Introduction to PYTHON

Demystifying the World of Artificial Intelligence and Exploring Its Potential

FUNCTIONS

- write **reusable** pieces/chunks of code, called functions
- functions are not run in a program until they are "**called**" or "**invoked**" in a program
- function characteristics:
  - has a **name**
  - has **parameters** (0 or more)
  - has a **docstring** (optional but recommended)
  - has a **body**
  - **returns** something

HOW TO WRITE and CALL/INVOKE A FUNCTION

```python
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)
```

HOW TO WRITE and CALL/INVOKE A FUNCTION

Name

parameters or arguments

Keyword

```
def is_even( i ):
```

HOW TO WRITE and CALL/INVOKE A FUNCTION

Name

parameters or arguments

Keyword

specification, docstring

```
def is_even( i ):
    """
    Input: i, a positive int
    Returns True if i is even, otherwise False
    """
```

HOW TO WRITE and CALL/INVOKE A FUNCTION

Name

parameters or arguments

Keyword

specification, docstring

```
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
```

Body

```
is_even(3)
```

later in the code, **you call the function using its name and values for parameters**

IN THE 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

expression to evaluate and return

keyword

VARIABLE SCOPE

- **formal parameter** gets bound to the value of **actual parameter** when function is called
- new **scope/frame/environment** created when enter a function
- **scope** is mapping of names to objects

VARIABLE SCOPE

- **formal parameter** gets bound to the value of **actual parameter** when function is called
- new **scope/frame/environment** created when enter a function
- **scope** is mapping of names to objects

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

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

formal parameter

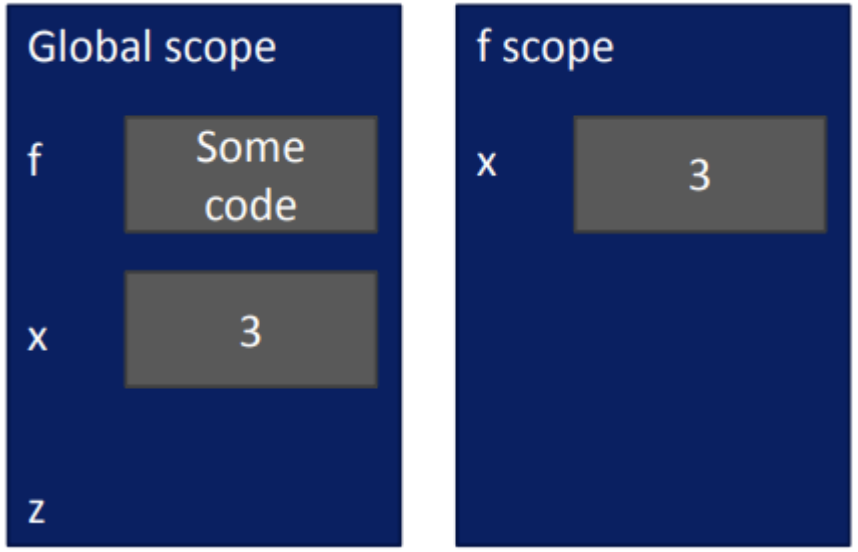
actual parameter

Function definition

Main program code
* initializes a variable x
* makes a function call f(x)
* assigns return of function to variable z

