

06 – Funciones en Python 3

Diego Andrés Álvarez Marín
Profesor Asociado
Universidad Nacional de Colombia
Sede Manizales

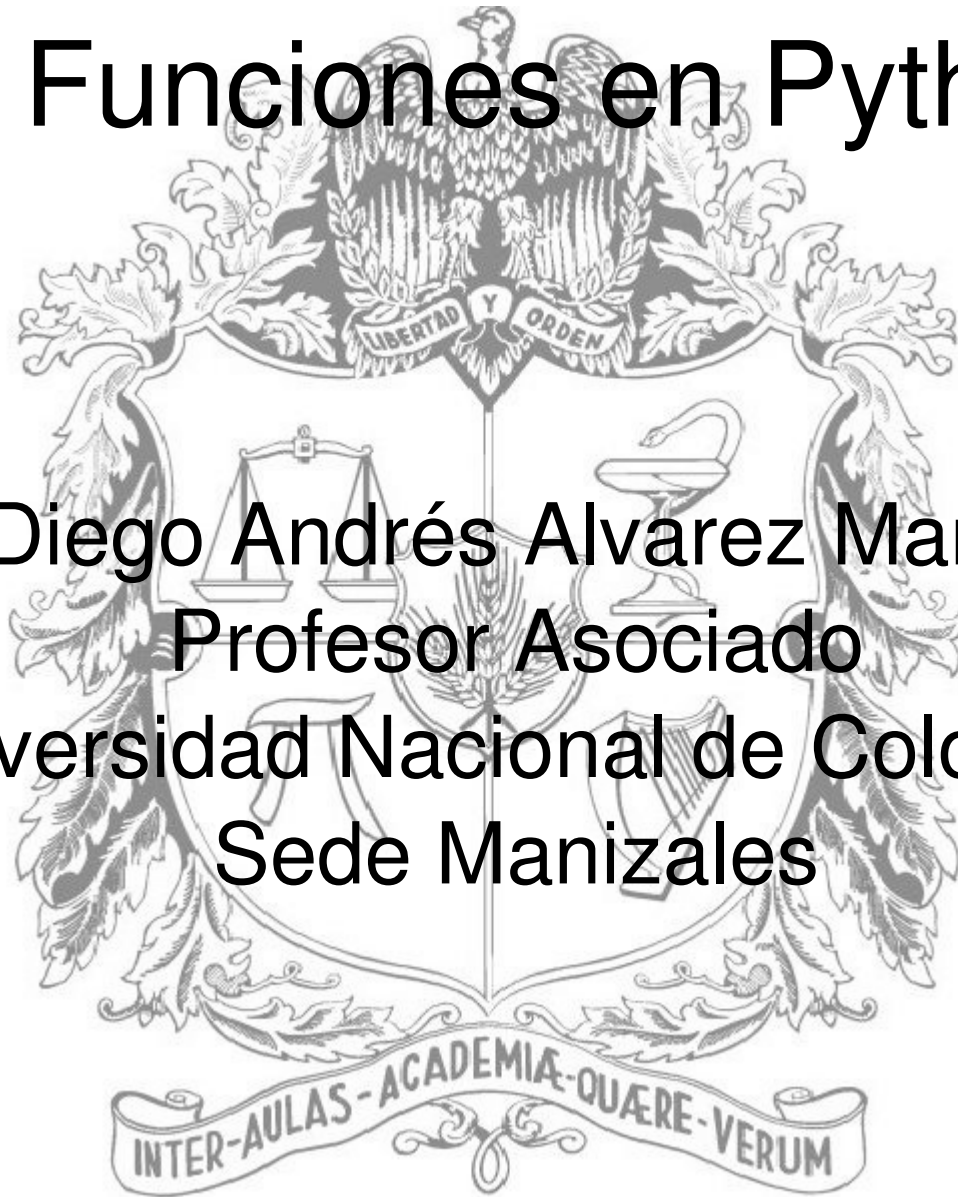


Tabla de contenido

- Funciones vs. Procedimientos vs. Métodos
- Definición de funciones
- Ámbito de una variable
- Paso de parámetros a variables: valor, referencia y “asignación”
- Variables globales
- Variables locales
- Tabla de símbolos
- Pila de llamadas
- return, global, nonlocal
- Argumentos opcionales y con nombre
- Funciones variádicas
- Expresiones Lambda
- Docstrings
- Funciones recursivas
- Funciones anidadas
- Funciones anónimas (lambda)
- Pasando funciones como argumentos de funciones

Divide y vencerás: la programación funcional

Divida un problema grande en muchos problemas pequeños

http://en.wikipedia.org/wiki/Comparison_of_programming_paradigms

Además al utilizar funciones:

- Se incrementa la claridad del mismo
- Permite reutilizar el código

¿Qué es una función?

¿Qué es un procedimiento?

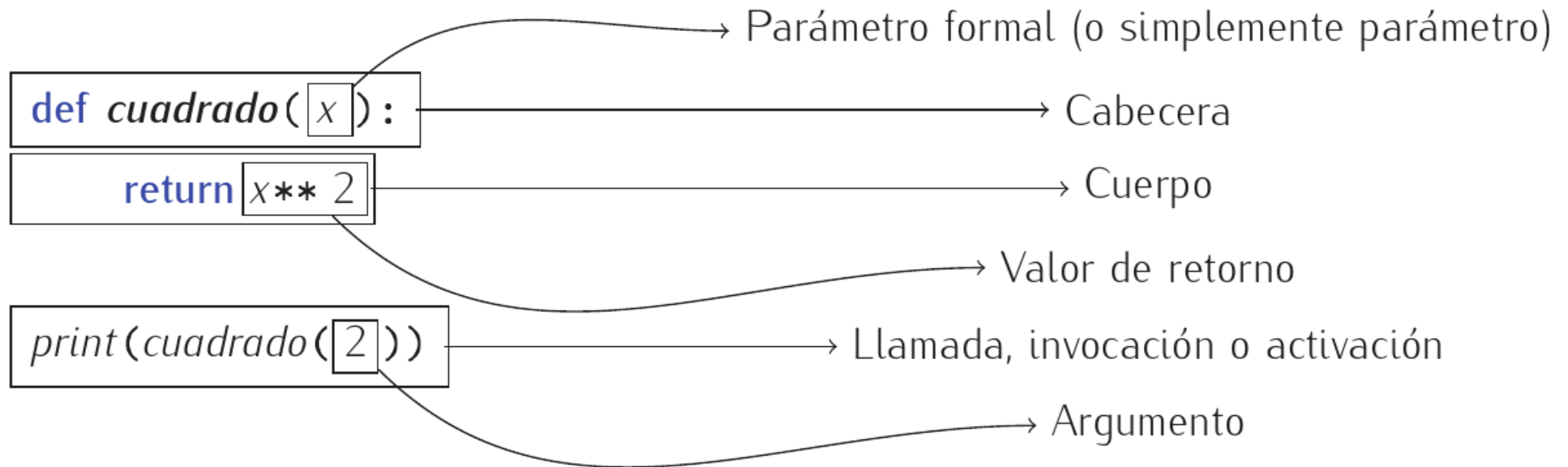
Una **función** es un fragmento de código con un nombre asociado que realiza una serie de tareas y devuelve un valor. A los fragmentos de código que tienen un nombre asociado y no devuelven valores se les suele llamar **procedimientos**. En Python no existen los procedimientos, ya que cuando el programador no especifica un valor de retorno, la función devuelve el valor **None**.

Distribución recomendada del código en un programa

- Se hace un comentario inicial (docstring) que describa brevemente la finalidad del programa.
- Se llaman las librerías (import)
- Se definen las constantes globales **que sean llamadas desde las funciones**
- Se reserva el espacio en memoria para las variables globales que lo requieran y **que sean llamadas desde las funciones**
- Se crean las funciones
- Finalmente se hace el procedimiento principal
- Las variables locales a un procedimiento se inicializan tan pronto como se necesiten.

Funciones

Las palabras reservadas **def** y **lambda** se utilizan para crear funciones.



```
>>> cuadrado = lambda x: x**2  
>>> print(cuadrado(2))
```

4

Esto es similar al uso
de “function handles”
(@) en MATLAB

Definición de funciones

En Python las funciones no tienen prototipo o archivos .h (como en C/C++); solo tienen declaración:

```
def tamaño_aproximado(tamaño, exacto=True):
```

Las funciones no definen un tipo de dato de retorno. Si la función no tiene un `return`, retornará un `None`.

Hasta Python 3.5, a las funciones tampoco se les especificaba el tipo de los parámetros (argumentos) de entrada; esto si se hace en C, C++ o Pascal. Sin embargo a partir de Python 3.5 esto ya es opcional, e incluso es recomendado, y se le conoce como `type annotations`. Ver estas diapositivas.

Definición y uso de funciones sin parámetros

Vamos a considerar ahora cómo definir e invocar funciones sin parámetros. En realidad hay poco que decir: lo único que debes tener presente es que es obligatorio poner paréntesis a continuación del identificador, tanto al definir la función como al invocarla.

En el siguiente ejemplo se define y usa una función que lee de teclado un número entero:

lee_entero.py

```
1 def lee_entero():  
2     return int(input())  
3  
4 a = lee_entero()
```

Recuerda: al llamar a una función los paréntesis *no* son opcionales.


```

>>>
def fib(n):
    """Escriba la serie de Fibonacci hasta el número n."""
    a, b = 0, 1
    while a < n:
        print(a, end=' ')
        a, b = b, a+b
    print()

>>> fib(200)
0 1 1 2 3 5 8 13 21 34 55 89 144
>>> fib
<function fib at 0x7fc6b3fcab70>
>>> resultado = fib(200)
0 1 1 2 3 5 8 13 21 34 55 89 144
>>> resultado
>>> print(resultado)
None
>>> print(fib(200))
0 1 1 2 3 5 8 13 21 34 55 89 144
None
>>> |

```

`fib` se entiende como una función cuya ubicación en memoria está en la dirección mostrada. `fib()` es la invocación a la función.

