

Direto ao ponto

Vamos apresentar o uso de funções em programas escritos em Python. Trata-se de um assunto importante que envolve muitos conceitos, os quais, em um primeiro momento, podem parecer demasiadamente abstratos ao programador iniciante. Assim sendo, a abordagem aqui adotada é a de primeiro mostrar como é feito, para depois aprofundar os conceitos.

Observe com atenção o código a seguir, bem como a Figura 5.1, que contém o resultado de sua execução.

Exemplo 5.1 Criação e uso de funções `def Soma(X, Y):` # linha 1

`R = X + Y`

`return R`

`a = int(input("Digite um valor para a: "))`

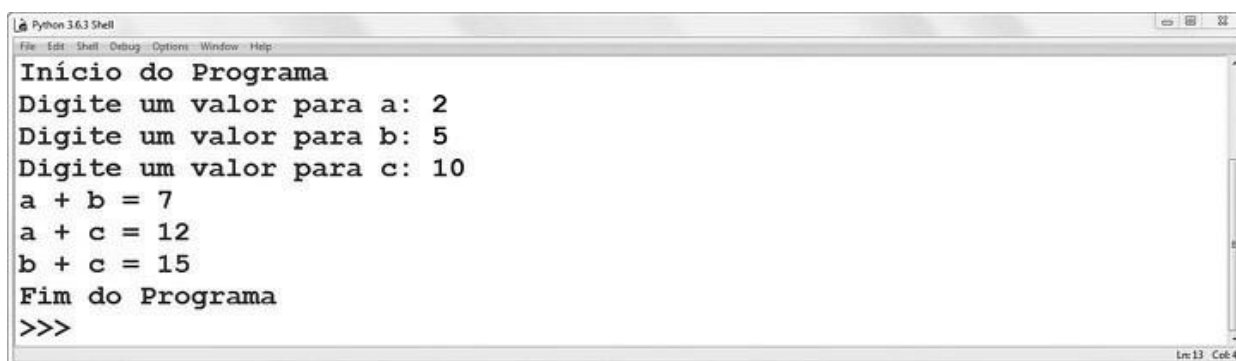
`b = int(input("Digite um valor para b: "))` # linha 2

`c = int(input("Digite um valor para c: "))`

`s = Soma(a, b)` # linha 3 `print("a + b = {}".format(s))` `s = Soma(a, c)` # linha 4

`print("a + c = {}".format(s))` `s = Soma(b, c)` # linha 5 `print("b + c = {}".format(s))`

`print("Fim do Programa")`



Exemplo de uso de função.

Nesse código está definida uma função chamada `Soma`, que calcula e retorna



a soma dois valores a ela fornecidos, por meio dos parâmetros X e Y. Note que essa função foi chamada três vezes no programa, nas linhas 3, 4 e 5. Em cada uma das chamadas foram passados diferentes valores, e a função calculou e retornou a soma dos mesmos.

Toda vez que uma função é chamada, o fluxo de execução dos comandos é interrompido no local da chamada e a execução é transferida para os comandos internos da função. Se houver parâmetros – como nesse exemplo há X e Y –, estes são devidamente carregados com os valores passados na chamada, antes de se executar o primeiro comando interno. Ao término da função, ou seja, após a execução de todos os comandos internos, a execução retorna para a instrução imediatamente seguinte ao ponto em que ocorreu a chamada.

Como fica claro neste primeiro exemplo, existem dois aspectos relevantes referentes ao uso de funções em programação:

1.A definição da função, que é o ponto do programa no qual ela é criada.

2.O uso da função, que são os pontos onde ela é usada (ou chamada). A chamada de uma função pode ocorrer na parte principal do programa ou dentro de outra função.

No Exemplo 5.1 a função calcula algo, uma soma, que poderia ser feita com uma simples expressão algébrica. A simplicidade desse exemplo pode levar o leitor iniciante nos estudos de programação a questionar qual é a importância de seu uso. Na prática, em uma leitura superficial, fica-se com a impressão de que o programa ficou mais complicado que o necessário. Bem, isso é verdade, ficou mesmo mais complicado. No entanto, este é apenas um exemplo inicial, em seguida serão apresentados conceitos que deverão responder a tal questionamento e compreender que o uso de funções na programação moderna, mais do que importante, é fundamental.

A importância das funções

Em programação de computadores, o termo “função” tem um significado totalmente diferente daquele empregado em outras áreas do conhecimento, como matemática, biologia ou química.

No contexto de programação, uma função é um conjunto de comandos, ou



bloco de código, ao qual se atribui um nome identificador que executa certa tarefa, e pode produzir e retornar algum resultado.

Um programa pode conter tantas funções quanto se queira, bastando ao programador desenvolvê-las conforme julgue adequado.

Além disso, as funções podem ser desenvolvidas, testadas e agrupadas em bibliotecas de modo a ficar disponíveis para uso em mais de um programa diferente, sempre que o programador necessite delas.