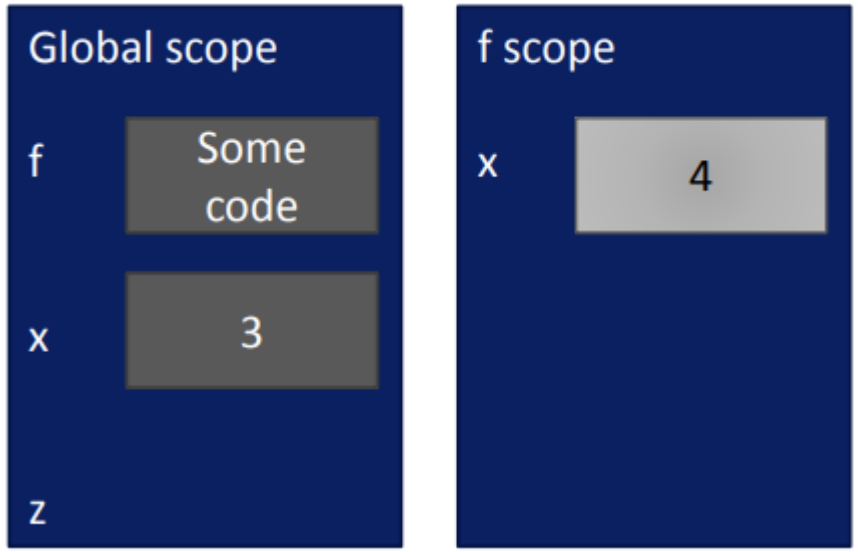
VARIABLE SCOPE

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

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



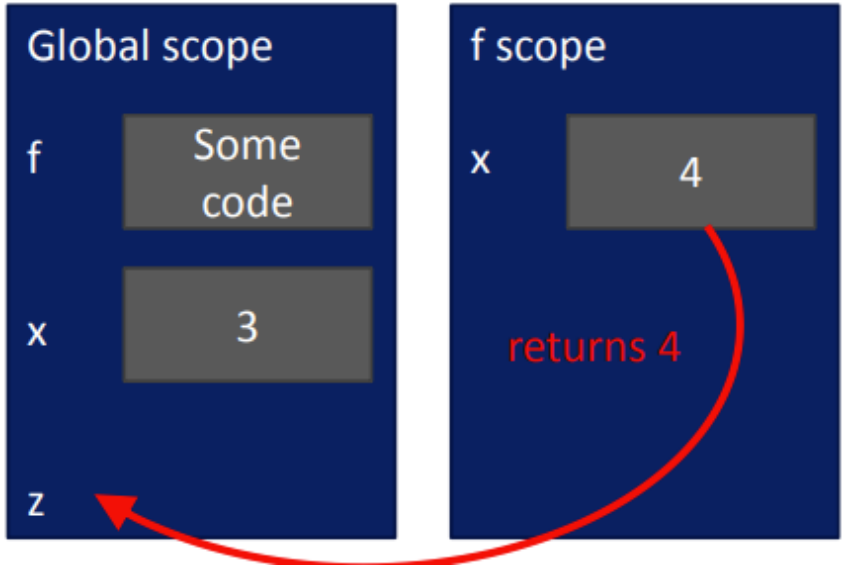
VARIABLE SCOPE

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

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



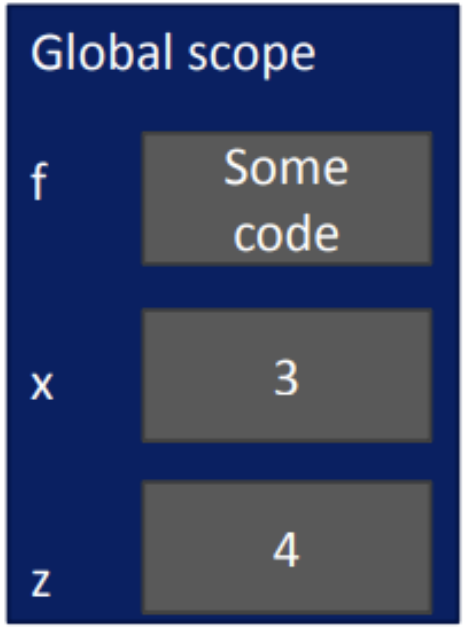
VARIABLE SCOPE

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

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



VARIABLE SCOPE

```python
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
def is_even( i ):
  """
  Input: i, a positive int
  Does not return anything
  """
  i%2 == 0
```

Without a return statement

- Python returns the value **None, if no return given**
- represents the absence of a value

return vs print

- return only has meaning **inside** a function
- only **one** return executed inside a function
- code inside function but after return statement not executed
- has a value associated with it, **given to function caller**

- print can be used **outside** functions
- can execute **many** print statements inside a function
- code inside function can be executed after a print statement
- has a value associated with it, **outputted** to the console

