HUAWEI

Fundunesp

Centro Paula Souza

SÃO PAULO
GOVERNO DO ESTADO
SÃO PAULO SÃO TODOS

Este material foi produzido como parte do projeto de colaboração entre

a empresa Huawei Brasil e o Centro Estadual de Educação Tecnológica Paula Souza,

representado pela Fatec de Sorocaba e Fatec de Jundiaí

- 2024 -

Sumário

Capítulo 8
COMANDOS CONDICIONAIS - APROFUNDAMENTO

No capítulo 4 do módulo básico apresentamos o comando condicional `if-elif-else`. Agora que você já avançou no curso e está mais familiarizado com o condicional é hora de saber que em Python existem outras duas formas de comando condicional. São elas:

- O comando match-case: adequado a situações em que múltiplos testes precisem ser realizados;
- O comando `if` de única linha (*inline if*): adequado a situações em que desejamos usar uma sintaxe concisa e legível.

8.1 COMANDO MATCH-CASE

O comando `match-case` foi introduzido na linguagem Python a partir da versão 3.10. Por isso é importante que se você for testá-lo tenha a certeza de estar usando essa versão ou uma superior. Esse comando se assemelha a um comando switch em C ou Java.

Este comando toma uma expressão e compara seu conteúdo com valores sucessivos dispostos em cláusulas `case`, buscando verificar se há correspondência, sendo que tais cláusulas são mutuamente exclusivas. Desse modo, ele permite implementar algoritmos onde múltiplos testes precisam ser realizados. A estrutura do comando é esta:

```
match <expressão>:
    case <valor1>:
        <bloco de comandos 1>
    case <valor2>:
        <bloco de comandos 2>
    ...
    case _:
        <bloco de comandos final>
```

A `expressão` pode ser um objeto, uma expressão aritmética ou o retorno de uma função. Qualquer dessas opções conterá um valor que será comparado com `valor1`, se houver correspondência o `bloco de comandos 1` será executado e todos os demais blocos serão ignorados; caso não haja correspondência com o `valor1`, então será testada a correspondência com o `valor2` executando o `bloco de comandos 2` e assim sucessivamente. Ao final, não havendo correspondência com nenhum dos valores listados, e havendo a cláusula `case _` (case underline) então o bloco subordinado a ela será executado. A ordem das cláusulas

`case` não importam, não sendo necessário que estejam ordenadas, contanto que a última delas seja a que contém o underline: `case _`.

O exemplo a seguir ilustra o uso do comando `match-case`.

Exemplo 8.1

```
n = -1
while n != 0:
    n = int(input('Digite um inteiro: '))
    match n:
        case 1:
            print('  você digitou um')
        case 2:
            print('  você digitou dois')
        case 3:
            print('  você digitou três')
        case _:
            print('  você digitou outra coisa')
print('Fim do Programa')
```

```
Digite um inteiro: 1
  você digitou um
Digite um inteiro: 2
  você digitou dois
Digite um inteiro: 3
  você digitou três
Digite um inteiro: 4
  você outra coisa
Digite um inteiro: 0
  você digitou outra coisa
Fim do Programa
```

Neste exemplo o programa fica em laço enquanto o objeto inteiro `n` for diferente de 0. Dentro do laço, cada valor lido em `n` será processado pelo comando `match` realizando as verificações de correspondências com cada uma das cláusulas `case` e processando o código adequado.

Este comando é bastante flexível com relação as verificações de correspondência que podem ser feitas. No exemplo anterior usamos apenas cláusulas case apenas com valores numéricos inteiros. O exemplo a seguir ilustra outras possibilidades.

Exemplo 8.2 — Este exemplo é ilustrativo e poderá ser implementado adiante

```
entrada = ObtemEntrada()
match entrada:
    case 1:
        print('  a entrada é 1')
    case 'texto':
        print('  a entrada é um texto')
    case [a, b]:
        print('  a entrada é uma lista com dois elementos')
    case _:
        print('  a entrada é uma outra opção')
```

```
(este exemplo apenas ilustra o comando match-case.
 neste momento não pode ser executado pois ainda não temos os elementos
 necessários para implementar a função ObtemEntrada().
 Mais adiante neste texto vamos aprender a criar funções como essa)
```

Vamos aplicar este comando em alguns exercícios.

Exercício Resolvido 8.1 Teste este código no PyCharm

Enunciado: *Escreva um programa para uma fábrica de calçados que leia o código LL de um calçado, que é um número inteiro com 2 dígitos. Exiba na tela a linha do calçado, conforme a tabela a seguir. Se o número fornecido não estiver na tabela, deve-se exibir a mensagem "Código inválido".*

LL	Linha de calçados	LL	Linha de calçados
16	Bebê	49	Masculino esportivo
23	Infantil feminino	52	Feminino formal salto baixo
25	Infantil masculino	53	Feminino formal salto alto
29	Infantil esportivo	55	Feminino casual salto baixo
42	Masculino formal	56	Feminino casual salto alto
43	Masculino casual	59	Feminino esportivo

```
LL = int(input('Digite a linha de calçados: '))
match LL:
    case 16:
        print('Bebê')
    case 23:
        print('Infantil feminino')
    case 25:
        print('Infantil masculino')
    case 29:
        print('Infantil esportivo')
    case 42:
        print('Masculino formal')
    case 43:
        print('Masculino casual')
    case 49:
        print('Masculino esportivo')
    case 52:
        print('Feminino formal salto baixo')
    case 53:
        print('Feminino formal salto alto')
    case 55:
        print('Feminino casual salto baixo')
    case 56:
        print('Feminino casual salto alto')
    case 59:
        print('Feminino esportivo')
    case _:
        print('Código fornecido é inválido')
(crie e execute este código para verificar seu funcionamento)
```

Vale mencionar que não há qualquer impedimento sobre usar o `match-case` em combinação com os outros comandos existentes em Python, como: condicional `if-elif-else`, laço de repetição `while` e iterador `for`.

Benefícios ao utilizar `match-case`

Este comando foi recebido na comunidade Python com muito entusiasmo e oferece diversos benefícios em termos de sintaxe e performance, incluindo:

- **Estrutura concisa**: permite transformar complexas estruturas de ifs aninhados em uma forma concisa e legível.
- **Legibilidade**: com sua sintaxe clara, torna o código legível e autoexplicativo.
- **Segurança**: possibilita que todos os padrões possíveis sejam tratados nas cláusulas `case`.

Um cuidado deve ser tomado quanto ao seu uso. Ao distribuir um script que use esse comando, poderá haver erro de execução do programa caso o usuário final esteja usando uma versão anterior à 3.10. Desse modo, seja bem claro na documentação do seu software informando ao usuário qual a versão mínima de Python que ele deve possuir para ser capaz de executá-lo.

Por fim, à medida que o tempo passe essa questão será menos significativa pois a tendência é a de que os usuários de Python façam atualizações de suas versões.

8.2 COMANDO `if` DE ÚNICA LINHA

8.2.1 CONCEITO

Este tipo de comando existe em outras linguagens de programação como C e Java. Também é conhecido pelo termo "operador ternário". Seu objetivo é permitir uma escrita compacta a comandos condicionais do tipo `if-else` (no caso, sem a presença do `elif`) cujo código subordinado seja constituído por apenas uma instrução.

Imagine que você precise escrever o seguinte condicional.

```
if X >= Y:
    print(X)
else:
    print(Y)
```

Trata-se de um condicional muito simples e, no entanto, ele ocupa quatro linhas no seu código. Embora isso não represente um problema e o código seja legível e compreensível a forma alternativa usando o `if` de única linha é considerado por muitos programadores mais vantajosa. Sua forma é:

```
print(X) if X >= Y else print(Y)
```

Esta linha faz exatamente a mesma tarefa realizada nas quatro linhas anteriores. Veja o exemplo:

Exemplo 8.3

```
>>> X = 10
>>> Y = 9
>>> print(f'X é maior = {X}') if X >= Y else print(f'Y é maior = {Y}')
    X é maior = 10

>>> # o inline-if acima é o mesmo que isto:
>>> if X >= Y:
...     print(f'X é maior = {X}')
... else:
...     print(f'Y é maior = {Y}')
    X é maior = 10

>>> # alterando o valor de Y obtemos a execução do else
>>> Y = 11
>>> print(f'X é maior = {X}') if X >= Y else print(f'Y é maior = {Y}')
    Y é maior = 11
```

(exemplo interativo feito com IDE Idle)

Perceba pelo exemplo que as duas formas de comando condicional – clássica e única linha – funcionam exatamente da mesma maneira. Desse modo, nos seus programas você pode adotar a maneira que considerar mais conveniente. Em termos de desempenho, ou seja, velocidade de execução do código, as duas formas são equivalentes e isso não será critério para escolha entre uma ou outra.

Mas lembre-se que o `if` de única linha suporta apenas um comando nos blocos de instruções subordinadas. E se o seu código contém muitas linhas nesses blocos você terá que usar a forma clássica.

Formalizando, o `if` de única linha tem a seguinte construção:

```
<comando para verdadeiro> if <condição> else <comando para falso>
```

Ele deve ser entendido assim: se a `condição` for verdadeira execute o `comando para verdadeiro`, senão execute o `comando para falso`.

8.2.2 OBRIGATORIEDADE DO `else`

Essa forma deve sempre ser usada completa, ou seja, a parte do `else` é obrigatória não podendo ser omitida. No Idle tente executar os comandos do exemplo abaixo. Você verá que o erro ocorre.

Mas há situações nas quais precisamos implantar um código que não tenha o `else`. Imagine uma situação em que tenhamos uma lista de códigos de produto e desejamos filtrar os códigos dentro de uma certa faixa de valores, por exemplo entre 120 e 200. Códigos fora dessa faixa não nos interessa. Para implementar algo assim poderíamos escrever o código abaixo no qual o `else` é desnecessário.

```
laço while ou for ...
    if codigo >= 120 and codigo <= 200:
        Lista.append(codigo)
```

Ainda assim é possível fazê-lo com um `if` de única linha, conforme mostrado no exemplo 8.4.

Exemplo 8.4

```
Codigos = [103, 117, 121, 135, 161, 189, 200, 204, 216]
Lista = []
print('Alternativa com if clássico')
for codigo in Codigos:
    if codigo >= 120 and codigo <= 200:
        Lista.append(codigo)
print(f'  códigos filtrados -> {Lista}')

print('\nAlternativa com if de única linha')
Lista = []
for codigo in Codigos:
    Lista.append(codigo) if codigo >= 120 and codigo <= 200 else 0
print(f'  códigos filtrados -> {Lista}')
```

```
Alternativa com if clássico
  códigos filtrados -> [121, 135, 161, 189, 200]

Alternativa com if de única linha
  códigos filtrados -> [121, 135, 161, 189, 200]
```

Perceba que bastou colocar `else 0` no `if` de única linha e as duas alternativas produziram o mesmo resultado. Com isso você já sabe como usar essa forma de comando condicional mesmo que não necessite da parte `else`.

No comando condicional de única linha
o else é obrigatório
Se não estiver presente você terá uma mensagem de erro de sintaxe

8.2.3 RETORNO DE VALOR

Adicionalmente precisamos acrescentar que o `if` de única linha tem uma característica que é de muito interesse para a escrita de código compacto e legível: essa forma de `if` produz um retorno de valor, ou seja, você pode escrever uma linha de código com a seguinte forma:

```
<objeto> = <comando para verdadeiro> if <condição> else <comando para falso>
```

Ao usar este recurso o objeto será carregado com o retorno deste `if`, seja ele qual for em função da condição ser verdadeira ou falsa.

Exemplo 8.5

```
X = int(input('Digite um inteiro: '))
paridade = 'par' if X % 2 == 0 else 'ímpar'
print(f'O valor {X} é {paridade}')
```

```
# primeira execução
Digite um inteiro: 10
O valor 10 é par

# segunda execução
Digite um inteiro: 11
O valor 11 é ímpar
```

O exemplo 8.5 lê um número inteiro e mostra na tela se ele é par ou ímpar. Para determinar a paridade foi feito um `if` de única linha que retorna o string 'par' caso X seja par; caso contrário ele retorna o string 'ímpar'. Depois de avaliar a condição o resultado é carregado no objeto paridade, que em seguida é usado no print para exibição na tela.

Compare essa solução com o exercício resolvido 4.1 do capítulo 4 do módulo básico e perceba que esta solução ficou mais compacta e é bastante legível.

8.2.4 CONCLUSÃO SOBRE COMANDOS CONDICIONAIS EM PYTHON

Em síntese, agora você já sabe que na linguagem Python estão disponíveis três alternativas de comando condicional.

- Condicional clássico no formato `if-elif-else`;
- Condicional múltiplo por correspondência de padrões no formato `match-case`;
- Condicional de única linha;

Você tem esses elementos da linguagem à sua disposição para escrever seus programas e pode escolher aquele que melhor se adeque a cada situação que ocorra em seus programas, por isso convém conhecê-los e saber aplicá-los às variadas situações que surgem no desenvolvimento de software.

Capítulo 9
CONJUNTOS

9.1 A CLASSE SET

Conjuntos em Python são implementados através da classe `set`, que permite a criação de coleções de objetos, com as seguintes características:

Elementos não repetidos

Todos os elementos dentro de um conjunto é único, ou seja, repetição de objetos com mesmo conteúdo não são aceitos.

Elementos não ordenados

A ordem dos elementos não tem relevância e não é possível ordená-los. Essa ordem é definida pelo interpretador Python e não seguirá qualquer critério que nós, programadores, possamos pretender (veja o Exemplo 9.1).

O conjunto é mutável

Os conjuntos são mutáveis, ou seja, podem receber novos elementos e podem sofrer exclusão de elementos.

Elementos são imutáveis

Todos os elementos de um conjunto são imutáveis e, uma vez que sejam incluídos, não podem ter seu valor alterado. Por consequência disto, elementos mutáveis como listas, dicionários e outros conjuntos não podem ser membros.

9.1.1 CRIAÇÃO DE UM CONJUNTO

Os conjuntos em Python podem ser criados com o uso do comando de atribuição de duas formas como mostrados nas linhas a seguir:

```
c1 = {16, 8, 29}
c2 = set()
```

Para criar um conjunto não vazio pode-se simplesmente usar um par de chaves `{}` contendo os elementos separados por vírgulas. Porém, para criar um conjunto vazio deve-se usar o construtor `set()`.

No exemplo 9.1 demonstramos a criação de conjuntos, no qual se vê que é uma tarefa simples.

Exemplo 9.1

```
# Criação de um conjunto não-vazio usando o par de chaves {}
>>> c1 = {16, 8, 29}
>>> type(c1)           # verificação do que foi criado
    <class 'set'>      # ok, é conjunto
>>> len(c1)
    3
>>> print(c1)
    {8, 16, 29}

# Criação de um conjunto vazio usando o construtor da classe set()
>>> c2 = set()
>>> type(c2)           # verificação do que foi criado
    <class 'set'>      # ok, é conjunto

# Tentativa de criação de um conjunto vazio usando o par de chaves {}
>>> c3 = {}
>>> type(c3)
    <class 'dict'>     # o uso das chaves dessa forma cria um dicionário

# Criação de um conjunto não-vazio usando o construtor da classe set()
>>> c4 = set((23, 9, 41))
>>> type(c4)           # verificação do que foi criado
    <class 'set'>      # ok, é conjunto
```

(exemplo interativo feito com IDE Idle)

Nesse exemplo, no caso do objeto `c3`, constatamos que ao tentar usar o par de chaves na tentativa de criar um conjunto vazio, acabamos criando um dicionário.

Atenção - Lembre-se sempre ao usar o par de chaves
`c = {}`
Essa atribuição não criará um conjunto vazio, mas sim um dicionário vazio
Para criar conjuntos vazio use sempre o construtor `set()`.

No caso do conjunto `c4` vemos que se pode criar um conjunto não vazio usando o construtor da classe. Neste caso o construtor deve receber como parâmetro iterável, podendo ser string, tupla, lista, conjunto ou dicionário. O exemplo 9.2 ilustra esses vários casos.

Exemplo 9.2

```
# Criação do conjunto a partir de string
>>> texto = 'texto qualquer'
>>> c = set(texto)
>>> print(c)
    {'o', 'x', 'a', 'q', 'e', 't', ' ', 'l', 'u', 'r'}
# observe que não há caracteres repetidos no conjunto

# Criação do conjunto a partir de tupla
>>> tupla = (14, 26, 33, 45, 50)
>>> c = set(tupla)
>>> print(c)
    {33, 45, 14, 50, 26}

# Criação do conjunto a partir de lista
>>> lista = (26, 15, 49, 65, 49)
>>> c = set(lista)
>>> print(c)
    {65, 49, 26, 15}
```

```
# Criação do conjunto a partir de outro conjunto
>>> c2 = set(c)
>>> print(c)
    {65, 49, 26, 15}

# Criação do conjunto a partir de um dicionário
>>> d = {'1235': 49, '1476': 32, '1329': 36}
>>> c = set(d)
>>> print(c)
    {'1329', '1476', '1235'}
```

(exemplo interativo feito com IDE Idle)

Neste exemplo 9.2 usamos diversas classes iteráveis diferentes para fornecer os dados para a criação de conjuntos. O último exemplo usa um dicionário, que ainda não foi visto. Mas não se preocupe com isso agora, pois os dicionários serão vistos no próximo capítulo e lá vamos nos aprofundar e apresentar outras opções de criação de conjuntos.

9.1.2 CONCEITOS: HASH E HASHABLE

Em língua portuguesa não existe uma palavra adequada para traduzir o conceito expresso pela palavra inglesa *hash*. Este é um termo que tem uso frequente no campo da tecnologia da informação, de modo que a hash tornou-se um neologismo, ou seja, o termo em inglês está incorporado ao português.

O hash é um número inteiro calculado a partir do conteúdo de um objeto e existirá em memória durante todo o tempo de existência do objeto. Como esse inteiro é calculado a partir do conteúdo do objeto, dois objetos da mesma classe e com mesmo conteúdo terão o mesmo hash. E este é o principal propósito do hash: ele permite comparações rápidas entre objetos Python.

Nem todas as classes de objetos Python podem ter o hash calculado. Apenas objetos de classes imutáveis tem um hash. A documentação do Python indica que um objeto é hashable se tiver um valor de hash que não muda durante sua vida útil. Se necessário é possível conhecer o número hash dos objetos Python usando a função `hash()`, como mostrado no exemplo a seguir.

Exemplo 9.3

```
>>> s1 = 'texto'
>>> hash(s1)
    -4138086375092316395
>>> s2 = 'qualquer coisa'
>>> hash(s2)
    -1669233493565127761
    valor = 13.75
>>> hash(valor)
    1729382256910270477
>>> x = 26
>>> hash(a)
    26
>>> y = 26.0
>>> hash(y)
    26
>>> t = (15, 23, 7)
>>> hash(t)
    -5803187897390630569
>>> l = [15, 23, 7]
>>> hash(l)
    Traceback (most recent call last):
      File "<pyshell#17>", line 1, in <module>
```

```
        hash(l)
    TypeError: unhashable type: 'list'
```

(exemplo interativo feito com IDE Idle)

O exemplo mostra diversas situações. Observe atentamente cada uma. No caso de números inteiros seu hash é o próprio número. Números reais com a parte decimal zerada são truncados e seu hash será a parte inteira do número, implicando que valores numéricos como `26` e `26.0` tenham o mesmo hash.

Tuplas são imutáveis e possuem o hash. Já as listas são mutáveis e não possuem um número hash, como pode ser visto pelo erro apresentado no exemplo.

9.1.3 Relação entre hash e conjuntos

Existe uma relação entre o número hash de um objeto e os conjuntos. Em Python, apenas objetos de classes hashable podem pertencer a conjuntos. Além disso, a regra dos elementos não repetidos em conjuntos é imposta através desse número hash. Por consequência, valores numéricos inteiros e reais como mencionado acima (o caso 26 e 26.0) não podem coexistir em um conjunto. Isso é mostrado neste exemplo:

Exemplo 9.4

```
>>> c = {26, 26.0}
>>> print(c)
    {26}
```

(exemplo interativo feito com IDE Idle)

9.2 Métodos da classe `set`

A classe `set` tem um conjunto de métodos que nos permitem executar operações com seus elementos. O quadro 9.1 mostra todos eles.

Quadro 9.1 – Métodos da classe `set`

Método	Descrição
`.add(hashable object)`	Acrescenta um objeto ao conjunto. Se o objeto não é hashable ocorre um TypeError
`.clear()`	Remove todos os elementos do conjunto.
`.copy()`	Retorna uma cópia do conjunto.
`.difference(conjuntos)`	Retorna um novo conjunto contendo a diferença de dois ou mais conjuntos passados como parâmetro.
`.difference_update(conjuntos)`	Atualiza o conjunto, removendo de seus elementos os que estejam nos conjuntos passados como parâmetro.
`.discard(object)`	Se o parâmetro estiver presente no conjunto, então o remove, caso não esteja não faz nada.
`.intersection(objects)`	Retorna um novo conjunto contendo a interseção de dois ou mais conjuntos.
`.intersection_update(objects)`	Atualiza o conjunto com a interseção entre seus elementos e os conjuntos passados como parâmetro.
`.isdisjoint(object)`	Retorna True se os dois conjuntos têm interseção vazia. False caso contrário.
`.issubset(object)`	Retorna True se o conjunto é está contido no conjunto passado como parâmetro.
`.issuperset(object)`	Retorna True se o conjunto contém todos os elementos no conjunto passado como parâmetro.
`.pop()`	Remove e retorna um elemento arbitrário do conjunto. Se o conjunto estiver vazio, gera uma exceção KeyError.

`.remove(object)`	Remove do conjunto um elemento que seja seu membro. Caso contrário, gera uma exceção KeyError.
`.symmetric_difference(object)`	Retorna um novo conjunto contendo a diferença simétrica de dois conjuntos.
`.symmetric_difference_update(object)`	Atualiza o conjunto com a diferença simétrica de seus elementos com o conjunto passado como parâmetro
`.union(objects)`	Retorna um novo conjunto com a união o conjunto com o s parâmetros passados.
`.update(objects)`	Atualiza o conjunto com a união de seus elementos com os elementos dos conjuntos passados como parâmetro.

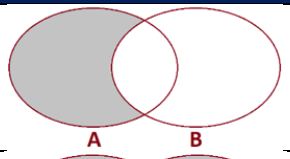
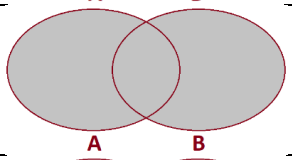
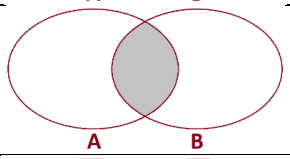
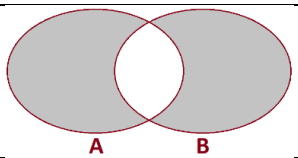
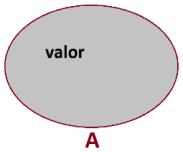
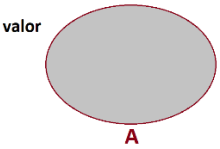
Exemplo 9.5

```
Este exemplo é muito longo, pois mostra os métodos da classe set.
  Então seu código não foi colocado aqui. O vídeo está disponível.

(exemplo interativo feito com IDE Idle)
```

9.3 Operações com conjuntos

Além desses métodos, também estão definidos operadores que podem ser usados com conjuntos. Quatro desses operadores são equivalentes a quatro métodos descritos no quadro 9.1. Os operadores são:

Quadro 9.2 – Operações com conjuntos

Operação	Operador	Descrição	Ilustração do retorno
Diferença	`A - B`	Retorna todos os valores de A que não estão em B Equivale ao método `.difference()`	
União	`A | B`	Retorna a união de A com B Equivale ao método `.union()`	
Interseção	`A & B`	Retorna a interseção de A com B Equivale ao método `.intersection()`	
Diferença Simétrica	`A ^ B`	Retorna os elementos da união de A com B e que não pertencem à sua interseção Equivale ao método `.symmetric_difference()`	
Pertence	`valor in A`	O valor está presente no conjunto A	
Não Pertence	`valor not in A`	O valor não está presente no conjunto A	

Neste quadro destacamos os métodos que equivalem a cada um dos operadores. A diferença entre eles é que os operadores se aplicam apenas a dois conjuntos por vez e os métodos podem ser usados com mais conjuntos em uma única operação.

Observe o exemplo 9.6 a seguir, que ilustra os resultados obtidos com essas operações.