Dividir para conquistar

A expressão “dividir para conquistar” ilustra um aspecto central relativo ao uso de funções que consiste em dividir um problema maior e mais complexo, em partes menores e mais simples. Em seguida, implementar a solução das partes simples a partir da criação de uma função para cada parte e, por fim, montar a solução completa juntando e justapondo as tais funções de modo apropriado.

Essa abordagem tem sido utilizada ao longo de décadas, e em todo este tempo se mostra comprovadamente eficaz. Para o programador iniciante, à primeira vista, pode parecer que o uso de funções é um complicador desnecessário na elaboração de um programa. Deve-se lembrar, no entanto, que os programadores iniciantes estão apenas dando os primeiros passos neste mundo da programação de computadores, e os programas que desenvolvem são pequenos contendo dezenas ou, no máximo, umas poucas centenas de linhas de código.

Reúso de código e eliminação da redundância

Um segundo aspecto fundamental relativo ao uso de funções em programas diz respeito à eliminação da redundância possibilitada pelo reúso de um código pronto.

É comum que um programador necessite executar certa tarefa diversas vezes no mesmo programa. Exatamente como foi feito no Exemplo 5.1, em que a função Soma foi utilizada três vezes. Suponha que essa função tivesse uma centena de linhas de código em vez de apenas duas. Fica claro o ganho em uma situação assim, pois, em vez de repetir um extenso código nos vários pontos do programa, escreve-se uma função que efetua a tarefa e a chama sempre que for preciso.

Nas demais seções deste capítulo serão apresentados e exemplificados todos os aspectos relativos à criação de funções e seu uso.



Definição e uso de funções

Definição de funções

Em Python, uma função é definida por meio de um cabeçalho que contém quatro elementos: a palavra reservada `def`, um nome válido, parâmetros entre parênteses e o caractere `:`. Esse cabeçalho é sucedido por um bloco de comandos identados que constitui o corpo da função. Todos esses elementos estão presentes no Exemplo 5.1.

O nome pode ser qualquer identificador válido segundo as regras vistas no Item 2.2. Após o nome da função, entre parênteses, são fornecidos os parâmetros que ela receberá, se existirem. Por fim, o caractere `:` indica ao interpretador o término do cabeçalho. Todos os comandos internos à função devem estar identados para que o interpretador reconheça que são comandos subordinados ao cabeçalho e, portanto, pertencentes à função.

Parâmetros de funções

Os parâmetros representam dados de entrada a serem utilizados pela função e são opcionais. No Exemplo 5.1 a função `Soma` necessita receber como dados de entrada os valores a serem somados. No caso, foram, então, incluídos dois parâmetros, `X` e `Y`. Não existe qualquer limite para a quantidade de parâmetros que uma função pode receber, de modo que o programador é livre para criar a função com tantos parâmetros quanto necessário.

Por outro lado, haverá funções que não necessitam de qualquer valor de entrada. Nesses casos, ainda assim os parênteses devem estar presentes no cabeçalho. O Exemplo 5.2 ilustra uma função desse tipo. Ela faz a leitura do teclado e o que for digitado é convertido para um número inteiro e retornado.

Exemplo 5.2 Função sem parâmetro de entrada `def LerInteiro():`

```
n = int(input("Digite um número inteiro: ")) return n
```

```
x = LerInteiro()
```

```
print("Valor lido na função = {0}".format(x))
```

No Exemplo 5.1 a função `Soma` foi utilizada para somar números inteiros. No entanto, os parâmetros `X` e `Y` que recebem os valores não têm qualquer tipificação.



Em outras palavras, seria possível passar números reais, complexos ou quaisquer outros tipos de dados para eles. Uma vez que o operador de adição “+” esteja definido para os dados que forem passados, a função deverá funcionar normalmente. Observem-se os Exemplos 5.3 e 5.4.

Exemplo 5.3 Função Soma com números reais passados como parâmetros def Soma(X, Y):

```
R = X + Y
```

```
return R
```

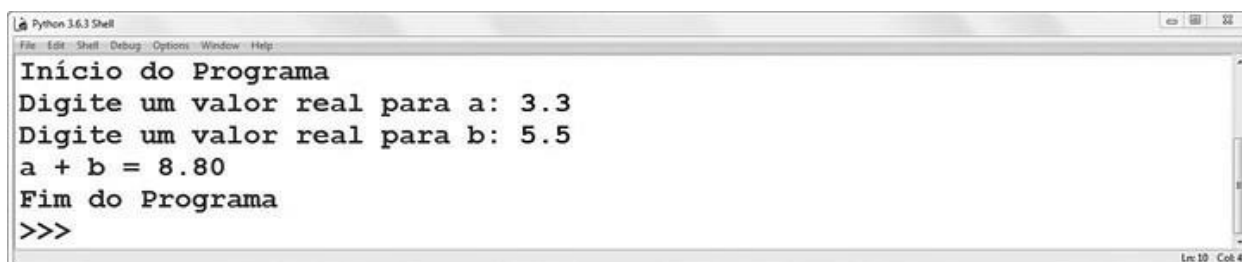
```
print("Início do Programa")
```

```
a = float(input("Digite um valor real para a: "))
```

```
b = float(input("Digite um valor real para b: ")) s = Soma(a, b)
```

```
print("a + b = {0:.2f}".format(s))
```

```
print("Fim do Programa")
```



```
Python 3.6.3 Shell
File Edit Shell Debug Options Window Help
Início do Programa
Digite um valor real para a: 3.3
Digite um valor real para b: 5.5
a + b = 8.80
Fim do Programa
>>>
```

