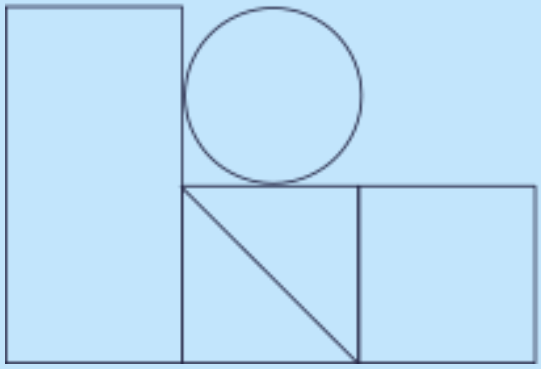
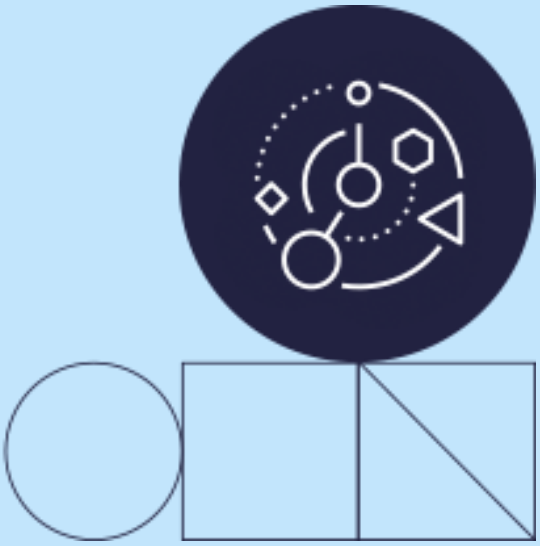
IBM SkillsBuild

IBM

Noções básicas e sintaxe de Python

Funções e variáveis





Índice

Introdução

Nesta unidade, vamos aprender como criar funções em Python. Uma função é um grupo de declarações agrupadas de tal forma que podem ser invocadas com o mesmo nome. As funções podem retornar um resultado e podem ser parametrizadas para permitir que o resultado da função seja diferente, dependendo de como ela é chamada.

As funções são também a unidade estrutural mais básica que maximiza a reusabilidade do código, permitindo-nos avançar para noções mais ambiciosas de projeto de software do que vimos até agora. Isto porque as funções nos permitem separar nossos programas em blocos menores, mais gerenciáveis, que podem ser reutilizados em várias partes de nosso software. A implementação de nosso código em funções torna nosso código mais reutilizável, mais fácil de programar e permite que outros programadores entendam nosso código mais facilmente.

Vantagens da utilização das funções

Aumenta a reusabilidade do código e minimiza a redundância (repetição)

As funções são a maneira mais simples e fácil de empacotar funcionalidades para que possam ser usadas em diferentes partes de um programa sem ter que repetir o código. Eles nos permitem agrupar e generalizar o código para que possamos usá-lo arbitrariamente com a freqüência necessária. Isto torna as funções um elemento essencial de fatorização do código, permitindo-nos reduzir a redundância e, portanto, reduzir o esforço necessário para manter nosso código.

Habilitar a decomposição de procedimentos

Ou seja, eles permitem que os programas sejam divididos em pequenas partes onde cada parte tem um papel bem definido. Geralmente é mais fácil programar pequenos pedaços de código e compô-los em programas maiores do que escrever todo o processo de uma só vez.

Nesta unidade, aprenderemos a sintaxe necessária para operar com funções, aprenderemos o conceito de área (scope) e veremos como parametrizar nossas funções para torná-las mais genéricas. Com tudo isso, estaremos prontos para iniciar um caminho muito mais ambicioso no mundo da programação Python, que continuaremos nas próximas unidades e cursos desta especialização.

Sintaxe da função básica em Python

A sintaxe necessária para declarar uma função é a seguinte:

```
def nombre_de_la_función(arg1, arg2, ...a
rgN):
    sentencias
    return          #El return es opciona
l
```

A declaração começa com a declaraçao **def** seguido do nome que queremos dar à função. Em seguida, escrevemos uma lista de argumentos de 0 a N (também chamados de parâmetros da função) encapsulados entre parênteses e terminamos a declaração com o caracter “:”.

Depois disso, e em uma nova linha, escreveremos o corpo de nossa função, que consistirá em um grupo de frases a serem executadas, terminando com a declaraçao opcional **return** acompanhado do valor a ser devolvido.