Exemplo 9.6 Teste este código no Idle

```
>>> A = set('ANTONIO CARLOS')
>>> print(A)
    {'O', 'I', 'T', 'R', 'C', 'A', 'N', ' ', 'S', 'L'}
>>> B = set('JOSÉ CARLOS')
>>> print(B)
    {'O', 'J', 'É', 'C', 'A', 'R', ' ', 'S', 'L'}
>>> A - B            # Diferença - elementos em A que não estão em B
    {'N', 'T', 'I'}
>>> B - A            # Diferença - elementos em B que não estão em A
    {'J', 'É'}
>>> A | B            # União
    {'J', 'É', 'N', ' ', 'O', 'I', 'T', 'C', 'A', 'R', 'S', 'L'}
>>> A & B            # Interseção
    {'O', 'C', 'A', 'R', ' ', 'S', 'L'}
>>> A ^ B            # Diferença Simétrica
    {'I', 'J', 'T', 'É', 'N'}
>>> 'X' in A         # Pertencimento
    False
>>> 'T' in A
    True
>>> 'X' not in A     # Não Pertencimento
    True
>>> 'T' not in A
    False
```

(exemplo interativo feito com IDE Idle)

Neste exemplo dois strings (dois nomes próprios – "ANTONIO CARLOS" E "JOSÉ CARLOS") são usados para gerar dois conjuntos de caracteres e, em seguida, as operações com conjuntos são exemplificadas. Agora faça seus próprios testes. Utilize o Idle e refaça esse exemplo para constatar o funcionamento dessas operações.

9.4 CONJUNTOS USADOS COMO ITERADORES

Os conjuntos podem ser usados como iteradores em um comando `for`, do mesmo modo que as classes de sequências. Observe bem esse exemplo. Note que o comando `for` foi usado duas vezes com um conjunto com os memos dados. A diferença entre eles é a ordem com que os dados foram posicionados no momento da criação do conjunto. Ao rodar o programa, a ordem com que os dados são exibidos na tela é a mesma para os dois casos. Assim este exemplo ilustra dois aspectos: primeiro mostra como que usar um conjunto em um laço iterador; segundo mostra que não importa qual ordem seja usada na criação do conjunto, uma vez inseridos no conjunto o interpretador Python decidirá em que posição cada um estará.

Exemplo 9.7 

```
print("primeira execução")
conjunto = {3.3, 18.6, 34.0, 43.1, 58.7}
for c in conjunto:
    print(c)

print("\nsegunda execução")
conjunto = {18.6, 3.3, 58.7, 34.0, 43.1}
for c in conjunto:
    print(c)
print('Fim do Programa')
```

```
primeira execução
```

```
34.0
3.3
18.6
58.7
43.1

segunda execução
34.0
3.3
18.6
58.7
43.1
Fim do Programa
```

Os conjuntos em Python são uma escolha importante quando for o caso de manter uma coleção de elementos não repetidos e com métodos que permitam à manutenção e operações com tais elementos. No entanto, lembre-se de que os conjuntos não suportam indexação, ordenação ou fatiamento. E caso essas características sejam necessárias você deverá usar uma lista.

Exercício Resolvido 9.1

Enunciado: *Escreva um programa que leia um inteiro Qtde e crie um conjunto com elementos numéricos inteiros aleatórios dentro do intervalo fechado [1, 50]. Mostre o conjunto gerado na tela.*

Lembre-se que os conjuntos não podem ter elementos repetidos, então a geração de números aleatórios pode representar um problema. Como resolver isso?

Cuidado: *Este programa tem potencial para entrar em laço infinito caso o valor fornecido para Qtde seja maior que 50. Faça o teste e depois responda a pergunta: por que isso ocorre?*

```
from random import randint
Qtde = int(input('Digite a quantidade: '))
conjunto = set()
while len(conjunto) < Qtde:
    conjunto.add(randint(1, 50))
print(conjunto)
print('Fim do Programa')
```

```
# primeira execução
Digite a quantidade: 12
{1, 34, 4, 40, 8, 11, 46, 17, 19, 22, 24, 31}
Fim do Programa

# segunda execução
Digite a quantidade: 60  # qualquer qtde > 50 faz o programa ficar em laço infinito
|
```

A segunda execução do exercício resolvido 9.1 foi inserida para mostrar como esse algoritmo entrará em laço infinito quando a quantidade fornecida for maior que 50. Assim respondendo à pergunta proposta no enunciado: isso ocorre porque a geração de números aleatórios está limitada a 50 valores (de 1 a 50). Como o conjunto não pode conter valores repetidos, então não há possibilidade matemática desse conjunto ser criado.

O próximo passo é elaborar alguma forma de controle para este programa, para evitar que a situação de laço infinito ocorra. Uma possibilidade é validar o valor fornecido para Qtde e não aceita-lo caso o valor digitado seja maior que 50.

Exercício Resolvido 9.2 Teste este código no PyCharm

Enunciado: *Escreva um programa que leia do teclado dois conjuntos de números inteiros digitados pelo usuário. Exiba na tela a união e a interseção desses conjuntos.*

```
Msg = 'Digite Valor: '
print('Dados do primeiro conjunto')
C1 = set()          # cria o primeiro conjunto vazio
x = int(input(Msg))
while x != 0:
    C1.add(x)       # acrescenta o novo elemento, se ainda não estiver no conjunto
    x = int(input(Msg))
print('Dados do segundo conjunto')
C2 = set()          # cria o segundo conjunto vazio
x = int(input(Msg))
while x != 0:
    C2.add(x)       # acrescenta o novo elemento, se ainda não estiver no conjunto
    x = int(input(Msg))
print(f'\nConjunto 1: {C1}')
print(f'Conjunto 2: {C2}')
print('\nUnião de C1 e C2')
print(C1 | C2)
print('\nInterseção de C1 e C2')
print(C1 & C2)
print('\nFim do Programa')
```

```
Dados do primeiro conjunto
Digite Valor: 16
Digite Valor: 8
Digite Valor: 44
Digite Valor: 0
Dados do segundo conjunto
Digite Valor: 12
Digite Valor: 14
Digite Valor: 16
Digite Valor: 10
Digite Valor: 8
Digite Valor: 0

Conjunto 1: {16, 8, 44}
Conjunto 2: {8, 10, 12, 14, 16}

União de C1 e C2
{8, 10, 44, 12, 14, 16}

Interseção de C1 e C2
{16, 8}

Fim do Programa
```

Exercício Resolvido 9.3 Teste este código no PyCharm

Enunciado: *Escreva um programa que leia um número inteiro Qtde e carregue uma lista com essa quantidade de inteiros aleatórios. Exiba a lista na tela. Em seguida elimine valores que estiverem repetidos, deixando apenas uma ocorrência de cada. Exiba a nova lista sem repetidos e o seu tamanho.*

```
from random import randint
Qtde = int(input('Digite a quantidade: '))
Lista = []
for i in range(Qtde):
    Lista.append(randint(1, 50))
print('\nLista gerada:')
print(Lista)
conjunto = set(Lista)   # converte a lista em conjunto
Lista = set(conjunto)   # retorna o conjunto para lista
```

```
print('\nLista com valores não repetidos:')
print(Lista)
print(f'A nova lista tem {len(conjunto)} elementos')
print('Fim do Programa')
```

```
Digite a quantidade: 20

Lista gerada:
[13, 17, 13, 11, 48, 14, 34, 16, 15, 11, 22, 24, 21, 26, 10, 38, 35, 45, 34, 42]

Lista com valores não repetidos:
{10, 11, 13, 14, 15, 16, 17, 21, 22, 24, 26, 34, 35, 38, 42, 45, 48}
A nova lista tem 17 elementos
Fim do Programa
```

9.5 A CLASSE `frozenset`

A classe `frozenset` é uma versão imutável da classe `set`. Em outras palavras `frozenset` pode ser usada para criação de conjuntos imutáveis, que uma vez criados não podem ter elementos adicionados ou removidos, mas que ainda assim suportam as operações de diferença, união, interseção e diferença simétrica. Para criar um `frozenset` é obrigatório usar o construtor da classe, o qual recebe como parâmetro um iterável, de modo análogo ao construtor da classe `set`. Isso é mostrado no exemplo a seguir:

Exemplo 9.8 — Teste este código no Idle

```
>>> c1 = frozenset((16, 8, 29))
>>> print(c1)
    frozenset({16, 8, 29})
>>> type(c1)
    <class 'frozenset'>
```

```
(exemplo interativo feito com IDE Idle)
```

O exemplo 9.9 a seguir reproduz o exemplo 9.6, porém com o uso de frozenset.

Exemplo 9.9 — Teste este código no Idle

```
>>> A = frozenset('ANTONIO CARLOS')
>>> print(A)
    frozenset({'O', 'I', 'T', 'R', 'C', 'A', 'N', ' ', 'S', 'L'})
>>> B = frozenset('JOSÉ CARLOS')
>>> print(B)
    frozenset({'O', 'J', 'É', 'C', 'A', 'R', ' ', 'S', 'L'})
>>> A - B
    frozenset({'N', 'T', 'I'})
>>> B - A
    frozenset({'J', 'É'})
>>> A | B
    frozenset({'J', 'É', 'N', ' ', 'O', 'I', 'T', 'C', 'A', 'R', 'S', 'L'})
>>> A & B
    frozenset({'O', 'C', 'A', 'R', ' ', 'S', 'L'})
>>> A ^ B
    frozenset({'I', 'J', 'T', 'É', 'N'})
>>> 'X' in A          # Pertencimento
    False
>>> 'T' in A
    True
>>> 'X' not in A      # Não Pertencimento
    True
>>> 'T' not in A
    False
```

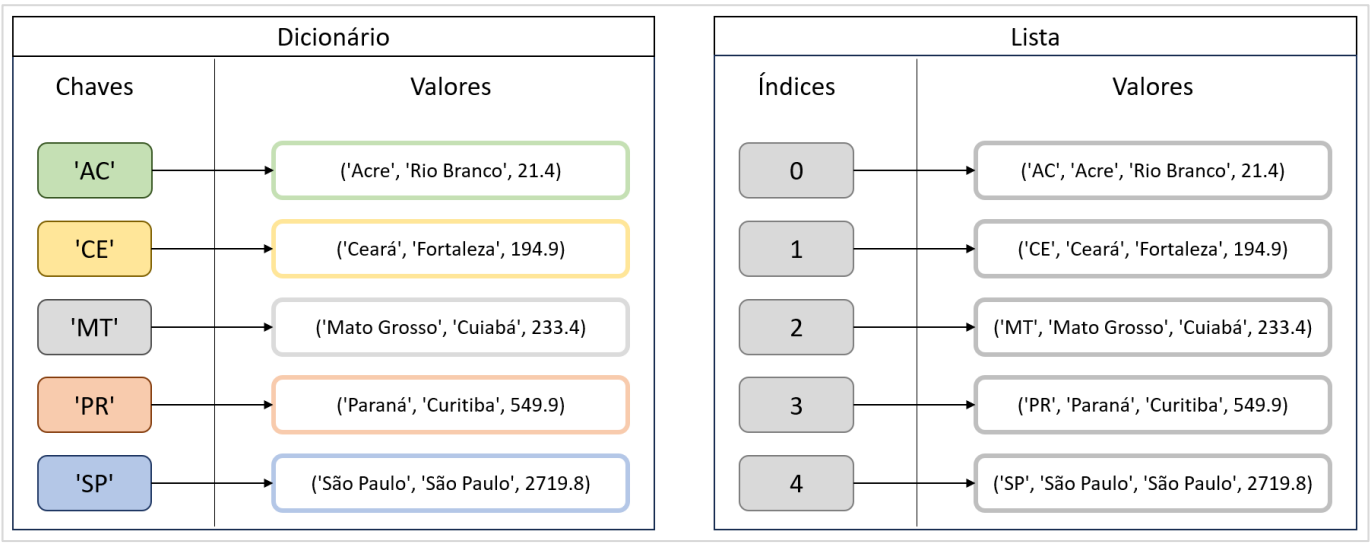
```
(exemplo interativo feito com IDE Idle)
```

Capítulo 10
DICIONÁRIOS

10.1 ESTRUTURA DO DICIONÁRIO

Dicionários em Python são coleções de pares chave-valor. E guardam certa semelhança com as listas. Para entender essa afirmação observe a figura 10.1.

Figura 10.1 – Estrutura de um Dicionário



fonte imagem: o Autor – fonte PIB estadual: IBGE 2021 (em R$ 1bilhão)

Imagine que você precisa elaborar uma aplicação que contenha dados de estados do Brasil: (sigla, nome, capital, PIB estadual em 2021). Você pode criar uma lista onde cada elemento seja uma tupla contendo os dados necessários. Essa lista teria uma aparência como mostrada no lado direito da imagem. Por ser um objeto sequencial, a lista exige o uso do índice numérico – 0, 1, 2, 3, etc – para acesso individual aos dados dos Estados.

Neste caso, o dicionário permite fazer algo mais bem elaborado para essa aplicação. O lado esquerdo da imagem mostra que ao adotar um dicionário podemos usar a sigla do Estado como **chave de acesso** e os dados associados são **o valor**.

Daí decorre a ideia da associação **chave-valor** mencionada.

A classe `dict` de Python é o elemento da linguagem que permite esse tipo de construção.

Apresentado dessa forma pode parecer que os dicionários são melhores que as listas. Não é verdade. Na prática, ambos tem muita importância na linguagem Python. Há situações mais adequadas para listas e outras mais adequadas para dicionários e um fato é certo: os dicionários e as listas estão entre as classes de objetos mais flexíveis e poderosos de Python.

Dicionários e listas se assemelham algumas características:

- são mutáveis;
- são dinâmicos, ou seja, podem crescer e diminuir conforme necessário;
- podem ser aninhados, ou seja, uma lista pode conter outra lista; um dicionário pode conter outro dicionário; um dicionário pode conter uma lista e vice-versa.

Dicionários e listas diferem na forma como os elementos são acessados:

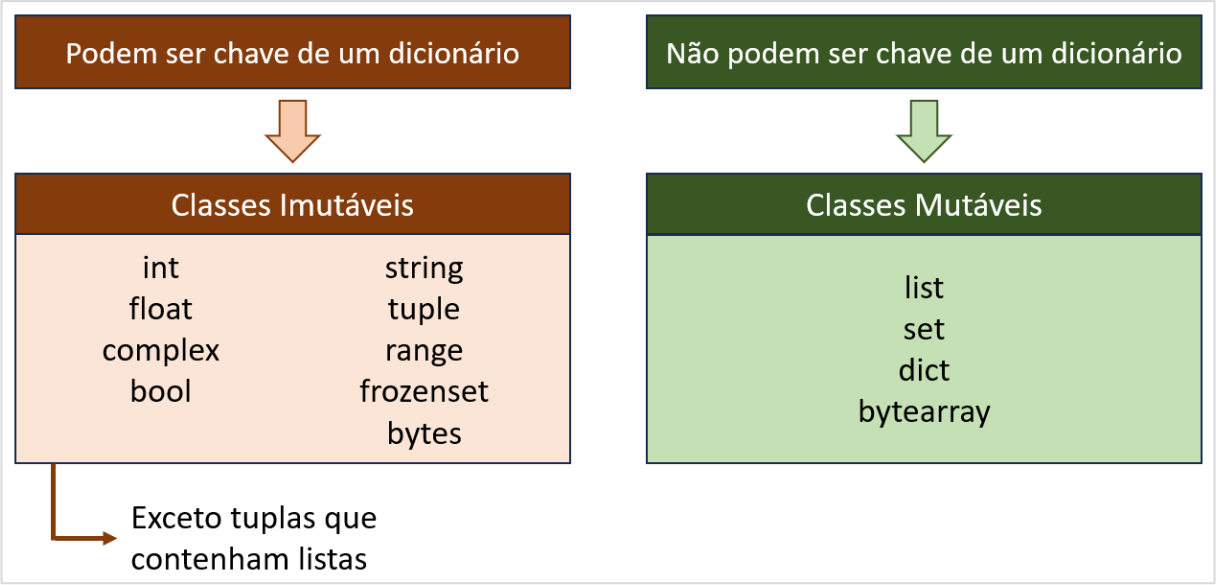
- Os elementos da lista são acessados pela sua posição na lista, via indexação.
- Os elementos do dicionário são acessados por meio de chaves.

Chave dos elementos de um dicionário

Quais classes de objetos podem ser usadas como chave de um dicionário?

Para simplificar, a resposta está na figura 10.2.

Figura 10.2 – Classes que podem ser chave de um dicionário



fonte: o Autor

Quaisquer objetos de classes imutáveis podem ser usados como chave de um elemento de um dicionário, exceto uma tupla que contenha uma ou mais listas.

Chaves de dicionário devem ser objetos *hashable*.
Pesquise sobre isso.
https://docs.python.org/pt-br/3/glossary.html#term-hashable

Valor dos elementos de um dicionário

O valor de um elemento de dicionário pode ser **qualquer** objeto.

10.2 CRIAÇÃO DE UM DICIONÁRIO

A sintaxe para uso dos dicionários se assemelha muito à sintaxe usada para as listas, com a diferença de que no lugar do índice utiliza-se a chave. Veja o exemplo 10.1 onde criamos um dicionário com três elementos, usando como chave um string e como valor outro string.

Exemplo 10.1

```
>>> UF = {'SP': 'São Paulo', 'RJ': 'Rio de Janeiro', 'MG': 'Minas Gerais'}
>>> type(UF)
    <class 'dict'>
>>> print(UF)
    {'SP': 'São Paulo', 'RJ': 'Rio de Janeiro', 'MG': 'Minas Gerais'}
(exemplo interativo feito com IDE Idle)
```

A sintaxe para criar um dicionário pré-carregado com dados envolve a especificação do elemento através do par chave-valor separados pelo caractere ':' (dois pontos). E os vários elementos devem estar separados por vírgula.

```
dicionario = {chave1: valor1, chave2: valor2, ...}
```

Alternativamente, pode-se criar o dicionário vazio e acrescentar os elementos como mostrado no exemplo 10.2

Exemplo 10.2

```
>>> UF = {}                         # cria o dicionário vazio
>>> type(UF)
    <class 'dict'>
>>> UF['SP'] = 'São Paulo'          # inclui primeiro elemento
>>> UF['RJ'] = 'Rio de Janeiro'     # inclui segundo elemento
>>> UF['MG'] = 'Minas Gerais'       # ...e assim por diante
>>> print(UF)
    {'SP': 'São Paulo', 'RJ': 'Rio de Janeiro', 'MG': 'Minas Gerais'}
(exemplo interativo feito com IDE Idle)
```

Ao usar essa construção abaixo podem acontecer duas coisas:

```
dicionario[chave] = valor
```

a) se a chave não existe no dicionário, então o elemento é criado;

b) se a chave já existe no dicionário, então o elemento é alterado;

Isso é demonstrado no exemplo 10.3

Exemplo 10.3

```
>>> D = {}           # cria o dicionário vazio
>>> D['a'] = 120     # insere novo elemento com chave 'a'
>>> print(D)
    {'a': 120}
>>> D['a'] = 250     # altera o elemento com chave 'a'
>>> print(D)
    {'a': 250}
(exemplo interativo feito com IDE Idle)
```

10.3 MÉTODOS DA CLASSE `dict`

A classe `dict` tem um conjunto de métodos que os programadores devem conhecer. Eles são apresentados no quadro 10.1.

Quadro 10.1 – Métodos da classe `dict`

Método	Descrição
`.clear()`	Remove todos os elementos do dicionário.
`.copy()`	Retorna uma cópia do dicionário.
`.fromkeys(iteravel [,v])`	Cria um novo dicionário usando os elementos do objeto `iteravel` como chave. Se o segundo parâmetro for fornecido será usado como valor para todos os elementos do novo dicionário.
`.get(key [,default])`	Retorna o valor associado com a chave `key` passada como parâmetro. Caso a chave não esteja presente retorna o segundo parâmetro `default`. O segundo parâmetro é opcional e na sua ausência o retorno será `None`.
`.items()`	Retorna o conjunto de elementos do dicionário na forma de tuplas contendo o par chave-valor de cada um. Muito útil para usar em iterações com o dicionário.
`.keys()`	Retorna uma coleção com todas as chaves dos elementos contidos no dicionário.
`.pop(key [,default])`	Remove a chave `key` do dicionário e retorna o valor a ela associado. Caso a chave não esteja presente retorna o segundo parâmetro `default`. Se `key` não existe e `default` não foi especificado gera a exceção `KeyError`.
`.popitem()`	Remove um elemento do dicionário, retornando-o na forma de um par chave-valor. Os elementos são retornados na ordem de uma pilha LIFO, ou seja, o último elemento a entrar é o primeiro a ser retirado. Se o dicionário estiver vazio gera a exceção `KeyError`
`.setdefault(key [,d])`	Se a chave `key` está no dicionário, retorna o seu valor. Senão, insere `key` com o valor `default` e retorna `default`. Por padrão default é o valor `None`. Este método representa uma forma alternativa de inicializar um dicionário
`.update(origem)`	Atualiza o dicionário a partir dos itens contidos no parâmetro `origem`. Se algum item de `origem` não estiver presente então será incluído; se estiver será atualizado.
`.values()`	Retorna uma coleção com todos os valores dos elementos contidos no dicionário.

Exemplo 10.4

```
Este exemplo é muito longo, pois mostra os métodos da classe dict.
  Então seu código não foi colocado aqui. O vídeo está disponível.
```

(exemplo interativo feito com IDE Idle)

10.4 ITERAÇÕES COM DICIONÁRIOS

O uso mais frequente de dicionários é como iterador em um comando `for`. Há três casos típicos.

10.4.1 CASO 1 – ITERAÇÃO SOBRE AS CHAVES

O exemplo 10.5 mostra a iteração sobre as chaves.

Exemplo 10.5

```
Primeira versão
Cores = {1: 'azul', 2: 'verde', 3: 'amarelo', 4: 'vermelho'}
print('Exibição do dicionário - Caso 1')
for x in Cores.keys():              # Cores.keys() foi usado explicitamente
    print(f'  {x} - {Cores[x]}')    # Cores[x] dá acesso ao valor associado a x
print('Fim do Programa')
```

```
Segunda versão
Cores = {1: 'azul', 2: 'verde', 3: 'amarelo', 4: 'vermelho'}
print('Exibição do dicionário - Caso 1')
for x in Cores:                     # Cores.keys() foi usado implicitamente
    print(f'  {x} - {Cores[x]}')
print('Fim do Programa')
```

```
Exibição do dicionário - Caso 1
  1 - azul
  2 - verde
  3 - amarelo
  4 - vermelho
Fim do Programa
```

Este exemplo mostra duas versões do programa com uma única diferença na linha do comando `for`. O objeto `x` é o objeto de controle do comando `for` e a cada iteração ele recebe a chave de um elemento contido na lista. O conjunto de chaves é fornecido pelo método `.keys()` e esse método é padrão (default) no dicionário de modo que escrever uma das alternativas abaixo dá o mesmo resultado.

```
for x in Cores.keys():
for x in Cores:
```

Além disso, uma vez que `x` contém a chave de um elemento é possível acessar seu valor com:

```
Cores[x]
```

E isso foi usado no momento de exibir o resultado na tela.

10.4.2 CASO 2 – ITERAÇÃO SOBRE OS VALORES

Há situações nos programas em que não estamos interessados nas chaves dos elementos do dicionário. Ao invés disso estamos interessados apenas nos valores.

Exemplo 10.6

```
Cores = {1: 'azul', 2: 'verde', 3: 'amarelo', 4: 'vermelho'}
print('Exibição do dicionário - Caso 2')
for cor in Cores.values():
    print(f'  {cor}')
print('Fim do Programa')
```

```
Exibição do dicionário - Caso 2
  azul
  verde
  amarelo
  vermelho
Fim do Programa
```

Para casos assim podemos usar o método `.values()` para iterar diretamente sobre os valores. Neste exemplo o objeto `cor` recebe diretamente os nomes das cores e os utiliza no processamento.

10.4.3 CASO – ITERAÇÃO CONJUNTA CHAVE-VALOR

Esta terceira situação também é muito frequente. Usando o método `.items()` podemos iterar simultaneamente com as chaves e os valores.

Exemplo 10.7

```
Cores = {1: 'azul', 2: 'verde', 3: 'amarelo', 4: 'vermelho'}
print('Exibição do dicionário - Caso 3')
for numero, nome in Cores.items():
    print(f'  nº da cor = {numero} - nome da cor = {nome}')
print('Fim do Programa')
```

```
Exibição do dicionário - Caso 3
  nº da cor = 1 - nome da cor = azul
  nº da cor = 2 - nome da cor = verde
  nº da cor = 3 - nome da cor = amarelo
  nº da cor = 4 - nome da cor = vermelho
Fim do Programa
```

Nesta solução aparecem dois objetos usados no controle do comando `for` desta forma:

```
for numero, nome in Cores.items():
```

Mas cuidado, não interprete errado. Você não está vendo "dois objetos" como controle do comando `for`. De fato, o Python é cheio de sutilezas e passo-a-passo você precisa aprender os conceitos e assim vai entender tais sutilezas.

Na prática o par (numero, nome) assim colocado no `for` é interpretado pelo Python como sendo uma tupla que recebe a tupla retornada pelo método `.items()`. Portanto o conceito é esse: se o retorno do método `.items()` é uma tupla, então o objeto que a recebe também será uma tupla.

