**Introdução da aula**



**Qual é o foco da aula?**

Nesta aula, vamos conhecer as funções em Python e o modo como aplicá-las a fim de facilitar a leitura e a manutenção das soluções propostas.

**Objetivos gerais de aprendizagem**

Ao longo desta aula, você irá:

* identificar a necessidade das funções na programação;
* examinar a utilização das funções em *Python*
* esclarecer as funções anônimas.

**Situação-problema**

O desenvolvimento de um software pode envolver diversas pessoas e até mesmo diferentes equipes. Orquestrar todos os envolvidos com as etapas do projeto é um desafio em qualquer empresa. Para ajudar nessa missão, a engenharia de software fornece diversas ferramentas, que especificam padrões de projeto e de trabalho. A metodologia ágil é uma dessas ferramentas, que habilita a entrega de partes do software para o cliente, por exemplo, em cada sprint (intervalo de tempo, que normalmente dura 15 dias) uma nova funcionalidade ou melhoria é entregue. Para conseguir essa agilidade na entrega, pessoas com diferentes expertise (conhecimento) são colocadas para trabalhar juntas, por exemplo, um design trabalha no layout da interface gráfica, um desenvolvedor trabalha nas funcionalidades front-end, enquanto outro trabalha no back-end e podemos ainda ter um DBA (Data Base Administrator) cuidando das conexões com o banco de dados.

Dando continuidade ao seu trabalho na empresa de consultora de software, o cliente que fabrica peças automotivas requisitou uma nova funcionalidade para o sistema: calcular o total de vendas. Como seus negócios estão expandindo, o cliente solicitou que o sistema seja capaz de receber valores em reais, dolár ou euro, e que seja feita a conversão para o valor em reais. Como essa entrega demanda a construção de uma interface gráfica, além da comunicação com banco de dados, dentre outros requisitos técnicos, você foi alocado em uma equipe ágil para criar a função que fará o cálculo do valor.

Após uma primeira reunião, a equipe fez um levantamento de requisitos e concluiu que a função a ser construída precisa considerar os seguintes itens:

* o valor do produto (obrigatório).
* a quantidade do produto (obrigatório).
* a moeda em que deve ser feito o cálculo (obrigatório, sendo padrão o real).
* a porcentagem do desconto que será concedida na compra (opcional).
* a porcentagem de acréscimo, que depende da forma de pagamento (opcional).

Uma conta não pode ter desconto e acréscimo ao mesmo tempo. Nessa primeira versão, você deve considerar o valor do dólar em R$ 5,00 e do euro 5,70. Ainda não foram definidos os detalhes de como os dados serão capturados e tratados após serem digitados e submetidos. Porém, você deve entregar a versão inicial da função, para que a equipe comece a fazer os primeiros testes.

Qual nome você dará para a função? Como você especificará os parâmetros que a função pode receber? A função retornará um valor ou somente imprimirá na tela o resultado? Hora de colocar a mão na massa e implementar a solução.

**As funções**



O assunto desta aula será funções. Por que precisamos aprender essa técnica de programação? Segundo Banin (2018, p. 119),

"... funções constituem um elemento de fundamental importância na moderna programação de computadores, a ponto de ser possível afirmar que atualmente nenhum programa de computador é desenvolvido sem o uso desse recurso".

Para entendermos, com clareza, o que é uma função, vamos pensar na organização de um escritório de advocacia. Nesse escritório, existe um secretário que possui a função de receber os clientes e agendar horários. Também trabalham nesse escritório três advogados, que possuem a função de orientar e representar seus clientes com base nas leis.

Não podemos deixar de mencionar os colaboradores que possuem a função de limpar o escritório e fazer reparos. Mas o que o escritório tem a ver com nossas funções? Tudo! Citamos algumas funções que podem ser encontradas nesse ambiente de trabalho, ou seja, um conjunto de tarefas/ações associada a um "nome". Podemos, então, resumir que uma função é uma forma de organização, usada para delimitar ou determinar quais tarefas podem ser realizadas por uma determinada divisão.

Essa ideia de organização em funções é o que se aplica na programação. Poderíamos implementar uma solução em um grande bloco de código, nesse caso, teríamos um cenário quase impossível de dar manutenção. Em vez de escrever dessa forma, podemos criar uma solução dividindo-a em funções (blocos), além de ser uma boa prática de programação, tal abordagem facilita a leitura, a manutenção e a escalabilidade da solução.

**Funções Built-in em Python**



Desde que escrevemos nossa primeira linha de código nessa disciplina >>print("hello world"), já começamos a usar funções, pois print() é uma função *built-in*do interpretador *Python*.

Uma função*built-in* é um objeto que está integrado ao núcleo do interpretador, ou seja, não precisa ser feita nenhuma instalação adicional, já está pronto para uso. O interpretador *Python*possui várias funções disponíveis, veja o quadro a seguir.

Funções built-in em Python. Fonte: Docs.Python.

Ao observar o quadro abaixo, podemos identificar algumas funções que já usamos:

* print(), para imprimir um valor na tela.
* enumerate(), para retornar a posição de um valor em uma sequência.
* input(), para capturar um valor digitado no teclado.
* int() e float(), para converter um valor no tipo inteiro ou float.
* range(), para criar uma sequência numérica.
* type(), para saber qual o tipo de um objeto (variável).

Ao longo da disciplina, iremos conhecer várias outras, mas certamente vale a pena você acessar a [documentação](https://docs.python.org/3/library/functions.html" \t "_blank) e explorar tais funções, aumentando cada vez mais seu repertório. Para mostrar o poder das funções e deixar um gostinho de quero mais, veja o código a seguir, com a utilização da função eval().

