Week 4 - Functions

CS115

INTRODUCTION TO PROGRAMMING IN PYTHON

WEEK 4

EXAMPLE –PROJECTOR

A projector is a black box

don’t know how it works

know the interface: input/output

connect any electronic to it that can communicate with that input

black box somehow converts image from input source to a wall, magnifying it

ABSTRACTION IDEA: do not need to know how projector works to use it

EXAMPLE –PROJECTOR

Projecting a large image for Olympics is decomposed into separate tasks for separate projectors

Each projector takes input and produces separate output

All projectors work together to produce larger image

DECOMPOSITION IDEA: different devices work together to achieve an end goal

CREATE STRUCTURE with DECOMPOSITION

In the projector example, separate devices allow for decomposition.

In programming, divide code into modules which are:

- self-contained
- used to break up code
- intended to be reusable
- keep code organized
- keep code coherent

Now we will explore how to achieve decomposition with functions.

Later, we will achieve decomposition with classes.

Functions

Sequence of instructions with a name

Reusable pieces/chunks of code

Examples: round(), randint(), len()

Functions are not run in a program until they are “called” or “invoked”

Function characteristics:

- has a name
- has parameters(0 or more)
- has a docstring (optional but recommended)
- has a body
- returns something

`return` Statement

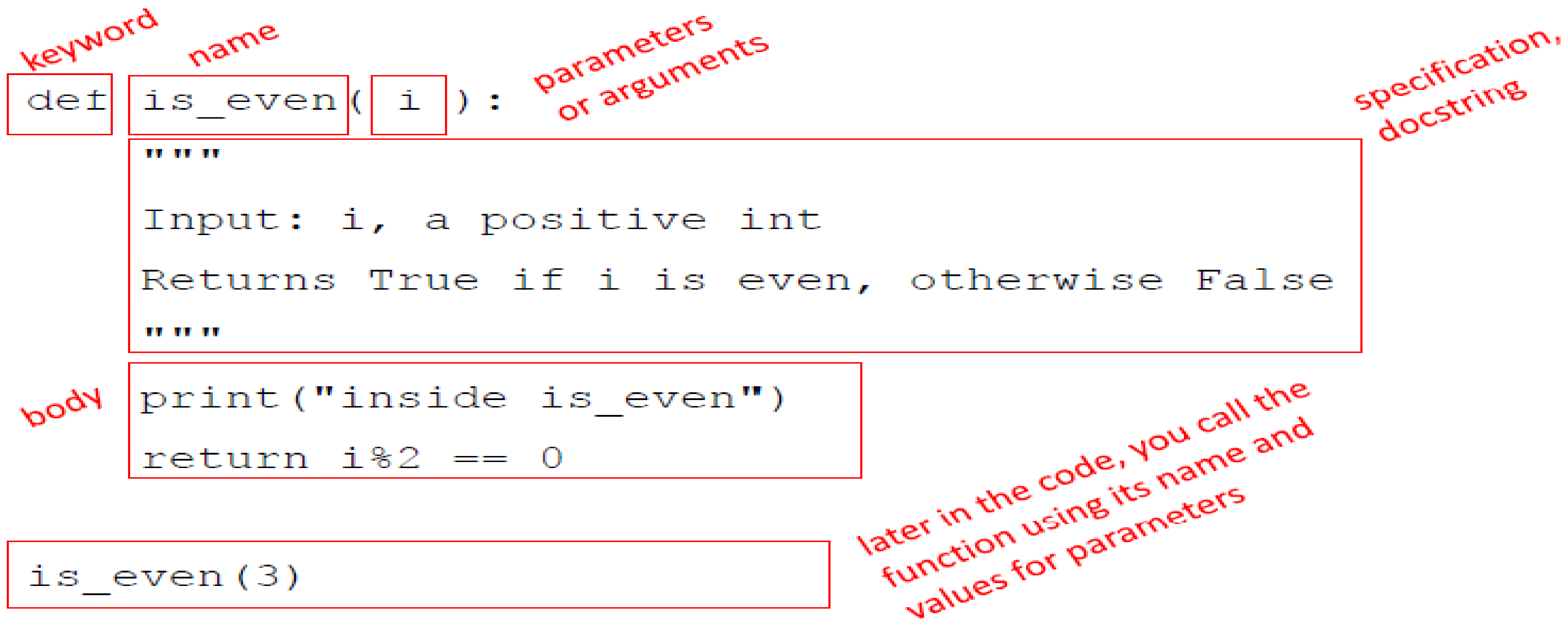
`return` only has meaning inside a function

Only one return executed inside a function

Any code inside function, but after return statement not executed

Has a value associated with it, returned to function caller

Writing and Invoking Functions



```
def is_even( i ):
    """
    Input: i, a positive int
    Returns True if i is even, otherwise False
    """
    print("inside is_even")
    return i%2 == 0

is_even(3)
```

keyword
name
parameters or arguments
specification, docstring
body
later in the code, you call the function using its name and values for parameters

Function Body

```
def is_even( i ):
    """
    Input: i, a positive int
    Returns True if i is even, otherwise False
    """
    print("inside is_even")
    return i%2 == 0
```

run some commands

keyword

expression to evaluate and return

Variable Scope

The scope of a variable is the part of the program from which the variable can be accessed.

Local variables are variables that are defined within a function.

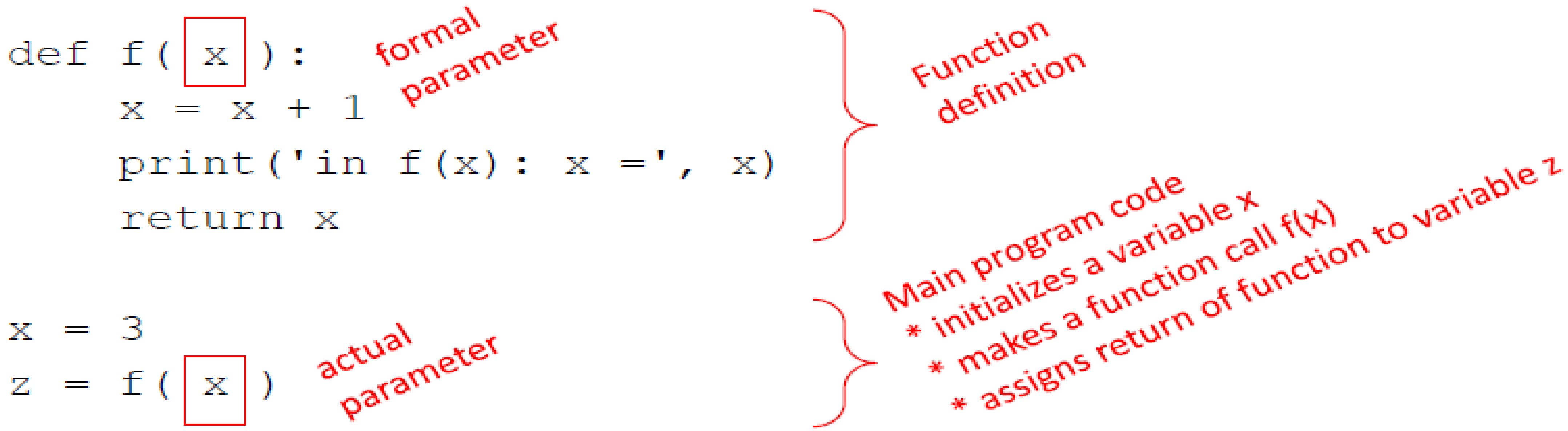
Local variables become available from the point that they are defined in a block until the end of the function in which it is defined.