Assim, a linha acima é o mesmo que escrever a linha a seguir:

```
for (numero, nome) in Cores.items():
```

ou ainda deste modo

```
for tupla in Cores.items():
```

Se usarmos este último veja como fica o programa:

Exemplo 10.8 — Teste este código no PyCharm

```
Cores = {1: 'azul', 2: 'verde', 3: 'amarelo', 4: 'vermelho'}
print('Exibição do dicionário - Caso 3')
for tupla in Cores.items():   # veja o uso do identificador 'tupla'
    print(f'  nº da cor = {tupla[0]} - nome da cor = {tupla[1]}') # e atenção aqui
print('Fim do Programa
```

```
Exibição do dicionário - Caso 3
  nº da cor = 1 - nome da cor = azul
  nº da cor = 2 - nome da cor = verde
  nº da cor = 3 - nome da cor = amarelo
  nº da cor = 4 - nome da cor = vermelho
Fim do Programa
```

Repare no uso do identificador 'tupla' nesse exemplo 10.8, bem como na forma como foi usado no `print`.

10.5 Exercícios resolvidos com Dicionários

Exercício Resolvido 10.1

Enunciado: *Escreva um programa que leia do teclado o código de um produto e seu preço unitário. O código é um string e o preço é real. Acrescente o par código:preço em um dicionário. O laço termina quando for fornecido um string vazio para o código. Ao final, exibir código e preço, um produto em cada linha.*

```
produtos = {}
print('Leitura dos dados')
cod = input('  Digite o código: ')   # lê o primeiro código
while cod != '':                     # se cod diferente de vazio segue no laço
    preco = float(input('  Digite o preço: ')) # lê o preço
    produtos[cod] = preco                      # acrescenta novo item ao dicionário
    cod = input('  Digite o código: ')         # lê o próximo cod
print('Fim da leitura dos dados\n')
print('Preço dos Produtos')
for cod in produtos.keys():  # faz iteração como Caso 1 - usando .keys()
    print(f'  produto {cod} custa R$ {produtos[cod]:7.2f}')
print("\nFim do programa")
```

```
Leitura dos dados
  Digite o código: 7899099016244
  Digite o preço: 16.9
  Digite o código: 7899099001288
  Digite o preço: 8.43
  Digite o código: 7899099002544
  Digite o preço: 166.35
  Digite o código: 7899099006931
  Digite o preço: 53.8
  Digite o código:
Fim da leitura dos dados

Preço dos Produtos
  produto 7899099016240 custa R$   16.90
  produto 7899099001280 custa R$    8.43
  produto 7899099002541 custa R$  166.35
  produto 7899099006936 custa R$   53.80

Fim do programa
```

Neste exercício foi fornecido um código com 13 dígitos que se assemelha a um código de barras e o preço foi exibido formatado ocupando 7 posições de tela, com 2 casas decimais.

Observe que nenhum cuidado foi tomado a respeito de eventual verificação se o produto já está no dicionário ou não. Desse modo, se o mesmo código for fornecido duas vezes com preços diferentes, o preço que vai prevalecer será o último. Esta questão será considerada no próximo exercício resolvido.

Um último comentário neste exercício é sobre a lógica implementada: note que fizemos uso da técnica de leitura do dado de controle na saída do laço, ou seja, a leitura do código do produto é última instrução dentro do laço. Por consequência, a primeira leitura deve ser feita antes do laço iniciar.

Exercício Proposto 10.1

Enunciado: *Altere a solução do exercício resolvido 10.1 para fazer a iteração com o método .items()*

Exercício Resolvido 10.2

Enunciado: *Escreva um programa que leia do teclado o código de um produto e seu preço unitário. O código é um string e o preço é real. Acrescente o par código:preço em um dicionário. O programa deve verificar se o código já está no dicionário e neste caso deve emitir uma mensagem de erro. O laço termina quando for fornecido um string vazio para o código. Ao final, exibir código e preço, um produto em cada linha.*

```
produtos = {}
print('Leitura dos dados')
while True:
    cod = input('  Digite o código: ')
    if cod == '':
        break         # interrompe o laço se cod == ''
    elif cod in produtos:
        print(f'  ...o código {cod} já está no cadastro')
        continue      # segue para a próxima iteração
    preco = float(input('  Digite o preço: '))
    produtos[cod] = preco    # acrescenta novo item ao dicionário
print('Fim da leitura dos dados\n')
print('Preço dos Produtos')
for cod in produtos.keys():
    print(f'  produto {cod} custa {produtos[cod]}')
print("\nFim do programa")
```

```
Leitura dos dados
  Digite o código: 7899099016240
  Digite o preço: 16.9
  Digite o código: 7899099001280
  Digite o preço: 8.43
  Digite o código: 7899099002541
  Digite o preço: 166.35
  Digite o código: 7899099001280
  ...o código 7899099001280 já está no cadastro
  Digite o código: 7899099016240
  ...o código 7899099016240 já está no cadastro
  Digite o código: 7899099006936
  Digite o preço: 53.8
  Digite o código:
Fim da leitura dos dados

Preço dos Produtos
  produto 7899099016240 custa 16.9
  produto 7899099001280 custa 8.43
  produto 7899099002541 custa 166.35
  produto 7899099006936 custa 53.8

Fim do programa
```

Nesta solução a forma de uso do dicionário é a mesma do exercício resolvido anterior. Porém, foi utilizada uma lógica bem diferente. Observe o comando `while` e o uso dos comandos `break` e `continue`.

A condição do `while` é sempre verdadeira. Quando o código é um string vazio o comando `break` interrompe esse `while`. Quando o código já está no cadastro o comando `continue` pula o restante do código e segue para a próxima repetição na qual será lido um novo código.

Exercício Resolvido 10.3

Enunciado: *Escreva um programa que leia dados dos Estados brasileiros: Sigla, Nome, Capital e PIB. A Sigla deve ser usada como chave para o dicionário e o valor deve ser uma tupla formada com (Nome, Capital, PIB). A leitura termina quando um string vazio for fornecido para a Sigla. Exibir os dados na tela.*

```
UF = {}
print('Leitura dos dados')
while True:
    sigla = input('  Digite a sigla: ')
    if sigla == '':
        break        # interrompe o laço se cod == ''
    elif sigla in UF:
        print(f'  ...o código {sigla} já está no cadastro')
        continue     # segue para a próxima iteração
    estado = input('  Digite o nome: ')
    capital = input('  Digite a capital: ')
    pib = float(input('  Digite o PIB: '))
    UF[sigla] = (estado, capital, pib)  # acrescenta novo item ao dicionário
print('Fim da leitura dos dados\n')

print('     {:15} {:15} {:>10}'.format('Estado', 'Capital', 'PIB (R$ bi)'))
for sigla, dados in UF.items():
    print('({}) {:15} {:15} {:10.1f}'.format(
        sigla,
        dados[0],
        dados[1],
        dados[2] )
    )
print("\nFim do programa")
```

```
Leitura dos dados
  Digite a sigla: AC
  Digite o nome: Acre
  Digite a capital: Rio Branco
  Digite o PIB: 21.4
  Digite a sigla: CE
  Digite o nome: Ceará
  Digite a capital: Fortaleza
  Digite o PIB: 194.9
  Digite a sigla: MT
  Digite o nome: Mato Grosso
  Digite a capital: Cuiabá
  Digite o PIB: 233.4
  Digite a sigla: PR
  Digite o nome: Paraná
  Digite a capital: Curitiba
  Digite o PIB: 549.9
  Digite a sigla: SP
  Digite o nome: São Paulo
  Digite a capital: São Paulo
  Digite o PIB: 2719.8
  Digite a sigla:
Fim da leitura dos dados

     Estado          Capital         PIB (R$ bi)
(AC) Acre            Rio Branco            21.4
(CE) Ceará           Fortaleza            194.9
(MT) Mato Grosso     Cuiabá               233.4
(PR) Paraná          Curitiba             549.9
(SP) São Paulo       São Paulo           2719.8

Fim do Programa
```

Veja os comentários após o exercício 10.4, que tem o mesmo enunciado, porém solução diferente.

Exercício Resolvido 10.4

Enunciado: *Escreva um programa que leia dados dos Estados brasileiros: Sigla, Nome, Capital e PIB. A Sigla deve ser usada como chave para o dicionário e o valor deve ser um dicionário aninhado contendo os objetos Nome, Capital e PIB. Um string vazio para a Sigla termina a leitura. Exibir os dados na tela.*

```
UF = {}
print('Leitura dos dados')
while True:
    sigla = input('  Digite a sigla: ')
    if sigla == '':
        break        # interrompe o laço se cod == ''
    elif sigla in UF:
        print(f'  ...o código {sigla} já está no cadastro')
        continue     # segue para a próxima iteração
    estado = input('  Digite o nome: ')
    capital = input('  Digite a capital: ')
    pib = float(input('  Digite o PIB: '))
    dicItem = {'nome': estado, 'capital': capital, 'pib': pib}
    UF[sigla] = dicItem    # acrescenta novo item ao dicionário
print('Fim da leitura dos dados\n')

print('     {:15} {:15} {:>10}'.format('Estado', 'Capital', 'PIB (R$ bi)'))
for sigla, dados in UF.items():
    print('({}) {:15} {:15} {:10.1f}'.format(
        sigla,
        dados['nome'],
        dados['capital'],
        dados['pib'] )
    )
print("\nFim do Programa")
```

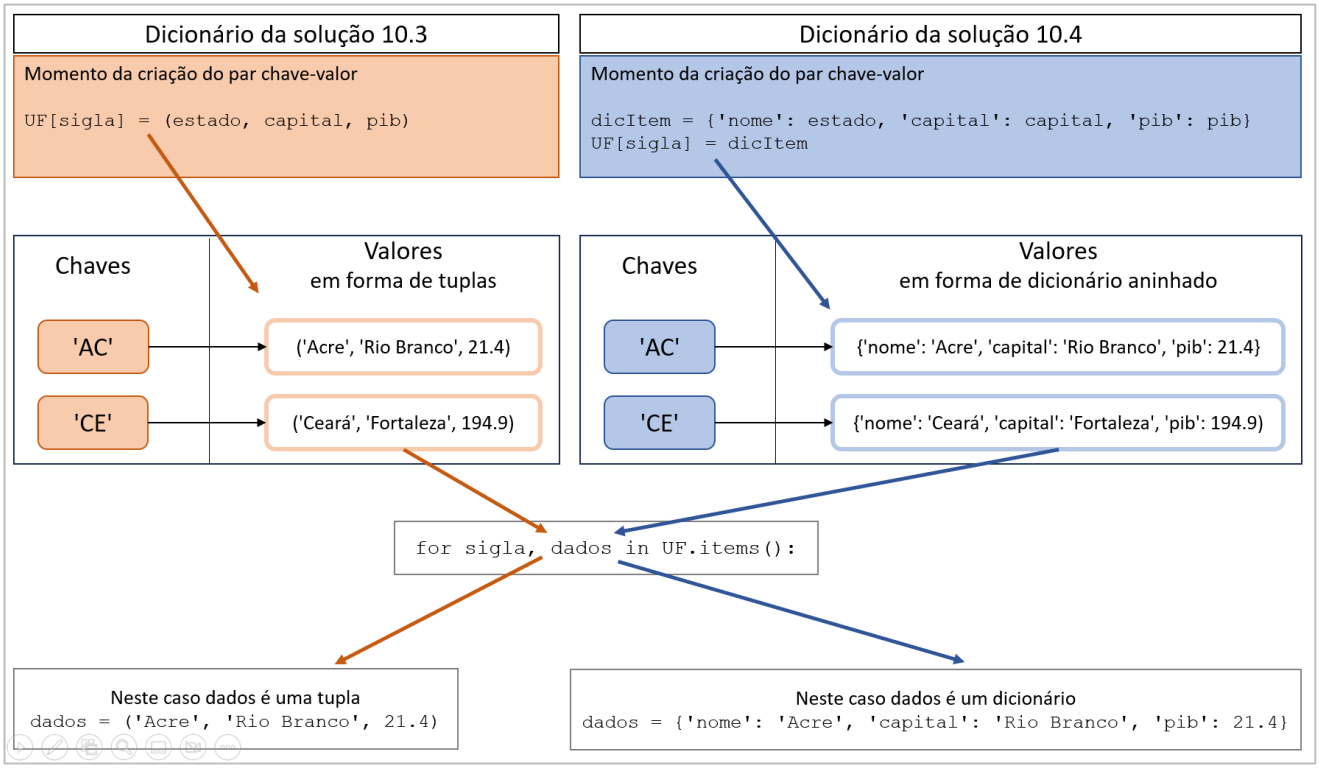
```
Leitura dos dados
  Digite a sigla: AC
  Digite o nome: Acre
  Digite a capital: Rio Branco
  Digite o PIB: 21.4
  Digite a sigla: CE
  Digite o nome: Ceará
  Digite a capital: Fortaleza
  Digite o PIB: 194.9
  Digite a sigla: MT
  Digite o nome: Mato Grosso
  Digite a capital: Cuiabá
  Digite o PIB: 233.4
  Digite a sigla: PR
  Digite o nome: Paraná
  Digite a capital: Curitiba
  Digite o PIB: 549.9
  Digite a sigla: SP
  Digite o nome: São Paulo
  Digite a capital: São Paulo
  Digite o PIB: 2719.8
  Digite a sigla:
Fim da leitura dos dados

     Estado          Capital         PIB (R$ bi)
(AC) Acre            Rio Branco             21.4
(CE) Ceará           Fortaleza             194.9
(MT) Mato Grosso     Cuiabá                233.4
(PR) Paraná          Curitiba              549.9
(SP) São Paulo       São Paulo            2719.8

Fim do Programa
```

Este é um comentário conjunto, válido para as soluções 10.3 e 10.4, as quais tem o mesmo enunciado, porém soluções distintas. Ambas usam a sigla como `chave` e a diferença relevante está na forma como foi estruturado o `valor` de cada elemento do dicionário. Essa diferença de estrutura implica em conhecermos dois aspectos da questão:

- Como criar o elemento do dicionário;
- Como usar o elemento do dicionário.

A figura 10.3 ilustra a situação.

Figura 10.3



fonte: o Autor

Momento da criação - em 10.3 o `valor` é uma tupla;

Isso implica que no momento da criação do elemento deverá ser usada a construção onde o lado direito da atribuição é uma tupla.

```
UF[sigla] = (estado, capital, pib)
```

Momento da criação - em 10.4 o valor é um dicionário aninhado.

Isso implica que no momento da criação do elemento deverá ser usada a construção onde o lado direito da atribuição é um dicionário.

```
dicItem = {'nome': estado, 'capital': capital, 'pib': pib}
UF[sigla] = dicItem
```

No que diz respeito ao uso do dicionário, em ambos utiliza-se o iterador `for` da forma abaixo:

```
for sigla, dados in UF.items():
```

Momento do uso - em 10.3 o `valor` é uma tupla;

Neste caso, `dados` será uma tupla contendo 3 elementos e deverá ser usado com índices:

```
print('({}) {:15} {:15} {:10.1f}'.format(
    sigla,
    dados[0],
    dados[1],
    dados[2] )
```

o acesso aos elementos da Tupla é feito com índices

Momento do uso - em 10.4 o `valor` é um dicionário aninhado;

Neste caso, `dados` será um dicionário contendo 3 elementos e deverá ser usado com `chave`:

```
print('({}) {:15} {:15} {:10.1f}'.format(
    sigla,
    dados['nome'],
    dados['capital'],
    dados['pib'] )
```

o acesso aos elementos do Dicionário é feito com as chaves usadas para criá-lo

Esteja atento a esses pontos e a cada novo problema escolha a estratégia que você considerar mais adequada à situação.

Exercício Resolvido 10.5 — Teste este código no PyCharm

Enunciado: *Considere o seguinte conjunto de dados: Nome + (N1, N2, N3, N4). Nome representa o nome de um aluno e deve ser usado como chave. N1, N2, N3, N4 representam as notas de provas desse aluno.*

Escreva um programa que leia os dados de Q alunos e determine a situação de cada aluno. O critério que garante a aprovação é que a média aritmética das 4 notas de prova seja maior ou igual 6,0. Q é a quantidade de alunos e este valor deve ser lido do teclado no começo do programa.

Para cada aluno o nome deve ser lido em separado e suas notas de prova devem ser lidas juntas na mesma linha, com um espaço em branco de separação.

Para cada aluno o programa deve mostras o Nome, as 4 notas de prova, a média final e a situação (aprovado/reprovado). As notas devem ser exibidas com uma casa decimal.

```
Alunos = {}
Q = int(input('Digite a qtde de alunos: '))
for i in range(Q):
    nome = input(f'  nome do aluno {i+1} >> ')
    notas = input(f'  notas do {nome} >> ')
    notas = notas.split()
    for j in range(len(notas)):
        notas[j] = float(notas[j])
    Alunos[nome] = tuple(notas)

print('\nResultado da Turma')
for nome, notas in Alunos.items():
    media = sum(notas)/len(notas)
    if media >= 6.0:
        situacao = 'aprovado'
    else:
        situacao = 'reprovado'
    print('{:12} {:4.1f} {:4.1f} {:4.1f} {:4.1f} {} com média = {:4.1f}'.format(
        nome,
        notas[0],
        notas[1],
        notas[2],
        notas[3],
        situacao,
        media)
    )
print('Fim do Programa')
```

```
Digite a qtde de alunos: 3
  nome do aluno 1 >> Tio Patinhas
  notas do Tio Patinhas >> 7.5  6.5  7.0  6.5
  nome do aluno 2 >> Pato Donald
  notas do Pato Donald >> 6.5  3.5  5.0  5.5
  nome do aluno 3 >> Margarida
  notas do Margarida >> 8.5  7.5  8.0  7.0

Resultado da Turma
Tio Patinhas  7.5  6.5  7.0  6.5 aprovado com média =  6.9
Pato Donald   6.5  3.5  5.0  5.5 reprovado com média =  5.1
Margarida     8.5  7.5  8.0  7.0 aprovado com média =  7.8
Fim do Programa
```

Nesta solução usamos uma tupla como valor para os elementos do dicionário. Reproduza esse programa e teste seu funcionamento no PyCharm.

No próximo capítulo vamos usar mais vezes os dicionários, para armazenar dados que serão lidos de arquivos em disco.

Exercício Proposto 10.2

Enunciado: *Escreva um programa que permaneça em laço lendo do teclado números inteiros entre 1 e 9. Outros valores devem ser ignorados. Esse laço termina quando for digitado zero ou qualquer valor negativo. O objetivo deste programa é contar quantas vezes cada valor entre 1 e 9 foi digitado.*

Ao término do laço de leitura o programa deve mostrar quais valores foram digitados e quantas vezes cada um. Obrigatoriamente use um dicionário.

Exercício Proposto 10.3

Enunciado: *Escreva um programa que leia e armazene em um dicionário os seguintes dados dos seus contatos: nome, número celular, email e data de aniversário.*

A chave deve ser o nome. ***O valor deve ser uma tupla contendo os demais dados****. Se o nome já existir no dicionário o programa deve perguntar se o usuário deseja alterar o cadastro.*

Ao digitar um string vazio para o nome, o programa interrompe a leitura. Antes de encerrar o programa apresente os dados em um formato de tabela.

Exercício Proposto 10.4

Enunciado: *Escreva um programa que leia e armazene em um dicionário os seguintes dados dos seus contatos: nome, número celular, email e data de aniversário.*

A chave deve o nome. ***O valor deve ser um dicionário aninhado contendo os demais dados****. Se o nome já existir no dicionário o programa deve perguntar se o usuário deseja alterar o cadastro.*

Ao digitar um string vazio para o nome, o programa interrompe a leitura. Antes de encerrar o programa apresente os dados em um formato de tabela.

Exercício Proposto 10.5

Enunciado: *Escreva um programa que leia e armazene em um dicionário os seguintes dados dos seus contatos: nome, número celular, email e data de aniversário.*

A chave deve o nome. O valor pode ser uma tupla ou um dicionário aninhado. Você escolhe.

Ao digitar um string vazio para o nome, o programa interrompe a leitura e apresente todos dados na tela na mesma formatação dos exercícios anteriores.

Neste exercício os nomes devem estar em ordem alfabética.

Dica

Use a função sorted() de Python.

```
# Considere o dicionário:
>>> UF = {
        'MT': {'nome': 'Mato Grosso', 'capital': 'Cuiabá', 'pib': 233.4},
        'PR': {'nome': 'Paraná', 'capital': 'Curitiba', 'pib': 549.9},
        'CE': {'nome': 'Ceará', 'capital': 'Fortaleza', 'pib': 194.9},
        'SP': {'nome': 'São Paulo', 'capital': 'São Paulo', 'pib': 2719.8},
        'AC': {'nome': 'Acre', 'capital': 'Rio Branco', 'pib': 21.4}
    } # não estão em ordem de sigla
>>> Siglas = sorted(UF) # ordena as chaves do dicionário
>>> Siglas
    ['AC', 'CE', 'MT', 'PR', 'SP']
(exemplo interativo feito com IDE Idle)

Teste no Idle e depois use no programa.
```

Exercício Proposto 10.6

Enunciado: *Escreva um programa para registrar os seguintes dados de uma frota de veículos de uma empresa:*

Placa (string – chave – obrigatório todas as letras maiúsculas)

Marca

Modelo

Tipo (caminhão, furgão, automóvel, motocicleta, etc)

Kilometragem

Data da Compra (string no formato AAAAMMDD – ano,mês,dia)

O programa deve ficar em laço enquanto a Placa for digitada. O laço termina quando for digitado FIM para a placa. Se for digitada uma placa com letras minúsculas o programa deve convertê-las para maiúsculas com o método .upper().

Para cada veículo leia todos os dados e carregue em um dicionário. Se uma placa já existente for digitada o programa deve avisar que já existe, mostrar seus dados e se usuário quiser fazer alteração em algum dado basta digitar o novo valor. Para os campos em que nada for digitado deve ser mantido o valor já cadastrado.

Ao final do laço mostre os dados na tela com uma formatação legível.

Desafio *Inclua no programa uma validação da placa, seguindo as seguintes regras:*

a) *Deve ter 7 caracteres*

b) *Os três primeiros devem ser letras*

c) *Os caracteres 4, 6 e 7 devem ser algarismos*

d) *O caractere 5 pode ser número (placa antiga) ou letra (nova placa padrão Mercosul)*

Capítulo 11
ARQUIVOS

11.1 ENTENDENDO OS ARQUIVOS

A esta altura do curso você já deve ter percebido que cresceu consideravelmente o volume de dados que estamos digitando e processando nos exemplos e exercícios. É natural que isso aconteça, principalmente após a introdução dos tópicos sobre objetos de classes estruturadas, como listas e dicionários. Afinal, classes assim existem exatamente para trabalhar com grandes volumes de dados.

Por outro lado, cada programa que fizemos até o momento exige que todos os dados sejam digitados, em seguida são processados e por fim os resultados exibidos em tela. Ao término do programa eles são perdidos, pois estavam apenas na memória do computador, não tendo sido salvos em lugar algum.

Deste modo, está na hora de aprender algum recurso que permita o salvamento de dados e por consequência, permita também a leitura deles quando necessário.

Existem basicamente duas tecnologias para salvamento e recuperação de dados: os Arquivos e os Sistemas Gerenciadores de Bancos de Dados (SGBD). Neste capítulo vamos aprender a trabalhar com Arquivos em Python.

Um arquivo de computador é um recurso de armazenamento de dados disponível em todos os tipos de dispositivos digitais, sejam computadores de qualquer porte ou dispositivos móveis. Para cumprir o objetivo de tornar permanente o armazenamento dos dados, os arquivos devem ser gravados em algum equipamento físico que não dependa de um suprimento constante de energia. Atualmente os equipamentos (hardware) mais usados para isto são os discos magnéticos e os dispositivos dotados de memória flash (SSD).

Em todos os sistemas computacionais quem fica responsável pelo acesso, gravação e leitura nos dispositivos físicos é o software básico, ou Sistema Operacional. Isso garante maior segurança, padronização e organização das unidades de armazenamento.

As linguagens de programação, por sua vez, são dotadas de comandos para realizar as operações com arquivos, mas dependem do Sistema Operacional para executar as operações no hardware. Assim, os comandos existentes nas linguagens tem o papel de acionar as funções de arquivos previstas no Sistema

Operacional. Esta forma de colaboração entre linguagem de programação e Sistema Operacional traz uma grande vantagem aos programadores, pois não precisam conhecer os muitos detalhes que envolvem as operações de acesso ao hardware para executar a tarefa de leitura e gravação de um arquivo.

Se tivéssemos condição de "olhar" um arquivo gravado no disco "veríamos", na prática, uma coleção de bytes rotulada com um nome. Os bytes do arquivo representam os dados de seu conteúdo. Quando esses bytes são lidos eles podem ser interpretados da maneira que quisermos, mas há necessidade de que essa interpretação seja adequada. Por exemplo, se os bytes gravados representam dados de uma imagem no formato .png e ao fazer a leitura você os interpretar como dados de uma música .mp3 vai dar tudo errado e, no máximo você ouvirá ruído.