**In [1]**

a **=** 2

b **=** 1

equacao **=** input("Digite a fórmula geral da equação linear (a \* x + b): ")

print(f"\nA entrada do usuário {equacao} é do tipo {type(equacao)}")

for x in range(5):

    y **=** eval(equacao)

    print(f"\nResultado da equação para x = {x} é {y}")

Digite a fórmual geral da equação linear (a \* x + b): a \* x + b  
  
A entrada do usuário a \* x + b é do tipo <class 'str'>  
  
Resultado da equação para x = 0 é 1  
  
Resultado da equação para x = 1 é 3  
  
Resultado da equação para x = 2 é 5  
  
Resultado da equação para x = 3 é 7  
  
Resultado da equação para x = 4 é 9

A função eval() usada no código recebe como entrada uma string (sequência de caracteres) digitada pelo usuário, que nesse caso é uma equação linear. Essa entrada é analisada e avaliada como uma expressão Python pela função eval(). Veja que, para cada valor de x, a fórmula é executada como uma expressão matemática (linha 8) e retorna um valor diferente.

A função eval() foi mostrada a fim de exemplificar a variedade de funcionalidades que as funções*built-in*possuem. Entretanto, precisamos ressaltar que eval é uma instrução que requer prudência para o uso, pois é fácil alguém externo à aplicação fazer uma "*injection*" de código intruso.

Utilize o [emulador](https://trinket.io/python3/402eca8dbc" \t "_blank) a seguir, para testar o código e experimentar novas funções*built-in* em *Python*.

**Função definida pelo usuário**



*Python*possui 70 funções *built-in*(quadro abaixo), que facilitam o trabalho do desenvolvedor, porém cada solução é única e exige implementações específicas. Diante desse cenário, surge a necessidade de implementar nossas próprias funções, ou seja, trechos de códigos que fazem uma determinada ação e que nós, como desenvolvedores, podemos escolher o nome da função, sua entrada e sua saída.

Vamos começar a desenvolver nossas funções, entendendo a sintaxe de uma função em Python. Observe o código a seguir.

**In [2]**

def nome\_funcao():

    # bloco de comandos

Veja que na linha 1 da entrada 2 (In [2]), usamos o comando "def" para indicar que vamos definir uma função. Em seguida, escolhemos o nome da função "imprimir\_mensagem", veja que não possui acento, nem espaço, conforme recomenda a [PEP 8](https://www.python.org/dev/peps/pep-0008/" \l "function-and-variable-names): os nomes das funções devem estar em minúsculas, com as palavras separadas por underline, conforme necessário, para melhorar a legibilidade.

Os nomes de variáveis seguem a mesma convenção que os nomes de funções. Nomes com *mixedCase* (mistura de maiúsculas e minúsculas) são permitidos apenas em contextos em que o nome já existe com o formato recomendado.

Em toda declaração de função, após especificar o nome, é preciso abrir e fechar parênteses, pois é dentro dos parênteses que os parâmetros de entrada da função devem ser definidos. Nessa versão da nossa função "imprimir\_mensagem", não existe nenhum parâmetro sendo passado como entrada. A linha 2 representa o conjunto de ações que essa função deve executar, no caso, somente imprimir uma mensagem na tela. Para que uma função execute suas ações, ela precisa ser "invocada", fazemos isso na linha 5, colocando o nome da função, acompanhada dos parênteses.

Agora vamos criar uma segunda versão da nossa função "imprimir\_mensagem". Observe o código a seguir:

**In [3]**

def imprimir\_mensagem(disciplina, curso):

    print(f"Minha primeira função em Python desenvolvida na disciplina: {disciplina}, do curso: {curso}.")

imprimir\_mensagem("Python", "ADS")

Minha primeira função em Python desenvolvida na disciplina: Python, do curso: ADS.

Veja na linha 1 da entrada 3 (In [3]), que agora a função recebe dois parâmetros. Esses parâmetros são variáveis locais, ou seja, são variáveis que existem somente dentro da função. Na linha 2, imprimimos uma mensagem usando as variáveis passadas como parâmetro e na linha 5, invocamos a função, passando como parâmetros dois valores do tipo string. O valor "*Python*" vai para o primeiro parâmetro da função e o valor "ADS" vai para o segundo parâmetro.

O que acontece se tentarmos atribuir o resultado da função "imprimir\_mensagem" em uma variável, por exemplo:

resultado = imprimir\_mensagem("Python", "ADS")? Como a função "imprimir\_mensagem" não possui retorno, a variável "resultado" receberá "None". Teste o código a seguir e veja o resultado:

**In [4]**

def imprimir\_mensagem(disciplina, curso):

    print(f"Minha primeira função em Python desenvolvida na disciplina: {disciplina}, do curso: {curso}.")

resultado **=** imprimir\_mensagem("Python", "ADS")

print(f"Resultado = {resultado}")

Minha primeira função em Python desenvolvida na disciplina: Python, do curso: ADS.  
  
Resultado = None

Para que o resultado de uma função possa ser guardado em uma variável, a função precisa ter o comando "return". A instrução "return", retorna um valor de uma função. Veja a nova versão da função "imprimir\_mensagem", agora, em vez de imprimir a mensagem, ela retorna a mensagem para chamada.

**In [5]**

def imprimir\_mensagem(disciplina, curso):

    return f"Minha primeira função em Python desenvolvida na disciplina: {disciplina}, do curso: {curso}."

mensagem **=** imprimir\_mensagem("Python", "ADS")

print(mensagem)

Minha primeira função em Python desenvolvida na disciplina: Python, do curso: ADS.

Veja na linha 2 da entrada 5 (In [5]) que, em vez de imprimir a mensagem, a função retorna (return) um valor para quem a invocou. O uso do "return" depende da solução e das ações que se deseja para a função.

Por exemplo, podemos criar uma função que limpa os campos de um formulário, esse trecho pode simplesmente limpar os campos e não retornar nada, mas também pode retornar um valor booleano como *True*, para informar que a limpeza aconteceu com sucesso. Portanto, o retorno deve ser analisado, levando em consideração o que se espera da função e como se pretende tratar o retorno, quando necessário.

\_\_\_\_\_

📝 **Exemplificando**

Vamos implementar uma função que recebe uma data no formato dd/mm/aaaa e converte o mês para extenso. Então, ao se receber a data 10/05/2020, a função deverá retornar: 10 de maio de 2020. Observe a implementação a seguir.

