

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

dos parâmetros entre parênteses. Exemplos de como chamar a função `soma` são apresentados a partir de ❶. No primeiro exemplo, chamamos `soma(2,9)`. Nesse caso, a função será chamada com `a` valendo 2, e `b` valendo 9. Os parâmetros são substituídos na mesma ordem em que foram definidos, ou seja, o primeiro valor como `a` e o segundo como `b`.

Funções são especialmente interessantes para isolar uma tarefa específica em um trecho de programa. Isso permite que a solução de um problema seja reutilizada em outras partes do programa, sem precisar repetir as mesmas linhas. O exemplo anterior utiliza dois parâmetros e imprime sua soma. Essa função não retorna valores como a função `len` ou a `int`. Vamos reescrever essa função de forma que o valor da soma seja retornado (Listagem 8.2).

► Listagem 8.2 – Definição do retorno de um valor

```
def soma(a,b):
    return(a+b) ❶
print(soma(2,9))
```

Veja que agora utilizamos a instrução `return` ❶ para indicar o valor a retornar. Observe também que retiramos o `print` da função. Isso é interessante porque a soma e a impressão da soma de dois números são dois problemas diferentes. Nem sempre vamos somar e imprimir a soma, por isso, vamos deixar a função realizar apenas o cálculo. Se precisarmos imprimir o resultado, podemos utilizar a função `print`, como no exemplo.

Vejamos outro exemplo, uma função que retorne verdadeiro ou falso, dependendo se o número é par ou ímpar (Listagem 8.3).

► Listagem 8.3 – Retornando se valor é par ou não

```
def épar(x):
    return(x%2==0)
print(épar(2))
print(épar(3))
print(épar(10))
```

Imagine agora que precisamos definir uma função para retornar a palavra `par` ou `ímpar`. Podemos reutilizar `épar` em outra função (Listagem 8.4).

► Listagem 8.4 – Reutilização da função `épar` em outra função

```
def épar(x):
    return(x%2==0)
def par_ou_impar(x):
    if épar(x): ❶
        return "par" ❷
    else:
        return "ímpar" ❸
print(par_ou_impar(4))
print(par_ou_impar(5))
```

Em ❶ chamamos a função `épar` dentro da função `par_ou_impar`. Observe que não há nada de especial nessa chamada: apenas repassamos o valor de `x` para a função `épar` e utilizamos seu retorno no `if`. Se a função retornar verdadeiro, retornaremos “par” em ❷; caso contrário, “ímpar” em ❸. Utilizamos dois `return` na função `par_ou_impar`. A instrução `return` faz com que a função pare de executar e que o valor seja retornado imediatamente ao programa ou função que a chamou. Assim, podemos entender a instrução `return` como uma interrupção da execução da função, seguido do retorno do valor. As linhas da função após a instrução `return` são ignoradas de forma similar à instrução `break` dentro de um `while` ou `for`.

Exercício 8.1 Escreva uma função que retorne o maior de dois números.

Valores esperados:

`máximo(5,6) == 6`
`máximo(2,1) == 2`
`máximo(7,7) == 7`

Exercício 8.2 Escreva uma função que receba dois números e retorne `True` se o primeiro número for múltiplo do segundo.

Valores esperados:

`múltiplo(8,4) == True`
`múltiplo(7,3) == False`
`múltiplo(5,5) == True`

Exercício 8.3 Escreva uma função que receba o lado (l) de um quadro e retorne sua área ($A = \text{lado}^2$).

Valores esperados:

`área_quadrado(4) == 16`

`área_quadrado(9) == 81`

Exercício 8.4 Escreva uma função que receba a base e a altura de um triângulo e retorne sua área ($A = (\text{base} \times \text{altura})/2$).

Valores esperados:

`área_triangulo(6, 9) == 27`

`área_triangulo(5, 8) == 20`

Vejamos agora um exemplo de função de pesquisa em uma lista (Listagem 8.5).

► Listagem 8.5 – Pesquisa em uma lista

```
def pesquise(lista, valor):
    for x,e in enumerate(lista):
        if e == valor:
            return x ①
    return None ②
L=[10, 20, 25, 30]
print(pesquise(L, 25))
print(pesquise(L, 27))
```

A função `pesquise` recebe dois parâmetros: a lista e o valor a pesquisar. Se o valor for encontrado, retornaremos o valor de sua posição em ①. Caso não seja encontrado, retornaremos `None` em ②. Observe que, se retornarmos em ①, tanto `for` quanto o `return` em ② são completamente ignorados. Dizemos que `return` marca o fim da execução da função. Usando estruturas de repetição e condicionais, podemos programar onde e como as funções retornarão, assim como decidir o valor a ser retornado.

Vejamos outro exemplo, calculando a média dos valores de uma lista (Listagem 8.6).

► Listagem 8.6 – Cálculo da média de uma lista

```
def soma(L):
    total=0
    for e in L:
        total+=e
    return total
def media(L):
    return(soma(L)/len(L))
```

Para calcular a média, definimos outra função chamada `soma`. Dessa forma, temos duas funções, uma para calcular a média e outra para calcular a soma da lista. Definir as funções dessa forma é mais interessante, pois uma função deve resolver apenas um problema. Poderíamos também ter definido a função média como na listagem 8.7.

