

# Linguagem de Programação



## 07: FUNÇÕES EM PYTHON

Nossos **objetivos** nesta aula são:

- Explicar o conceito de funções, passagem de parâmetros e funções com e sem retorno;
- Explicar a noção do efeito da chamada de uma função;
- Diferenciar escopo local e global;
- Definir e usar funções em Python;
- Explicar o conceito de argumentos e parâmetros de funções em Python;
- Escrever programas em Python usando funções definidas pelo programador.



A referência para esta aula é o **Capítulo 5 (Funções)** do livro:

DIERBACH, C. *Introduction to Computer Science Using Python: A Computational Problem Solving Focus*. 1st Edition, New York: Wiley, 2012.



### INTRODUÇÃO

Até agora vimos conceitos fundamentais do Python, tais como entrada e saída de dados, variáveis, operadores, expressões e estruturas de controle do fluxo de execução. Do ponto de vista prático, contudo, essas estruturas sozinhas ainda não são suficientes.

Vamos pensar em problemas mais complexos. Por exemplo, um smartphone contém cerca de 10 milhões de linhas de código, imagine o esforço para desenvolver e depurar softwares dessa dimensão.



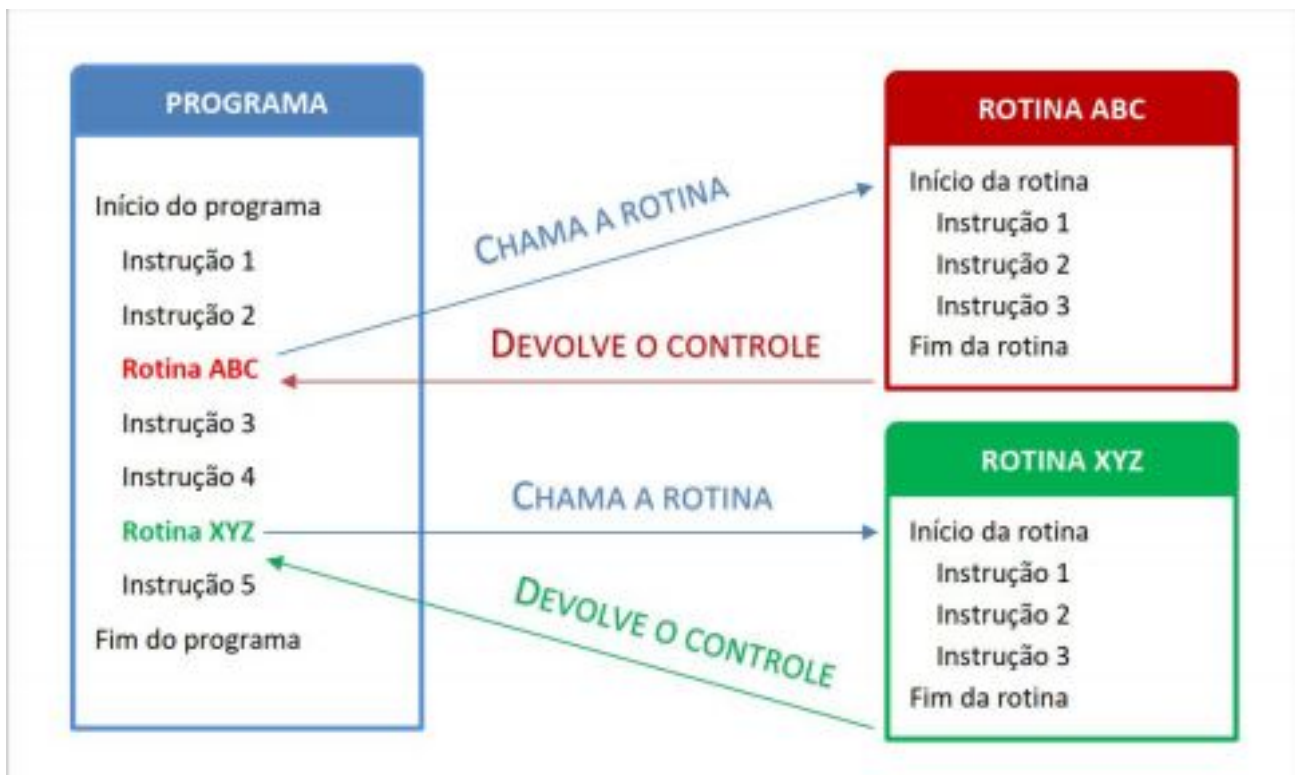
Para gerenciar a complexidade de grandes problemas, é útil quebrá-los em subproblemas menores. Então, cada subproblema pode ser analisado e resolvido separadamente, permitindo soluções mais simples, legíveis e reutilizáveis! Algo importante para não perder tempo reconstruindo códigos.

Nesse sentido, funções são blocos de construção fundamentais no desenvolvimento de software e muito usadas na programação de maneira geral.

## ROTINAS E FUNÇÕES

Uma rotina é definida como um grupo de instruções que executa alguma tarefa bem definida e identificada por um nome. Algoritmos maiores, por exemplo programas completos, podem ser compostos por diversas rotinas que são independentes do programa principal (quanto maior a independência melhor).

