

Podemos também criar tuplas vazias, escrevendo apenas os parênteses:

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

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

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

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

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

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

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

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

CAPÍTULO 7

Trabalhando com strings

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

► Listagem 7.1 – Alteração de uma string

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

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

► Listagem 7.2 – Convertendo uma string em lista

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

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

7.1 Verificação parcial de strings

Quando você precisar verificar se uma string começa ou termina com alguns caracteres, você pode usar os métodos `startswith` e `endswith`. Esses métodos verificam apenas os primeiros (`startswith`) ou os últimos (`endswith`) caracteres da string, retornando `True` caso sejam iguais ou `False` em caso contrário.

► Listagem 7.3 – Verificação parcial de strings

```
>>> nome="João da Silva"
>>> nome.startswith("João")
True
>>> nome.startswith("joão")
False
>>> nome.endswith("Silva")
True
```

Observe a listagem 7.3. Veja que comparamos “João da Silva” com “joão” e obtivemos `False`. Esse é um detalhe pequeno, mas importante, pois `startswith` e `endswith` consideram letras maiúsculas e minúsculas como letras diferentes.

Você pode resolver esse tipo de problema convertendo a string para maiúsculas ou minúsculas antes de realizar a comparação. O método `lower` retorna uma cópia da string com todos os caracteres minúsculos, e o método `upper` retorna uma cópia com todos os caracteres maiúsculos. Veja os exemplos na listagem 7.4.

► Listagem 7.4 – Exemplos de conversão em maiúsculas e minúsculas

```
>>> s="O Rato roeu a roupa do Rei de Roma"
>>> s.lower()
'o rato roeu a roupa do rei de roma'
>>> s.upper()
'O RATO ROEU A ROUPA DO REI DE ROMA'
>>> s.lower().startswith("o rato")
True
>>> s.upper().startswith("O RATO")
True
```

Não se esqueça de comparar com uma string onde todos os caracteres são maiúsculos ou minúsculos, dependendo se você utilizou `upper` ou `lower`, respectivamente.

Outra forma de verificar se uma palavra pertence a uma string é utilizando o operador `in`. Vejamos os exemplos da listagem 7.5.

► Listagem 7.5 – Pesquisa de palavras em uma string usando `in`

```
>>> s="Maria Amélia Souza"
>>> "Amélia" in s
True
>>> "Maria" in s
True
>>> "Souza" in s
True
>>> "a A" in s
True
>>> "amélia" in s
False
```

Você também pode testar se uma string não está contida em outra, utilizando `not in` (Listagem 7.6).

► Listagem 7.6 – Pesquisa de palavras em uma string usando `not in`

```
>>> s="Todos os caminhos levam a Roma"
>>> "levam" not in s
False
>>> "Caminhos" not in s
True
>>> "AS" not in s
True
```

Veja que aqui também letras maiúsculas e minúsculas são diferentes. Você pode combinar `lower` e `upper` com `in` e `not in` para ignorar esse tipo de diferença, como mostra a listagem 7.7.

► Listagem 7.7 – Combinação de `lower` e `upper` com `in` e `not in`

```
>>> s="João comprou um carro"
>>> "joão" in s.lower()
True
>>> "CARRO" in s.upper()
```

```
True
>>> "comprou" not in s.lower()
False
>>> "barco" not in s.lower()
True
```

O operador `in` também pode ser utilizado com listas normais, facilitando, assim, a pesquisa de elementos dentro de uma lista.

7.2 Contagem

Se você precisar contar as ocorrências de uma letra ou palavra em uma string, utilize o método `count`. Veja o exemplo na listagem 7.8.

► Listagem 7.8 – Contagem de letra e palavras

```
>>> t="um tigre, dois tigres, três tigres"
>>> t.count("tigre")
3
>>> t.count("tigres")
2
>>> t.count("t")
4
>>> t.count("z")
0
```

7.3 Pesquisa de strings

Para pesquisar se uma string está dentro de outra e obter a posição da primeira ocorrência, você pode utilizar o método `find` (Listagem 7.9).

► Listagem 7.9 – Pesquisa de strings com find

```
>>> s="Alô mundo"
>>> s.find("mun")
4
>>> s.find("ok")
-1
```

Caso a string seja encontrada, você obterá um valor maior ou igual a zero, ou -1, em caso contrário. Observe que o valor retornado, quando maior ou igual a zero, é igual ao índice que pode ser utilizado para obter o primeiro caractere da string procurada.