**In [6]**

def converter\_mes\_para\_extenso(data):

    mes **=** '''janeiro fevereiro março

      abril maio junho julho agosto

      setembro outubro novembro

      dezembro'''**.**split()

    d, m, a **=** data**.**split('/')

    mes\_extenso **=** mes[int(m) **-** 1] # O mês 1, estará na posição 0!

    return f'{d} de {mes\_extenso} de {a}'

print(converter\_mes\_para\_extenso('10/05/2021'))

10 de maio de 2021

* Linha 1 - Definimos o nome da função e os parâmetros que ela recebe.
* Linha 2 - Criamos uma lista com os meses, por extenso. Veja que criamos uma string e usamos a função split(), que "quebra" a string a cada espaço em branco, criando uma lista e elementos.
* Linha 6 - Usamos novamente a função split(), mas agora passando como parâmetro o caractere '/', isso quer dizer que a string será cortada sempre que ele aparecer. Nessa linha também usamos a atribuição múltipla. Ao cortar a string 'dd/mm/aaaa', temos uma lista com três elementos: ['dd', 'mm', 'aaaa'], ao usar a atribuição múltipla, cada valor da lista é guardado dentro de uma variável, na ordem em que foram declaradas. Então d = 'dd', m = 'mm', a = 'aaaa'. O número de variáveis tem que ser adequado ao tamanho da lista, senão ocorrerá um erro.
* Linha 7 - Aqui estamos fazendo a conversão do mês para extenso. Veja que buscamos na lista "mes" a posição m - 1, pois, a posição inicia em 0. Por exemplo, para o mês 5, o valor "maio", estará na quarta posição a lista "mes".
* Linha 8 - Retornamos a mensagem, agora com o mês em extenso.
* Linha 10 - Invocamos a função, passando como parâmetro a data a ser convertida.

Aproveite o [emulador](https://trinket.io/python3/d5f556b911" \t "_blank) para testar o código e implementar outras funções.

**Funções com parâmetros definidos e indefinidos**



Sobre os argumentos que uma função pode receber, para nosso estudo, vamos classificar em seis grupos:

1. parâmetro posicional, obrigatório, sem valor default (padrão).
2. parâmetro posicional, obrigatório, com valor default (padrão).
3. parâmetro nominal, obrigatório, sem valor default (padrão).
4. parâmetro nominal, obrigatório, com valor default (padrão).
5. parâmetro posicional e não obrigatório (args).
6. parâmetro nominal e não obrigatório (kwargs).

No grupo 1, temos os parâmetros que vão depender da ordem em que forem passados, por isso são chamados de posicionais (a posição influencia o resultado). Os parâmetros desse grupo são obrigatórios, ou seja, tentar um invocar a função, sem passar os parâmetros, acarreta um erro. Além disso, os parâmetros não possuem valor default. Observe a função "somar" a seguir.

**In [7]**

def somar(a, b):

    return a **+** b

r **=** somar(2)

print(r)

**---------------------------------------------------------------------------**

**TypeError**                                 Traceback (most recent call last)

**<ipython-input-7-4456bbb97a27>** in <module>

      2     **return** a **+** b

      3

**----> 4**r **=** somar**(2)**

      5 print**(**r**)**

**TypeError**: somar() missing 1 required positional argument: 'b'

A função "somar" na entrada 7 (In [7]) foi definida de modo a receber dois parâmetros, porém na linha 4, quando ela foi invocada, somente um parâmetro foi passado, o que resultou no erro "missing 1 required positional argument", que traduzindo quer dizer: "falta 1 argumento posicional obrigatório". Para que a função execute sem problema, ela deve ser invocada passando os dois argumentos, por exemplo: r = somar(2, 3).

No grupo 2, também temos os parâmetros posicionais e obrigatórios, porém vamos definir um valor default (padrão), assim, quando a função for invocada, caso nenhum valor seja passado, o valor default é utilizado. Observe a função "calcular\_desconto" a seguir.

**In [8]**

def calcular\_desconto(valor, desconto**=**0): # O parâmetro desconto possui zero valor default

    valor\_com\_desconto **=** valor **-** (valor **\*** desconto)

    return valor\_com\_desconto

valor1 **=** calcular\_desconto(100) # Não aplicar nenhum desconto

valor2 **=** calcular\_desconto(100, 0.25) # Aplicar desconto de 25%

print(f"\nPrimeiro valor a ser pago = {valor1}")

print(f"\nSegundo valor a ser pago = {valor2}")

