6

TEXTO BASE

LINGUAGEM DE PROGRAMAÇÃO

Faculdade IMPACTA

Texto base

6

Criação de funções em Python

Prof. Me. Lucio Nunes de Lira
Prof. MSc. Rafael Maximo Carreira Ribeiro

Resumo

Nesta aula os objetivos são: (I) conhecer as funções importadas; (II) definir nossas próprias funções; (III) entender a relação entre argumentos e parâmetros de funções; (IV) compreender o que é valor de retorno; (V) introduzir o conceito de escopo de variáveis; (VI) escrever documentação nas funções; (VII) organizar o código-fonte.

6.1. Motivação

Conhecemos funções pré-definidas e sabemos como usá-las, mas não podemos nos limitar a elas, pois frequentemente teremos problemas para os quais não existem funções prontas. Seria inviável criar e integrar à linguagem uma função para cada problema existente, basicamente por dois motivos: 1) novos problemas surgem o tempo todo e; 2) o pacote de instalação do Python ficaria cada vez maior e eventualmente não teríamos mais espaço disponível. Por isso, veremos como criar nossas próprias funções e quais as principais vantagens ao usá-las em nossos programas.

💎 VOCÊ CONHECE?

Donald Knuth é professor emérito da Universidade de Stanford e um cientista da computação mundialmente conhecido. Também é autor da coleção de livros "*The Art of Computer Programming*", referência em Ciência da Computação.

Knuth é um dos principais responsáveis pelo campo da Análise de Algoritmos e pela criação do sistema de tipografia TeX.

Fonte da imagem: https://www-cs-faculty.stanford.edu/~knuth/

6.2. Introdução

Após criar soluções para diversos problemas, é comum que certas sequências de código se tornem frequentes, às vezes repetidas no mesmo programa. Uma forma de reduzir a duplicação de código é criar funções com os códigos mais utilizados.

A sintaxe para definição de uma função em Python é bastante simples, mas para usarmos as funções corretamente precisamos entender:

- Quando e por que devemos criar uma função;
- Como o funciona a passagem de valores para uma função;
- Como a função pode retornar um valor de resposta;
- O que acontece com as variáveis que criamos dentro de uma função.

As principais vantagens do uso de funções são:

- Abstração e reusabilidade, evitando duplicidade de código;
- Modularização, permitindo que um problema inicial seja dividido em problemas menores e mais fáceis de serem resolvidos;
- Separação de escopo, criando área de trabalho local para a função e evitando conflito entre variáveis internas da função e demais variáveis do programa.

Estas vantagens levam a um código com maior legibilidade, que será mais fácil de manter, atualizar e corrigir. A abstração está relacionada a separação entre "o que" precisa ser feito e "como" será feito. Quando utilizamos apenas funções integradas e importadas, a preocupação é apenas com "o que" a função faz, ao criar nossas próprias funções o "como" também é nossa responsabilidade. Porém, uma vez que nossas funções estão definidas e devidamente testadas, aumentando a confiança que estão corretas, voltamos a lidar apenas com o "o que", podendo utilizá-las de modo semelhante às funções integradas e importadas, usufruindo das mesmas vantagens.

6.3. Funções importadas

Vimos em Python as funções integradas, que podem ser usadas em qualquer trecho de código, pois são reconhecidas automaticamente pelo interpretador e, consequentemente, estão sempre disponíveis. Agora, aprenderemos a criar nossas próprias funções, mas antes disso, veremos como usar funções desenvolvidas por outros programadores e que estão disponíveis para uso em pacotes ou módulos extras.

O Python possui um conjunto de módulos extras que chamamos de *biblioteca padrão*[1]. Os módulos dessa biblioteca são instalados junto com o interpretador, mas não são carregados automaticamente ao executarmos o Python. Quando precisamos usar um módulo da biblioteca padrão, primeiro temos que *importá-lo* para o nosso código-fonte, indicando ao interpretador para carregá-lo e disponibilizá-lo para uso.

[1] Veja mais detalhes em: https://docs.python.org/pt-br/3/library/

A importação é feita com o comando `import`. A Codificação 6.1 tem um exemplo de importação do módulo de matemática `math`, que contém diversas funções e constantes matemáticas.

```
>>> import math
>>> math.pi
3.141592653589793
>>> math.log(10)
2.302585092994046
```

Codificação 6.1: Exemplo de importação do módulo `math` e uso de seus conteúdos.