Se o objetivo for pesquisar, mas da direita para a esquerda, utilize o método `rfind`, que realiza essa tarefa (Listagem 7.10).

► Listagem 7.10 – Pesquisa de strings com rfind

```
>>> s="Um dia de sol"
>>> s.rfind("d")
7
>>> s.find("d")
3
```

Tanto `find` quanto `rfind` suportam duas opções adicionais: `início (start)` e `fim (end)`. Se você especificar `início`, a pesquisa começará a partir dessa posição. Se você especificar o `fim`, a pesquisa utilizará essa posição como último caractere a considerar na pesquisa. Veja os exemplos na listagem 7.11.

► Listagem 7.11 – Pesquisa de strings, limitando o início ou o fim

```
>>> s="um tigre, dois tigres, três tigres"
>>> s.find("tigres")
15
>>> s.rfind("tigres")
28
>>> s.find("tigres",7) #início=7
15
>>> s.find("tigres",30) #início=30
-1
>>> s.find("tigres",0,10) #início=0 fim=10
-1
```

Podemos usar o valor retornado por `find` e `rfind` para achar todas as ocorrências da string. Por exemplo, o programa da listagem 7.12 produz a saída da listagem 7.13.

► Listagem 7.12 – Pesquisa de todas as ocorrências

```
s="um tigre, dois tigres, três tigres"
p=0
while(p>-1):
    p=s.find("tigre", p)
    if p>=0:
        print("Posição: %d" % p)
        p+=1
```

► Listagem 7.13 – Resultado da pesquisa

Posição: 3

Posição: 15

Posição: 28

Os métodos `index` e `rindex` são bem parecidos com `find` e `rfind`, respectivamente. A maior diferença é que se a substring não for encontrada, `index` e `rindex` lançam uma exceção do tipo `ValueError`.

Exercício 7.1 Escreva um programa que leia duas strings. Verifique se a segunda ocorre dentro da primeira e imprima a posição de início.

1^a string: AABBEFAATT

2^a string: BE

Resultado: BE encontrado na posição 3 de AABBEFAATT

Exercício 7.2 Escreva um programa que leia duas strings e gere uma terceira com os caracteres comuns às duas strings lidas.

1^a string: AAACTBFF

2^a string: CBT

Resultado: CBT

A ordem dos caracteres da string gerada não é importante, mas deve conter todas as letras comuns a ambas.

Exercício 7.3 Escreva um programa que leia duas strings e gere uma terceira apenas com os caracteres que aparecem em uma delas.

1^a string: CTA

2^a string: ABC

3^a string: BT

A ordem dos caracteres da terceira string não é importante.

Exercício 7.4 Escreva um programa que leia uma string e imprima quantas vezes cada caractere aparece nessa string.

String: TTAAC

Resultado:

T: 2x

A: 2x

C: 1x

Exercício 7.5 Escreva um programa que leia duas strings e gere uma terceira, na qual os caracteres da segunda foram retirados da primeira.

1^a string: AAFTGGAA

2^a string: TG

3^a string: AAAA

Exercício 7.6 Escreva um programa que leia três strings. Imprima o resultado da substituição na primeira, dos caracteres da segunda pelos da terceira.

1^a string: AATTCGAA

2^a string: TG

3^a string: AC

Resultado: AAAACCAA

7.4 Posicionamento de strings

Python também traz métodos que ajudam a apresentar strings de formas mais interessantes. Vejamos o método `center`, que centraliza a string em um número de posições passado como parâmetro, preenchendo com espaços à direita e à esquerda até que a string esteja centralizada (Listagem 7.14).

► Listagem 7.14 – Centralização de texto em uma string

```
>>> s="tigre"
>>> print("X"+s.center(10)+"X")
X tigre X
>>> print("X"+s.center(10,".")+ "X")
X..tigre...X
```

Se, além do tamanho, você também passar o caractere de preenchimento, este será utilizado no lugar de espaços em branco.

Se o que você deseja é apenas completar a string com espaços à esquerda, você pode utilizar o método `ljust`. Se deseja completar com espaços à direita, utilize `rjust` (Listagem 7.15).