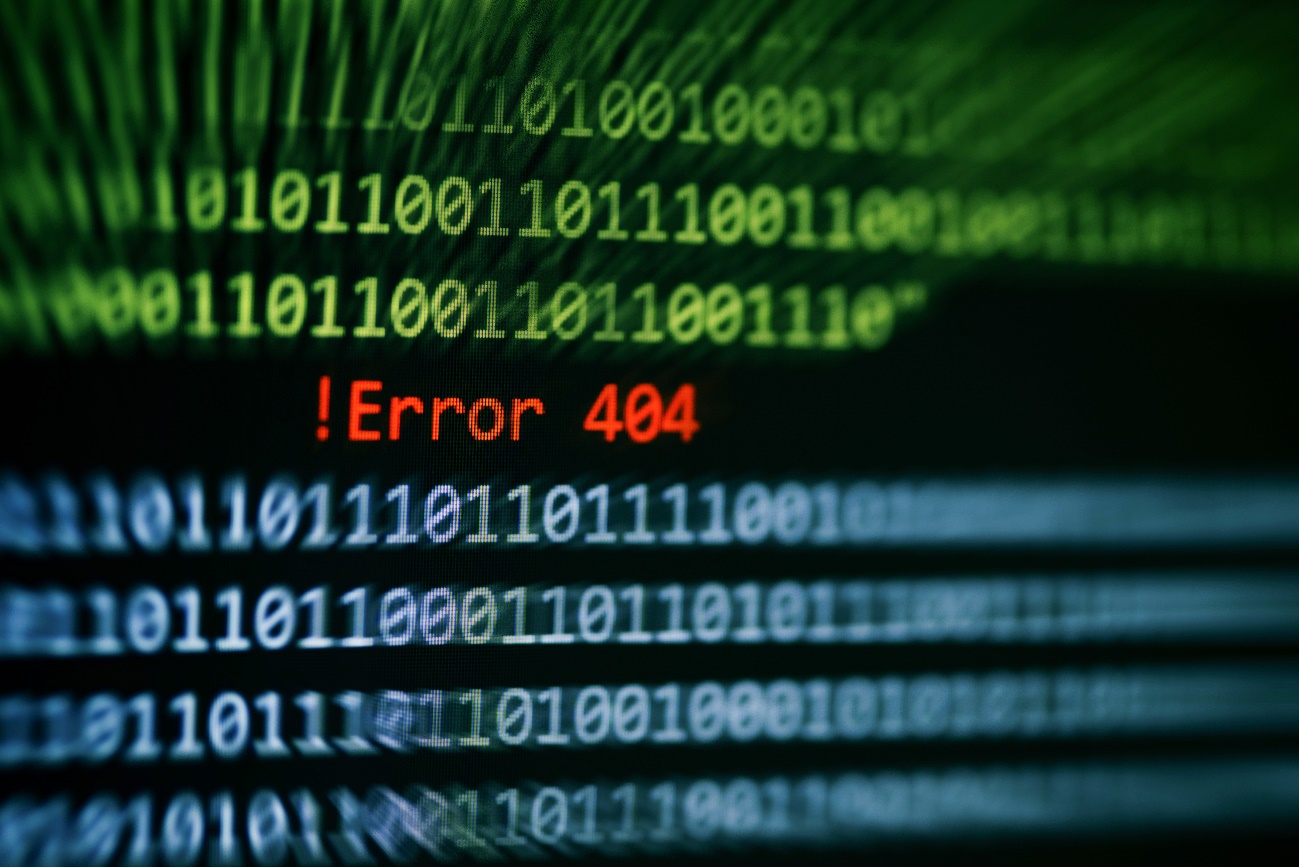
Primeiro valor a ser pago = 100

Segundo valor a ser pago = 75.0

Primeiro valor a ser pago = 100  
  
  
Segundo valor a ser pago = 75.0

A função "calcular\_desconto" na entrada 8 (In [8]) foi definida de modo a receber dois parâmetros: "valor" e "desconto". O parâmetro "valor" não possui valor default, já o parâmetro "desconto" possui zero como valor default, ou seja, se a função for invocada e o segundo parâmetro não for passado, será usado o valor padrão definido para o argumento. Veja que, na linha 5, a função é invocada, sem passar o segundo argumento e não causa erro, pois existe o valor default. Já na linha 6 a função é invocada passando tanto o valor quanto o desconto a ser aplicado.

**Os erros**



A obrigatoriedade do argumento, quando não atendida, pode resultar em um erro, conforme vimos na entrada 7 (In [7]). Porém, para o conceito de parâmetros posicionais não existe nenhum erro de interpretação associado, ou seja, o interpretador não irá informar o erro, mas pode haver um erro de lógica. Observe o código a seguir:

**In [9]**

def cadastrar\_pessoa(nome, idade, cidade):

    print("\nDados a serem cadastrados:")

    print(f"Nome: {nome}")

    print(f"Idade: {idade}")

    print(f"Cidade: {cidade}")

cadastrar\_pessoa("João", 23, "São Paulo")

cadastrar\_pessoa("São Paulo", "João", 23)

Dados a serem cadastrados:  
  
Nome: João  
  
Idade: 23  
  
Cidade: São Paulo  
  
  
  
Dados a serem cadastrados:  
  
Nome: São Paulo  
  
Idade: João  
  
Cidade: 23

A função "cadastrar\_pessoa" na entrada 9 (In [9]) foi definida de modo a receber três parâmetros: "nome", "idade" e "cidade". Observe a chamada da função da linha 8, foram passados os argumentos: "João", 23, "São Paulo". O primeiro valor, "João", foi atribuído ao primeiro parâmetro na função, "nome". O segundo valor, 23, foi atribuído ao segundo parâmetro na função, "idade". O terceiro valor, "São Paulo", foi atribuído ao terceiro parâmetro na função, "cidade", portanto o resultado foi exatamente o que esperávamos.

Agora observe a chamada na linha 9, foram passados os argumentos: "São Paulo", "João", 23. O primeiro valor, "São Paulo", foi atribuído ao primeiro parâmetro na função, "nome". O segundo valor, "João", foi atribuído ao segundo parâmetro na função, "idade". O terceiro valor, 23, foi atribuído ao terceiro parâmetro na função "cidade", tais atribuições implicam um erro lógico, pois os dados não foram atribuídos às variáveis corretas.

Com o exemplo da função "cadastrar\_pessoa", fica claro como a posição dos argumentos, na hora de chamar a função, deve ser conhecida e respeitada, pois a passagem dos valores na posição incorreta pode acarretar erros lógicos.

O grupo de parâmetros 3 é caracterizado por ter parâmetros nominais, ou seja, agora não mais importa a posição dos parâmetros, pois eles serão identificados pelo nome, os parâmetros são obrigatórios, ou seja, na chamada da função é obrigatório passar todos os valores e sem valor default (padrão). Observe a função "converter\_maiuscula".

**In [10]**

def converter\_maiuscula(texto, flag\_maiuscula):

    if flag\_maiuscula:

        return texto**.**upper()

    else:

        return texto**.**lower()

texto **=** converter\_maiuscula(flag\_maiuscula**=**True, texto**=**"João") # Passagem nominal de parâmetros

print(texto)

JOÃO

A função "converter\_maiuscula" na entrada 10 (In [10]) foi definida de modo a receber dois parâmetros: "texto" e "flag\_maiuscula". Caso "flag\_maiuscula" seja True, a função deve converter o texto recebido em letras maiúsculas, com a função built-in upper(), caso contrário, em minúsculas, com a função built-in lower(). Como a função "converter\_maiuscula" não possui valores default para os parâmetros, então a função deve ser invocada passando ambos valores.

Agora observe a chamada na linha 8, primeiro foi passado o valor da flag\_maiuscula e depois o texto. Por que não houve um erro lógico? Isso acontece porque a chamada foi feita de modo nominal, ou seja, atribuindo os valores às variáveis da função e, nesse caso, a atribuição não é feita de modo posicional.

O grupo de funções da categoria 4 é similar ao grupo 3: parâmetro nominal, obrigatório, mas nesse grupo os parâmetros podem possuir valor default (padrão). Observe a função "converter\_minuscula" a seguir.