► Listagem 8.7 – Soma e cálculo da média de uma lista

```
def média(L):
    total=0
    for e in L:
        total+=e
    return total/len(L)
```

Qual das duas formas escolher depende do tipo de problema que se quer resolver. Se você não precisa do valor da soma dos elementos de uma lista em nenhuma parte do programa, a segunda forma é interessante. A primeira (`soma` e `média` definidas em duas funções separadas) é mais interessante em longo prazo e é também uma boa prática de programação. Conforme formos criando funções, podemos armazená-las para uso em outros programas. Com o tempo, você vai colecionar essas funções, criando uma biblioteca ou módulo. Veremos mais sobre isso quando falarmos de importação e módulos. Por enquanto, tente fixar duas regras: uma função deve resolver apenas um problema e, quanto mais genérica for sua solução, melhor ela será em longo prazo.

Para saber se sua função resolve apenas um problema, tente defini-la sem utilizar a conjunção “e”. Se ela faz isso e aquilo, já é um sinal que efetua mais de uma tarefa e que talvez tenha que ser desmembrada em outras funções. Não se preocupe agora em definir funções perfeitas, pois à medida que você for ganhando experiência na programação esses conceitos se tornarão mais claros.

Resolver os problemas da maneira mais genérica possível é se preparar para reutilizar a função em outros programas. O fato de a função só poder ser utilizada no mesmo programa que a definiu não é um grande problema, pois à medida que os programas se tornam mais complexos eles exigem soluções próprias e detalhadas. No entanto, se todas as suas funções servirem apenas a um programa, considere isso como um sinal de alerta.

Vejamos o que não fazer na listagem 8.8.

► Listagem 8.8 – Como não escrever uma função

```
def soma(L):
    total=0
    x = 0
    while x<5:
        total+=L[x]
        x+=1
    return total

L=[1,7,2,9,15]
print(soma(L)) ❶
print(soma([7,9,12,3,100,20,4])) ❷
```

Você saberia dizer por que a função funcionou em ❶, mas retornou um valor incorreto em ❷?

A forma como definimos a função soma não é genérica, ou melhor, só funciona em casos onde devemos somar listas com cinco elementos. É para esse tipo de erro que você deve ficar atento. Se a função deve somar todos os elementos de qualquer lista passada como parâmetro, devemos ao menos presumir que podemos passar listas com tamanhos diferentes. Explicaremos mais sobre isso quando falarmos de testes.

Um problema clássico de programação é o cálculo do fatorial. O fatorial de um número é utilizado em estatística para calcular o número de combinações e permutações de conjuntos. Seu cálculo é simples, por isso, é muito utilizado como exemplo em cursos de programação. Para calcular o fatorial, multiplicamos o número por todos os números que o precedem até chegarmos em 1.

Por exemplo:

- fatorial de 3: $3 \times 2 \times 1 = 6$.
- fatorial de 4: $4 \times 3 \times 2 \times 1 = 24$

E assim por diante. Um caso especial é o fatorial de 0 que é definido como 1. Vejamos uma função que calcule o fatorial de um número na listagem 8.9.

► Listagem 8.9 – Cálculo do fatorial

```
def fatorial(n):
    fat = 1
    while n>1:
        fat*=n
        n-=1
    return fat
```

Se você rastrear essa função, verá que calculamos o fatorial multiplicando o valor de n pelo valor anterior (fat) e que começamos do maior número até chegarmos em 1. Observe que a forma que definimos a função satisfaz o caso especial do fatorial de zero.

Poderíamos também ter definida a função como na listagem 8.10.

► Listagem 8.10 – Outra forma de calcular o fatorial

```
def fatorial(n):
    fat=1
    x=1
    while x<=n:
        fat*=x
        x+=1
    return fat
```

Existem várias formas de resolver o problema, e todas podem estar corretas. Para decidir se nossa implementação está correta, temos que observar os valores retornados e compará-los aos valores de referência.

Exercício 8.5 Reescreva a função da listagem 8.5 de forma a utilizar os métodos de pesquisa em lista, vistos no capítulo 7.

Exercício 8.6 Reescreva o programa da listagem 8.8 de forma a utilizar `for` em vez de `while`.

Dica: Python tem funções para calcular a soma, o máximo e o mínimo de uma lista.

```
>>> L=[5,7,8]
>>> sum(L)
20
>>> max(L)
8
>>> min(L)
5
```

8.1 Variáveis locais e globais

Quando usamos funções, começamos a trabalhar com variáveis internas ou locais e com variáveis externas ou globais. A diferença entre elas é a visibilidade ou escopo.

Uma variável local a uma função existe apenas dentro dela, sendo normalmente inicializada a cada chamada. Assim, não podemos acessar o valor de uma variável local fora da função que a criou e, por isso, passamos parâmetros e retornamos valores nas funções, de forma a possibilitar a troca de dados no programa.

Uma variável global é definida fora de uma função, pode ser vista por todas as funções do módulo e por todos os módulos que importam o módulo que a definiu.