Para chamar a função simplesmente digitamos o nome da função seguido dos argumentos que queremos passar a ela, encapsulados entre parênteses. Vamos ver alguns exemplos:

```
def suma(a, b):   # Definimos la función
"suma". Tiene 2 parámetros.
    return a+b      # "return" devuelve
el resultado de la función.

suma(2, 3)        # Llamada a la función.
Hay que pasarle dos parámetros.

# Resultado: 5

def en_pantalla(frase1, frase2):
    print(frase1, frase2)   # "return" no
es obligatorio

en_pantalla('Me gusta', 'Python')

# Resultado: Me gusta Python
```

Como você pode ver, não é necessário usar ***return***.
Em funções que não têm ***return***, retornan ***None***.

```
def suma(a, b):      # Definimos la
función "suma". Tiene 2 parámetros.
    return a+b       # "return" devuelve
el resultado de la función.

x = suma (2, 3)
print(x)             # Guardamos el
resultado en x
```

Funções e polimorfismo

Mencionamos que uma das vantagens de utilizar funções é que elas permitem a reusabilidade do código. Isto é ainda mais verdadeiro em Python, onde muitos tipos de dados suportam o polimorfismo, ou seja, cada tipo de dado sabe como se comportar quando confrontado com uma grande variedade de operadores. Isto é diretamente aplicável ao uso de funções, por isso podemos encontrar casos como os seguintes:

```
def suma(a, b):      # Definimos la
función "suma". Tiene 2 parámetros.
    return a+b       # "return" devuelve
el resultado de la función.

suma (2, 3)          # Función con ints
# Resultado = 5

suma(2.7, 4.0)       # Función con floats
# Resultado = 6.7
```

```
suma('Me gusta', 'Python')  # Función con
strings
```

Isto porque em Python **as funções não têm nenhum tipo** (lembre-se da unidade de digitação dinâmica). Em um grande número de casos, o tipo de saída de uma função dependerá do tipo dos parâmetros que passamos para ela (há exceções a esta regra, por exemplo, se estamos forçando o tipo de saída na declaração ***return***). Esta é uma idéia central na linguagem que a torna muito flexível e fácil de reutilizar ao programar.

Em Python, uma função não tem que se preocupar com os tipos de entrada e saída. É o próprio intérprete que verificará se os tipos que passamos para a função suportam os protocolos que codificamos dentro dela. Na verdade, se não os suportar, o intérprete gerará uma exceção, o que nos poupa de ter que fazer a verificação de erros em nossa função. Se o fizéssemos, estaríamos reduzindo a flexibilidade da função, o que normalmente não é desejável, a menos que seja uma decisão de projeto.

Funções aninhadas

É possível criar funções dentro das funções.

```
def f1(a):      # Función que "encierra" a f2 (enclosing)
    print(a)
    b = 100
    def f2(x):      # Función anidada
        print(x)    # Llamamos a f2 desde f1
    f2(b)

f1('Python')        # Llamamos a f1
```

Resultado:

```
✓ def f1(a): # Función que "encierra" a f2

Python
100
```

Como podemos ver, é possível criar funções dentro de outras funções.

Recurssão

Como em outras linguagens de programação, em Python uma função pode chamar-se a si mesma, gerando recursividade. É importante levar em conta que não se gera uma recorrência infinita, ou seja, a função deve ter uma condição de saída. Um exemplo muito comum de recorrência na programação é a função que calcula o fatorial de um número (lembre-se que o fatorial de x é igual a x * (x-1) * (x-2) * ... * 1

```
def factorial(x):
    if x>1:
        return x*factorial(x-1)
    else:
        return 1

factorial(5)
```

Nesta função, vemos que a condição de saída de recorrência é satisfeita quando x é igual a 1.

Retorno de múltiplos valores simultaneamente

As funções podem retornar qualquer objeto com a declaração ***return***. Assim, se combinarmos o retorno de uma tupla, com o desempacotamento prolongado que Python permite, podemos simular o retorno de múltiplos valores:

```
def maxmin(lista):
    return max(lista), min(lista)   # Devielveuna tupla de 2 elementos

l = [1, 3, 5, 6, 0]
maximo, minimo = maxmin(l)        # Desempaqueta la tupla en 2 variables

print(minimo, maximo, sep= ' ')
```

Resultado:

```
✓ def maxmin(lista): ···

0 6
```

Experimente

Fazer uma função que decomponha um número em fatores. Ele deve retornar uma lista dos fatores desse número. Lembre-se de que a decomposição em fatores de um número consiste em encontrar o conjunto de números primos cuja multiplicação dá como resultado esse número.

Dica: A primeira coisa que a função deve fazer é encontrar todos os números primos menos do que o número em questão.

Escopo de uma função

Agora que começamos a operar com funções, é hora de ir um pouco mais longe com o conceito de nomes em Python. Quando usamos um nome em um programa, o intérprete Python procura por esse nome em um namespace, que chamamos ***namespace***. Um *namespace* é um local onde reside um conjunto de nomes. Se você vem de outras linguagens de programação, você estará familiarizado com este conceito.

