PYTHON DEVELOPMENT

FUNÇÕES

5

LISTA DE FIGURAS

LISTA DE CÓDIGOS-FONTE

SUMÁRIO

FUNÇÕES

1 É HORA DE MODULARIZAR

1.1 O que são funções e por que é importante criá-las?

Por mais divertido que seja escrever um código e resolver problemas por meio de programação (especialmente com o Python), é uma tarefa que exige um nível considerável de esforço e concentração.

E esse esforço todo é mais bem aproveitado quando podemos, também, aproveitar todo o trabalho que tivemos para criar trechos complexos dos nossos códigos. É justamente esse um dos motivos para criarmos as *funções*.

Uma função nada mais é do que um trecho do seu programa que foi *isolado* dos demais e só será executado quando for convocado.

Um exemplo que podemos usar para tentar entender o que são as funções são as redes de fast food: os funcionários já estão *programados* para executarem a tarefa de *montar um lanche*. Quando você chega no balcão e faz seu pedido, o funcionário não precisa *reescrever* todo o código de fazer lanche, ele apenas *chama* essa função (e faz isso toda vez que um cliente novo chega).

Além de poupar um pouco de esforço com o reuso de códigos, é recomendável usar funções, porque sem elas seria inviável resolvermos diversos problemas computacionais que exigem que a mesma série de passos seja aplicada a todos os valores de um intervalo (lembra como o *map* era interessante?).

Por isso, neste capítulo, você receberá uma das últimas ferramentas na sua jornada de aprendizado básico da linguagem Python!

1.2 Funções sem parâmetros

Para entendermos como criar uma função, vamos partir do script abaixo (que não tem nenhuma função criada):

```
    nome = input("Por favor, informe seu nome: ")

    print(f"Olá, {nome}! É muito bom saber que o Python te
faz feliz!")
```

Código-fonte 1 – Programa sem funções
Fonte: Elaborado pelo autor (2022)

Certamente não se trata de um script simples, mas vamos partir do seguinte pressuposto: se precisássemos exibir a mesma mensagem em outros pontos do nosso projeto e quiséssemos criar uma função para cuidar dessa exibição.

A primeira etapa é entender a sintaxe das funções em Python:

```
    def nome_da_funcao(parâmetros):
        instruções que a função irá executar
```

Código-fonte 2 – Sintaxe da criação de uma função em Python
Fonte: Elaborado pelo autor (2022)

Neste caso, vamos chamar nossa função de *exibe_saudacao()*, e não utilizaremos nenhum parâmetro.

Nosso código ficaria assim:

```
    def exibe_saudacao():
        print(f"Olá, {nome}! É muito bom saber que o Python
te faz feliz!")


    nome = input("Por favor, informe seu nome: ")
```

Código-fonte 3 – Script com função que exibe a saudação
Fonte: Elaborado pelo autor (2022)

Ao executar o script, você deve ter percebido que nossa função não fez nada! Isso ocorreu porque uma função não é executada até que seja chamada; e, para que possamos chamá-la, basta escrever seu nome:

```
    def exibe_saudacao():
        print(f"Olá, {nome}! É muito bom saber que o Python
te faz feliz!")


    nome = input("Por favor, informe seu nome: ")

    exibe_saudacao()
```

Código-fonte 4 – Script com chamada função que exibe a saudação
Fonte: Elaborado pelo autor (2022)

Agora sim, nossa função foi executada e exibiu novamente a mensagem! Por menos impressionante que isso pareça, a partir de agora, sempre que quisermos exibir essa mesma mensagem, basta chamar a função novamente.

