Introducción a las ciencias de la computación
y programación en Python

Funciones: descomposición y abstracción

Rodrigo Chang

Banco de Guatemala

Rodrigo Chang <rrcp@banguat.gob.gt>

Abstract

"Programs must be written for people to read, and only incidentally for machines to execute."
— Harold Abelson, Structure and Interpretation of Computer Programs

Veremos cómo estructurar programas en funciones para ocultar detalles.
Hablaremos de los conceptos de alcance (*scope*) y espacio de nombres (*name space*) en Python.

¿Cómo escribimos código?

- Hasta ahora:
  - Hemos visto algunos mecanismos del lenguaje.
  - Código en un bloque (celda) representa algún cómputo.
  - Cada archivo (celda) es una pieza de código.
    - Cada código es una secuencia de instrucciones.
- Algunos problemas:
  - Es fácil para problemas a *pequeña escala*.
  - Es **muy desordenado** para problemas más grandes.
  - Es difícil **seguir** los detalles.
  - Información correcta ⇒ parte código del código.

Buenas prácticas

- Más código **no** es necesariamente algo bueno.
- Cantidad de *funcionalidad* ⇒ buenos programadores.
- Necesitamos un mecanismo para **descomponer** el código y **abstraer** la funcionalidad.
- Para esto, introducimos las **funciones**.

Abstracción

- Un proyector es una caja negra — no sabemos cómo funciona.
- Sabemos cómo se utiliza y para qué sirve.
- Conocemos la entrada: cualquier dispositivo electrónico que se pueda conectar.
- Y la salida que produce.
- Es una caja negra que convierte y magnifica imágenes de la entrada hacia una pantalla (pared).

Esta es la idea detrás de la abstracción: no necesitamos saber cómo funciona para utilizarlo.

Descomposición

- Supongamos que queremos proyectar imágenes para las Olimpiadas con varios proyectores.
- Cada proyector recibe una entrada y produce una salida individual.
- Todos producen una imagen más grande.

Esta es la idea detrás de la descomposición: diferentes dispositivos trabajan en conjunto para lograr un objetivo.

Crear estructura con la descomposición

- En programación, el código se divide en **módulos**:
  - autocontenidos
  - permiten **separar** el código,
  - **reutilizarlo**,
  - mantenerlo **organizado** y
  - **coherente**.

Ahora utilizaremos funciones para descomponer el código, más adelante utilizaremos clases.

Omitir detalles con la abstracción

- En programación, puede pensarse en una **caja negra**:
  - No se quieren/necesitan ver los detalles.
  - Esconder detalles tediosos en el código.
- Esto lo lograremos con la **especificación adecuada** de una función:
  - ¿Qué recibe como entradas y qué devuelve como salidas?.
- También será útil definir un **docstring**.

Ahora utilizaremos funciones para descomponer el código, más adelante utilizaremos clases.

Funciones

- **Piezas** de código reutilizables.
- No se ejecutan hasta ser **llamadas** o **“invocadas”**.
- Características:
  - Nombre
  - Parámetros (0 o más)
  - docstring
  - cuerpo
  - devuelven un resultado

Ejemplo

- Veamos una función sencilla

```
# Def. de funcion par
def esPar(i):
    """
    Input: i, entero positivo
    Devuelve True si i es par, si no, False
    """
    remainder = i % 2
    return remainder == 0
```

Alcance (*scope*) de las variables

- Una función crea un nuevo ambiente/marco/*scope* **cuando se ejecuta**.
- **scope** se refiere al mapeo de nombres a objetos.
- El parámetro formal recibe el valor del parámetro real.

```
def f(x):
    x = x + 1
    print('dentro de f(x): x =', x)
    return x

x = 3
z = f( x )
```

- Ver en Python Tutor

¿Qué pasa si `z = f( x = x )`?