Quando atribuímos um nome em Python, ele está associado ao espaço de nomes ao qual pertence. O namespace em que reside um nome define seu escopo ***(scope)***, ou seja, a visibilidade que este nome terá em relação a outras variáveis. Por exemplo,

Nomes atribuídos dentro do escopo de uma função só são visíveis para códigos que residem dentro dessa função. Isto significa que um nome declarado dentro de uma função não pode ser referenciado de fora da função.

O escopo de uma variável depende de onde ela é atribuída. Em outras palavras, dependendo do local onde atribuímos uma variável, estaremos definindo o escopo em que ela pode ser utilizada. As variáveis podem ser atribuídas em três âmbitos:

- Se uma variável é atribuída dentro de uma função, ela é local para essa função.
- Se uma variável é atribuída dentro de uma função, ela é não-local para todos os escopos de função aninhados dentro dela.
- Se uma variável é atribuída fora de qualquer função, ela é global ao arquivo no qual ela é criada.

Aqui está um exemplo:

```
a = 'Python'                    #Scope global
(al módulo)
print('Valor fuera:', a)

def funcion():
    a=33
    print('Valor dentro', a)    #Scope
local a la función

funcion()

print('Valor fuera', a)
```

Resultado:

```
✓ a = 'Python' #Scope global (al módulo) ···

Valor fuera: Python
Valor dentro 33
Valor fuera Python
```

Escopos e suas propriedades

Como podemos ver, as funções permitem que você crie namespaces âmbitos aninhados que limitam os âmbitos de acesso e evitam colisões de nomes variáveis. Assim, em funções criamos escopos locais que contrastam com o escopo global dos módulos. Estes escopos têm as seguintes propriedades:

- As variáveis que são criadas no escopo do módulo são variáveis globais para aquele arquivo. Quando importamos um módulo de outro módulo, estas variáveis se tornam atributos do módulo importado, de modo que podem ser acessadas como variáveis simples do módulo. Por exemplo, a variável math.pi é global para o módulo math, mas se importarmos o módulo math, podemos acessá-lo como se fosse um atributo do mesmo (por isso, acessamo-lo por meio de math.pi).

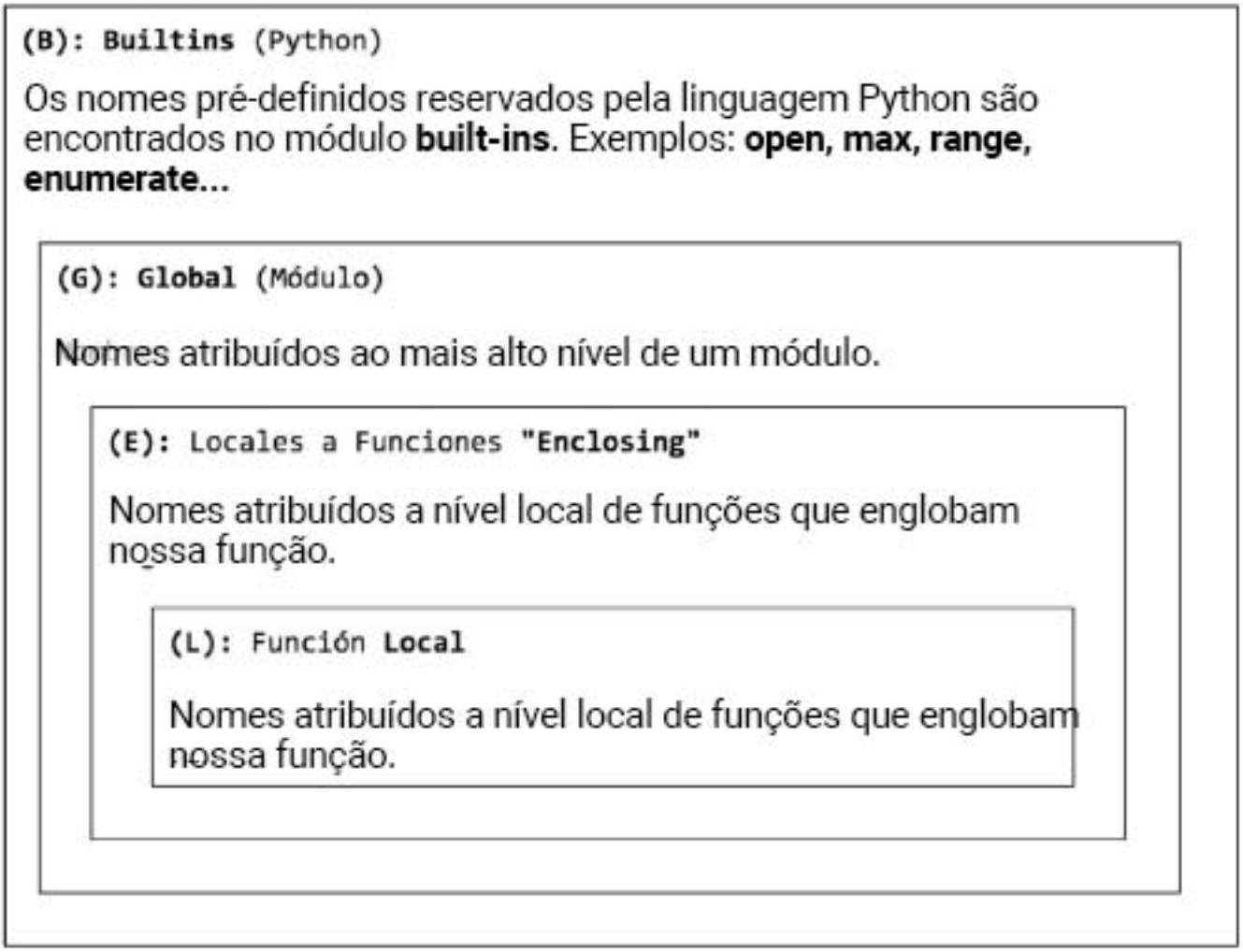
- O escopo global é na verdade um escopo de arquivo. Ou seja, o escopo global afeta apenas o módulo (arquivo) no qual suas variáveis estão localizadas. Em Python escopo global significa escopo do módulo.
- Os nomes atribuídos dentro de uma função são locais por padrão. Todos os nomes definidos dentro de uma função têm como escopo o namespace local para essa função. Entretanto, é possível fazer atribuições a variáveis fora do escopo local da função. Isto é feito utilizando declarações globais e não locais dentro da função. No entanto, esta prática geralmente não é recomendada.
- Todos os outros nomes são locais para as funções que envolvem nossa função (funções nas quais nossa função está aninhada), globais ou built-ins. Nomes não atribuídos em nossa função ou são locais para funções em um nível mais alto de aninhamento (chamadas enclosing), globais para o módulo, ou pertencem ao módulo built-ins, um escopo de nível mais alto onde todos os tipos básicos de Python são definidos.
- Cada vez que uma função é chamada, um escopo local é criado para essa função. Ou seja, um namespace onde todos os nomes dessa função são armazenados.