A figura a seguir ilustra um exemplo de um programa principal que chama duas rotinas durante sua execução.



Uma rotina pode ser invocada ou chamada quantas vezes for necessário em um dado programa. No momento em que é chamada (a execução do programa chega na linha do código onde está o nome da rotina), o controle de execução é redirecionado para o código da rotina.

Enquanto a rotina está em execução, o código posterior à linha com o nome da rotina fica em estado de espera, pois não é possível seguir em frente enquanto a rotina não terminar sua tarefa.

Quando uma rotina termina de ser executada, o controle da execução retorna automaticamente para o ponto onde foi chamada e o processamento continua com as instruções sucessoras.

As rotinas podem ser pré-definidas da própria linguagem de programação ou construídas pelo programador quando necessário. Existem diversas rotinas prontas em Python, que inclusive já vimos nos capítulos anteriores, como por exemplo, para receber uma entrada do teclado, converter uma string para número e exibir mensagens na tela.

Em Python, a definição ou criação de uma nova função é feita com a palavra chave **def**, da seguinte maneira:

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

Vejamos o exemplo das funções a seguir:

```
# Função 1  
def exibe_media(n1, n2, n3):  
    media = (n1 + n2 + n3) / 3  
    print('A média é:', media)  
  
# Função 2  
def retorna_media(x, y, z):  
    media = (x + y + z) / 3  
    return media
```

A primeira linha de definição de uma função é chamada cabeçalho ou **assinatura**. A assinatura da função inicia-se com a palavra chave **def** seguida por um identificador, que é o nome da função. Na função #1, o identificador é **exibe\_media**, e na função #2 é **retorna\_media**.

O nome da função segue as mesmas regras para criação de nomes de variáveis em Python:

- Pode conter apenas letras, números e o símbolo de sublinhado;
- Não pode começar com um número; e
- Não pode ser uma palavra reservada da linguagem, como por exemplo while, if, for, return, entre outras.<sup>1</sup>

O nome da função é então seguido por um par de parênteses e, entre eles, os identificadores chamados de **parâmetros formais** ou simplesmente parâmetros. Na função #1 os parâmetros são n1, n2, n3, já na função 2, são x, y, z. E por fim, como todos os comandos que criam blocos em Python, finalizamos a linha da assinatura com o símbolo de dois pontos.

Os parâmetros formais são variáveis locais da função que serão criadas no momento em que a função for chamada, sendo os valores a serem atribuídos passados na chamada da função. Assim como variáveis que criamos, os parâmetros devem ter um nome que faça referência ao dado que será armazenado. As funções podem ser construídas com qualquer número de parâmetros, inclusive zero, no caso deixamos os parênteses vazios.

<sup>1</sup> A lista completa pode ser vista em [https://docs.python.org/pt-br/3/reference/lexical\\_analysis.html#keywords](https://docs.python.org/pt-br/3/reference/lexical_analysis.html#keywords)

Os parâmetros servem, essencialmente, para comunicar um dado externo para a função. Por exemplo, para calcular a média de três números, a função precisa saber quais são os três números, por isso os valores são passados como parâmetros para ela.

Após a assinatura da função, escrevemos o bloco de código da função, onde são inseridas as instruções que a compõe. Aqui podemos usar os parâmetros como variáveis existentes no nosso código, pois eles serão criados como variáveis locais no momento em que a função for chamada.

A regra para criação de funções, assim como variáveis, é de que uma função deve ser definida antes de ser chamada, mas para melhorar a organização do código, costumamos definir todas as funções no começo do arquivo.

## SOLICITANDO A EXECUÇÃO DE UMA ROTINA

Ao definirmos uma função, o Python executa apenas a linha da assinatura, criando na memória um objeto que representa a função e contém suas instruções, mas o bloco de código da função não é executado até que a função seja chamada.

Quando chamamos uma rotina ou função, seja no programa principal ou em outra rotina (muitas vezes o programa principal é chamado de rotina-principal), estamos solicitando a execução do bloco de código contido naquela função ou rotina. Para fazer esta chamada, devemos especificar o nome da função e passar os argumentos necessários.

```
>>> exibe_media(4, 4, 1)
A média é: 3.0
```

No exemplo acima, chamamos a função *exibe\_media* com os valores 4, 4, e 1, a função então executou o código contido em seu interior e podemos ver o resultado disso na tela. Quando chamamos a função dessa forma, os valores passados serão associados aos parâmetros da função pela sua posição ou ordem, isto é, o primeiro valor será atribuído ao primeiro parâmetro, o segundo valor ao segundo parâmetro, e assim sucessivamente.