Como `fib()` no retorna nada, se entiende que retorna un `None`

Renombrando funciones

```
>>> fib
<function fib at 0x7fc6b3fcab70>
>>> f = fib
>>> f(200)
0 1 1 2 3 5 8 13 21 34 55 89 144
>>> f
<function fib at 0x7fc6b3fcab70>
>>> |
```

- Observe que `f` y `fib` (sin los paréntesis) son referencias a la misma dirección de memoria.

Funciones vs. Métodos

- función(arg1, arg2, arg3)
- arg1.método(arg2, arg3)

```
>>> len('Hola')
```

```
4
```

```
>>> 'Hola'.lower()
```

```
'hola'
```

```
>>> 'Hola'.upper()
```

```
'HOLA'
```

```
>>> |
```

} Función

} Método

{ El diseño de métodos se aprenderá cuando se estudie un tema llamado “programación orientada a objetos”.

Ámbito de variables (variable scope)

El ámbito de una variable determina el rango del código sobre el cual dicha variable existe y por lo tanto puede ser utilizada. Esto sirve para que se pueda volver a definir una variable con un mismo nombre en diferentes partes del programa sin que hayan conflictos entre ellas.

Se recomienda que el ámbito de una variable sea tan pequeño como sea posible, de modo que uno como programador minimice el número de variables que tenga que considerar cuando se programe, lo cual redundará en menores errores de programación.

Ámbito de variables (variable scope)

http://en.wikipedia.org/wiki/Scope_%28computer_science%29

Si una variable es declarada dentro de una función, ésta será válida solo dentro de esa función y se destruirá al terminar la función. Adicionalmente, la variable no podrá verse ni usarse fuera la función. La variable dentro de la función es una **variable local** y solo tiene alcance dentro de la función en que se creó y dentro de sus funciones hijas (anidadas), pero no en funciones hermanas ni en las funciones padres.

Una variable definida fuera de cualquier función es una **variable global** y cualquier función puede acceder a ella y modificarla a través de las palabras clave **global** y **nonlocal**.

Ámbito de las variables

- **Variables globales:** pueden leerse dentro y fuera de las funciones, pero solamente pueden modificarse por fuera de las funciones, o usando la palabra reservada **global**.
- **Variables locales:** solamente pueden leerse y escribirse dentro de la función que la definió. El valor de la variable local no se mantiene entre llamados a funciones.
- **Variables estáticas:** son variables que conservan su valor entre llamados a funciones. Esta palabra clave no existe en Python, pero si en lenguaje C/C++. Se pueden implementar en Python mediante un truco con respecto a la definición de parámetros por defecto.

Variables globales

Póngale a las variables globales un nombre con sentido.

<code>a, b, c, i, j</code>	<code># nombres inadecuados para una var. global</code>
<code>tablero_juego</code>	<code># buen nombre para una variable global</code>
<code>NUM_MAX_ITERAC</code>	<code># buen nombre para una constante global</code>

Únicamente declare variables globales cuando sea estrictamente necesario. Nunca las utilice si una variable local o estática podría hacer el mismo trabajo.

Cambiando variables globales desde dentro de una función

Si se trata de cambiar el valor de una variable global, dentro de una función, se creará una variable local dentro de la función con el mismo nombre que la variable global. La variable global ya no será accesible dentro de la función. Solamente se tendrá acceso a la variable local.

Si se quiere cambiar el contenido de la variable global dentro de la función se debe utilizar la palabra reservada **global**.

Cambiando variables globales desde funciones

Modificar las variables globales desde una función no es una buena práctica de programación. Únicamente en contadas ocasiones esta justificado que una función modifique las variables globales. Se dice que modificar variables globales desde una función es un efecto secundario de la llamada a la función.

Si cada función de un programa largo modificara libremente el valor de las variables globales, el programa sería bastante difícil de entender, y por lo tanto de corregir en el futuro.

La palabra reservada “global”

```
1 def mifuncion1():
2     x = 99 # este "x" es una variable local a mifuncion1()
3     print("x (mifuncion1) =", x)
4
5 def mifuncion2():
6     global x # se declara "x" como global para poder modificarla
7     x = -21 # este "x" es la variable global
8     print("x (mifuncion2) =", x)
9
10 def mifuncion3():
11     print("x (mifuncion3) =", x) # este "x" hace referencia a la variable global
12
13     x = 42 # este "x" es una variable global
14     print("x =", x)
15
16     mifuncion1()
17     print("x =", x) # mifuncion1() no cambió el contenido de "x"
18
19     mifuncion2()
20     print("x =", x) # mifuncion2() cambió el contenido de "x"
21
22     mifuncion3()
```

Line: 23 of 23 Col: 1 LINE INS

daalvarez@eredron:~ > python3 04_variables_globales_locales.py

```
x = 42
x (mifuncion1) = 99
x = 42
x (mifuncion2) = -21
x = -21
x (mifuncion3) = -21
```

```

1 ▼ def fun1(s):
2     x = 3; y = s # x, y y s son variables locales
3     print('fun1: x = {0}, y = {1}'.format(x,y))
4
5 ▼ def fun2(s):
6     x = 1; y = s # x, y y s son variables locales
7     print('fun2: x = {0}, y = {1}'.format(x,y))
8
9 ▼ def fun3(s):
10    global x      # x es la variable global
11    x = 1; y = s  # y y s son variables locales
12    print('fun3: x = {0}, y = {1}'.format(x,y))
13
14
15    x = 5;      # este x es una variable global
16    print('x = {0}'.format(x))
17    fun1(4); print('x = {0}'.format(x))
18    fun2(2); print('x = {0}'.format(x))
19    fun3(2); print('x = {0}'.format(x))

```

Line: 21 of 21 Col: 1

LINE INS

06

daa@heimdall ~ \$ python3 06_global_vs_local.py

x = 5

fun1: x = 3, y = 4

x = 5

fun2: x = 1, y = 2

x = 5

fun3: x = 1, y = 2

x = 1

daa@heimdall ~ \$ █

```

1  #include <stdio.h>
2
3  int x = 100; int y = 100; int z = 100; // Variables globales
4
5  int suma(int x, int y)
6  {
7      z = 300; // z es la variable global
8      return x + y; // x, y son variables locales
9  }
10
11 int multiplicacion(int x, int y)
12 {
13     y = x * y; // x, y son variables locales
14     z += 10; // z es la variable global
15     return y;
16 }
17
18 int agrega_50(int x)
19 {
20     int z = 50; // x, z son variables locales
21     y = x + z; // y es la variable global
22     return y;
23 }
24
25 int main(void)
26 {
27     printf("result = %d\n", suma(2,3) + multiplicacion(3,4) + agrega_50(1) + z);
28     for(int i = 0; i<5; i+=1) printf("3x4 = %d\n", multiplicacion(3,4));
29     printf("x = %d, y = %d, z = %d\n", x, y, z);
30     return 0;
31 }

```

Nombre no recomendado para las variables globales

Programa hecho en lenguaje C

Line: 33 of 33 Col: 1 LINE INS

daalvarez@eredron:~ > gcc -Wall 06_porque_no_abusar_de_variables_globales.c -std=c99

daalvarez@eredron:~ > ./a.out

result = 378

3x4 = 12

3x4 = 12

3x4 = 12

3x4 = 12

3x4 = 12

x = 100, y = 51, z = 360

Si cada función de un programa largo modificara libremente el valor de las variables globales, el programa sería bastante difícil de entender, y por lo tanto de corregir en el futuro. Por lo tanto, es muy mala práctica de programación hacer uso indiscriminado de las variables globales.

Variables estáticas

https://en.wikipedia.org/wiki/Static_variable

Las variables estáticas son variables locales a una función que conservan su valor entre llamados de funciones.

```
1 def foo():  
2     foo.contador += 1  
3     print("El contador va en %d" % foo.contador)  
4     foo.contador = 0 # inicialización de la variable estática  
5  
6     foo()  
7     foo()  
8     foo()  
9     foo()
```

Line 10, Column 1

INSERT Soft Tabs: 4 UTF-8 Pyl

```
daalvarez@eredron ~ $ python3 06_static_variables.py
```

```
El contador va en 1
```

```
El contador va en 2
```

```
El contador va en 3
```

```
El contador va en 4
```

```
daalvarez@eredron ~ $ █
```

Paso de parámetros: por valor o por referencia

- **Por valor:** se hace una copia local de la información dentro de la función y por lo tanto el valor original no cambia.
- **Por referencia:** se transfiere la dirección de memoria de la variable con un puntero y por lo tanto cualquier cambio a la variable hecha por la función se observa en la función invocadora. Se hace para:
 - o para cambiar el valor de la variable dentro de la función
 - o cuando los datos a pasar ocupan mucho espacio en memoria y se quiere evitar perder tiempo creando una copia de los datos; en este caso se debe tener la precaución de no modificar los datos.

```

1  #include <iostream>
2
3  using namespace std;
4
5  void intercambio_valor(int x, int y)
6  {
7      int tmp;    // x, y y tmp son variables locales
8      tmp = x;
9      x = y;
10     y = tmp;
11 }
12
13 void intercambio_referencia(int &x, int &y)
14 {
15     int tmp;    // tmp es una variable local
16     tmp = x;    // x, y son referencias
17     x = y;
18     y = tmp;
19 }
20
21 int main (void)
22 {
23     int a = 100, b = 200; // a y b son variables locales a main()
24
25     cout << "a = " << a << " b = " << b << endl;
26     intercambio_valor(a,b);    // paso de parámetros por valor
27     cout << "a = " << a << " b = " << b << endl;
28     intercambio_referencia(a,b); // paso de parámetros por referencia
29     cout << "a = " << a << " b = " << b << endl;
30
31     return 0;
32 }