Resolução do nome. A regra LEGB

Todas as regras acima podem ser resumidas pela regra **LEGB (Local, Enclosing, Global, Builtin).** Esta regra dita a ordem na qual procuramos um nome quando tentamos acessá-lo. Ao tentar acessar um nome, o intérprete Python procura apenas 4 escopos diferentes. Além disso, esta busca é sempre feita na mesma ordem.

- **L**: Primeiro, olhamos para o nível local de nossa função.
- **E:** Analisamos então as funções que encerram nossa função *(enclosing)*
- **G:** O escopo global do módulo é então procurado.
- **B:** Finalmente, é procurado no módulo *builtins.*

A Figura 1 mostra um exemplo desta busca e como estes campos de busca são englobados. Note que esta busca termina no primeiro lugar onde o nome procurado é encontrado. No caso em que o nome não é encontrado nestes escopos, Python retorna um erro.



Escopos de busca da LEGB. Quando uma variável é referenciada, Python realiza uma busca começando no nível **local** e trabalhando até ***builtins.*** A busca termina no primeiro campo onde o nome referenciado é encontrado.

Aqui está um exemplo de tal busca.

```
G = 'Esta variable es de ámbito Global'
def f1():
    E='Esta variable es local a f1.
Enclosing a f2'
    def f2():
        L = 'Esta variable es local a f2'
        print(L, E, G, sep = '\n')
        f2()

f1()
```

```
Esta variable es local a f2
Esta variable es local a f1. Enclosing a f2
Esta variable está en ámbito Global (de
módulo)
```

Como podemos ver, ***f2*** é capaz de referenciar as variáveis ***E*** y ***G***. Agora vamos ver o que acontece se declararmos essas mesmas variáveis no escopo do ***f2.***

```
G = 'Esta variable es de ámbito Global'
def f1():
    E='Esta variable es local a f1.
Enclosing a f2'
    def f2():
        L = 'L es local a f2'
        E = 'E también es local a f2'
        G = 'G también es local a f2'
        print(L, E, G, sep = '\n')
        f2()

f1()
```

```
L es local a f2
E también es local a f2
G también es local a f2
```

Neste caso, a busca pára na área de ***local*** de ***f2*** como declaramos variáveis com o mesmo nome no mesmo escopo (***E*** e ***G***). Finalmente, vamos ver o que acontece quando tentamos acessar desde o escopo de uma função até o escopo de outra função.

```
G = 'Esta variable es de ámbito Global'
def f1():
    E='Esta variable es local a f1.
Enclosing a f2'
    def f2():
        L = 'L es local a f2'
        E = 'E también es local a f2'
        G = 'G también es local a f2'
        print(L, E, G, sep = '\n')
    def f3():
        print(L)        # DEVUELVE ERROR
    f2()
    f3()

f1()
```

