Quando estudamos funções, aprendemos que elas podem receber dados (parâmetros) e podem fornecer uma resposta (retorno). Porém, o número de parâmetros era fixo para cada função: um dado para cada parâmetro que declaramos na definição da função. Da mesma forma, a função poderia retornar exatamente um resultado.

Em alguns casos, mais flexibilidade seria útil. Utilizando tuplas e dicionários conseguimos essa flexibilidade.

**1. Funções com retorno múltiplo**

Vejamos um caso simples: uma função que retorna os valores máximo e mínimo de uma coleção, separados por vírgula. Vamos imprimir o resultado e verificar o que acontece.

def max\_min(colecao):

maior = max(colecao)

menor = min(colecao)

return maior, menor

numeros = [3, 1, 4, 1, 5, 9, 2]

resposta = max\_min(numeros)

print(resposta)

print(type(resposta)) # mostra o tipo da variável resposta

maior = resposta[0]

menor = resposta[1]

Se você executar o resultado acima, verá que o retorno da função é uma tupla. Lembre-se que expressões contendo valores separados por vírgula em Python, mesmo na ausência de parênteses, são tratadas como tuplas.

No capítulo de tuplas, estudamos a operação de *desempacotamento de tuplas*. Sua aplicação neste caso pode ajudar a de fato lidar com essa função como sendo uma função que retorna múltiplos valores em vez de simplesmente uma função que retorna uma tupla:

def max\_min(colecao):

maior = max(colecao)

menor = min(colecao)

return maior, menor

numeros = [3, 1, 4, 1, 5, 9, 2]

maior\_num, menor\_num = max\_min(numeros)

print(maior\_num)

print(menor\_num)

Saída na tela:

9

1

Todas as variações de desempacotamento de tupla que já estudamos, incluindo o uso do operador \* para agrupar e/ou descartar parte dos valores retornados podem ser empregadas aqui.

**2. Parâmetros com valores padrão**

Uma primeira forma de trabalhar com a ideia de parâmetros opcionais é atribuir valores padrão para nossos parâmetros. Quando fazemos isso, quando a função for chamada, o parâmetro pode **ou** não ser passado. Caso ele não seja passado, é adotado o valor padrão.

Devemos primeiro colocar os parâmetros "comuns" (conhecidos como *argumentos posicionais*) para depois colocar os argumentos com valor padrão. Imagine, por exemplo, uma função que padroniza *strings* jogando todo seu conteúdo para minúsculas ou maiúsculas. Podemos implementá-la da seguinte maneira:

def padroniza\_string(texto, lower=True):

if lower:

return texto.lower()

else:

return texto.upper()

print(padroniza\_string('Sem passar o SEGUNDO argumento'))

print(padroniza\_string('Passando SEGUNDO argumento True', lower=True))

print(padroniza\_string('Passando SEGUNDO argumento False', lower=False))

Saída na tela:

sem passar o segundo argumento

passando segundo argumento true

PASSANDO SEGUNDO ARGUMENTO FALSE

**3. Funções com quantidade variável de parâmetros**

Talvez você já tenha notado que o print é uma função. Se não notou, esse é um bom momento para pensar a respeito. Nós sempre usamos com parênteses, nós passamos informações dentro dos parênteses (os dados a serem impressos) e ele faz um monte de coisa automaticamente: converte todos os dados passados para *string*, concatena todas as *strings* com um espaço entre elas e as escreve na tela.

Algo que o print tem que as nossas funções não tinham é a capacidade de receber uma quantidade variável de parâmetros/argumentos. Nós podemos passar 0 dados (e, neste caso, ele apenas pulará uma linha), 1 argumento, 2 argumentos, 3 argumentos... Quantos dados quisermos e ele funcionará para todos esses casos. Se temos que declarar todos os parâmetros, como fazer para que múltiplos dados possam ser passados?

**3.1. Agrupando parâmetros**

A solução é utilizar o operador \*.  Ao colocarmos o \* ao lado do nome de um parâmetro na definição da função, estamos dizendo que aquele argumento será uma coleção. Mais especificamente, uma tupla. Porém, o usuário não irá passar uma tupla. Ele irá passar quantos argumentos ele quiser, separados por vírgula, e o Python automaticamente criará uma tupla.