Exemplo 5.4 Função Soma com strings passados como parâmetros def Soma(X, Y):

```
R = X + Y
```

```
return R
```

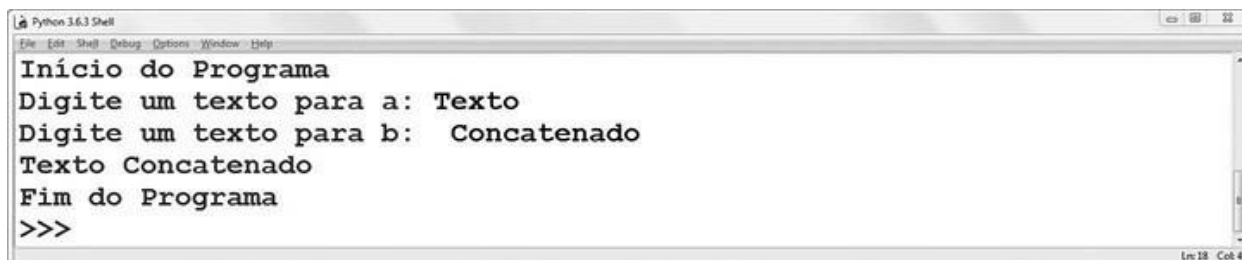
```
print("Início do Programa")
```

```
a = input("Digite um texto para a: ")
```



```
b = input("Digite um texto para b: ") s = Soma(a, b)
print(s)

print("Fim do Programa")
```



```
Python 3.6.3 Shell
File Edit Shell Debug Options Window Help
Início do Programa
Digite um texto para a: Texto
Digite um texto para b: Concatenado
Texto Concatenado
Fim do Programa
>>>
```

Sem qualquer alteração no cabeçalho ou no corpo da função Soma, e apenas fazendo adaptações nos dados de entrada lidos fora da função, foi possível executar o programa sem qualquer problema. Essa possibilidade de X e Y receberem números inteiros, no Exemplo 5.1, números reais, em 5.2, e strings, em 5.3, mostra que a função Soma pode funcionar de maneira polimórfica, ou seja, o interpretador busca adequar o comportamento do programa ao tipo de dado que está sendo utilizado na operação.

A seguir, outro exemplo envolvendo listas. O operador “+” está definido para listas e é capaz de produzir uma lista resultante juntando as duas listas passadas.

Exemplo 5.5 Função Soma com listas passadas como parâmetros def Soma(X, Y):

```
R = X + Y

return R
print("Início do Programa")

a = [1, 2, 3]

b = [11, 12, 13]

s = Soma(a, b) print(s)
print("Fim do Programa")
```



```
Python 3.6.3 Shell
File Edit Shell Debug Options Window Help
Início do Programa
[1, 2, 3, 11, 12, 13]
Fim do Programa
>>> type(s)
<class 'list'>
>>>
```

Isto tudo é possível porque Python é uma linguagem que utiliza tipagem dinâmica. E, uma vez que o operador “+” esteja definido para o tipo de dado que é passado, então, o código da função será executado de maneira correta.

Se tal operador não estiver definido, então, o interpretador levantará uma exceção que poderá ser tratada, conforme visto no Item 3.3.

Programadores experientes em outras linguagens, como C, C++ ou Java, podem estranhar um recurso como este, pois tais linguagens utilizam o conceito de tipagem estática, segundo o qual a função é criada com parâmetros de tipos bem definidos. No entanto, os tempos atuais apresentam demandas que exigem das linguagens flexibilidade com concisão, e a tipagem dinâmica é uma resposta eficaz a essa demanda. Por esse motivo, Python, bem como PHP, JavaScript e outras linguagens mais recentes a adotam.

Parâmetros com valores-padrão

Os parâmetros podem apresentar valores-padrão – *default* – atribuídos na definição da função. Quando um parâmetro tem valor-padrão, ele se torna opcional na chamada da função, e caso seja omitido o valor-padrão é utilizado.