Também é possível passar valores que estejam salvos em alguma variável do nosso código:

```
>>> a = 4
>>> b = 4
>>> c = 1
>>> exibe_media(a, b, c)
A média é: 3.0
```

A seguir, podemos ver um exemplo do código completo que pede os números ao usuário, calcula e exibe a média usando a função *exibe\_media*:

```
# Função de boas-vindas ao usuário.
def bem_vindo():
    print('Bem-vindo! Vamos calcular a média')

# Função que calcula e exibe a média aritmética de três números.
def exibe_media(n1, n2, n3):
    media = (n1 + n2 + n3) / 3
    print('A média é:', media)

# Programa principal.
a = int(input('Primeiro valor: '))
b = int(input('Segundo valor: '))
c = int(input('Terceiro valor: '))

# Dá boas-vindas ao usuário.
bem_vindo()

# Executa a função de exibe_media, que calcula e exibe a média dos
valores passados como parâmetros.
exibe_media(a, b, c)
```

## DIFERENÇA ENTRE return E print

No exemplo dado das funções *exibe\_media* e *retorna\_media*, podemos observar que uma delas faz um print do resultado e a outra retorna o resultado. Mas o que significa retornar o resultado?

Toda função, ao encerrar, retorna (ou devolve) o controle de execução à instrução seguinte àquela de sua chamada. Isso nós já vimos, porém, existe outra característica em funções com alta relevância, que é o retorno (ou devolução) de um valor de resposta.

Em Python, toda função que não tenha um retorno explícito, como é o caso da função *exibe\_media*, ao chegar ao final do seu bloco de código, automaticamente retorna a constante **None**. Essa constante significa literalmente “nada”, e indica que a execução da função terminou e que o programa que a chamou pode continuar sua execução.

Quando queremos fazer com que a função retorne um valor, adicionamos no seu bloco de código

a palavra chave **return**, seguida do valor que queremos retornar, como feito no exemplo da função *retorna\_media*. E ao ler esta instrução, o Python encerra a execução da função e devolve para o código que a chamou o valor passado, sinalizando que a execução do programa que a chamou pode continuar. Vale ressaltar que o que é retornado é sempre um valor, um dado, e não a variável em si.

5

Vejamos o exemplo do código completo para a função *retorna\_media* então:

```
# Função de boas-vindas ao usuário.
def bem_vindo():
    print('Bem-vindo! Vamos calcular a média')

# Função que calcula e retorna a média aritmética de três números.
def retorna_media(x, y, z):
    media = (x + y + z) / 3
    return media

# Programa principal.
a = int(input('Primeiro valor: '))
b = int(input('Segundo valor: '))
c = int(input('Terceiro valor: '))

# Dá boas-vindas ao usuário.
bem_vindo()

# Executa a função de média e armazena o valor retornado.
resposta = retorna_media(a, b, c)

# Exibindo a resposta da função.
print('A média é {:.2f}'.format(resposta))
```

Podemos observar que agora o valor da média não é exibido pela função *retorna\_media*, devemos atribuir o resultado dessa função a uma variável e então exibir esse resultado posteriormente. Isso pode parecer mais trabalhoso a primeira vista, mas separa as responsabilidades de cada parte do código, permitindo que a função *retorna\_media* seja usada em outros contextos, pois ela apenas devolve o resultado, que pode então ser manipulado conforme a necessidade do problema.

Uma função que não retorna valor, geralmente, se limita a ações ou efeitos colaterais, tal como exibir os dados de saída na tela. É muito importante não confundir **retornar** ou devolver um valor com **exibir** algo, esse é um erro comum de iniciantes no uso de funções. Quando um valor é retornado, é possível executar operações sobre ele, como somas, subtrações etc. Quando um valor é somente exibido em uma função, por exemplo, não é possível armazená-lo em uma variável.

Uma forma clara de se acostumar com essa diferença é associar funções que devolvem valor com perguntas, por exemplo: *raiz\_quadrada(n)*. É fácil perceber que para essa função o que se espera é

uma resposta para a pergunta “qual é a raiz quadrada do parâmetro n?”. Note, a resposta não é dada com um `print` dentro da função, mas sim usando o comando **return**.

Já as funções sem retorno de valor podem ser associadas a ordens, por exemplo: *exibe\_idade(ano\_nascimento, ano\_atual)*. Note que a própria intuição ao ler a assinatura da função deixa claro que não se trata de uma pergunta, e sim de uma ordem cuja ação inclusive pode ser deduzida a partir da própria assinatura: calcular e exibir a idade de alguém com base nos argumentos passados para a função, que são o ano de nascimento e o ano atual. Aqui podemos

6

observar também a vantagem de escolhermos nomes descritivos para nossas funções e variáveis, facilitando a leitura do código.

Uma dúvida comum é quando usar cada tipo de função, e a resposta depende da situação. Em um exercício, procure pelos termos *exibe* ou *retorna* e seus sinônimos, já em um ambiente profissional, a experiência irá indicar o formato mais adequado. O mais importante agora é entender a diferença e como isso afeta o resto do código.

