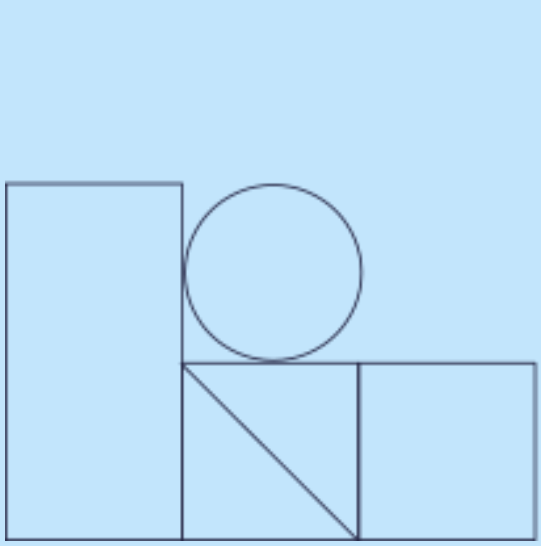
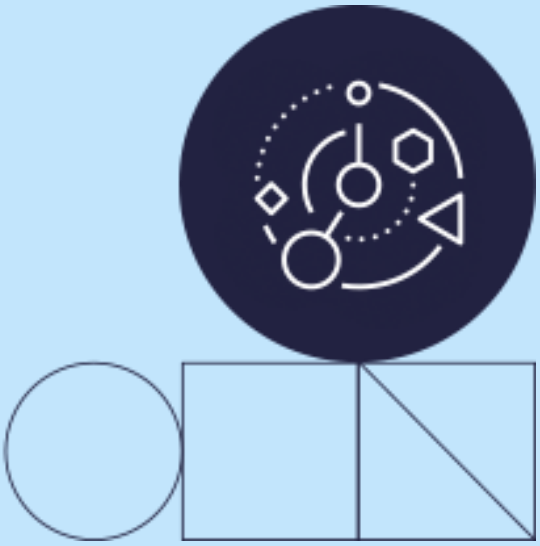
IBM SkillsBuild

IBM

Conceptos básicos y sintaxis de Python

Funciones y variables





Índice

Introducción	3
Ventajas de utilizar las funciones	4
Aumentan la reusabilidad de código y minimizan la redundancia (repetición)	4
Permiten la descomposición procedural	4
Sintaxis básica de una función en Python	4
Funciones y polimorfismo	5
Funciones anidadas	6
Recursividad	6
Devolviendo múltiples valores simultáneamente	6
Experimenta	6
Ámbito de una función	7
Ámbitos y sus propiedades	7
Resolución de nombres. La regla LEGB	8
Parámetros y argumentos	11
Argumentos de las funciones	11
Un vistazo a las distintas formas de paso de argumentos	12
Consideraciones de diseño al programar con funciones	16
Acoplamiento	16
Cohesión	17
Experimenta	17
Definición de variables	18
Asignar un valor a una variable en Python	18

Introducción

En esta unidad vamos a aprender cómo crear funciones en Python. Una función es un grupo de sentencias agrupadas de tal forma que pueden ser invocadas por un mismo nombre. Las funciones pueden devolver un resultado y ser parametrizadas para permitir que el resultado de las mismas sea diferente en función de cómo se la ha llamado.

Las funciones son también la unidad estructural más básica que maximiza la reusabilidad de código, por lo que nos permite avanzar hacia nociones de diseño de software más ambiciosas que las vistas hasta ahora. Esto es gracias a que las funciones nos permiten separar nuestros programas en pequeños bloques más manejables que pueden ser reutilizados en diversas partes de nuestro software. El hecho de implementar nuestro código en funciones hace que este sea más reusable, más fácil de programar y permite que otros programadores puedan entender nuestro código más fácilmente.

Ventajas de utilizar las funciones

Aumentan la reusabilidad de código y minimizan la redundancia (repetición)

Las funciones son la manera más simple y sencilla de empaquetar funcionalidades de manera que se puedan utilizar en diversas partes de un programa sin tener que repetir código. Nos permiten agrupar y generalizar código de manera que podamos utilizarlo arbitrariamente tantas veces como necesitemos. Esto convierte a las funciones en un elemento primordial de la factorización de código, lo que nos permite reducir la redundancia y, por lo tanto, reducir el esfuerzo necesario en mantener nuestro código.