Exemplo 5.6 Função soma com parâmetro Y com valor-padrão def Soma(X, Y = 1):

R = X + Y

return R

print(“Início do Programa”)

a = int(input(“Digite um valor para a: “))




```
b = int(input("Digite um valor para b: ")) s = Soma(a, b)
print("a + b = {}".format(s)) s = Soma(a)
print("a + 1 = {}".format(s)) print("Fim do Programa")
```



Neste caso, o parâmetro Y tem valor-padrão igual a 1. Assim sendo, na chamada da função, caso o segundo parâmetro seja omitido, o valor 1 será assumido como valor de Y. Na execução do Exemplo 5.6 foram digitados os valores: 2 para o objeto a e 10 para o objeto b.

Na primeira chamada da função Soma, foram passados a e b: `s = Soma(a, b)` # s resulta igual a 12 pois `a = 2` e `b = 10`

Já na segunda chamada da função Soma, foi passado apenas a: `s = Soma(a)` # s resulta igual a 3 pois `a = 2` e o segundo # parâmetro foi omitido, então, Y assumiu o valor 1.

Ao definir o cabeçalho de uma função, é preciso respeitar a regra de que primeiro devem ser relacionados todos os parâmetros que não apresentam valor-padrão e, depois, aqueles que os apresentam. O interpretador Python não aceita que um parâmetro sem valor-padrão seja declarado após outro que o contém.

Parâmetros nomeados

Outra característica de Python é a possibilidade de utilizar parâmetros nomeados.

Até o momento, por ser algo intuitivo, nada foi dito sobre a ordem de atribuição dos parâmetros. E a forma que se vem utilizando até aqui é a **passagem posicional**. Nos vários exemplos anteriores admitiu-se que a ordem de atribuição dos objetos passados na chamada da função é posicional, ou seja, o primeiro objeto passado na chamada é assumido pelo primeiro parâmetro relacionado no cabeçalho, o segundo passado na chamada é assumido pelo segundo relacionado no cabeçalho, e assim por diante. Essa suposição está correta, sendo assim mesmo.

```
def Soma(X, Y):
```



... ↑ ↑

`s = Soma(a, b) # a é passado para X e b é passado para Y`

No entanto, a ordem pode ser trocada, desde que se faça referência ao nome do parâmetro que receberá o valor passado, ficando assim:

`s = Soma(Y = b, X = a) # a é passado para X e b é passado para Y # porém, as ordens estão trocadas`

Essa forma de fazer a passagem de parâmetros é conhecida como **passagem nomeada** e pode ser utilizada em qualquer chamada de função. Nestes casos, a ordem não faz diferença, desde que, na chamada, todos os parâmetros sejam nomeados.

Aqui, há de se tomar um cuidado. Misturar parâmetros posicionais e nomeados em uma mesma chamada é possível, porém, deve-se respeitar a regra de que os primeiros podem ser posicionais, porém, após o primeiro parâmetro nomeado, todos devem ser nomeados.

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

...

`Funcao(3, 6, 9, 12) # chamada Ok.`

`# Os parâmetros são posicionais`

`Funcao(N=6, P=12, M=3, O=9) # chamada Ok. # Os parâmetros são nomeados`

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

`# neste caso os dois primeiros são posicionais e os outros dois # são nomeados`

`Funcao(3, N=6, 9, 12) # chamada incorreta. Gera erro. Funcao(3, N=6, O=9, P=12) # chamada Ok.`

Empacotamento e desempacotamento de parâmetros

Por fim, existe uma terceira alternativa para passagem de parâmetros que é utilizada com um pouco menos de frequência, por se aplicar a situações mais



específicas. Trata-se de uma opção muito útil nos casos em que é preciso escrever uma função sem saber exatamente quantos parâmetros serão passados.

Esse número arbitrário será encapsulado em uma tupla que será passada para a função. Dentro da função, essa tupla poderá ser utilizada de qualquer maneira que o programador precise.

Exemplo 5.7 Função com empacotamento de parâmetros

```
>>> def Soma(*valores): r = 0
for i in valores:

    r += i return r
>>> Soma(3, 9)

12

>>> Soma(1, 2, 3, 4)

10

>>> Soma(1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1)

12

>>> Soma() 0
>>> Soma(5) 5
```

No Exemplo 5.7 foi definida a função `Soma`. Essa função recebe um parâmetro que está qualificado com o operador “*”, e o interpretador Python assumirá que está recebendo uma tupla. Quando a função é chamada, todos os parâmetros presentes na chamada, independentemente da quantidade, são coletados e convertidos em uma nova tupla que associa o objeto “valores” a essa tupla.

O caso inverso também é possível. Veja o Exemplo 5.8. A função `ExibeFormatado()` deve receber três parâmetros posicionais. É possível utilizar uma



lista ou uma tupla para passar tais parâmetros, porém, há duas condições:

- É preciso utilizar o operador “*” para informar ao interpretador que a lista (ou tupla) deve ser desempacotada.
- É preciso que a lista (ou tupla) tenha exatamente o número de parâmetros posicionais esperados pela função. Caso isso não aconteça, ocorrerá erro, como mostrado a seguir.