## Exemplos

Vamos criar uma função que receba como parâmetros um valor e exiba a mensagem “O resultado é: <valor>”. Essa função deverá devolver um valor ou é apenas uma ação? Perguntamos algo a ela ou demos uma ordem? Aqui estamos dando uma ordem à função “Pegue este valor e exiba-o na tela com a mensagem pedida.”

```
def exibe_resultado(valor):  
    print('O resultado é:', valor)
```

E agora vamos criar uma função que receba dois valores inteiros, referentes ao ano de nascimento de uma pessoa e ao ano atual, calcule e devolva a idade dessa pessoa. Nesse caso, isso parece mais uma ordem ou uma pergunta? Poderíamos trocar o enunciado para “Se estamos no ano X, qual a idade de uma pessoa que nasceu no ano Y?”, portanto aqui estamos fazendo uma pergunta, e usaremos então uma função com retorno de valor.

```
def calcula_idade(ano_nascimento, ano_atual):  
    idade = ano_atual - ano_nascimento  
    return idade
```

Na função *exibe\_resultado* não temos a intenção de capturar o retorno, pois o seu valor não nos interessa (serve apenas para marcar o fim da execução), então podemos dizer que esta é uma função sem retorno de valor, e ao chamá-lo, não vamos atribuir seu resultado a nenhuma variável.

```
>>> resultado = exibe_resultado(10)  
O resultado é: 10  
>>> resultado  
>>>
```

Se atribuirmos o retorno da função *exibe\_resultado* a uma variável, e em seguida tentarmos acessar o valor dessa variável na shell do Python, veremos que não há nenhum valor nela, pois por padrão, a shell não exibe o valor None. Para verificar, podemos fazer um print da variável:

```
>>> print(resultado)
None
```

7

### Para praticar

O que será exibido ao digitarmos cada uma das seguintes instruções na shell do Python (IDLE)?

```
>>> def saudacao(nome):
>>> def diga_ola(): print('Olá!') return 'Olá, ' + nome + '!'
>>> diga_ola >>> saudacao
>>> diga_ola() >>> saudacao('Pedro')
>>> t = diga_ola() >>> r = saudacao('Pedro')
>>> t >>> r
>>> print(t) >>> print(r)
>>> t is None >>> r is None
```

## PASSAGEM DE ARGUMENTOS PARA FUNÇÕES

Vimos que funções podem ser definidas contendo parâmetros e que para chamá-las é preciso escrever o nome da função acompanhado dos argumentos correspondentes aos parâmetros. Vamos ver isso mais detalhadamente agora.

### ARGUMENTOS POSICIONAIS

A correspondência entre os argumentos da chamada e os parâmetros da função pode ser determinada de duas maneiras. O comportamento padrão, que vimos até agora, é chamado **posicional**, pois a passagem dos valores para os parâmetros é determinada pela ordem em que os argumentos são passados.

Considere o exemplo a seguir, no qual uma função recebe como parâmetros um nome e uma senha para atendimento, e deve exibir a frase 'Olá <nome>, sua senha é <senha>'.

```
def recepcao_cliente(nome, senha):
    print('Olá {}, sua senha é {}'.format(nome, senha))
```

Note que os parâmetros são usados em uma ordem específica e se chamarmos a função com os



parâmetros em uma ordem diferente, o resultado será alterado.

```
>>> nome = 'Julia'
>>> senha = 25
>>> recepcao_cliente(nome, senha)
Olá Julia, sua senha é 25
>>> recepcao_cliente(senha, nome)
Olá 25, sua senha é Julia
```

## 8

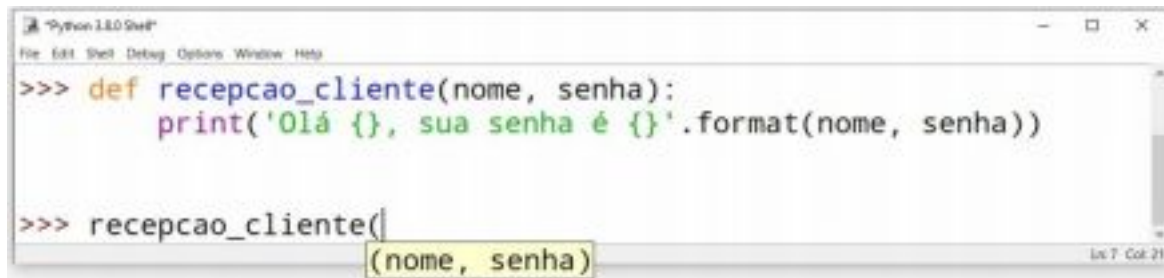
Observe que os valores dos argumentos são copiados para os parâmetros na ordem em que são escritos, independentemente do nome da variável em que estejam salvos, ou seja, o primeiro argumento é associado ao primeiro parâmetro, o segundo argumento é associado ao segundo parâmetro, e assim sucessivamente caso houvesse mais parâmetros. Ou seja, no comportamento padrão do Python, um argumento é associado a um parâmetro particular baseado em sua posição na sequência de argumentos que corresponda à mesma posição do parâmetro na sequência de parâmetros.