Para mostrar um pouco mais do potencial das funções, observe o exemplo a seguir:

```
     #Função que calcula a velocidade média
     def calcular_velocidade_media():
         #calcular a velocidade média
         velocidade_media = distancia/tempo
         #exibir o resultado
         print(f"A velocidade média é {velocidade_media}")

     #aqui começa o programa principal
     #solicitar distância e tempo
     distancia = float(input("Digite a distância percorrida
"))
     tempo = float(input("Digite o tempo da viagem "))
     calcular_velocidade_media()
```

Código-fonte 5 – Script com função que calcula a velocidade média
Fonte: Elaborado pelo autor (2022)

Se partirmos desse mesmo princípio, as possiblidades são infinitas! Podemos criar funções que calculam descontos, que validam o login de um usuário, que formatam textos, que exibem menus... enfim! Há todo um novo horizonte que se abre.

Perceba que a nossa função está usando as variáveis *distância* e *tempo* que só existem no “programa principal” (a parte do código que não está na função), e que se você excluir essas variáveis, a função não funcionará mais. E isso é um problema. Por isso começaremos a pensar em *parâmetros*.

1.3 Funções com parâmetros

Existem funções que podemos criar e que não dependem de nenhum tipo de valor preexistente para funcionarem.

Veja esta função de exemplo, abaixo:

```
    def exibe_menu():
        print("1 - Realizar a soma ")
        print("2 - Realizar a subtração ")
        print("3 - Realizar a multiplicação ")
        print("4 - Realizar a divisão ")
        print("5 - Sair")
```

Código-fonte 6 – Função que exibe as opções de um menu
Fonte: Elaborado pelo autor (2022)

É uma função útil, que pode ser usada toda vez que precisarmos exibir o menu do nosso sistema e *não exige* valores preexistentes para funcionar.

Se voltarmos a pensar em nosso script de cálculo da velocidade média, porém, perceberemos quão é necessário saber de antemão os valores da distância e do tempo para realizar o cálculo. E não nos importa se esses valores serão fornecidos pelo usuário, se virão de outros sistemas ou se estarão escritos diretamente no código. O que importa é que eles existam.

Para esse caso, podemos criar os parâmetros. Parâmetros são dados que uma função recebe quando é chamada e de que precisa para conseguir realizar sua tarefa. Podemos criar os parâmetros em Python indicando entre parênteses na hora da criação da função (como mostramos na sintaxe). No caso da função de velocidade média temos:

```
    #Função que calcula a velocidade média com base em dois
parâmetros fornecidos
    def calcular_velocidade_media(distancia, tempo):
        #calcular a velocidade média
        velocidade_media = distancia/tempo
        #exibir o resultado
        print(f"A velocidade média é {velocidade_media}")

    #aqui começa o programa principal
    #solicitar distância e tempo
    dist_digitada   =   float(input("Digite   a   distância
percorrida "))
    tempo_dititado = float(input("Digite o tempo da viagem
"))
    #Ao chamar a função agora é necessário indicar valores
para os parâmetros distancia e tempo
    calcular_velocidade_media(dist_digitada, tempo_dititado)
```

Código-fonte 7 – Função de velocidade média com parâmetros
Fonte: Elaborado pelo autor (2022)

Apesar de o interpretador do Python não fazer checagem de tipo de parâmetros, é possível incluir uma *dica* que será analisada pela IDE para auxiliá-lo a informar os parâmetros do tipo correto.

Nossa função de velocidade média, por exemplo, funcionará apenas com valores numéricos (e preferencialmente do tipo float). Podemos fazer essa anotação na criação da função para que a IDE nos avise sobre essa tipagem:

```
    #Função que calcula a velocidade média com base em dois
parâmetros fornecidos, com dica sobre o tipo
    def     calcular_velocidade_media(distancia:     float,
tempo:float ):
        #calcular a velocidade média
        velocidade_media = distancia/tempo
        #exibir o resultado
        print(f"A velocidade média é {velocidade_media}")
```

Código-fonte 8 – Função de velocidade média com indicação de tipo
Fonte: Elaborado pelo autor (2022)

Com essa nova função, ao digitar o código a IDE nos sugere fornecer argumentos do tipo float:

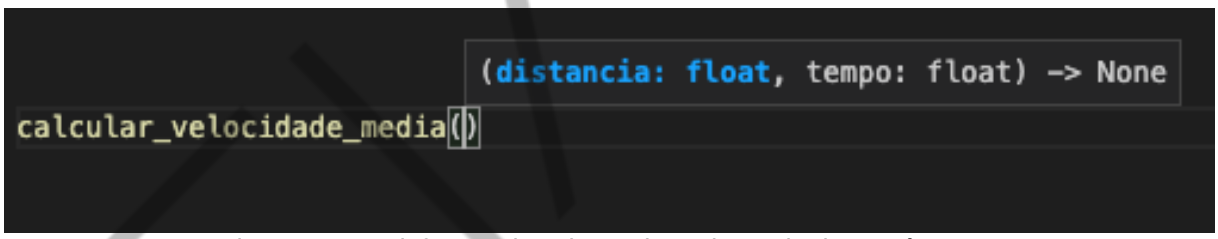

Figura 1 – IDE informando sobre o tipo adequado dos parâmetros
Fonte: Elaborado pelo autor (2022)

Apesar de as anotações de tipo não impactarem no funcionamento do interpretador do Python, utilizá-las facilitará a sua vida na hora de fazer as chamadas da sua função.

1.3.1 Funções com parâmetros padrão

É possível criarmos parâmetros *opcionais* em nossas funções e fazemos isso atribuindo valor padrão.

Veja a versão modificada da nossa função de velocidade média:

```
    def calcularVelocidadeMedia(distancia: float,
tempo:float, unidade_medida="km/h"):
        velocidade_media = distancia/tempo
        print(f"A velocidade média é {velocidade_media}
{unidade_medida}")
```

Código-fonte 9 – Função de velocidade média com parâmetro com valor padrão
Fonte: Elaborado pelo autor (2022)

Note que além de termos incluído um novo argumento, unidade_medida, indicamos um valor para ele “km/h”. Isso quer dizer que ao chamar a função, o desenvolvedor poderá optar por passar um valor para esse parâmetro ou não passar valor algum (nesse caso será assumido o valor padrão).

Observe as duas formas de chamar a função e os resultados obtidos:

```
    def calcular_velocidade_media(distancia: float,
tempo:float, unidade_medida="km/h"):
        velocidade_media = distancia/tempo
        print(f"A velocidade média é {velocidade_media}
{unidade_medida}")


    calcular_velocidade_media(200, 10) #chamada omitindo o
argumento unidade_medida
    calcular_velocidade_media(200, 10, "m/s") #chamada
passando o valor "m/s" para o argumento unidade_media
```

Código-fonte 10 – Chamada da função de velocidade média com parâmetro com valor padrão
Fonte: Elaborado pelo autor (2022)

Utilizar valores padrão em argumentos é uma ótima forma de tornar suas funções mais flexíveis.

1.3.2 Funções com argumentos variáveis (*args)

Imagine que você deseja exibir a mesma mensagem anunciando a promoção de um produto para uma série de clientes da sua empresa. A mensagem será idêntica, mas o nome dos clientes será diferente. Como receber todos os nomes dos clientes em uma função se não sabemos quantos serão?

Para casos em que precisamos trabalhar com um número variável de argumentos em um mesmo parâmetro, utilizamos o símbolo de asterisco na frente do nome do parâmetro, o que comumente é chamado de *args:

```
    def exibe_promocao(*clientes):
        for cliente in clientes:
            print(f"Olá, {cliente}!\nQueremos te avisar que a
nova X-WING está em promoção!")
```

Código-fonte 11 – Função com argumentos variáveis para exibição de mensagem
Fonte: Elaborado pelo autor (2022)

Como a função pode receber uma série de argumentos para o parâmetro **clientes*, podemos iterar por essa estrutura (como apresentado no código anterior). Mas como chamar essa função? Observe o código abaixo:

```
    def exibe_promocao(*clientes):
        for cliente in clientes:
            print(f"Olá, {cliente}!\nQueremos te avisar que a
nova X-WING está em promoção!")

    exibe_promocao("Luke", "Princesa Leia", "Mestre Yoda")
```

Código-fonte 12 – Chamada de função com argumentos variáveis
Fonte: Elaborado pelo autor (2022)

Basta indicarmos os argumentos que gostaríamos de passar na hora de chamar a nossa função e ela funcionará como o esperado.