Deste modo, ao ler um arquivo do dispositivo você precisa conhecer sua natureza, ou, a estrutura dos seus bytes. Falando sobre essa estrutura do arquivo saiba que há duas categorias essenciais: os arquivos texto e os arquivos binários.

Arquivos texto

Nos arquivos texto os bytes representam caracteres legíveis. Esse tipo de arquivo é muito usado para gravação de coisas como: código fonte de programas; arquivos HTML e CSS de páginas web; arquivos XML e CSV (comma separated values) usados para intercâmbio de dados entre sistemas; arquivos TXT em geral. Em termos práticos, todo arquivo texto pode ser diretamente aberto e lido em programas como o Bloco de Notas do Windows ou o Notepadqq do Linux.

Arquivos binários

Os arquivos binários, por sua vez, ao serem abertos e lidos devem ter seus bytes devidamente interpretados. Como exemplo de arquivos binários comuns temos vários tipos: imagens, áudio, vídeo, bancos de dados, programas executáveis compilados, arquivos compactados através de um algoritmo de compressão, etc. Cada um desses tipos tem formato próprio para os bytes e esse formato deve ser considerado e interpretado no momento da leitura.

Para quem está começando em programação, trabalhar diretamente com os bytes de arquivos binários é mais complicado, devido a essa multiplicidade de formatos existentes.

Neste e-book trabalharemos apenas com arquivos texto.

11.2 CODIFICAÇÃO DE ARQUIVOS TEXTO

Os arquivos texto são bem mais simples que os arquivos binários, mas eles também existem em diferentes versões. Precisamos conhecer o que é denominado "codificação do arquivo texto".

A base dessa codificação são as tabelas de caracteres.

A memória do computador comporta apenas bits 0 e 1. Grupos de 8 bits formam 1 byte. Bits e bytes, porém, são informações de natureza numérica e precisamos de alguma forma de estabelecer uma relação entre o número inteiro de um byte e letras, algarismos e sinais de pontuação como 'A', 'B', 'a', 'b', '0', '1', '+', '*', ou qualquer outro.

Para isso foram criadas tabelas de correspondência entre números inteiros e caracteres. Historicamente a primeira dessas tabelas foi a "Tabela ASCII" (*American Standard Code for Information*

Interchange). Mas essa tabela é limitada pois permite a codificação de apenas 256 caracteres, em seu limite máximo – que é o limite de combinações para 1 byte.

Quando os sistemas computacionais estavam restritos ao idioma inglês a tabela ASCII supria as necessidades. Com o tempo surgiu a necessidade de suportar outros idiomas nos sistemas computacionais, como o português ('é', 'É', 'á', 'ô', 'Â', 'ç', 'Ç', etc); todos os idiomas latinos; os alfabetos grego, cirílico, árabe; os idiomas japonês, chinês, coreano; e isso é só uma parte.

Para suprir toda essa gama de idiomas foi criado o padrão Unicode.

Para conhecer mais o Unicode acesse
página inicial: https://home.unicode.org/
tabelas: https://unicode.org/charts/

Do padrão Unicode deriva um outro padrão denominado UTF-8 que é muito usado nos idiomas de origem latina e comporta todos os caracteres acentuados que eles contém (essa definição não é perfeita, mas atende às nossas necessidades neste texto).

Assim, nossos arquivos texto estarão codificados ou como ASCII ou como UTF-8.

Esse assunto é extenso e recheado de detalhes que não cabem no escopo deste texto, porém o que precisamos saber é que para trabalharmos com arquivos texto em Python precisaremos usar a codificação certa. Sobre isso existem duas situações:

- No momento da **gravação do arquivo**: precisamos escolher uma codificação para gravar;
- No momento da **leitura do arquivo**: precisamos saber com qual codificação ele foi gravado e fazer a leitura usando a codificação certa.

No momento da leitura, se usarmos a codificação errada pode acontecer do programa dar erro e ser interrompido ou de termos os caracteres errados na memória. Este ponto é mostrado na prática no exercício resolvido 11.7.

Agora passamos aos aspectos práticos do trabalho com arquivos.

11.3 Abertura e fechamento do arquivo

Para ler ou gravar arquivos a primeira tarefa é abri-lo. Em Python, para abrir um arquivo deve-se utilizar a função `open` que contém diversos parâmetros como mostrado a seguir. O único parâmetro obrigatório é o `nome` do arquivo. Além do `nome`, os parâmetros do nosso interesse são `modo` e `encoding`.

```
arquivo = open(nome, modo, buffering, encoding, errors, newline, closefd, opener)
```

O retorno da função `open` pode ser atribuído a um objeto. No caso acima usamos o identificador `arquivo`.

Parâmetro `nome`

Neste parâmetro passamos o nome do arquivo. Esse nome deve seguir as regras de nomes de arquivos do sistema operacional onde o programa será executado, podendo incluir a letra do dispositivo e nomes de pastas e subpastas onde será gravado. Se dispositivo e pasta não estiverem presentes o arquivo será gravado na mesma pasta onde está o programa.

Parâmetro `modo`

Este parâmetro define qual a finalidade da abertura do arquivo conforme o quadro 11.1.

Quadro 11.1 – Opções de modo de abertura de arquivos

Grupo	Caractere	Significado
G1	**r**	Abre para leitura. Esta opção é padrão e será assumida se nada for especificado.
	w	Abre para escrita, eliminando todo o seu conteúdo (não haverá pedido de confirmação).
	x	Abre para criação exclusiva. S o arquivo já existir o comando gera um erro.
	a	Abre para escrita, e caso o arquivo já exista mantém seu conteúdo e anexa novos dados ao final.
G2	**b**	Abre o arquivo no modo binário.
	t	Abre o arquivo no modo texto. Esta opção é padrão e será assumida se nada for especificado.
	+	Aberto para atualização. Permite leitura e escrita concomitantes.

Grupo 1

Os modos do grupo 1 são mutuamente exclusivos, ou seja, você deve escolher um deles.

Se nenhum for especificado `r` é o padrão.

Grupo2

Os modos do grupo 2 são mutuamente exclusivos – `b` para arquivos binários e `t` para arquivos texto.

Se nenhum for especificado `t` é o padrão.

Opção +

Ao usar essa opção você poderá ler e gravar no arquivo na mesma operação de abertura.

Parâmetro `encoding`

Este parâmetro só se aplica a arquivos texto e diz respeito à codificação descrita na seção 11.2.

As duas opções frequentemente usadas nos idiomas latinos são `'ANSI'` e `'UTF-8'`, escritas exatamente dessa forma, em letras maiúsculas.

Demais parâmetros

Os demais parâmetros – `buffering`, `errors`, `newline`, `closefd`, `opener` – fogem ao escopo deste texto, bastando dizer que seus valores padrão atendem às necessidades dos nossos exemplos e exercícios.

Em qualquer linguagem de programação
arquivos sempre devem ser abertos, usados e fechados.
Não esqueça de fechar, pois pode resultar em arquivo corrompido.

Terminado o trabalho com um arquivo é preciso fechá-lo. Isso é feito com o método `.close()`.

```
arquivo.close()
```

11.4 MÉTODOS DOS OBJETOS DE ARQUIVO

A classe que define arquivos em Python tem o nome "`_.io.TextIOWrapper`". O quadro 11.2 mostra os métodos mais usados dessa classe.

Quadro 11.2 – Métodos da classe `_io.TextIOWrapper`

Método	Descrição
`.close()`	Fecha o arquivo que foi aberto com `open`. Se o arquivo está aberto para gravação, primeiro descarrega seu buffer.
`.flush()`	Descarrega o buffer de um arquivo aberto para gravação, sem fechá-lo.
`.read(`*tamanho*`)`	Lê e retorna a quantidade `tamanho` de caracteres. Se `tamanho` for negativo ou None lê todos os caracteres do arquivo.
`.readline()`	Lê uma linha do arquivo e avança o cursor para o início da próxima. Retorna um string com o conteúdo da linha, incluindo o caractere '\n' no final se ele estiver presente no arquivo.
`.readlines()`	Lê todas as linhas do arquivo e retorna-as como uma lista de strings, incluindo o '\n' no final de cada uma. É preciso ter cuidado com essa opção no caso de arquivos muito grandes.
`.write(`*dados*`)`	Grava no arquivo o string `dados`. É obrigatório que dados seja da classe string.
`.writelines(`*lst*`)`	Grava no arquivo todos os strings contidos na lista `lst`.

11.5 EXERCÍCIOS RESOLVIDOS USANDO ARQUIVOS

Neste primeiro exercício resolvido vamos gravar uma sequência de números reais em um arquivo.

Exercício Resolvido 11.1

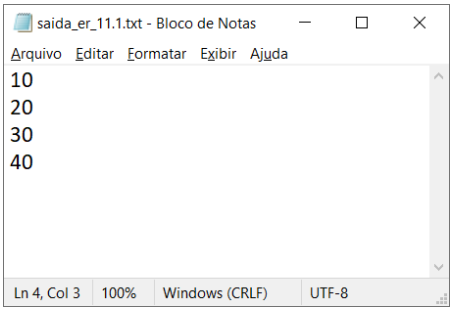
Enunciado: *Escreva um programa que permaneça em laço lendo números inteiros até que seja digitado 0. Todos os valores digitados, exceto o zero, devem ser gravados em um arquivo em disco, um por linha.*

Usar o método `.write()`

```
arq = open('saida_er_11.1.txt', 'w')        # abre o arquivo para gravação
A = int(input('Digite um inteiro: '))
while A != 0:
    arq.write(f'{A}\n')                     # grava uma linha do arquivo
    A = int(input('Digite um inteiro: '))
arq.close()                                 # fecha o arquivo
print('Fim do Programa')
```

```
Digite um inteiro: 10
Digite um inteiro: 20
Digite um inteiro: 30
Digite um inteiro: 40
Digite um inteiro: 0
Fim do Programa
```

A imagem ao lado mostra o resultado da gravação feita neste exercício resolvido. Para gravar o arquivo '`saida_er_11.1.txt`' usamos o método `.write()`. Observe no código que o string a ser gravado no arquivo inclui o caractere de pulo de linha '`\n`'. Sem essa inclusão todos os números gravados no arquivo ficarão na mesma linha. Isso é mostrado no vídeo. A imagem ao lado mostra o resultado.



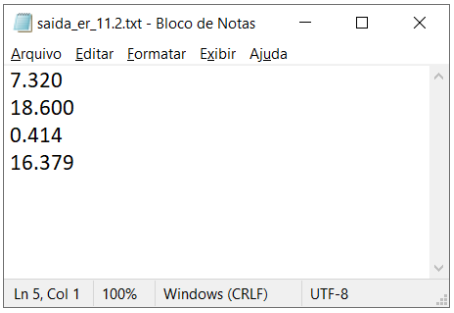
Exercício Resolvido 11.2 Teste este código no PyCharm

Enunciado: *Escreva um programa que permaneça em laço lendo números reais até que seja digitado 0. Todos os valores digitados, exceto o zero, devem ser gravados em um arquivo em disco, um por linha, com 3 casas decimais. Usar o método* `.write()`

```
arq = open('saida_er_11.2.txt', 'w')        # abre o arquivo para gravação
A = float(input('Digite um real: '))
while A != 0:
    arq.write(f'{A:.3f}\n')                 # grava uma linha do arquivo
    A = float(input('Digite um real: '))
arq.close()                                 # fecha o arquivo
print('Fim do Programa')
```

```
Digite um real: 7.32
Digite um real: 18.6
Digite um real: 0.414
Digite um real: 16.379
Digite um real: 0
Fim do Programa
```

Esta solução, muito parecida com a anterior tem a única diferença no formato dos números que são gravados no arquivo de saída. A imagem ao lado mostra o resultado.



Agora reproduza este programa usando o PyCharm e verifique o arquivo gravado.

No Exercício resolvido 11.3 nós resolvemos novamente o exercício 11.2 porém agora usando outro método da classe de arquivos: o método `.writelines()`.

Exercício Resolvido 11.3

Enunciado: *Escreva um programa que permaneça em laço lendo números reais até que seja digitado 0. Todos os valores digitados, exceto o zero, devem ser gravados em um arquivo em disco, um por linha, com três casas decimais. Usar o método* `.writelines()`

```
Lst = []
A = float(input('Digite um real: '))
while A != 0:
    Lst.append(f'{A:.3f}\n')
    A = float(input('Digite um real: '))
arq = open('saida_er_11.3.txt', 'w')        # abre o arquivo para gravação
arq.writelines(Lst)                         # grava toda a lista em linhas do arquivo
arq.close()                                 # fecha o arquivo
print('Fim do Programa')
```

```
Digite um real: 7.32
Digite um real: 18.6
Digite um real: 0.414
Digite um real: 16.379
Digite um real: 0
Fim do Programa
```

Neste caso os mesmos dados foram digitados e o resultado quanto à gravação do arquivo é exatamente o mesmo. Isso é mostrado no vídeo. A diferença está na solução. Nas soluções anteriores o arquivo foi aberto no início do programa e permaneceu aberto durante todo o laço. Nesta nova solução o arquivo é aberto apenas no momento da gravação, sem que seja necessário aguardar o processamento do laço. Em muitos

casos essa estratégia é interessante e desejável, ou seja, os dados são coletados e processados e quando estiverem prontos são gravados todos de uma vez, numa única operação.

Exercício Resolvido 11.4

Enunciado: *Escreva um programa que permaneça que leia um arquivo de entrada, sabendo que esse arquivo tem um número inteiro em cada linha. Todos os números lidos devem ser mostrados na tela. Mostrar também a soma dos valores, a quantidade, a média aritmética, o menor valor e o maior valor.*

Usar um laço while e na leitura usar o método `.readline()`

```
Lst = []
arqEntr = open('entrada_er_11.4.txt', 'r')  # abre o arquivo para leitura
linha = arqEntr.readline()                  # lê a primeira linha
while linha != '':
    Lst.append(int(linha))                  # converte linha para inteiro e coloca na lista
    linha = arqEntr.readline()              # lê a próxima linha
arqEntr.close()                             # fecha o arquivo
print('Valores lidos do arquivo')
print(Lst)
Soma = sum(Lst)     # calcula a soma dos elementos da lista
print(f'Soma dos valores = {Soma}')
Qtde = len(Lst)     # determina a quantidade dos elementos da lista
print(f'Quantidade = {Qtde}')
print(f'Média dos valores = {Soma/Qtde}')
Minimo = min(Lst)   # determina o menor elemento da lista
print(f'Mínimo dos valores = {Minimo}')
Maximo = max(Lst)   # determina o maior elemento da lista
print(f'Máximo dos valores = {Maximo}')
print('Fim do Programa')
```

```
Valores lidos do arquivo
[128, 446, 90, 363, 185, 59, 254, 148, 241, 74]
Soma dos valores = 1988
Quantidade = 10
Média dos valores = 198.8
Mínimo dos valores = 59
Máximo dos valores = 446
Fim do Programa
```

Dados do arquivo
128
446
90
363
185
59
254
148
241
74

Nesta solução adotamos uma lógica mais clássica e própria de outras linguagens de programação. Então esta solução não é muito Pythônica. O objetivo aqui é didático, para que você possa enxergar como Python faz as coisas.

Na leitura usamos o método `.readline()`, que retornará o conteúdo da linha. Na primeira linha está o valor 128, mas o `.readline()` o retorna como um texto incluindo o caractere fim de linha, ou seja, o objeto linha estará carregado com `'128\n'`. Mas isso não é motivo de preocupação pois quando fazemos a conversão para inteiro o caractere `'\n'` é eliminado. Cada valor lido é colocado na lista `Lst` para que na segunda parte possamos usar as funções de soma, tamanho, mínimo e máximo.

Quando acaba a leitura do arquivo o método `.readline()` retorna um string vazio e foi isso que usamos para controlar o laço de leitura.

Exercício Resolvido 11.5

Enunciado: *Escreva um programa que permaneça que leia um arquivo de entrada, sabendo que esse arquivo tem um número inteiro em cada linha. Todos os números lidos devem ser mostrados na tela. Mostrar também a soma dos valores, a quantidade, a média aritmética, o menor valor e o maior valor. Usar aqui o mesmo arquivo de entrada do exercício anterior.*

Usar um iterador for e o arquivo como iterável.

```
Lst = []
for linha in open('entrada_er_11.4.txt'):    # abre o arquivo e o usa como iterável
    Lst.append(int(linha))          # converte linha para inteiro e coloca na lista
print('Valores lidos do arquivo')
print(Lst)
Soma = sum(Lst)     # calcula a soma dos elementos da lista
print(f'Soma dos valores = {Soma}')
Qtde = len(Lst)     # determina a quantidade dos elementos da lista
print(f'Quantidade = {Qtde}')
print(f'Média dos valores = {Soma/Qtde}')
Minimo = min(Lst)   # determina o menor elemento da lista
print(f'Mínimo dos valores = {Minimo}')
Maximo = max(Lst)   # determina o maior elemento da lista
print(f'Máximo dos valores = {Maximo}')
print('Fim do Programa')
```

```
Valores lidos do arquivo
[128, 446, 90, 363, 185, 59, 254, 148, 241, 74]
Soma dos valores = 1988
Quantidade = 10
Média dos valores = 198.8
Mínimo dos valores = 59
Máximo dos valores = 446
Fim do Programa
```

Esta solução usa o recurso de iteração com comando `for` em substituição ao `while` da solução 11.4. A construção do comando for envolvendo arquivo implica na colocação da função `open()` como iterável do `for`. Nessa construção o objeto `linha` é o objeto de controle e ele receberá uma linha do arquivo por vez possibilitando que o programador faça qualquer processamento necessário com os dados desse string.

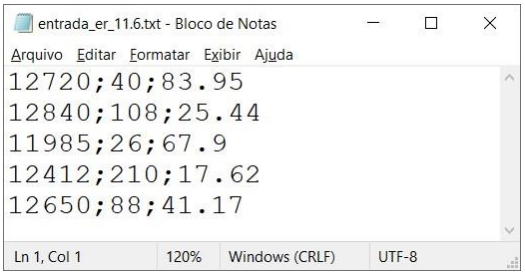
```
for linha in open('entrada_er_11.4.txt'):
    Lst.append(int(linha))
```

Ao fazer isso não precisamos nos preocupar em fechar o arquivo, pois o iterador fará isso automaticamente.

Usar esta forma é mais veloz do que o laço `while` no exercício anterior, pois o comando `for` é otimizado para estas situações. Essa forma é, sem dúvida, muito mais Pythônica que a anterior.

Figura 11.1 – Dados para o exercício resolvido 11.6

Código	Qtde. Estq	Pç. Unitário
12720	40	83.95
12840	108	25.44
11985	26	67.9
12412	210	17.62
12650	88	41.17



Exercício Resolvido 11.6

Enunciado: *Escreva um programa que leia um arquivo de entrada carregando seus dados em um dicionário e ao final exibindo-os na tela. A figura 11.1 mostra o do lado esquerdo a natureza dos dados que serão lidos e do lado direito mostra o formato do arquivo.*

Esse formato é conhecido como CSV. Arquivos CSV são muito usados em diversas áreas da computação, em especial em Análise de Dados. O que caracteriza um arquivo CSV é que cada linha tem um conjunto de dados de alguma forma relacionados e separados por um caractere delimitador. No arquivo deste exercício o delimitador é um ponto-e-vírgula ";"

Neste caso, cada linha tem: código de produto (`int`), a quantidade em estoque (`int`), preço (`float`). Use o código como chave para o dicionário e valor deve ser em formato de tupla.

```
Estoque = {}
for linha in open('entrada_er_11.6.txt'):
    lst = linha.rstrip().split(';')
    cod = int(lst[0])
    qtde = int(lst[1])
    pcunit = float(lst[2])
    Estoque[cod] = (qtde, pcunit)
print('Valores carregados no dicionário')
print(Estoque)
print('\nExibição dos dados na forma de tabela')
TotGeral = 0
for cod, dados in Estoque.items():
    tot = dados[0] * dados[1]
    TotGeral += tot
    print(f' {cod}: {dados[0]:5d} x {dados[1]:6.2f} = {tot:8.2f}')
print(' ' * 24, f'{TotGeral:8.2f}')
print('\nFim do Programa')
```

```
Valores carregados no dicionário
{12720: (40, 83.95), 12840: (108, 25.44), 11985: (26, 67.9), 12412: (210, 17.62), 12650: (88, 41.17)}

Exibição dos dados na forma de tabela
 12720:    40 x  83.95 =  3358.00
 12840:   108 x  25.44 =  2747.52
 11985:    26 x  67.90 =  1765.40
 12412:   210 x  17.62 =  3700.20
 12650:    88 x  41.17 =  3622.96
                        15194.08

Fim do Programa
```

Aqui juntamos técnicas de leitura de arquivos com técnicas de manipulação de dicionários. O que está feito aqui é similar ao que foi feito nos exercícios resolvidos 10.3 e 10.5, nos quais usamos tuplas como valor para o elemento do dicionário. Neste exercício a diferença fundamental é a origem dos dados, que foram lidos do arquivo e não digitados no teclado.

Uma linha do código acima merece destaque:

```
    lst = linha.rstrip().split(';')
```

Perceba que neste código estamos usando dois métodos da classe string concatenados: `.rstrip()` e `.split()`. O `.rstrip()` é necessário para eliminar o caractere '\n' do final, porque ao ser lido do arquivo esse caractere virá junto. Se o exibirmos na tela logo após a leitura ele terá a aparência a seguir.

```
linha = '12720;40;83.95\n'
```

Na sequência o `.split()` é usado para fazer a separação das partes, tendo o ';' como delimitador.

Exercício Resolvido 11.7

Enunciado: *Neste exercício vamos verificar como funciona a codificação dos arquivos texto em Python.*

Escreva um programa que grave as duas linhas de texto abaixo em um arquivo. Em seguida leia esse arquivo e mostre na tela o que foi lido. As codificações que vamos testar são ANSI e UTF-8 e elas deverão ser lidas do teclado.

Texto a ser gravado no arquivo

```
Gravação de Arquivo
acentos: á, é, í, Á, É, Í, ç, Ç
```

```
codGravacao = input('Digite a codificação de Gravação: ')
codLeitura = input('Digite a codificação de Leitura: ')

print('Etapa de gravação do arquivo')
arq = open('teste.txt', 'w', encoding=codGravacao)
arq.write('Gravação de Arquivo\n')
arq.write('acentos: á, é, í, Á, É, Í, ç, Ç\n')
arq.close()

print('\nEtapa de leitura do arquivo')
arq = open('teste.txt', 'r', encoding=codLeitura)
s = arq.readline()
print(s.rstrip())
s = arq.readline()
print(s.rstrip())
arq.close()

print('Fim do Programa')
```

```
primeira execução - grava como ANSI e lê como ANSI
Digite a codificação de Gravação: ANSI
Digite a codificação de Leitura: ANSI
Etapa de gravação do arquivo

Etapa de leitura do arquivo
Gravação de Arquivo
acentos: á, é, í, Á, É, Í, ç, Ç
Fim do Programa

segunda execução - grava como UTF-8 e lê como UTF-8
Digite a codificação de Gravação: UTF-8
Digite a codificação de Leitura: UTF-8
Etapa de gravação do arquivo

Etapa de leitura do arquivo
Gravação de Arquivo
acentos: á, é, í, Á, É, Í, ç, Ç
Fim do Programa

terceira execução - grava como UTF-8 e lê como ANSI
Digite a codificação de Gravação: UTF-8
Digite a codificação de Leitura: ANSI
Etapa de gravação do arquivo

Etapa de leitura do arquivo
GravaÃ§Ã£o de Arquivo
acentos: Ã¡, Ã©, Ã-, Ã□, Ã‰, Ã□, Ã§, Ã‡
Fim do Programa
```

```
quarta execução - grava como ANSI e lê como UTF-8
Digite a codificação de Gravação: ANSI
Digite a codificação de Leitura: UTF-8
Etapa de gravação do arquivo

Etapa de leitura do arquivo
Traceback (most recent call last):
  File "resolvido_9.7.py", line 12, in <module>
    s = arq.readline()
        ^^^^^^^^^^^^^^
  File "<frozen codecs>", line 322, in decode
UnicodeDecodeError: 'utf-8' codec can't decode byte 0xe7 in position 5: invalid
continuation byte
```

Este programa lê dois strings que serão usados com o parâmetro `encoding` do comando `open`. Assim temos a liberdade de testar as 4 combinações entre ANSI e UTF-8 para leitura e gravação.

Perceba que nas duas primeiras execuções correu tudo bem, pois nelas o arquivo é lido e gravado com a mesma codificação. Os problemas surgem quando misturamos codificações.