FUNCTIONS AS ARGUMENTS

- 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

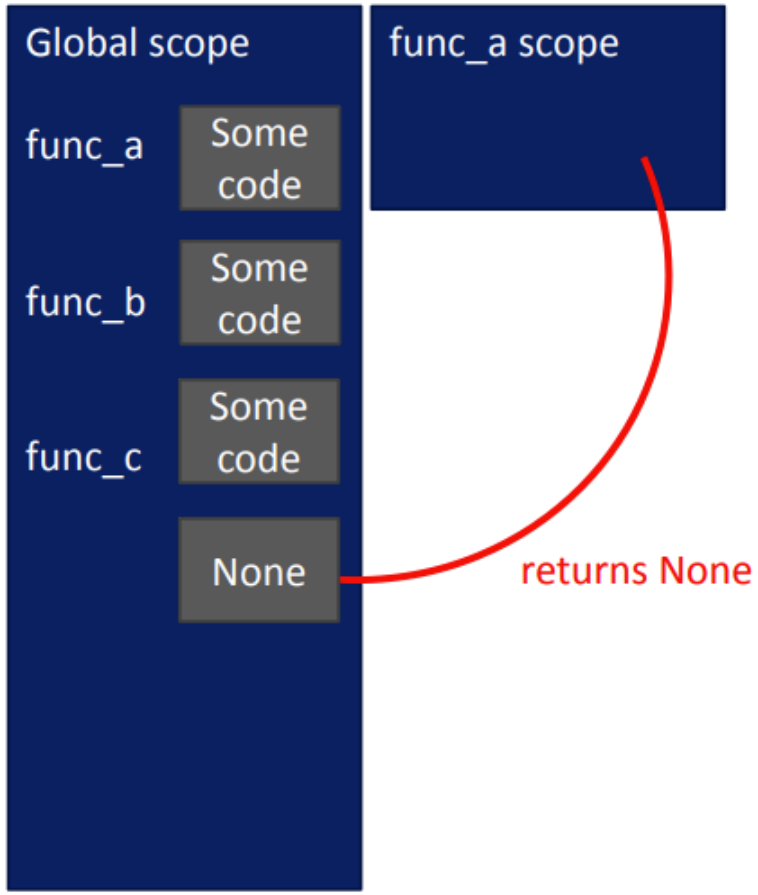
call func b, takes one parameter

call func_c, takes one parameter, another function

FUNCTIONS AS ARGUMENTS

- arguments can take on any type, even functions

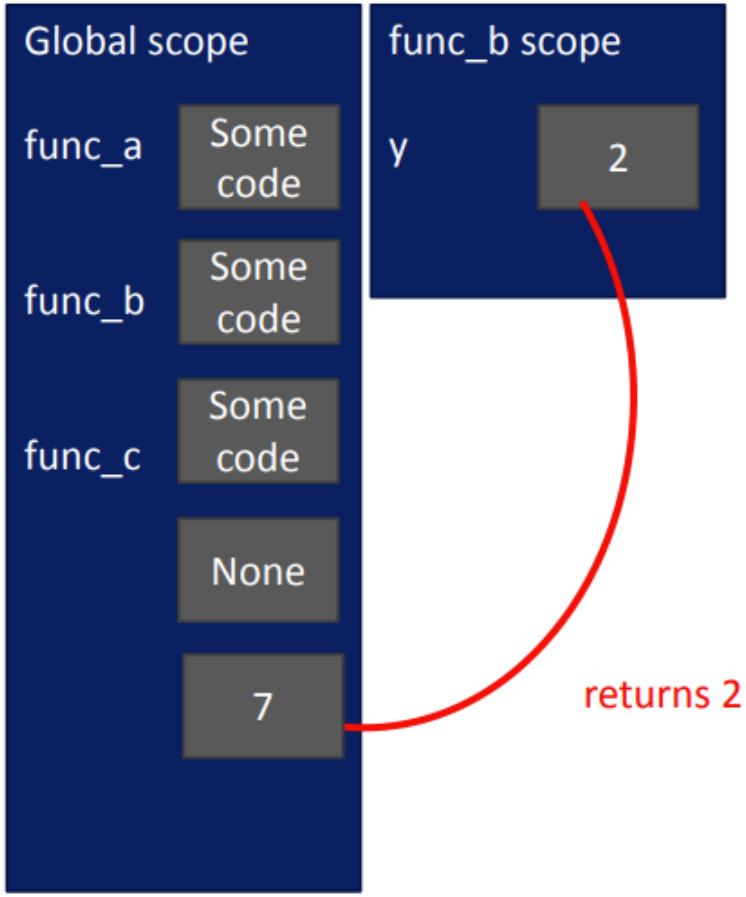
```
def func_a():
  print 'inside func_a'
def func_b(y):
  print 'inside func_b'
  return y
def func_c(z):
  print 'inside func_c'
  return z()

print func_a()
print 5 + func_b(2)
print func_c(func_a)
```