Permiten la descomposición procedural

Es decir, permiten descomponer programas en pequeñas partes donde cada parte tiene un rol bien definido. Por lo general es más fácil programar pequeñas piezas de código y componerlas en programas más grandes que escribir todo el proceso de una sola vez.

En esta unidad vamos a conocer la sintaxis necesaria para operar con funciones, aprenderemos el concepto de ámbito (scope) y veremos cómo parametrizar nuestras funciones para que sean más genéricas. Con todo esto estaremos preparados para iniciar un camino mucho más ambicioso en el mundo de la programación en Python y que continuaremos en las próximas unidades y cursos de esta especialización.

Sintaxis básica de una función en Python

La sintaxis necesaria para declarar una función es la siguiente:

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

La declaración empieza con la sentencia **def** seguida del nombre que le queremos dar a la función. Seguidamente escribiremos un listado de 0 a N argumentos (también llamados parámetros de la función) encapsulados entre paréntesis y finalizamos la declaración con el carácter “:”.

Tras esto, y en una nueva línea, escribiremos el cuerpo de nuestra función, que consistirá en un grupo de sentencias a ejecutar finalizando con la sentencia opcional **return** acompañada del valor a devolver.

Para llamar a la función simplemente escribiremos el nombre de la misma seguida de los argumentos que le queremos pasar, encapsulados entre paréntesis. Veamos un par de ejemplos:

```
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 puedes ver, no es necesario utilizar ***return***. En las funciones que no tienen ***return***, devuelven ***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
```

Funciones y polimorfismo

Hemos comentado que una de las ventajas del uso de funciones es que permiten la reusabilidad de código. Esto es más cierto aún en Python, donde muchos tipos de datos soportan polimorfirsmo, es decir, cada tipo de dato sabe cómo comportarse ante una gran variedad de operadores. Esto es directamente aplicable al uso de funciones, por lo que podemos encontrarnos casos como el siguiente:

```
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
```

Esto es así porque en Python **las funciones no tienen tipo** (recordemos la unidad de Tipado Dinámico). En un gran número de casos el tipo de salida de una función dependerá del tipo de los parámetros que le pasemos(Hay excepciones a esta regla como por ejemplo que nosotros estemos forzando el tipo de salida en la sentencia ***return***). Esta es una idea central en el lenguaje que le dota de una gran flexibilidad y de facilidades de reusar a la hora de programar.

En Python una función no tiene por qué preocuparse de los tipos de entrada y salida. Es el propio intérprete el que se encargará de verificar que los tipos que le pasamos a la función soportan los protocolos que hayamos codificado en el interior de la misma. De hecho, si no los soporta, el intérprete se encargará de generar una excepción, lo que nos evita la tarea de tener que hacer control de errores en nuestra función. Si lo hiciéramos, estaríamos restando flexibilidad a la función, cosa que no suele ser deseable, salvo que sea una decisión de diseño.

Funciones anidadas

Es posible crear funciones dentro de funciones.

```
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 vemos es posible crear funciones dentro de otras funciones.

Recursividad

Al igual que en otros lenguajes de programación en Python una función puede llamarse a sí misma, generando recursividad. Es importante tener en cuenta que no se genere una recursividad infinita, es decir, la función debe tener una condición de salida. Un ejemplo muy habitual de recursividad en programación es la función que calcula el factorial de un número (recordemos que el factorial de x es igual a x * (x-1) * (x-2) * ... * 1

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

factorial(5)
```

En esta función vemos que la condición de salida de la recursividad se cumple cuando x es igual a 1.