Resultado:

```
⊗ G = 'Esta variable es de ámbito Global' ···

L es local a f2
E también es local a f2
G también es local a f2

---------------------------------------------------
------------------------
NameError                                 Tracebac
k (most recent call last)
j:\WORKSPACE\10 PYTHON\temp.py in line 14
     18     f2()
     19     f3()
---> 21 f1()

j:\WORKSPACE\10 PYTHON\temp.py in line 12, in f1()
     17     print(L)        # DEVUELVE ERROR
     18 f2()
---> 19 f3()

j:\WORKSPACE\10 PYTHON\temp.py in line 10, in f1.<
locals>.f3()
      16 def f3():
---> 17     print(L)

NameError: name 'L' is not defined
```

Desta vez, quando tentamos acessar uma variável local a ***f2*** de ***f3***, obtemos um erro de nome. Isto porque a busca do nome é feita a partir do escopo local de ***f3*** para cima, passando primeiro para o escopo local de ***f1***, depois para o escopo **global** e, finalmente, para **builtins**. Em nenhum caso pesquisamos o escopo da ***f2***. Observe, entretanto, que a ***f3*** tem acesso aos escopos global e builtins:

```
G = 'Esta variable es de ámbito Global'
def f1():
```

```
    E='Esta variable es local a f1.
Enclosing a f2'
    def f2():
        L = 'L es local a f2'
        E = 'E también es local a f2'
        G = 'G también es local a f2'
        print(L, E, G, sep = '\n')
    def f3():
        print(E, G, sep = '\n')
    f2()
    f3()

f1()
```

Resultado:

```
✓ G = 'Esta variable es de ámbito Global' ···

L es local a f2
E también es local a f2
G también es local a f2
Esta variable es local a f1. Enclosing a f2
Esta variable es de ámbito Global
```

Parâmetros e argumentos

Argumentos sobre as funções

Quando definimos uma função, podemos parametrizá-la pelos argumentos que passamos a ela. Por exemplo, a função a seguir é capaz de multiplicar quaisquer dois números, uma vez que estamos definindo por meio de dois parâmetros de entrada (argumentos):

```
def suma(a, b):
    return a+b

suma(2, 3)

suma(40, 30)
```

Resultado:

```
5
70
```

Algumas coisas a ter em mente ao passar argumentos para as funções Python:

- Ao passar um argumento para uma função, estamos criando uma atribuição a uma variável com o nome do argumento no escopo local da função.
- Atribuir um novo valor ao argumento a partir de dentro da função não afeta o exterior.
- Se passarmos um objeto mutável para uma função, e ele o modifica dentro da função, pode afetar o exterior.

Isto porque os argumentos em Python são **passados por referência**. Se nos lembrarmos do que vimos na unidade Dynamic Typing, entenderemos melhor o que isto significa.

- Se passamos um objeto imutável, é como se o estivéssemos fazendo por valor. Ou seja, como não podemos modificar o objeto, é como se o estivéssemos copiando para o escopo local da função.
- Se passarmos um objeto mutável, ao fazê-lo por referência, qualquer mudança dentro do objeto que for feita dentro da função será observada de fora da função.

Aqui estão alguns exemplos. Vamos começar passando objetos imutáveis:

```
def suma(a, b):      # Modificamos a y b
en el scope de suma()
    a = 3
    b = 4
    return a+b

a, b = 5, 10
print(suma(a, b))
print(a, b)          # a y b no han
cambiado fuera de la función
```

Por outro lado, quando passamos um objeto mutável para uma função que o modifica internamente, a mudança afeta o exterior da função:

```
def minimo(l):
    l[0] = 1000     # Modificamos la
lista en el interior
    return min(l)

l = [1, 2, 3]
print(l)

print(minimo(l))    # Modifica la lista
aquí

print(l)
```

Resultado:

```
✓ def minimo(l): ···

[1, 2, 3]
2
[1000, 2, 3]
```

Lembre-se que para evitar este comportamento podemos fazer uma cópia da lista que passamos para a função:

```
def minimo(l):
    l[0] = 1000     # Modificamos la
lista en el interior
    return min(l)

l = [1, 2, 3]
print(minimo(l[:])) # minimo modifica la
lista aquí
print(l)
```

Resultado:

```
✓ def minimo(l): ···

2
[1, 2, 3]
```

Uma visão geral das diferentes formas de passagem de argumentos

Por padrão, os argumentos são passados por posição. Ou seja, os argumentos de uma chamada de função devem ser passados na mesma ordem em que a função foi definida.

Entretanto, uma das grandes vantagens do Python é que ele nos permite passar argumentos para funções de várias maneiras diferentes.