**In [11]**

def converter\_minuscula(texto, flag\_minuscula**=**True): # O parâmetro flag\_minuscula possui True como valor default

    if flag\_minuscula:

        return texto**.**lower()

    else:

        return texto**.**upper()

texto1 **=** converter\_minuscula(flag\_minuscula**=**True, texto**=**"LINGUAGEM de Programação")

texto2 **=** converter\_minuscula(texto**=**"LINGUAGEM de Programação")

print(f"\nTexto 1 = {texto1}")

print(f"\nTexto 2 = {texto2}")

Texto 1 = linguagem de programação  
  
Texto 2 = linguagem de programação

A função "converter\_minuscula" na entrada 11 (In [11]) foi definida de modo a receber dois parâmetros, porém um deles possui valor default. O parâmetro flag\_minuscula, caso não seja passado na chamada da função, receberá o valor *True*.

Veja a chamada da linha 8, passamos ambos os parâmetros, mas na chamada da linha 9, passamos somente o texto. Para ambas as chamadas o resultado foi o mesmo, devido o valor *default*atribuído na função. Se não quiséssemos o comportamento *default*, aí sim precisaríamos passar o parâmetro, por exemplo: texto = converter\_minuscula(flag\_minuscula=False,texto="LINGUAGEM de Programação").

Até o momento, para todas as funções que criamos, sabemos exatamente o número de parâmetros que ela recebe. Mas existem casos em que esses parâmetros podem ser arbitrários, ou seja, a função poderá receber um número diferente de parâmetros a cada invocação. Esse cenário é o que carateriza os grupos 5 e 6 de funções que vamos estudar.

No grupo 5, temos parâmetros posicionais indefinidos, ou seja, a passagem de valores é feita de modo posicial, porém a quantidade não é conhecida. Observe a função "obter\_parametros" a seguir.

**In [12]**

def imprimir\_parametros(**\***args):

    qtde\_parametros **=** len(args)

    print(f"Quantidade de parâmetros = {qtde\_parametros}")

    for i, valor in enumerate(args):

        print(f"Posição = {i}, valor = {valor}")

print("\nChamada 1")

imprimir\_parametros("São Paulo", 10, 23.78, "João")

print("\nChamada 2")

imprimir\_parametros(10, "São Paulo")

Chamada 1  
  
Quantidade de parâmetros = 4  
  
Posição = 0, valor = São Paulo  
  
Posição = 1, valor = 10  
  
Posição = 2, valor = 23.78  
  
Posição = 3, valor = João  
  
  
Chamada 2  
  
Quantidade de parâmetros = 2  
  
Posição = 0, valor = 10  
  
Posição = 1, valor = São Paulo

A função "imprimir\_parametros" na entrada 12 (In [12]) foi definida de modo a obter parâmetros arbitrários. Tal construção é feita, passando como parâmetro o \*args. O parâmetro não precisa ser chamado de args, mas é uma boa prática. Já o asterisco antes do parâmetro é obrigatório.

Na linha 2, usamos a função *built-in*len() (length) para saber a quantidade de parâmetros que foram passados. Como se trata de parâmetros posicionais não definidos, conseguimos acessar a posição e o valor do argumento, usando a estrutura de repetição for e a função enumerate(). Agora observe as chamadas feitas nas linhas 10 e 13, cada uma com uma quantidade diferente de argumentos, mas veja na saída que os argumentos seguem a ordem posicional, ou seja, o primeiro vai para a posição 0, o segundo para a 1 e assim por diante.

[Teste o passo a passo de execução do código](http://pythontutor.com/). Clique em "next" para visualizar a execução de cada linha de código.

O último grupo de funções possui parâmetro nominal e não obrigatório. O mecanismo é parecido com as do grupo 5, porém agora a passagem é feita de modo nominal e não posicional, o que nos permite acessar tanto o valor do parâmetro quanto o nome da variável que o armazena. Veja a função "imprimir\_parametros" a seguir:

**In [13]**

def imprimir\_parametros(**\*\***kwargs):

    print(f"Tipo de objeto recebido = {type(kwargs)}\n")

    qtde\_parametros **=** len(kwargs)

    print(f"Quantidade de parâmetros = {qtde\_parametros}")

    for chave, valor in kwargs**.**items():

        print(f"variável = {chave}, valor = {valor}")

print("\nChamada 1")

imprimir\_parametros(cidade**=**"São Paulo", idade**=**33, nome**=**"João")

print("\nChamada 2")

imprimir\_parametros(desconto**=**10, valor**=**100)

Chamada 1  
  
Tipo de objeto recebido = <class 'dict'>  
  
  
  
  
Quantidade de parâmetros = 3  
  
variável = cidade, valor = São Paulo  
  
variável = idade, valor = 33  
  
variável = nome, valor = João  
  
  
  
  
Chamada 2  
  
Tipo de objeto recebido = <class 'dict'>  
  
  
  