Funciones sin `return`

- También llamadas procedimientos.
- En Python, devuelven `None` (ausencia de valor).

```
# Def. de funcion par
def esPar(i):
    """
    Input: i, entero positivo
    Imprime True si i es par, si no, False
    """
    remainder = i % 2
    print(remainder == 0)
```

`return` **VS.** `print`

`return`:

- Significado dentro de una función.
- Se ejecuta **solo uno**.
- Después de `return`, código es ignorado.
- Tiene valor asociado, devuelto a quién invoca la función.

- Puede ser utilizado fuera de una función.
- Pueden ejecutarse **varios**.
- Después de `print`, código es ejecutado.
- Devuelve `None`, pero su salida es hacia la **consola**.

Funciones como argumentos

- arguments can take on any type, even functions

```
def func_a():
    print 'inside func_a'
def func_b(y):
    print 'inside func_b'
    return y
def func_c(z):
    print 'inside func_c'
    return z()
print func_a()
print 5 + func_b(2)
print func_c(func_a)
```

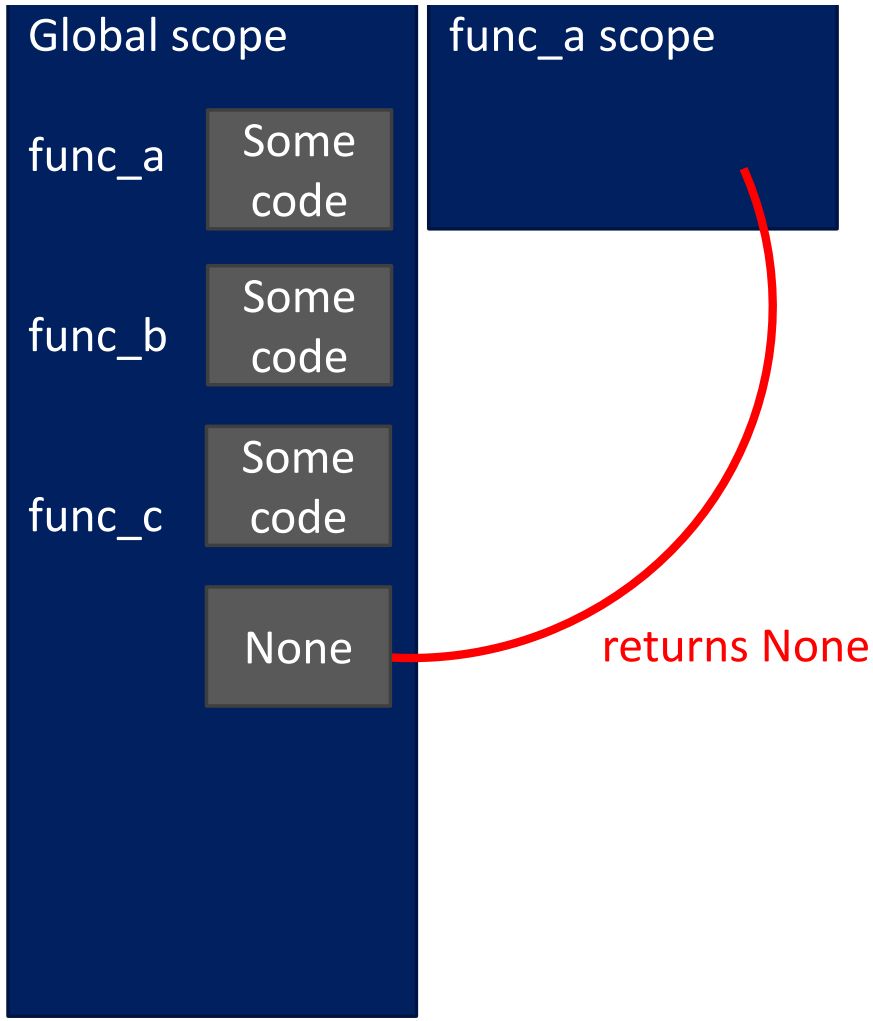
call `func_a`, takes no parameters

call `func_b`, takes one parameter

call `func_c`, takes one parameter, another function

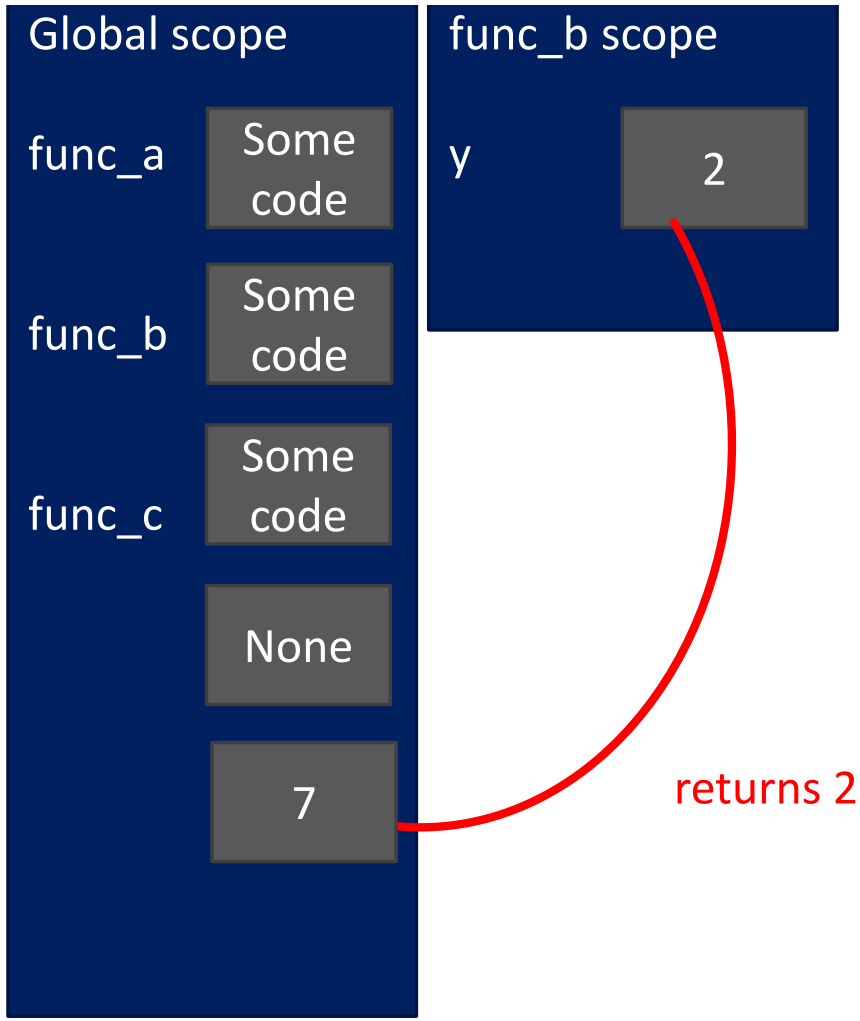
Funciones como argumentos

```
def func_a():
    print 'inside func_a'
def func_b(y):
    print 'inside func_b'
    return y
def func_c(z):
    print 'inside func_c'
    return z()
print func_a()
print 5 + func_b(2)
print func_c(func_a)
```