O exemplo abaixo cria uma função de somatório que pode receber uma quantidade arbitrária de números.

def somatorio(\*numeros):

# remova o símbolo de comentário das linhas abaixo para entender melhor o parâmetro

# print (numeros)

# print(type(numeros))

soma = 0

for n in numeros:

soma = soma + n

return soma

s1 = somatorio(5, 3, 1)

s2 = somatorio(2, 4, 6, 8, 10)

s3 = somatorio(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)

print(s1, s2, s3)

Saída na tela:

9 30 55

**3.2. Expandindo uma coleção**

O exemplo acima funciona muito bem quando o usuário da função possui vários dados avulsos, pois ele os agrupa em uma coleção. Mas o que acontece quando os dados já estão agrupados?

def somatorio(\*numeros):

print (numeros)

print(type(numeros))

soma = 0

for n in numeros:

soma = soma + n

return soma

lista = [1, 2, 3, 4, 5]

s = somatorio(lista)

print(s)

Note que o programa dará erro, pois como os print dentro da função ilustram, foi criada uma tupla, e na primeira posição da tupla foi armazenada a lista. Isso não funciona com a lógica que projetamos.

Para casos assim, utilizaremos o operador \* na chamada da função também. Na definição, o operador \* indica que devemos agrupar itens avulsos em uma coleção. Na chamada, ele indica que uma coleção deve ser expandida em itens avulsos.

def somatorio(\*numeros):

print (numeros)

print(type(numeros))

soma = 0

for n in numeros:

soma = soma + n

return soma

lista = [1, 2, 3, 4, 5]

s = somatorio(\*lista)

print(s)

Saída na tela:

(1, 2, 3, 4, 5)

<class 'tuple'>

15

No programa acima, a lista é expandida em 5 valores avulsos, e em seguida a função agrupa os 5 itens em uma tupla chamada "numeros".

**4. Parâmetros opcionais**

Outra possibilidade são funções com parâmetros opcionais. Note que isso é diferente de termos quantidade variável de parâmetros.

No caso da quantidade variável, normalmente são diversos parâmetros com a mesma utilidade (números a serem somados, valores a serem exibidos, etc).

Já os parâmetros opcionais são informações distintas que podem ou não ser passadas para a função. Você pode ou não passá-los, e sempre deve indicar o seu nome ao passá-los.

Já estudamos uma forma de parâmetros opcionais utilizando valores padrão. Mas para funções com uma **grande** quantidade de parâmetros opcionais, existe outra forma utilizando dicionários, apelidada como \*\*kwargs.

**4.1. Criando \*\*kwargs**

Para criar parâmetros opcionais, usaremos \*\*, e os parâmetros passados serão agrupados em um dicionário: o nome do parâmetro será uma chave, e o valor será o respectivo valor.

O exemplo abaixo simula o cadastro de usuários em uma base de dados. Um usuário pode fornecer seu nome, seu CPF ou ambos.

def cadastro(\*\*usuario):

if not ('nome') in usuario and not ('cpf') in usuario:

print('Nenhum dado encontrado!')

else:

if 'nome' in usuario:

print(usuario['nome'])

if 'cpf' in usuario:

print(usuario['cpf'])

print('-----')

cadastro(nome = 'João', cpf = 123456789) # tem ambos

cadastro(nome = 'José') # tem apenas nome

cadastro(cpf = 987654321) # tem apenas cpf

cadastro(rg = 192837465) # não tem nome nem cpf

Saída na tela:

João

123456789

-----

José

-----

987654321

-----

Nenhum dado encontrado!

**4.2. Expandindo um dicionário**

Analogamente ao caso dos parâmetros múltiplos, é possível que o usuário da função já tenha os dados organizados em um dicionário. Neste caso, basta usar \*\* na chamada da função para expandir o dicionário em vários parâmetros opcionais:

maria = {'nome':'Maria', 'cpf':2468135790}

cadastro(\*\*maria)

**Ordem dos parâmetros**

Caso sua função vá combinar múltiplos tipos de parâmetro, sempre siga a seguinte ordem: argumentos posicionais (os comuns), argumentos com asterisco (tupla), argumentos com valor padrão e argumentos com dois asteriscos (dicionário). Por exemplo, na função abaixo:

def funcao(a, b, \*c, d=0, e=1, \*\*f)

Quando ela for chamada, o Python fará o seguinte:

* os primeiros 2 valores serão atribuídos, respectivamente, para *a* e *b*.
* os próximos valores, independentes de quantos sejam, serão incluídos na tupla *c*.
* se os valores *d* e/ou *e* forem passados explicitamente pelo nome, os valores passados serão adotados, senão, serão adotados os valores padrão.
* quaisquer outros valores passados por nome serão incluídos no dicionário *f*.