Exemplo 5.8 Função com desempacotamento de parâmetros

```
>>> def ExibeFormatado(a, b, c): print("1º valor = {}".format(a))  
print("2º valor = {}".format(b))  
  
print("3º valor = {}".format(c))
```

```
>>> L = [31, 77, 193]
```

```
>>> ExibeFormatado(*L) 1º valor = 31
```

```
2º valor = 77 3º valor = 193
```

```
>>> L = [43, 22, 323, 31]
```

```
>>> ExibeFormatado(*L) Traceback (most recent call last):
```

```
File "<pyshell#32>", line 1, in <module> ExibeFormatado(*L)
```

```
TypeError: ExibeFormatado() takes 3 positional arguments but 4 were given
```

A escolha entre utilizar um método ou outro de chamada de função é do programador. Há aqueles que preferem uma forma e outros que preferem outra. Em ambos os casos, haverá argumentos a favor e contra, e não é objetivo deste capítulo discuti-los. Fica aqui o convite para que teste as várias formas e adote a que considerar mais adequada ao seu estilo e às necessidades de seus algoritmos.

Retornos de funções

Em todas as linguagens é possível existir uma função que não retorna qualquer valor, bem como é possível retornar um ou muitos valores. Em Python isso também é assim.

Para que uma função tenha retorno basta utilizar a instrução `return`, que produz



dois efeitos: retorna o objeto que é colocado à sua frente e encerra a função imediatamente.

Em todos os exemplos anteriores a função Soma retornava um resultado contido no objeto R e terminava a função. Porém, nesses exemplos o `return` era a última linha do código da função. É possível, no entanto, que exista um `return` em qualquer ponto da função, como mostrado no Exemplo 5.9. Nesse exemplo há `return` em dois pontos distintos do código. No primeiro, se `X` for negativo, então, a função retornará o valor `-1` e será imediatamente encerrada. Caso contrário, seguirá sua execução até chegar à última linha, gerando um retorno igual a `0`.

Exemplo 5.9 Instrução *return* em pontos diversos dentro da função `def FuncaoF(X)`:

```
... # vários comandos aqui if X < 0:
return -1 else
... # mais comandos aqui, subordinados ao else

... # mais um pouco de comandos return 0
```

Em funções que não têm retorno a instrução `return` não é utilizada. Nestes casos, uma vez chamada, sua execução prosseguirá desde a primeira até a última instrução de seu bloco de código. Funções sem valor de retorno são chamadas de um modo diferente, bastando escrever seu nome como se fosse um comando e passar os parâmetros apropriados, conforme ilustrado no Exemplo 5.10.

Exemplo 5.10 Função que não retorna valor `def ExibeLista(L)`:

```
for x in L:
```

```
    print(x)
```

```
Pares = [2, 4, 6, 8, 10]
```

```
print("Exibição da lista, sendo um elemento por linha") ExibeLista(Pares)
```

Por sua vez, as funções que retornam valor podem ser chamadas do mesmo modo mostrado no Exemplo 5.10. Nesse caso, a função é executada, e sua ação interna, qualquer que seja, terá os devidos efeitos, porém, seu retorno não seria



aproveitado.

Normalmente, porém, as funções que produzem retornos têm estes devidamente aproveitados. Tal aproveitamento pode ocorrer de diversas maneiras, dependendo do tipo de retorno que se tem. A seguir, são apresentadas algumas, porém, não todas, as possibilidades.

- Atribuído a um objeto:

$s = \text{Soma}(a, b)$ # válido para qualquer caso

- Utilizado em meio a uma expressão aritmética: $s = 2 * \text{Soma}(a, b) / 10$ # para retorno numérico

- Utilizado em uma condição:

$\text{if } \text{Soma}(a, b) > 0$: # p/ retorno passível de comparação

- Utilizado como iterador:

for x in Operacoes(a, b) # se o retorno for um iterador # veja o Exemplo 5.11 a seguir

Retorno de múltiplos valores

Até agora, os exemplos utilizados retornavam um único valor. Porém, é possível escrever uma função que retorne múltiplos valores, como mostrado no Exemplo 5.11. Na função Operacoes desse exemplo são calculadas, respectivamente, a adição, a subtração, a multiplicação e a divisão dos dois parâmetros X e Y passados à função. O retorno é produzido escrevendo o comando return sucedido dos quatro valores calculados pela função.

O interpretador encapsula os vários elementos de retorno em uma tupla que é atribuída ao identificador “s”, o qual recebe o retorno da chamada da função. Essa tupla “s” pode ser utilizada da maneira que o programador desejar, utilizando os recursos vistos no Capítulo 4.