► Listagem 7.15 – Preenchimento de strings com espaços

```
>>> s="tigre"
>>> s.ljust(20)
'tigre'
>>> s.rjust(20)
'          tigre'
>>> s.ljust(20,".")
'tigre.....'
>>> s.rjust(20,"-")
'-----tigre'
```

Essas funções são úteis quando precisamos criar relatórios ou simplesmente alinhar a saída dos programas.

7.5 Quebra ou separação de strings

O método `split` quebra uma string a partir de um caractere passado como parâmetro, retornando uma lista com as substrings já separadas. Veja um exemplo na listagem 7.16.

► Listagem 7.16 – Separação de strings

```
>>> s="um tigre, dois tigres, três tigres"
>>> s.split(",")
['um tigre', ' dois tigres', ' três tigres']
>>> s.split(" ")
['um', 'tigre,', 'dois', 'tigres,', 'três', 'tigres']
>>> s.split()
['um', 'tigre,', 'dois', 'tigres,', 'três', 'tigres']
```

Observe que o caractere que utilizamos para dividir a string não é retornado na lista, ou seja, ele é utilizado para separar a string e depois descartado. Se você deseja separar uma string, com várias linhas de texto, você pode utilizar o método `splitlines` (Listagem 7.17).

► Listagem 7.17 – Quebra de strings de várias linhas

```
>>> m="Uma linha\noutra linha\nne mais uma\n"
>>> m.splitlines()
['Uma linha', 'outra linha', 'e mais uma']
```

7.6 Substituição de strings

Para substituir trechos de uma string por outros, utilize o método `replace`. Com o método `replace`, o primeiro parâmetro é a string a substituir; e o segundo, o conteúdo que a substituirá. Opcionalmente, podemos passar um terceiro parâmetro que limita quantas vezes queremos realizar a repetição. Vejamos alguns exemplos na listagem 7.18.

► Listagem 7.18 – Substituição de strings

```
>>> s="um tigre, dois tigres, três tigres"
>>> s.replace("tigre", "gato")
'um gato, dois gatos, três gatos'
```

```
>>> s.replace("tigre", "gato", 1)
'um gato, dois tigres, três tigres'
>>> s.replace("tigre", "gato", 2)
'um gato, dois gatos, três tigres'
>>> s.replace("tigre", "")
'um , dois s, três s'
>>> s.replace("", "-")
'-u-m- -t-i-g-r-e-, - -d-o-i-s- -t-i-g-r-e-s-, - t-r-ê-s- -t-i-g-r-e-s-'
```

Se você passar uma string vazia no segundo parâmetro, o trecho será apagado. Se o primeiro parâmetro for uma string vazia, o segundo será inserido antes de cada caractere da string.

7.7 Remoção de espaços em branco

O método `strip` é utilizado para remover espaços em branco do início ou fim da string. Já os métodos `lstrip` e `rstrip` removem apenas os caracteres em branco à esquerda ou à direita, respectivamente (Listagem 7.19).

► Listagem 7.19 – Remoção de espaços em branco com `strip`, `lstrip` e `rstrip`

```
>>> t=" Olá "
>>> t.strip()
'Olá'
>>> t.lstrip()
'Olá '
>>> t.rstrip()
' Olá'
```

Se você passar um parâmetro tanto para `strip` quanto para `lstrip` ou `rstrip`, este será utilizado como caractere a remover (Listagem 7.20).

► Listagem 7.20 – Remoção de caracteres com `strip`, `lstrip` e `rstrip`

```
>>> s=".....Olá///...."
>>> s.lstrip(".")
'//Olá///...'
```

```
>>> s.rstrip(".")
'....Olá///'
>>> s.strip(".")
'//Olá///'
>>> s.strip("./")
'Olá'
```

7.8 Validação por tipo de conteúdo

Strings em Python podem ter seu conteúdo analisado e verificado utilizando-se métodos especiais. Esses métodos verificam se todos os caracteres são letras, números ou uma combinação deles. Vejamos alguns exemplos na listagem 7.21.