Na terceira execução gravamos com UTF-8 e lemos com ANSI e os caracteres acentuados ficaram todos errados. Na quarta execução gravamos com ANSI e lemos com UTF-8 e aí a situação ficou pior porque deu erro e interrompeu o programa.

Assim, com este exercício constatamos que usar a codificação correta é mandatório.

Exercício Proposto 11.1

Enunciado: *Reescreva o programa exercício resolvido 11.6 usando um dicionário aninhado no lugar da tupla como valor para o dicionário Estoque.*

Exercício Proposto 11.2

Enunciado: *Escreva um programa que leia um arquivo CSV de entrada que tenha dois inteiros em cada linha. O primeiro é um código de produto e o segundo é a quantidade vendida. O programa deve totalizar quantos itens foram vendidos para cada produto.*

Dica: use um dicionário tendo o código como chave e a quantidade como valor. Para cada código lido do arquivo verifique se ele já existe no dicionário usando o operador in. Se não existir, inclua; se existir some a quantidade existente com a nova quantidade lida do arquivo.

Dados p/ ex.prop 11.2
120;4
140;1
190;6
120;12
150;8
120;5
140;6
190;1
120;4
150;8
140;2
190;10
120;3
150;6
140;5
140;1
190;2
120;4
150;1

Exercício Proposto 11.3

Enunciado: *Escreva um programa que leia um número inteiro N (10 < N < 10.000) e grave um arquivo com N linhas com os dados listados na tabela abaixo. O arquivo deve ter o nome 'Estoque.csv e deve usar o caractere ';' (ponto e vírgula) como delimitador. Não é necessário que o arquivo esteja ordenado.*

Campo	*Descrição*
Código do produto	*Número inteiro entre 10000 e 50000. Gerar aleatórios. Não pode haver repetição deste código e pede-se que não sejam sequenciais.*
Quantidade em estoque	*Número inteiro entre 1 e 3800. Gerar aleatórios.*
Preço unitário compra	*Número real entre 1.80 e 435.90. Gerar aleatórios.*
Alíquota do ICMS	*Alíquota do imposto ICMS. Essa alíquota deve ser 7%, 12% ou 18%. (Não colocar o caractere '%' no arquivo). Gerar aleatórios.*

Dicas/sugestões (você pode segui-las ou não – é sua escolha)

- Para garantir que Código do Produto não tenha repetidos use um dicionário na geração dos dados.

- Gere os dados e coloque no dicionário. Depois grave no arquivo de saída.

- O valor dos elementos do dicionário pode ser no formato de tupla ou de dicionário aninhado.

Exercício Proposto 11.4

Enunciado: *Escreva um programa que grave o arquivo NUMEROS.TXT com 2.000 números, um em cada linha, gerados com a função randint() do módulo random no intervalo [1, 5.000].*

Variação: *Altere este programa substituindo o tamanho fixo de 2.000 por uma quantidade de entrada a ser lida do teclado.*

Exercício Proposto 11.5

Enunciado: *Escreva um programa que leia o arquivo NUMEROS.TXT gerado no exercício proposto 11.4, colocando-os em uma lista. Ordene a lista usando o .sort() e grave os números ordenados no arquivo ORDENADOS.TXT.*

Exercício Proposto 11.6

Enunciado: *Escreva um programa que leia um arquivo de entrada contendo números inteiros, sendo um por linha, e os coloque em uma lista. Em seguida pense em alguma forma de remover os valores repetidos, deixando apenas uma cópia de cada valor.*

A lista resultante após a eliminação dos repetidos, deve ser ordenada e salva no arquivo UNICOS.TXT, um inteiro por linha.

Exercício Proposto 11.7

Enunciado: *Escreva um programa para registrar os seguintes dados de uma frota de veículos de uma empresa:*

Placa (string – chave – obrigatório todas as letras maiúsculas)

Marca

Modelo

Tipo (caminhão, furgão, automóvel, motocicleta, etc)

Kilometragem

Data da Compra (string no formato AAAAMMDD – ano,mês,dia)

O programa deve ficar em laço enquanto a Placa for digitada. O laço termina quando for digitado FIM para a placa. Se for digitada uma placa com letras minúsculas o programa deve convertê-las para maiúsculas com o método .upper().

Para cada veículo leia todos os dados e carregue em um dicionário. Se uma placa já existente for digitada o programa deve avisar que já existe, mostrar seus dados e se usuário quiser fazer alteração em algum dado basta digitar o novo valor. Para os campos em que nada for digitado deve ser mantido o valor já cadastrado.

Ao final do laço grave todos os dados em um arquivo CSV usando o caractere ";" como delimitador.

Detalhe: *Este exercício é uma extensão do exercício proposto 10.6, acrescentando a parte referente à gravação do arquivo*

Exercício Proposto 11.8

Enunciado: *Escreva um programa para ler o arquivo CSV gerado no exercício proposto 11.7 com os dados de uma frota de veículos de uma empresa:*

Placa (string - chave)

Marca

Modelo

Tipo (caminhão, furgão, automóvel, motocicleta, etc)

Kilometragem

Data da Compra (string no formato AAAAMMDD – ano,mês,dia)

O programa deve ler o arquivo, carregar um dicionário e exibir os dados na tela com um layout legível.

Capítulo 12
FUNÇÕES

12.1 CONCEITO DE SUBPROGRAMA

Em programação de computadores, subprogramas têm um papel preponderante no desenvolvimento de softwares. Um subprograma é um trecho de código ao qual atribuímos um nome e que poderá ser usado adiante em um programa maior e mais complexo. O subprograma efetuará parte da tarefa que programa maior precisa realizar.

Atualmente o termo mais usual para subprogramas é "função". Neste contexto o termo "função" tem um significado totalmente diferente daquele empregado em outras áreas do conhecimento, como matemática, biologia ou química.

Assim, em programação, uma função é um conjunto de comandos, ou bloco de código, ao qual se atribui um nome identificador e que executa certa tarefa, podendo receber parâmetros de entrada bem como produzir e retornar algum resultado.

Um programa pode conter tantas funções quanto se queira, bastando ao programador desenvolvê-las conforme julgue adequado. E cada função pode ser utilizada no programa tantas vezes quanto o necessário.

Além disso, as funções podem ser desenvolvidas, testadas e agrupadas em módulos e pacotes de modo a ficar disponíveis para uso futuro em diferentes programas.

12.1.1 DIVIDIR PARA CONQUISTAR

Esta expressão - "dividir para conquistar" - ilustra um conceito importante relativo ao uso de funções. A ideia é dividir um problema complexo, em partes menores e mais simples de serem implementadas. Após

a implementação das partes deve-se juntá-las de modo conveniente construindo-se, por fim, a solução completa do problema.

Esta ideia ganhou força nos anos 1980, tem sido utilizada desde então e ao longo deste tempo se mostrou comprovadamente eficaz.

Para o iniciante em programação pode parecer que o uso de funções é um complicador desnecessário na elaboração de um programa. Isso ocorre porque o iniciante trabalha com pequenos programas e poucas linhas de código de cada vez. No entanto, em termos de quantidade código, os softwares profissionais são muito diferentes dos pequenos programas iniciais que um estudante escreve. E, de fato, não existe a menor chance de não se usar funções no mundo profissional.

12.1.2 Reutilização de código

Outro fator importante do uso de funções é a possibilidade de reutilizar código que já esteja pronto e devidamente testado. Suponha que você precisa incorporar em seu programa uma tarefa complexa como, por exemplo, acesso a um banco de dados, conversão de formatos de imagens ou qualquer outra que você possa pensar. Em uma situação assim o que é melhor: criar tais funcionalidades do zero ou usar funções que estejam prontas e disponíveis na forma de bibliotecas?

Bem, me parece que se o objetivo for produtividade a segunda alternativa é melhor, certo?

É disso que se trata a reutilização de código. Aproveitar aquilo que já existe, não importando se foi feito por você, pelo seu colega de empresa ou por terceiros do outro lado do mundo.

Nesse sentido, o ecossistema da linguagem Python é maravilhoso. A quantidade de bibliotecas disponíveis, testadas, usadas em milhares de aplicações e em constante expansão faz dessa linguagem uma das mais atraentes hoje em dia.

12.1.3 Manutenção do código

O uso de funções permite, pelo menos em teoria, que manutenções no código sejam feitas de forma mais rápida, segura e eficaz. A ressalva incluída – "pelo menos em teoria" – é necessária porque nem sempre essa afirmação é verdadeira, nos casos em que as funções não foram elaboradas de modo adequado, não parametrizadas corretamente e havendo muitas interdependências entre elas e o programa principal.

Caso haja parametrização correta e tais interdependências não existam, então a afirmação é válida.

Manutenções de código são necessárias sempre que algum requisito é alterado. Por exemplo, suponha que o governo altere as regras de cálculo de impostos aplicáveis às operações de compra e venda de produtos. Nessa situação, se um software aplicado ao comércio for bem escrito, basta alterar as funções responsáveis pelo cálculo dos impostos, sem precisar mexer nos outros aspectos. Fazendo essa alteração nas funções específicas, testando-as e constatando que estão corretas, então uma nova versão do software pode ser liberada já com as modificações contempladas.

A ideia então é que para ajustar um software a um novo requisito você só precisa alterar as funções relacionadas a esse requisito.

12.2 Funções em Python – definição e uso

Agora vamos ver como definir e usar uma função na linguagem Python.

A criação de uma função requer o uso da palavra reservada `def` desta forma:

```
def <nomefuncao>():
    <código>
```

Em seguida à `def` coloca-se o nome da função, seguido de parênteses `()` e por fim o caractere dois pontos ":". As regras para nomes da funções são as mesmas dos nomes de objetos, ou seja, deve conter apenas letras, algarismos e underline e não pode começar com algarismo. Letras maiúsculas e minúsculas são interpretadas como caracteres diferentes.

Observe este exemplo onde é definida a função `ExibeHifens()`, dentro da qual há o código necessário para exibição de um string com cinco hífens.

Exemplo 12.1

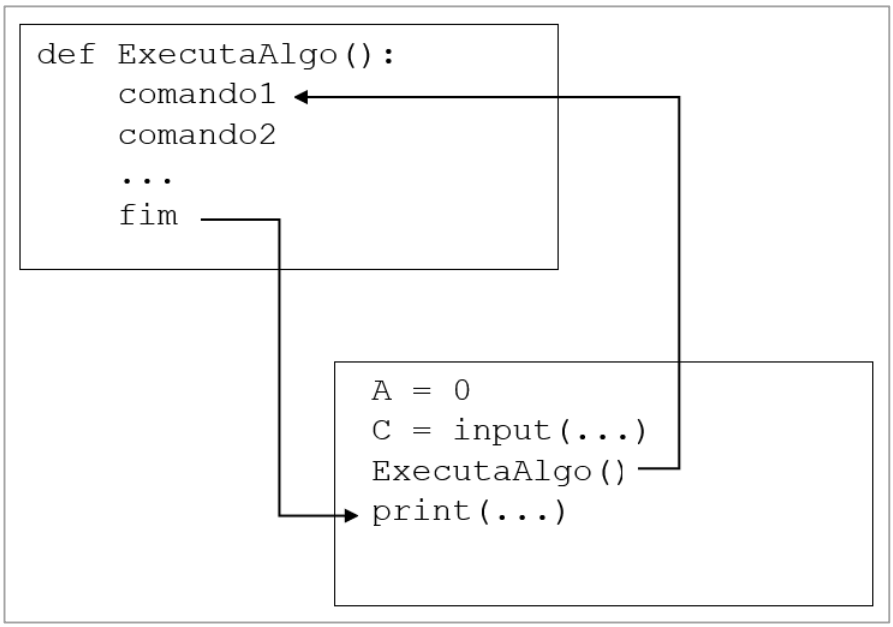
```
def ExibeHifens():
    print('-----')

Lista = [10, 20, 30, 40]
for item in Lista:
    print(item)
    ExibeHifens()
print('Fim do Programa')
```

```
10
-----
20
-----
30
-----
40
-----
Fim do Programa
```

Uma vez definida a função note que ela foi usada na parte principal do programa, dentro do iterador `for`. Assim cada valor contido na lista foi exibido, seguido de uma chamada para a execução da função.

Chamadas a funções alteram o fluxo de execução de um programa, conforme ilustrado na figura 12.1. Sempre que uma chamada ocorre, o programa é desviado para o código interno à função; esse código é executado e ao seu término ocorre o retorno ao comando imediatamente posterior à chamada dessa função que está terminando.

Figura 12.1 – Chamada de função



fonte: o Autor

Funções como a do exemplo 12.1 são muito limitadas, pois não recebem parâmetros de entrada, nem produzem retorno de saída. Vamos agora entender o que são esses importantes elementos.

12.2.1 RETORNO DA FUNÇÃO

Uma função pode ou não retornar algum dado. Em Python é possível que uma função retorne objetos de qualquer classe existente na linguagem. Para gerar um retorno é preciso usar o comando `return` dentro da função. Uma vez que a função possua retorno é possível atribui-lo a um objeto como mostrado nos exemplos a seguir.

Exemplo 12.2

```
def LerInteiro():
    n = int(input('Digite um inteiro: '))
    return n

valor = LerInteiro()
print(f'Valor lido com uso da função = {valor}')
```

```
Digite um inteiro: 512
Valor lido com uso da função = 512
```

Neste programa a função `LerInteiro()` é responsável por realizar a leitura de um inteiro, carregando o objeto `n`. Em seguida, usando o comando `return` o objeto `n` é retornado. Na parte principal do programa o objeto `valor` recebe tal retorno, passando a existir e conter o conteúdo lido dentro da função.

Exemplo 12.3

```
from random import randint
def CarregaLista():
    L = []
    for i in range(10):
        L.append(randint(1, 1000))
    return L
```

```
valores = CarregaLista()
print(f'Lista gerada >> {valores}')
```
```
Lista gerada >> [458, 808, 742, 142, 263, 360, 504, 935, 498, 506]
```

Neste segundo caso a função CarregaLista gera e retorna uma lista contendo 10 números inteiros. A técnica usada é exatamente a mesma nos dois exemplos, demonstrando que uma função pode retornar um único inteiro, ou uma lista com vários elementos, ou ainda objetos de quaisquer classes.

Ainda, o retorno pode ser uma tupla com muitos e variados elementos. Essencialmente, não há qualquer limite para a quantidade de objetos que possam ser retornados por uma função.

Sobre isso, leia com atenção o exemplo 12.4 e faça seus testes, reproduzindo-o no PyCharm. Nele, dentro da função são gerados 7 objetos de diferentes classes e retornados como uma tupla. Na parte principal do programa essa tupla é usada em um `for` iterador exibindo na tela o conteúdo e a classe do objeto.

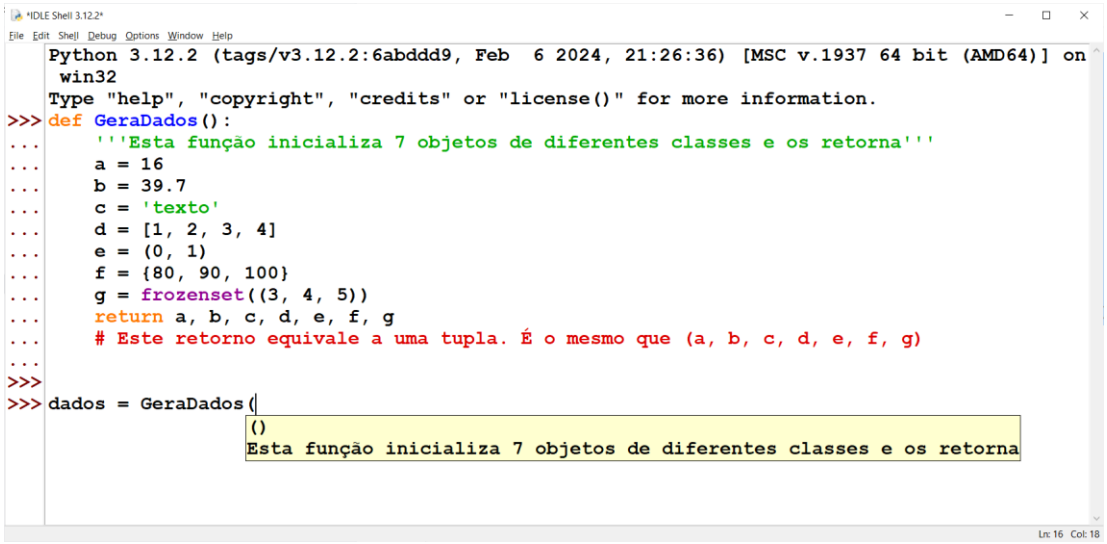
Exemplo 12.4 — Teste este código no PyCharm

```
def GeraDados():
    """Esta função inicializa 7 objetos de diferentes classes e os retorna"""
    a = 16
    b = 39.7
    c = 'texto'
    d = [1, 2, 3, 4]
    e = (0, 1)
    f = {80, 90, 100}
    g = frozenset((3, 4, 5))
    return a, b, c, d, e, f, g
    # Este retorno equivale a uma tupla. É o mesmo que (a, b, c, d, e, f, g)

Dados = GeraDados()
print('Relação de elementos gerados na função')
for x in Dados:
    print(f'  {x} é objeto da classe {type(x)}')
```
```
Relação de elementos gerados na função
  16 é objeto da classe <class 'int'>
  39.7 é objeto da classe <class 'float'>
  texto é objeto da classe <class 'str'>
  [1, 2, 3, 4] é objeto da classe <class 'list'>
  (0, 1) é objeto da classe <class 'tuple'>
  {80, 90, 100} é objeto da classe <class 'set'>
  frozenset({3, 4, 5}) é objeto da classe <class 'frozenset'>
```

12.2.2 DOCUMENTAÇÃO DE FUNÇÃO

Observe ainda que neste exemplo foi incluído no início da função um comentário do tipo *docstring* (aquele que é definido com três aspas de abertura e outras três de fechamento). Isso é um recurso importante e é a forma padrão de documentação das funções em Python. Veja a figura 12.2 a seguir.

Figura 12.2 – Documentação de função



Esta figura foi obtida com o uso do ambiente Idle. O comentário *docstring* presente no início da função é replicado na caixa amarela mostrando que esse comentário é usado como documentação da função – daí advém o nome *docstring* – e aproveitado pelo Python para instruir o programador sobre o uso da função.

12.2.3 PARÂMETROS (OU ARGUMENTOS) DE ENTRADA

Parâmetros ou argumentos de entrada alimentam a função com dados necessários ao trabalho que ela pode realizar. É comum que esses dois termos sejam usados como sinônimos neste contexto.

Os parâmetros de uma função Python podem ser classificados em quatro categorias que veremos:

- obrigatórios – sempre precisam ser fornecidos;
- opcionais – podem ser omitidos pois possuem um valor-padrão atribuído;
- nomeados – devem ser especificados através de seu nome;
- arbitrários – aceita qualquer quantidade de objetos, através do recurso de empacotamento;

Volte ao exemplo 12.3 e observe com atenção que a lista gerada naquele exemplo está limitada a 10 elementos e os elementos são limitados à faixa de valores [1, 1000]. Esses limites fazem com que esta função tenha vícios de origem e isso não é uma forma apropriada de construção de funções.

A solução para essas limitações é a parametrização da função. Isso consiste em fornecer dados de entrada que possam ser usados no seu código interno. O próximo exemplo é a mesma solução do exemplo 12.3 com algumas modificações e mostra como fazer parametrizar a função.

Exemplo 12.5

```
from random import randint
def CarregaLista(qtde):
    L = []
    for i in range(qtde):
        L.append(randint(1, 1000))
    return L

q = int(input('Digite a quantidade desejada: '))
valores = CarregaLista(q)
print(f'Lista gerada >> {valores}')
print(f'A lista gerada tem {len(valores)} elementos')
```

```
# primeira execução
Digite a quantidade desejada: 6
```

```
Lista gerada >> [359, 837, 121, 329, 843, 385]
A lista gerada tem 6 elementos

# segunda execução
Digite a quantidade desejada: 12
Lista gerada >> [440, 995, 57, 521, 92, 46, 205, 356, 293, 542, 832, 22]
A lista gerada tem 12 elementos
```

Nesta versão a função `CarregaLista()` recebe o parâmetro `qtde` que é usado no iterador `for` para produzir uma lista com essa quantidade de elementos. Na parte principal do programa é lido um número inteiro no objeto `q` e esse valor é passado ao parâmetro `qtde`.

Ampliando ainda mais a parametrização produzimos o programa a seguir acrescentando os limites `a` e `b` para geração dos números aleatórios.

Exemplo 12.6

```
from random import randint
def CarregaLista(qtde, a, b):
    L = []
    for i in range(qtde):
        L.append(randint(a, b))
    return L

q = int(input('Digite a quantidade desejada: '))
lmin = int(input('Digite o limite mínimo: '))
lmax = int(input('Digite o limite máximo: '))
valores = CarregaLista(q, lmin, lmax)
print(f'Lista gerada >> {valores}')
print(f'A lista gerada tem {len(valores)} elementos')
```

```
# primeira execução
Digite a quantidade desejada: 8
Digite o limite mínimo: 12000
Digite o limite máximo: 15000
Lista gerada >> [13993, 14619, 14080, 12251, 14981, 13603, 13447, 13391]
A lista gerada tem 8 elementos

# segunda execução
Digite a quantidade desejada: 6
Digite o limite mínimo: -200
Digite o limite máximo: 0
Lista gerada >> [-122, -120, -141, -194, -159, -183]
A lista gerada tem 6 elementos
```

Elaborar funções devidamente parametrizadas é tarefa de todo bom programador, no entanto, concebê-las de maneira adequada exige treino e experiência. Então a recomendação é: pratique bastante.

No exercício resolvido 12.1 vamos desenvolver uma função que calcula e retorna as quatro operações entre dois números reais.

Exercício Resolvido 12.1 — Teste este código no PyCharm

Enunciado: *Escreva um programa que leia dois números reais e calcule as 4 operações aritméticas entre eles usando uma função. Exiba o resultado com duas casas decimais.*

```
def Operacoes(a, b):
    soma = a + b
    dife = a - b
    mult = a * b
    divi = a / b
```

```
    return soma, dife, mult, divi

v1 = float(input('Digite o valor 1: '))
v2 = float(input('Digite o valor 2: '))
resultados = Operacoes(v1, v2)
print('Resultados')
print(f'  soma = {resultados[0]:.2f}')
print(f'  diferença = {resultados[1]:.2f}')
print(f'  multiplicação = {resultados[2]:.2f}')
print(f'  divisão = {resultados[3]:.2f}')
```

```
Digite o valor 1: 12.6
Digite o valor 2: 4.4
Resultados
  soma = 17.00
  diferença = 8.20
  multiplicação = 55.44
  divisão = 2.86
```

12.2.4 PARÂMETRO COM VALOR PADRÃO

Os parâmetros de uma função podem possuir valores padrão – também conhecidos como *default*. Quando um parâmetro tem valor padrão, ele se torna opcional na chamada da função. Se for omitido o padrão é utilizado.

Exemplo 12.7

```
def Saudacao(nome, mensagem = 'Olá'):
    print(mensagem, nome)

Saudacao('José', 'Bom dia')
Bom dia José
Saudacao('José')
Olá José
```

```
(exemplo interativo feito com IDE Idle)
```

Neste exemplo o parâmetro mensagem tem o valor padrão 'Olá'. Note que na primeira execução foi usado o string 'Bom dia' porque foi explicitamente passado. Na segunda execução, com a omissão da mensagem, o valor padrão 'Olá' foi usado.

Em Python existe uma regra sobre parâmetros sem e com valor padrão: sempre, primeiro devem ser relacionados todos os parâmetros que não têm valor padrão e depois aqueles que têm. O interpretador Python não aceita que um parâmetro sem valor padrão seja declarado após outro que o contém.

12.2.5 PARÂMETROS NOMEADOS

Outro recurso importante de Python são os parâmetros nomeados.

Até o momento trabalhamos com parâmetros posicionais. Considere o trecho de código abaixo, onde a função `Diferenca` recebe os parâmetros `A` e `B`. Na chamada são passados os parâmetros `X` e `Y`. É intuitivo entender que o valor de `X` é passado para `A` e o valor de `Y` é passado para `B`. E é isso que ocorre de fato.

```
def Diferenca(A, B):
    ...
R = Diferenca(X, Y)
```

No entanto, o Python aceita que a ordem seja trocada, desde que se faça referência ao nome do parâmetro que receberá o valor passado. O exemplo 12.8 ilustra a situação:

Exemplo 12.8

```
>>> def Diferenca(A, B):
>>>     return A - B


>>> X = 12
>>> Y = 7
>>> Diferenca(X, Y) # Isso é o que fizemos até o momento – passagem posicional
    5

# outra forma de fazer e a passagem nomeada
>>> Diferenca(B = Y, A = X) # B recebe Y e A recebe X, e B vem antes de A
    5
>>> Diferenca(B = X, A = Y) # B recebe X e A recebe Y
    -5
```

(exemplo interativo feito com IDE Idle)

Essa forma de fazer a passagem de parâmetros pode ser utilizada em qualquer chamada de função. Nestes casos, a ordem não faz qualquer diferença todos os parâmetros sejam nomeados e tenham um valor atribuído.