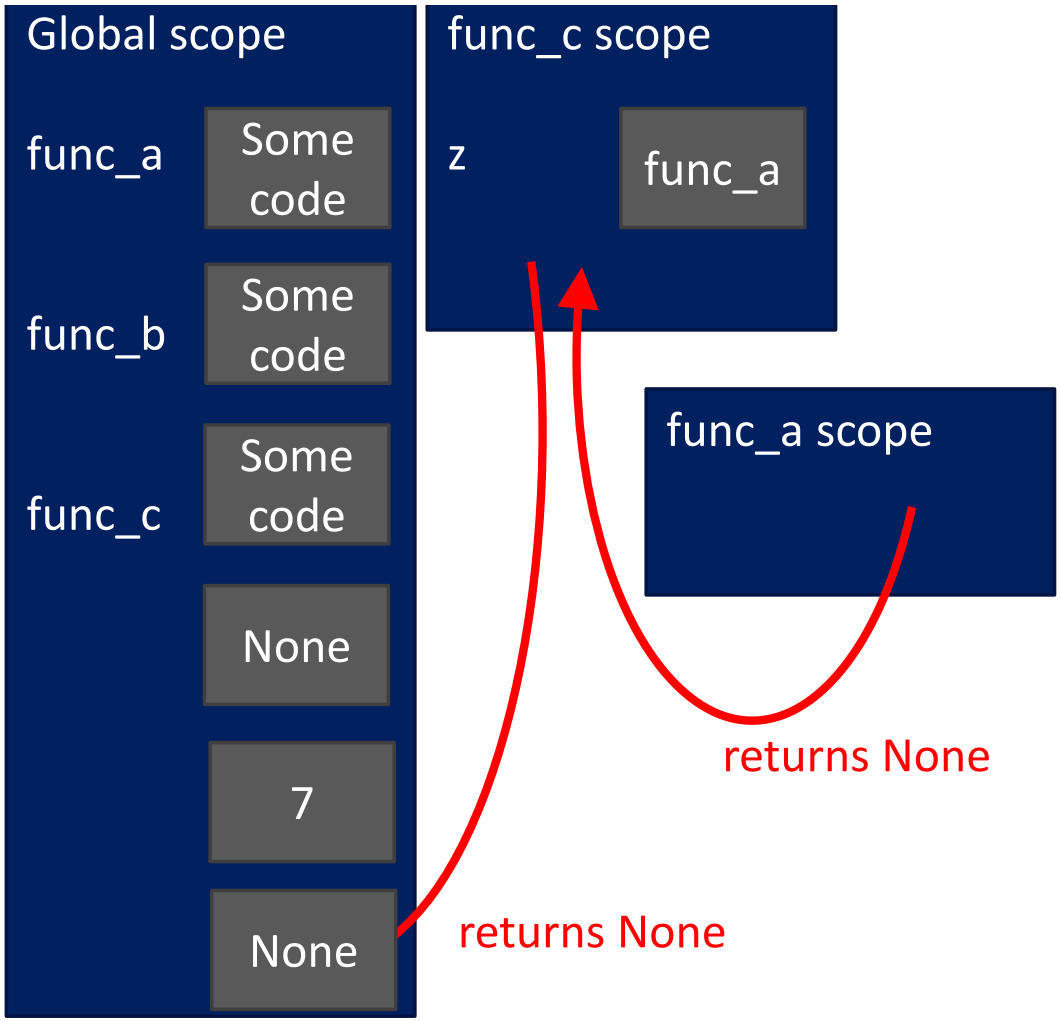
Funciones como argumentos

```
def func_a():
    print 'inside func_a'
def func_b(y):
    print 'inside func_b'
    return y
def func_c(z):
    print 'inside func_c'
    return z()
print func_a()
print 5 + func_b(2)
print func_c(func_a)
```



Funciones como argumentos

```
def func_a():
    print 'inside func_a'
def func_b(y):
    print 'inside func_b'
    return y
def func_c(z):
    print 'inside func_c'
    return z()
print func_a()
print 5 + func_b(2)
print func_c(func_a)
```



Scope (ambiente/alcance)

- Dentro de una función, **se puede acceder** a una variable definida afuera.

```
def g(y):
    print(x)
    print(x+y)

x = 5
g(x)
print(x)
```

- Otro ejemplo:

```
def g(y):
    print(x)

x = 5
g(x)
print(x)
```

Scope (ambiente/alcance)

- Dentro de una función, **no se puede modificar una variable[1] definida afuera.**

```
def h(y):
    x += 1

x = 5
h(x)
print(x)
```

- ¿Cómo funcionan los argumentos en Python?