Por posição.

A forma padrão de passar argumentos, da esquerda para a direita.

```
def f(a, b, c):
    print(a, b, c)

f(1, 2, 3)
```

Resultado:

```
✓ def f(a, b, c): ···

1 2 3
```

Por keywords (palavras-chave)

Ao invés de chamar a função com seus argumentos em ordem, a função é passada especificando o nome do argumento seguido do valor a ser passado. O nome da sintaxe=valor é usado. Como você está especificando o nome dos argumentos explicitamente, não é necessário ordená-los por posição.

```
def f(a, b, c):
    print(a, b, c)

f(c=12, a=10, b=100)
```

Resultado:

```
✓ def f(a, b, c): ···

10 100 12
```

Especificação de valores padrão na definição da chamada

É possível definir quais são os valores padrão que os argumentos de uma função têm. Assim, se ao chamar a função, não passarmos nenhum dos argumentos que têm um valor padrão, esse valor será utilizado.

```
def f(a, b=10, c=30):
    print(a, b, c)

f(1)

f(1, 12)

f(1, 12, 19)
```

Resultado:

```
✓ def f(a, b=10, c=30): ···

1 10 30
1 12 30
1 12 19
```

Ao especificar na função que um conjunto de argumentos lhe será transmitido

A definição da função indica que um número arbitrário de argumentos será passado para a função, que será classificado por posição da esquerda para a direita (indicado por um asterisco * imediatamente antes do nome) ou por keywords (indicado por um asterisco duplo ** imediatamente antes do nome).

```
def f(*args):       # Acepta número
arbitrario de argumentos
    print(args)

f()                 # Si no hay
argumentos, args es una tupla vacía

f(1)

f(1, 2)

f(1, 2, 3, 4, 5, 6)
```

Resultado:

```
✓ def f(*args): # Acepta número arbitrario de

()
(1,)
(1, 2)
(1, 2, 3, 4, 5, 6)
```

Se utilizarmos a sintaxe de duplo risco, especificamos que passaremos os argumentos pelo nome:

```
def f(**Kargs):       # Acepta número de
argumentos por nombre
    print(Kargs)

f()                 # Si no hay
argumentos, Kargs es un diccionario vacío

f(a=1)

f(a=1, b=2)

f(a=1, c=3, b=2)
```

Resultado:

```
✓ def f(**Kargs): # Acepta número de argumentos

{}
{'a': 1}
{'a': 1, 'b': 2}
{'a': 1, 'c': 3, 'b': 2}
```

Desempacotar uma coleção de argumentos posicionais ou keyword

Ao chamar uma função você pode usar a sintaxe de * para desembalar uma coleção em uma série de argumentos separados por posição.

```
✓ def f(**Kargs): # Acepta número de argumentos ···

{}
{'a': 1}
{'a': 1, 'b': 2}
{'a': 1, 'c': 3, 'b': 2}
```

Resultado:

```
✓ def f(a, b, c, d): ···
1 2 3 4
100 1 2 3
100 200 1 2
```

Da mesma forma, um dicionário pode ser desempacotado usando a sintaxe de duplo asterisco **.

```
def f(a, b, c, d):
    print(a, b, c, d)

argumentos = {'b':20, 'a':1, 'c':300,
'd':4000}
f(**argumentos)     # Desempaquetando
diccionario argumentos con **

argumentos = {'b':200, 'c':300, 'd':400}
f(10, **argumentos) # Podemos combinar
argumentos posicionales con dict
```

Resultado:

```
✓ def f(a, b, c, d): ···
1 20 300 4000
10 200 300 400
```

Usando argumentos que só podem ser passados por código (keyword-only)

Este tipo de argumento só pode ser passado como uma chave (keyword) e que nunca será preenchido por cargo. Isto é útil em funções que podem aceitar qualquer número de argumentos e ainda permitir configurações opcionais.

```
def f(a, *, b, c):      # Define 'b' y
'c' como keyword-only con el *
    print(a, b, c)

f(1, b=10, c=100)

f(1, 10, 100)           # Error al no
pasar argumentos Keyword-only
```

Resultado:

```
⊗ def f(a, *, b, c): # Define 'b' y 'c' como ···
1 10 100

  ------------------------------------------------
  ------------------------
  TypeError                                 Tracebac
  k (most recent call last)
  j:\WORKSPACE\10 PYTHON\temp.py in line 5
        6     print(a, b, c)
        8 f(1, b=10, c=100)
  ----> 9 f(1, 10, 100)

  TypeError: f() takes 1 positional argument but 3 w
  ere given
```

```
def f(a, *b, c):      # Hay que pasar 'c'
por clave obligatoriamente
    print(a, b, c)

f(1, c=2)

f(1, 2, c=3)

f(1, 2, 3, 4, 5, c=10)
```

