos dados

```
1 soma = 0
2 for r in x:
3     soma += r
4
5 media = soma / len(x)
6 print('Valor da soma: {:.4f}'.format(soma))
7 print('Valor da média: {:.4f}'.format(media))
```

Mais uma vez, para computar a variância e o desvio padrão precisamos percorrer a lista e, para cada elemento, computar o quadrado da diferença em relação à média. Mais uma vez usamos o comando `for` para isso.

Programa 12.5 Calculando a variância e o desvio padrão

```
1 variancia = 0.0
2 for r in x:
3     variancia += (r - media) ** 2
4
5 variancia /= len(x) -1
6 dp = math.sqrt(variancia)
7 print('Valor da variância: {:.4f}'.format(variancia))
8 print('Valor do desvio padrão: {:.4f}'.format(dp))
```

Para o cálculo da mediana, precisamos saber qual é o elemento central da lista. Para isso, fica bem mais fácil se tivermos a lista ordenada de forma crescente. Ou seja, rearranjamos os valores, colocando os menores no início e os maiores no final da lista. Existem vários algoritmos conhecidos para fazer isso e poderíamos implementar qualquer um deles. Mas, a biblioteca do Python já fornece funções para isso.

```
>>>p = [5,-3,-1,8,13]
>>>sorted(p)
[-3, -1, 5, 8, 13]
>>>p
[5,-3,-1,8,13]
>>>p.sort()
>>>p
[-3, -1, 5, 8, 13]
>>>
```

Nesse exemplo, vemos duas formas de ordenar a lista. A primeira, usa a função `sorted` que cria uma cópia da lista passada como parâmetro, e não

modifica a ordenação dessa lista. Podemos, por exemplo, atribuir esse valor a uma segunda variável, fazendo `q = sorted(p)`. A segunda forma, usa a função `sort` que não retorna nenhum valor. Ela modifica a ordenação da lista original, no caso, armazenada em `p`. Então, se quisermos uma nova lista ordenada, preservando a lista original, usamos `sorted`. Se quisermos modificar a lista original, usamos `sort`.

Essas funções podem ser usadas em listas de qualquer tipo, incluindo listas com tipos diferentes de dados, desde que os elementos possam todos ser comparados entre si. Podemos, então, ordenar uma lista que só contem strings ou uma que contenha valores `int` e `float` mas não podemos comparar, por exemplo, uma que tenha valores `int` e strings.

```
>>>p = [5,-3,-1.8,8,13]
>>>sorted(p)
[-3, -1.8, 5, 8, 13]
>>>q = ['a', 'h', 'c']
>>>sorted(q)
['a', 'c', 'h']
>>>sorted(p+q)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unorderable types: int() < str()
>>>
```

Voltando à estatística, uma vez que tenhamos a lista de valores ordenada, basta usar o seu elemento central como mediana. Se o número de elementos for par, usamos a média dos dois elementos centrais. Só precismos tomar cuidado de usar a divisão inteira, ao calcular o índice do elemento central.

Programa 12.6 Calculando a mediana

```
1 x_ordenado = sorted(x)
2 if len(x_ordenado) % 2 == 1:
3     mediana = x_ordenado[len(x_ordenado)//2]
4 else:
5     mediana = (x_ordenado[len(x_ordenado)//2] +
6                x_ordenado[len(x_ordenado)//2-1] ) / 2
7
8 print('Valor da mediana: {:.4f}'.format(mediana))
```

Para computar a moda, precisamos saber quantas vezes cada um dos elementos do conjunto de dados apareceu. Para isso, criamos o que chamamos de "vetor de frequências". Usando uma lista, vamos usar a variável `x_ordenado` para fazer essa contagem. A ideia é a seguinte: vamos criar uma lista `freq` que armazena na posição i, o número de vezes que aparece o elemento de `x_ordenado[i]`. Isso pode ser feito da seguinte maneira:

Programa 12.7 Calculando o vetor de frequências

```
1 freq = []
2 freq.append(1)
3 for i in range(1,len(x_ordenado)):
4     if x_ordenado[i] == x_ordenado[i-1]:
5         freq.append(freq[i-1]+1)
6     else:
7         freq.append(1)
8
9 print('O vetor de freqências: {}'.format(freq))
```

Como a lista `x_ordenado` está ordenada, todos os valores repetidos estão adjacentes. Então, na linha 2 do programa acima, sabemos que o primeiro valor de `x_ordenado` aparece uma única vez. No laço do comando `for`, visitamos a próxima posição dessa lista e se ela for igual ao valor anterior, inserimos em `freq` o valor anterior dessa lista incrementado de uma unidade. Caso contrário, ou seja, apareceu um novo valor na lista `x_ordenado`, então o valor inserido em `freq` é um. Na Figura 12.1, vemos um exemplo de como fica o vetor de frequência para um conjunto de dados.

x_ordenado:	0	2	3.3	3.3	6	7.5	8	8	8	8	11	13.5
freq:	1	1	1	2	1	1	1	2	3	4	1	1

Figura 12.1: Exemplo de um vetor de frequências

Depois disso, vamos achar na lista `freq` qual é o maior valor e a posição correspondente em `x_ordenado` é o valor da moda. No exemplo da Figura 12.1, o maior valor aparece na posição 9 do vetor de frequência, que corresponde ao valor 8, que esta em `x_ordenado[9]`. Só precisamos ter cuidado pois podemos ter mais do que um valor para a moda, ou seja, duas posições diferentes na lista `freq` que têm o valor máximo.

Programa 12.8 Calculando a moda

```
1 maximo = max(freq)
2 moda = []
3 for i in range(len(freq)):
4     if freq[i] == maximo:
5         moda.append(x_ordenado[i])
6
7 print('Valor da moda: {}'.format(moda))
```

No programa acima, a variável `maximo` recebe o valor do maior elemento da lista `freq`, dado pela função `max`, da biblioteca padrão Python. Em seguida,

percorremos cada elemento da lista `freq` e verificamos se ela contém esse valor máximo. Se contém, adicionamos o valor correspondente de `x_ordenado` em um anova lista `moda`. Ao final do laço, os elementos correspondentes à moda, estão armazenados nessa lista.

12.4 Tuplas

Tuplas são estruturas muito semelhantes a listas. A diferença é que, a tupla é "imutável", quer dizer que, uma vez criada, ela não pode ser modificada. Para criar uma tupla, pode-se usar os mesmos recursos que usamos para criar uma lista. Com algumas diferenças na forma de escrever os comandos.

```
>>>t = ('Marcio', 55, 1.73)
>>>r = ()
>>>q = tuple()
>>>p = tuple([1,2,3,4])
>>>
```

No primeiro comando, criamos uma tupla com um string, um número inteiro e um número de ponto flutuante. No segundo e no terceiro, mostramos como criar uma tupla vazia. No quarto comando, criamos uma tupla a partir de uma lista com quatro números inteiros.

Na verdade, podemos representar uma tupla apenas como uma sequência de valores separados por vírgula, ou seja, sem usar os parênteses. A seguir criamos a mesma tupla do exemplo anterior e uma outra com os números inteiros de 1 a 4.

```
>>>t = 'Marcio', 55, 1.73
>>>r = 1,2,3
>>>
```

O acesso aos elementos da tupla é igual ao dos elementos de uma lista. Além disso, podemos utilizar as mesmas funções que aplicamos às listas, desde que a tupla não seja modificada. Vejamos alguns exemplos.