Vejamos um exemplo dos dois casos:

```
EMPRESA="Unidos Venceremos Ltda"
def imprime_cabeçalho():
    print(EMPRESA)
    print("-" * len(EMPRESA))
```

A função `imprime_cabeçalho` não recebe parâmetros nem retorna valores. Ela simplesmente imprime o nome da empresa e traços abaixo dele. Observe que utilizamos a variável `EMPRESA` definida fora da função. Nesse caso, `EMPRESA` é uma variável global, podendo ser acessada em qualquer função.

Variáveis globais devem ser utilizadas o mínimo possível em seus programas, pois dificultam a leitura e violam o encapsulamento da função. A dificuldade da leitura é em procurar pela definição e conteúdos fora da função em si, que podem mudar entre diferentes chamadas. Além disso, uma variável global pode ser alterada por qualquer função, tornando a tarefa de saber quem altera seu valor realmente mais trabalhosa. O encapsulamento é comprometido porque a função

depende de uma variável externa, ou seja, que não é declarada dentro da função nem recebida como parâmetro.

Embora devamos utilizar variáveis globais com cuidado, isso não significa que elas não tenham uso ou que possam simplesmente ser classificadas como má prática.

Um bom uso de variáveis globais é guardar valores constantes e que devem ser acessíveis a todas as funções do programa, como o nome da empresa.

Elas também são utilizadas em nosso programa principal e para inicializar o módulo com valores iniciais. Tente utilizar variáveis globais apenas para configuração e com valores constantes. Por exemplo, se na função `imprime_cabeçalho` o nome da empresa mudasse entre uma chamada e outra, ele não deveria ser armazenado em uma variável global, mas passado por parâmetro.

Devido à capacidade da linguagem Python de declarar variáveis à medida que precisarmos, devemos tomar cuidado quando alteramos uma variável global dentro de uma função. Vejamos um exemplo:

```
a=5 ❶
def muda_e_imprime():
    a=7 ❷
    print("A dentro da função: %d" % a)
    print("a antes de mudar: %d" % a) ❸
    muda_e_imprime() ❹
    print("a depois de mudar: %d" % a) ❺
```

Execute o programa e analise seu resultado. Você sabe por que o valor de `a` não mudou? Tente responder a essa questão antes de continuar lendo.

Em ❶, criamos a variável global `a`. Em ❷, temos a variável local da função, também chamada `a`, recebendo 7. Em ❸ e ❺, imprimimos o valor da variável global `a` e, em ❹, o valor da variável local `a`. Para o computador, essas variáveis são completamente diferentes, embora tenham o mesmo nome. Em ❶, podemos acessar seu conteúdo porque realizamos a impressão dentro da função. Da mesma forma, a variável `a` que imprimimos é a variável local, valendo 7. Essa variável local deixa de existir no final da função e explica porque não alteramos a variável global `a`.

Se quisermos modificar uma variável global dentro de uma função, devemos informar que estamos usando uma variável global antes de inicializá-la, na primeira linha de nossa função. Essa definição é feita com a instrução `global`. Vejamos o exemplo modificado:

```
a=5

def muda_e_imprime():
    global a ❶
    a=7 ❷
    print("A dentro da função: %d" % a)
    print("a antes de mudar: %d" % a)
    muda_e_imprime()
    print("a depois de mudar: %d" % a)
```

Agora, em **❶**, estamos trabalhando com a variável `a` global. Assim, quando fizemos `a=7` em **❷**, trocamos o valor de `a`.

Lembre-se de limitar o uso de variáveis globais.

A utilização da instrução `global` pode sinalizar um problema de construção no programa. Lembre-se de utilizar variáveis globais apenas para configuração e, de preferência, como constantes. Dessa forma, você utilizará `global` apenas nas funções que alteram a configuração de seu programa ou módulo. A instrução `global` facilita a escolha de nomes de variáveis locais nos programas, pois deixa claro que estamos usando uma variável preeexistente e não criando uma nova.

Tente definir variáveis globais de forma a facilitar a leitura de seu programa, como usar apenas letras maiúsculas em seus nomes ou um prefixo, por exemplo, `EMPRESA` ou `G_EMPRESA`.

8.2 Funções recursivas

Uma função pode chamar a si mesma. Quando isso ocorre temos uma função recursiva. O problema do fatorial é interessante para demonstrar o conceito de recursividade. Podemos definir o fatorial de um número como sendo esse número multiplicado pelo fatorial de seu antecessor. Se chamarmos nosso número de `n`, temos uma definição recursiva do fatorial como:

$$\text{fatorial}(n) = \begin{cases} 1 & 0 \leq n \leq 1 \\ n \times \text{fatorial}(n - 1) & \text{outro caso} \end{cases}$$

Em Python, definiríamos a função recursiva para cálculo do fatorial como o programa da listagem 8.11.

► Listagem 8.11 – Função recursiva do fatorial

```
def fatorial(n):
    if n==0 or n == 1:
        return 1
    else:
        return n*fatorial(n-1)
```

Vamos adicionar algumas linhas para facilitar o rastreamento, alterando o programa para imprimir na entrada da função (Listagem 8.12).