```

Paso de parámetros por valor y por referencia en un programa hecho en C++

Line: 34 of 34 Col: 1

LINE OVR

daalvarez@eredron:~ > g++ -Wall 06_paso_parametros_valor_referencia.cpp

daalvarez@eredron:~ > ./a.out

a = 100 b = 200

a = 100 b = 200

a = 200 b = 100

Paso de parámetros a una función en Python

Los parámetros de una función siempre se pasan como si se hubieran pasado con el operador `=`. Esto lo llaman en la comunidad Python en español **paso de parámetros por asignación** (en inglés **call-by-sharing** o **call-by-assignment**). Es decir, cuando se pasan listas o diccionarios, se pasa una referencia al objeto, no el valor del objeto.

```
>>> def fun(a,b):  
        a.append('xyz')  
        b = b+10
```

```
>>> L = [1, 2, 3]  
>>> x = 10  
>>> fun(L, x)  
>>> L  
[1, 2, 3, 'xyz']  
>>> x  
10  
>>> fun(L[:], x)  
>>> L  
[1, 2, 3, 'xyz']  
>>> x  
10  
>>> fun(L, x)  
>>> L  
[1, 2, 3, 'xyz', 'xyz']  
>>> |
```


Tipos de datos

mutables vs inmutables

- **INMUTABLES:** An object with a fixed value. Immutable objects include numbers, strings and tuples. Such an object cannot be altered. A new object has to be created if a different value has to be stored. They play an important role in places where a constant hash value is needed, for example as a key in a dictionary: ejemplo: floats, cadenas, tuplas
- **MUTABLES:** Mutable objects can change their value but keep their id(): ejemplo: listas, conjuntos, diccionarios

```
>>> x = 1; id(x)
10455040
>>> x = 2; id(x)
10455072
>>> x = 3; id(x)
10455104
>>> c = 'a'; id(c)
140284911786392
>>> c = 'b'; id(c)
140284912055440
>>> c = 'c'; id(c)
140284911986872
>>> L = [1]; id(L)
140284885525320
>>> L.append(2); id(L)
140284885525320
>>> L.append(3); id(L)
140284885525320
```

If you're used to most traditional languages (C/C++), you have a mental model of what happens in the following sequence:

```
a = 1  
a = 2
```

You believe that `a` is a memory location that stores the value `1`, then is updated to store the value `2`. That's not how things work in Python. Rather, `a` starts as a reference to an object with the value `1`, then gets reassigned as a reference to an object with the value `2`. Those two objects may continue to coexist even though `a` doesn't refer to the first one anymore; in fact they may be shared by any number of other references within the program.

When you call a function with a parameter, a new reference is created that refers to the object passed in. This is separate from the reference that was used in the function call, so there's no way to update that reference and make it refer to a new object.

Los argumentos a una función se pasan por asignación en Python

Las razones que sustentan esto son dos:

- El parámetro que se pasa en realidad es una referencia a un objeto (pero la referencia se pasa por valor)
- Algunos tipos de datos son mutables, pero otros no lo son

Por lo tanto:

- Si pasa un **objeto mutable** en una función, la función obtiene una referencia a ese mismo objeto y este se puede mutar como uno quiera, pero si se cambia esta referencia dentro de la función, el ámbito exterior no sabrá nada al respecto, y después que haya terminado, la referencia externa todavía apuntará al objeto original.
- Si pasa un **objeto inmutable** a una función, usted no puede cambiar la referencia externa; ni siquiera podrá mutar el objeto. 27

Pasando un tipo mutable (una lista)

```
1 def intentemos_cambiar_el_contenido_de_la_lista(L):
2     '''
3     Esta función modifica la lista que se pasa a la función
4     '''
5     print ('recibió', L)
6     L.append('four')
7     print ('cambió a', L)
8
9 def intentemos_cambiar_la_referencia_a_la_lista(L):
10    '''
11    Esta función modifica la referencia a la lista que se pasó como parámetro
12    a la función
13    '''
14    print ('recibió', L)
15    L = ['cambiamos', 'la', 'referencia', 'L']
16    print ('cambió a', L)
17
18 lista_exterior = ['one', 'two', 'three']
19 print ('antes, lista_exterior =', lista_exterior)
20 intentemos_cambiar_el_contenido_de_la_lista(lista_exterior)
21 print ('después, lista_exterior =', lista_exterior)
22
23 print(80*'-')
24
25 lista_exterior = ['one', 'two', 'three']
26 print ('antes, lista_exterior =', lista_exterior)
27 intentemos_cambiar_la_referencia_a_la_lista(lista_exterior)
28 print ('después, lista_exterior =', lista_exterior)
```

Como el parámetro que se pasó es una referencia a lista_exterior, no una copia de lista_exterior, se puede modificar la lista y estos cambios se verán en el ámbito exterior.

En la línea 15 se cambia la referencia L (es decir L queda apuntando a un nuevo objeto), por lo que cualquier cambio a L no tendrá efecto en lista_exterior.

```
<
Line: 30 of 30 Col: 1    LINE    INS
daalvarez@eredron:~ > python3 06_pasando_mutables_como_parametro_a_funcion.py
antes, lista_exterior = ['one', 'two', 'three']
recibió ['one', 'two', 'three']
cambió a ['one', 'two', 'three', 'four']
después, lista_exterior = ['one', 'two', 'three', 'four']
-----
antes, lista_exterior = ['one', 'two', 'three']
recibió ['one', 'two', 'three']
cambió a ['cambiamos', 'la', 'referencia', 'L']
después, lista_exterior = ['one', 'two', 'three']
```

Pasando tipos inmutables

```
1 def intentemos_cambiar_los_contenidos_de_los_parametros(num, boo, cad, tupla):
2     print('recibió', num, boo, cad, tupla)
3     print('ids = ', hex(id(num)), hex(id(boo)), hex(id(cad)), hex(id(tupla)))
4     num += 10
5     boo = not boo
6     cad += 'xxx'
7     tupla = (10,20,30)
8     print('cambió a', num, boo, cad, tupla)
9     print('ids = ', hex(id(num)), hex(id(boo)), hex(id(cad)), hex(id(tupla)))
10
11 a = 1
12 b = True
13 c = 'Hola'
14 d = ('a','b','c')
15
16 print('antes de llamar función: ', a, b, c, d)
17 print('ids = ', hex(id(a)), hex(id(b)), hex(id(c)), hex(id(d)))
18 intentemos_cambiar_los_contenidos_de_los_parametros(a, b, c, d)
19 print('ids = ', hex(id(a)), hex(id(b)), hex(id(c)), hex(id(d)))
--
```

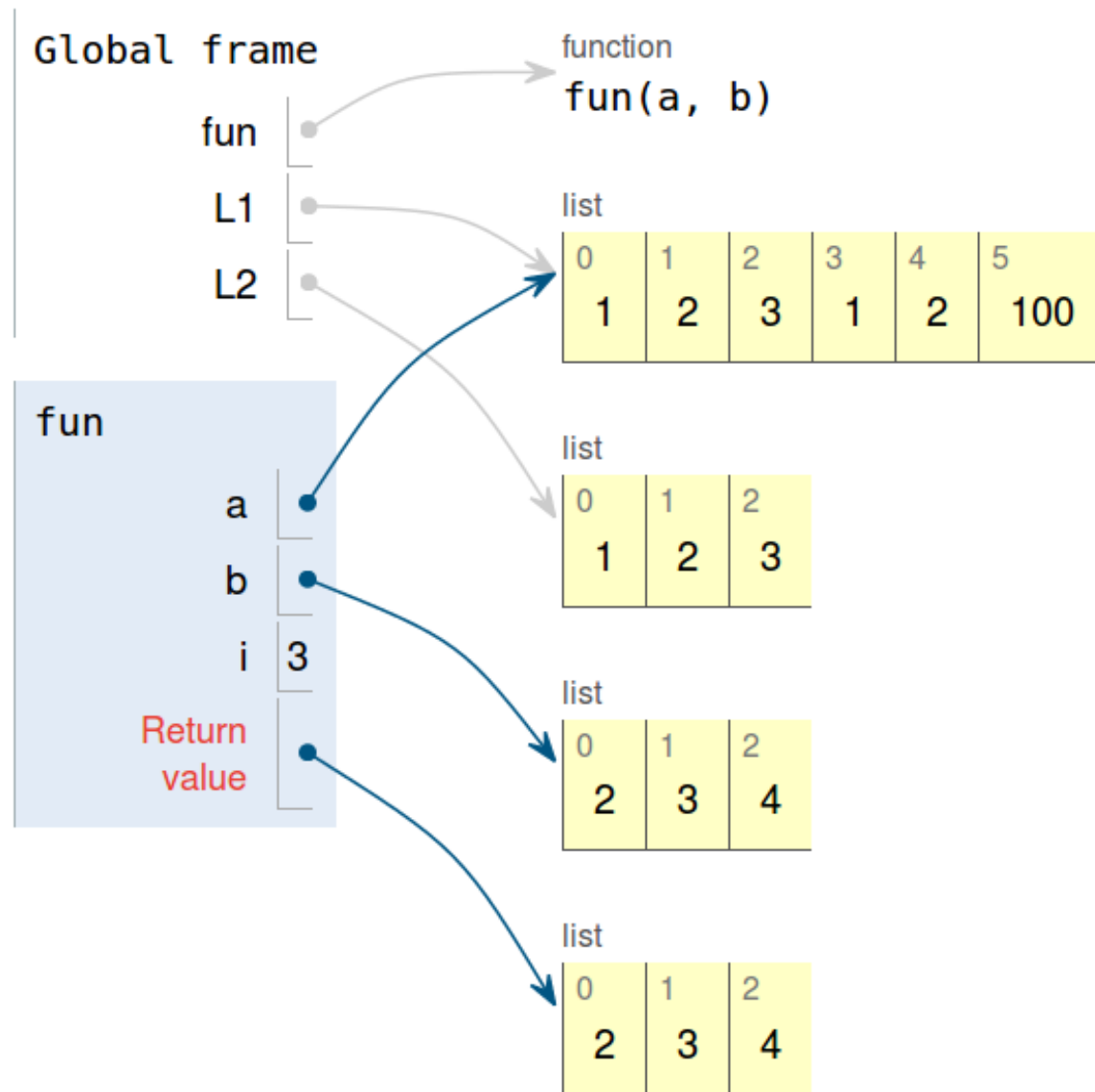
Line: 21 of 21 Col: 1 LINE INS

```
daalvarez@eredron:~ > python3 06_pasando_inmutables_como_parametro_a_funcion.py
antes de llamar función:  1 True Hola ('a', 'b', 'c')
ids =  0x9f8800 0x9de840 0x7fc77a96ef10 0x7fc77a96ab40
recibió 1 True Hola ('a', 'b', 'c')
ids =  0x9f8800 0x9de840 0x7fc77a96ef10 0x7fc77a96ab40
cambió a 11 False Holaxxx (10, 20, 30)
ids =  0x9f8940 0x9ba8e0 0x7fc77a975618 0x7fc77aa09c18
ids =  0x9f8800 0x9de840 0x7fc77a96ef10 0x7fc77a96ab40
```

```

1  def fun(a,b):
2      b = b[:]
3      for i in b:
4          a.append(i)
5
6      b += [4]
7      a[-1] = 100
8      del b[0]
9      return b[:]
10
11 L1 = [1,2,3]
12 L2 = [1,2,3]
13 L3 = fun(L1,L2)
14
15 print('L1 = ', L1)
16 print('L2 = ', L2)
17 print('L3 = ', L3)