É pouco provável, porém, que no uso dessa função você tenha os nomes dos clientes escritos diretamente no código. É mais provável que estejam em uma lista, por exemplo. Nesse caso, basta colocarmos um asterisco antes do nome da lista para que o Python entenda que todos os valores que estão nela devem ser colocados no parâmetro clientes:

```
    def exibe_promocao(*clientes):
        for cliente in clientes:
            print(f"Olá, {cliente}!\nQueremos te avisar que a
nova X-WING está em promoção!")
    lista = ["Luke", "Princesa Leia", "Mestre Yoda"]
    exibe_promocao(*lista)
```

Código-fonte 13 – Passagem de lista para argumentos variáveis
Fonte: Elaborado pelo autor (2022)

1.3.3 Funções com argumentos variáveis nomeados (**kwargs)

Além de recebermos argumentos variáveis em um parâmetro, podemos fazer isso de forma *nomeada*. Isso quer dizer que podemos criar um parâmetro que aceita um número variável de argumentos e na hora de chamar a função podemos dar *nomes* a esses argumentos.

Esse tipo de parâmetro pode ser criado indicando dois sinais de asterisco na frente do nome do parâmetro, e são comumente chamados de **kwargs (argumentos com palavra-chave). Observe o exemplo a seguir:

```
    def exibe_saudacao(**cliente):
        print(f"É muito bom ter você como cliente, 
{cliente['nome']} {cliente['sobrenome']}")


    exibe_saudacao(nome="André", sobrenome="David")
```

Código-fonte 14 – Criação e chamada de função que recebe argumentos variáveis nomeados
Fonte: Elaborado pelo autor (2022)

Note que a “mágica” acima aconteceu porque indicamos que o parâmetro ***cliente* poderia receber argumentos nomeados, o que permitiu que na hora da chamada *exibe_saudacao(nome=”André”, sobrenome=”David”)* fosse criado um dicionário no parâmetro *cliente*, contendo as chaves *nome* e *sobrenome*.

Podemos provar isso exibindo o tipo do argumento cliente dentro da função, suas chaves e valores:

```
    def exibe_saudacao(**cliente):
        print(f"O argumento cliente é do tipo 
{type(cliente)}, contendo as chaves {cliente.keys()} e valores 
{cliente.values()}")

        print(f"É muito bom ter você como cliente, 
{cliente['nome']} {cliente['sobrenome']}")


    exibe_saudacao(nome="André", sobrenome="David")
```

Código-fonte 15 – Exibição do tipo do argumento **cliente
Fonte: Elaborado pelo autor (2022)

E se você ficou com a pulga atrás da orelha, se perguntando como é que alguém vai saber quais chaves deve usar, a resposta é justamente que é preciso tomar cuidado com esse tipo de estrutura!

Se contarmos com a boa vontade do usuário da nossa função para passar as *chaves* que queremos, podemos cair em uma armadilha. Por isso, uma boa forma de usar os **kwargs é justamente nas funções em que nos interessa percorrer um dicionário. Veja o caso a seguir:

```
def exibe_ficha(**dados):
    print(dados)
    print("----FICHA----")
    for chave, valor in dados.items():
        print(f"{chave.upper()} -- {valor}")


ficha_cliente = {
    "nome":"Dino da Silva Sauro",
    "estado civil":"casado",
    "camisa":"xadrez",
    "filhos": True
}

exibe_ficha(**ficha_cliente)
```

Código-fonte 16 – Passagem de dicionário para o **kwargs
Fonte: Elaborado pelo autor (2022)

1.4 Funções com return

Todas as nossas funções até agora têm algum tipo de exibição de mensagem no seu código. Essa prática não é proibida e nem errada, mas limita o uso das nossas funções aos casos em que queremos, obrigatoriamente, exibir os dados na tela.