► Listagem 7.21 – Validação de strings por seu conteúdo

```
>>> s="125"
>>> p="alô mundo"
>>> s.isalnum()
True
>>> p.isalnum()
False
>>> s.isalpha()
False
>>> p.isalpha()
False
```

O método `isalnum` retorna verdadeiro se a string não estiver vazia, e se todos os seus caracteres são letras e/ou números. Se a string contiver outros tipos de caracteres, como espaços, vírgula, exclamação, interrogação ou caracteres de controle, retorna `False`.

Já `isalpha` é mais restritivo, retornando verdadeiro apenas se todos os caracteres forem letras, incluindo vogais acentuadas. Retorna falso se algum outro tipo de caractere for encontrado na string ou se estiver vazia.

O método `isdigit` verifica se o valor consiste em números, retornando `True` se a string não estiver vazia e contiver apenas números. Se a string contiver espaços, pontos, vírgulas ou sinais (+ ou -), retorna falso (Listagem 7.22).

► Listagem 7.22 – Validação de strings com números

```
>>> "771".isdigit()
True
>>> "10.4".isdigit()
False
>>> "+10".isdigit()
False
>>> "-5".isdigit()
False
```

Os métodos `isdigit` e `isnumeric` são parecidos, e diferenciá-los envolve um conhecimento de Unicode mais aprofundado. `isdigit` retorna `True` para caracteres definidos como dígitos numéricos em Unicode, indo além de nossos 0 a 9, como, por exemplo um 9 tibetano (`\u0f29`). `isnumeric` é mais abrangente, incluindo dígitos e representações numéricas como frações, por exemplo, 1/3 (`\u2153`). Vejamos alguns testes na listagem 7.23.

► Listagem 7.23 – Diferenciação de `isnumeric` de `isdigit`

```
>>> umterço="\u2153"
>>> novetibetano="\u0f29"
>>> umterço.isdigit()
False
>>> umterço.isnumeric()
True
>>> novetibetano.isdigit()
True
>>> novetibetano.isnumeric()
True
```

Podemos também verificar se todos os caracteres de uma string são letras maiúsculas ou minúsculas usando `isupper` e `islower`, respectivamente (Listagem 7.24).

► Listagem 7.24 – Verificação de maiúsculas e minúsculas

```
>>> s="ABC"
>>> p="abc"
>>> e="aBc"
```

Capítulo 7 ■ Trabalhando com strings

```
>>> s.isupper()
True
>>> s.islower()
False
>>> p.isupper()
False
>>> p.islower()
True
>>> e.isupper()
False
>>> e.islower()
False
```

Temos também como verificar se a string contém apenas caracteres em branco, como espaços, marcas de tabulação (TAB), quebras de linha (LF) ou retorno de carro (CR). Para isso, vamos utilizar o método `isspace`, como mostra a listagem 7.25.

► Listagem 7.25 – Verificação se a string contém apenas caracteres de espaçamento

```
>>> "\t\n\r      ".isspace()
True
>>> "\tAlô".isspace()
False
```

Se você precisar verificar se algo pode ser impresso na tela, o método `isprintable` pode ajudar. Ele retorna `False` se algum caractere que não pode ser impresso for encontrado na string. Você pode utilizá-lo para verificar se a impressão de uma string pode causar efeitos indesejados no terminal ou na formatação de um arquivo (Listagem 7.26).

► Listagem 7.26 – Verificação se a string pode ser impressa

```
>>> "\n\t".isprintable()
False
>>> "\nAlô".isprintable()
False
>>> "Alô mundo".isprintable()
True
```

7.9 Formatação de strings

A versão 3 do Python introduziu uma nova forma de representar máscaras em strings. Essa nova forma é mais poderosa que as tradicionais máscaras que utilizamos, combinando %d, %s, %f.

A nova forma representa os valores a substituir, entre chaves. Vejamos um exemplo na listagem 7.27.

► Listagem 7.27 – Formatação de strings com o método format

```
>>> "{0} {1}".format("Alô", "Mundo")
'Alô Mundo'
>>> "{0} x {1} R${2}".format(5,"maçã", "1.20")
'5 x maçã R$1.20'
```

O número entre colchetes é uma referência aos parâmetros passados ao método `format`, onde 0 é o primeiro parâmetro; 1, o segundo; e assim por diante, como os índices de uma lista. Uma das vantagens da nova sintaxe é poder utilizar o mesmo parâmetro várias vezes na string (Listagem 7.28).