FUNCTIONS AS ARGUMENTS

- arguments can take on any type, even functions

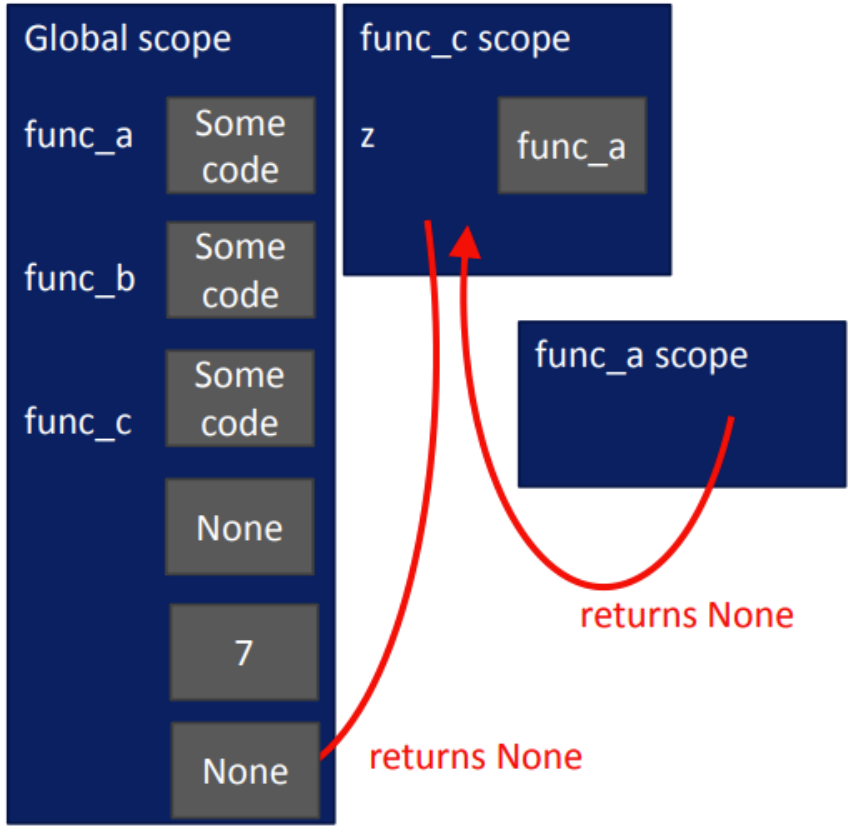
```
def func_a():
  print 'inside func_a'
def func_b(y):
  print 'inside func_b'
  return y
def func_c(z):
  print 'inside func_c'
  return z()

print func_a()
print 5 + func_b(2)
print func_c(func_a)
```



FUNCTIONS AS ARGUMENTS

- 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)
```



SCOPE EXAMPLE

- **inside** a function, **can access** a variable defined outside
- **inside** a function, **cannot modify** a variable defined

outside -- can using **global variables**, but frowned upon

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

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

x is re-defined in scope of f

different x objects

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

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

x from outside g

x inside g is picked up from scope that called function g

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

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

UnboundLocalError: local variable 'x' referenced before assignment

SCOPE EXAMPLE

- **inside** a function, **can access** a variable defined outside
- **inside** a function, **cannot modify** a variable defined

outside -- can using **global variables**, but frowned upon