► Listagem 8.12 – Função modificada para facilitar o rastreamento

```
def fatorial(n):
    print("Calculando o fatorial de %d" % n)
    if n==0 or n == 1:
        print("Fatorial de %d = 1" % n)
        return 1
    else:
        fat = n*fatorial(n-1)
        print(" fatorial de %d = %d" % (n, fat) )
        return fat
fatorial(4)
```

Que produz como resultado a tela da listagem 8.13.

► Listagem 8.13 – Cálculo do fatorial de 4

```
Calculando o fatorial de 4
Calculando o fatorial de 3
Calculando o fatorial de 2
Calculando o fatorial de 1
Fatorial de 1 = 1
fatorial de 2 = 2
fatorial de 3 = 6
fatorial de 4 = 24
```

A sequência de Fibonacci é outro problema clássico no qual podemos aplicar funções recursivas. A sequência pode ser entendida como o número de casais de coelhos que teríamos após cada ciclo de reprodução, considerando que cada ciclo

dá origem a um casal. Embora matematicamente a sequência de Fibonacci revele propriedades mais complexas, limitaremos-nos à história dos casais de coelhos. A sequência começa com dois números 0 e 1. Os números seguintes são a soma dos dois anteriores. A sequência ficaria assim: 0, 1, 1, 2, 3, 5, 8, 13, 21, ... Assim, uma função para calcular o enésimo termo da sequência de Fibonacci pode ser definida como:

$$\text{fibonacci}(n) = \begin{cases} n & n \leq 1 \\ \text{fibonacci}(n - 1) + \text{fibonacci}(n - 2) & \text{outro caso} \end{cases}$$

Sendo implementada em Python como apresentado na listagem 8.14.

► **Listagem 8.14 – Função recursiva de Fibonacci**

```
def fibonacci(n):
    if n<=1:
        return n
    else:
        return fibonacci(n-1)+fibonacci(n-2)
```

Exercício 8.7 Defina uma função recursiva que calcule o maior divisor comum (M.D.C.) entre dois números a e b , onde $a > b$.

$$\text{mdc}(a, b) = \begin{cases} a & b = 0 \\ \text{mdc}(b, a - b \lfloor \frac{a}{b} \rfloor) & a > b \end{cases}$$

Onde $a - b \lfloor \frac{a}{b} \rfloor$ pode ser escrito em Python como: $a \% b$.

Exercício 8.8 Usando a função mdc definida no exercício anterior, defina uma função para calcular o menor múltiplo comum (M.M.C.) entre dois números.

$$\text{mmc}(a, b) = \frac{|a \times b|}{\text{mdc}(a, b)}$$

Onde $|a \times b|$ pode ser escrito em Python como: $\text{abs}(a*b)$.

Exercício 8.9 Rastreie o programa da listagem 8.12 e compare o resultado com o apresentado na listagem 8.13.

Exercício 8.10 Reescreva a função para cálculo da sequência de Fibonacci, sem utilizar recursão.

8.3 Validação

Funções são muito úteis para validar a entrada de dados. Vejamos o código para ler um valor inteiro, limitado por um valor de mínimo e máximo (Listagem 8.15). Esse trecho repete a entrada de dados até termos um valor válido.

► **Listagem 8.15 – Exemplo de validação sem usar uma função**

```
while True:
    v=int(input("Digite um valor entre 0 e 5:"))
    if v<0 or v>5:
        print("Valor inválido.")
    else:
        break
```

Podemos transformá-lo em uma função que receba a pergunta e os valores de máximo e mínimo (Listagem 8.16).

► **Listagem 8.16 – Validação de inteiro usando função**

```
def faixa_int(pergunta, minímo, máximo):
    while True:
        v=int(input(pergunta))
        if v<minímo or v>máximo:
            print("Valor inválido. Digite um valor entre %d e %d" % (minímo,máximo))
        else:
            return v
```

Esse tipo de verificação é muito importante quando nosso programa só funciona com uma faixa de valores. Quando verificamos os dados do programa, estamos

realizando uma validação. A validação é muito importante para evitarmos erros difíceis de detectar depois de termos escrito o programa. Sempre que seu programa receber dados, seja pelo teclado ou por um arquivo, verifique se esses dados estão na faixa e no formato adequado para o bom funcionamento.

Exercício 8.11 Escreva uma função para validar uma variável string. Essa função recebe como parâmetro a string, o número mínimo e máximo de caracteres. Retorne verdadeiro se o tamanho da string estiver entre os valores de máximo e mínimo, e falso em caso contrário.

Exercício 8.12 Escreva uma função que receba uma string e uma lista. A função deve comparar a string passada com os elementos da lista, também passada como parâmetro. Retorne verdadeiro se a string for encontrada dentro da lista, e falso em caso contrário.

8.4 Parâmetros opcionais

Nem sempre precisaremos passar todos os parâmetros para uma função, preferindo utilizar um valor previamente escolhido como padrão, mas deixando a possibilidade de alterá-lo, caso necessário. Vejamos uma função que imprime uma barra na tela na listagem 8.17.