Primeiro, importamos o módulo `math`. Isso criará uma variável de mesmo nome que nos dará acesso ao módulo (descubra o tipo do dado referenciado pela variável usando a função integrada `type`). Denominamos os conteúdos de um módulo como *propriedades*. Para acessar as propriedades, usamos um ponto após o nome da variável, seguido pelo nome da propriedade que queremos acessar. A descrição completa do módulo de matemática está disponível na documentação do Python (PSF, 2021).

Lembrando que se a propriedade for uma função, será necessário usar um par de parênteses para executá-la, semelhante a qualquer função que usamos até agora, pois caso contrário, acessaremos somente a referência para o objeto da função na memória.

Ao criar códigos-fonte, recomenda-se que as importações estejam no começo do arquivos, sendo necessário importar cada módulo uma única vez em um mesmo arquivo. Inclusive, instruções que tentem importar um módulo já importado não terão efeito.

6.4. Funções definidas pelo programador

Para criarmos novas funções é preciso defini-las. Em Python, para definir uma função, utiliza-se o comando `def`, conforme a estrutura exposta na Codificação 6.2.

```
def <nome da função>(<parâmetros>):
    <bloco de código>
```

Codificação 6.2: Estrutura das funções definidas pelo programador.

Crie *nomes de funções* que sejam claros indicativos do que ela faz, pois isso facilitará a legibilidade e uso. As regras para nomes de funções são idênticas àquelas para variáveis, até porque em Python o nome da função também é uma variável, só que uma variável que referencia um código de função que está na memória.

Portanto evite criar funções com o mesmo nome de outras variáveis, pois isso poderá gerar problemas em que a variável que referencia a função tem seu conteúdo sobrescrito por outro valor, ou uma variável qualquer tem seu conteúdo perdido por passar a referenciar o código de uma função.

Após o nome da função, deve-se colocar um *par de parênteses* e, entre eles, os *parâmetros*, se existirem. Os parâmetros são variáveis internas da função que recebem os argumentos passados quando a função é chamada. Note que o par de parênteses é obrigatório, mas, dependendo da função, pode não haver parâmetros, se houver mais de um, devem ser separados por vírgulas.

Na linha seguinte aos *dois pontos*, que sinalizam o fim da assinatura[2] da função, está o *bloco de código*, indentado para indicar que pertence à função, assim como é feito nas estruturas de seleção. Nesse bloco de código, pode-se inserir qualquer instrução válida, inclusive chamadas a outras funções integradas, importadas ou definidas pelo próprio programador. É possível definir funções dentro de outra, mesmo não sendo uma prática frequente. No entanto, chamar funções dentro de outra é algo bastante utilizado!

Escreva a Codificação 6.3 no editor, salve o arquivo como "funcao.py" e execute-o pressionando `[F5]` ou clicando em "Run > Run Module" na janela do editor.

```
def soma(n1, n2):
    s = n1 + n2
    return s
```

Codificação 6.3: Função que retorna como resposta a soma dos dois parâmetros.

Ocorreu algo na *Shell*? Apenas reiniciou, mas não exibiu a soma? Aparentemente nada aconteceu, porém a função `soma` foi criada corretamente e está na memória, pronta para ser chamada. Para conferir, digite na *Shell* apenas o nome da função, sem os parênteses e veja se obtém um resultado similar ao da Codificação 6.4.

```
>>> soma
<function soma at 0x0000017FC3BB7CA0>
```

Codificação 6.4: Endereço da função `soma` na memória.

Se obteve resultado similar, provavelmente sua função foi definida corretamente. Ao inserir apenas o nome da função na *Shell*, ou seja, sem os parênteses para chamá-la, o que vemos é o endereço de memória onde está o código da função, inclusive o endereço é um número natural que está representado em base hexadecimal (`0x`).

Note que `soma` é o nome da função e, como mencionamos, em Python o nome da função é uma variável que faz referência ao seu código. A referência é feita por meio do endereço da função, ou seja, a variável guarda o endereço de memória onde está o código da função para ser executado. Isso permite, por exemplo, que outra variável também receba o endereço e possamos chamar a função por meio dessa nova variável. Quer testar? Execute a Codificação 6.5 na *Shell* e se surpreenda!

[2] O nome da função juntamente com a sequência de seus parâmetros é chamada de assinatura/cabeçalho da função. Em outras linguagens, esse conceito também pode incluir o tipo dos parâmetros e o tipo do retorno da função, mas isso depende de como a linguagem trata a tipagem de dados.

```
>>> imprime = print
>>> imprime('Programando em português!')
Programando em português!
```

Codificação 6.5: Atribuição da referência de uma função à outra variável.