► Listagem 7.28 – Uso do mesmo parâmetro mais de uma vez

```
>>> "{0} {1} {0}".format("-", "x")
'- x -'
```

Isso também permite a completa reordenação da mensagem, como imprimir os parâmetros em outra ordem (Listagem 7.29).

► Listagem 7.29 – Alteração da ordem de utilização dos parâmetros

```
>>> "{1} {0}".format("primeiro", "segundo")
'segundo primeiro'
```

A nova sintaxe permite também especificar a largura de cada valor, utilizando o símbolo de dois pontos (:) após a posição do parâmetro, como 0:10 da listagem 7.30, que significa: substitua o primeiro parâmetro com uma largura de 10 caracteres. Se o primeiro parâmetro for menor que o tamanho informado, espaços serão utilizados para completar as posições que faltam. Se o parâmetro for maior que o tamanho especificado, ele será impresso em sua totalidade, ocupando mais espaço que o inicialmente especificado.

► Listagem 7.30 – Limitação do tamanho de impressão dos parâmetros

```
>>> "{0:10} {1}".format("123", "456")
'123        456'
>>> "X{0:10}X".format("123")
'X123      X'
>>> "X{0:10}X".format("123456789012345")
'X123456789012345X'
```

Podemos também especificar se queremos os espaços adicionais à esquerda ou à direita do valor, utilizando os símbolos de maior (>) e menor (<) logo depois dos dois pontos (Listagem 7.31).

► Listagem 7.31 – Especificação de espaços à esquerda ou à direita

```
>>> "X{0:<10}X".format("123")
'X123      X'
>>> "X{0:>10}X".format("123")
'X      123X'
```

Se quisermos o valor entre os espaços, de forma a centralizá-lo, podemos utilizar o circunflexo (^), como mostra a listagem 7.32.

► Listagem 7.32 – Centralização

```
>>> "X{0:^10}X".format("123")
'X 123      X'
```

Se quisermos outro caractere no lugar de espaços, podemos especificá-lo logo após os dois pontos (Listagem 7.33).

► Listagem 7.33 – Especificação de espaços à esquerda ou à direita

```
>>> "X{0:<10}X".format("123")
'X123.....X'
>>> "X{0:!>10}X".format("123")
'X!!!!!!123X'
>>> "X{0:^*10}X".format("123")
'X***123****X'
```

Se o parâmetro for uma lista, podemos especificar o índice do elemento a substituir, dentro da máscara (Listagem 7.34).

► Listagem 7.34 – Máscaras com elementos de uma lista

```
>>> "{0[1]} {0[2]}".format(["123", "456"])
'123 456'
```

O mesmo é válido para dicionários (Listagem 7.35).

► Listagem 7.35 – Máscaras com elementos de um dicionário

```
>>> "{0[nome]} {0[telefone]}".format({ "telefone": 572, "nome": "Maria"})
'Maria 572'
```

Observe que dentro da string escrevemos nome e telefone entre colchetes, mas sem aspas, como normalmente faríamos ao utilizar dicionários. Essa sintaxe é especial para o método `format`.

7.9.1 Formatação de números

A nova sintaxe também permite a formatação de números. Por exemplo, se especificarmos o tamanho a imprimir com um zero à esquerda, o valor será impresso com a largura determinada e com zeros à esquerda completando o tamanho (Listagem 7.36).

► Listagem 7.36 – Zeros à esquerda

```
>>> "{0:05}".format(5)
'00005'
```

Podemos também utilizar outro caractere, diferente de 0, mas, nesse caso, devemos escrever o caractere à esquerda do símbolo de igualdade (Listagem 7.37).

► Listagem 7.37 – Preenchimento com outros caracteres

```
>>> "{0:*=7}".format(32)
'*****32'
```

Podemos também especificar o alinhamento dos números que estamos imprimindo usando <, > e ^ (Listagem 7.38).

► Listagem 7.38 – Combinação de vários códigos de formatação

```
>>> "{0:^^10}".format(123)
'***123***'
>>> "{0:<10}".format(123)
'123*****'
>>> "{0:>10}".format(123)
'*****123'
```

Podemos também utilizar uma vírgula para solicitar o agrupamento por milhar (Listagem 7.39), e o ponto para indicar a precisão de números decimais, ou melhor, a quantidade de casas após a vírgula.