Variables that are defined outside of functions have a global scope. This means that as we will see their values can be accessed from inside all functions, however they cannot be updated.

Variable Scope and Functions

The **formal parameters** will be assigned the values passed in the function call, the **actual parameters**.

New scope/frame/environment is created when a function is entered.

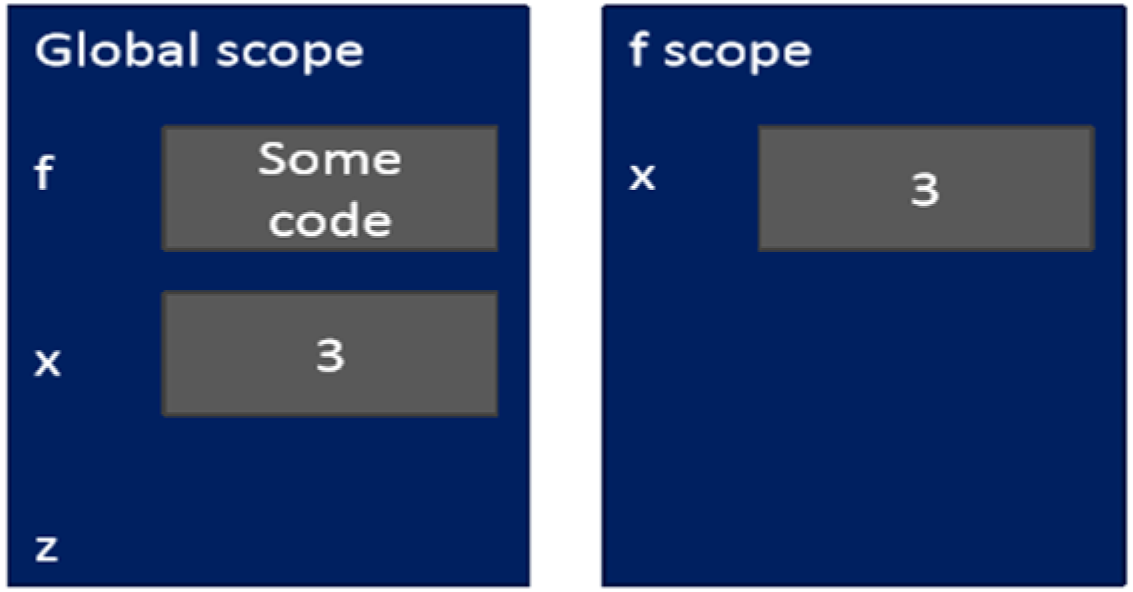


```
def f( x ):
    x = x + 1
    print('in f(x): x =', x)
    return x

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

formal parameter
Function definition
Main program code
* initializes a variable x
* makes a function call f(x)
* assigns return of function to variable z
actual parameter

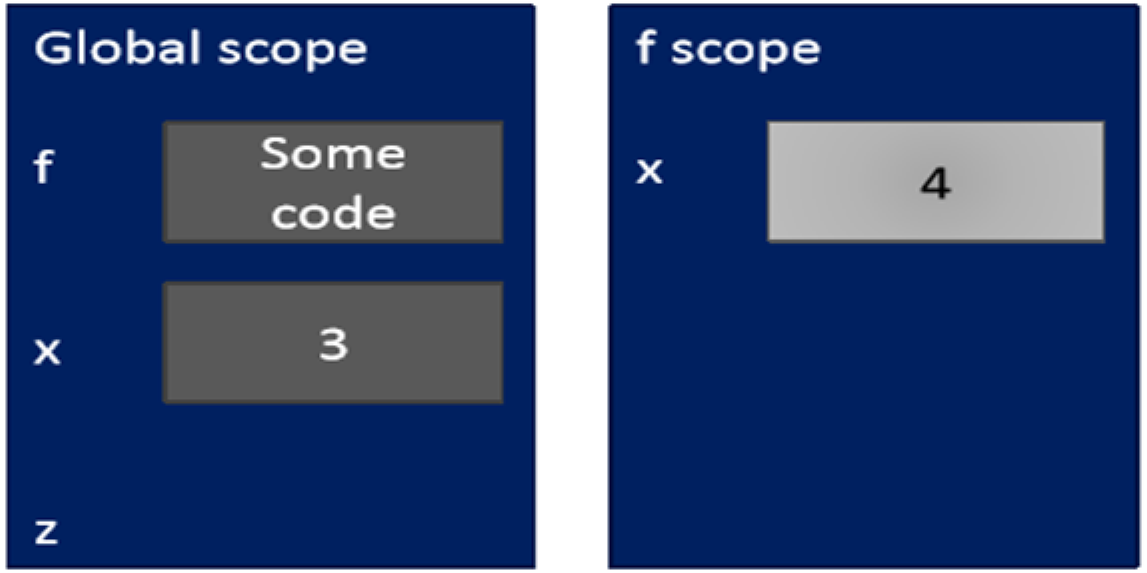
VARIABLE SCOPE

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

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



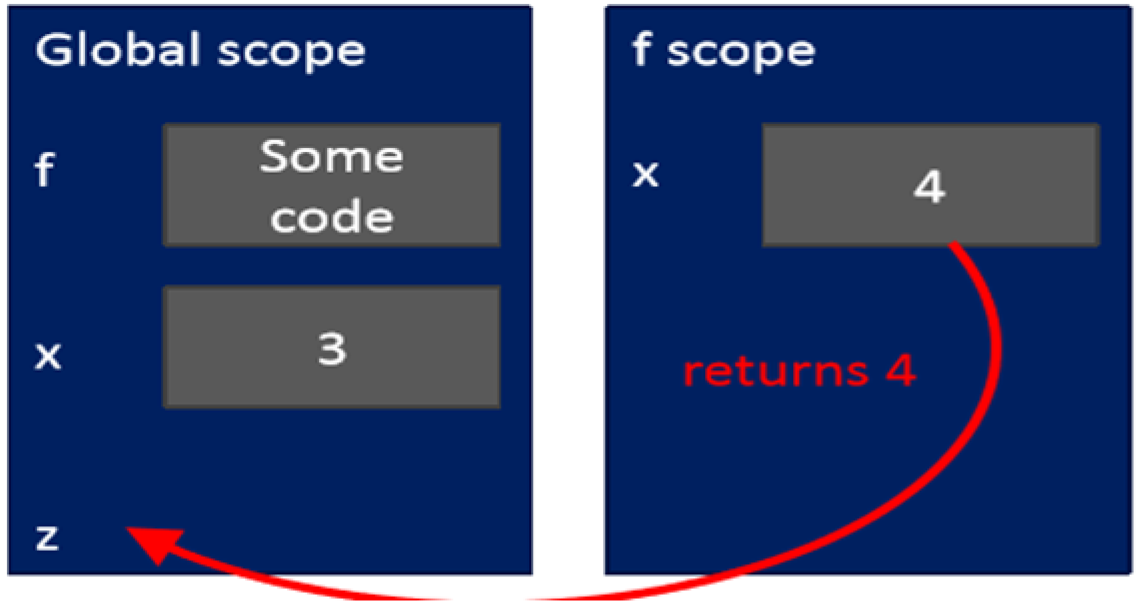
VARIABLE SCOPE

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

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



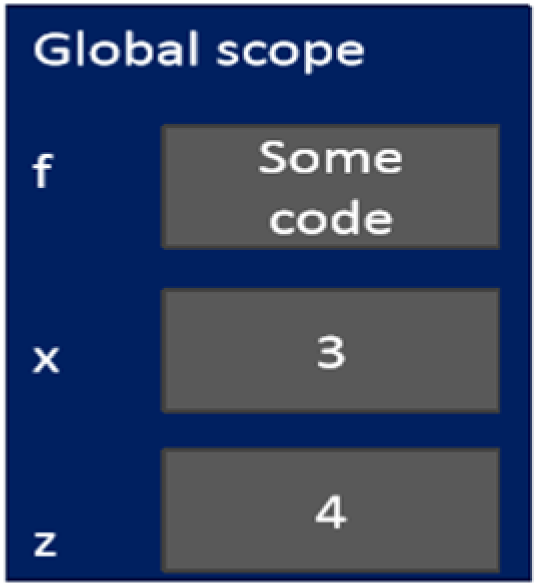
VARIABLE SCOPE

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

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



VARIABLE SCOPE

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

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



One Warning: If NO return STATEMENT

Python returns the value None, if no return given

None is of type NoneValue and represents the absence of a value.