```



Line: 19 of 19 Col: 1 LINE INS

daalvarez@eredron:~ > python3 06_ejemplo1.py

L1 = [1, 2, 3, 1, 2, 100]

L2 = [1, 2, 3]

L3 = [2, 3, 4]

La tabla de símbolos

http://en.wikipedia.org/wiki/Symbol_table

- Cuando una función llama a otra función, se crea una nueva tabla de símbolos para esa función. Las variables definidas dentro de la función son locales (se crean en la tabla de símbolos de la función).
- Las variables se buscan de la siguiente forma:
 - Tabla de símbolos local
 - Cuando se utilizan subfunciones, tabla de símbolos local de las funciones padre.
 - Tabla de símbolos global
 - Tabla de símbolos de los nombres de variable definidos en librerías (table of built-in names)

La pila de llamadas (call stack)

http://en.wikipedia.org/wiki/Call_stack

La pila (de llamadas) es una estructura dinámica de datos LIFO, que almacena la información sobre las funciones activas de un programa.

La principal razón de su uso, es seguir el curso del punto al cual cada función activa debe retornar el control cuando termine de ejecutar. Las funciones activas son las que se han llamado pero todavía no han completado su ejecución ni retornando al lugar siguiente desde donde han sido llamadas. Si, por ejemplo, `DibujaCuadrado()` llama a `DibujaLinea()` desde cuatro lugares diferentes, el código de `DibujaLinea()` debe tener una manera de saber a donde retornar. Esto es típicamente hecho por un código que, para cada llamada dentro de `DibujaCuadrado()`, pone la dirección de la instrucción después de la sentencia de llamada particular (la "dirección de retorno") en la pila de llamadas.

Si se consume todo el espacio asignado para la pila de llamadas, ocurre un error llamado **desbordamiento de pila** (stack overflow).

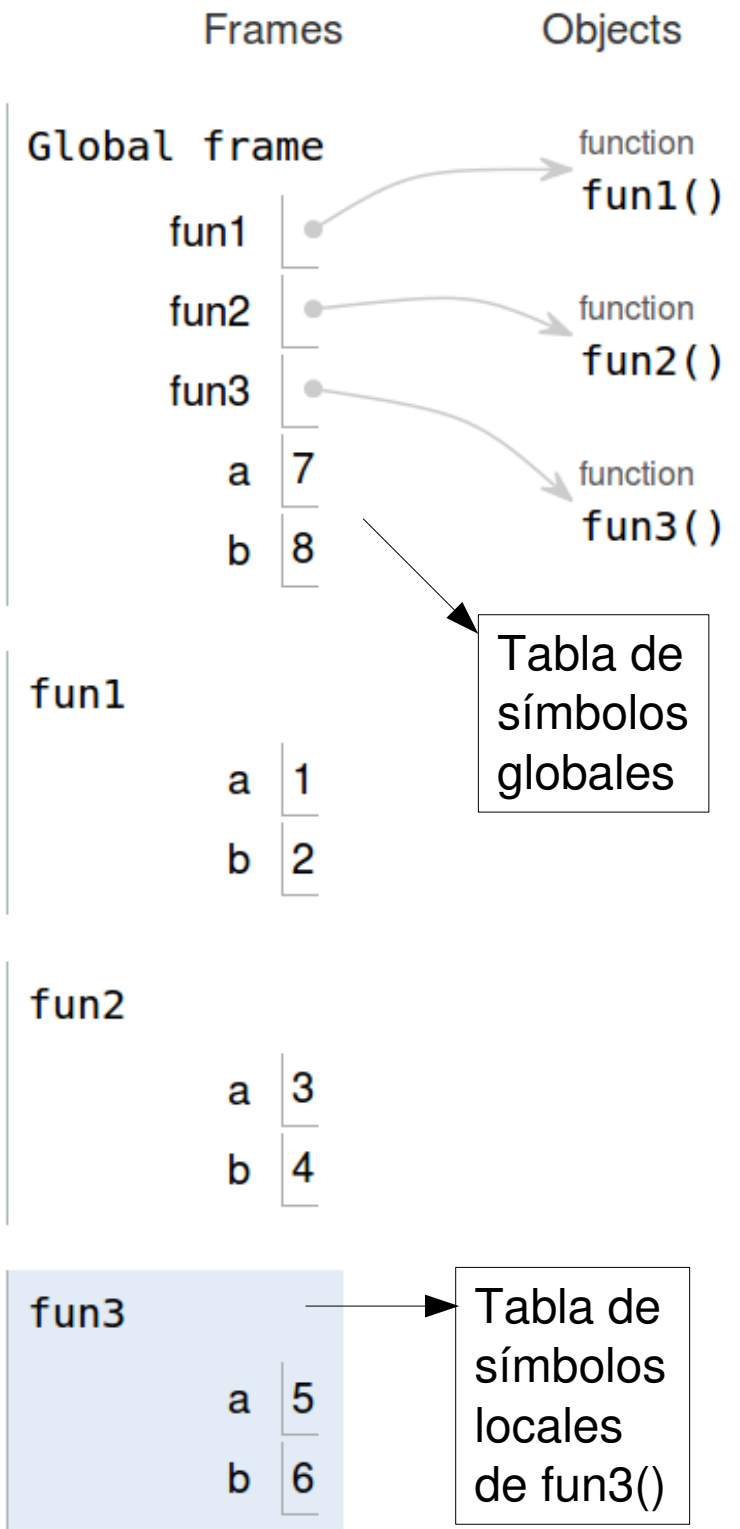
La pila de llamadas

```
1 def fun1():
2     a = 1; b = 2
3     print('fun1a(): a = {0}, b = {1}'.format(a,b))
4     fun2()
5     print('fun1b(): a = {0}, b = {1}'.format(a,b))
6
7 def fun2():
8     a = 3; b = 4
9     print('fun2a(): a = {0}, b = {1}'.format(a,b))
10    fun3()
11    print('fun2b(): a = {0}, b = {1}'.format(a,b))
12
13 def fun3():
14     a = 5; b = 6
15     print('fun3(): a = {0}, b = {1}'.format(a,b))
16
17 # PROCEDIMIENTO PRINCIPAL
18 a = 7; b = 8
19 print('fun0a(): a = {0}, b = {1}'.format(a,b))
20 fun1()
21 print('fun0b(): a = {0}, b = {1}'.format(a,b))
```

Line: 23 of 23 Col: 1 LINE INS

daalvarez@eredron:~ > python3 06_pila_de_llamadas.py

```
fun0a(): a = 7, b = 8
fun1a(): a = 1, b = 2
fun2a(): a = 3, b = 4
fun3(): a = 5, b = 6
fun2b(): a = 3, b = 4
fun1b(): a = 1, b = 2
fun0b(): a = 7, b = 8
```



Funciones recursivas

http://en.wikipedia.org/wiki/Recursion_%28computer_science%29

Son funciones que se llaman a si mismas.

Se debe poner mucho cuidado al escribir una función recursiva ya que esta no se debe llamar a si misma indefinidamente; debe haber un punto en el que la función en verdad retorne un valor. De lo contrario al alcanzar una profundidad de `sys.getrecursionlimit` Python lanzaría una excepción `RuntimeError` con el objeto de proteger la memoria RAM del computador.

La pila de llamadas de una función recursiva

Frames

Objects

Global frame
factorial

function
factorial(n)

factorial
n 4

factorial
n 3

factorial
n 2

factorial
n 1
Return value 1

```
1 def factorial(n):  
2     # Recuerde que math.factorial() existe  
3     if n<0:  
4         return None  
5     if n==0 or n==1:  
6         return 1  
7     else:  
8         return n*factorial(n-1)  
9  
10 print('4! =', factorial(4))
```

```
1 n = 0
2
3 ▼ def llamando_a_funcion():
4     global n
5     n += 1
6     print(n)
7     llamando_a_funcion()
8
9 llamando_a_funcion()
```

< Line: 11 of 11 Col: 1 LINE INS

```
daalvarez@eredron:~ > python3 06_stack_overflow_recursion.py >salida.txt 2>error.txt
daalvarez@eredron:~ > head -n 2 salida.txt; tail -n 2 salida.txt
```

```
1
2
997
998
```

```
daalvarez@eredron:~ > head -n 6 error.txt; tail -n 4 error.txt
```

Traceback (most recent call last):

```
File "06_stack_overflow_recursion.py", line 9, in <module>
    llamando_a_funcion()
File "06_stack_overflow_recursion.py", line 7, in llamando_a_funcion
    llamando_a_funcion()
File "06_stack_overflow_recursion.py", line 7, in llamando_a_funcion
    llamando_a_funcion()
File "06_stack_overflow_recursion.py", line 6, in llamando_a_funcion
    print(n)
```

RuntimeError: maximum recursion depth exceeded while getting the str of an object

```
daalvarez@eredron:~ > █
```

Esta función se está autollamando de forma recursiva. Para prevenir que se agote toda la memoria RAM del PC, una vez se alcanzó 999 iteraciones (ver `sys.getrecursionlimit()`), Python lanzó la excepción `RuntimeError`, lo cual produce el desbordamiento de la pila (**Stack Overflow**).