Retomando à criação da função, repare que existem dois momentos distintos ao lidarmos com funções definidas pelo próprio programador: (I) a definição da função e; (II) a invocação da função, também conhecido como "chamar a função".

O comando `def` só indica que a função deve ser definida e, após ser executado, disponibiliza a função na memória e a associa ao nome escolhido, porém não a chama automaticamente. Obviamente, toda função deve estar definida antes de ser chamada.

Após a função estar definida, para executá-la basta chamá-la de modo idêntico às funções integradas e importadas, usando parênteses após seu nome e com os argumentos entre eles. Veja exemplos de chamadas à `soma` na Codificação 6.6.

```
>>> soma(3, 5)
8
>>> soma(7, -9)
-2
```

Codificação 6.6: Chamadas à função `soma` definida pelo programador.

Sabemos que ao chamar uma função podemos passar valores como argumentos se, é claro, a função for definida com parâmetros que receberão esses argumentos. Assim, podemos entender os parâmetros com variáveis de entrada da função, e os argumentos como os valores que serão atribuídos a essas variáveis.

Por padrão, os argumentos são atribuídos aos parâmetros seguindo a ordem em que são passados na chamada da função. Logo, o primeiro argumento será atribuído ao primeiro parâmetro, o segundo argumento será atribuído ao segundo parâmetro e assim por diante. Veja na Figura 6.1 duas funções definidas pelo programador e suas respectivas chamadas, note a relação entre os argumentos e os parâmetros.

Na Codificação 6.3, os parâmetros da função `soma` são `n1` e `n2`, e ela foi chamada duas vezes na Codificação 6.6, uma com os argumentos `3` e `5`, e outra com os argumentos `7` e `-9`, *retornando*, em cada chamada, o resultado da soma de seus parâmetros. O retorno de um valor é feito com o comando `return` seguido pelo valor retornado, que pode ser originado de uma variável ou expressão mais complexa.

Modificaremos a Codificação 6.3 para melhorar a visualização do fluxo de execução do programa contido em "funcao.py", altere-o conforme a Codificação 6.7.

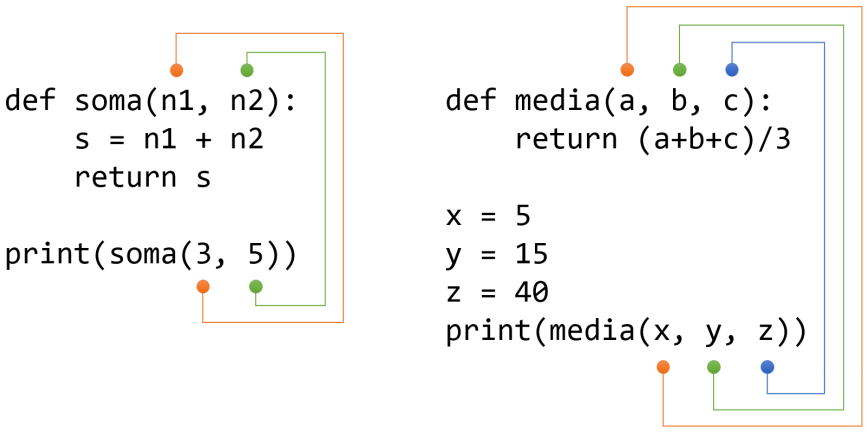

Figura 6.1: Relação entre argumentos e parâmetros. Fonte: Elaborado pelo autor.

```
def soma(n1, n2):                          #1
    print('Início do bloco da função')     #2
    s = n1 + n2                            #3
    return s                               #4
    print('Fim do bloco da função')        #5

print('Fora da função!')                   #6
```

Codificação 6.7: "funcao.py" alterado para melhor visualização do fluxo de execução.

Após a execução da Codificação 6.7, o resultado é o ilustrado na Figura 6.2. Novamente, a função `soma` foi definida, mas como ela não foi chamada, seu bloco de código ainda não foi executado. Agora que está carregada na memória, vamos refazer as chamadas à `soma` e observar o resultado, como consta na Codificação 6.8.