```
def is_even_one(val):
    if val % 2 == 0:
        print('number is even')
    else:
        print('number is odd')

def is_even_two(val):
    if val % 2 == 0:
        return True

r1 = is_even_one(4)
print('Value: ', r1, 'Type: ', type(r1))
r2 = is_even_two(4)
print('Value: ', r2, 'Type: ', type(r2))

r3 = is_even_two(5)
print('Value: ', r3, 'Type: ', type(r3))
```

Output:

```
number is even
Value:  None Type:  <class 'NoneType'>
Value:  True Type:  <class 'bool'>
Value:  None Type:  <class 'NoneType'>
```

Exercises:

1. Write a function that returns the sum of the digits of a positive integer. Use this function to find the sum of digits of the positive numbers input by the user.
   - See: `04_exercise1.py`

2. Write a function that takes a number as a parameter and returns the reverse of the number.
   - See: `04_exercise2.py`

3. Write a function that converts a given string **s** of a binary number in base 2 to its decimal equivalent and returns the decimal value.

   - First write a function, named `is_binary`, that that takes a string parameter and returns True if the parameter string is a binary string with 0s and 1s (e.g., '101'), False otherwise (e.g., '123').

   - Then write a function, named convert_to_decimal, that takes a string parameter and uses your `is_binary` function to check if the parameter string is a binary number. If it is a binary, then your function converts it to its equivalent decimal. Else it displays an appropriate message to indicate that it is not a valid string for a binary number.

     See: `04_exercise3.py`

Summary – when a function is called:

1) Actual parameters are evaluated, formal parameters are bound to the values of the actual parameters.
2) Point of execution (control) moves from the calling statement to the first statement inside the function.
3) Code in the (indented) body of the function is executed until either:
   a) A return statements is encountered, and the value following the return statement is returned.
   b) There are no more statements within the body of the function, and `None` is returned.
4) The point of execution is transferred back to the code following the invocation.

Keyword Arguments

Up to now we have used ‘positional’ arguments, where the first formal parameter is bound to the first actual parameter, the second to the second, etc.

We can also use keyword arguments, where formal parameters are bound to actual parameters using the name of the formal parameter.

Keyword Arguments

```
def printName(firstName, lastName, reverse):
    if reverse:
        print(lastName + ',' + firstName)
    else:
        print(firstName, lastName)

printName(‘Joe', ‘Smith', False)
printName(‘Joe', ‘Smith', reverse = False)
printName(‘Joe', lastName = ‘Smith', reverse = False)
printName(lastName = ‘Smith', firstName = ‘Joe', reverse = False)
```

The statements above are equivalent, all call printName with the same actual parameter values.

Keyword arguments can appear in any order, however a non-keyword argument cannot follow a keyword argument, so the following is not allowed.

```
    printName(‘Joe’, lastName=‘Smith’, False)
```

Default Values

Keyword arguments can be used together with default parameter values.

```
def printName(firstName, lastName, reverse = False):
    if reverse:
        print(lastName + ',' + firstName)
    else:
        print(firstName, lastName)
```

The default values allow programmers to call a function with fewer than the specified number of arguments.

The following statements:

```
printName(‘Joe', ‘Smith')
printName(‘Joe', ‘Smith', True)
printName(‘Joe', lastName = ‘Smith', reverse = True)
```

Would output, with the last two being semantically equivalent:

```
Joe Smith
Smith,Joe
Smith,Joe
```

Functions as Arguments

Function parameters (arguments) can take on any type, even functions.

This means that a function can be passed as a parameter to another function.

When a function is passed as a parameter, the function is not executed, the *reference* to the function is passed.

Example:

```
def funa(x):
   #What is the difference between the following statements.
   print(x)
   print(x())

def funb():
    return 50

funa(funb)
```

Output:

```
<function funb at 0x0000019DE1FC6730>
50
```

Function Calls as Arguments

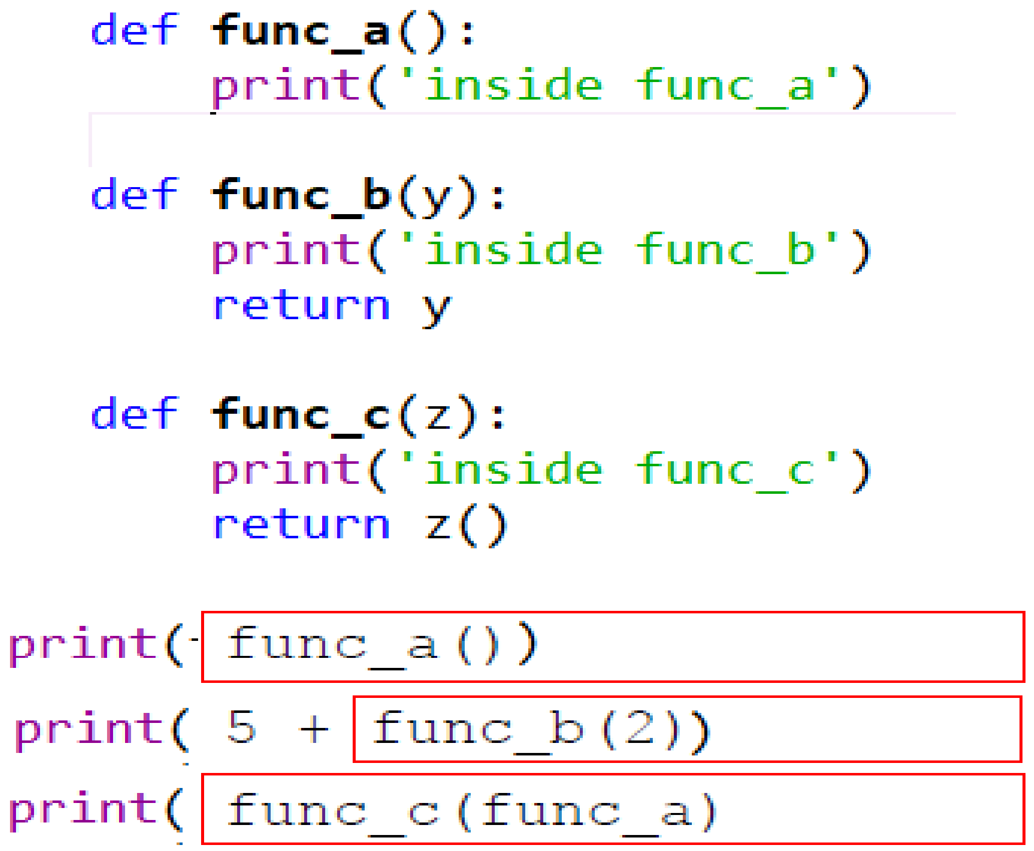
Compare the previous code example with the one below, what is the difference?