  
Quantidade de parâmetros = 2  
  
variável = desconto, valor = 10  
  
variável = valor, valor = 100

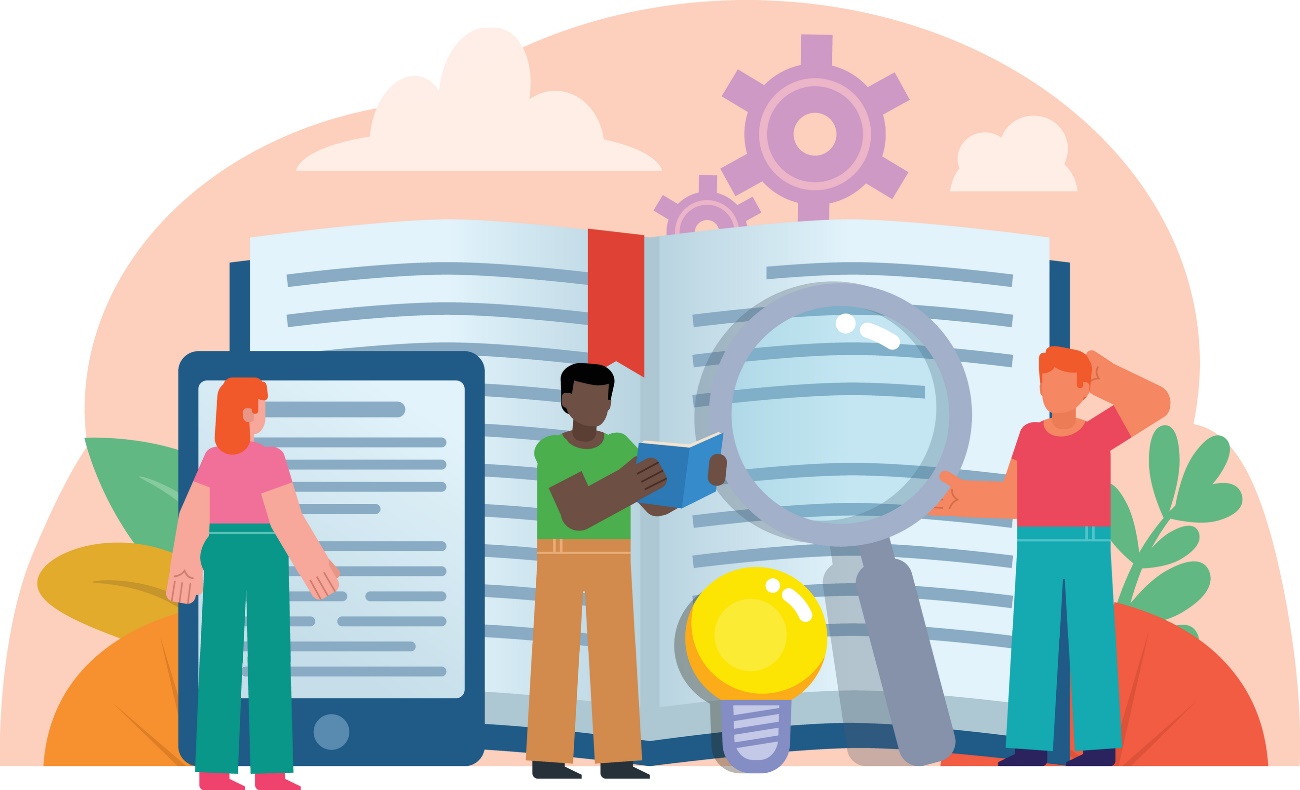
A função "imprimir\_parametros" na entrada 13 (In [13]) foi definida de modo a obter parâmetros nominais arbitrários. Tal construção é feita, passando como parâmetro o \*\*kwargs. O parâmetro não precisa ser chamado de kwargs, mas é uma boa prática. Já os dois asteriscos antes do parâmetro é obrigatório.

Na linha 2, estamos imprimindo o tipo de objeto recebido, você pode ver na saída que é um dict (dicionário), o qual estudaremos nas próximas aulas. Na linha 3, usamos a função *built-in* len() (length) para saber a quantidade de parâmetros que foram passados. Como se trata de parâmetros nominais não definidos, conseguimos acessar o nome da variável em que estão atribuídos o valor e o próprio valor do argumento, usando a estrutura de repetição "for" e a função items() na linha 17.

A função items não é built-in, ela pertence aos objetos do tipo dicionário, por isso a chamada é feita como "kwargs.items()" (ou seja, objeto.função). Agora observe as chamadas feitas nas linhas 11 e 14, cada uma com uma quantidade diferente de argumentos, mas veja na saída que os argumentos estão associados ao nome da variável que foi passado.

Utilize o [emulador](https://trinket.io/python3/402eca8dbc" \t "_blank) a seguir para testar os códigos e criar novas funções para explorar as diferentes formas de construir uma função e passar parâmetros.

**Funções anônimas em Python**



Já que estamos falando sobre funções, não podemos deixar de mencionar um poderoso recurso da linguagem *Python*: a expressão "lambda". Expressões *[lambda](https://docs.python.org/3/reference/expressions.html" \l "lambda" \t "_blank)*(às vezes chamadas de formas lambda) são usadas para criar funções anônimas. Uma função anônima é uma função que não é construída com o "def" e, por isso, não possui nome. Esse tipo de construção é útil, quando a função faz somente uma ação e é usada uma única vez. Observe o código a seguir:

**In [14]**(lambda x: x **+** 1)(x**=**3)

Out[ ]: 4

Na entrada 14 (In [14]), criamos uma função que recebe como parâmetro um valor e retorna esse valor somado a 1. Para criar essa função anônima, usamos a palavra reservada "lambda" passando como parâmetro "x". O dois pontos é o que separa a definição da função anônima da sua ação, veja que após os dois pontos, é feito o cálculo matemático x + 1. Na frente da função, já a invocamos passando como parâmetro o valor x=3, veja que o resultado é portanto 4.

A função anônima pode ser construída para receber mais de um parâmetro. Observe o código a seguir:

**In [15]**

(lambda x, y: x **+** y)(x**=**3, y**=**2)

Out[ ]: 5

Na entrada 15 (In [15]), criamos uma função anônima que recebe como parâmetro dois valores e retorna a soma deles. Para criar essa função anônima, usamos a palavra reservada "lambda" passando como parâmetro "x, y". Após os dois pontos, é feito o cálculo matemático x + y. Na frente da função, já a invocamos passando como parâmetro o valor x=3 e y=2, veja que o resultado é portanto 5.