► Listagem 8.17 – Função para imprimir uma barra na tela

```
def barra():
    print(" " * 40)
```

Nesse exemplo, a função `barra` não recebe algum parâmetro e pode ser chamada com `barra()`. Porém, tanto o asterisco (*) quanto a quantidade de caracteres a exibir na barra podem precisar ser alterados. Por exemplo, podemos utilizar uma barra de asteriscos em uma parte de nosso programa e uma barra com pontos em outra. Para resolver esse problema, podemos utilizar parâmetros opcionais.

► Listagem 8.18 – Função para imprimir uma barra na tela com parâmetros opcionais

```
def barra(n=40, caractere=" "):
    print(caractere * n)
```

Essa nova definição (Listagem 8.18) permite utilizar a função `barra` como antes, ou seja, `barra()`, onde caractere será igual a " " e n será igual a 40. Ao usarmos parâmetros opcionais, estamos especificando que valores devem ser utilizados caso um novo valor não seja especificado, mas ainda assim deixando a possibilidade de passarmos outro valor. Veja os exemplos da listagem 8.19.

► Listagem 8.19 – Passagem de parâmetros opcionais

```
>>> barra(10) # faz com que n seja 10
*****
>>> barra(10, "-") # n = 10 e caractere="-"
-----
```

Parâmetros opcionais são úteis para evitar a passagem desnecessária dos mesmos valores, mas preservando a opção de passar valores, se necessário.

Podemos combinar parâmetros opcionais com obrigatórios na mesma função. Vejamos a função `soma` na listagem 8.20.

► Listagem 8.20 – Função soma com parâmetros obrigatórios e opcionais

```
def soma(a, b, imprime=False):
    s = a + b
    if imprime:
        print(s)
    return s
```

No exemplo, `a` e `b` são parâmetros obrigatórios e `imprime` é um parâmetro opcional. Essa função pode ser utilizada como mostra a listagem 8.21.

► Listagem 8.21 – Uso da função soma com parâmetros obrigatórios e opcionais

```
>>> soma(2,3)
5
>>> soma(3,4, True)
7
7
>>> soma(5,8, False)
13
```

Embora possamos combinar parâmetros opcionais e obrigatórios, eles não podem ser misturados entre si, e os parâmetros opcionais devem sempre ser os últimos. Vejamos uma definição inválida na listagem 8.22.

► **Listagem 8.22 – Definição inválida da função soma com parâmetros opcionais antes dos obrigatórios**

```
def soma(imprime=True, a, b):
    s = a + b
    if imprime:
        print(s)
    return s
```

Essa definição é inválida porque o parâmetro opcional `imprime` é seguido por parâmetros obrigatórios `a` e `b`.

8.5 Nomeando parâmetros

Python suporta a chamada de funções com vários parâmetros, mas até agora vimos apenas o caso em que fizemos a chamada da função passando os parâmetros na mesma ordem em que foram definidos. Quando especificamos o nome dos parâmetros, podemos passá-los em qualquer ordem. Vejamos o exemplo da função `retângulo`, definida na listagem 8.23.

► **Listagem 8.23 – Função retângulo com parâmetros obrigatórios e opcionais**

```
def retângulo(largura, altura, caractere="*"):
    linha = caractere * largura
    for i in range(altura):
        print(linha)
```

A função `retângulo` pode ser chamada como exemplifica a listagem 8.24.

► **Listagem 8.24 – Chamando a função retângulo nomeando os argumentos**

```
>>> retângulo(3,4)
***
***
```

```
>>> retângulo(largura=3, altura=4)
***
***
```

...

```
>>> retângulo(altura=4, largura=3)
***
***
```

...

```
>>> retângulo(caractere="-", altura=4, largura=3)
***
```

...

```
>>> retângulo(largura=3, caractere="*", altura=4)
***
```

...

```
>>> retângulo(caractere="*", altura=4, largura=3)
***
```

...

Uma vez que utilizamos o nome de cada parâmetro para fazer a chamada, a ordem de passagem deixa de ser importante. Observe também que o parâmetro `caractere` continua opcional, mas se o especificarmos, devemos também fazê-lo com nome. Quando especificamos o nome de um parâmetro, somos obrigados a especificar o nome de todos os outros parâmetros também. Por exemplo, as chamadas abaixo são inválidas:

► **Listagem 8.25 – Chamadas inválidas da função retângulo**

```
retângulo(largura=3, 4)
retângulo(largura=3, altura=4, "")
```

Uma exceção a essa regra é quando combinamos parâmetros obrigatórios e opcionais. Por exemplo, podemos passar os parâmetros obrigatórios sem usar seus nomes, mas respeitando a ordem usada na declaração e parâmetros nomeados apenas para escolher que parâmetros opcionais usar. Lembre-se de que ao passar o primeiro parâmetro nomeado (usando seu nome), todos os parâmetros seguintes também devem ter seus nomes especificados.

8.6 Funções como parâmetro

Um poderoso recurso de Python é permitir a passagem de funções como parâmetro. Isso permite combinar várias funções para realizar uma tarefa. Vejamos um exemplo na listagem 8.26.