Se puede incrementar el número de llamadas recursivas con:

```
sys.setrecursionlimit(10000) # aumenta la profundidad de la pila de llamadas a 10000
```

Sin embargo, es mejor práctica de programación convertir el algoritmo a uno que no necesite recursiones. Desde este punto de vista, no todos los algoritmos se deberían programar de forma recursiva en Python. Por ejemplo un algoritmo para calcular los 10000 primeros números de la serie Fibonacci en Python que utilice recursiones tendría problema con este lenguaje, pero no con C/C++. Python limita el número de llamadas recursivas para prevenir un fallo en memoria en Python.

En conclusión, use `sys.setrecursionlimit()` con cuidado, porque un valor muy alto podría hacer que Python falle.

Se debe tener en cuenta que las funciones recursivas pueden agotar la memoria de pila. Observe este ejemplo en lenguaje C

```
1  #include <stdio.h>
2
3  // gcc -Wall -o 05_agotando_memoria_pila 05_agotando_memoria_pila.c
4
5  int n;
6
7  void llamando_a_funcion();
8
9  int main(void)
10 {
11     llamando_a_funcion();
12     return 0;
13 }
14
15 void llamando_a_funcion()
16 {
17     printf("%d\n", n++);
18     llamando_a_funcion();
19 }
20
21
```

Line: 21 Col: 1 Characters: 241 INS LINE UTF-8 05_agotando_memoria_pila.c

261748
261749
261750
261751
Segmentation fault
[daalvarez@localhost 05-Funciones]\$

```

1 ▼ def es_primo(num):
2 ▼     '''Función que detemina si num es primo o no.
3
4     es_primo(num)
5
6     Parametro de entrada:
7     num: número entero o lista de números enteros
8
9     Retorna:
10    Si num es un entero, retorna True o False dependiendo si num es primo
11    Si num es una lista de enteros, retorna un vector con True o False
12    dependiendo si num los elementos de num son primos o no.
13    '''
14
15 ▼    if isinstance(num, int):          # en caso que num sea un entero
16 ▼        if num == 2:
17            return True
18 ▼        elif (num < 2) or (num%2 == 0):
19            return False
20 ▼        else:
21 ▼            for i in range(3, num//2 + 1, 2):
22 ▼                if num%i == 0:
23                    return False
24                    return True
25 ▼    elif isinstance(num, list):      # en caso que num sea una lista
26        vec_es_primo = [es_primo(num[i]) for i in range(len(num))]
27        return vec_es_primo
28
29 print(es_primo(37))
30 print(es_primo([97,98,99,100,101,102,103]))

```

Line 32, Column 1

INSERT Soft Tabs: 4 UTF-8 Python

```

daalvarez@eredron ~ $ python3 06_es_primo.py
True
[True, False, False, False, True, False, True]
daalvarez@eredron ~ $

```

return

La palabra reservada **return** termina la ejecución de una función y retorna el control a la función que la llamó:

return valor_a_retornar

valor_a_retornar es opcional; en este caso, un **return** sin valor a retornar devolverá un **None**

Si se sale de la función por el final de la misma también retorna un **None**

Dentro de una función pueden existir varios **return**

return

```
>>>
def fib2(n):
    """Retorna la serie de Fibonacci hasta el número n."""
    resultado = []
    a, b = 0, 1
    while a < n:
        resultado.append(a)
        a, b = b, a+b
    return resultado
```

```
>>> fib2
<function fib2 at 0x7fc6af9417b8>
>>> fib2(200)
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144]
>>> r = fib2(200)
>>> r
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144]
>>> |
```

Como hacer una función que devuelva varios parámetros?

Respuesta: utilizando sequence unpacking

```
1 def fun1(x):
2     return x+1, x**2, 3*x
3
4 def fun2(x):
5     return [x+1, [1,2,3], 'Hola']
6
7 def fun3(x):
8     return (x+1, [1,2,3], 'Hola')
9
10 a,b,c = fun1(10);    print(a,b,c)
11 [a,b,c] = fun1(10);  print(a,b,c)
12 (a,b,c) = fun1(10);  print(a,b,c)
13 a,b,c = fun2(10);    print(a,b,c)
14 a,b,c = fun3(10);    print(a,b,c)
15
16 a,b = fun3(10) # se están recibiendo dos parámetros y se están entregando 3
17 print(a,b,c)
```

Line: 19 of 19 Col: 1 LINE INS

```
daalvarez@eredron:~ > python3 06_retornando_varios_parametros.py
```

```
11 100 30
```

```
11 100 30
```

```
11 100 30
```

```
11 [1, 2, 3] Hola
```

```
11 [1, 2, 3] Hola
```

```
Traceback (most recent call last):
```

```
File "06_retornando_varios_parametros.py", line 16, in <module>
```

```
    a,b = fun3(10) # se están recibiendo dos parámetros y se están entregando 3
```

```
ValueError: too many values to unpack (expected 2)
```

```
daalvarez@eredron:~ > █
```

Funciones que retornan varios parámetros

```
1 def minmax(a, b, c):
2     # Calcular el mínimo
3     if a < b:
4         if a < c:
5             min = a
6         else:
7             min = c
8     else:
9         if b < c:
10            min = b
11        else:
12            min = c
13
14    # Calcular el máximo
15    if a > b:
16        if a > c:
17            max = a
18        else:
19            max = c
20    else:
21        if b > c:
22            max = b
23        else:
24            max = c
25
26    return [min, max]
```

```
28 [mínimo, máximo] = minmax(10, 2, 5)
29 print('El_mínimo_es', mínimo)
30 print('El_máximo_es', máximo)
```

Argumentos opcionales y con nombre

```
1 def fun1(val1, val2, val3, calc_suma=True):
2     if calc_suma: # Calcule la suma
3         return val1 + val2 + val3
4     else:         # Calcule mejor el promedio
5         return (val1 + val2 + val3)/3
6
7 def fun2(val1=0, val2=0):
8     return val1 - val2
9
10 print(fun1(1, 2, 3))
11 print(fun1(1, 2, 3, False))
12 print(fun1(1, 2, 3, calc_suma=False))
13 print(' ..... ')
14 print(fun2(10, 3))
15 print(fun2(val1=10, val2=3))
16 print(fun2(val2=3, val1=10)) # aquí los parámetros están en diferente orden
17 print(fun2(val1=10))
18 print(fun2(val2=3))
```

Line: 21 of 23 Col: 3 LINE INS

daa@heimdall ~ \$ python3 06_par_opcionales.py

6
2.0
2.0

7
7
7
10
-3

daa@heimdall ~ \$

Python permite que los
argumentos de funciones tengan
nombres y valores por defecto.

Argumentos opcionales y con nombre

```
>>> fun2(10,3)
7
>>> fun2(val2=3)
-3
>>> fun2(val1=10, 3)
SyntaxError: non-keyword arg after keyword arg
>>> fun2(val2=3, 10)
SyntaxError: non-keyword arg after keyword arg
>>> |
```

Después de nombrar un argumento, todos los argumentos posteriores deben nombrarse.

```
def parrot(voltage, state='a stiff', action='vroom', type='Norwegian Blue'):
    print("-- This parrot wouldn't", action, end=' ')
    print("if you put", voltage, "volts through it.")
    print("-- Lovely plumage, the", type)
    print("-- It's", state, "!")
```

accepts one required argument (`voltage`) and three optional arguments (`state`, `action`, and `type`). This function can be called in any of the following ways:

```
parrot(1000)                                # 1 positional argument
parrot(voltage=1000)                        # 1 keyword argument
parrot(voltage=1000000, action='VOOOOOM')  # 2 keyword arguments
parrot(action='VOOOOOM', voltage=1000000)  # 2 keyword arguments
parrot('a million', 'bereft of life', 'jump') # 3 positional arguments
parrot('a thousand', state='pushing up the daisies') # 1 positional, 1 keyword
```

but all the following calls would be invalid:

```
parrot()                                # required argument missing
parrot(voltage=5.0, 'dead')             # non-keyword argument after a keyword argument
parrot(110, voltage=220)                 # duplicate value for the same argument
parrot(actor='John Cleese')             # unknown keyword argument
```

In a function call, keyword arguments must follow positional arguments. All the keyword arguments passed must match one of the arguments accepted by the function (e.g. `actor` is not a valid argument for the `parrot` function), and their order is not important. This also includes non-optional arguments (e.g. `parrot(voltage=1000)` is valid too). No argument may receive a value more than once.

```

1 i = 5
2
3 ▼ def fun1(arg=i):
4     print(arg)
5
6 ▼ def fun2(val, arr=[]):
7     arr.append(val)
8     return arr
9
10 # Solución al problema con fun2()
11 ▼ def sol_fun2(val, arr=None):
12 ▼     if arr is None:
13         arr = []
14         arr.append(val)
15
16     return arr
17
18 ▼ def fun3(a, b=5):
19     b = b+a
20     return b
21
22 i = 6; fun1()
23 print(fun2(1)); print(fun2(2)); print(fun2(3))
24 print(sol_fun2(1)); print(sol_fun2(2)); print(sol_fun2(3))
25 print(fun3(1), fun3(2), fun3(3))

```

Argumentos opcionales y con nombre

Tenga en cuenta que los valores por defecto se evalúan al momento de definir la función. Si no se tiene cuidado, esto puede producir problemas potenciales.

```

Line: 27 of 27 Col: 1  LINE  INS  06_parametros_por_defecto
daa@heimdall ~ $ python3 06_parametros_por_defecto.py

```

```

5
[1]
[1, 2]
[1, 2, 3]
[1]
[2]
[3]
6 7 8

```

Funciones variádicas

http://en.wikipedia.org/wiki/Variadic_function

Son funciones que toman un número variable de argumentos.