### ARGUMENTOS NOMEADOS

A linguagem Python também permite outra forma de passar argumentos para funções, que são os argumentos nomeados, argumentos por palavra chave, em inglês *keyword arguments*. Neste caso, colocamos o nome do parâmetro, um símbolo de igual (sem espaços) e o valor que será passado como argumento para este parâmetro. Agora a ordem não mais importa pois estamos definindo diretamente qual parâmetro irá receber qual argumento.

```
>>> nome = 'Julia'
>>> senha = 25
>>> recepcao_cliente(senha=10, nome='Pedro')
Olá Pedro, sua senha é 10
>>> recepcao_cliente(senha=senha, nome=nome)
Olá Julia, sua senha é 25
>>> recepcao_cliente(senha=nome, nome=senha)
Olá 25, sua senha é Julia
```

Podemos pensar nisso como uma operação de atribuição, mas ao invés de atribuir um valor a uma variável, estamos passando um argumento para um parâmetro específico da função. Para isso precisamos saber exatamente qual o nome que foi definido para este parâmetro no momento de criação da função, portanto a maioria dos editores abre uma janela mostrando algumas informações da função no momento que abrimos o primeiro parênteses, e lá podemos ver os nomes dos parâmetros, veja resultado do teste na IDLE do Python na figura a seguir.

A screenshot of a Python 3.8.0 Shell window. The window has a menu bar with 'File', 'Edit', 'Shell', 'Debug', 'Options', 'Window', and 'Help'. The main area shows a Python script with a function definition: `>>> def recepcao_cliente(nome, senha):` followed by an indented `print('Olá {}, sua senha é {}'.format(nome, senha))`. Below this, there is a function call: `>>> recepcao_cliente(` followed by a yellow-highlighted `(nome, senha)` on the next line. The status bar at the bottom right shows 'Ln 7 Col: 21'.

É possível ainda mesclar a passagem de parâmetros posicionais e nomeados, desde que os parâmetros posicionais venham antes dos nomeados. Caso isso não seja respeitado, o Python irá levantar um erro de sintaxe, como podemos ver a seguir:

```
>>> recepcao_cliente(senha=25, 'Ana')
SyntaxError: positional argument follows keyword argument
```

9

### Para praticar

O que será exibido ao digitarmos cada uma das seguintes instruções na shell do Python (IDLE)?

```
>>> def soma(primeiro, ultimo):
    if primeiro > ultimo:
        s = -1
    else:
        s = 0
        for i in range(primeiro, ultimo + 1):
            s += i
    return s

>>> soma(1, 10)

>>> soma(10, 1)

>>> soma(primeiro=1, ultimo=10)

>>> soma(ultimo=10, primeiro=1)
```

### ARGUMENTOS OPCIONAIS

Há ainda mais um recurso importante para apresentarmos, os parâmetros opcionais. Para criarmos um parâmetro como opcional devemos definir no momento da criação da função os valores padrões que os argumentos opcionais irão assumir quando forem omitidos na chamada da função.

```
def recepcao_cliente(nome, senha=1):
    print('Olá {}, sua senha é {}'.format(nome, senha))
```

Agora ao chamarmos a função acima, podemos fazê-lo com um ou dois parâmetros, e quando o segundo parâmetro for omitido, irá automaticamente receber o valor 1. Note que continuamos podendo usar a passagem de argumentos de maneira posicional, nomeada ou mista, independentemente dos parâmetros terem ou não valores padrão. A existência de um valor padrão apenas nos diz se aquele parâmetro é obrigatório ou opcional para que a função seja chamada corretamente.

```
>>> recepcao_cliente('Rafa')
Olá Rafa, sua senha é 1
>>> recepcao_cliente('Rafa', 23)
Olá Rafa, sua senha é 23
```

A sintaxe do Python não permite que um parâmetro obrigatório seja definido depois de um parâmetro opcional, levantando um erro de sintaxe quando isso acontece.

```
10
>>> def recepcao_cliente(nome='anonimo', senha):
    print('Olá {}, sua senha é {}'.format(nome, senha))
SyntaxError: non-default argument follows default argument
```

### Para praticar

O que será exibido ao digitarmos cada uma das seguintes instruções na shell do Python (IDLE)?

```
>>> def soma(primeiro, ultimo, passo=1):
    if primeiro > ultimo:
        s = -1
    else:
        s = 0
        for i in range(primeiro, ultimo + 1, passo):
            s += i
    return s

>>> soma(1, 10)

>>> soma(1, 10, 2)

>>> soma(primeiro=1, ultimo=10)

>>> soma(passo=3, ultimo=10, primeiro=1)

>>> soma(1, ultimo=10, passo=2)

>>> soma(1, passo=2, ultimo=10)
```

## ESCOPO DE VARIÁVEIS

Em Python, e muitas outras linguagens, temos dois tipos de variáveis: globais e locais. Toda variável que criamos em uma rotina do Python é automaticamente uma variável global, e toda variável que criamos dentro da definição de uma função é uma variável local. Mas qual a diferença e a importância disso?

Uma variável é dita **local** quando é acessível apenas dentro de uma única rotina ou função. Em Python, todas as variáveis que criamos dentro de uma função são locais àquela função, e seu ciclo de vida, isto é, o período em que a variável pode existir na memória, é limitado pela duração de execução da função que a contém.

