NÃO PODE FALTAR

FUNÇÕES EM PYTHON

Vanessa Cadan Scheffer

IMPLEMENTANDO SOLUÇÕES EM PYTHON MEDIANTE FUNÇÕES

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

Fonte: Shutterstock.

Deseja ouvir este material?

Áudio disponível no material digital.

O assunto desta seção 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 1.3.

Quadro 1.3 | Funções built-in em Python

abs()	delattr()	hash()	memoryview()	set()
all()	dict()	help()	min()	setattr()
any()	dir()	hex()	next()	slice()
ascii()	divmod()	id()	object()	sorted()
bin()	enumerate()	input()	oct()	staticmethod()
bool()	eval()	int()	open()	str()
breakpoint()	exec()	isinstance()	ord()	sum()
bytearray()	filter()	issubclass()	pow()	super()
bytes()	float()	iter()	print()	tuple()
callable()	format()	len()	property()	type()
chr()	frozenset()	list()	range()	vars()
classmethod()	getattr()	locals()	repr()	zip()
compile()	globals()	map()	reversed()	__import__()
complex()	hasattr()	max()	round()	

Fonte: https://docs.python.org/3/library/functions.html

Ao observar o Quadro 1.3, 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 e explorar tais funções, aumentando cada vez mais seu repertório. Para mostrar o

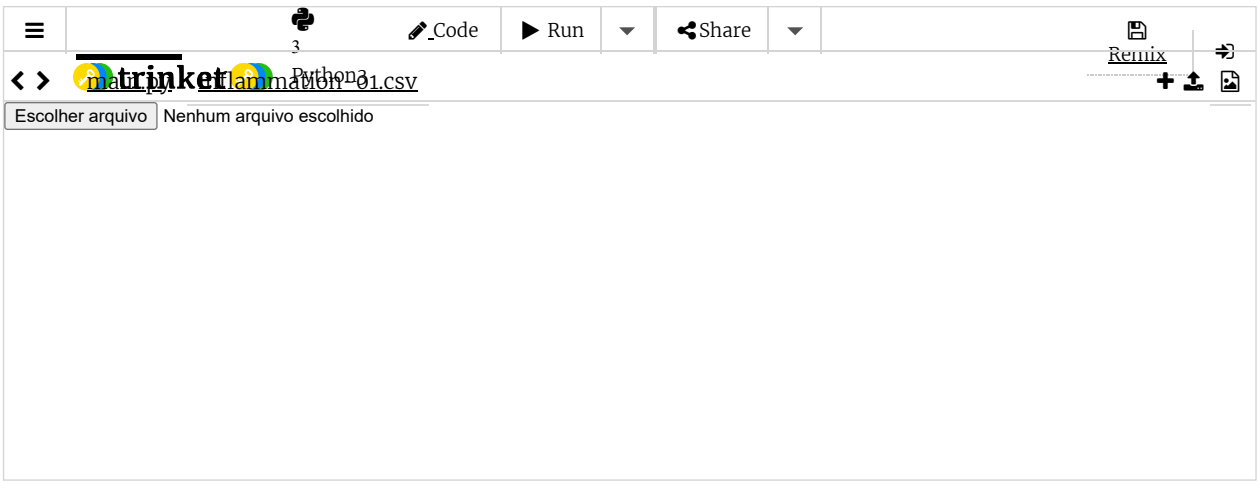
In [1]:
```
a = 2
b = 1

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 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 1.3), 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/#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.

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
```

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.

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

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 arbitráios, 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.