```
def funa(x):
   print(x)

def funb():
    return 50

funa(funb())
```

Output:

```
50
```

Functions as Arguments



```
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

Functions as Arguments

```
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

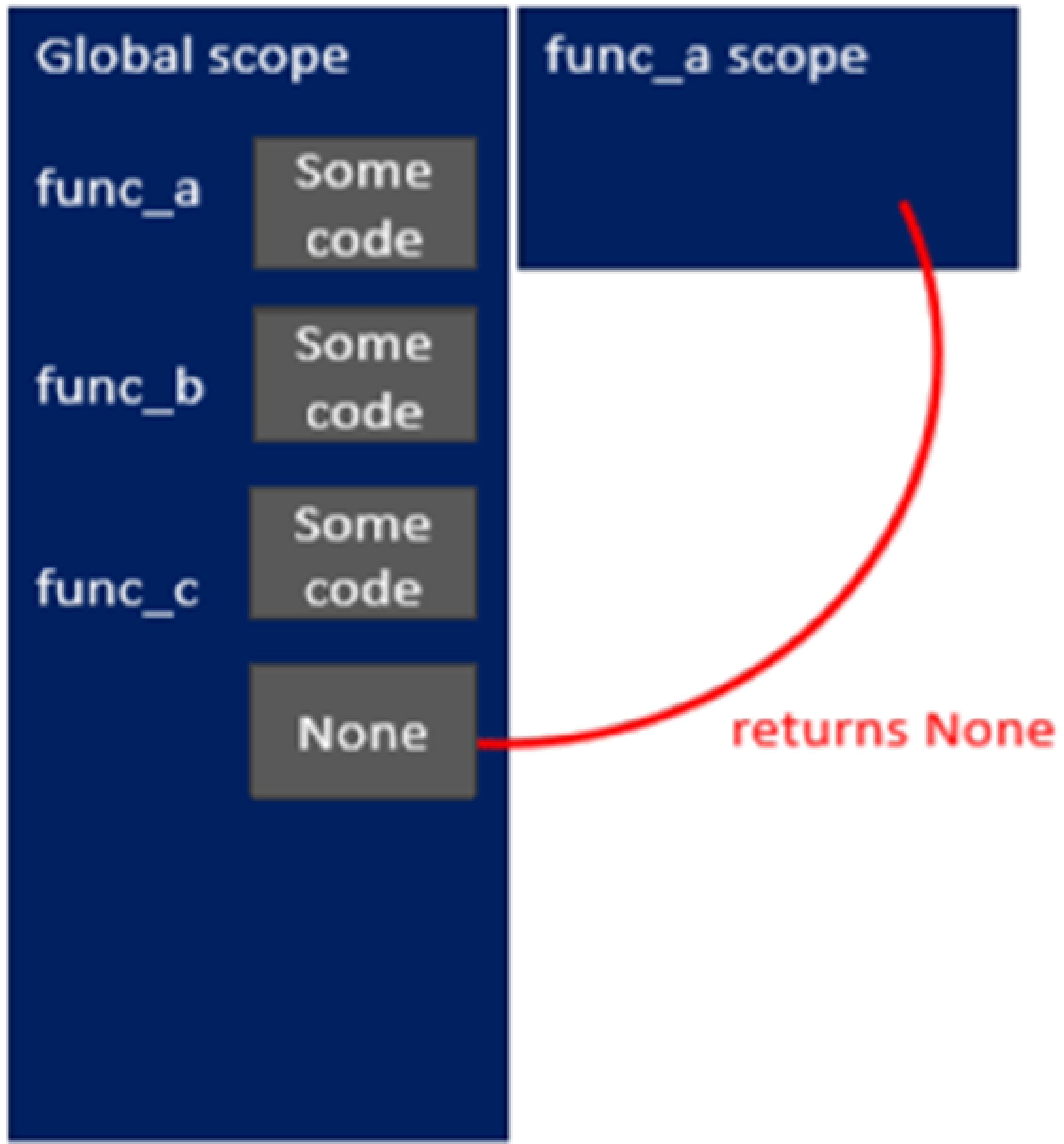
call func_c, takes one parameter, another function

Output:

```
inside func_a
None
inside func_b
7
inside func_c
inside func_a
None
```

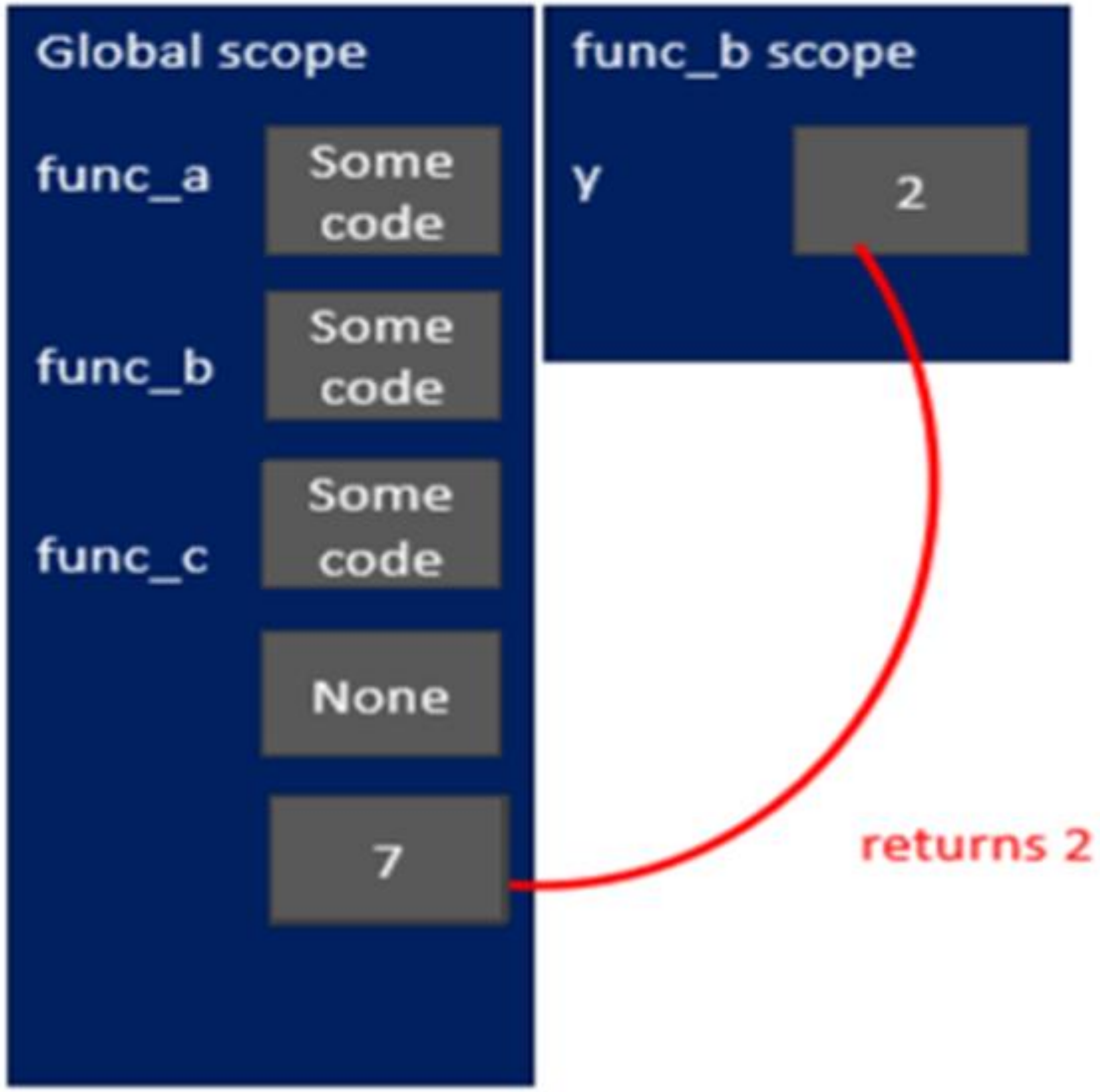
FUNCTIONS AS ARGUMENTS

```
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))
```



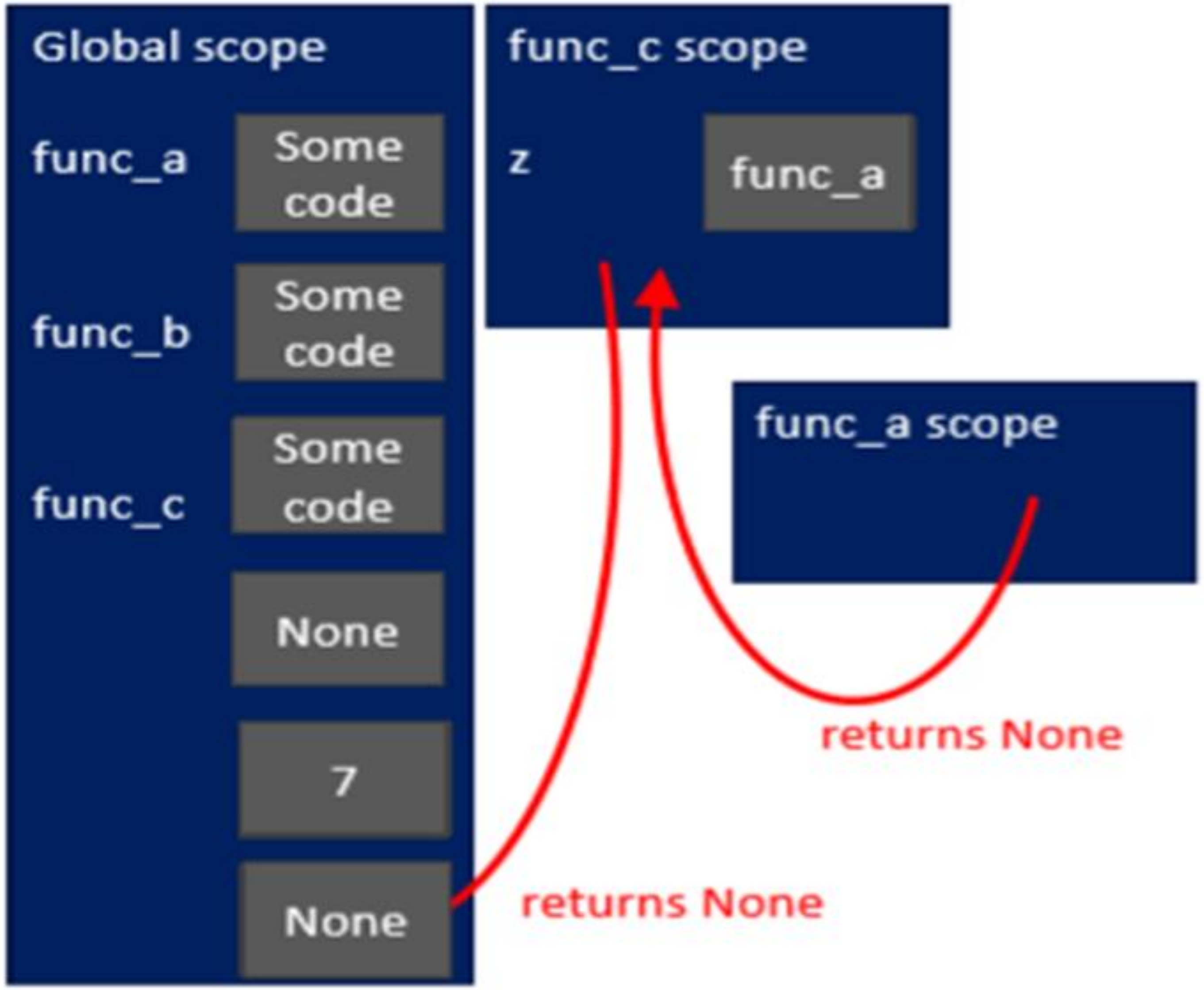
FUNCTIONS AS ARGUMENTS