Porém, muito cuidado deve ser tomado ao misturar parâmetros posicionais e nomeados em uma mesma chamada. Isso é algo possível, mas deve-se respeitar a seguinte regra: na chamada os primeiros parâmetros podem ser posicionais e após o primeiro parâmetro nomeado ser escrito, todos os subsequentes também devem ser nomeados. Veja os casos indicados abaixo.

```
def FazAlgo(M, N, O, P):      # esta função recebe 4 parâmetros
  ...

FazAlgo(3, 6, 9, 12)          # chamada Ok.
                              # Todos os parâmetros são posicionais

FazAlgo(N=6, P=12, M=3, O=9)  # chamada Ok.
                              # Todos os parâmetros são nomeados

FazAlgo(3, 6, P=12, O=9)      # chamada Ok (note que P e O estão invertidos).
# neste caso os dois primeiros são posicionais e os outros dois são nomeados

FazAlgo(3, N=6, 9, 12)        # chamada incorreta. Gera erro pois há parâmetros
                              # posicionais (9 e 12) após um parâmetro nomeado

FazAlgo(3, N=6, P=9, O=12)    # chamada Ok.
```

12.2.6 EMPACOTAMENTO DE PARÂMETROS

Em Python existe uma alternativa adicional para passagem de parâmetros que é denominada "empacotamento". Trata-se de uma opção poderosa e flexível aplicável aos casos em que é preciso escrever uma função sem saber exatamente quantos parâmetros serão passados.

A ideia fundamental do empacotamento é que os parâmetros são encapsulados em uma tupla que é passada para a função. Dentro da função, essa tupla é utilizada da maneira que o programador precise. Veja o exemplo:

Exemplo 12.9

```
>>> def Somatoria(*dados):
```

```
        r = 0
        for i in dados:
            r += i
        return r

>>> Somatoria(3, 6, 9)
    18
>>> Somatoria(5, 5)
    10
>>> Somatoria(1, 1, 1, 1, 1, 1, 1, 1, 1, 1)
    10
```
(exemplo interativo feito com IDE Idle)

A função `Somatoria` possui o parâmetro `dados`, que está qualificado com o caractere `*` indicando ao interpretador é um qualificador que faz parte do nome do parâmetro. Dentro da função `*dados` foi tratado como iterador e uma somatória de seus valores foi produzida.

Argumentos empacotados podem ser misturados com parâmetros normais. No entanto, normalmente, os argumentos empacotados estarão no final da lista de parâmetros da função da seguinte forma:

```
def Funcao(p1, p2, *p)
```

Os parâmetros p1 e p2 são normais e *p é um empacotamento.

Quaisquer parâmetros normais que ocorram após o empacotamento obrigatoriamente devem ser tratados como parâmetros nomeados. Normalmente esses parâmetros pós-empacotamento terão definidos um valor padrão.

```
def Funcao(p1, p2, *p, extra='valor padrão')
```

Isso permite construções interessantes. Veja o exemplo a seguir. Nesse exemplo é definida uma função que recebe o empacotamento `valores` e o parâmetro `separador`. O separador é um string que tem como valor default uma vírgula seguida de um espaço em branco `', '`. Essa função executa a concatenação dos strings recebidos em `valores`, tendo como elemento de ligação o string `separador`.

Exemplo 12.10

```
>>> def MontaSaida(*valores, separador = ', '):
        saida = separador.join(valores)
        return saida

>>> MontaSaida('Maçã', 'Laranja', 'Banana', 'Melão')
    'Maçã, Laranja, Banana, Melão'
>>> MontaSaida('Maçã', 'Laranja', 'Banana', 'Melão', separador = ' -*- ')
    'Maçã -*- Laranja -*- Banana -*- Melão'
```
(exemplo interativo feito com IDE Idle)

Observe que na chamada da função, se o valor do separador for omitido, então o valor padrão é usado. E se for especificado ele é usado.

12.2.7 DESENPACOTAMENTO DE PARÂMETROS

O desempacotamento é a situação inversa ao empacotamento. Nela, os valores já estão em uma lista ou tupla e precisam ser distribuídos a um determinado número de objetos individuais. Neste caso os parâmetros da função serão parâmetros normais e uma lista ou tupla será passada, como mostrado a seguir:

Exemplo 12.11

```
>>> def Calcula(a, b, c):    # recebe 3 parâmetros normais
        return (a + b) / c

>>> L1 = [12, 20, 5]         # tem 3 elementos. Vai funcionar.
>>> Calcula(*L1)
    6.4
>>> L2 = [12, 20, 5, 14]     # tem 4 elementos. Vai ocorrer erro.
>>> Calcula(*L2)
    Traceback (most recent call last):
      File "<pyshell#69>", line 1, in <module>
        Calcula(*L2)
    TypeError: Calcula() takes 3 positional arguments but 4 were given
```

(exemplo interativo feito com IDE Idle)

Há dois requisitos obrigatórios no uso do desempacotamento:

- a chamada da função deve conter um objeto de classe sequência, string, tupla ou lista, precedido pelo caractere *
- A quantidade de elementos presentes na sequência a ser passada deve coincidir com a quantidade de parâmetros que a função recebe. Caso isso não ocorra, será gerada uma mensagem de erro, conforme mostrado acima.

12.3 Escopo de objetos em funções

A partir do conteúdo sobre funções visto até o momento podemos apontar que em um programa escrito em Python existem dois ambientes distintos:

a. Ambiente externo à função – que chamaremos de ambiente global.
b. Ambiente interno à função – que chamaremos de ambiente local.

O escopo diz respeito ao estudo dos objetos presentes nesses ambientes e da maneira como eles são visíveis durante seu tempo de existência no programa.

A definição básica de visibilidade é:

- todos os objetos criados fora de qualquer função são denominados globais e
- todos os objetos criados dentro de uma função são denominadas locais.

Os objetos locais existem apenas enquanto a função está em execução. Quando uma função é chamada, seus objetos internos são criados, passam a existir, ocupando parte da memória do computador, e podem ser utilizados normalmente. Quando a função termina, esses objetos são removidos da memória, deixam de existir e os dados que continham são descartados.

Os valores de retorno da função também deixam de existir, porém, antes de serem descartados são atribuídos aos objetos que os recebem na chamada da função.

Exemplo 12.2 reprodução

```
def LerInteiro():
    n = int(input('Digite um inteiro: '))
    return n

valor = LerInteiro()
print(f'Valor lido com uso da função = {valor}')
```

```
Digite um inteiro: 512
Valor lido com uso da função = 512
```

Assim, vamos retornar ao exemplo 12.2 visto anteriormente e reescrito acima para facilitar o entendimento. Observe-o atentamente. Dentro da função `LerInteiro()` existe o objeto `n`. Este objeto é local e existirá apenas enquanto a função estiver em execução.

Por sua vez, o objeto `valor` é global pois foi definido na parte principal do programa. Após criado ele existirá durante todo o tempo de existência do programa, seja durante o processamento da parte principal ou durante o processamento da função.

Objetos globais podem ser manipulados tanto fora quanto dentro das funções. Objetos locais só podem ser manipulados dentro da função à qual pertencem.

Cuidado com os nomes de objetos

Vamos ver do que se trata essa recomendação sobre o cuidado com os nomes. Veja o próximo exemplo:

Exemplo 12.12 

```
def funcao1():
    print('dentro da função1 temos >>', a, b)

def funcao2():
    a = 100
    b = 200
    print('dentro da função2 temos >>', a, b)

a = 10
b = 20
funcao1()
funcao2()
print('fora da função temos >>', a, b)
```

```
dentro da função1 temos >> 10 20      -> linha de saída 1
dentro da função2 temos >> 100 200    -> linha de saída 2
fora da função temos >> 10 20         -> linha de saída 3
```

Perceba a confusão de nomes que pode ocorrer. Tanto fora, como dentro da função são usados dois objetos com os nomes `a`, `b`. No escopo global, não haverá dúvidas: `a` e `b` são os objetos globais criados fora da função. Porém, dentro das funções quem são `a` e `b`, locais ou globais? A resposta é que, havendo igualdade de nomes de objetos locais e globais, dentro das funções serão referenciados os objetos locais.

Vamos agora compreender as três linhas de saída acima:

1. Essa saída é produzida dentro da `funcao1()` e são exibidos os valores 10 e 20. Isso ocorre porque dentro da `funcao1()` não há declaração de qualquer objeto, então a função usou os objetos `a` e `b` globais;
2. Essa saída é produzida dentro da `funcao2()` e são exibidos os valores 100 e 200. Isso ocorre porque objetos locais `a` e `b` foram criados e carregados com 100 e 200. Nessa saída foram usados os objetos locais, uma vez que eles existem na `funcao2()`;
3. Essa saída é produzida na parte principal do programa. Os valores 10 e 20 exibidos são os que estão dentro dos objetos globais `a` e `b`.

Agora vamos fazer uma pequena alteração no exemplo acima, produzindo o código a seguir.

Exemplo 12.13 — Teste este código no PyCharm

```
def funcao1():
    print('dentro da função1 temos >>', a, b)

def funcao2():
    global a  # nesta linha informamos que a é o objeto global
    a = 100   # nesta linha estamos alterando o valor do a global
    b = 200   # nesta linha estamos criando um objeto local b
    print('dentro da função2 temos >>', a, b)

a = 10
b = 20
funcao1()
funcao2()
print('fora da função temos >>', a, b)
```

```
dentro da função1 temos >> 10 20      -> linha de saída 1
dentro da função2 temos >> 100 200    -> linha de saída 2
fora da função temos >> 100 20        -> linha de saída 3
```

A alteração refere-se ao uso da palavra reservada `global` para declarar dentro da `funcao2()` que o objeto `a` usado na função é o objeto global. Isso faz com que dentro dessa função, caso o valor de `a` seja alterado, essa alteração se refletirá fora da função. Com isso, perceba que há uma diferença na linha de saída 3, que neste caso exibe os valores 100 e 20. O valor original 10 de `a` foi alterado dentro da função para 100.

Sobre escopo de objetos podemos fazer a seguinte síntese:

- Objeto global é visível tanto fora como dentro de funções,
- Objeto global é manipulável dentro da função apenas se for qualificado com a palavra `global`;
- Objeto local é visível e manipulável apenas na função em que foi criado;
- Se houver conflito de nome de objeto dentro da função o objeto local será referenciado;
- Evite conflitos de nomes entre objetos locais e globais.

Cuidado
ao usar os mesmos nomes para objetos globais e locais
O Python permite que você faça isso, mas é sua responsabilidade gerenciar as atribuições de valores de maneira correta

Boas Práticas
Evite usar nomes idênticos para objetos locais e globais

12.4 Anotações em funções Python

As anotações em uma função de Python (*function annotations*) são informações opcionais relativas às classes dos parâmetros e do retorno gerado pela função. No exemplo 12.14 são criadas duas funções que fazem exatamente o mesmo trabalho. A primeira é uma função como temos visto até o agora; a segunda contém anotações, indicando que os parâmetros `a` e `b` são da classe `int` e o retorno da função é `int`.

Exemplo 12.14 — Teste este código no Idle

```
# função sem annotations
def Soma(a, b):
    return a + b

# função com annotations
def SomaAnot(a: int, b: int) -> int:
    return a + b
```

```
(exemplo interativo feito com IDE Idle)
```

As anotações de parâmetro são definidas por dois pontos após o nome do parâmetro, seguido do nome da classe esperada para aquele parâmetro.

A anotação de retorno da função é definida pelo literal `->` (hífen seguido do sinal de maior, simbolizando uma seta), seguido pelo nome da classe que será o retorno gerado pela função.

A anotações não afetam absolutamente nada da execução da função e existem para serem usadas como documentação dos parâmetros e do retorno.

(consulte PEP 3107 e PEP 484 para obter mais informações).

Exercício Resolvido 12.2

Enunciado: *Escreva uma função que recebe um número inteiro como parâmetro de entrada e retorna o texto "PAR" ou "ÍMPAR". Use-a em um programa principal*

```
def Paridade(a):
    if a % 2 == 0:
        return 'PAR'
    else:
        return 'ÍMPAR'

n = int(input('Digite um inteiro: '))
print(Paridade(n))
```

```
# primeira execução
Digite um inteiro: 48
PAR

# segunda execução
Digite um inteiro: 53
ÍMPAR
```

Exercício Proposto 12.1

Enunciado: *No exercício resolvido 12.2 foi usado o comando condicional clássico. Altere o código dentro da função substituindo o* `if-else` *clássico por um* `if` *de única linha.*

Exercício Resolvido 12.3

Enunciado: *Escreva uma função que receba dois números inteiros A e B como parâmetros de entrada e retorne True se A for divisível por B e False caso contrário. Escreva o programa principal para testar a função.*

```
def Divisivel(A, B):
    return True if A % B == 0 else False

valorA = int(input('Digite A: '))
valorB = int(input('Digite B: '))
if Divisivel(valorA, valorB):
    print(f'{valorA} é divisível por {valorB}')
else:
    print(f'{valorA} não é divisível por {valorB}')
```

```
# primeira execução
Digite A: 85
Digite B: 17
85 é divisível por 17

# segunda execução
Digite A: 170
Digite B: 20
170 não é divisível por 20
```

Exercício Resolvido 12.4

Enunciado: *Escreva uma função que receba como parâmetro de entrada um número inteiro de 5 dígitos de [10000, 99999] que represente códigos de produtos vendidos em uma loja. A função deve calcular e retornar o dígito verificador utilizando a regra de cálculo explicada a seguir. Escreva o programa principal para testar a função.*

Regra: *Considere o código 31483, em que cada dígito é multiplicado por um peso começando em 2 e terminando em 6. Os valores obtidos são somados, e do total obtido calcula-se o resto de sua divisão por 7.*

Dígito	3	1	4	8	3	
Peso	2	3	4	5	6	
Multiplicação	6	3	16	40	18	Soma todos = 83
						Resto de 83 por 7 = 6

```
def CalcDigito(cod):
    s = str(cod)     # transforma o código em string para usar como iterador
    peso = 2
    dv = 0
    for a in s:      # processa cada algarismo em s
```

```
        dv += peso * int(a) # converte algarismo para int, multiplica pelo peso e acumula
        peso += 1   # incrementa o peso
    return dv % 7

codigo = int(input('Digite o código: '))
while codigo > 0:
    digito = CalcDigito(codigo)
    print(f'Código: {codigo} -> dígito: {digito}')
    codigo = int(input('Digite o código: '))
print('\nFim do Programa')
```

```
Digite o código: 31483
Código: 31483 -> dígito: 6
Digite o código: 11000
Código: 11000 -> dígito: 5
Digite o código: 12350
Código: 12350 -> dígito: 3
Digite o código: 12500
Código: 12500 -> dígito: 0
Digite o código: 12600
Código: 12600 -> dígito: 4
Digite o código: 12750
Código: 12750 -> dígito: 5
Digite o código: 0

Fim do Programa
```

Exercício Proposto 12.2

Enunciado: *No exercício resolvido 12.4 não foi feita uma validação do código lido do teclado. Sendo assim, experimente digitar códigos que são menores que 10000 ou maiores que 99999 e veja o que acontece.*

Em seguida implemente a validação do código lido e só efetue o cálculo do dígito verificador se ele for válido.

Exercício Resolvido 12.5

Enunciado: *Escreva uma função que receba como parâmetro de entrada dois números reais Min e Max. Essa função deve ler do teclado um número real e retorná-lo caso esteja dentro do intervalo fechado [Min, Max]. Caso contrário, a função deve exibir uma mensagem de erro e ler um novo valor.*

```
def LeReal(pLMin, pLMax):
    a = float(input('Digite um valor real: '))    # lê o valor
    while a < pLMin or a > pLMax:  # permanece em laço enquanto o valor for inválido
        print(f'O valor {a} está fora dos limites [{pLMin}, {pLMax}]')
        a = float(input('Digite um valor real: '))
    return a  # quando o valor for válido retorna

LMin = float(input('Digite o valor mínimo: '))
LMax = float(input('Digite o valor máximo: '))
controle = 's'
while controle == 's' or controle == 'S':
    Valor = LeReal(LMin, LMax)
    print(f'O valor lido é {Valor}')
    controle = input('\nQuer digitar outro valor (s/n)? ')
print('Fim do Programa')
```

```
Digite o valor mínimo: 2500
Digite o valor máximo: 5000
Digite um valor real: 3750
O valor lido é 3750.0
```

```
Quer digitar outro valor (s/n)? s
Digite um valor real: 554.3
O valor 554.3 está fora dos limites [2500.0, 5000.0]
Digite um valor real: 5543.0
O valor 5543.0 está fora dos limites [2500.0, 5000.0]
Digite um valor real: 4320
O valor lido é 4320.0

Quer digitar outro valor (s/n)? n
Fim do Programa
```

Para testar a função deste programa fizemos, na parte principal, um laço `while` que permanece em execução enquanto a escolha do usuário for o caractere 's' (minúsculo ou maiúsculo). Qualquer outro caractere encerra esse laço. Dentro da função um laço garante que o valor digitado seja válido, dentro dos limites fornecidos no início do programa e passados como parâmetro pelo programa principal.

Exercício Resolvido 12.6

Enunciado: *Escreva um programa que leia quatro notas de um aluno, calcule a média e mostre a situação do aluno que será 'APROVADO' ou 'REPROVADO'.*

O programa deve ler as quatro notas separadas por um espaço em branco em uma mesma linha de digitação. As notas lidas devem ser separadas, convertidas para número real e inseridas em uma lista, junto com a média e a situação do aluno. Isso tudo deverá ser feito dentro de uma função. Médias a partir de 7.0 indicam aprovação; menos que isso reprovação.

A ordem das notas na digitação deve ser: P1 P2 P3 NT

Escreva o programa principal para testar sua função. A saída deste programa deve mostrar todas as notas, a média e a situação (você é livre para elaborar o layout de saída).

Cálculo Média *MF = 0.3 * P1 + 0.3 * P2 + 0.3 * P3 + 0.1 * NT*

```
def CalculaMedia(pNotas):
    """Recebe o string pNotas, faz a separação dos valores, calcula e retorna a média"""
    pNotas = pNotas.split() # pNotas recebido também será usado como retorno da função
    for i in range(len(pNotas)):     # converte as notas lidas para float
        pNotas[i] = float(pNotas[i])
    media = pNotas[0] * 0.3 + pNotas[1] * 0.3 + pNotas[2] * 0.3 + pNotas[3] * 0.1
    situacao = 'APROVADO' if media >= 7 else 'REPROVADO'
    pNotas.append(media)             # acrescenta media na lista de retorno
    pNotas.append(situacao)          # acrescenta situacao na lista de retorno
    return pNotas

notas = input('Digite 4 notas (P1 P2 P3 NT): ')
resultado = CalculaMedia(notas) # chama a função que retorna a lista com resultados
print(f'P1 = {resultado[0]:.1f}; ', end='')
print(f'P2 = {resultado[1]:.1f}; ', end='')
print(f'P3 = {resultado[2]:.1f}; ', end='')
print(f'NT = {resultado[3]:.1f} -> ', end='')
print(f'Média = {resultado[4]:.1f} ', end='')
print(f'Situação: {resultado[5]}')
```

```
# primeira execução
Digite 4 notas (P1 P2 P3 NT): 5.0 5.0 5.0 10.0
P1 = 5.0; P2 = 5.0; P3 = 5.0; NT = 10.0 -> Média = 5.5 Situação: REPROVADO

# segunda execução
Digite 4 notas (P1 P2 P3 NT): 7.5 6.2 6.4 9.8
P1 = 7.5; P2 = 6.2; P3 = 6.4; NT = 9.8 -> Média = 7.0 Situação: APROVADO

# terceira execução
Digite 4 notas (P1 P2 P3 NT): 6.8 8.3 7.2 8.5
P1 = 6.8; P2 = 8.3; P3 = 7.2; NT = 8.5 -> Média = 7.5 Situação: APROVADO
```

Exercício Proposto 12.3

Enunciado: *Altere o programa anterior mudando a regra de cálculo da média final. Na nova regra as notas de prova P1, P2 e P3 devem ser analisadas e a menor nota deve ser descartada. As duas melhores notas serão chamadas de N1 e N2. A nota de trabalho será considerada e a nova fórmula é:*

*MF = 0.4 * N1 + 0.4 * N2 + 0.2 * NT*

Casos de teste:

```
(P1 P2 P3 NT): 5.0 5.0 5.0 10.0
N1 = 5.0; N2 = 5.0; NT = 10.0 -> Média = 6.9 Situação: REPROVADO

(P1 P2 P3 NT): 7.5 6.2 6.4 9.8
N1 = 6.4; N2 = 7.5; NT = 9.8 -> Média = 7.5 Situação: APROVADO

(P1 P2 P3 NT): 6.8 8.3 7.2 8.5
N1 = 7.2; N2 = 8.3; NT = 8.5 -> Média = 7.9 Situação: APROVADO
```

Exercício Resolvido 12.7

Enunciado: *Escreva um programa que verifique se um número inteiro lido é primo. Lembrando: um número primo é divisível apenas por 1 e por ele mesmo. A verificação do primo deve ser feita dentro de uma função.*

```
def Primo(V):
    """Se V for primo retorna True, senão retorna False"""
    if V == 2:          # V é 2
        return True
    elif V % 2 == 0:    # V é par maior que 2
        return False
    else:               # testa se V ímpar é primo
        raiz = pow(V, 0.5) # a raiz de V é o limite dos testes necessários
        i = 3
        while i <= raiz:
            if V % i == 0:
                return False # se for divisível retorna falso imediatamente
            i += 2
        return True # se chegar no final do laço então é primo

N = int(input('Digite um inteiro: '))
if Primo(N):
    print(f'{N} é primo')
else:
    print(f'{N} não é primo')
```

```
# primeira execução
Digite um inteiro: 317
317 é primo

# segunda execução
Digite um inteiro: 215
215 não é primo
```

Exercício Proposto 12.4

Enunciado: *Escreva um programa que leia dos inteiros A, B e carregue uma lista com todos os números primos situados no intervalo fechado [A, B]. Use a função* `Primo()` *apresentada no exercício anterior. Ao final exiba a lista na tela.*

Adicionalmente escreva uma função para exibir a lista de primos de uma forma organizada na tela.

Exercício Proposto 12.5

Enunciado: *Crie uma função que receba um ângulo em graus, e retorne esse ângulo convertido para radianos. O valor de PI está disponível no módulo* `math`*. Importe o módulo e use* `math.pi`

Escreva o programa principal para testar a função.

Regra *AngRadiano = AngGraus * PI / 180*

Exercício Proposto 12.6

Enunciado: *Crie uma função que receba um número de 1 a 12 e retorne nome do mês correspondente. Se o valor for outro o retorno da função deve ser o string "Inválido".*

Escreva o programa principal para testar a função.

Exercício Proposto 12.7

Enunciado: *Escreva uma função que carregue e retorne uma lista com todos os elementos da sequência de Fibonacci menores que um parâmetro passado à função.*

Escreva o programa principal para testar a função.

A sequência de Fibonacci é definida da seguinte forma: a) os dois primeiros termos da sequência são 0 e 1. Do terceiro termo em diante cada termo é a soma dos dois anteriores.

Caso de teste: *Se ValorLimite = 120, então a sequência é: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89*

Exercício Proposto 12.8

Enunciado: *Escreva uma função que receba uma lista como parâmetro de entrada e retorne uma tupla contendo quatro valores na seguinte ordem: a soma, a média, o menor e o maior valor dentre todos os elementos nela contidos. Considere que nessa lista ocorram apenas números reais. Escreva um programa para testar essa função, exibindo na tela os resultados.*

Exercício Proposto 12.9

Enunciado: *Escreva uma função que receba uma lista de números inteiros e elimine os eventuais elementos repetidos nela contidos e a retorne.*

Dica: *Dentro da função use a classe* `set`.

Escreva o programa principal para testar a função.

Exercício Proposto 12.10

Enunciado: *Escreva uma função que receba um parâmetro string. Essa função deve retornar True se o string for palíndromo e False caso não seja. Espaços em branco, algarismos e sinais de pontuação devem ser eliminados, se estiverem presentes no parâmetro.*

Definição *Um string palíndromo é aquele que pode ser lido da esquerda para a direita e da direita para a esquerda, por exemplo: IRENE RI -> elimine o espaço em branco IRENERI*

Outros palíndromos para teste

SUBI NO ONIBUS

ANOTARAM A DATA DA MARATONA

A CARA RAJADA DA JARARACA

AABBCCDDCCBBAA *(não é uma frase, mas é palíndromo)*

12.5 Funções Recursivas

O termo Recursividade se refere a uma importante característica aplicável às funções e devido à sua importância é um elemento que existe em todas as linguagens de programação.