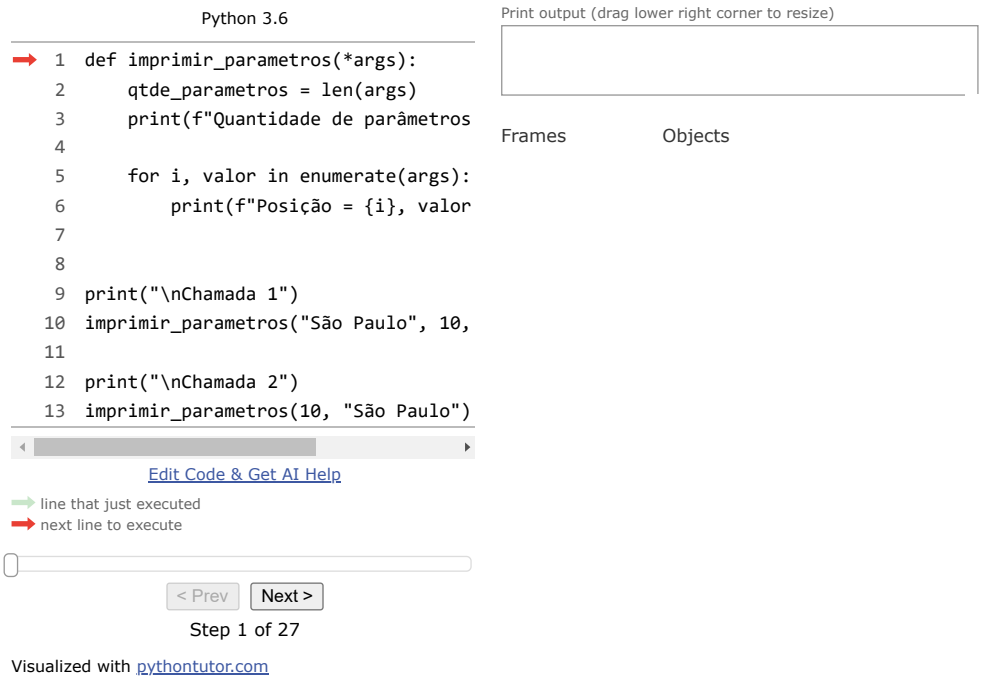
In [12]:
```
def imprimir_parametros(*args):
    qtde_parametros = len(args)
    print(f"Quantidade de parâmetros = {qtde_parametros}")

    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.

A seguir você pode testar o passo a passo de execução do código. Clique em "next" para visualizar a execução de cada linha de código.

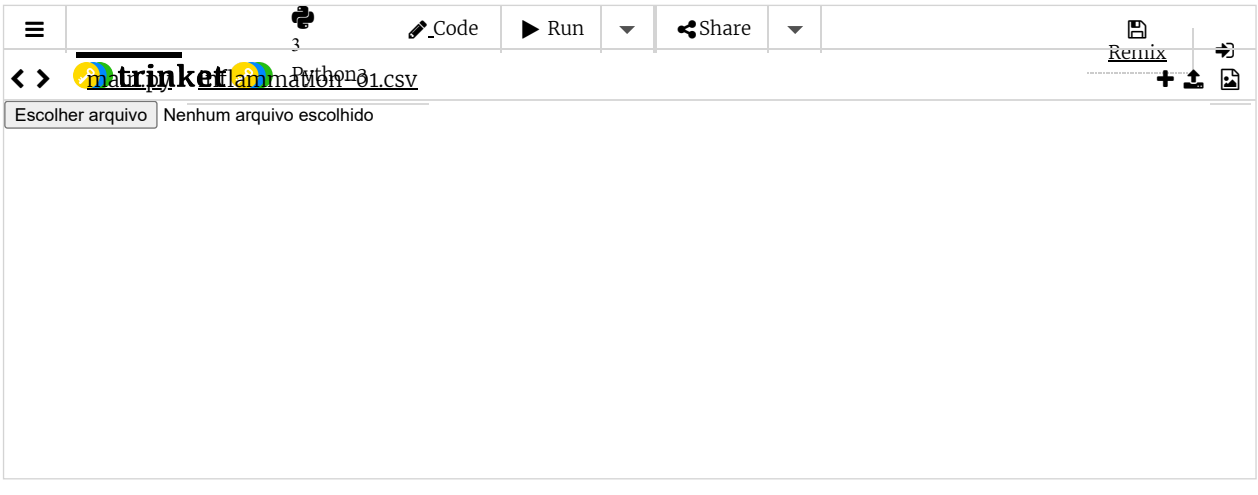