```
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))
```



FUNCTIONS AS ARGUMENTS

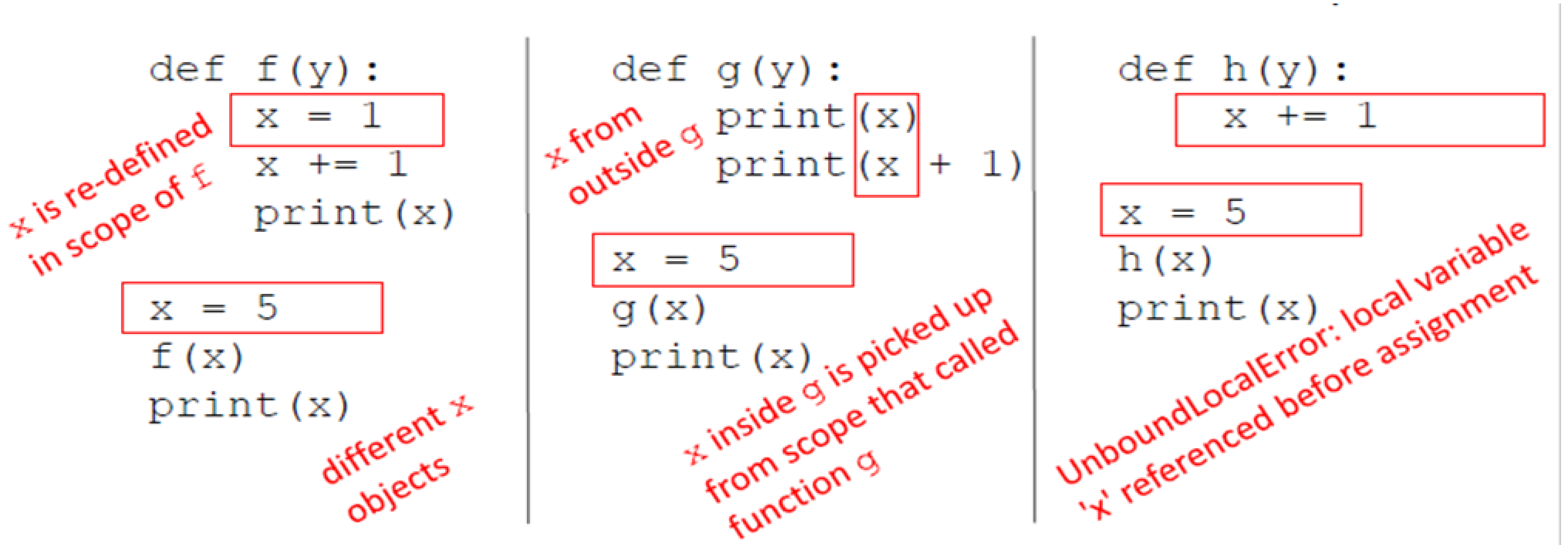
```
def func_a():
    print('inside func_a')
def func_b(y):
    print('inside func_b')
    return y
def func_c(z):
    print('inside func_c')
    return z()
print(func_a())
print(5 + func_b(2))
print(func_c(func_a))
```



SCOPE EXAMPLE

Inside a function, can access a variable defined outside.

Inside a function, cannot modify a variable defined outside – you can using global variables, but it is not recommended.



```
def f(y):
    x = 1
    x += 1
    print(x)

x = 5
f(x)
print(x)
```

```
def g(y):
    print(x)
    print(x + 1)

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

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

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

x is re-defined in scope of f
different x objects
x from outside g
x inside g is picked up from scope that called function g
UnboundLocalError: local variable 'x' referenced before assignment

Output:
2
5

Output:
5
6
5

SCOPE EXAMPLE

```
def f(y):
    x = 1
    x += 1
    print(x)

x = 5
f(x)
print(x)
```

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


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

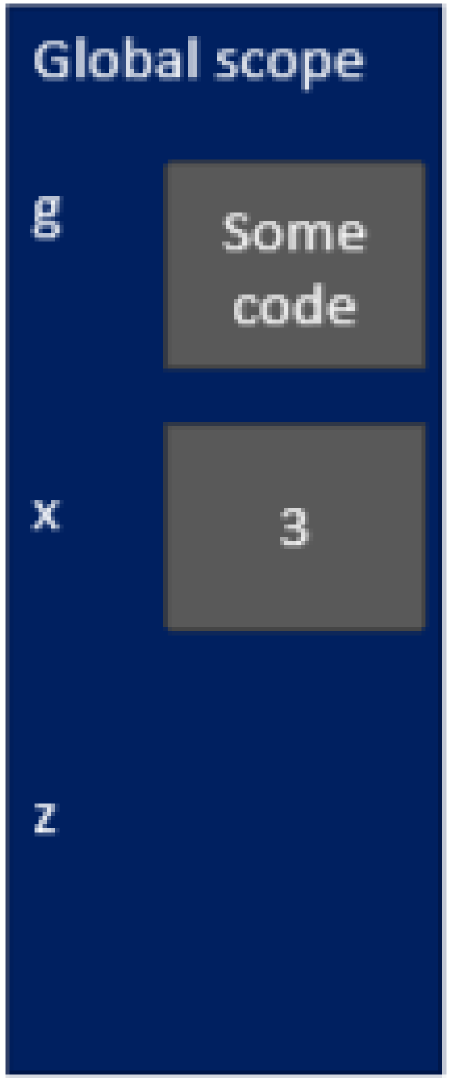
```
def h(y):
    x += 1

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

x from global/main program scope

SCOPE DETAILS

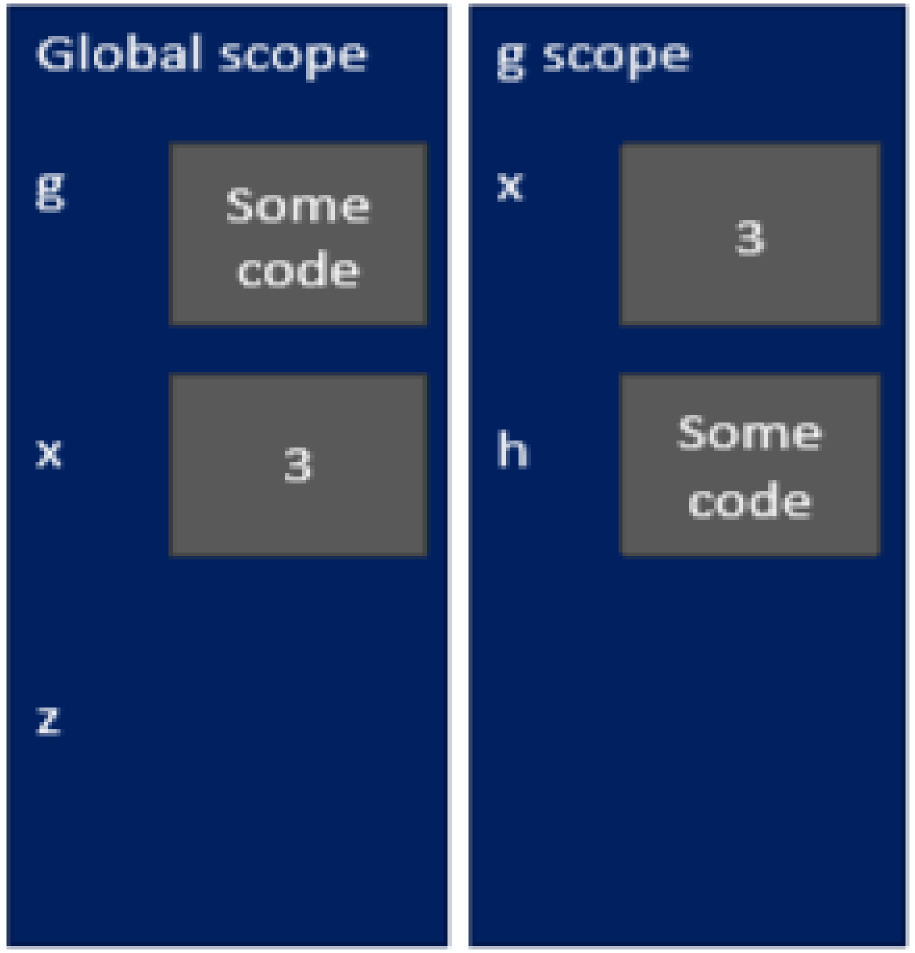
```
def g(x):
    def h():
        x = 'abc'
    x = x + 1
    print('g: x =', x)
    h()
    return x