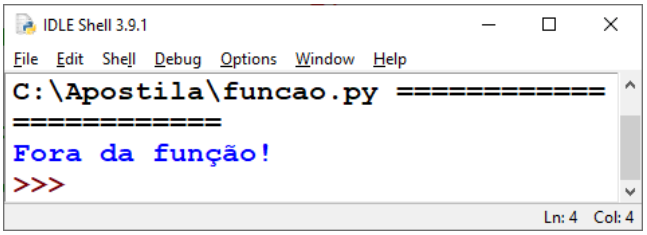

Figura 6.2: Resultado da execução da Codificação 6.7. Fonte: Elaborado pelo autor.

```
>>> soma(3, 5)
Início do bloco da função
8
>>> soma(7, -9)
Início do bloco da função
-2
```

Codificação 6.8: Novas chamadas à função `soma` definida pelo programador.

Notamos que a cada chamada à `soma` o primeiro `print` é executado, mas não o segundo, por quê? A razão é que o segundo `print` está após a instrução `return`, que sempre é executada antes dele. Quando uma função executa um comando `return`, ela retorna o valor da expressão à direita de `return`, se existir, e encerra sua execução, devolvendo o controle do fluxo para à instrução que a chamou. Logo, qualquer instrução no bloco da função posterior à execução de um `return` é *inalcançável*.

VOCÊ SABIA?

Código inalcançável (do inglês *unreachable code*) é uma parte do código-fonte de um programa que jamais será executada, pois o fluxo de execução jamais a alcançará. Geralmente códigos inalcançáveis são produzidos por erros de lógica ou provenientes de código legado. Em geral, queremos evitá-lo, mas existem algumas razões que podem justificar sua existência. Veja mais em: https://en.wikipedia.org/wiki/Unreachable_code

É possível que uma função tenha mais de um `return`, o que pode ser útil caso estejam em blocos de código diferentes como, por exemplo, em uma estrutura de seleção, que a depender de sua condição executará um ou outro `return`, jamais os dois. A função `par` na Codificação 6.9 retorna um valor booleano indicando se `n` é par.

```
def par(n):
    if n % 2 == 0:
        return True
    else:
        return False

print(f'{4} é par? {par(4)}')
print(f'{7} é par? {par(7)}')
```

Codificação 6.9: Exemplo de função com dois comandos `return`.

6.5. Retorno de valor vs. exibição de valor

Na Codificação 6.3, poderíamos ficar tentados a afirmar que a função `soma` exibe o resultado da soma, mas ao olharmos para o seu bloco de código, vemos que ela retorna o valor da variável `s` ao invés de exibi-lo. Nem mesmo há `print` na função!

O retorno da função `soma` só foi exibido devido ao comportamento interativo da *Shell*, que exibe o resultado de qualquer expressão, o que inclui os valores retornados por funções. Para visualizar a diferença, faremos as mesmas chamadas, porém agora no editor do IDLE. Altere novamente o arquivo “funcao.py”, conforme a Codificação 6.10.

O que acontece ao executar a Codificação 6.10? As somas serão exibidas? Tente prever antes de visualizar a resposta na Figura 6.3.

```
def soma(n1, n2):                          #1
    print('Início do bloco da função')  #2
    s = n1 + n2                            #3
    return s                               #4

print('Fora da função!')                   #5
soma(3, 5)                                 #6
soma(7, -9)                                #7
```

Codificação 6.10: Chamadas à função `soma` sem exibição dos valores retornados.

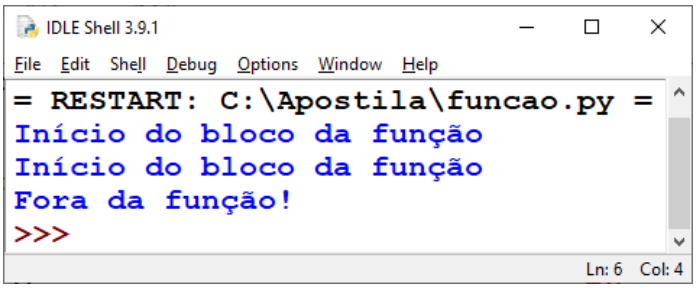

Figura 6.3: Resultado da execução da Codificação 6.10. Fonte: Elaborado pelo autor.

A função retornou, normalmente, um valor a cada chamada, porém esse valor não foi aproveitado para outra operação como, por exemplo, uma exibição. O mesmo ocorre ao criar uma instrução que seja unicamaente uma expressão, como na segunda linha da Codificação 6.11 e que o resultado da execução está ilustrado na Figura 6.4.

```
print('Começo do programa!')
2 * 3 + 4
print('Fim do programa!')
```

Codificação 6.11: Exemplo de expressão avaliada, mas com resultado não aproveitado.