A linguagem *Python*, nos permite atribuir a uma variável uma função anônima, dessa forma, para invocar a função, fazemos a chamada da variável. Observe o código a seguir:

**In [16]**

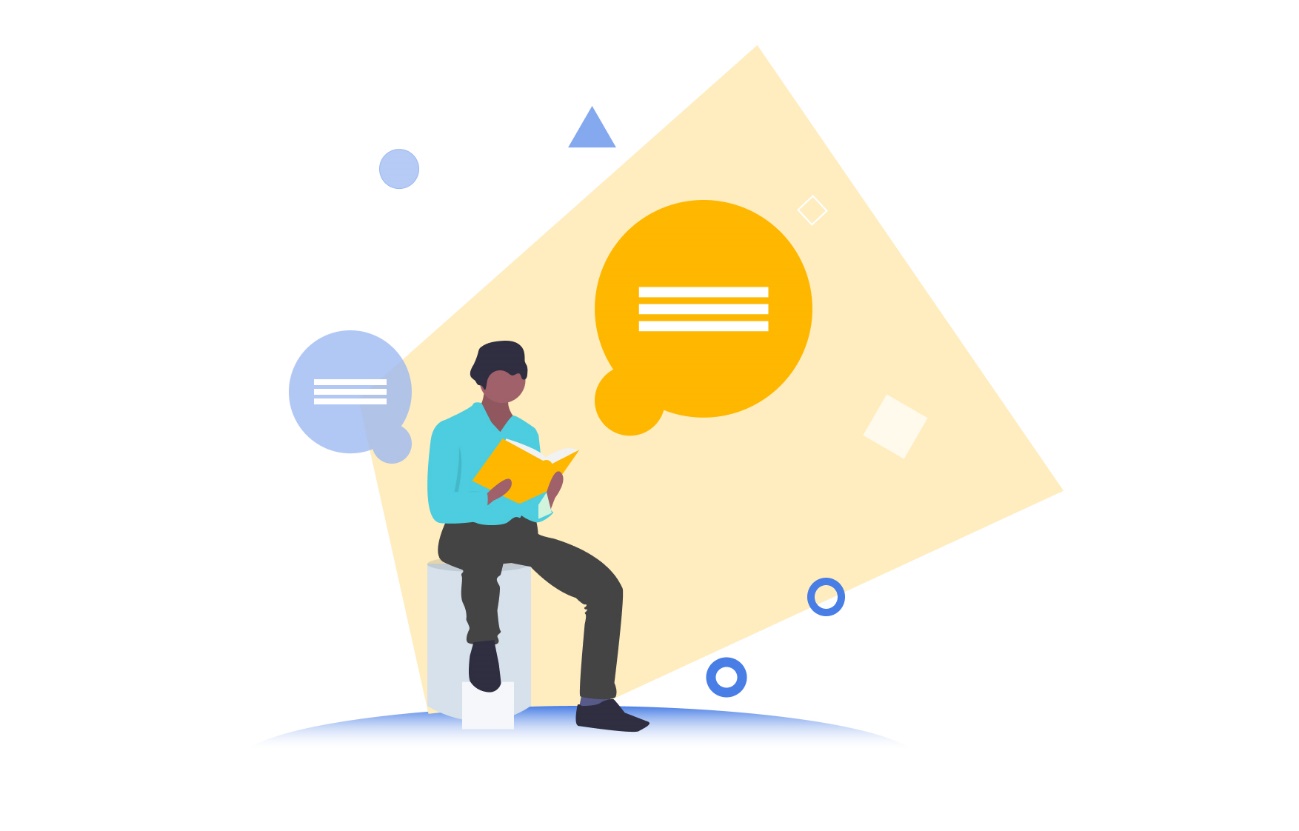
somar **=** lambda x, y: x **+** y

somar(x**=**5, y**=**3)

**Out[ ]:**8

Na entrada 16 (In [16]), criamos uma função anônima que recebe como parâmetro dois valores e retorna a soma deles, essa função foi atribuída a uma variável chamada "somar". Veja que na linha 2, fazemos a chamada da função através do nome da variável, passando os parâmetros que ela requer.

**Conclusão**



Em seu novo projeto, você foi designado a implementar uma solução que envolve o cálculo de uma compra. Para esse cálculo existem parâmetros que são obrigatórios e outros opcionais, portanto a função deve ser capaz de lidar com esse tipo de dinâmica. Observe uma possível implementação a seguir.

**In [1]**

def calcular\_valor(valor\_prod, qtde, moeda**=**"real", desconto**=**None, acrescimo**=**None):

    v\_bruto **=** valor\_prod **\*** qtde

    if desconto:

        v\_liquido **=** v\_bruto **-** (v\_bruto **\*** (desconto **/** 100))

    elif acrescimo:

        v\_liquido **=** v\_bruto **+** (v\_bruto **\*** (acrescimo **/** 100))

    else:

        v\_liquido **=** v\_bruto

    if moeda **==** 'real':

        return v\_liquido

    elif moeda **==** 'dolar':

        return v\_liquido **\*** 5

    elif moeda **==** 'euro':

        return v\_liquido **\*** 5.7

    else:

        print("Moeda não cadastrada!")

        return 0

valor\_a\_pagar **=** calcular\_valor(valor\_prod**=**32, qtde**=**5, desconto**=**5)

print(f"O valor final da conta é {valor\_a\_pagar}")

O valor final da conta é 152.0

Sobre a solução proposta, observe o nome da função "calcular\_valor", veja que estamos seguindo as recomendações de nomenclatura, usando somente letras em minúsculo e com o underline separando as palavras. Outro detalhe é a utilização do verbo no infinitivo "calcular", toda função executa ações, por isso, é uma boa prática escolher verbos infinitos.

A função foi desenvolvida de modo a receber cinco parâmetros, sendo três deles obrigatórios e dois opcionais. Nessa implementação, para construir os parâmetros opcionais atribuímos o valor *None*às variáveis, nesse caso, como tem um valor padrão, mesmo sendo None, a função pode ser invocada sem a passagem desses parâmetros.

Usamos as estruturas condicionais (if) para verificar se foram passados valores para desconto ou acréscimo, caso tenha valores, serão diferentes de None e, então, os devidos cálculos são realizados. Após os cálculos de desconto, é feito o teste para saber qual moeda foi usada na compra e fazer a conversão para real.

Muitas vezes, uma solução pode ser implementada de diferentes formas. Observe no código a seguir uma outra implementação, usando o \*\*kwargs para os argumentos opcionais. Nesse caso, um dicionário é recebido e precisa ter o valor extraído. Na próxima aula, você aprenderá esse tipo de objeto.