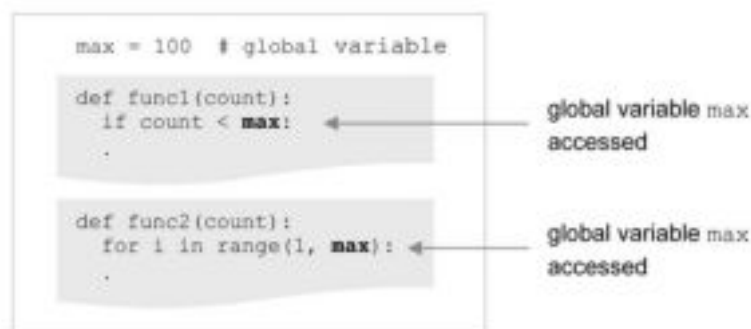
Os parâmetros formais de uma função são criados como variáveis locais no momento em que a função é chamada, e todas as variáveis criadas dentro da função são destruídas quando a execução da rotina em questão é encerrada.

11

Uma variável é dita **global** quando é acessível a todo o código. Em Python, toda variável criada fora de qualquer função será global e seu ciclo de vida estará limitado pela duração de execução do programa.

Todas as funções no mesmo escopo de uma variável global podem ver, acessar e alterar as variáveis globais, portanto passar valores para uma função através do uso de variáveis globais é considerado uma má prática de programação, salvo em raras exceções.

Veja no exemplo a seguir que a variável `max` é definida fora das funções `func1` e `func2`, e portanto é considerada global, sendo acessível por ambas funções.



### Para praticar

- 1) Considere o programa a seguir, que recebe um número inteiro e positivo, validando a entrada e em seguida conta a quantidade de divisores desse número. Reescreva o programa criando 2 funções, para:
  - a) Receber e validar a entrada do usuário; e
  - b) Calcular a quantidade de divisores do número passado como parâmetro.

Escreva também a parte principal do código, que irá chamar as funções na ordem correta para receber a entrada, fazer os cálculos e por fim exibir o resultado.

```
# Programa que exibe a quantidade de divisores de um número
# inteiro positivo

# Validação do dado de entrada.
n = int(input('Digite um número inteiro e positivo: '))
while n <= 0:
    n = int(input('Digite um número inteiro e positivo: '))

# Contagem dos divisores do número lido.
quantidade_divisores = 0
for divisor in range(1, n+1):
    resto = n % divisor
    if resto == 0:
        quantidade_divisores += 1

print('{} tem {} divisores'.format(n, quantidade_divisores))
```

12

- 2) Escreva uma função que receba como parâmetro um número inteiro e positivo  $n$  e retorne um valor booleano indicando se  $n$  é **primo**. Esta função deve usar a função do exercício anterior.
- 3) Escreva uma função que receba como parâmetro dois números inteiros positivos  $a$  e  $b$ , sabendo que  $a \leq b$  e exiba em ordem crescente todos os números primos do intervalo fechado  $[a, b]$ . Use as funções dos exercícios anteriores para verificar se um número é primo ou não.
- 4) Escreva uma função que receba como parâmetro dois números inteiros positivos  $a$  e  $b$ , sabendo que  $a \leq b$  e retorne a soma dos números primos no intervalo fechado  $[a, b]$ . Use as funções dos exercícios anteriores para te ajudar no processo.
- 5) Considere o problema de **conversão de temperatura**. Escreva um programa que permita ao usuário converter uma faixa de temperatura de Fahrenheit para Celsius (o usuário deve digitar 'C') e de Celsius para Fahrenheit (o usuário deve digitar 'F'). Para a construção do programa você deve escrever as seguintes funções:
  - a) `exibe_msg()`: Exibe uma mensagem para o usuário informando a utilidade do programa e quais os dados de entrada. Não há parâmetros e nem retorno de valor.
  - b) `converte_para()`: Exibe uma mensagem para o usuário solicitando a entrada do tipo da conversão, armazena a opção de conversão ('F' ou 'C') e retorna o valor lido.
  - c) `exibe_fahrenheit_to_celsius(start, end)`: Recebe como parâmetros o início do intervalo `start` e o fim do intervalo `end` em Fahrenheit. Cada valor do intervalo deverá ser convertido para Celsius e exibido para o usuário.

d) `exibe_celsius_to_fahrenheit(start, end)`: Recebe como parâmetros o início do intervalo `start` e o fim do intervalo `end` em Celsius. Cada valor do intervalo deverá ser convertido para Fahrenheit e exibido para o usuário.

6) Exibir o calendário mensal de qualquer mês e ano entre **1800 e 2099**.

Representação: o mês e o ano, o número de dias de cada mês, os nomes dos dias da semana, o dia da semana e em qual dia da semana cai o primeiro do mês.

Fevereiro de 2018						
D	S	T	Q	Q	S	S
28	29	30	31	1	2	3
4	5	6	7	8	9	10
11	12	13	14	15	16	17
18	19	20	21	22	23	24
25	26	27	28	1	2	3
4	5	6	7	8	9	10

**1ª. etapa:** Exibir quantos dias há em um mês informado pelo usuário (lembre-se de anos bissextos!)

Para determinar quantos dias tem um determinado mês, é necessária atenção especial a fevereiro, pois possui número variável de dias dependendo de ano ser ou não bissexto. Escreva uma **função** chamada `ano_bissexto(ano)` que retorna um valor booleano `True` ou `False` indicando se ano é bissexto.