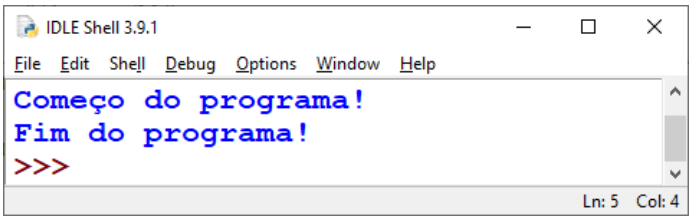

Figura 6.4: Resultado da execução da Codificação 6.11. Fonte: Elaborado pelo autor.

Portanto, temos basicamente duas opções para que o valor final produzido por uma função seja exibido:

1) Exibir o valor retornado pela função na instrução em que ela foi chamada, como na Codificação 6.9, ou atribuir o valor à uma variável e depois exibi-la;
2) Alterar o código da função para que o valor seja exibido e não retornado.

Para entender as opções, faça duas cópias da última versão de “funcao.py” (Codificação 6.10) e renomeie-os para “funcao_return.py” e “funcao_print.py”.

Em “funcao_return.py”, substitua o código após a definição da função pela Codificação 6.12. Execute e veja o resultado na Figura 6.5.

```
print('Fora da função!')                #5
resultado1 = soma(3, 5)                 #6
print('1ª Soma é:', resultado1)         #7
print('2ª Soma é:', soma(7, -9))        #8
```

Codificação 6.12: Atribuição e exibição do valor retornado pela função `soma`.

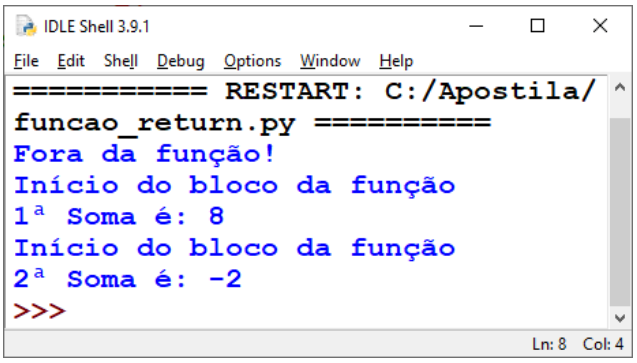

Figura 6.5: Execução do arquivo “funcao_return.py”. Fonte: Elaborado pelo autor.

Em “funcao_print.py”, substitua apenas o código da definição da função pela Codificação 6.13. Execute e veja o resultado na Figura 6.6.

```
def soma(n1, n2):                       #1
    print('Início do bloco da função')  #2
    s = n1 + n2                         #3
    print('A soma é:', s)               #4
```

Codificação 6.13: Alteração da função `soma` que deixa de retornar a soma, apenas a exibe.

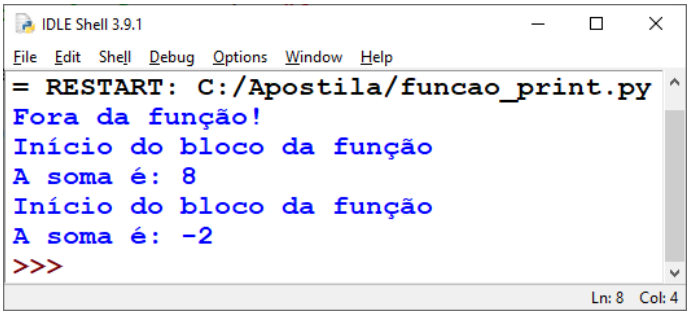

Figura 6.6: Execução do arquivo “funcao_print.py”. Fonte: Elaborado pelo autor.

Após executar o arquivo “funcao_print.py”, chame `soma` na *Shell*, e atribua seu valor de retorno a uma variável, como exemplificado na Codificação 6.14.

```
>>> teste = soma(2, 3)
A soma é: 5
```

Codificação 6.14: Atribuição do valor retornado por uma função sem `return` explícito.

Note que a função foi executada corretamente, inclusive sua instrução `print` cumpriu o papel esperado, exibindo a soma. Porém, há algo estranho na Codificação 6.14. Qual o valor retornado por uma função que nem mesmo tem um `return` explícito? Descubra ao tentar exibir o valor da variável `teste`, como na Codificação 6.15. Repare que não é possível visualizar o conteúdo desta variável sem o uso de `print`.

```
>>> print(teste)
None
```

Codificação 6.15: Valor da variável que recebeu uma atribuição na Codificação 6.14.