Se pensarmos em funções que realizam cálculos, por exemplo, seria interessante poder aplicá-las também em cenários que não exijam exibir mensagens na tela.

Para cenários como esse, podemos utilizar a cláusula *return* nas nossas funções. Essa cláusula serve para *interromper o funcionamento da função* e *devolver um valor* para quem a chamou.

Pense na função abaixo:

```
def soma(a, b):
    resultado = a + b
    print(resultado)
```

Código-fonte 17 – Função de soma sem return
Fonte: Elaborado pelo autor (2022)

Seria interessante usar essa função para que o resultado pudesse ser armazenado em um banco de dados, escrito em um arquivo de texto, utilizado em outra função ou até mesmo em um print. Portanto vamos modificá-la para que ela não exiba o valor na tela, mas sim retorne esse valor. Da seguinte forma:

```
    def soma(a, b):
        resultado = a + b
        return resultado
```

Código-fonte 18 – Função de soma com return
Fonte: Elaborado pelo autor (2022)

Agora que nossa função não exibe mais uma mensagem na tela, não adianta apenas escrevermos o nome dela para fazermos a chamada: precisamos definir para onde irá o resultado da função. O script abaixo apresenta duas formas diferentes em que essa função poderia ser chamada:

```
    def soma(a, b):
        resultado = a + b
        return resultado

    valor1 = int(input("Informe o prieiro valor que deseja somar "))
    valor2 = int(input("Informe o segundo valor que deseja somar "))

    resposta = soma(valor1, valor2)

    print(f"A variável resposta recebeu o return da função soma() e agora contém: {resposta}")
    print(f"A função soma() está sendo chamada dentro deste print para os valores 10 e 15 e retornou: {soma(10, 15)}")
```

Código-fonte 19 – Função e chamada de soma com return
Fonte: Elaborado pelo autor (2022)

1.5 Um script de funções

Agora que aprendemos a criar funções que recebem ou não recebem argumentos, que retornam ou não retornam dados, é hora de pensarmos em como organizar nosso código.

Para isso, vamos pensar em um cenário no qual queremos criar as seguintes funções:

- Função EXIBIR MENU - exibirá na tela as opções para calcular a velocidade média, converter a temperatura ou sair.
- Função ALUNO DE FÍSICA - exibirá o menu. Essa função deverá também receber a opção selecionada pelo usuário, solicitar os dados necessários

antes da chamada das outras funções e, por fim, chamar as funções corretas.

- CALCULAR VELOCIDADE MÉDIA - uma função que receberá os argumentos distância, tempo, unidade de medida (o padrão será km/h, como já fizemos anteriormente) e retornará uma STRING contendo o resultado com a unidade de medida.
- CONVERTER TEMPERATURA - receberá como argumento uma temperatura e a unidade de medida dessa temperatura. Caso a unidade seja Celsius, deverá converter para Fahrenheit. Caso seja Fahrenheit, deverá converter para Celsius. Retornar o número em formato float.

Nossa primeira função será a de calcular velocidade média, já que ela já estava pronta. Criaremos essa e as outras funções em um arquivo chamado *funções.py*:

```
    def calcular_velocidade_media(distancia:float,
tempo:float, unidade_medida="km/h"):
        if tempo == 0:
            return 0
        velocidade_media = distancia / tempo
        return f"{velocidade_media} {unidade_medida}"
```

Código-fonte 20 – Função calcular_velocidade_media no arquivo funcoes.py
Fonte: Elaborado pelo autor (2022)

A grande diferença dessa função de cálculo de velocidade média para a anterior é que esta retorna uma string em vez de retornar em formato numérico.