12.5.1 O que é Recursividade?

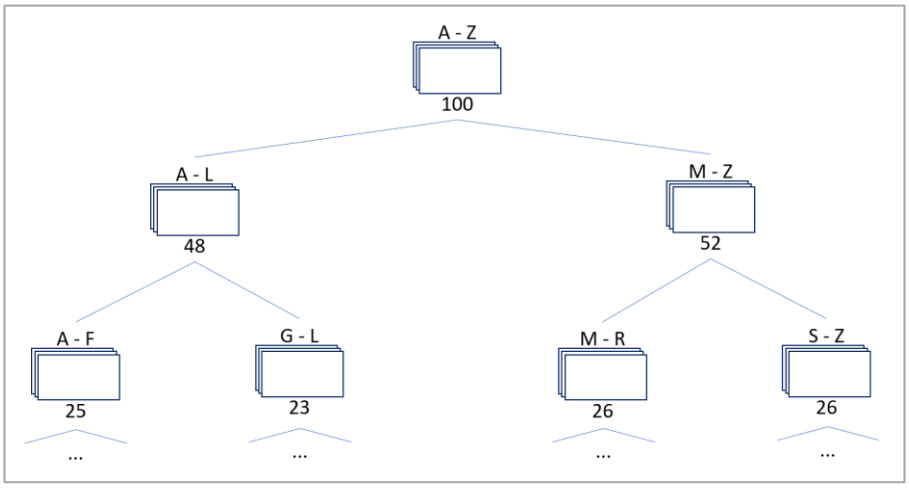
Antes de ver como isso é feito em programas de computador vamos entender a ideia central.

Para isso vamos recorrer a um exemplo do dia a dia. Imagine que você esteja encarregado de coletar fichas de inscrição para algum evento e que essas fichas são físicas, em papel (coisa estranha hoje em dia, mas pode acontecer...). Depois de coletadas todas as fichas, seu supervisor pede que você as coloque em ordem alfabética pelo nome do inscrito. Muito bem, são 100 inscritos e você percebe que vai dar algum trabalho. No entanto, você conhece recursividade, então folheia ficha por ficha e vai separando:

- Nomes iniciando entre A e L para a esquerda → gera uma nova pilha com 48 fichas (suposição)
- nomes iniciando entre M e Z para a direita → gera uma nova pilha com 52 fichas (suposição)

Esse processo ilustrado na figura 12.3 mostra que é possível dividirmos cada pilha em duas menores.

Figura 12.3 – Exemplo de Recursividade



fonte: o Autor

Na primeira passada pelas fichas você gerou dois grupos menores. Em seguida você repete o processo com esses dois grupos, gerando quatro grupos menores ainda. E assim por diante, tanto quanto queira.

Se isso for feito uma terceira vez, você terá 8 grupos de fichas cada um com quantidades talvez entre 10 a 16 fichas e para estes grupos pequenos é bem rápido colocar em ordem alfabética. Depois é só juntar todos os grupos já ordenados e você terá o conjunto completo em ordem alfabética.

Isso é Recursividade. E em programação de computadores pode ser implementada através de funções.

Recursividade Objetivo: solucionar um problema maior dividindo-o em vários problemas menores, resolvendo-os e juntando as partes para produzir a solução final

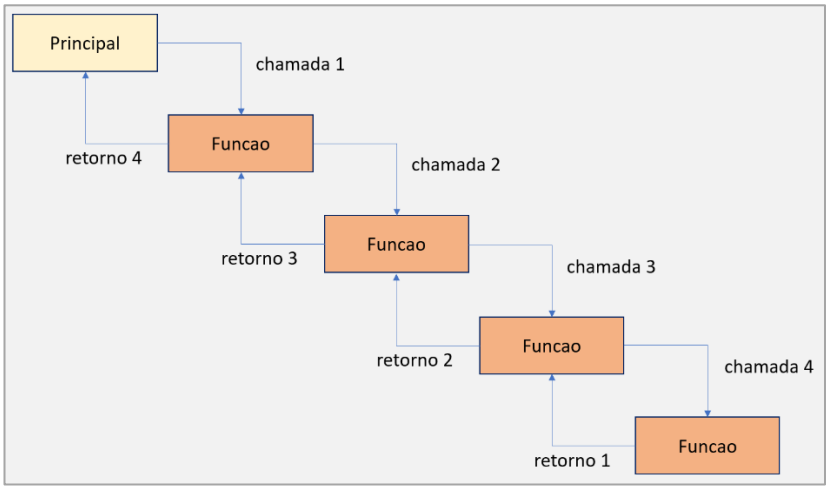
12.5.2 RECURSIVIDADE EM PROGRAMAÇÃO DE COMPUTADORES

Em programação de computadores o que define uma função recursiva é possibilidade dela fazer uma chamada a si mesma. Veja o trecho de código a seguir, que representa uma situação hipotética. Nesse trecho está definida a função cujo nome é `FuncaoRecursiva()`, que recebe dois parâmetros `pA` e `pB`. Dentro dela se uma condição for verdadeira ela retorna algum resultado; se for falsa ela calcula dois novos valores `x` e `y` e em seguida realiza uma nova chamada à `FuncaoRecursiva()`. Nessa segunda chamada o parâmetro `pA` receberá o valor de `x` calculado na primeira chamada e o parâmetro `pB` receberá o valor de `y`.

```
def FuncaoRecursiva(pA, pB):
    if <condição>:
        return algo
    else:
        <comando 1 - define x>
        <comando 2 - define y>
        R = FuncaoRecursiva(x, y)
        return R
```

Esse processo pode se repetir várias vezes, podendo haver terceira, quarta, quinta, etc chamadas, conforme ilustrado na figura 12.4. Quando a `condição` se tornar verdadeira em uma das chamadas, então ela será a última e iniciará um processo de retorno sucessivo. Cada chamada da função ao terminar leva ao retorno para a chamada anterior. Com isso todas as chamadas vão retornando até que ocorra o retorno total de volta para o ponto onde ocorreu a primeira chamada feita.

Figura 12.4 – Ilustração do esquema de chamadas recursivas de uma função



fonte: o Autor

Funções recursivas são capazes de executar tarefas repetitivas sem que uma estrutura de repetição seja usada, como `for` ou `while`. E em muitos casos, um algoritmo recursivo é mais rápido que um laço tradicional pois tem a característica de dividir um problema maior em diversos pequenos subproblemas, como exemplificado no exemplo da classificação das fichas em ordem alfabética.

Um exemplo clássico de operação recursiva computacional é o cálculo do fatorial de um número: N!

Pela definição de fatorial sabemos que N! = N * (N – 1) * (N – 2) * ... * 3 * 2 * 1

Este é um problema que pode ser dividido facilmente da seguinte forma: o fatorial de 5 é a mesma coisa que 5 multiplicado pelo fatorial de 4, ou seja: 5! = 5 * 4! .

Essa divisão pode ser estendida para 4!, 3!, 2! e 1!

Generalizando podemos escrever que: N! = N * (N – 1)! e a partir dessa ideia é possível conceber uma função recursiva que calcule o fatorial de N através de chamadas sucessivas a si mesma. O exemplo 12.15 mostra como ficará essa função

Exemplo 12.15

```
def Fatorial(N):
    if N <= 1:
        return 1
    else:
        return N * Fatorial(N-1) # preste muita atenção a essa linha – Linha 5

Entrada = int(input('Digite um inteiro: '))
F = Fatorial(Entrada)
print(f'O fatorial de {Entrada} é {F}')
```

```
Digite um inteiro: 6
O fatorial de 6 é 720
```

No exemplo o valor 6 foi fornecido como dado de entrada. Sabe-se que o `6! = 720`, portanto, verifica-se que o programa está correto. O ponto agora é explicar o que está sendo feito na função `Fatorial()`.

O programa foi executado com o valor 6 fornecido para o objeto `Entrada`, que é o objeto usado na leitura. Assim, na chamada de `Fatorial()` o valor passado para o parâmetro `N` foi 6. Na Linha 5 da função encontra-se o ponto-chave. Nessa linha o comando `return N * Fatorial(N-1)` provoca uma segunda chamada à própria função, porém, passando como parâmetro o valor `N-1`, ou seja, 5. Ao ser chamada pela terceira vez o parâmetro será 4, e assim por diante, até que ocorra uma chamada com parâmetro `N = 1`. Quando isso ocorrer, a condição do comando `if` na linha 2 avaliará como verdadeiro e a função retornará à chamada anterior, retornando o valor 1. Esse retorno será multiplicado por `N = 2` e retornará para a chamada anterior, e assim sucessivamente, conforme ilustrado no Quadro 12.1.

Quadro 12.1 – Lista de chamadas e retornos da função Fatorial

Chamada nº	Condição na entrada	Ciclo de entrada com N * Fatorial(N-1)	Ciclo de saída das chamadas
0	-	Função principal	Recebe 720 da chamada que fez
1	N = 6	6 * Fatorial (5)	Calcula 6 * 120 = 720 e retorna 720
2	N = 5	5 * Fatorial (4)	Calcula 5 * 24 = 120 e retorna 120
3	N = 4	4 * Fatorial (3)	Calcula 4 * 6 = 24 e retorna 24
4	N = 3	3 * Fatorial (2)	Calcula 3 * 2 = 6 e retorna 6
5	N = 2	2 * Fatorial (1)	Calcula 2 * 1 = 2 e retorna 2
6	N = 1	return 1	Retorna 1 à chamada anterior

As funções recursivas são em sua maioria soluções mais elegantes e simples se comparadas a funções tradicionais ou iterativas. Porém, essa elegância e simplicidade têm um preço que requer muita atenção em sua implementação. Não é incomum que quem está iniciando em programação de computadores sinta alguma dificuldade em elaborar funções recursivas que funcionem corretamente. Assim como tudo, aprender a escrever funções recursivas exige estudo, treino e, especialmente, o desenvolvimento da habilidade de pensar recursivamente. Portanto, treine bastante fazendo exercícios recursivos.

12.6 Recursividade - Síntese

Uma função recursiva é uma função capaz de resolver um problema complexo através da solução de instâncias menores do mesmo problema. Essa técnica é usada com frequência em variadas situações de programação para resolver problemas que podem ser divididos em subproblemas mais simples e semelhantes, resolvendo-os e mesclando as soluções com o problema original.

12.6.1 Diretrizes para escrever uma função recursiva

Para escrever uma função recursiva você precisa estudar o problema e identificar dois elementos: o caso base; e o caso recursivo, que são descritos a seguir.

Caso Base

Toda função recursiva deve ter um caso base. O caso base é o cenário mais simples que não requer recursão adicional. O caso base constitui a necessária condição de encerramento que impede que a função se chame indefinidamente. Sem um caso base adequado, uma função recursiva pode levar a uma recursão infinita. No exemplo 12.15 o caso base ocorre quando o parâmetro N vale 1.

Caso Recursivo

No caso recursivo, a função chama a si mesma com os parâmetros modificados. Esta é a essência da recursão em que se utiliza a técnica de resolver um problema maior dividindo-o em instâncias menores do mesmo problema. O caso recursivo deve se aproximar do caso base a cada iteração, ou seja, a cada nova chamada recursiva os parâmetros passados devem estar cada vez mais perto dos valores que produzem o caso base. No exemplo 12. 15 os casos recursivos ocorrem enquanto o parâmetro N é maior que 1.

Nesse exemplo, verifique que a cada nova chamada recursiva o parâmetro passado é N-1, ou seja, a cada chamada o valor passado se aproxima do caso base N = 1.

12.6.2 Vantagens da recursividade

- Funções recursivas geralmente deixam o código mais limpo e elegante;
- Uma tarefa complexa pode ser dividida em sub-tarefas mais simples;
- Gerar uma sequência é consideravelmente mais fácil com recursão do que usando laços aninhados em outros laços;

12.6.3 Desvantagens da recursividade

- Às vezes, a lógica por trás da recursão é difícil de entender, principalmente para quem está iniciando;
- As chamadas recursivas são caras computacionalmente, pois, em geral, usam mais memória do que um laço;
- Funções recursivas são mais difíceis de depurar (localizar e corrigir erros) do que laços;

Vamos agora apresentar alguns exercícios resolvidos.

Exercício Resolvido 12.8

Enunciado: *Escreva um programa que utilize uma função recursiva para realizar uma contagem regressiva. Este programa deve ler do teclado um inteiro que representa a quantidade de toques dessa contagem regressiva. Quando a contagem chegar em zero o programa deve exibir na tela a mensagem "NO AR!!!"*

```
from time import sleep
def Contagem(Cont):
    if Cont == 0:
        print('NO AR!!!')
    else:
        print(Cont)
        sleep(1)
        Contagem(Cont-1)

toques = int(input('Digite a quantidade de toques da contagem: '))
print(f'Atenção para o toque de {toques} segundos...')
Contagem(toques)
```

```
Digite a quantidade de toques da contagem: 5
Atenção para o toque de 5 segundos...
5
4
3
2
1
NO AR!!!
```

Neste exercício a função `Contagem()` recebe o parâmetro `Cont` que deve ser um número inteiro positivo.

Quando `Cont` for igual a zero – este é o caso base – a função exibe a mensagem "`NO AR!!!`";

Enquanto `Cont` for maior que zero – este é o caso recursivo – a função exibe o `Cont` na tela (para produzir a contagem regressiva) e em seguida faz nova chamada à função `Contagem()` desta vez passando como parâmetro o valor `Cont-1`. Perceba que ao fazer isso, a nova chamada se aproxima do caso base.

Adicionalmente, foi usada a função `sleep()`, que pertence ao módulo `time`, para fazer uma pausa na execução do código. O parâmetro passado a essa função é um número real que representa o tempo de pausa em segundos. Ao usá-la criamos uma saída do programa com um efeito interessante para a contagem regressiva, pois após o printe o programa esperará um tempo até o próximo, gerando uma impressão de contagem, de fato. No vídeo deste exercício você pode ver esse efeito.

Exercício Resolvido 12.9

Enunciado: *Escreva uma função recursiva para calcular a somatória dos termos de uma Progressão Aritmética definida pelos parâmetros Prim (primeiro termo), Razao e Qtde (quantidade de elementos). Esses parâmetros devem ser lidos do teclado. Escreva o programa principal para testar a função, que deve exibir na tela o valor dessa soma. Teste seu programa com os casos de teste a seguir.*

Casos de teste: *para Prim = 7; Razao = 4; Qtde = 7 → Soma = 133*

para Prim = 12; Razao = 8; Qtde = 15 → Soma = 1020

para Prim = 2; Razao = 3; Qtde = 6 → Soma = 57

```
def soma(P, R, Q):
    """P: primeiro termo - R: razão - Q: quantidade"""
    if Q > 0:
        a = P + soma(P+R, R, Q-1)
        return a
    else:
        return 0

Prim = int(input('Digite o primeiro termo: '))
Razao = int(input('Digite a razão: '))
Qtde = int(input('Digite a quantidade: '))
Resultado = soma(Prim, Razao, Qtde)
print(f'para Prim = {Prim}; Razao = {Razao}; Qtde = {Qtde} -> Soma = {Resultado}')
```

```
# primeira execução
Digite o primeiro termo: 7
Digite a razão: 4
Digite a quantidade: 7
para Prim = 7; Razao = 4; Qtde = 7 -> Soma = 133

#segunda execução
Digite o primeiro termo: 12
Digite a razão: 8
Digite a quantidade: 15
para Prim = 12; Razao = 8; Qtde = 15 -> Soma = 1020

#terceira execução
Digite o primeiro termo: 2
Digite a razão: 3
Digite a quantidade: 6
para Prim = 2; Razao = 3; Qtde = 6 -> Soma = 57
```

Quadro 12.2 – Lista de chamadas e retornos da função Fatorial

Chamada nº	Condição na entrada	Ciclo de entrada com P + soma (P+R, R, Q-1)	Ciclo de saída das chamadas
0	-	Função principal	Recebe 133 da chamada que fez
1	P = 7, R = 4, Q = 7	7 + Fatorial (7+4, 4, 7-1)	Calcula 7 + 126 e retorna 133
2	P = 11, R = 4, Q = 6	11 + Fatorial (11+4, 4, 6-1)	Calcula 11 + 115 e retorna 126
3	P = 15, R = 4, Q = 5	15 + Fatorial (15+4, 4, 5-1)	Calcula 15 + 100 e retorna 115
4	P = 19, R = 4, Q = 4	19 + Fatorial (19+4, 4, 4-1)	Calcula 19 + 81 e retorna 100
5	P = 23, R = 4, Q = 3	23 + Fatorial (23+4, 4, 3-1)	Calcula 23 + 58 e retorna 81
6	P = 27, R = 4, Q = 2	27 + Fatorial (27+4, 4, 2-1)	Calcula 27 + 31 e retorna 58
7	P = 31, R = 4, Q = 1	31 + Fatorial (31+4, 4, 1-1)	Calcula 31 + 0 e retorna 31
8	P = 35, R = 4, Q = 0	return 0	Retorna 0 à chamada anterior

Na prática esta função recursiva para o cálculo da PA se assemelha em muito à função recursiva para o cálculo do fatorial. O quadro 12.2 ilustra os ciclos de chamadas e retornos sucessivos, mostrando como o

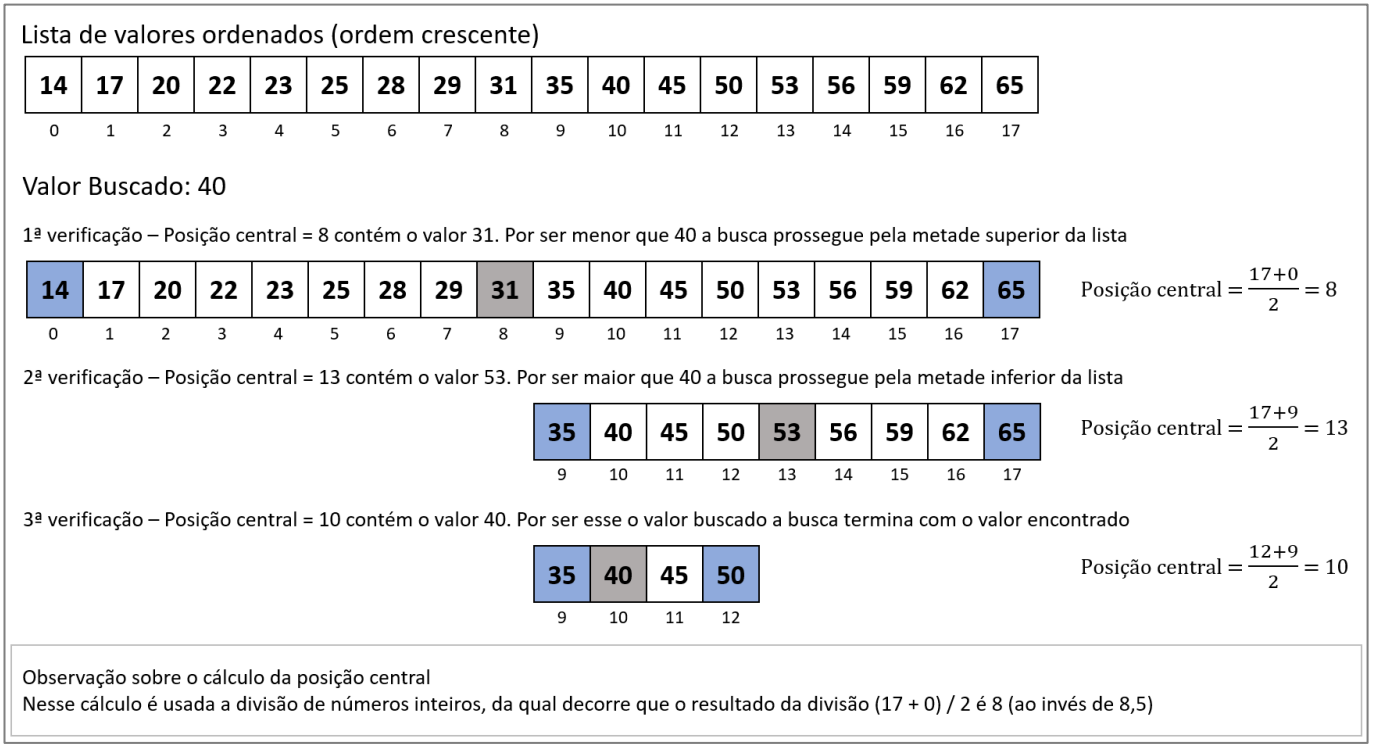
resultado é produzido. Nela, a cada chamada precisamos enviar o que seria o "novo primeiro termo" que é a soma do termo anterior com a razão, bem como enviamos a quantidade subtraída de 1, pois cada chamada executa a incorporação de um termo à somatória. O caso base ocorre quando essa quantidade é igual a zero; e enquanto essa quantidade é maior que zero temos o caso recursivo.

Busca Binária

Busca binária é um algoritmo muito usado para localizar um valor específico dentro de uma lista de valores que esteja organizada em ordem crescente ou decrescente – esta organização é um requisito obrigatório para a busca binária, que é reconhecida por ser rápida e eficiente. Vamos agora explicar o conceito do algoritmo e em seguida apresentar o exercício resolvido 12.10 no qual fazemos sua implementação.

Considere a lista de valores no topo da figura 12.5 e imagine que queremos verificar se o valor 40 está na lista, sem usar qualquer função pronta do Python. Como este é um exercício para compreensão de funções recursivas vamos implementar essa busca como se em Python não existisse tal recurso.

Figura 12.5 – Ilustração do algoritmo de Busca Binária



fonte: o Autor

A lista do exemplo contém 18 elementos, com índice inicial = 0 e o índice final = 17. Na primeira verificação vamos calcular o índice da posição central da lista e comparamos seu conteúdo com o valor 40 procurado: caso seja igual, o algoritmo termina; caso seja diferente, devemos atualizar um dos índices limite de procura na lista e em seguida o algoritmo deve prosseguir para a 2ª verificação.

Como o valor procurado 40 é maior que o valor 31 da posição central, então nossa busca agora estará limitada à parte superior da lista. Atualizamos o índice inicial para 9, o índice final permanece em 17 e repetimos todo o processo anterior. Ao fazer isso o problema da busca fica limitado a uma das metades da lista, dividindo o problema original em outro menor. Esse processo pode ser recursivo, uma vez que são os mesmos passos, porém, realizados com índices alterados. O caso base ocorre em uma de duas situações:

a) achamos o valor procurado; b) os índices inicial e final são reduzidos até não haver nenhum elemento entre eles e não achamos o valor procurado. E o caso recursivo ocorre enquanto `índice inicial` < `índice final` e ainda não achamos o valor procurado.

Exercício Resolvido 12.10

Enunciado: *Escreva um programa que implemente uma busca binária através de uma função recursiva. O programa principal deve permanecer em laço lendo valores inteiros que deve ser buscados na lista.*

Caso de Teste: *Neste exercício será usada uma lista contendo valores fixos e que foi apresentada na figura 12.5. O objetivo é que você possa testar o programa com os valores como mostrado aqui.*

```
def BuscaBin(valor, lista, ini, fim):
    """Procura valor em lista, entre as posições ini:fim"""
    if ini > fim:
        return False
    meio = (ini + fim) // 2
    if valor == lista[meio]:
        return True
    elif valor < lista[meio]:
        # chamada para verificar a metade esquerda da lista
        return BuscaBin(valor, lista, ini, meio-1)
    else:
        # chamada para verificar a metade direita da lista
        return BuscaBin(valor, lista, meio+1, fim)

L = [14, 17, 20, 22, 23, 25, 28, 29, 31, 35, 40, 45, 50, 53, 56, 59, 62, 65]
X = int(input('Digite um valor para pesquisa na lista: '))
while X != 0:
    Achou = BuscaBin(X, L, 0, len(L)-1)
    if Achou != 0:
        print(f'{X} está na lista')
    else:
        print(f'{X} não está na lista')
    X = int(input('Digite um valor para pesquisa na lista: '))
print('Fim do Programa')
```

```
Digite um valor para pesquisa na lista: 40
40 está na lista
Digite um valor para pesquisa na lista: 41
41 não está na lista
Digite um valor para pesquisa na lista: 14
14 está na lista
Digite um valor para pesquisa na lista: 10
10 não está na lista
Digite um valor para pesquisa na lista: 65
65 está na lista
Digite um valor para pesquisa na lista: 66
66 não está na lista
Digite um valor para pesquisa na lista: 0
Fim do Programa
```

A função `BuscaBin()` acima recebe 4 parâmetros: o `valor` a ser procurado; a lista; e as posições inicial e final a serem consideradas na busca. No início da função verificamos se `ini > fim`. Caso isso aconteça chegamos no caso base em que o valor procurado não está na lista. Por outro lado, se `ini >= fim` então calculamos a posição central e verificamos se nela está o `valor` procurado: caso esteja atingimos o segundo caso base e encontramos o `valor`. Se nenhuma dessas duas situações ocorrer fazemos a chamada recursiva para verificar uma das metades da lista de entrada, conforme indicado nos comentários do código acima.