[1] Excepto si la variable es mutable

Detalles en el *scope*

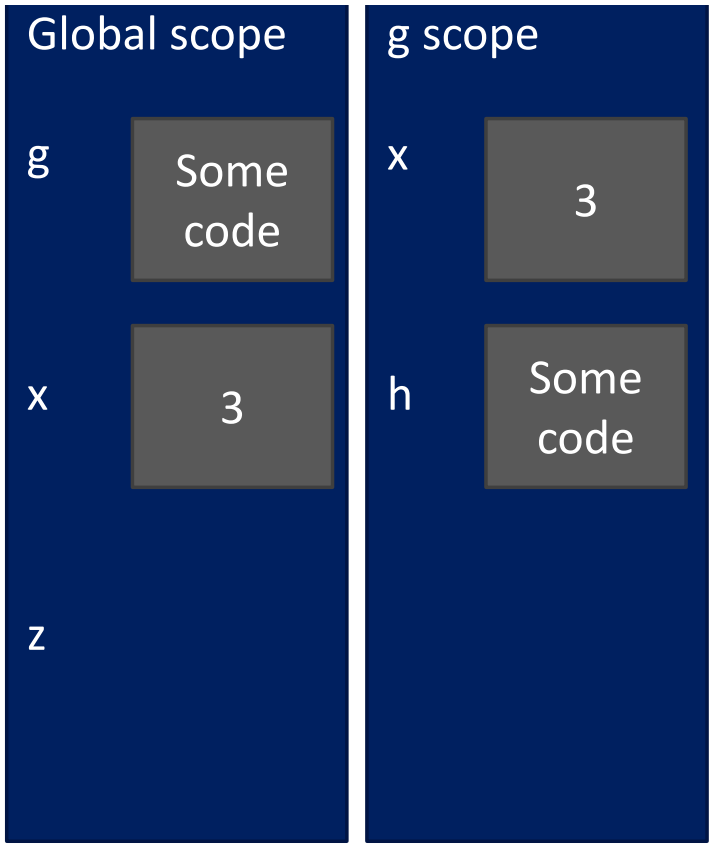
- Veamos un ejemplo más detallado:

```
def g(x):
    def h():
        x = 'abc'
    x = x + 1
    print('in g(x): x =', x)
    h()
    return x

x = 3
z = g(x)
```

- Ver en Python Tutor

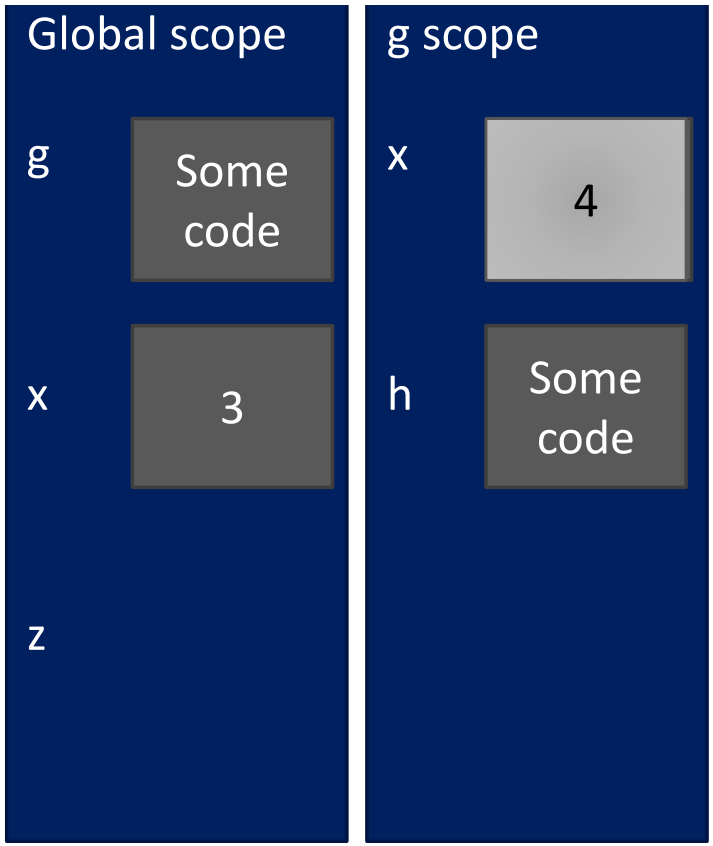
Detalles en el *scope*

```
def g(x):
    def h():
        x = 'abc'
    x = x + 1
    print('g: x =', x)
    h()
    return x

x = 3
z = g(x)
```