Funciones variádicas

```
1 def mi_fun(aa1, aa2, *bbb, **ccc):
2     print('El argumento aa1 es', aa1)
3     print('El argumento aa2 es', aa2)
4     print('-'*70)
5     print('El argumento bbb es', bbb)
6     for i,b in enumerate(bbb):
7         print('bbb[', i, '] =', b)
8     print('-'*70)
9     print('El argumento ccc es', ccc)
10    keys = sorted(ccc.keys())
11    for kw in keys:
12        print('ccc[', kw, '] =', ccc[kw])
13
14    mi_fun('Arg 1', 'Arg 2', 'Arg 3', 'Arg 4', a2 = 'Arg 5', a3 = 123, a1 = [1,2,3])
```

- El argumento * va antes que el argumento **
- El argumento * recibe una tupla
- El argumento ** recibe un diccionario
- El orden con los que se guardan los argumentos en ** es indefinido

Line: 16 of 16 Col: 1 LINE INS

daalvarez@eredron:~ > python3 04_parametros_variables_a_una_funcion.py

El argumento aa1 es Arg 1

El argumento aa2 es Arg 2

El argumento bbb es ('Arg 3', 'Arg 4')

bbb[0] = Arg 3

bbb[1] = Arg 4

El argumento ccc es {'a3': 123, 'a1': [1, 2, 3], 'a2': 'Arg 5'}

ccc[a1] = [1, 2, 3]

ccc[a2] = Arg 5

ccc[a3] = 123

daalvarez@eredron:~ > □

4.7.3. Arbitrary Argument Lists

Finally, the least frequently used option is to specify that a function can be called with an arbitrary number of arguments. These arguments will be wrapped up in a tuple (see [Tuples and Sequences](#)). Before the variable number of arguments, zero or more normal arguments may occur.

```
def write_multiple_items(file, separator, *args):  
    file.write(separator.join(args))
```

Normally, these `variadic` arguments will be last in the list of formal parameters, because they scoop up all remaining input arguments that are passed to the function. Any formal parameters which occur after the `*args` parameter are 'keyword-only' arguments, meaning that they can only be used as keywords rather than positional arguments.

```
>>> def concat(*args, sep="/"):  
...     return sep.join(args)  
...  
>>> concat("earth", "mars", "venus")  
'earth/mars/venus'  
>>> concat("earth", "mars", "venus", sep=".")  
'earth.mars.venus'
```

4.7.4. Unpacking Argument Lists

The reverse situation occurs when the arguments are already in a list or tuple but need to be unpacked for a function call requiring separate positional arguments. For instance, the built-in `range()` function expects separate *start* and *stop* arguments. If they are not available separately, write the function call with the `*`-operator to unpack the arguments out of a list or tuple:

```
>>> list(range(3, 6))           # normal call with separate arguments
[3, 4, 5]
>>> args = [3, 6]
>>> list(range(*args))         # call with arguments unpacked from a list
[3, 4, 5]
```

In the same fashion, dictionaries can deliver keyword arguments with the `**`-operator:

```
>>> def parrot(voltage, state='a stiff', action='vroom'):
...     print("-- This parrot wouldn't", action, end=' ')
...     print("if you put", voltage, "volts through it.", end=' ')
...     print("E's", state, "!")
...
>>> d = {"voltage": "four million", "state": "bleedin' demised", "action": "VOOM"}
>>> parrot(**d)
-- This parrot wouldn't VOOM if you put four million volts through it. E's bleedin' demised !
```

Documentando una función: los docstring

Los docstring

(document string)
documentan lo que hace la función. Si se define debe ponerse inmediatamente de la línea con **def**. La importancia de esta es que se vuelve un atributo de la función (entendida como un objeto). Los docstrings se utilizan para generar la documentación impresa del programa.

```
1  ▼ sufijos = { 1000: ['KB', 'MB', 'GB', 'TB'],
2                1024: ['KiB', 'MiB', 'GiB', 'TiB']}
3
4  ▼ def tamaño_aproximado(tamaño, exacto=True):
5  ▼     ''' Convierte un número de bytes a otras unidades.
6
7         tamaño_aproximado(tamaño, exacto=True)
8
9         Parámetros de entrada:
10             tamaño: tamaño dado en bytes
11             exacto: True  = división entre 1024
12                   False = división entre 1000
13
14     Retorna una cadena de texto
15     '''
16
17  ▼     if tamaño < 0:
18         raise ValueError('"tamaño" debe ser positivo')
19
20     multiple = 1024 if exacto else 1000
21  ▼     for s in sufijos[multiple]:
22         tamaño /= multiple
23  ▼         if tamaño < multiple:
24             return '{0:.1f} {1}'.format(tamaño, s)
25
26     raise ValueError('"tamaño" es demasiado grande')
27
28  ▼ if __name__ == '__main__':
29     print(tamaño_aproximado(1000000000, False))
30     print(tamaño_aproximado(1000000000))
```

Line 31, Column 1

INSERT Soft Tab

daalvarez@eredron ~ \$ python3 prog_11_diccionarios.py

1.0 GB

953.7 MiB

Documentando una función

Convenciones de la documentación:

- La primera línea del docstring debe describir el propósito de la función. Comience esta línea con una mayúscula y termínela con un punto.
- La segunda línea déjese en blanco
- Después de esto describa como tal la función: uso, parámetros de entrada y de salida, efectos secundarios, etc. Describa qué hace la función, no cómo lo hace.

```
In [1]: %run prog_11_diccionarios.py
1.0 GB
953.7 MiB
```

```
In [2]: from prog_11_diccionarios import tamaño_aproximado
```

```
In [3]: help(tamaño_aproximado)
Help on function tamaño_aproximado in module prog_11_diccionarios:
```

```
tamaño_aproximado(tamaño, exacto=True)
    Convierte un número de bytes a otras unidades.
```

```
tamaño_aproximado(tamaño, exacto=True)
```

Parámetros de entrada:

tamaño: tamaño dado en bytes
exacto: True = división entre 1024
False = división entre 1000

Retorna una cadena de texto

```
In [4]: print(tamaño_aproximado.__doc__)
Convierte un número de bytes a otras unidades.
```

```
tamaño_aproximado(tamaño, exacto=True)
```

Parámetros de entrada:

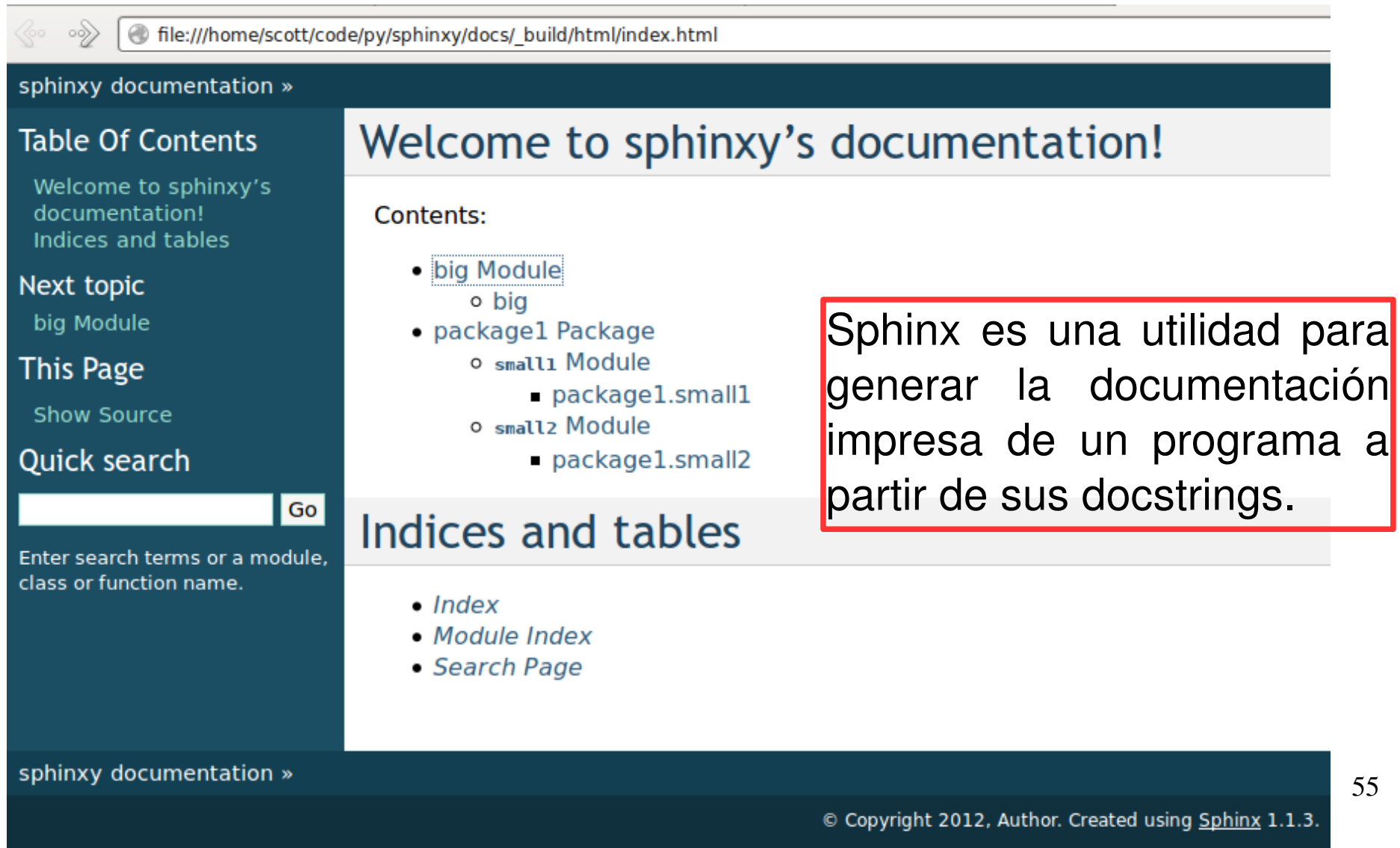
tamaño: tamaño dado en bytes
exacto: True = división entre 1024
False = división entre 1000

Retorna una cadena de texto

Imprimiendo el docstring

Sphinx

<http://sphinx-doc.org/>



The screenshot shows a web browser window displaying the Sphinx documentation. The address bar shows the file path: `file:///home/scott/code/py/sphinx/docs/_build/html/index.html`. The page has a dark blue header and footer. The main content area is white. On the left, there is a dark blue sidebar with white text. The sidebar contains links to the Table of Contents, Next topic, This Page, and Quick search. The main content area has a large heading 'Welcome to sphinx's documentation!' and a 'Contents:' section with a list of links. A red box highlights a text block on the right side of the page.

sphinx documentation »

Table Of Contents

Welcome to sphinx's documentation!
Indices and tables

Next topic

[big Module](#)

This Page

[Show Source](#)

Quick search

Enter search terms or a module, class or function name.