Um ano é bissexto quando umas das condições a seguir for verdadeira:

13

- Múltiplo de quatro, mas não de cem;
- Múltiplo de quatrocentos.

Que traduzindo para o Python, temos:

```
(ano % 4 == 0 and ano % 100 != 0) or (ano % 400 == 0)
```

Devido à precedência dos operadores, os parênteses são desnecessários aqui, mas podemos colocá-los para facilitar a leitura do código.

Crie uma **função** chamada `dias_mes(mes, ano)` que retorna um valor natural indicando a quantidade de dias do `mes` em `ano`. Esta função deve usar a função que verifica se o ano é bissexto.

Considere que o usuário poderá repetir a inserção dos dados indefinidas vezes, o processo terminará apenas quando o valor -1 for inserido para mês (*flag* de encerramento). Faça a interação com o usuário por meio de uma função chamada `main()` que solicitará os valores de mês e ano.

**2ª. etapa:** Determinar o dia da semana de uma data informada.

Escreva uma **função** chamada **dia\_semana** que recebe os parâmetros **dia**, **mes** e **ano** (todos números naturais) e retorna uma string com o dia da semana correspondente à data informada, seguindo o seguinte algoritmo:

a. Seja a variável **digitos\_seculo** igual aos dois primeiros dígitos do ano; b. Seja a variável **digitos\_ano** igual aos dois últimos dígitos do ano; c. Seja a variável **valor** igual a **digitos\_ano + (digitos\_seculo // 4)**; d. Se **digitos\_seculo** for igual a 18 então adicione 2 a **valor**. Caso contrário, se **digitos\_seculo** for igual a 20 adicione 6 a **valor**.

e. Ajustar a variável **valor** de acordo com o mês fornecido, da seguinte maneira: i. Se **mes** for igual a **janeiro** e o ano não for bissexto, adicione 1 a **valor**. Caso contrário,

ii. Se **mes** for igual a **fevereiro** e o ano for bissexto, adicione 3 a **valor**. Caso contrário,

iii. Se **mes** for igual a **fevereiro** e o ano não for bissexto adicione 4 a **valor**. Caso contrário,

iv. Se **mes** for igual a **março** ou **novembro** adicione 4 a **valor**. Caso contrário,

v. Se **mes** for igual a **maio** adicione 2 a **valor**. Caso contrário,

vi. Se **mes** for igual a **junho** adicione 5 a **valor**. Caso contrário,

vii. Se **mes** for igual a **agosto** adicione 3 a **valor**. Caso contrário,

viii. Se **mes** for igual a **outubro** adicione 1 a **valor**. Caso contrário,

ix. Se **mes** for igual a **setembro** ou **dezembro** adicione 6 a **valor**.

f. A variável **valor** é baseada no dia do mês. Para obter o dia da semana, fazemos:

i.  $\text{valor} = (\text{value} + \text{dia}) \% 7$

14

g. Converta o resultado obtido em valor para a string do dia da semana usando a seguinte tabela como referência:

**1 2 3 4 5 6 0**

Domingo Segunda Terça Quarta Quinta Sexta Sábado

Obs.: A entrada de dados deve contemplar uma mensagem de erro caso o usuário digite valores inválidos.

### **EXERCÍCIOS PROPOSTOS**

- 1) É possível chamar uma função a partir de qualquer parte de um programa? (considere que a função existe)
- 2) É possível colocar mais de uma chamada de função em uma mesma expressão?
- 3) Podemos ter uma chamada de função cujos argumentos passados sejam também chamadas de funções?

4) As afirmações a seguir são verdadeiras?

a) Toda função que retorna um valor deve conter pelo menos um parâmetro. b) Uma variável global em Python é uma variável que é definida fora de qualquer definição de função.

c) Em Python, não existem variáveis locais, todas as variáveis são sempre globais, inclusive as que são criadas dentro da definição de uma função.

5) Considerando a definição da função **minha\_funcao** e sua posterior chamada no programa principal, identifique os parâmetros formais e os argumentos atuais (valores passados para a função).

```
def minha_funcao(n1, n2, n3)
    ...

a = 3
minha_funcao(a, 8, a+5)
```

6) Considerando que a função **mdc** a seguir funciona apenas se os argumentos passados obedecerem às suas restrições:

```
# Função que calcula o maior divisor comum de n1 e n2.
# Restrição: n1 e n2 são inteiros e n1 <= n2.
def mdc(n1, n2)
    ...
```

Indique se as chamadas da função estão ou não apropriadas e justifique sua

resposta. 15

			(A) (B) (C)
>>> a = 10	>>> a = 10.0	>>> a = 10	
>>> b = 20	>>> b = 20	>>> b = '20'	
>>> mdc(a, b)	>>> mdc(a, b)	>>> mdc(a, b)	
			(D) (E) (F)
>>> a = 10	>>> a = 10	>>> a = 10	
>>> b = 20	>>> b = '20'	>>> b = '20'	
>>> c = 30	>>> c = 30	>>> c = 30	
>>> mdc(a + 2*b, c)	>>> mdc(a, c - int(b))	>>> mdc(a + c, int(b))	

7) Faça um programa que calcule e exiba o fatorial de um número natural n fornecido pelo usuário. O programa deverá conter a função fatorial(n) que devolve o fatorial de n.