Devolviendo múltiples valores simultáneamente

Las funciones pueden devolver cualquier objeto con la sentencia ***return.*** Por ello, si combinamos el devolver una tupla, con el desempaquetado extendido que permite Python, podemos simular la devolución de múltiples 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
```

Experimenta

Realiza una función que realice la descomposición en factores de un número. Deberá devolver una lista con los factores de dicho número. Recordad que la descomposición en factores de un número consiste en hallar el conjunto de números primos cuya multiplicación dé dicho número como resultado.

Pista: Lo primero que debe hacer la función es hallar todos los números primos inferiores al número en cuestión.

Ámbito de una función

Ahora que ya empezamos a operar con funciones, es el momento de avanzar un poco más con el concepto de nombres en Python. Cuando usamos un nombre en un programa, el intérprete de Python, busca ese nombre en un espacio de nombres al que llamamos ***namespace***. Un *namespace* es un lugar donde residen un conjunto de nombres. Si venís de otros lenguajes de programación este concepto os será familiar.

Cuando asignamos un nombre en Python, éste es asociado al espacio de nombres al que le corresponde. El namespace en el que reside un nombre define su ámbito ***(scope)***, es decir, la visibilidad que tendrá este nombre respecto a otras variables. Por ejemplo,

Los nombres asignados dentro del ámbito de una función, son sólo visibles por el código que reside dentro de esa misma función. Esto significa que un nombre declarado dentro de una función no puede ser referenciado desde fuera de ésta.

El ámbito de una variable depende del lugar donde se asignó. Dicho de otra manera, dependiendo el lugar donde asignamos una variable estaremos definiendo el ámbito en el que ésta puede ser utilizada. Las variables pueden ser asignadas en tres ámbitos:

- Si una variable es asignada en el interior de una función, es local a esa función.
- Si una variable es asignada dentro de una función, será nonlocal a todos los ámbitos de las funciones anidadas a la primera.
- Si una variable es asignada fuera de cualquier función esta es global al fichero en el que ha sido creada.

Veamos un ejemplo:

```
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
```

Ámbitos y sus propiedades

Como vemos, las funciones permiten crear namespaces anidados que limitan los ámbitos de acceso y evitan la colisión de nombres entre variables. Así, en las funciones creamos ámbitos locales que contrastan con el ámbito global de los módulos. Estos ámbitos tienen las siguientes propiedades:

- Las variables que se crean en ámbito del módulo son variables globales para ese fichero. Cuando importamos un módulo desde otro módulo, esas variables se convierten en atributos del módulo importado, por lo que pueden ser accedidas como simples variables del mismo. Por ejemplo, la variable math.pi es global al módulo math, pero si importamos el módulo math, podemos acceder a ella como si fuera un atributo del mismo (por lo que accedemos a ella mediante math.pi).

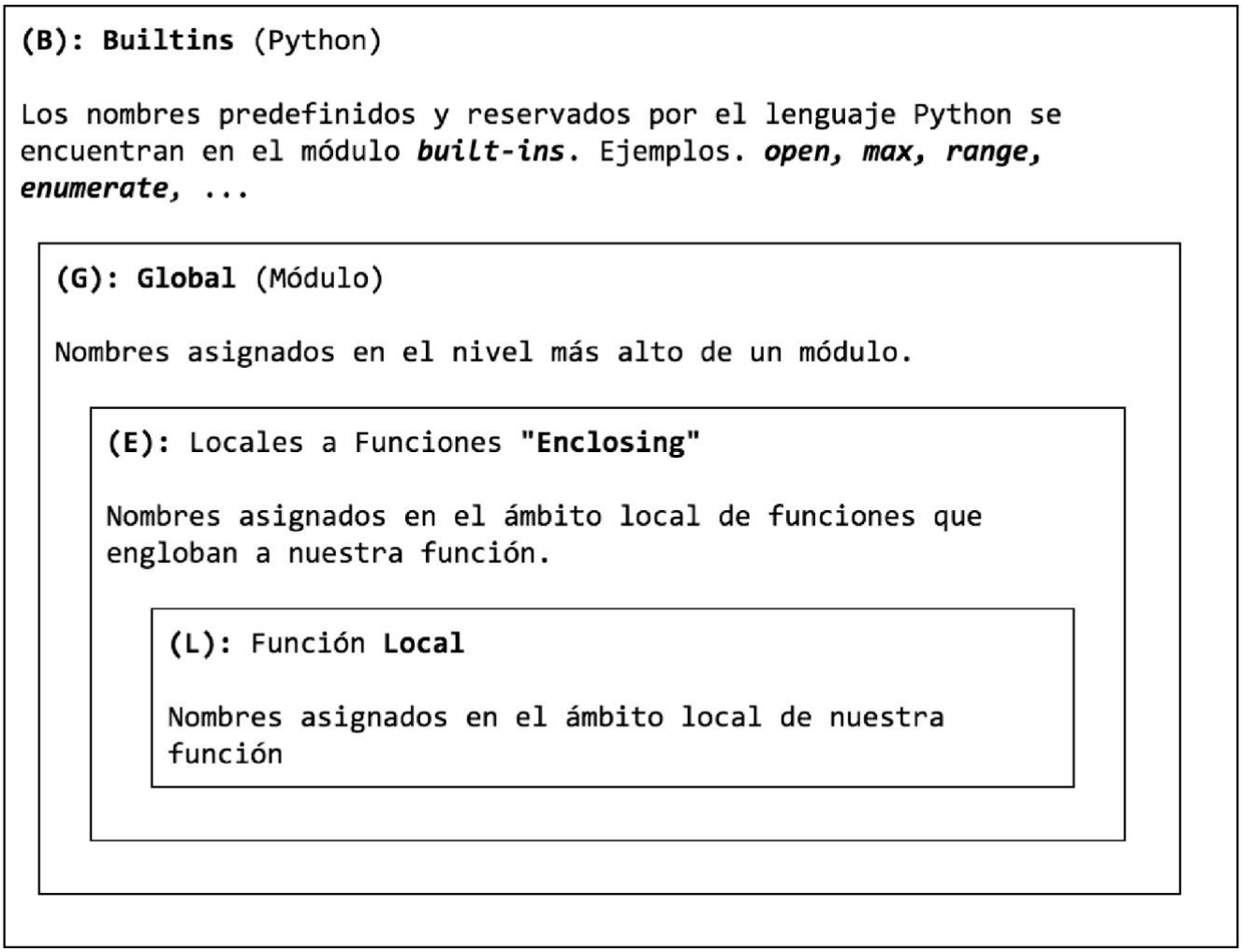
- El ámbito global es en realidad un ámbito de fichero. Es decir, el ámbito global afecta sólo al módulo (fichero) en el que se encuentran sus variables. En Python ámbito global significa ámbito de módulo.
- Los nombres asignados dentro de una función son locales por defecto. Todos los nombres definidos en el interior de una función tienen como ámbito el namespace local a esa función. Aun así, es posible realizar asignaciones a variables que fuera del ámbito local de la función. Esto se hace mediante utilizando las sentencias global y nonlocal en el interior de la función. No obstante, esta es una práctica generalmente no recomendada.
- Todos los otros nombres son locales a funciones que engloban a la nuestra (funciones en las que nuestra función es anidada), globales o built-ins. Los nombres no asignados en nuestra función son, o locales a funciones de un nivel de anidamiento superior (llamadas enclosing), o globales al módulo o bien pertenecen al módulo built-ins, un ámbito de mayor nivel donde se definen todos los tipos básicos de Python.
- Cada vez que llamamos a una función, se crea un ámbito local a esa función. Es decir, se crea un namespace donde se guardan todos los nombres de esa función.

Resolución de nombres. La regla LEGB

Todas las reglas anteriores se pueden resumir mediante la regla **LEGB (Local, Enclosing, Global, Builtin).** Esta regla dicta el orden de búsqueda de un nombre cuando intentamos acceder a él. Cuando intentamos acceder a un nombre, el intérprete de Python busca sólo en 4 ámbitos distintos. Además, esta búsqueda se hace siempre en el mismo orden.

- **L**: Primero se busca en el ámbito *local* de nuestra función.
- **E:** Tras ello se busca en las funciones que encierran a nuestra función *(enclosing)*
- **G:** Luego se busca en el ámbito *global* del módulo.
- **B:** Finalmente se busca en el módulo *builtins.*

La Figura 1 nos muestra un ejemplo de esta búsqueda y de cómo se engloban estos ámbitos de búsqueda. Notad que esta búsqueda termina en el primer lugar donde se encuentre el nombre buscado. En el caso en el que el nombre no se encuentre en estos ámbitos, Python devuelve un error.



Los ámbitos de búsqueda LEGB. Cuando se referencia una variable, Python realiza una búsqueda empezando a nivel **local** y subiendo hasta ***builtins.*** La búsqueda finaliza en el primer ámbito donde se encuentra el nombre referenciado.

Veamos a continuación un ejemplo de esta búsqueda.