**In [2]**

def calcular\_valor(valor\_prod, qtde, moeda**=**"real", **\*\***kwargs):

    v\_bruto **=** valor\_prod **\*** qtde

    if 'desconto' in kwargs:

        desconto **=** kwargs['desconto']

        v\_liquido **=** v\_bruto **-** (v\_bruto **\*** (desconto **/** 100))

    elif 'acrescimo' in kwargs:

        acrescimo **=** kwargs['acrescimo']

        v\_liquido **=** v\_bruto **+** (v\_bruto **\*** (acrescimo **/** 100))

    else:

        v\_liquido **=** v\_bruto

    if moeda **==** 'real':

        return v\_liquido

    elif moeda **==** 'dolar':

        return v\_liquido **\*** 5

    elif moeda **==** 'euro':

        return v\_liquido **\*** 5.7

    else:

        print("Moeda não cadastrada!")

        return 0

valor\_a\_pagar **=** calcular\_valor(valor\_prod**=**32, qtde**=**5, desconto**=**5)

print(f"O valor final da conta é {valor\_a\_pagar}")

O valor final da conta é 152.0

\_\_\_\_\_

**➕ Pesquise mais**

Ganhar habilidade em programação exige estudo e treino (muito treino). Acesse a biblioteca virtual no e busque pelo livro:  **Estudo Dirigido de Algoritmos**. Na página 209, do capítulo 12 da referida obra: Sub-rotinas do tipo função, você encontrará o exemplo 3, que te desafia a criar uma função "calculadora".

Dependendo do parâmetro que a função receber, uma operação deve ser feita. O livro discute a solução usando fluxograma e português estruturado, utilize o emulador e faça a implementação usando a linguagem Python.

**Referências**



A ORIGEM do “Hello World”. **Ciência da Computação**, 2015. Disponível em: https://cienciacomputacao.com.br/curiosidade/a-origem-do-hello-world/. Acesso em: 13 jun. 2021.

ANACONDA Individual Edition: The World's Most Popular Python/R Data Science Platform. The World's Most Popular Python/R Data Science Platform. Disponível em: https://www.anaconda.com/distribution/. Acesso em: 13 jun. 2021.

DALLAQUA, C. **Curso Python 3**: aula 1 - notebook jupyter. 1 vídeo ( 2,54 min.), 2016. Disponível em: https://www.youtube.com/watch?v=m0FbNlhNyQ8. Acesso em: 13 jun. 2021.

VISUAL Studio Code. Disponível em: https://visualstudio.microsoft.com/pt-br/. Acesso em: 13 jun. 2021.

FARIA, F. A. **Lógica de Programação**: aula 01 - introdução. São José dos Campos: Unifesp, 2016. 33 slides, color. Disponível em: https://www.ic.unicamp.br/~ffaria/lp2s2016/class01/lp-aula01.pdf. Acesso em: 13 jun. 2021.

GOOGLE. **Colaboratory**: Frequently Asked Questions. Disponível em: https://research.google.com/colaboratory/faq.html. **Acesso em: 10 abr. 2020.**

PYCHARM: The Python IDE for Professional Developers. Disponível em: https://www.jetbrains.com/pycharm/. Acesso em: 13 jun. 2021.

PYTHON Beginners Guide. Última atualização em 19 set. 2019, por Chris M. Disponível em: https://wiki.python.org/moin/BeginnersGuide/Overview. Acesso em: 13 jun. 2021.

PYTHON SOFTWARE FOUNDATION (org.). **History of the software**. Disponível em: https://docs.python.org/3.0/license.html. Acesso em: 13 jun. 2021.

PYTHON SOFTWARE FOUNDATION. **Numeric Types**: int, float, complex. Disponível em: https://docs.python.org/3/library/stdtypes.html. Acesso em: 13 jun. 2021.

PYTHON SOFTWARE FOUNDATION. **PEP 373**: Python 2.7 Release Schedule. Disponível em: https://www.python.org/dev/peps/pep-0373/. Acesso em: 13 jun. 2021.

ROSSUM, G. V.; WARSAW, B.; COGHLAN, N.**PEP 8**: Style Guide for Python Code. Disponível em: https://www.python.org/dev/peps/pep-0008/#introduction. Acesso em: 13 jun. 2021.

THE JUPYTER Notebook. Disponível em: https://jupyter.org/. Acesso em: 13 jun. 2021.

VISUAL Studio Code. Disponível em: https://visualstudio.microsoft.com/pt-br/. Acesso em: 13 jun. 2021.

BANIN, S. L. **Python 3 - conceitos e aplicações**: uma abordagem didática. São Paulo: Érica, 2018.

GRIFFITHS, D.; BARRY, P.**Head First Programming**: A learner's guide to programming using the Python language. [S.l.]: O'Reilly Media, 2009.

MANZANO, J. A. N. G.; OLIVEIRA, J. F. de. **Algoritmos**: lógica para desenvolvimento de programação de computadores. 29. ed. São Paulo: Érica, 2019.

PYTHON SOFTWARE FOUNDATION. **More Control Flow Tools**. Disponível em: https://docs.python.org/3/tutorial/controlflow.html. Acesso em: 13 jun. 2021.

PYTHON SOFTWARE FOUNDATION. Numeric Types: int, float, complex. Disponível em: https://docs.python.org/3/library/stdtypes.html. Acesso em: 13 jun. 2021.

SUBSECRETARIA DE TRIBUTAÇÃO E CONTENCIOSO. **Imposto sobre a renda das pessoas físicas**. Disponível em: http://receita.economia.gov.br/acesso-rapido/tributos/irpf-imposto-de-renda-pessoa-fisica. Acesso em: 13 jun. 2021.

BANIN, S. L. **Python 3 - conceitos e aplicações:** uma abordagem didática. São Paulo: Érica, 2018.

MANZANO, J. A. N. G; OLIVEIRA, J. F. de. Estudo Dirigido de Algoritmos. 15. ed. São Paulo: Érica, 2012.

PYTHON SOFTWARE FOUNDATION. **Built-in Functions**. Disponível em: https://docs.python.org/3/library/functions.html. Acesso em: 13 jun. 2021.

PYTHON SOFTWARE FOUNDATION. **Function and Variable Names**. Disponível em: https://www.python.org/dev/peps/pep-0008/#function-and-variable-names. Acesso em: 13 jun. 2021..

PYTHON SOFTWARE FOUNDATION. **Lambdas**. Disponível em: https://docs.python.org/3/reference/expressions.html#lambda. Acesso em: 13 jun. 2021.