Exatamente! Em Python, uma função que não executa um comando `return`, ou que o executa sem um valor à direita para ser retornado, devolve como resposta um valor especial em Python, o `None`. Em Python, esse valor é um indicativo do "nada". Essa é uma característica de Python. Em outras linguagens, tentar atribuir a uma variável o valor de retorno de uma função que não retorna explicitamente algo pode gerar erro.

VOCÊ SABIA?

Algumas linguagens de programação diferenciam funções que retornam valor daquelas que não retornam. Linguagens como C, classificam funções sem valor de retorno como funções do tipo *void* (vazia). A linguagem Pascal, por exemplo, possui *function* e *procedure*, sendo este último algo semelhante à uma função, porém sem valor retornado. Veja mais em: https://en.wikipedia.org/wiki/Void_type

A pergunta final é: "*quando usar `return` e quando usar `print`?*". A resposta é: "*depende*". Depende porque é necessário saber qual o objetivo da função que será criada.

Na maioria das vezes, utilizaremos `return`, pois em geral construímos funções que respondem uma pergunta: "*qual a soma de dois números?*", "*qual a raiz quadrada de um número?*", "*quantos caracteres uma string possui*?", "*este número é par?*". Fazemos uma pergunta e esperamos um valor como resposta, que pode ser um número, um valor booleano, uma *string* ou qualquer outro tipo de dado do Python.

Já o `print` será usado quando dermos uma ordem à função para exibir algo na tela, sem esperar uma resposta, como: "*exiba o menu de opções!*", "*mostre uma mensagem de saudação!*", e assim por diante. Nestes casos não há necessidade de retorno de valor. Esse tipo de função faz parte de uma classe de funções conhecidas como *funções nulas*, que são funções que executam alguma ação (como exibir algo na tela ou outros efeitos) e, em Python, sempre retornam o valor `None`.

Portanto, em Python, toda função retorna um valor: (a) explicitamente, quando está à direita de um comando `return` ou; (b) implicitamente, quando não há `return` ou quando este não está acompanhado de um valor a ser retornado, neste caso retorna-se `None`. Note que há funções que não precisam retornar um valor, o algoritmo não exige, mas por questão de projeto da linguagem Python retornarão automaticamente `None`.

6.6. Escopo de variáveis

De modo simplificado, podemos pensar no escopo como a região do programa na qual a ligação de um identificador com um determinado valor na memória seja válida. Por exemplo, após executar a instrução `n = 2`, o escopo de `n` é toda parte do código na qual podemos acessar exatamente essa variável usando o identificador `n`. Se tentarmos acessar `n` fora do seu escopo, teremos acesso a outro espaço de memória ou um erro de identificador não definido.

Em Python, variáveis criadas na raíz do código, isto é, fora de funções, terão *escopo global*, e serão acessíveis em todo o código-fonte onde foram criadas, a partir de sua instrução de criação. Já variáveis criadas no interior de funções terão *escopo local* e, portanto, acessíveis apenas dentro da função em que foram criadas. Insira a Codificação 6.16 no editor e a execute.

```
def diga_ola():
    print(f'Olá {nome}!') # acessa a variável global

nome = 'Megan'            # variável global
diga_ola()
```

Codificação 6.16: Exemplo de função que acessa uma variável global.

A variável `nome` foi criada fora de qualquer função, então é global, isso significa que quando o Python não encontrar o identificador `nome` no escopo local da função, irá procurá-lo no escopo global. Caso o nome também não esteja no escopo global, será disparado um erro de "nome não definido".

Variáveis locais têm prioridade sobre globais, caso existam duas variáveis como o mesmo identificador a local será a utilizada por padrão, como na Codificação 6.17.

```
def diga_ola(nome):       # parâmetros são variáveis locais
    print(f'Olá {nome}!') # acessa a variável local

nome = 'Megan'            # variável global
diga_ola('Maria')
print(f'Tchau {nome}!')   # acessa a variável global
```

Codificação 6.17: Exemplo de função que acessa sua variável local.

Veja no Python Tutor, ilustrado na Figura 6.7, uma representação da separação na memória do escopo global e do escopo local na execução da Codificação 6.17.

Figura 6.7: Escopo global e local no Python Tutor. Fonte: Elaborado pelo autor.