```
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)
```

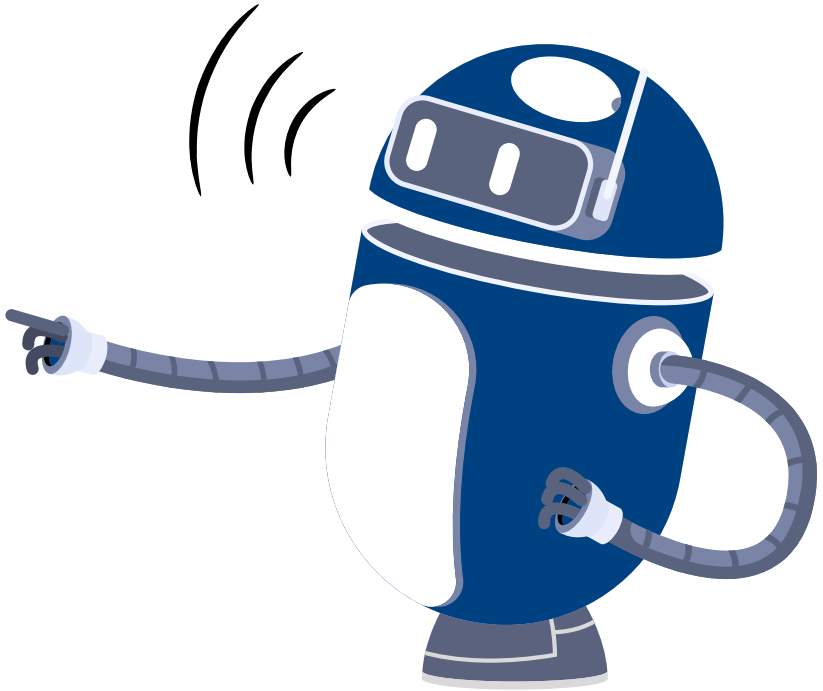
```
def h(y):
    x += 1

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

x from global/main program scope

HARDER SCOPE EXAMPLE

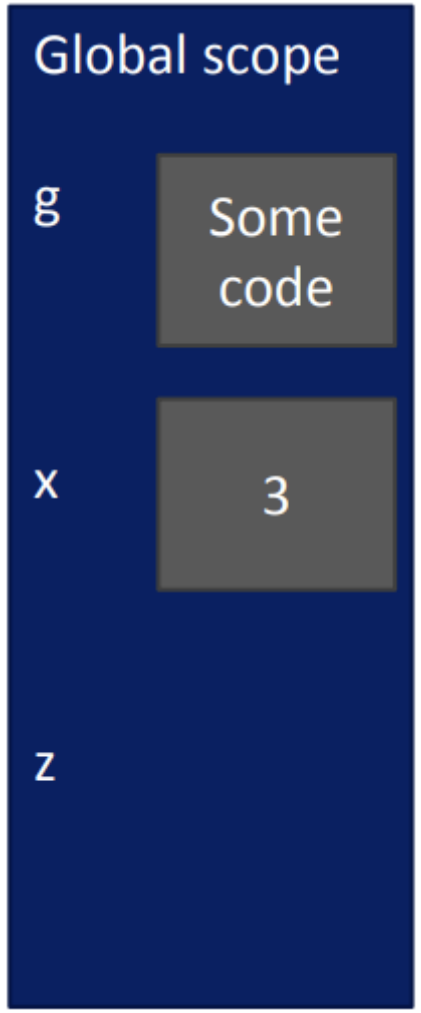
IMPORTANT and TRICKY!