```
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 vemos, ***f2*** es capaz de referenciar las variables ***E*** y ***G***. Ahora veamos qué pasa si declaramos esas mismas variables en el ámbito de ***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
```

En este caso, la búsqueda se detiene en el ámbito ***local*** de ***f2*** ya que en ese mismo ámbito hemos declarado variables con el mismo nombre (***E*** y ***G***). Por último, veamos qué pasa cuando intentamos acceder desde el ámbito de una función al de otra.

```
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
```

Esta vez, cuando intentamos acceder a una variable local a ***f2*** desde ***f3,*** nos salta un error de nombre. Esto es porque la búsqueda de nombres se hace desde el ámbito local de ***f3*** hacia arriba, pasando primero al ámbito local de ***f1,*** luego al **global** y, finalmente, a **builtins.** En ningún caso entramos a buscar en el ámbito de ***f2.*** Fijaos, sin embargo, que ***f3*** sí que tiene acceso a los ámbitos global y 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
```

Parámetros y argumentos

Argumentos de las funciones

Cuando definimos una función podemos parametrizarla mediante los argumentos que le pasamos. Por ejemplo, la siguiente función es capaz de multiplicar dos números cualquiera ya que la estamos definiendo mediante dos parámetros de entrada (argumentos):

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

suma(2, 3)

suma(40, 30)
```

Resultado:

```
5
70
```

Algunos aspectos que tenemos que tener en cuenta cuando pasamos argumentos a funciones en Python:

- Al pasarle un argumento a una función, estamos creando una asignación a una variable con el nombre del argumento en el ámbito local de la función.
- Asignar un nuevo valor al argumento desde dentro de la función, no afecta al exterior.
- Si le pasamos un objeto mutable a una función, y esta lo modifica en su interior, puede afectar al exterior.

Esto es así porque los argumentos en Python se ***pasan por referencia.*** Si recordamos lo visto en la unidad de Tipado Dinámico, entenderemos mejor lo que esto significa.

- Si pasamos un objeto inmutable, es como si lo hiciéramos por valor. Es decir, al no poder modificar el objeto, es como si hiciéramos una copia del mismo al ámbito local de la función.
- Si pasamos un objeto mutable, al hacerlo por referencia, cualquier cambio dentro del objeto que se haga dentro de la función será observado desde fuera de la misma.

Aquí tenemos un par de ejemplos. Empecemos pasando objetos inmutables:

```
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
```

En cambio, cuando pasamos un objeto mutable a una función que lo modifica internamente, el cambio afecta al exterior de la función:

```
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]
```

Recordad que para evitar este comportamiento podemos hacer una copia de la lista que le pasamos a la función:

```
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]
```

Un vistazo a las distintas formas de paso de argumentos

Por defecto, los argumentos se pasan por posición. Es decir, los argumentos de la llamada a una función tienen que pasarse en el mismo orden en el que se definió la función.

Sin embargo, una de las grandes ventajas de Python es que nos permite pasarles argumentos a las funciones de distintas maneras.

Por posición.

La manera por defecto de pasar argumentos, de izquierda a derecha.

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

f(1, 2, 3)
```

Resultado:

```
✓ def f(a, b, c): ···

1 2 3
```

Por keywords (palabras clave)

En lugar de llamar a la función con sus argumentos en orden, se le pasan especificando el nombre del argumento seguido del valor que le queremos pasar. Se utiliza la sintaxis nombre=valor. Al estar indicando el nombre de los argumentos explícitamente, no hace falta que estos estén ordenados por posición.

```
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
```

Especificando valores por defecto en la definición de la llamada

Es posible definir qué valores por defecto tienen los argumentos de una función. Así, si al llamar a la función, no le pasamos alguno de los argumentos que tiene un valor por defecto, se utilizará dicho valor.

```
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
```

Especificando en la función que se le pasará una colección de argumentos

En la definición de la función se indica que se le pasará un número arbitrario de argumentos que estarán ordenados por posición de izquierda a derecha (indicado con un asterisco * justo antes del nombre) o por keywords (indicado con un doble asterisco ** justo antes del nombre).

```
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)
```

Si usamos la sintaxis de doble asterisco, especificamos que le pasaremos los argumentos por nombre:

```
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}
```

Desempaquetando una colección de argumentos posicionales o por keyword

Cuando llamamos a una función se le pude utilizar la sintaxis de * para desempaquetar una colección en una serie de argumentos separados por posición.

```
✓ 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
```

Así mismo, se puede desempaquetar un diccionario utilizando la sintaxis de doble 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
```

Utilizando argumentos que sólo pueden ser pasados por clave (keyword-only)

Este tipo de argumentos sólo puede ser pasado por clave (keyword) y que nunca serán rellenados por posición. Esto es útil en funciones que pueden recibir cualquier número de argumentos y aun así permiten configuraciones opcionales.