Qualquer atribuição a variável dentro de uma função gera uma variável local, porém se a intenção é alterar variáveis globais, deve-se indicar com o comando `global` sucedido pelo nome das variáveis entre vírgulas, como na Codificação 6.18.

```
def diga_ola():
    global nome, titulo             # são identificadores globais
    print(f'Olá {titulo} {nome}!') # acessa a variável global
    nome = 'Megan'                  # modifica a variável global
    titulo = 'Sra.'                 # modifica a variável global

titulo = 'Dra.'                     # variável global
nome = 'Maria'                      # variável global
diga_ola()
print(f'Tchau {titulo} {nome}!')
```

Codificação 6.18: Exemplo de função que modifica variáveis globais.

O uso de variáveis globais em funções é desencorajado, pois viola o encapsulamento[3] das funções, dificulta a leitura do código e correções. Note que as funções que usam variáveis globais dependem de recursos que podem ser alterados por outros trechos de código, o que pode gerar caos em programas mais complexos.

Por isso, caso precise informar valores externos às funções, use os parâmetros. Desta forma a função terá os dados necessários para executar seu algoritmo sem violar o encapsulamento.

[3] Simplificadamente, o encapsulamento diz respeito a uma função ocultar seus detalhes de funcionamento interno, de modo que tudo que for necessário para que ela funcione esteja contido nela.

6.7. Texto de documentação de funções

Ao definirmos novas funções, é comum acoplarmos comentários para explicar o funcionamento para quem as usará, que podem ser outros programadores ou nós mesmos. Se feito conforme um padrão, muitos editores de código poderão identificar essa documentação e mostrá-la de modo conveniente, para isso existem as *docstrings*. Vamos criar uma documentação para nossa função `soma`. Altere o arquivo "funcao.py" para que fique conforme a Codificação 6.19, incluindo a *string* de documentação.

```
def soma(n1, n2):
    """Retorna a soma de dois números.

    Parâmetros
    ----------
    n1, n2: int ou float
        Números a serem somados.

    Retorno
    -------
    int ou float
        Resultado da soma de n1 com n2.
    """
    s = n1 + n2
    return s
```

Codificação 6.19: Exemplo de *docstring* em uma função.

A primeira linha é um resumo sobre o que a função faz, em seguida podemos adicionar outras informações sobre a função, como seus parâmetros e valor de retorno. Aqui utilizamos o guia de *docstrings* criado pelo NUMPY (2021). O guia recomenda que tanto o código quanto as *strings* de documentação estejam em inglês, mas como toda recomendação de guia de estilo, o mais importante é manter a coerência interna em um projeto, por isso fizemos o exemplo em português.

Após inserir uma *docstring* na função, podemos visualizar seu conteúdo de algumas formas diferentes, dentre as quais abordaremos duas: (I) digitando o nome da função seguida da abertura de parênteses, tanto na *Shell* quanto no editor, aguardando alguns segundos para a exibição de um balão com a assinatura da função, junto com a primeira linha da *docstring*, como podemos ver na Figura 6.8; (II) usando a função `help`, passando como argumento o nome da função que contém a *docstring*, desta forma a documentação completa será exibida, conforme a Figura 6.9.

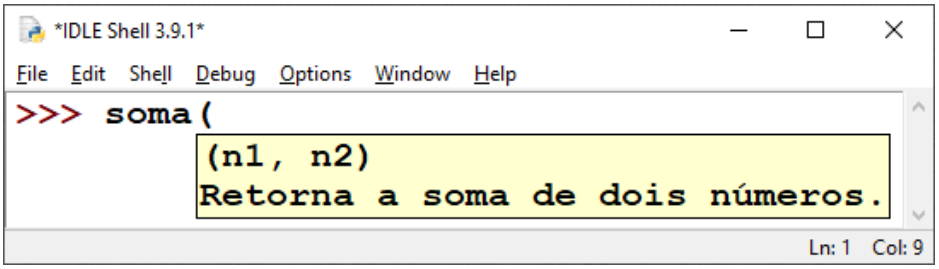

Figura 6.8: Visualização da *docstring* parcial na *Shell*. Fonte: Elaborado pelo autor.