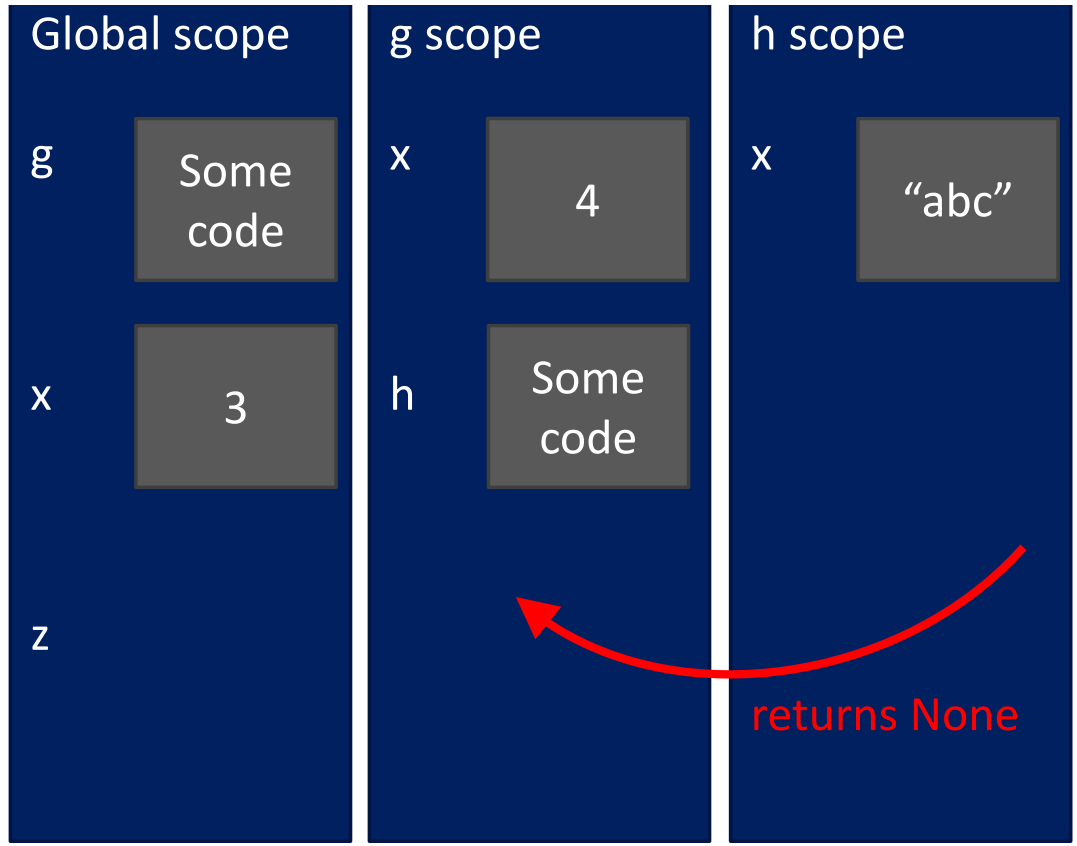
Detalles en el *scope*

```
def g(x):
    def h():
        x = 'abc'
    x = x + 1
    print('g: x =', x)
    h()
    return x

x = 3
z = g(x)
```