Exemplo:  $5! = 5*4*3*2*1*0! = 120$ , sendo  $0! = 1$  por definição.

8) Dado um número inteiro positivo, escreva um programa modularizado para calcular a soma de seus dígitos. O seu programa deve conter uma função que leia um número inteiro e



positivo e retorna-o, bem como outra função chamada **soma\_digitos** que recebe um inteiro positivo e retorna a soma de seus dígitos.

- 9) A série de Fibonacci é formada por uma sequência de número em que cada termo é calculado como a soma dos dois termos anteriores, resultando em :

1, 1, 2, 3, 5, 8, 13, 21, 34, ...

Escreva um programa modularizado que apresente a série de Fibonacci até o n-ésimo termo, sendo  $n > 0$ . O seu programa deve conter uma função que leia um número inteiro positivo e retorna-o, bem como outra função chamada **fibonacci**, que recebe um inteiro positivo **n** e retorna o n-ésimo termo da sequência.

- 10) Escreva um programa modularizado que calcule o MMC (Mínimo Múltiplo Comum) entre dois números naturais. O seu programa deve conter uma função que leia um número natural e o retorna, bem como outra função chamada **mmc** que recebe dois números naturais e retorna o MMC entre eles.

- 11) Escreva um programa modularizado que calcule o MDC (Máximo Divisor Comum) entre dois números naturais. O seu programa deve conter uma função que leia um número natural e o retorna, bem como outra função chamada **mdc** que recebe dois números naturais e retorna o MDC entre eles.

## **TAREFA MÍNIMA**

### **Exercício 1 - Resolvido**

Elabore um programa com uma função `area_triangulo` que peça ao usuário a base e altura de triângulo, calcule e exiba a sua área.

#### **RESOLUÇÃO ✓**

```
def area_triangulo():
    base = float(input('Digite a base: '))
    altura = float(input('Digite a altura: '))
    area = base * altura / 2
    print('A área do triângulo é: {:.2f}'.format(area))

area_triangulo()
```

## Exercício 2

Elabore um programa com uma função que peça ao usuário o peso e a altura de uma pessoa, calcule e exiba o seu IMC, calculado como o peso dividido pelo quadrado da altura.

## Exercício 3 - Resolvido

Elabore um programa peça ao usuário a base e altura de triângulo, calcule e exiba a sua área, usando uma função `area_triangulo`, que deve receber a base e altura do triângulo e exibir o valor da área calculado.

### RESOLUÇÃO ✓

```
def area_triangulo(base, altura):
    area = base * altura / 2
    print('A área do triângulo é: {:.2f}'.format(area))

base = float(input('Digite a base: '))
altura = float(input('Digite a altura: '))
area_triangulo(base, altura)
```

## Exercício 4

Elabore um programa que peça ao usuário o peso e a altura de uma pessoa, calcule e exiba o seu IMC, definindo e utilizando a função `imc`, que irá receber o peso e a altura como parâmetros, calcular e exibir o resultado. Novamente, o IMC é calculado como o peso dividido pelo quadrado da altura.

17

## Exercício 5 - Resolvido

Elabore um fluxograma com uma função que recebe como parametro a base e altura de triangulo e calcula e retorna a sua área.

### RESOLUÇÃO ✓

```
def area_triangulo(base, altura):
    area = base * altura / 2
    return area

base = float(input('Digite a base: '))
altura = float(input('Digite a altura: '))
area = area_triangulo(base, altura)
print('A área do triângulo é: {:.2f}'.format(area))
```

## Exercício 6

Elabore um programa que peça ao usuário o peso e a altura de uma pessoa, calcule e exiba o seu IMC, definindo e utilizando a função **imc**, que irá receber o peso e a altura como parâmetros, calcular e retornar o resultado. Novamente, o IMC é calculado como o peso dividido pelo quadrado da altura.

18

## RESPOSTAS DA TAREFA MÍNIMA

### Exercício 2

```
def imc():
    peso = float(input('Digite o peso: '))
    altura = float(input('Digite a altura: '))
    indice = peso / altura**2
    print('O IMC calculado é {:.1f}'.format(indice))
    imc()
```

### Exercício 4

```
def imc(peso, altura):
```

```
indice = peso / altura**2
print('O IMC calculado é {:.1f}'.format(indice))

peso = float(input('Digite o peso: '))
altura = float(input('Digite a altura: ')) imc(peso, altura)
```

### **Exercício 6**

```
def imc(peso, altura):
    indice = peso / altura**2
    return indice

peso = float(input('Digite o peso: '))
altura = float(input('Digite a altura: ')) indice = imc(peso, altura)
print('O IMC calculado é {:.1f}'.format(indice)) 19
```