Exemplo 5.11 Função com múltiplos retornos def Operacoes(X, Y):


ad = X + Y su = X – Y mu = X * Y di = X / Y



```
return ad, su, mu, di print("Início do Programa")
a = int(input("Digite um valor para a: "))

b = int(input("Digite um valor para b: ")) s = Operacoes(a, b)
print(s)

print("Fim do Programa")
```



```
Python 3.6.3 Shell
File Edit Shell Debug Options Window Help
Início do Programa
Digite um valor para a: 12
Digite um valor para b: 4
(16, 8, 48, 3.0)
Fim do Programa
>>> type(s)
<class 'tuple'>
>>>
```

Alternativamente, é possível utilizar o recurso de atribuição múltipla do Python, como mostrado a seguir. Os objetos r1 a r4 recebem o retorno da função Operacoes segundo a posição relativa de cada uma, de modo que r1 recebe a adição, r2, a subtração, r3, a multiplicação, e r4, a divisão.

```
>>> r1, r2, r3, r4 = Operacoes(a, b)
```

```
>>> print(r1) 16
```

```
>>> print(r2) 8
```

```
>>> print(r3)
```

```
>>> print(r4) 3.0
```

Escopo de funções

Com o que foi visto até agora, pode-se depreender que em um programa escrito em Python existem dois ambientes distintos:

Ambiente externo à função – que será chamado de Global.

Ambiente interno à função – que será chamado de Local.

Escopo diz respeito ao estudo desses ambientes e da maneira como eles se



relacionam.

Durante a execução de um programa, todos os objetos criados fora de qualquer função são denominados globais e todos os objetos criados dentro de uma função são denominados 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 plenamente. 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.

Assim, recorrendo ao Exemplo 5.1 verifica-se que os objetos a, b e s ali presentes são globais e existem durante todo o tempo em que o programa estiver em execução, **inclusive dentro das funções**. Por sua vez, X, Y e R são locais e só existem durante a execução da função.

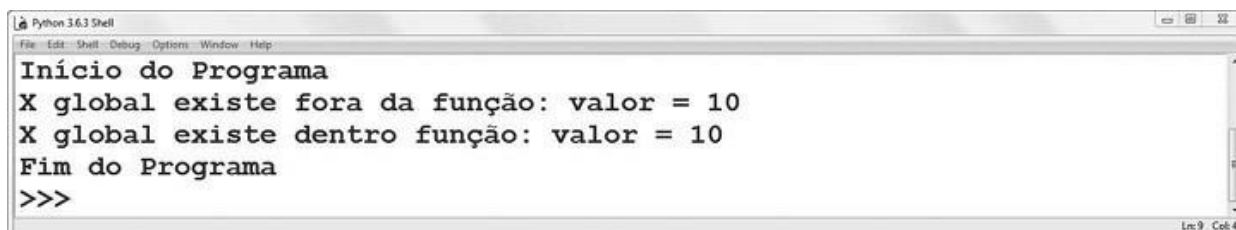
Agora, recorrendo ao Exemplo 5.12, cabe aprofundar um pouco mais o entendimento de escopo em Python. Nesse exemplo o objeto global X foi definido com o valor 10. Em seguida, a função EstudaEscopo foi chamada e dentro dela é feito o print de X, que, por ser global, está disponível dentro da função e pode ser utilizado no comando print.

Exemplo 5.12 Escopo de funções

```
def EstudaEscopo():  
    print("X global existe dentro função: valor = {}".format(X))  
print("Início do Programa")
```

```
X = 10
```

```
print("X global existe fora da função: valor = {}".format(X))  
EstudaEscopo()  
print("Fim do Programa")
```



```
Python 3.6.3 Shell  
File Edit Shell Debug Options Window Help  
Início do Programa  
X global existe fora da função: valor = 10  
X global existe dentro função: valor = 10  
Fim do Programa  
>>>
```



Portanto, desse exemplo se constata que, de fato, o objeto X está disponível dentro e fora da função. Qualquer objeto que seja criado dentro da função terá escopo local. Fazendo uma pequena alteração a esse exemplo tem-se uma nova situação em que foi incluído o objeto local Y e que recebe o valor X *

2.

Exemplo 5.13 Escopo de funções def EstudaEscopo():

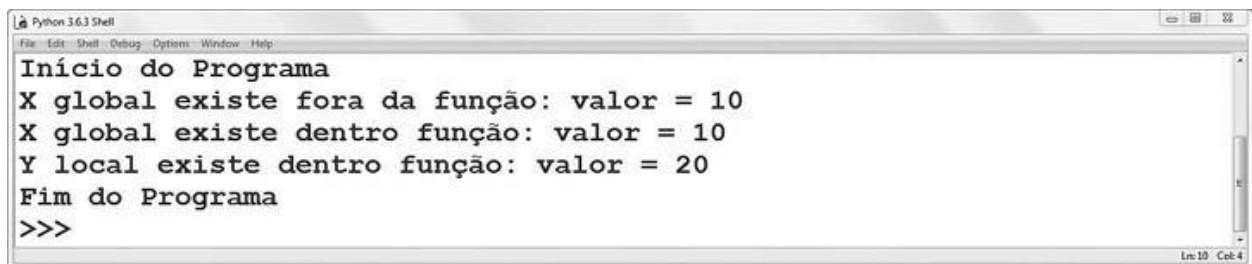
Y = X * 2

```
print("X global existe dentro função: valor = {0}".format(X)) print("Y local existe dentro função: valor = {0}".format(Y)) print("Início do Programa")
```