► Listagem 8.26 – Funções como parâmetro

```
def soma(a,b):
    return a+b
def subtração(a,b):
    return a-b
def imprime(a,b, foper):
    print(foper(a,b)) ❶
imprime(5,4, soma) ❷
imprime(10,1, subtração) ❸
```

As funções `soma` e `subtração` foram definidas normalmente, mas a função `imprime` recebe um parâmetro chamado `foper`. Nesse caso, `foper` é a função que passaremos como parâmetro. Em ❶, passamos os parâmetros `a` e `b` para a função `foper` que ainda não conhecemos. Observe que passamos `a` e `b` a `foper` como faríamos com qualquer outra função. Em ❷, chamamos a função `imprime`, passando a função `soma` como parâmetro. Observe que, para passar a função `soma` como parâmetro, escrevemos apenas seu nome, sem passar qualquer parâmetro ou mesmo usar parênteses, como faríamos com uma variável normal. Em ❸, chamamos a função `imprime`, mas dessa vez passando a função `subtração` como `foper`.

Passar funções como parâmetro permite injetar funcionalidades dentro de outras funções, tornando-as configuráveis e mais genéricas. Vejamos outro exemplo na listagem 8.27.

► Listagem 8.27 – Configuração de funções com funções

```
def imprime_lista(L, fimpressão, fcondição):
    for e in L:
        if fcondição(e):
            fimpressão(e)
def imprime_elemento(e):
    print("Valor: %d" % e)
def épar(x):
    return x % 2 == 0
```

```
def éimpar(x):
    return not épar(x)
L=[1,7,9,2,11,0]
imprime_lista(L, imprime_elemento, épar)
imprime_lista(L, imprime_elemento, éimpar)
```

A função `imprime_lista` recebe uma lista e duas funções como parâmetro. A função `fimprime` é utilizada para realizar a impressão de cada elemento, mas só é chamada se o resultado da função passada como `fcondição` for verdadeiro. No exemplo, chamamos `imprime_lista` passando a função `épar` como parâmetro, fazendo com que apenas os elementos pares sejam impressos. Depois, passamos a função `éimpar`, de forma a imprimir apenas elementos cujo valor seja ímpar.

8.7 Empacotamento e desempacotamento de parâmetros

Outra flexibilidade da linguagem Python é poder passar parâmetros empacotados em uma lista. Vejamos um exemplo na listagem 8.28.

► Listagem 8.28 – Empacotamento de parâmetros em uma lista

```
def soma(a,b):
    print(a+b)
L=[2,3]
soma(*L) ❶
```

Em ❶, estamos utilizando o asterisco para indicar que queremos desempacotar a lista `L` utilizando seus valores como parâmetro para a função `soma`. No exemplo, `L[0]` será atribuído a `a` e `L[1]` a `b`. Esse recurso permite armazenar nossos parâmetros em listas e evita construções do tipo `soma(L[0], L[1])`.

Vejamos outro exemplo, onde utilizaremos uma lista de listas para realizar várias chamadas a uma função dentro de um `for` (Listagem 8.29).

► Listagem 8.29 – Outro exemplo de empacotamento de parâmetros em uma lista

```
def barra(n=10, c="*"):
    print(c*n)
L=[[5, "-"], [10, "*"], [5, [6,"."]]]
for e in L:
    barra(*e)
```

Observe que mesmo usando o empacotamento de parâmetros, recursos como parâmetros opcionais ainda são possíveis quando e contém apenas um elemento.

8.8 Desempacotamento de parâmetros

Podemos criar funções que recebem um número indeterminado de parâmetros utilizando listas de parâmetros (Listagem 8.30).

► Listagem 8.30 – Função soma com número indeterminado de parâmetros

```
def soma(*args):
    s=0
    for x in args:
        s+=x
    return s
soma(1,2)
soma(2)
soma(5,6,7,8)
soma(9,10,20,30,40)
```

Também podemos criar funções que combinem parâmetros obrigatórios e uma lista de parâmetros (Listagem 8.31).

► Listagem 8.31 – Função imprime_maior com número indeterminado de parâmetros

```
def imprime_maior(mensagem, *numeros):
    maior = None
    for e in numeros:
        if maior == None or maior < e:
            maior = e
    print(mensagem, maior)
imprime_maior("Maior:",5,4,3,1)
imprime_maior("Max:", *[1,7,9])
```

Observe que o primeiro parâmetro é `mensagem`, tornando-o obrigatório. Assim, `imprime_maior()` retorna erro, pois o parâmetro `mensagem` não foi passado, mas `imprime_maior("Max:")` escreve `None`. Isso porque `numeros` é recebido como uma lista, podendo inclusive estar vazia.

8.9 Funções Lambda

Podemos criar funções simples, sem nome, chamadas de funções lambda. Vejamos um exemplo na listagem 8.32.