```
In [13]: def imprimir_parametros(**kwargs):
             print(f"Tipo de objeto recebido = {type(kwargs)}\n")
             qtde_parametros = len(kwargs)
             print(f"Quantidade de parâmetros = {qtde_parametros}")

             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 a seguir para testar os códigos e criar novas funções para



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 (às vezes chamadas de formas lambda) são usadas para criar funções anônimas https://docs.python.org/3/reference/expressions.html#lambda. 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
```

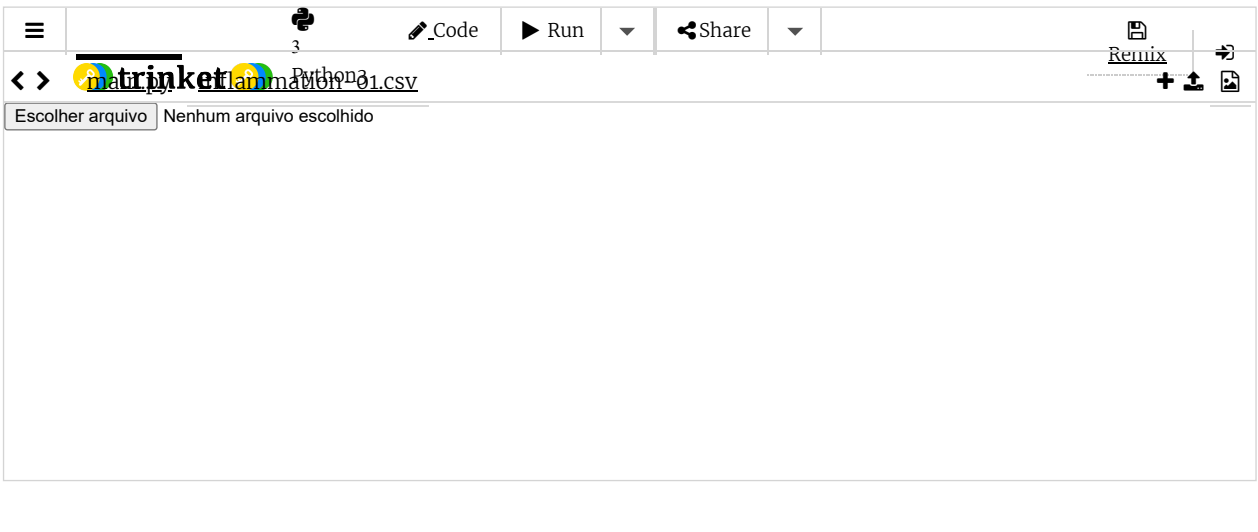
Na entrada 15 (In [15]), criamos uma função anônima que recebe como parâmetro dois valores e retorna a soma deles. Para criar essa função anônima, usamos a palavra reservada "lambda" passando como parâmetro "x, y". Após os dois pontos, é feito o cálculo matemático x + y. Na frente da função, já a invocamos 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 chamadada função através do nome da variável, passando os parâmetros que ela requer.

Na próxima aula, vamos aprender sobre outros tipos de dados em Python e voltaremos a falar mais sobre as funções lambda aplicadas nessas estruturas.



REFERÊNCIAS E LINKS ÚTEIS

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: 20 abr. 2020.

PYTHON SOFTWARE FOUNDATION. **Function and Variable Names**. Disponível em: https://www.python.org/dev/peps/pep-0008/#function-and-variable-names. Acesso em: 20 abr. 2020.

PYTHON SOFTWARE FOUNDATION. **Lambdas**. Disponível em: https://docs.python.org/3/reference/expressions.html#lambda. Acesso em: 20 abr. 2020.