```
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
```

Los argumentos tras el '*' se convierten en *keyword-only*:

Veremos que hay muchas funciones escritas en Python que aprovechan estas funcionalidades como, por ejemplo, las funciones builtin **zip** y **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:

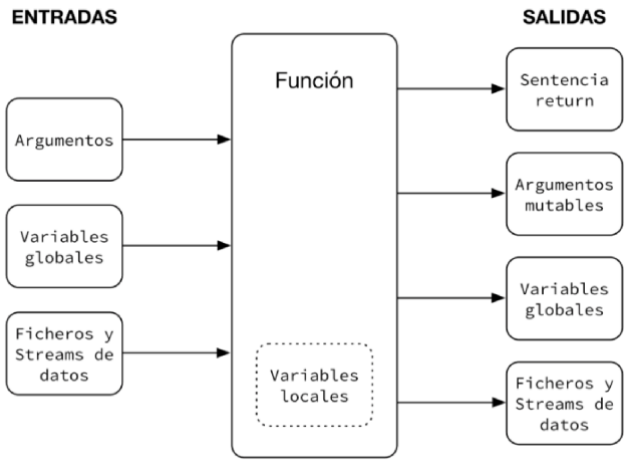
```
✓ la = [1, 2, 3, 4, 5] ···

[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 vemos este tipo de paso de argumentos dota a las funciones de Python de una gran flexibilidad. Además, el uso de argumentos por defecto y de argumentos *keyword-only* fomentan la simplicidad y legibilidad de nuestro código.

Consideraciones de diseño al programar con funciones

Para finalizar esta unidad sobre funciones, acabaremos con algunas consideraciones de diseño que deberéis tener en cuenta cuando programéis con ellas. Para ello es necesario entender qué entradas y salidas posibles tiene cualquier función. Tenéis un detalle en la Figura 2.



En la figura se describen los principales mecanismos con los que una función interacciona con el mundo que la rodea. Cuando programéis funciones, es importante tener en cuenta dos aspectos importantes de diseño que ayudan a que un código sea más usable y mantenible. Estos aspectos son:

- **Cohesión**: se refiere a como se descompone un programa en pequeñas tareas individuales.
- **Acoplamiento**: se refiere a como se comunican las funciones y como dependen entre ellas.

Un buen código está altamente cohesionado y tiene poco acoplamiento. Es decir, todas las funciones de vuestro código ayudan a una misma tarea, pero no dependen unas de otras. A continuación, podéis ver un listado de directrices que os ayudarán ayudan a aumentar la cohesión y reducir el acoplamiento.

Acoplamiento

- Crea funciones que sólo reciben inputs por argumentos y sólo usan la sentencia ***return*** para su salida. Esto reduce las dependencias externas de las funciones. Una función ***f(args)*** que cumple este criterio siempre devuelve el mismo resultado para unos mismos argumentos. Es decir, la función se hace más predecible y, por tanto, más fácil de entender y de mantener.
- Utiliza variables globales sólo cuando sea realmente necesario. Las variables globales crean dependencias entre funciones y hacen que los programas sean difíciles de debugar, modificar y reusar.
- Evita cambiar argumentos mutables salvo que el código que hace la llamada así lo espera. El problema es el mismo que con las variables globales, pero además crea fuerte acoplamiento entre la función que llama y la función llamada.
- No cambies atributos de otros módulos directamente. Esto crea un fuerte acoplamiento entre módulos al igual que las variables globales lo hacen entre funciones. Si necesitáis cambiar algún atributo de un módulo usad las funciones que el propio módulo os ha proporcionado para ello. Si no tiene, entonces seguro que cambiar sus atributos puede ser una muy mala idea.