Contents:

- [big Module](#)
 - [big](#)
- [package1 Package](#)
 - [small1 Module](#)
 - [package1.small1](#)
 - [small2 Module](#)
 - [package1.small2](#)

Indices and tables

- [Index](#)
- [Module Index](#)
- [Search Page](#)

Sphinx es una utilidad para generar la documentación impresa de un programa a partir de sus docstrings.

sphinx documentation »

© Copyright 2012, Author. Created using [Sphinx 1.1.3](#).

La ruta de búsqueda de archivos (the search path)

```
>>> import sys
>>> sys.path
['', '/home/daa', '/usr/bin', '/usr/lib/python3.4', '/usr/lib/python
3.4/plat-i386-linux-gnu', '/usr/lib/python3.4/lib-dynload', '/usr/lo
cal/lib/python3.4/dist-packages', '/usr/lib/python3/dist-packages']
>>> sys
<module 'sys' (built-in)>
>>> sys.path.insert(0, '/home/daa/Cursos_UNAL/Programacion_de_PCs/py
thon')
>>> sys.path
['/home/daa/Cursos_UNAL/Programacion_de_PCs/python', '', '/home/daa'
, '/usr/bin', '/usr/lib/python3.4', '/usr/lib/python3.4/plat-i386-li
nux-gnu', '/usr/lib/python3.4/lib-dynload', '/usr/local/lib/python3.
4/dist-packages', '/usr/lib/python3/dist-packages']
>>> |
```

Inserta al principio de la lista

Nota: los módulos “built-in” no tienen un archivo .py asociado, ya que fueron escritos en lenguaje C.

El efecto de agregar un archivo a la ruta desaparece al cerrar Python.⁵⁷

Mostrar todas las variables en memoria

- `dir()` will give you the list of in scope variables:
- `globals()` will give you a dictionary of global variables
- `locals()` will give you a dictionary of local variables

Funciones anidadas

http://en.wikipedia.org/wiki/Nested_function

```
1  def fun1(x):  
2      z = 1  
3  
4  def fun2(y):  
5      return x+y+z  
6  
7  def fun3(y):  
8      return x*y + z  
9  
10     return fun2(3) + fun3(4) + z  
11  
12 print(fun1(10))
```

Una función anidada (nested) es una función encapsulada dentro de otra función. Solo puede ser llamada por la función que contenedora o por las funciones anidadas en el mismo nivel. Se utilizan para ocultar procedimientos que sólo son útiles localmente. Se deben definir junto con la declaración de variables al principio de la función.

Line: 14 of 14 Col: 1 LINE INS 06_funciones_anidadas.py

daa@heimdall ~ \$ python3 06_funciones_anidadas.py

Funciones anidadas y la palabra reservada **nonlocal**

```
1 def outer():
2     # Se declaran las variables
3     x = 1
4
5     # Se declaran las subfunciones
6     def inner():
7         # nonlocal x
8         x = 2
9         print("inner:", x)
10
11     # código de outer()
12     inner()
13     print("outer:", x)
14
15 outer()
```

Line: 7 of 17 Col: 1 LINE INS

```
daa@heimdall ~ $ python3 06_nonlocal.py
inner: 2
outer: 1
daa@heimdall ~ $
```

```
1 def outer():
2     # Se declaran las variables
3     x = 1
4
5     # Se declaran las subfunciones
6     def inner():
7         nonlocal x
8         x = 2
9         print("inner:", x)
10
11     # código de outer()
12     inner()
13     print("outer:", x)
14
15 outer()
```

Line: 7 of 17 Col: 1 LINE INS

```
daa@heimdall ~ $ python3 06_nonlocal.py
inner: 2
outer: 2
daa@heimdall ~ $
```

nonlocal permite asignar valores a variables que están en un ámbito exterior, pero que no es el ámbito global. La variable referida no es ni local ni global. http://en.wikipedia.org/wiki/Non-local_variable

Funciones anidadas y la palabra reservada **nonlocal**

```
1 def outer():
2     # Se declaran las variables
3     x = 1
4
5     # Se declaran las subfunciones
6     def inner():
7         # nonlocal x
8         x += 2
9         print("inner:", x)
10
11     # código de outer()
12     inner()
13     print("outer:", x)
14
15 outer()
```

Line: 7 of 17 Col: 1 LINE INS 06_nonlocal.py UTF

```
daa@heimdall ~ $ python3 06_nonlocal.py
Traceback (most recent call last):
  File "06_nonlocal.py", line 15, in <module>
    outer()
  File "06_nonlocal.py", line 12, in outer
    inner()
  File "06_nonlocal.py", line 8, in inner
    x += 2
UnboundLocalError: local variable 'x' referenced
before assignment
daa@heimdall ~ $
```

```
1 def outer():
2     # Se declaran las variables
3     x = 1
4
5     # Se declaran las subfunciones
6     def inner():
7         nonlocal x
8         x += 2
9         print("inner:", x)
10
11     # código de outer()
12     inner()
13     print("outer:", x)
14
15 outer()
```

Line: 7 of 17 Col: 1 LINE INS 06_non

```
daa@heimdall ~ $ python3 06_nonlocal.py
inner: 3
outer: 3
```

Funciones anónimas

Las funciones anónimas son funciones que no necesitan tener un nombre. Se crean con la palabra reservada `lambda`; `lambda` retorna una función; dicha función puede asignársele un nombre (en dicho caso funcionará igual a una creada con `def`) o simplemente se le puede pasar como argumento a otra función.

Las funciones creadas con `lambda` solo contienen una expresión, la cual retorna su resultado inmediatamente. Se utiliza para mejorar la claridad del programa o para evitar llenar el código con pequeñas funciones `def` de una sola línea.

Si usted necesita funciones con varios comandos o funciones complicadas, es preferible que las defina con `def`.

Aquí `f()` y `g()` hacen lo mismo y se pueden utilizar exactamente de la misma forma.

Observe que `lambda` no requiere un `return`.

La expresión con `lambda` se puede utilizar en esos lugares donde se espera una expresión. No hay necesidad de asignarla a una variable.

```
>>> def f(x):  
        return 2*x
```

```
>>> f(5)  
10
```

```
>>> g = lambda x: 2*x  
>>> g(5)
```

```
10
```


```
>>> (lambda x: 2*x)(5)  
10
```

```
>>> def hacer_incremento(n):  
        return lambda x: x + n
```

```
>>> inc2 = hacer_incremento(2)  
>>> inc6 = hacer_incremento(6)  
>>> print(inc2(42), inc6(42))  
44 48
```

```
>>> print(hacer_incremento(22)(33))  
55
```

`n` está en el
ámbito
actual



```
>>> suma = lambda x,y: x+y  
>>> suma(2,4)  
6
```

```
>>>  
>>> suma = lambda x,y=10: x+y  
>>> suma(2,4)  
6  
>>> suma(2)  
12
```

```
>>> L = [lambda x:x**2, lambda x:x**3, lambda x:x**4]
>>> for f in L: print(f(3))
```

```
9
27
81
>>> print(L[0](11))
```

```
121
>>>
>>>
>>> def f1(x): return x**2
```

```
>>> def f2(x): return x**3
```

```
>>> def f3(x): return x**4
```

```
>>> L = [f1, f2, f3]
>>> for f in L: print(f(3))
```

```
9
27
81
>>> print(L[0](11))
```

```
121
>>>
>>> min = (lambda x, y: x if x < y else y)
>>> min(2,3)
```

```
2
>>> |
```

Un buen tutorial de funciones anónimas es:

https://pythonconquerstheuniverse.wordpress.com/2011/08/29/lambda_tutorial/

map() y filter()

map(*function*, *iterable*, ...)

Return an iterator that applies *function* to every item of *iterable*, yielding the results. If additional *iterable* arguments are passed, *function* must take that many arguments and is applied to the items from all iterables in parallel. With multiple iterables, the iterator stops when the shortest iterable is exhausted. For cases where the function inputs are already arranged into argument tuples, see [itertools.starmap\(\)](#).

filter(*function*, *iterable*)

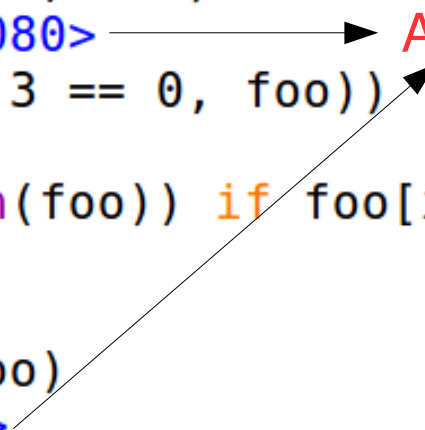
Construct an iterator from those elements of *iterable* for which *function* returns true. *iterable* may be either a sequence, a container which supports iteration, or an iterator. If *function* is `None`, the identity function is assumed, that is, all elements of *iterable* that are false are removed.