HERE



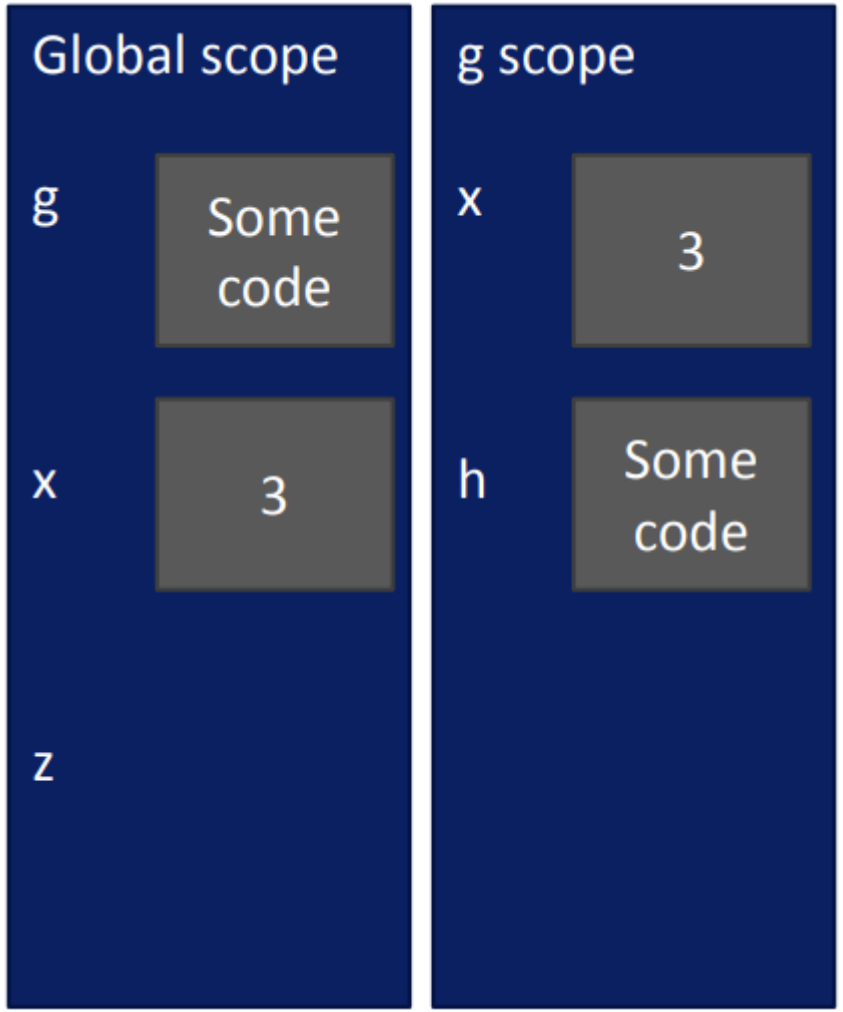
SCOPE DETAILS

```python
def g(x):
  def h():
    x = 'abc'
    x = x + 1
    print('g: x =', x)
    h()
    return x
x = 3
z = g(x)
```



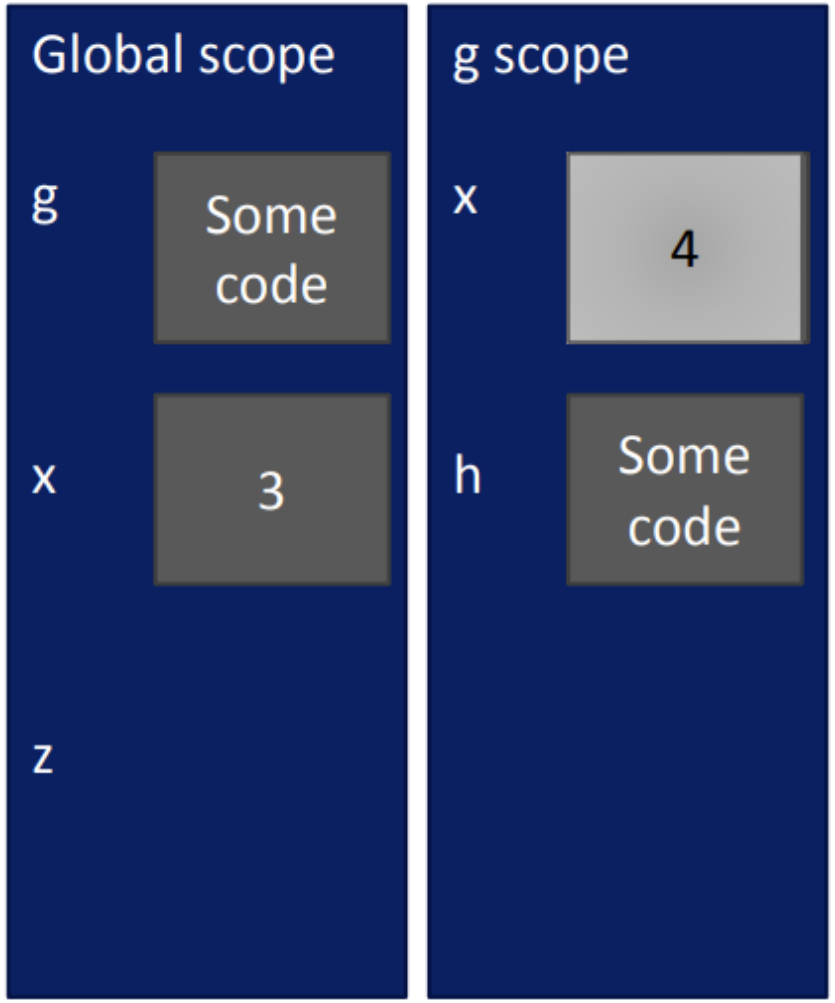
SCOPE DETAILS

```python
def g(x):
  def h():
    x = 'abc'
    x = x + 1
    print('g: x =', x)
    h()
    return x
x = 3
z = g(x)
```