► Listagem 7.39 – Separação de milhares

```
>>> "{0:10,}".format(7532)
'    7,532'
>>> "{0:10.5f}".format(1500.31)
'1500.31000'
>>> "{0:10,.5f}".format(1500.31)
'1,500.31000'
```

A nova sintaxe também permite forçar a impressão de sinais ou apenas reservar espaço para uma impressão eventual (Listagem 7.40).

► Listagem 7.40 – Impressão de sinais de positivo e negativo

```
>>> "{0:+10} {1:-10}".format(5,-6)
'      +5      -6'
>>> "{0:-10} {1: 10}".format(5,-6)
'      5      -6'
>>> "{0: 10} {1:+10}".format(5,-6)
'      5      -6'
```

Quando trabalhamos com formatos numéricos, devemos indicar com uma letra o formato que deve ser adotado para a impressão. Essa letra informa como devemos exibir um número. A lista completa de formatos numéricos é apresentada nas tabelas 7.1 e 7.2.

Tabela 7.1 – Formatos de números inteiros

Código	Descrição	Exemplo (45)
b	Binário	101101
c	Caractere	-
d	Base 10	45
n	Base 10 local	45
o	Octal	55
x	Hexadecimal com letras minúsculas	2d
X	Hexadecimal com letras maiúsculas	2D

O formato `b` imprime o número utilizando o sistema binário, ou seja, de base 2, com apenas 0 e 1 como dígitos. O formato `o` imprime o número utilizando o sistema octal, ou seja, de base 8, com dígitos de 0 a 7. Já o formato `c` imprime o número convertendo-o em caractere, utilizando a tabela Unicode. Tanto o formato `x` quanto o `X` imprimem os números utilizando o sistema hexadecimal, base 16. A diferença é que `x` utiliza letras minúsculas, e `X`, maiúsculas. Vejamos outros exemplos na listagem 7.41.

► Listagem 7.41 – Formatação de inteiros

```
>>> "{:b}".format(5678)
'1011000101110'
>>> "{:c}".format(65)
'A'
>>> "{:o}".format(5678)
'13056'
>>> "{:x}".format(5678)
'162e'
>>> "{:X}".format(5678)
'162E'
```

Os formatos `d` e `n` são parecidos. O formato `d` é semelhante ao que utilizamos ao formatar os números com `%d`. A diferença entre o formato `d` e `n` é que o `n` leva em consideração as configurações regionais da máquina do usuário. Vejamos alguns exemplos na listagem 7.42. Observe que, antes de configurarmos a máquina para o português do Brasil, o resultado de `d` e `n` eram iguais. Após a configuração regional, o formato `n` passou a exibir os números utilizando pontos para separar os milhares. Se você utiliza Windows, modifique "pt_BR.utf-8" para "Portuguese_Brazil" nas listagens 7.42 e 7.43.

Listagem 7.42 – O formato `d` e o formato `n`

```
>>> "{:d}".format(5678)
'5678'
>>> "{:n}".format(5678)
'5678'
>>> import locale
>>> locale.setlocale(locale.LC_ALL,"pt_BR.utf-8")
'pt_BR.utf-8'
>>> "{:n}".format(5678)
'5.678'
```

Tabela 7.2 – Formatos de números decimais

Código	Descrição	Exemplo (1.345)
e	Notação científica com e minúsculo	1.345000e+00
E	Notação científica com e maiúsculo	1.345000E+00
f	Decimal	1.345000
g	Genérico	- 1.345
G	Genérico	1.345
n	Local	1,345
%	Percentual	134.500000%

Para números decimais, temos também vários códigos. O código `f` já conhecemos e funciona de forma semelhante ao que utilizamos em `%f`. O formato `n` utiliza as configurações regionais para imprimir o número. Em português, essa configuração utiliza o ponto como separador de milhar e a vírgula como separador decimal, produzindo números mais fáceis de entender. Vejamos exemplos na listagem 7.43.