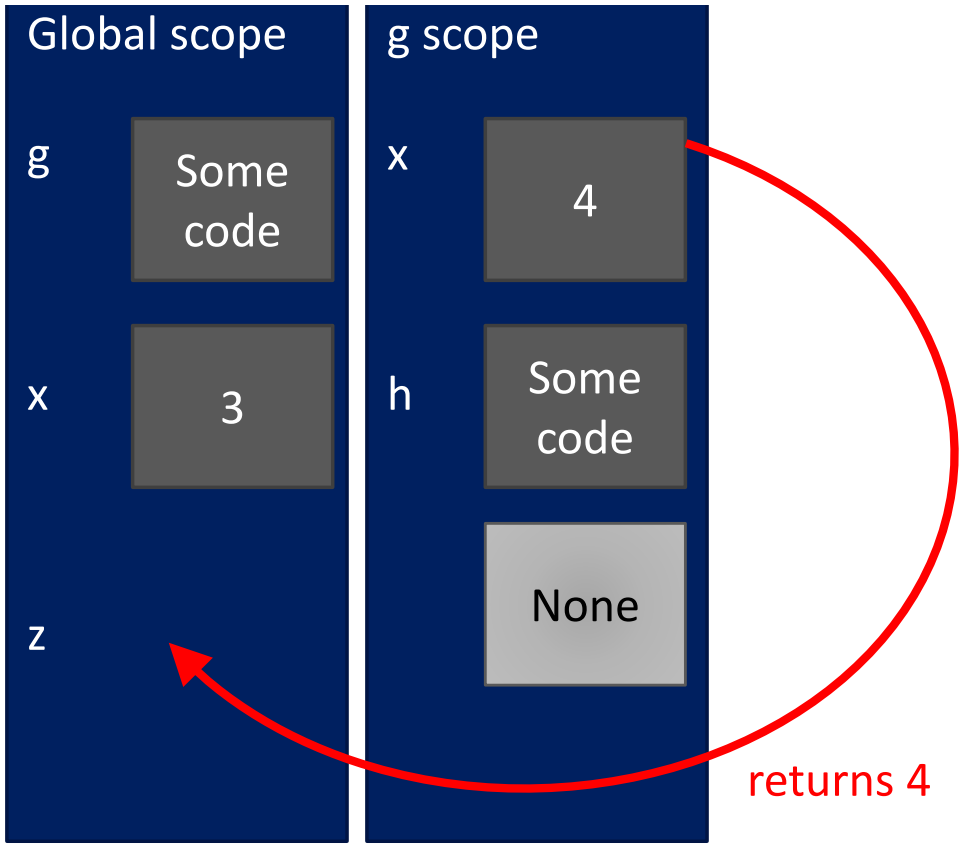
Detalles en el *scope*

```
def g(x):
    def h():
        x = 'abc'
    x = x + 1
    print('g: x =', x)
    h()
    return x

x = 3
z = g(x)
```



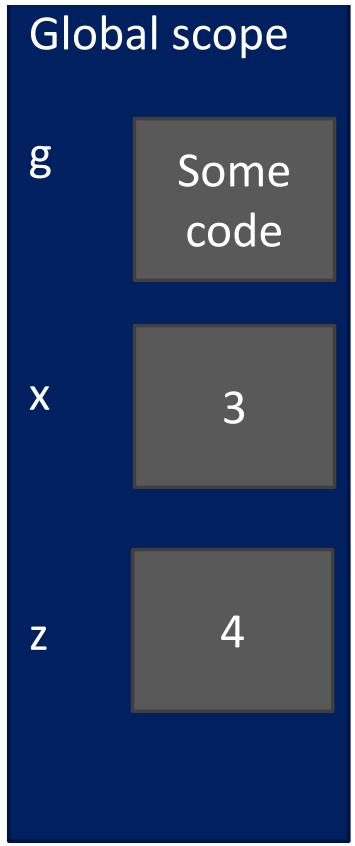
Detalles en el *scope*

```
def g(x):
    def h():
        x = 'abc'
    x = x + 1
    print('g: x =', x)
    h()
    return x

x = 3
z = g(x)
```



Detalles en el *scope*

```
def g(x):
    def h():
        x = 'abc'
    x = x + 1
    print('g: x =', x)
    h()
    return x

x = 3
z = g(x)
```



Descomposición y abstracción

- Concepto **poderoso**.
- El código es **REUTILIZABLE** y solamente tiene que ser **depurado** una vez.