► Listagem 8.32 – Função lambda que recebe um valor e retorna o dobro dele

```
a=lambda x: x*2 ❶
print(a(3)) ❷
```

Em ❶, definimos uma função lambda que recebe um parâmetro, no caso `x`, e que retorna o dobro desse número. Observe que tudo foi feito em apenas uma linha. A função é criada e atribuída a variável `a`. Em ❷, utilizamos `a` como uma função normal.

Funções lambda também podem receber mais de um parâmetro, como mostra a listagem 8.33.

► Listagem 8.33 – Função lambda que recebe mais de um parâmetro

```
aumento=lambda a,b: (a*b/100)
aumento(100, 5)
```

8.10 Módulos

Depois de criarmos várias funções, os programas ficaram muito grandes. Precisamos armazenar nossas funções em outros arquivos e, de alguma forma usá-las, sem precisar reescrevê-las, ou pior, copiar e colar.

Python resolve o problema com módulos. Todo arquivo `.py` é um módulo, podendo ser importado com o comando `import`.

Vejamos dois programas: `entrada.py` (Listagem 8.34) e `soma.py` (Listagem 8.35).

► Listagem 8.34 – Módulo entrada (entrada.py)

```
def valida_inteiro(mensagem, mínimo, máximo):
    while True:
        try:
            v=int(input(mensagem))
            if v >= mínimo and v <= máximo:
                return v
        except ValueError:
            print("Por favor, entre com um número inteiro")
```

```

else:
    print("Digite um valor entre %d e %d" % (mínimo, máximo))
except:
    print("Você deve digitar um número inteiro")

```

► Listagem 8.35 – Módulo soma (soma.py) que importa entrada

```

import entrada
L=[]
for x in range(10):
    L.append(entrada.valida_inteiro("Digite um número:", 0, 20))
print("Soma: %d" % (sum(L)))

```

Utilizando o comando `import` foi possível chamar a função `valida_inteiro`, definida em `entrada.py`. Observe que, para chamar a função `valida_inteiro` escrevemos o nome do módulo antes do nome da função: `entrada.valida_inteiro`.

Usando módulos, podemos organizar nossas funções em arquivos diferentes e chamá-las quando necessário, sem precisar reescrever tudo.

Nem sempre queremos utilizar o nome do módulo para acessar uma função: `valida_inteiro` pode ser mais interessante que `entrada.valida_inteiro`.

Para importar a função `valida_inteiro` de forma a poder chamá-la sem o prefixo do módulo, substitua o `import` na listagem 8.35 por `from entrada import valida_inteiro`.

Depois, modifique o programa substituindo `entrada.valida_inteiro` por apenas `valida_inteiro`.

Você deve utilizar esse recurso com atenção, pois informar o nome do módulo antes da função é muito útil quando os programas crescem, servindo de dica para que se saiba que módulo define tal função, facilitando sua localização e evitando o que se chama de conflito de nomes. Dizemos que um conflito de nomes ocorre quando dois ou mais módulos definem funções com nomes idênticos. Nesse caso, utilizar a notação de chamada `módulo.função` resolve o conflito, pois a combinação do nome do módulo com o nome da função é única, evitando a dúvida de descobrir o módulo que define a função que estamos chamando.

Outra construção que deve ser utilizada com cuidado é `from entrada import *`. Nesse caso, estariamos importando todas as definições do módulo `entrada`, e não apenas a função `valida_inteiro`. Essa construção é perigosa, porque se dois módulos definirem funções com o mesmo nome, a função utilizada no programa será a do último `import`. Isso é especialmente difícil de encontrar em programas grandes, e seu uso não é aconselhado.

8.11 Números aleatórios

Uma forma de gerar valores para testar funções e popular listas é utilizar números aleatórios ou randômicos. Um número aleatório pode ser entendido como um número tirado ao acaso, sem qualquer ordem ou sequência predeterminada, como em um sorteio.

Para gerar números aleatórios em Python, vamos utilizar o módulo `random`. O módulo traz várias funções para geração de números aleatórios e mesmo números gerados com distribuições não uniformes. Se quiser saber mais sobre distribuições, consulte um bom livro de estatística ou a Wikipédia (http://pt.wikipedia.org/wiki/Distribuição_de_probabilidade) para matar a curiosidade. Vejamos a função `randint`, que recebe dois parâmetros, sendo o primeiro o início da faixa de valores a considerar para geração; e o segundo, o fim dessa faixa. Tanto o início quanto o fim são incluídos na faixa.

► Listagem 8.36 – Gerando números aleatórios

```

import random
for x in range(10):
    print(random.randint(1,100))

```

Na listagem 8.36, utilizamos `randint` com 1 e 100, logo, os números retornados devem estar na faixa entre 1 e 100. Se chamarmos o número retornado de `x`, teremos `1 <= x <= 100`. Cada vez que executarmos esse programa, teremos valores diferentes sendo gerados. Essa diferença também é interessante para trabalharmos com jogos ou introduzirmos elementos de sorte em nosso programa.

► Listagem 8.37 – Adivinhando o número

```

import random
n=random.randint(1,10)
x=int(input("Escolha um número entre 1 e 10:"))
if (x==n):
    print("Você acertou!")
else:
    print("Você errou.")