► Listagem 7.43 – Formatação de números decimais

```
>>> "{:f}".format(1579.543)
'1579.543000'
>>> "{:n}".format(1579.543)
'1579.54'
>>> import locale
>>> locale.setlocale(locale.LC_ALL,"pt_BR.utf-8")
'pt_BR.utf-8'
>>> "{:n}".format(1579.543)
'1.579,54'
```

Os formatos `e` e `E` imprimem o número utilizando notação científica. Isso quer dizer que a parte decimal vai ser substituída por um expoente. Por exemplo: 1004.5 em notação científica será impresso `1.004500e+03`. Para entender esse formato, devemos entender suas partes. O número à esquerda de `e` é o que chamamos de mantissa, e o número à direita, o expoente. A base é sempre 10 e, para recompor o número, multiplicamos a mantissa pela base 10 elevada ao expoente. No caso de `1.004500e+03`, teríamos:

$$1.004500 \times 10^3 = 1.004500 \times 1000 = 1004.5$$

A vantagem de utilizar notação científica é poder representar números muito grandes, ou muito pequenos, em pouco espaço. A diferença entre o formato `e` e `E` é que queremos exibir o `e` que separa a mantissa do expoente, ou seja, um `e` minúsculo ou um `E` maiúsculo. Os formatos `g` e `G` são chamados de genéricos, pois, dependendo do número, eles são exibidos como no formato `f` ou como no `e` ou `E`. O tipo `%` simplesmente multiplica o valor por 100 antes de imprimi-lo, assim, `0,05` é impresso como `5%`. Veja alguns exemplos na listagem 7.44.

► Listagem 7.44 – Formatação de números decimais

```
>>> "{:8e}".format(3.141592653589793)
'3.141593e+00'
>>> "{:8E}".format(3.141592653589793)
'3.141593E+00'
>>> "{:8g}".format(3.141592653589793)
' 3.14159'
>>> "{:8G}".format(3.141592653589793)
' 3.14159'
>>> "{:8g}".format(3.14)
'   3.14'
>>> "{:8G}".format(3.14)
'   3.14'
>>> "{:5.2%}".format(0.05)
'5.00%
```

7.10 Jogo da forca

Vejamos um jogo muito simples, mas que ajuda a trabalhar com strings. O jogo da forca é simples e pode até se tornar divertido.

► Listagem 7.45 – Jogo da forca

```
palavra = input("Digite a palavra secreta:").lower().strip() ❶
for x in range(100):
    print() ❷
digitadas = []
acertos = []
erros = 0
while True:
    senha="" ❸
    for letra in palavra:
        senha +=letra if letra in acertos else "."
    print(senha)
    if senha == palavra:
        print("Você acertou!")
        break
    tentativa = input("\nDigite uma letra:").lower().strip()
    if tentativa in digitadas:
        print("Você já tentou esta letra!")
        continue ❹
    else:
        digitadas += tentativa
        if tentativa in palavra:
            acertos += tentativa
        else:
            erros += 1
            print("Você errou!")
    print("X====\nX : ")
    print("X 0 " if erros >= 1 else "X")
    linha2=""
    if erros == 2:
        linha2 = " | "
    elif erros == 3:
        linha2 = " \| "
    elif erros >= 4:
        linha2 = " \|/ "
```

```

print("X%s" % linha2)
linha3=""
if erros == 5:
    linha3+=" / "
elif erros>=6:
    linha3+=" / \ "
print("X%s" % linha3)
print("X\n=====")
if erros == 6:
    print("Enforcado!")
    break

```

Veja como é simples escrever um jogo da forca em Python. Vamos analisar o que essa listagem faz. Em ①, aproveitamos o retorno de `input` para chamar os métodos de string `lower` e `strip`. Observe como escrevemos uma chamada após a outra. Isso é possível porque `input`, `lower` e `strip` retornam um objeto `string`. O resultado final da string digitada pelo usuário, convertida para letras minúsculas e com espaços em branco no início e no fim eliminados, é atribuído à variável `palavra`. É essa variável que vai armazenar a palavra a ser adivinhada pelo jogador.

Em ② pulamos várias linhas para que o jogador não veja o que foi digitado como `palavra`. A ideia é que um jogador escreva uma palavra secreta, e que outro tente descobri-la.