Exercício Proposto 12.11

Enunciado: *Reescreva o programa principal do exercício resolvido 12.10 de modo que no início seja feita a leitura de um inteiro Qtde e a lista de valores seja gerada com Qtde números aleatórios não repetidos. Exiba a lista na tela. Em seguida inicie o processo de busca semelhante ao implementado no exercício 12.10 para testar a função recursiva BuscaBin(). Lembre-se: a lista precisa estar ordenada.*

Exercício Proposto 12.12

Enunciado: *Escreva um programa que calcule a somatória de todos os números inteiros contidos em uma lista. Implemente essa solução usando uma função recursiva.*

Dicas: *Use fatiamento. Considere que L = L[0] + L[1:] , ou seja, uma lista pode ser entendida como sendo a união de seu primeiro elemento (L[0]) com o resto da lista (fatiamento L[1:]).*

O caso base ocorre quando L está vazia, len(L) == 0, e enquanto len(L) > 0 ocorre o caso recursivo.

Exercício Proposto 12.13

Enunciado: *Escreva uma função recursiva para determinar o N-ésimo termo da sequência de Fibonacci. Escreva o programa principal para testar essa função. Neste programa deve ser lido um valor inteiro N e, usando a função recursiva, devem ser calculado e exibido o N-ésimo termo da sequência.*

A sequência de Fibonacci é definida da seguinte forma: a) os dois primeiros termos da sequência são 0 e 1. Do terceiro termo em diante cada termo é a soma dos dois anteriores.

Casos de teste: *p/ N = 5 -> termo = 3 p/ N = 7 -> termo = 8 p/ N = 10 -> termo = 34*

Exercício Proposto 12.14

Enunciado: *Escreva um programa para mostrar na tela o Triângulo de Pascal. Use uma função recursiva para gerar seus elementos.*

Como dado de entrada, esse programa deve ler um número inteiro que será a quantidade de linhas do triângulo a ser exibido.

Para informações sobre o Triângulo de Pascal consulte estes links

https://brasilescola.uol.com.br/matematica/triangulo-pascal.htm

https://www.educamaisbrasil.com.br/enem/matematica/triangulo-de-pascal

Exercício Proposto 12.15

Enunciado: *Escreva um programa que seja capaz de identificar se um subtexto está contido dentro de um texto. Use função recursiva para fazer essa verificação e retornar True caso esteja e False caso não esteja. Considere que o subtexto e o texto contém apenas letras minúsculas.*

Exemplo: *Texto = 'abelardo barbosa foi um grande comunicador brasileiro, criativo, divertido e influente'*

Subtexto = 'abel' → resultado True

Subtexto = 'brasil' → resultado True

Subtexto = 'abcdefg' → resultado False

12.7 FUNÇÕES LAMBDA

Uma função lambda de Python é uma função pequena e anônima que pode receber um ou mais parâmetros e seu código se constitui de uma única expressão.

Ao contrário da função formal que é criada com `def` e pode ser usada no código tantas vezes quanto se queira, a função lambda é criada e usada exclusivamente em um único e determinado ponto do código.

Apesar de simples, trata-se de um recurso poderoso e flexível com uma gama muito grande de usos. A estrutura para sua criação consiste na palavra-chave `lambda`, seguida de uma lista de argumentos, o caractere dois-pontos ':' e a expressão a ser calculada/executada, desta forma:

```
lambda <argumentos>: <expressão>
```

Isto é equivalente a uma função normal assim:

```
def FuncaoNormal(<argumentos>):
    return <expressão>
```

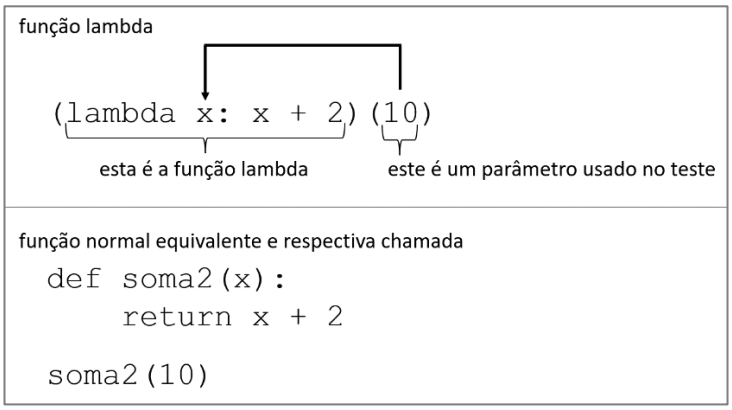
A seguir o exemplo 12.16 mostra uma função lambda capaz de receber um parâmetro x e retornar x + 2.

Exemplo 12.16

```
>>> (lambda x: x + 2)(10)
    12
(exemplo interativo feito com IDE Idle)
```

Neste exemplo, dentro dos primeiros parênteses temos a função lambda e no segundo parênteses temos o valor 10, que nada mais é do que o valor usado no teste da função. A figura 12.6 mostra a equivalência entre a função lambda e a função normal.

Figura 12.6 – Comparação entre função lambda e função normal



12.7.1 FUNÇÃO LAMBDA ATRIBUÍDA A UM OBJETO

Funções lambda por definição são anônimas, no entanto, é possível atribuí-la a um objeto. Após isso esse objeto pode ser usado para realizar diversas chamadas sucessivas à ela.

Exemplo 12.17 Teste este código no Idle

```
>>> media = lambda n1, n2, n3: (n1 + n2 + n3) / 3
>>> print(media(8.0, 6.5, 7.1))
    7.2
>>> print(media(7.4, 5.8, 6.3))
    6.5
>>> print(media(3.3, 2.7, 6.9))
    4.3
(exemplo interativo feito com IDE Idle)
```

Neste exemplo 12.17 o objeto `media` recebeu o retorno a função lambda. A partir disso, o objeto `media` terá um funcionamento equivalente à uma função normal nomeada. A seguir um novo exemplo mostra o uso de uma função lambda para montar uma concatenação de strings.

Exemplo 12.18 Teste este código no Idle

```
>>> endereco = lambda rua, num, cep: f'{rua}, {num} - {cep}'
>>> endereco('Av. dos Estados', 3586, '08310-000')
    'Av. dos Estados, 3586 - 08310-000'
>>> endereco('Rua Tutóia', 583, '09240-030')
    'Rua Tutóia, 583 - 09240-030'
(exemplo interativo feito com IDE Idle)
```

12.7.2 USO COMBINADO DE FUNÇÕES LAMBDA COM A FUNÇÃO `map`

A função `map()` aplica a `função` ao `iterável` passado. Caso a `função` tenha mais de um argumento outros `iteráveis` devem ser passados.

```
map(<função>, <iterável>, <*iteráveis>)
```

Vamos ver os exemplos. No 12.19 a lista `dados` é aplicada à função `map()` com o uso da função lambda que multiplica por 10 cada valor da lista.

Exemplo 12.19 

```
>>> dados = [2, 3, 4, 5, 6]
>>> list(map(lambda x: x * 10, dados))
    [20, 30, 40, 50, 60]
(exemplo interativo feito com IDE Idle)
```

No exemplo 12.20 a função lambda recebe dois parâmetros e retorna a soma dos dois. Neste caso a função `map()` é alimentada com as listas `dados1` e `dados2`.

Exemplo 12.20 Teste este código no Idle

```
>>> dados1 = [2, 3, 4, 5, 6]
>>> dados2 = [100, 200, 300, 400, 500]
>>> list(map(lambda x, y: x + y, dados1, dados2))
    [102, 203, 304, 405, 506]
(exemplo interativo feito com IDE Idle)
```

12.7.3 USO COMBINADO DE FUNÇÕES LAMBDA COM A FUNÇÃO `filter`

A função `filter()` tem a estrutura a seguir.

```
filter(<função>, <iterável>)
```

Ela cria um iterador a partir dos elementos do `iterável` para os quais a `função` é verdadeira. O exemplo mostra a aplicação combinada de função lambda com a função `filter()`.

Exemplo 12.21 Teste este código no Idle

```
>>> dados = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> list(filter(lambda x: x % 3 == 0, dados))
    [3, 6, 9]
(exemplo interativo feito com IDE Idle)
```

Neste exemplo são mostrados os elementos da lista `dados` que são divisíveis por 3.

12.7.4 ORDENAÇÃO DE DICIONÁRIOS POR VALOR COM O USO DE FUNÇÕES LAMBDA

Como já vimos, dicionários são elementos poderosos de Python. Há situações em que precisamos fazer certas manipulações de seus dados em que as funções lambda tem papel preponderante.

Considere o dicionário `prods` criado no exemplo 12.22. Nele, a proposta é termos um código de produto como chave de cada elemento e um número real que é o preço unitário. Os elementos do dicionário não apresentam qualquer ordenação.

Em seguida, usamos a função `sorted()` para realizar uma ordenação. Por padrão essa função produzirá um retorno que é uma lista de tuplas contendo os elementos do dicionário original ordenados pela chave.

Exemplo 12.22

```
>>> prods = {'121': 23.90, '189': 14.33, '152': 36.95, '175': 19.35}
>>> sorted(prods.items())   # ordena o dicionário pelas chaves dos elementos
    [('121', 23.9), ('152', 36.95), ('175', 19.35), ('189', 14.33)]

# ordenação do dicionário pelos valores dos elementos
>>> sorted(prods.items(), key = lambda x: x[1])
    [('189', 14.33), ('175', 19.35), ('121', 23.9), ('152', 36.95)]
```

(exemplo interativo feito com IDE Idle)

Caso queiramos ordenar os elementos do dicionário pelos preços unitários teremos que usar uma função lambda com o parâmetro `key` da função `sorted()`. Para fazer isso necessitamos da atribuição `key = lambda x: x[1]`. Como cada elemento manipulado em `sorted()` é uma tupla: `x[0]` contém o código e `x[1]` contém o preço. Assim, a expressão retornada pela função lambda deve ser `x[1]` que será atribuído ao parâmetro `key` da função `sorted()` que será usada na ordenação.

Capítulo 13
LIST COMPREHENSION

13.1 CONCEITO

List Comprehension é um recurso existente na linguagem Python que fornece uma forma compacta e prática de criar listas sem a necessidade do uso de laços for ou while. Este recurso oferece uma sintaxe mais curta quando você deseja criar uma nova lista com base nos valores de outra lista.

Usos frequentes desse recurso são:

- criação de uma lista onde os elementos resultam de um cálculo aplicado a cada elemento de um iterável ou de uma sequência pré-existente; ou
- criação de um subconjunto (sublista) através da aplicação de certas condições a um iterável ou sequência pré-existente.

Não há uma tradução do termo "*list comprehension*" para português que seja amplamente aceita e difundida, por isso neste texto usaremos o termo em inglês.

Suponha que, dada uma lista carregada com valores inteiros, queiramos gerar uma segunda lista com todos esses valores multiplicados por 2. No exemplo 13.1 mostramos dois modos de fazer isso.

Exemplo 13.1

```
>>> lista = [31, -17, 26, 15, -35, -9, 20]
# Modo 1: usando um laço iterador e o método append para criar a nova lista
>>> modo1 = []
>>> for x in lista:
>>>     modo1.append(x * 2)
>>> print(modo1)
    [62, -34, 52, 30, -70, -18, 40]

# Modo 2: usando list comprehension
>>> modo2 = [x * 2 for x in lista]
>>> print(modo2)
    [62, -34, 52, 30, -70, -18, 40]
```

(exemplo interativo feito com IDE Idle)

Os dois modos usados acima produzem o mesmo resultado. No modo 1 é preciso criar uma lista vazia e implementar um laço iterador para acrescentar cada elemento à nova lista. No modo 2 com apenas uma linha fizemos o mesmo trabalho, mostrando que list comprehension é mais compacto, prático e legível.

No exemplo 13.2 acrescentamos um detalhe a mais em relação ao exemplo anterior: queremos criar um filtro, adicionando à lista resultante apenas os elementos positivos da lista original, multiplicados por 2. Para isso, pode-se acrescentar à expressão list comprehension uma cláusula `if` que será usada para decidir quais elementos serão incluídos na nova lista, e quais não serão.

Exemplo 13.2

```
>>> lista = [31, -17, 26, 15, -35, -9, 20]
# Modo 1: usando um laço iterador e o método append para criar a nova lista
>>> modo1 = []
>>> for x in lista:
>>>     if x > 0:
>>>         modo1.append(x * 2)
>>> print(modo1)
    [62, 52, 30, 40]

# Modo 2: usando list comprehension
>>> modo2 = [x * 2 for x in lista if x > 0]
>>> print(modo2)
    [62, 52, 30, 40]
```

(exemplo interativo feito com IDE Idle)

Em síntese, list comprehension é um modo legível, compacto e elegante de criar uma lista a partir de qualquer objeto iterável existente, consistindo em uma maneira mais simples de criar uma nova lista a partir dos valores de uma lista que você já possui.

Geralmente é uma única linha de código entre colchetes. E você pode usá-lo para filtrar, formatar, modificar ou realizar outras pequenas tarefas em iteráveis existentes, como listas, tuplas, strings, range, conjuntos, dataframes, listas de arrays e assim por diante.

13.2 Estrutura do list comprehension

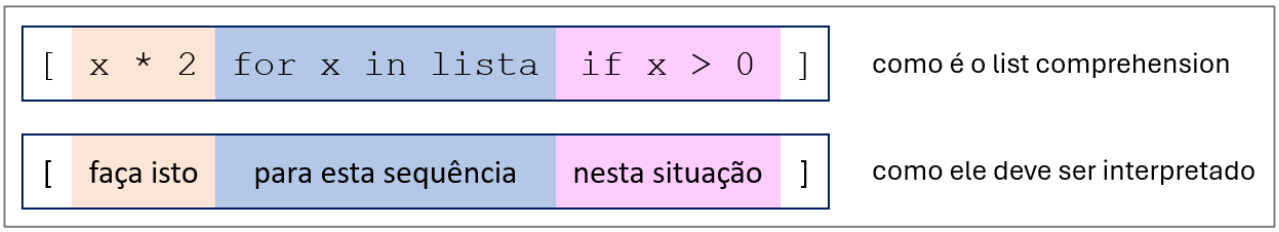
Agora que já vimos os exemplos vamos dar atenção às partes que compõe uma expressão list comprehension. Deve ser construída dentro de colchetes e sua estrutura é esta:

```
<nova lista> = [<expressão> for <elemento> in <iterável> if <condição>]
```

- `<nova lista>` é a lista a ser criada;
- `<expressão>` é qualquer expressão válida em Python que retorne algum objeto e use o `<elemento>`
- `<elemento>` é um objeto presente no `<iterável>`
- `<iterável>` é uma lista pré-existente, ou qualquer iterável produzido usando os recursos existentes no Python, como `range()`, `map()`, `zip()`, etc
- `<condição>` é uma condição válida envolvendo o `<elemento>`

Graficamente podemos ilustrar essa construção conforme mostrado na figura 12.1

Figura 13.1 – Interpretação do List Comprehension



Exercício Resolvido 13.1

Enunciado: *Considere que você deve aplicar um aumento percentual a todos os preços que estão em uma lista. Escreva um programa que leia essa lista do teclado. Os valores devem ser lidos enquanto não for digitado zero. Na sequência leia a porcentagem de aumento.*

Em seguida, usando list comprehension, faça a aplicação desse aumento a todos os valores e mostre na tela.

```
Precos = []
print('Forneça os preços para a lista. Zero para terminar.')
preco = float(input('  digite um valor real: '))
while preco != 0:
    Precos.append(preco)
    preco = float(input('  digite um valor real: '))
print('Os preços são estes:')
print(Precos)

aumento = float(input('Digite a porcentagem de aumento (sem o %): '))
# este list comprehension aplica o aumento
NovosPrecos = [valor * (1+aumento/100) for valor in Precos]
# exibição do resultado, um valor por linha
for valor in NovosPrecos:
    print(f'{valor:.2f}')
```

```
Forneça os preços para a lista. Zero para terminar.
  digite um valor real: 50
  digite um valor real: 120
  digite um valor real: 20
  digite um valor real: 100
  digite um valor real: 200
  digite um valor real: 0
Os preços são estes:
[50.0, 120.0, 20.0, 100.0, 200.0]
Digite a porcentagem de aumento (sem o %): 10
55.00
132.00
22.00
110.00
220.00
```

13.3 USO COMBINADO DE LIST COMPREHENSION E INLINE IF

Os vários recursos existentes em Python podem – e devem – ser usados em conjunto. Nesta seção a ideia é combinar list comprehension e o comando `inline if`, visto no capítulo 8 deste Ebook.

Para isto vamos trabalhar com uma modificação do enunciado do exercício 13.1.

Exercício Resolvido 13.2 — Esta solução está incluída no vídeo do ex.resolvido 13.1

Enunciado: *Altere o programa anterior para aplicar a porcentagem de aumento apenas para os valores menores que 100, sendo que os demais devem ser mantidos inalterados.*

```
Precos = []
print('Forneça os preços para a lista. Zero para terminar.')
preco = float(input('  digite um valor real: '))
while preco != 0:
    Precos.append(preco)
    preco = float(input('  digite um valor real: '))
print('Os preços são estes:')
print(Precos)

aumento = float(input('Digite a porcentagem de aumento (sem o %): '))
# este list comprehension aplica o aumento
NovosPrecos = [valor * (1+aumento/100) for valor in Precos if valor < 100]
# exibição do resultado, um valor por linha
for valor in NovosPrecos:
    print(f'{valor:.2f}')
```

```
Forneça os preços para a lista. Zero para terminar.
  digite um valor real: 50
  digite um valor real: 120
  digite um valor real: 20
  digite um valor real: 100
  digite um valor real: 200
  digite um valor real: 0
Os preços são estes:
[50.0, 120.0, 20.0, 100.0, 200.0]
Digite a porcentagem de aumento (sem o %): 10
55.00
22.00
```

Veja a seguir a única diferença entre os exercícios resolvidos 13.1 e 13.2:

```
NovosPrecos = [valor * (1+aumento/100) for valor in Precos]       # versão usada no 13.1

NovosPrecos = [valor * (1+aumento/100) for valor in Precos if valor < 100]   # e no 13.2
```

No 13.2 foi aplicada a cláusula `if` ao list comprehension.

Porém isso teve um efeito colateral indesejado neste caso. A lista `NovosPrecos` ficou limitada a dois valores porque ocorreu a filtragem da lista original. Os elementos que não se enquadravam no critério `valor < 100` não foram incluídos na lista gerada. E isso está correto, do ponto de vista do list comprehension, que foi projetado para ser assim.

Agora voltemos ao enunciado do 13.2, destacando a frase que contém este requisito "*sendo que os demais devem ser mantidos inalterados.*" A questão é: como resolvemos isso usando list comprehension? Ou seja, precisamos que alguns valores sofram o aumento fornecido e outros não, porém todos os valores da lista original maiores ou iguais a 100 devem estar na lista gerada.

A solução para esta questão é o uso de um inline `if` dentro do list comprehenshion, desta forma:

```
NovosPrecos = [valor * (1+aumento/100) if valor < 100 else valor for valor in Precos]
```

Para melhor compreensão vamos separar as partes. A lista NovosPrecos será construída com um list comprehension que envolve uma `expressao` e um iterável.

```
NovosPrecos = [expressao for valor in Precos]  # este é o list comprehension que a produz
```

A `expressao` por sua vez, é este inline `if`:

```
expressao = valor * (1+aumento/100) if valor < 100 else valor
```

Esta combinação resulta na linha mostrada antes

```
NovosPrecos = [valor * (1+aumento/100) if valor < 100 else valor for valor in Precos]
```

que por sua vez é equivalente a este código

```
NovosPrecos = []
for valor in Precos:
    if valor < 100:
        NovosPrecos.append(valor * (1+aumento/100))
    else:
        NovosPrecos.append(valor)
```

Ao criar os seus programas você pode optar por usar um desses dois modos de fazer pois ambos estão corretos. Porém o uso combinado de list comprehension com inline `if` é uma solução mais Pythonica e usada com mais frequência pelos programadores experientes.

13.4 USO COMBINADO DE LIST COMPREHENSION E FUNÇÕES LAMBDA

List comprehension e funções lambda são outra combinação poderosa em Python. Vamos demonstrar isso com o exercício resolvido 13.3. Neste exercício temos uma lista de temperaturas anotadas em graus Fafrenheit (usado nos Estados Unidos) e desejamos convertê-la para graus Celsius.

Exercício Resolvido 13.3

Enunciado: *Escreva um programa que converta uma lista de temperaturas registradas na escala Fahrenheit para graus na escala Celsius. Exiba os valores resultantes com uma casa decimal.*

A conversão é feita com uso da seguinte fórmula:

$$Celsius = (Fahrenheit - 32).\,^{5}/_{9}$$

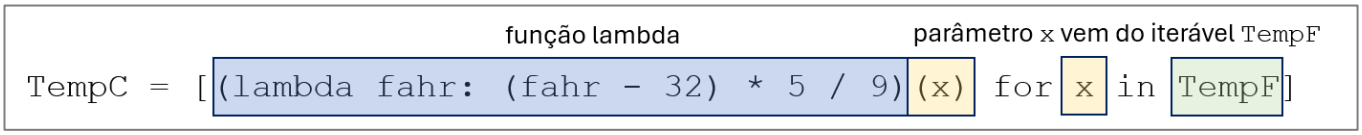
```
>>> TempF = [83, 91, 79, 95, 104, 100, 98]
>>> TempC = [(lambda fahr: (fahr - 32) * 5 / 9)(x) for x in TempF]
>>> for tc in TempC:
>>>     print(f'{tc:.1f}', end='  ')
    28.3  32.8  26.1  35.0  40.0  37.8  36.7
(exemplo interativo feito com IDE Idle)
```

Neste exemplo o cálculo dos elementos da lista `TempC` é feito com esta função lambda

```
(lambda fahr: (fahr - 32) * 5 / 9)
```

Esta função, por sua vez, esta inserida no list comprehension, recebendo o parâmetro `x` cuja origem é a lista iterável `TempF`.

Figura 13.2 – Ilustração da combinação de list comprehension com função lambda



13.5 Conclusão

Além de ser simples, compacta e rápida, o recurso de list comprehension do Python também é confiável e eficiente. E você pode usá-lo em diversas circunstâncias, tais como fazer uma filtragem, modificações ou formatação em outros objetos que estejam dentro de iteráveis. Também é uma boa escolha se você deseja manter seu código compacto e legível.

Na prática verifica-se com frequência que uma mesma operação feita com list comprehension executa mais rápido do que com o uso de iterações for ou laços while. Não é possível estabelecer isso como regra, mas sempre é possível implementar as duas soluções e testar qual é mais veloz. Portanto, embora não seja consenso total, existe entre os programadores Python a experiência de que list comprehension é mais veloz.

Por outro lado, quando é necessário fazer uma filtragem mais complexa de dados, associada a múltiplos critérios de escolha o list comprehension pode não ser uma boa ideia. Embora seja possível criar list comprehension com condicionais aninhados, esse estilo de código acaba ficando muito embolado e com a legibilidade muito comprometida, perdendo-se assim uma das principais vantagens do uso desse recurso.

Assim, como em muitas outras situações, sempre que tiver que decidir entre usar um laço ou um list comprehension, analise o caso específico em que você está trabalhando, e um conjunto de boas práticas e seu bom senso para decidir qual solução é mais conveniente.

Exercício Proposto 13.1

Enunciado: *Use list comprehension para produzir uma lista com todos os valores inteiros positivos divisíveis por N menores que Limite. Os números inteiros N e Limite devem ser lidos do teclado.*

Dica: *Use a classe range*

Exercício Proposto 13.2

Enunciado: *Escreva um programa que leia um número inteiro positivo N do teclado e crie uma lista com N números aleatórios entre -1000 e 1000. Em seguida use list comprehension para produzir uma nova lista que contenha apenas os valores positivos da lista original.*

Dica: *Use a a função randint() para gerar os números aleatórios*

Exercício Proposto 13.3

Enunciado: *Escreva um programa que leia um número inteiro positivo N do teclado e crie uma lista com N números aleatórios entre -1000 e 1000. Em seguida use list comprehension para produzir duas novas listas, uma com os valores positivos e o zero e outra com valores negativos.*

Dica: *Use a a função randint() para gerar os números aleatórios*

Exercício Proposto 13.4

Enunciado: *Escreva um programa que leia um texto qualquer. Em seguida inicie um laço no qual será lido um único caractere. Esse laço termina quando essa leitura for vazia (nenhum caractere digitado).*

Para cada caractere fornecido, use um list comprehension para determinar e informar quantas ocorrências desse caractere há no texto.

HUAWEI

Fundunesp

Centro Paula Souza

SÃO PAULO
GOVERNO DO ESTADO
SÃO PAULO SÃO TODOS