X = 10

```
print("X global existe fora da função: valor = {0}".format(X)) EstudaEscopo()
```

```
print("Fim do Programa")
```



Com o Exemplo 5.13 chega-se à situação que se quer discutir neste momento. Do modo como está construído o exemplo, se for acrescentada a linha X = 39 dentro da função, como mostrado a seguir, o leitor iniciante pode ser levado a deduzir que está sendo feita uma alteração no valor do objeto global X. Porém, ao executar esse programa, terá uma surpresa ao notar que X global continua com o conteúdo 10, embora dentro da função exiba o valor 39.

```
def EstudaEscopo():
```

```
X = 39
```

```
Y = X * 2
```



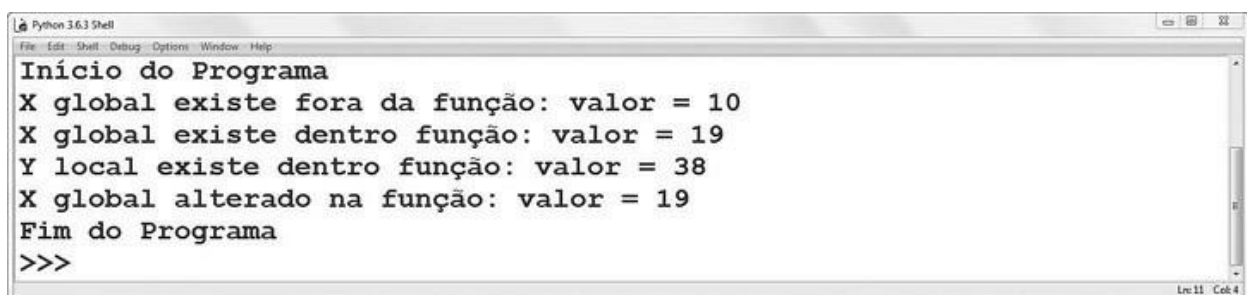
```
print("X global existe dentro função: valor = {0}".format(X)) print("Y local existe dentro função: valor = {0}".format(Y))
```

Como o interpretador cria os objetos em tempo de execução, o que ocorreu com essa alteração é que foi criado o objeto local com o identificador X, e a partir daí, dentro da função, quaisquer referências a X dizem respeito ao objeto local, e não mais ao global.

Caso o desejo do programador seja alterar o conteúdo do global X dentro da função, então, deve-se recorrer à diretiva "global" para informar o interpretador que se quer alterar o objeto global dentro da função, em vez de criar um objeto X local. O código fica assim:

Exemplo 5.14 Escopo de funções

```
def EstudaEscopo():  
    global X  
  
    X = 19 # aqui está sendo alterado o objeto X global  
    Y = X * 2  
    print("X global existe dentro função: valor = {0}".format(X))  
    print("Y local existe dentro função: valor = {0}".format(Y))  
    print("Início do Programa")  
  
    X = 10  
  
    print("X global existe fora da função: valor = {0}".format(X))  
    EstudaEscopo()  
    print("X global alterado na função: valor = {0}".format(X))  
  
    print("Fim do Programa")
```



A questão levantada com o uso do objeto X só aconteceu porque houve uma coincidência de nomes de objetos, sendo um de escopo global, e outro, de escopo local. Recomenda-se fortemente evitar tal situação. A medida simples que pode ser



tomada e que evita que isso aconteça é jamais utilizar objetos globais e locais com nomes idênticos. Essa recomendação não se restringe ao Python. Em geral, ela é válida na maioria das linguagens de programação.

Documentação de funções

A comunidade Python estimula e encoraja os programadores a sempre criar documentação apropriada para as funções que desenvolvem. Essa documentação é feita no próprio código do programa utilizando o recurso conhecido como *docstring*, que foi mencionado no Capítulo 2, Item 2.8. Observe-se o Exemplo 5.15, no qual foi inserido um *docstring* para a função `Operacoes`. Note que ele deve, obrigatoriamente, ser o primeiro elemento dentro da função e deve acompanhar a indentação.

Uma vez definido o *docstring* para a função, ele será utilizado para exibir a caixa de dica no momento de usá-la no IDLE, ou quando for utilizado o comando `help`. As PEPs 8 e 257 tratam desse assunto com mais profundidade.