Para a construção da função de conversão de temperatura teremos:

```
    def converter_temperatura(temperatura:float,
unidade_medida="celsius"):
        if unidade_medida == "celsius":
            return temperatura * 1.8 + 32
        elif unidade_medida == "fahrenheit":
            return (temperatura - 32) / 1.8
        else:
            return 0
```

Código-fonte 21 – Função converter_temperatura no arquivo funcoes.py
Fonte: Elaborado pelo autor (2022)

Já a função de exibição do menu será simples, uma vez que apresenta apenas uma série de prints:

```
    def exibir_menu():
        print("MENU")
        print("1 - Calcular a velocidade média ")
        print("2 - Converter a temperatura ")
        print("3 - Sair")
```

Código-fonte 22 – Função exibir_menu no arquivo funcoes.py
Fonte: Elaborado pelo autor (2022)

Por fim temos a função *aluno_de_fisica()* que fará a chamada para todas as demais:

```
    def aluno_de_fisica():
        op = 0
        while op != 3:
            exibir_menu()
            op = int(input("Informe a opção desejada: "))
            if op == 1:
                distancia_percorrida = float(input("Informe a
distância: "))
                tempo_viagem = float(input("Iforme o tempo da
viagem: "))
                medida = input("Informe a unidade de medida:
")
                print(f"A      velocidade      média      é
{calcular_velocidade_media(distancia_percorrida,
tempo_viagem, medida)}")

            elif op == 2:
                temperatura_informada = float(input("Informe
a temperatura que deseja converter: "))
                medida = input("A temperatura informada está
em celsius ou fahrenheit")
                print(f"O   resultado   da   conversão   é
{converter_temperatura(temperatura_informada, medida)}")
            elif op == 3:
                print("Saindo do sistema...")
                break
            else:
                print("Opção inválida!")
```

Código-fonte 23 – Função aluno_de_fisica no arquivo funcoes.py
Fonte: Elaborado pelo autor (2022)

Para encerrarmos esta seção com um sabor a mais, vamos fazer a chamada da função aluno_de_fisica em um outro arquivo! Vamos criar um arquivo chamado main.py e dentro dele incluir o seguinte conteúdo:

```
    def converter_temperatura(temperatura:float,
unidade_medida="celsius"):
        if unidade_medida == "celsius":
            return temperatura * 1.8 + 32
        elif unidade_medida == "fahrenheit":
            return (temperatura - 32) / 1.8
        else:
            return 0
```

Código-fonte 24 – Arquivo main.py com a importação e chamada da função aluno_de_fisica
Fonte: Elaborado pelo autor (2022)

Agora você é capaz não apenas de criar as suas funções, mas de organizá-las dentro de diferentes arquivos!

1.6 É bom documentar!

A última coisa que queremos é que um outro desenvolvedor encontre dificuldade para utilizar as funções que criamos, ou até mesmo que no futuro nós não lembremos o que as funções fazem.

Para resolver esse problema, o Python conta com a possibilidade de adicionar um comentário especial nas funções com três aspas, que poderá ser lida através do método __doc__ ou da função help. Veja este caso abaixo:

```
    def somar(a:float, b:float):
        """Recebe dois argumentos, a e b, tipo float e realiza
a soma.
         Retorna o resultado no formato float"""
        return a + b
```

Código-fonte 25 – Sintaxe de criação de um set
Fonte: Elaborado pelo autor (2022)

Agora que criamos o nosso comentário de documentação, os programadores que utilizarem nossa função poderão acessá-lo da seguinte forma:

```
    print(f"Chamando o método doc para ler a documentação:
{somar.__doc__}")
    print(f"Chamando a função help para ler a documentação:
{help(somar)}")
```

Código-fonte 26 – Sintaxe de criação de um set
Fonte: Elaborado pelo autor (2022)

É importante lembrar que o help() é uma função nativa do Python que exibe a documentação em um visualizador de texto. Para encerrar sua execução, será preciso pressionar a tecla “q” no seu teclado.

Estamos quase chegando ao fim da nossa jornada e a próxima parada são os arquivos externos!

REFERÊNCIAS

LUTZ, M. **Learning Python.** Sebastopol: O'Reilly Media Inc, 2013.

PUGA, S.; RISSETI, G**. Lógica de programação e estrutura de dados com aplicações em Java.** São Paulo: Pearson Prentice Hall, 2009.