Resultado:

```
✓ def f(a, *b, c): # Hay que pasar 'c' por clave ···
1 () 2
1 (2,) 3
1 (2, 3, 4, 5) 10
```

Argumentos após "*" tornam-se *keyword-only*:

Veremos que há muitas funções escritas em Python que tiram proveito dessas funcionalidades, tais como as funções builtin **zip** e **print**.

```
la = [1, 2, 3, 4, 5]
lb = list('abcde')
lc = list('ABCDE')

zlist = list(zip(la, lb, lc))   # zip
soporta cualquier número
                                # de
argumentos posicionales
```

```
zlist

a, b, c = zip(*zlist)           # El * en
zip desempaqueta lista de tuplas

print(la, lb, lc, sep = '\n')

print(la, lb)                   #
Seperador por defecto es espacio
```

Resultado:

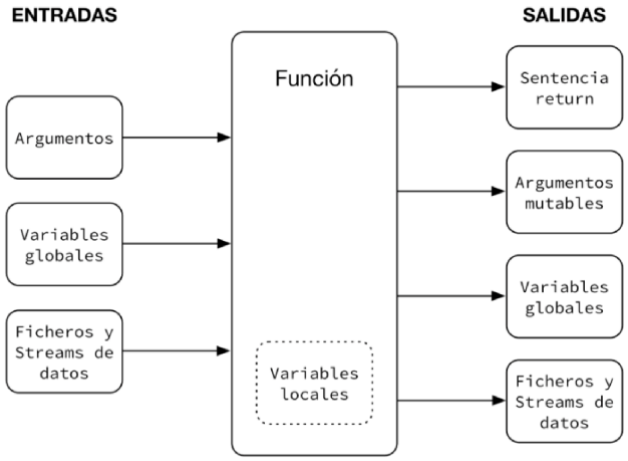
```
✓ la = [1, 2, 3, 4, 5] ···

[1, 2, 3, 4, 5]
['a', 'b', 'c', 'd', 'e']
['A', 'B', 'C', 'D', 'E']
[1, 2, 3, 4, 5] ['a', 'b', 'c', 'd', 'e']
```

Como podemos ver, este tipo de passagem de argumentos dá às funções Python uma grande flexibilidade. Além disso, o uso de argumentos padrão e *keyword-only* promover a simplicidade e a legibilidade de nosso código.

Considerações de projeto ao programar com funções

Para concluir esta unidade sobre funções, terminaremos com algumas considerações de projeto que você deve ter em mente ao programar com elas. Para isso, é necessário entender quais entradas e saídas são possíveis para qualquer função. Você tem uma explicação detalhada no Figura 2.



A figura descreve os principais mecanismos pelos quais uma função interage com o mundo ao seu redor. Ao programar funções, é importante ter em mente dois importantes aspectos de projeto que ajudam a tornar um código mais utilizável e de fácil manutenção. Estes aspectos são:

- **Coesão**: refere-se a como um programa é dividido em pequenas tarefas individuais.
- **Acoplamento**: refere-se a como as funções se comunicam e dependem umas das outras.

O bom código é altamente coeso e tem pouco acoplamento. Ou seja, todas as funções em seu código ajudam na mesma tarefa, mas não dependem uma da outra. Abaixo está uma lista de diretrizes que o ajudarão a aumentar a coesão e reduzir o acoplamento.

Acoplamento

- Crie funções que só aceitam entrada para argumentos e só usam o ***return*** para sua saída. Isto reduz as dependências externas das funções. Uma função ***f(args)*** que cumpre este critério sempre retorna o mesmo resultado para os mesmos argumentos. Ou seja, a função se torna mais previsível e, portanto, mais fácil de entender e manter.
- Usar variáveis globais somente quando realmente necessário. Variáveis globais criam dependências entre funções e tornam os programas difíceis de debugar, modificar e reutilizar.
- Evite mudar os argumentos mutáveis, a menos que o código de chamada o espere. O problema é o mesmo que com as variáveis globais, mas também cria um forte acoplamento entre a função chamada e a função chamada.
- Não altere diretamente os atributos de outros módulos. Isto cria um forte acoplamento entre módulos, assim como as variáveis globais criam um forte acoplamento entre funções. Se você precisar alterar um atributo de um módulo, utilize as funções fornecidas pelo próprio módulo. Se não o fizer, então mudar seus atributos pode ser uma péssima idéia.

Coesão

- Cada função deve ter um único propósito. O código bem projetado é composto de funções que fazem apenas uma coisa. Normalmente o objetivo de uma função deve ser resumido em uma frase curta. Se a frase que descreve sua função for muito genérica (p. ex. *“esta função resolve meu problema”*) ou se tiver muitas conjunções (p. ex. *“esta função calcula X e processa Y”*), considerar dividi-lo em partes mais simples.
- Todas as funções devem ser relativamente pequenas. Este ponto está relacionado com o anterior. Tenha em mente que o código escrito em Python é muito conciso. Se sua função tem mais de uma dúzia ou duas linhas ou contém vários níveis de aninhamento, é provável que você tenha um problema de projeto..