Cohesión

- Cada función debe tener un único propósito. Un código bien diseñado está compuesto de funciones que hacen una sola cosa. Normalmente el propósito de una función debería poder resumirse en una frase corta. Si la frase que describe vuestra función es muy genérica (p. ej. *"esta función resuelve mi problema"*) o si tiene muchas conjunciones (p. ej. "*esta función calcula X y procesa Y"*), considerad separarla en partes más simples.
- Todas las funciones deberían ser relativamente pequeñas. Este punto está relacionado con el anterior. Tened en cuenta que el código escrito en Python es muy conciso. Si vuestra función ocupa más de una docena o dos de líneas o contiene varios niveles de anidación, es probable que estéis ante un problema de diseño.

En resumen, intentad que vuestro código se componga de muchas funciones pequeñas e independientes entre ellas.

Experimenta

Crea una función log que acepte cualquier número de argumentos y los imprima por pantalla en una misma línea. La línea debe empezar con el prefijo 'LOG: '.

Modifica la función log para que usuario pueda especificar cualquier prefijo que desee.

Modifica la función log para que el prefijo tenga el valor por defecto 'LOG: '.

Modifica la función log para que el usuario pueda establecer tanto prefijo como separador entre argumentos. Ambos deben pasarse sólo por los nombres (no por posición) 'sep' y 'prefix'. No hace falta que estos tengan valor por defecto.

Modfica la función log para que ahora 'sep' y 'prefix' tengan un valor por defecto.

Modifica la función log para que acepte el siguiente diccionario. Recuerda que hay que pasarlo desempaquetándolo con la sintaxis de doble asterisco (**).

Definición de variables

Las variables son uno de los dos componentes básicos de cualquier programa.

En su esencia, un programa está compuesto por datos e instrucciones que manipulan esos datos. Normalmente, los datos se almacenan en memoria (memoria RAM) para que podamos acceder a ellos.

Entonces, ¿qué es una variable? Una variable es una forma de identificar, de forma sencilla, un dato que se encuentra almacenado en la memoria del ordenador o dicho de otra manera, un espacio en la memoria del ordenador en el que se guardará un valor que podrá cambiar a lo largo de la ejecución del programa. Imaginemos que una variable es un contenedor en el que se almacena un dato, el cual, puede cambiar durante el flujo del programa. Una variable nos permite acceder fácilmente a dicho dato para ser manipulado y transformado.

Una variable es un concepto fundamental en cualquier lenguaje de programación. Es una ubicación de memoria reservada que almacena y manipula datos. En algunos lenguajes de programación, las variables se pueden entender como "cajas" en las que se guardan los datos, pero en Python las variables son "etiquetas" que permiten hacer referencia a los datos (que se guardan en unas "cajas" llamadas objetos).

A diferencia de otros lenguajes de programación, Python no tiene comandos para declarar una variable. Una variable se crea en el momento en que se le da un nombre y se le asigna un valor.

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

Supongamos que queremos mostrar el resultado de sumar 1 + 2. Para mostrar el resultado, debemos indicarle al programa dónde se encuentra dicho dato en memoria y, para ello, hacemos uso de una variable:

```
# 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)
```

Asignar un valor a una variable en Python

Tal y como hemos visto en el ejemplo anterior, para asignar un valor (un dato) a una variable se utiliza el operador de asignación `=`.

En la operación de asignación se ven involucradas tres partes:

- El operador de asignación `=`
- Un identificador o nombre de variable, a la izquierda del operador.
- Un literal, una expresión, una llamada a una función o una combinación de todos ellos a la derecha del operador de asignación.
-

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'
```

Cuando asignamos un valor a una variable por primera vez, se dice que en ese lugar se define e inicializa la variable. En un script o programa escrito en Python, podemos definir variables en cualquier lugar del mismo. Sin embargo, es una buena práctica definir las variables que vayamos a utilizar al principio.

Si intentamos usar una variable que no ha sido definida/inicializada previamente, el intérprete nos mostrará un error:

```
print(a)
```

Estrechamente relacionado con las variables se encuentra el concepto de tipo de dato. Al asignar un valor a una variable, dicho valor pertenece a un conjunto de valores conocido como tipo de dato. Un tipo de dato define una serie de características sobre esos datos y las variables que los contienen como, por ejemplo, las operaciones que se pueden realizar con ellos. En Python, los tipos de datos básicos son los numéricos (enteros, reales y complejos), los boolenaos (True, False) y las cadenas de caracteres.

Los tipos de datos, su declaración y la forma de trabajar con ellos se verá con profundidad en el siguiente tema.