Note that `filter(function, iterable)` is equivalent to the generator expression `(item for item in iterable if function(item))` if function is not `None` and `(item for item in iterable if item)` if function is `None`.

filter(), map()

```
>>> foo = [2, 18, 9, 22, 17, 24, 8, 12, 27]
>>>
>>> filter(lambda x: x % 3 == 0, foo)
<filter object at 0x7f37f93b5080>
>>> list(filter(lambda x: x % 3 == 0, foo))
[18, 9, 24, 12, 27]
>>> [foo[i] for i in range(len(foo)) if foo[i]%3 == 0]
[18, 9, 24, 12, 27]
>>>
>>> map(lambda x: 2*x + 10, foo)
<map object at 0x7f37f93b5160>
>>> list(map(lambda x: 2*x + 10, foo))
[14, 46, 28, 54, 44, 58, 26, 34, 64]
>>> [(2*foo[i] + 10) for i in range(len(foo))]
[14, 46, 28, 54, 44, 58, 26, 34, 64]
```

Ambos son iteradores



De todas formas, es preferible utilizar list comprehensions que filter() y map().

```
1 def cuadrado(x):
2     return x**2
3
4 def cubo(x):
5     return x**3
6
7 def integral_definida(f, a, b, n):
8     if n == 0:
9         sumatorio = 0.0
10    else:
11        deltax = (b-a) / n
12        sumatorio = 0.0
13        for i in range(n):
14            sumatorio += deltax * f(a + i * deltax)
15    return sumatorio
16
17 a = 1
18 b = 2
19 print('Integración entre {0} y {1}'.format(a, b))
20 print('Integral de x**2:', integral_definida(cuadrado, a, b, 100))
21 print('Integral de x**3:', integral_definida(cubo, a, b, 100))
```

Pasando funciones como argumentos a funciones

Cuando se pasa la función como parámetro, no se usan los paréntesis con argumentos; solo se pasa el nombre de la función.



```

1 def suma(a,b):          return a+b
2 def resta(a,b):         return a-b
3 def multiplicacion(a,b): return a*b
4 def division(a,b):      return a/b
5
6 ▼ def aritmetica_con_if_elif(a,b,opcion):
7     if opcion == '+': return suma(a,b)
8     elif opcion == '-': return resta(a,b)
9     elif opcion == '*': return multiplicacion(a,b)
10    elif opcion == '/': return division(a,b)
11
12 ▼ def aritmetica_con_funciones_como_parametros(a,b,fun):
13     return fun(a,b)
14
15 print('La suma de 2 y 5 es', aritmetica_con_if_elif(2,5,'+'))
16 print('La suma de 2 y 5 es', aritmetica_con_funciones_como_parametros(2,5,suma))

```

Line: 18 of 18 Col: 1 LINE INS

```

daa@heimdall ~ $ python3 06_funciones_como_par_a_funciones.py
La suma de 2 y 5 es 7
La suma de 2 y 5 es 7
daa@heimdall ~ $ █

```

Decoradores

Un decorador es una función que toma otra función y extiende el comportamiento de dicha función sin modificarla explícitamente.

```
1 def mi_decorador(func):
2     def envolver():
3         print(">>> ", end="")
4         func()
5         print(" <<<")
6     return envolver
7
8 @mi_decorador
9 def diga_hola1():
10     print("Hola 1", end="")
11
12 def diga_hola2():
13     print("Hola 2", end="")
14
15 diga_hola2 = mi_decorador(diga_hola2)
16
17 diga_hola1()
18 diga_hola2()
```

Line: 20 of 20 Col: 1 LINE INS

```
daalvarez@heimdall:~ > python 06_decoradores.py
>>> Hola 1 <<<
>>> Hola 2 <<<
daalvarez@heimdall:~ > □
```

Decorando funciones que aceptan parámetros

```
1 def hacer_dos_veces(func):
2     def envolver_hacer_dos_veces():
3         func()
4         func()
5     return envolver_hacer_dos_veces
6
7 @hacer_dos_veces
8 def saludar(nombre):
9     print(f"Hola {nombre}")
10
11 saludar("Pepito")
```

Line: 13 of 13 Col: 1 LINE INS

daalvarez@heimdall:~ > python 06_decoradores_02.py

Traceback (most recent call last):

File "06_decoradores_02.py", line 11, in <module>

saludar("Pepito")

TypeError: envolver_hacer_dos_veces() takes 0 positional arguments but 1 was given

```
1 def hacer_dos_veces(func):
2     def envolver_hacer_dos_veces(*args, **kwargs):
3         func(*args, **kwargs)
4         func(*args, **kwargs)
5     return envolver_hacer_dos_veces
6
7 @hacer_dos_veces
8 def saludar(nombre):
9     print(f"Hola {nombre}")
10
11 saludar("Pepito")
```

Ahora el decorador trabaja con funciones que aceptan parámetros.

Line: 13 of 13 Col: 1 LINE INS

daalvarez@heimdall:~ > python 06_decoradores_03.py

Hola Pepito

Hola Pepito

Decorando funciones que retornan valores

```
1 def hacer_dos_veces(func):
2     def envolver_hacer_dos_veces(*args, **kwargs):
3         func(*args, **kwargs)
4         func(*args, **kwargs)
5     return envolver_hacer_dos_veces
6
7 @hacer_dos_veces
8 def saludar(nombre):
9     print(f"Hola {nombre}")
10
11 saludo = saludar("Pepito")
12 print(saludo)
```

Line: 14 of 15 Col: 1 LINE INS

daalvarez@heimdall:~ > python 06_decoradores_04.py

Hola Pepito
Hola Pepito
None

```
1 def hacer_dos_veces(func):
2     def envolver_hacer_dos_veces(*args, **kwargs):
3         func(*args, **kwargs)
4         return func(*args, **kwargs)
5     return envolver_hacer_dos_veces
6
7 @hacer_dos_veces
8 def saludar(nombre):
9     print("Creando un saludo ...")
10    return f"Hola {nombre}"
11
12 saludo = saludar("Pepito")
13 print(saludo)
```

Line: 16 of 16 Col: 1 LINE INS

daalvarez@heimdall:~ > python 06_decoradores_05.py
Creando un saludo ...
Creando un saludo ...
Hola Pepito

Decoradores

Hay mucho más sobre decoradores en:

- <https://realpython.com/primer-on-python-decorators/>
- Decoradores y clases
- Decoradores con parámetros
- Decoradores anidados
- Clases como decoradores
- etc.

Dynamically typed language

Python es un lenguaje de **tipeado dinámico** (dynamically typed language). Esto quiere decir que los tipos de datos se asocian a los valores almacenados en las variables, no a la variable misma.

Esto implica que las variables pueden tomar cualquier tipo de valor en cualquier punto del programa; sin embargo el tipo de dato únicamente se verifica cuando se realiza una acción con los datos.

Lenguajes como Java y C/C++ son lenguajes de tipeado estático (statically typed languages). La ventaja del tipeado estático es que este ayuda a prevenir errores y permite que el compilador genere código más veloz y eficiente en términos del uso de la memoria.

Type annotations

(nuevo con Python 3.5)

- Python no es estricto con respecto a los type annotations. Simplemente son ayudas para incrementar la legibilidad del programa, ya que el intérprete de Python ignora tales órdenes.
- Ventajas:
 - Hacen que el código sea más fácil de mantener, revisar y encontrar errores.
 - Previenen errores inesperados porque se pasó a la función datos de un tipo para la cual no estaba diseñada.

Type annotations + mypy

```
1 def funcion(a: str, b: str, times: int) -> str:
2     return (a + b) * times
3
4 print(funcion("¡Buenos ", "días! ", 3))
5
6 # Observe que lastimosamente el intérprete de Python
7 # no le hace caso a las type annotations.
8 print(funcion(1, 2, 3))
9
10 edad: int = 20
11 print(edad) # 20
12 edad = 'Veinte'
13 print(edad)
```

Observe que podemos llamar a nuestro código de forma incorrecta, pero, un programador podría ver que se está usando la función de una forma diferente a la que diseñó.

```
Line 15, Column 1
INSERT Soft Tabs: 4 UTF-8 Python
daalvarez@eredron ~ $ python 06_type_annotations_1.py
¡Buenos días! ¡Buenos días! ¡Buenos días!
9
20
Veinte
daalvarez@eredron ~ $ mypy 06_type_annotations_1.py
06_type_annotations_1.py:8: error: Argument 1 to "funcion" has incompatible type "int"; expected "str"
06_type_annotations_1.py:8: error: Argument 2 to "funcion" has incompatible type "int"; expected "str"
06_type_annotations_1.py:12: error: Incompatible types in assignment (expression has type "str", variable has type "int")
```

Instale **mypy** con el comando de consola: `python3 -m pip install mypy`

mypy is an experimental optional static type checker for Python that aims to combine the benefits of dynamic typing and static typing. **mypy** combines the expressive power and convenience of Python with a powerful type system and compile-time type checking.

Documentación: <https://mypy.readthedocs.io/>

```
1 from typing import List, Dict, Tuple, Optional, Union
2
3 # names es una lista de cadenas
4 def print_names(names: List[str]) -> None:
5     for student in names:
6         print(student)
7
8 # grades es un diccionario con keys=cadenas y values=flotantes
9 def print_name_and_grade(grades: Dict[str, float]) -> None:
10     for student, grade in grades.items():
11         print(student, grade)
12
13 # Cree su propio tipo de dato:
14 # Point es un tipo de dato que contiene dos ints (x,y)
15 Point = Tuple[int, int]
16
17 # points es una lista de puntos
18 def print_points(points: List[Point]):
19     for point in points:
20         print("X:", point[0], " Y:", point[1])
21
22 def get_api_response() -> Tuple[int, int]:
23     successes, errors = ... # Some API call
24     return successes, errors
25
26 # some_num o es un número entero o es un None
27 def try_to_print(some_num: Optional[int]):
28     if some_num:
29         print(some_num)
30     else:
31         print('Value was None!')
32
33 # grade o es un entero o es una cadena
34 def print_grade(grade: Union[int, str]):
35     if isinstance(grade, str):
36         print(grade + ' percent')
37     else:
38         print(str(grade) + '%')
```

Type annotations: el módulo typing

<https://docs.python.org/3/library/typing.html>

Referencias

- Wikipedia
- <http://www.inventwithpython.com/>
- <http://www.diveintopython3.net/>
- Documentación de Python:
 - <https://docs.python.org/3/tutorial/index.html>
 - <https://docs.python.org/3/>
- Marzal Varó, Andrés; Gracia Luengo, Isabel; García Sevilla, Pedro (2014). Introducción a la programación con Python 3. Disponible en: <http://dx.doi.org/10.6035/Sapientia93>
- <https://realpython.com/primer-on-python-decorators/>