Em ③, temos uma nova forma de condição que facilita decisões simples. O `if` imediato, ou na mesma linha, serve para decidir o valor a retornar, dependendo de uma condição. É como o `if` que já conhecemos, mas o valor verdadeiro fica à esquerda do `if`, e a condição, à sua direita. O `else` não tem `:` e indica o resultado caso a condição seja falsa. No exemplo, letra é o valor a retornar se a condição `letra in acertos` for verdadeira, e `"."` é o valor se o resultado for falso. Assim:

```
senha += letra if letra in acertos else "."
```

substitui:

```

if letra in acertos:
    senha += letra
else:
    senha += "."

```

A vantagem dessa construção é que escrevemos tudo em uma só linha. Esse recurso é interessante para expressões simples e não deve ser utilizado para todo e qualquer caso. No jogo da forca, usamos essa construção para mostrar a palavra secreta na tela, mas substituindo todas as letras ainda não adivinhadas por um `"."`.

Em ④, utilizamos a instrução `continue`. A instrução `continue` é similar ao `break` que já conhecemos, mas serve para indicar que devemos ignorar todas as linhas até o fim da repetição e voltar para o início, sem terminá-la. No jogo da forca, isso faz com que a execução passe de ④ para o `while` diretamente, pulando todas as linhas depois dela. Quando utilizada com `while`, `continue` causa a reavaliação da condição da repetição; quando utilizada com `for`, faz com que o próximo elemento seja utilizado. Considere `continue` como vá para o fim da repetição e volte para a linha de `for` ou `while`.

Exercício 7.7 Modifique o programa de forma a escrever a palavra secreta caso o jogador perca.

Exercício 7.8 Modifique o jogo da forca de forma a utilizar uma lista de palavras. No início, pergunte um número e calcule o índice da palavra a utilizar pela fórmula: `índice = (número * 776) % len(lista_de_palavras)`.

Exercício 7.9 Modifique o programa para utilizar listas de strings para desenhar o boneco da forca. Você pode utilizar uma lista para cada linha e organizá-las em uma lista de listas. Em vez de controlar quando imprimir cada parte, desenhe nessas listas, substituindo o elemento a desenhar.

Exemplo:

```

>>> linha = list("X-----")
>>> linha
['X', '-', '-', '-', '-', '-', '-']
>>> linha[6] = "|"
>>> linha
['X', '-', '-', '-', '-', '-', '|']
>>> "".join(linha)
'X-----|'

```

Exercício 7.10 Escreva um jogo da velha para dois jogadores. O jogo deve perguntar onde você quer jogar e alternar entre os jogadores. A cada jogada, verifique se a posição está livre. Verifique também quando um jogador venceu a partida. Um jogo da velha pode ser visto como uma lista de 3 elementos, onde cada elemento é outra lista, também com três elementos.

Exemplo do jogo:

```
x | o |
-----+
| x | x |
-----+
|   | o
```

Onde cada posição pode ser vista como um número. Confira abaixo um exemplo das posições mapeadas para a mesma posição de seu teclado numérico.

```
7 | 8 | 9
-----+
4 | 5 | 6
-----+
1 | 2 | 3
```

CAPÍTULO 8

Funções

Podemos definir nossas próprias funções em Python. Sabemos como usar várias funções, como `len`, `int`, `float`, `print` e `input`. Neste capítulo, veremos como declarar novas funções e utilizá-las em programas.

Para definir uma nova função, utilizaremos a instrução `def`. Vejamos como declarar uma função de soma que recebe dois números como parâmetros e os imprime na tela (Listagem 8.1).

► Listagem 8.1 – Definição de uma nova função

```
def soma(a,b): ❶
    print(a+b) ❷
soma(2,9) ❸
soma(7,8)
soma(10,15)
```

Observe em ❶ que usamos a instrução `def` seguida pelo nome da função, no caso, `soma`. Após o nome e entre parênteses, especificamos o nome dos parâmetros que a função receberá. Chamamos o primeiro de `a` e o segundo de `b`. Observe também que usamos : após os parâmetros para indicar o início de um bloco.

Em ❷, usamos a função `print` para exibir `a+b`. Observe que escrevemos ❷ dentro do bloco da função, ou seja, mais à direita.

Diferentemente do que já vimos até agora, essas linhas não serão executadas imediatamente, exceto a definição da função em si. Na realidade, a definição prepara o interpretador para executar a função quando esta for chamada em outras partes do programa. Para chamar uma função definida no programa, faremos da mesma forma que as funções já definidas na linguagem, ou seja, nome da função seguido