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

Some code



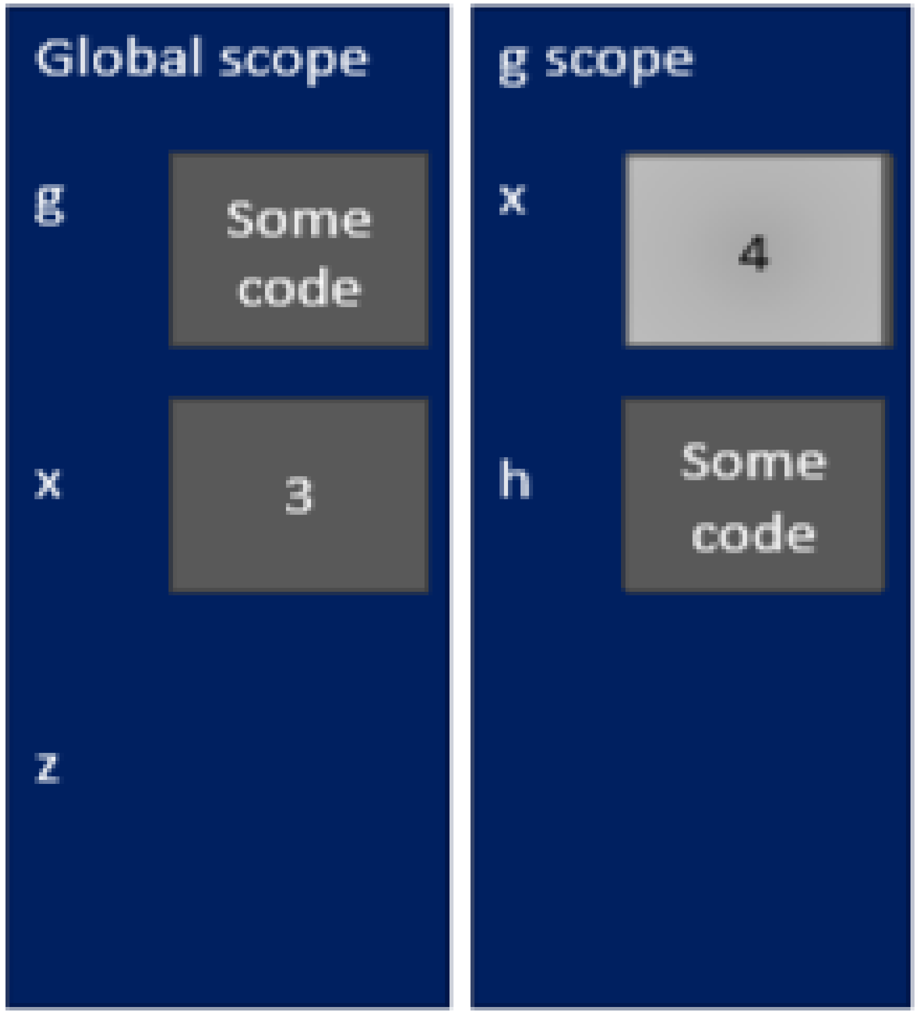
SCOPE DETAILS

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

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



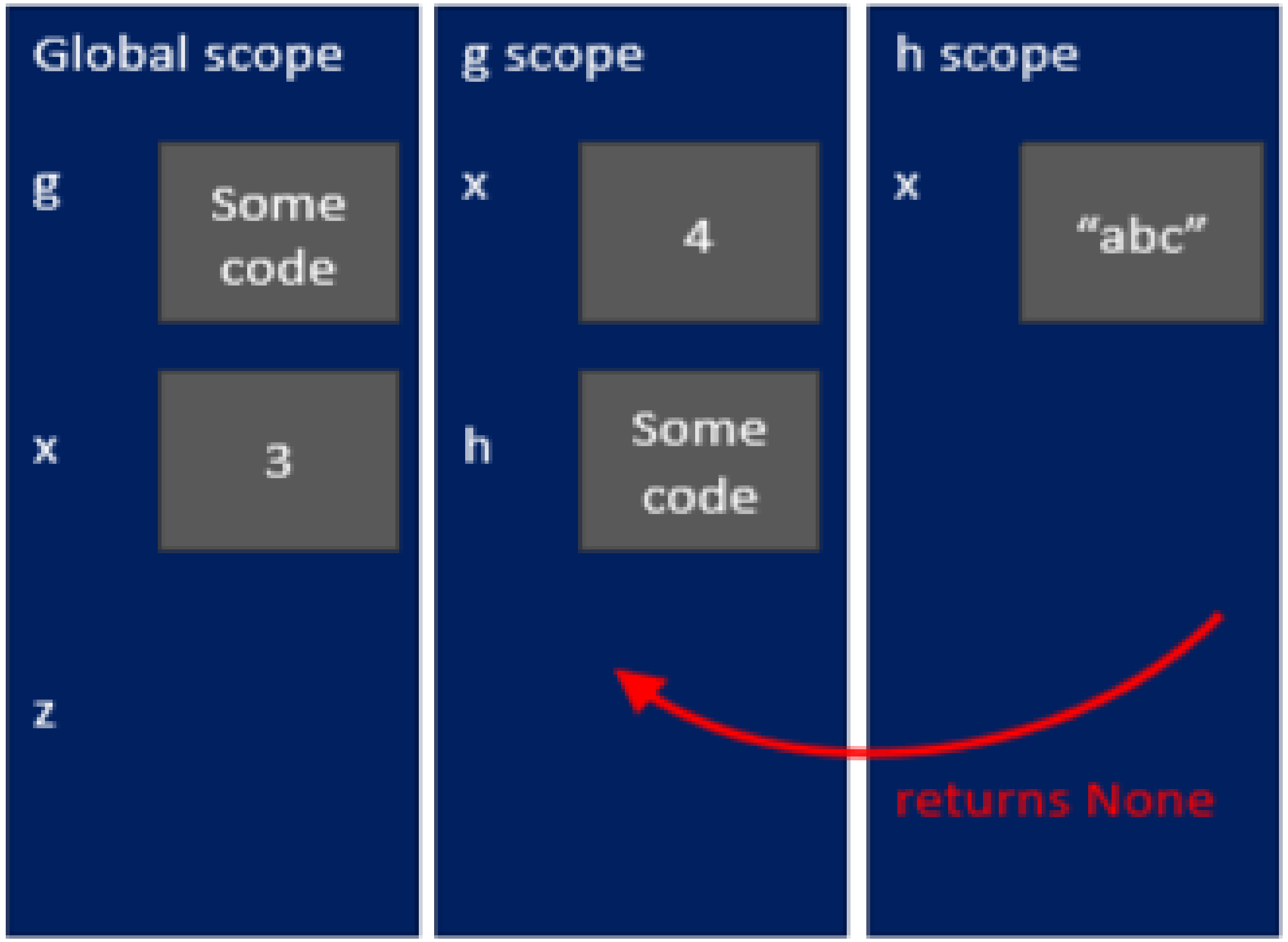
SCOPE DETAILS

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

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



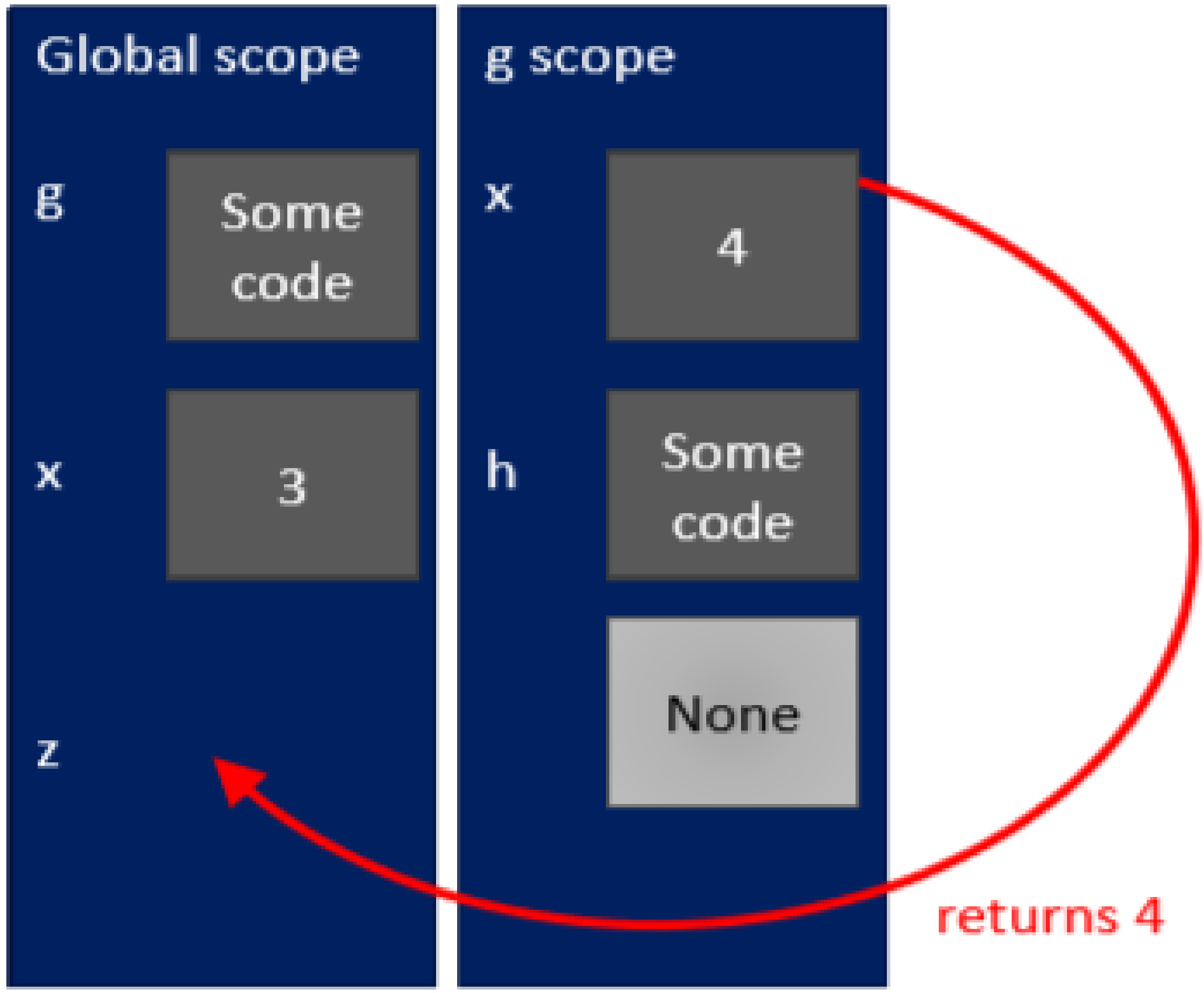
SCOPE DETAILS

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

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



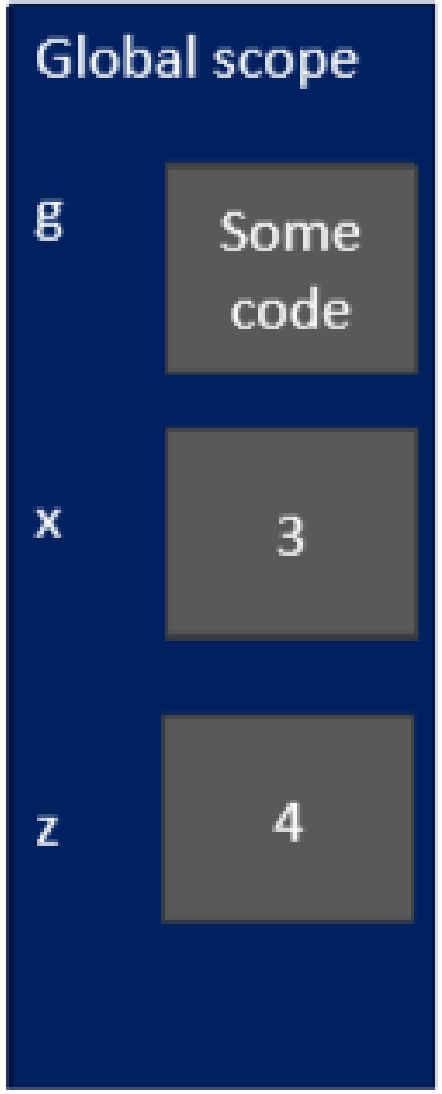
SCOPE DETAILS

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

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



SCOPE DETAILS

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

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



Trace the following program and find the output:

```
def f(x):
    def g():
        x = 'abc'
        print('x = ', x)
    def h():
        z = x
        print('z = ', z)

    x = x + 1
    print('x = ', x)
    h()
    g()
    print('x = ', x)
    return g

x = 3
z = f(x)
print('x = ', x)
print('z = ', z)
z()
```

Trace the following program and find the output:

```
def f(x):
    def g():
        x = 'abc'
        print('x = ', x)
    def h():
        z = x
        print('z = ', z)

    x = x + 1
    print('x = ', x)
    h()
    g()
    print('x = ', x)
    return g

x = 3
z = f(x)
print('x = ', x)
print('z = ', z)
z()
```

Output:

```
x =  4
z =  4
x =  abc
x =  4
x =  3
z =  <function f.<locals>.g at 0x0000027C3EC98400>
x =  abc
```

Global Variables

Python also supports global variables: variables that are defined outside functions.

A global variable is visible from within all functions.

Any function that wants to update a global variable must include a `global` declaration.

Variables that are assigned values/updated inside functions, with the same name as global variables, will not be considered global without the `global` declaration. They will be considered as local variables.

Warning: Global variables are not generally recommended. It is a contradiction of modularity to have variables be accessible in functions in which they are not used. Global variables can cause unexpected results.

Global Variables - Example

```
def f():
    x = 1
    print(x)

x = 5
f()
print(x)
```

Output:

```
1
5
```

```
def g() :
    global x
    x = 1
    print(x)

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

Output:

```
1
1
```

Terms of Use