```

Veja o programa da listagem 8.37. Mesmo lendo a listagem, não podemos determinar a resposta. A cada execução, um número diferente pode ser escolhido.

Esse tipo de aleatoriedade é utilizado em vários jogos e torna a experiência única, fazendo com que o mesmo programa possa ser utilizado várias vezes com resultados diferentes.

Exercício 8.13 Altere o programa da listagem 8.37 de forma que o usuário tenha três chances de acertar o número. O programa termina se o usuário acertar ou errar três vezes.

Podemos também gerar números aleatórios fracionários ou de ponto flutuante com a função `random` (Listagem 8.38). A função `random` não recebe parâmetros e retorna valores entre 0 e 1.

► Listagem 8.38 – Números aleatórios entre 0 e 1 com `random`

```
import random
for x in range(10):
    print(random.random())
```

Para obter valores fracionários dentro de uma determinada faixa, podemos usar a função `uniform` (Listagem 8.39).

► Listagem 8.39 – Números aleatórios de ponto flutuante com `uniform`

```
import random
for x in range(10):
    print(random.uniform(15,25))
```

Podemos utilizar a função `sample` para escolher aleatoriamente elementos de uma lista. Essa função recebe a lista e a quantidade de amostras (*samples*) ou elementos que queremos retornar. Vejamos como gerar os números de um cartão do jogo da Lotto (Listagem 8.40).

► Listagem 8.40 – Seleção de amostras de uma lista aleatoriamente

```
import random
print(random.sample(range(1,101), 6))
```

Se quisermos embaralhar os elementos de uma lista, podemos utilizar a função `shuffle`. Ela recebe a lista a embaralhar, alterando-a (Listagem 8.41).

► Listagem 8.41 – Ação de embaralhar elementos de uma lista

```
import random
a=list(range(1,11))
random.shuffle(a)
print(a)
```

Exercício 8.14 Altere o programa da listagem 745, o jogo da forca. Escolha a palavra a adivinhar utilizando números aleatórios.

8.12 A função `type`

A função `type` retorna o tipo de uma variável, função ou objeto em Python. Vejamos alguns exemplos no interpretador (Listagem 8.42):

► Listagem 8.42 – A função `type`

```
>>> a=5
>>> print(type(a))
<class 'int'>
>>> b="Olá"
>>> print(type(b))
<class 'str'>
>>> c=False
>>> print(type(c))
<class 'bool'>
>>> d=0.5
>>> print(type(d))
<class 'float'>
>>> f=print
>>> print(type(f))
<class 'builtin_function_or_method'>
>>> g=[]
>>> print(type(g))
<class 'list'>
>>> h={}
>>> print(type(h))
```

```
<class 'dict'>
>>> def função():
    pass
>>> print(type(função))
<class 'function'>
```

Observe que o tipo retornado é uma classe. Vejamos como utilizar essa informação em um programa (Listagem 8.43).

► Listagem 8.43 – Utilizando a função type em um programa

```
import types
def diz_o_tipo(a):
    tipo = type(a)
    if tipo == str:
        return("String")
    elif tipo == list:
        return("Lista")
    elif tipo == dict:
        return("Dicionário")
    elif tipo == int:
        return("Número inteiro")
    elif tipo == float:
        return("Número decimal")
    elif tipo == types.FunctionType:
        return("Função")
    elif tipo == types.BuiltinFunctionType:
        return("Função interna")
    else:
        return(str(tipo))
print(diz_o_tipo(10))
print(diz_o_tipo(10.5))
print(diz_o_tipo("Alô"))
print(diz_o_tipo([1,2,3]))
print(diz_o_tipo({"a":1, "b":50}))
print(diz_o_tipo(print))
print(diz_o_tipo(None))
```

Vamos verificar como utilizar a função `type` para exibir os elementos de uma lista na qual os elementos são de tipos diferentes. Dessa forma, você pode executar operações diferentes dentro da lista, como verificar se um elemento é outra lista ou um dicionário.

► Listagem 8.44 – Usando type com os elementos de uma lista

```
L=[ 2, "Alô", [ "!" ], { "a":1, "b":2 } ]
for e in L:
    print(type(e))
```

Vejamos como navegar em uma lista usando o tipo de seus elementos. Imagine uma lista na qual os elementos podem ser do tipo string ou lista. Se forem do tipo string, podemos simplesmente imprimi-los. Se forem do tipo lista, teremos que imprimir cada elemento.

► Listagem 8.45 – Navegando listas a partir do tipo de seus elementos

```
import types
L=["a", [ "b", "c", "d" ], "e"]
for x in L:
    if type(x)==str:
        print(x)
    else:
        print("Lista:")
        for z in x:
            print(z)
```

Exercício 8.15 Utilizando a função `type`, escreva uma função recursiva que imprima os elementos de uma lista. Cada elemento deve ser impresso separadamente, um por linha. Considere o caso de listas dentro de listas, como `L=[1, [2,3,[5,6,7]]]`. A cada nível, imprima a lista mais à direita, como fazemos ao indentar blocos em Python. Dica: envie o nível atual como parâmetro e utilize-o para calcular a quantidade de espaços em branco à esquerda de cada elemento.