Em resumo, tente fazer com que seu código consista em muitas pequenas funções que são independentes umas das outras.

Experimente

Crie uma função de registro que aceite qualquer número de argumentos e os imprima na tela em uma única linha. A linha deve começar com o prefixo ‘LOG:’.

Modifica a função log para permitir que o usuário especifique qualquer prefixo desejado.

Modificar a função log para que o prefixo tenha um valor padrão ‘LOG: ’.

Modificar a função log para permitir ao usuário definir tanto o prefixo quanto o separador entre os argumentos. Ambos devem ser passados somente pelos nomes (não pela posição) 'sep' e 'prefix'. Estes não precisam ter um valor padrão.

Modifica a função log para que agora 'sep' e 'prefix' tenham um valor padrão.

Modifique a função log para aceitar o seguinte dicionário. Lembre-se de passá-lo desembalado com a sintaxe de asterisco duplo (**).

Definindo Variáveis

As variáveis são um dos dois componentes básicos de qualquer programa.

Em sua essência, um programa é composto de dados e instruções que manipulam esses dados. Normalmente, os dados são armazenados na memória (RAM) para que possamos acessá-los.

Então, o que é uma variável? Uma variável é uma forma de identificar, de maneira simples, um dado que é armazenado na memória do computador ou, em outras palavras, um espaço na memória do computador onde é armazenado um valor que pode mudar durante a execução do programa. Imaginemos que uma variável é um recipiente no qual um pedaço de dado é armazenado, o qual pode mudar durante o fluxo do programa. Uma variável nos permite acessar facilmente esses dados para serem manipulados e transformados.

Uma variável é um conceito fundamental em qualquer linguagem de programação. É um local reservado de memória que armazena e manipula os dados. Em algumas linguagens de programação, as variáveis podem ser pensadas como "caixas" nas quais os dados são armazenados, mas em Python, as variáveis são "etiquetas" que permitem que você se refira aos dados (que são armazenados em "caixas" chamadas objetos).

Ao contrário de outras linguagens de programação, Python não tem comandos para declarar uma variável. Uma variável é criada assim que lhe é dado um nome e lhe é atribuído um valor.

```
x = 15
y = "Ana"
print(x)
print(y)
```

Suponha que queremos exibir o resultado da adição de 1 + 2. Para exibir o resultado, devemos dizer ao programa onde esses dados estão na memória e, para isso, fazemos uso de uma variável:

```
# Guardamos en la variable suma el
resultado de 1 + 2
suma = 1 + 2
# Accedemos al resultado de 1 + 2 a
través de la variable suma
print(suma)
```

Atribuição de um valor a uma variável em Python

Como vimos no exemplo anterior, para atribuir um valor (dados) a uma variável, usamos o operador de atribuição `=`.

Três partes estão envolvidas na operação de cessão:

- O operador de cessão `=`
- Um identificador ou nome variável, à esquerda do operador.
- Uma literal, uma expressão, uma chamada de função, ou uma combinação de todas elas à direita do operador de atribuição.

Ejemplos:

```
# Asigna a la variable <a> el valor 1
a = 1
# Asigna a la variable <a> el resultado
de la expresión 3 * 4
a = 3 * 4
# Asigna a la variable <a> la cadena de
caracteres 'Pythonista'
a = 'Pythonista'
```

Quando atribuímos um valor a uma variável pela primeira vez, é onde a variável é definida e rubricada. Em um roteiro ou programa escrito em Python, podemos definir variáveis em qualquer parte do

roteiro. No entanto, é uma boa prática definir as variáveis que você vai usar no início.

Se tentarmos usar uma variável que não tenha sido previamente definida/inicializada, o intérprete nos mostrará um erro:

```
print(a)
```

Estreitamente relacionado às variáveis está o conceito de tipo de dados. Quando um valor é atribuído a uma variável, esse valor pertence a um conjunto de valores conhecido como um tipo de dado. Um tipo de dado define uma série de características sobre esses dados e as variáveis que os contêm, tais como as operações que podem ser realizadas sobre eles. Em Python, os tipos de dados básicos são numéricos (inteiro, real e complexo), booleanos (True, False) e cadeias de caracteres.

Os tipos de dados, sua declaração e como trabalhar com eles serão abordados em profundidade no próximo tópico.

A variável de lista que acabamos de declarar nada mais é do que uma lista de números. Este é outro dos tipos básicos de **Python** que veremos mais adiante. Por enquanto, basta entender que as listas são coleções de objetos encomendados.