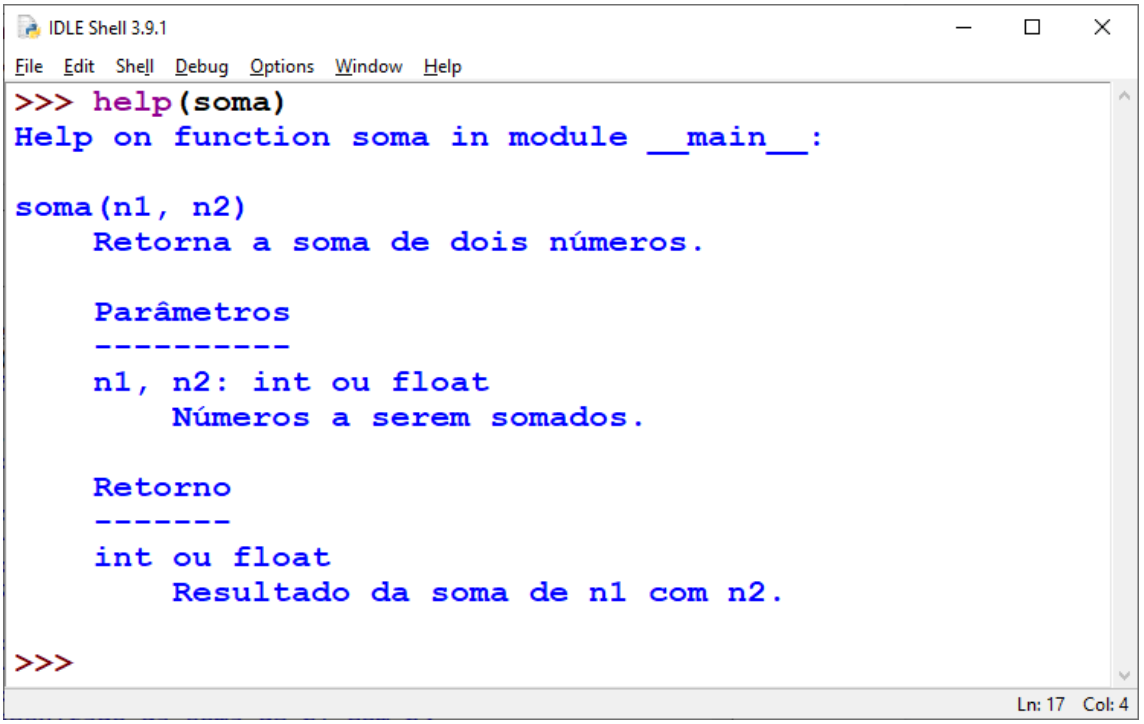

Figura 6.9: Visualização da *docstring* integral na *Shell*. Fonte: Elaborado pelo autor.

6.8. Organização do código-fonte

Aprendemos a importar módulos da biblioteca padrão e a criar nossas próprias funções, agora veremos algumas recomendações de organização para melhorar a legibilidade, rastreamento de erros e manutenção de nossos códigos-fonte:

1) As importações de bibliotecas devem ser feitas no início do código-fonte;
2) Após as importações, se definem as funções;
3) Por último, escreve-se o código principal que chamará as funções e resolverá o problema. É comum denominar o código principal de *programa principal*.

Veja um exemplo de aplicação dessas recomendações na Codificação 6.20.

```
# Importações
import math

# Definição de funções
def dobro(x):
    return 2 * x

def saudacao(nome):
    print(f'Bem-vindo {nome}!')

# Código principal
nome = input('Qual o seu nome? ')
saudacao(nome)
dobro_pi = dobro(math.pi)
print(f'O dobro de {math.pi:.4f} é {dobro_pi:.4f}')
```

Codificação 6.20: Modelo simplificado de organização para códigos-fonte.

VAMOS LER!

Há muito conhecimento relativo às funções em Python e suas implicações práticas, apenas começamos a explorá-lo e estaremos em constante aprendizado.

Para aprender um pouco mais sobre funções, leia o capítulo 3 do livro Pense em Python, disponível em: https://penseallen.github.io/PensePython2e/03-funcoes.html.

E para se aprofundar ainda mais em alguns dos detalhes envolvidos na definição de funções leia: https://realpython.com/defining-your-own-python-function/

Bibliografia e referências

Docstring Guide. **Numpydoc** 2021. Disponível em: <https://numpydoc.readthedocs.io/en/latest/format.html>. Acesso em: 25 jan. 2021.

DOWNEY, A. B. **Pense em Python**. 1 ed. São Paulo: Novatec Editora Ltda., 2016.

PSF. **math: Mathematical Functions**. 2021. Disponível em: <https://docs.python.org/3/library/math.html>. Acesso em: 26 jan. 2021.

STURTZ, J. Defining Your Own Python Functions. **Real Python**, 2020. Disponível em: <https://realpython.com/defining-your-own-python-function/>. Acesso em: 25 jan. 2021.