Exemplo 5.15 Uso de *docstrings* para documentar uma função `def Operacoes(X, Y)`:

```
"""Realiza operações aritméticas com X e Y Retorna uma tupla contendo resultados na ordem adição, subtração, multiplicação, divisão
```

```
"""
```

```
ad = X + Y su = X - Y mu = X * Y di = X / Y
```

```
return ad, su, mu, di
```



```
Python 3.6.3 Shell
File Edit Shell Debug Options Window Help
>>> def Operacoes(X, Y):
    """Realiza operações aritméticas com X e Y
    Retorna uma tupla contendo resultados na ordem
    adição, subtração, multiplicação, divisão
    """
    ad = X + Y
    su = X - Y
    mu = X * Y
    di = X / Y
    return ad, su, mu, di

>>> Operacoes(
    (X, Y)
    Realiza operações aritméticas com X e Y
    Retorna uma tupla contendo resultados na ordem
    adição, subtração, multiplicação, divisão
```

Recursividade

Funções recursivas são aquelas que chamam a si mesmas. É um dos exemplos clássicos de função recursiva é o cálculo do fatorial de um número. No Capítulo 3 foi resolvido um exercício que calculava $N!$ usando um laço while.

Agora, será resolvido esse mesmo problema usando uma função recursiva. Toda função recursiva tem uma condição de parada. Essa condição determina em que ponto ela não mais chama a si mesma, iniciando o processo de saída das sucessivas chamadas.

Exemplo 5.16 Função recursiva – cálculo de $N!$

```
def Fatorial(N): # linha 1
    if N <= 1: # linha 2
        return 1 # linha 3
    else: # linha 4
        return N * Fatorial(N-1) # linha 5
print("Início do Programa")
X = int(input("Digite N: "))
F = Fatorial(X)
print("O fatorial de {0} é {1}".format(X, F))

print("Fim do Programa")
```

```
Python 3.6.3 Shell
File Edit Shell Debug Options Window Help
Início do Programa
Digite N: 6
O fatorial de 6 é 720
Fim do Programa
>>>
```



Na execução do Exemplo 5.16 foi fornecido como dado de entrada o valor 6. 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 X, 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 de (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 5.1.

Chamada	Condição na entrada	Ciclo de entrada com N * Fatorial (N-1)	Ciclo de saída das chamadas
0	—	Função principal	Recebe 720 da chamada 1
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.

Quadro 5.1 Ciclo de chamadas recursivas da função fatorial.



Outra maneira de ver o que está ocorrendo dentro de uma função recursiva é incluir nela um ou mais prints exibindo na tela os dados com os quais a função trabalha.

Funções comuns – números primos

Função recursiva – busca binária em lista ordenada

Escreva uma função que recebe dois parâmetros: uma lista L contendo números inteiros e organizada em ordem crescente; um número inteiro N. Essa função deve verificar se N está contido em L utilizando o algoritmo de busca binária e retornar à posição em que ele se encontra ou retornar 0 caso N não esteja na lista.

Considerações preliminares

O algoritmo de busca binária é clássico na programação de computadores e todo programador deve conhecê-lo. Trata-se de um algoritmo capaz de determinar se um valor está ou não presente em uma grande coleção de dados, partindo do pressuposto de que a coleção está ordenada, seja de maneira crescente ou decrescente. Esse algoritmo é bem mais eficiente – ou seja, mais rápido – que o algoritmo de busca sequencial visto no Capítulo 4 (exceção feita aos poucos casos em que o valor procurado está entre os primeiros da sequência).

A busca binária baseia-se no paradigma da divisão e conquista, no qual a coleção é sucessivamente dividida ao meio, reduzindo-se o espaço de busca e sempre comparando-se o valor buscado com o elemento que está no meio desse espaço de busca. Se o valor procurado for igual ao elemento do meio, então, a função retorna com sucesso. Se o valor procurado for menor a busca continua, porém, restrita à metade inicial da coleção, e caso seja maior a busca continua restrita à metade final.

Ilustração do algoritmo de busca binária.



Valor Buscado: 42

Lista

3	8	11	14	16	19	25	29	31	37	42	46	53	58	60	63	71	82
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17

Primeira tentativa – Elemento do meio 31 é menor que 42: a busca prossegue pela metade superior

3	8	11	14	16	19	25	29	31	37	42	46	53	58	60	63	71	82
---	---	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----

Primeira tentativa – Elemento do meio 58 é maior que 42: a busca prossegue pela metade inferior

37	42	46	53	58	60	63	71	82
----	----	----	----	----	----	----	----	----

Primeira tentativa – Elemento do meio 42 é o valor procurado: a busca termina com sucesso

37	42	46	53
----	----	----	----

Observação sobre os índices usados. O índice do elemento do meio é calculado com matemática de números inteiros. Assim, se o índice do início é 9 e o índice do final é 12 o índice do meio será:

$$\text{meio} = (\text{início} + \text{final}) // 2 \text{ que resulta meio} = 10$$