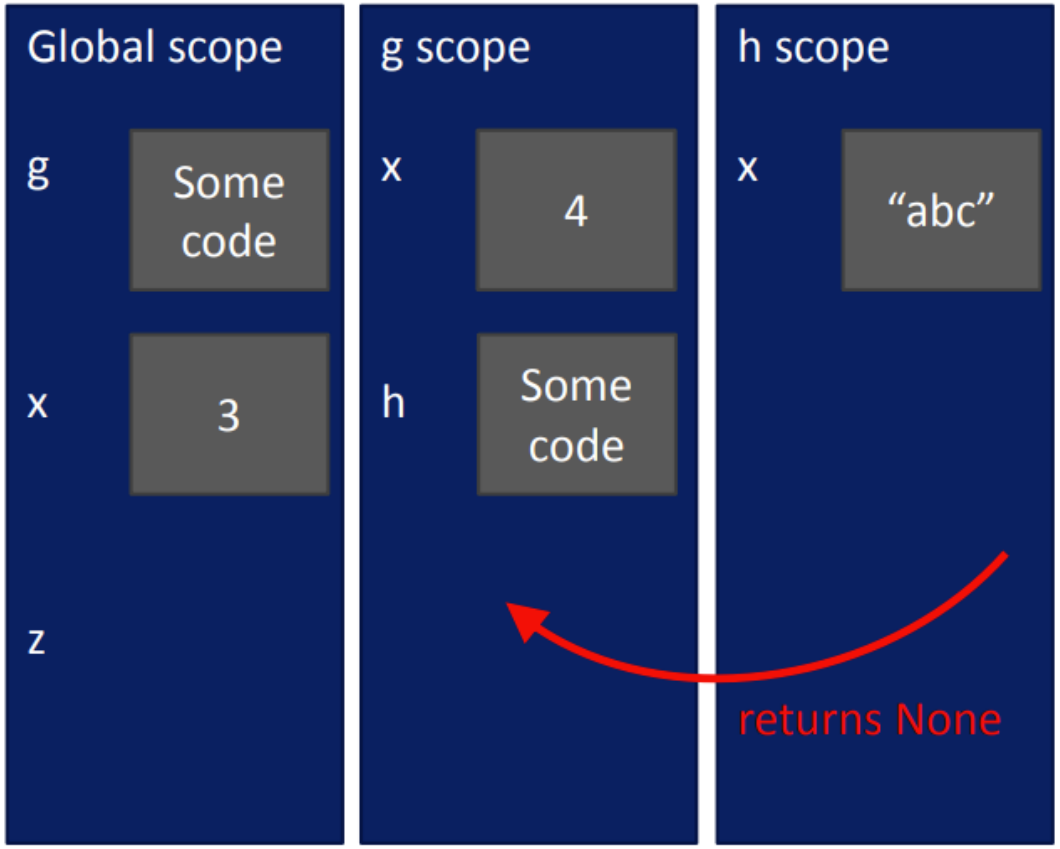
SCOPE DETAILS

```python
def g(x):
  def h():
    x = 'abc'
    x = x + 1
    print('g: x =', x)
    h()
    return x
x = 3
z = g(x)
```



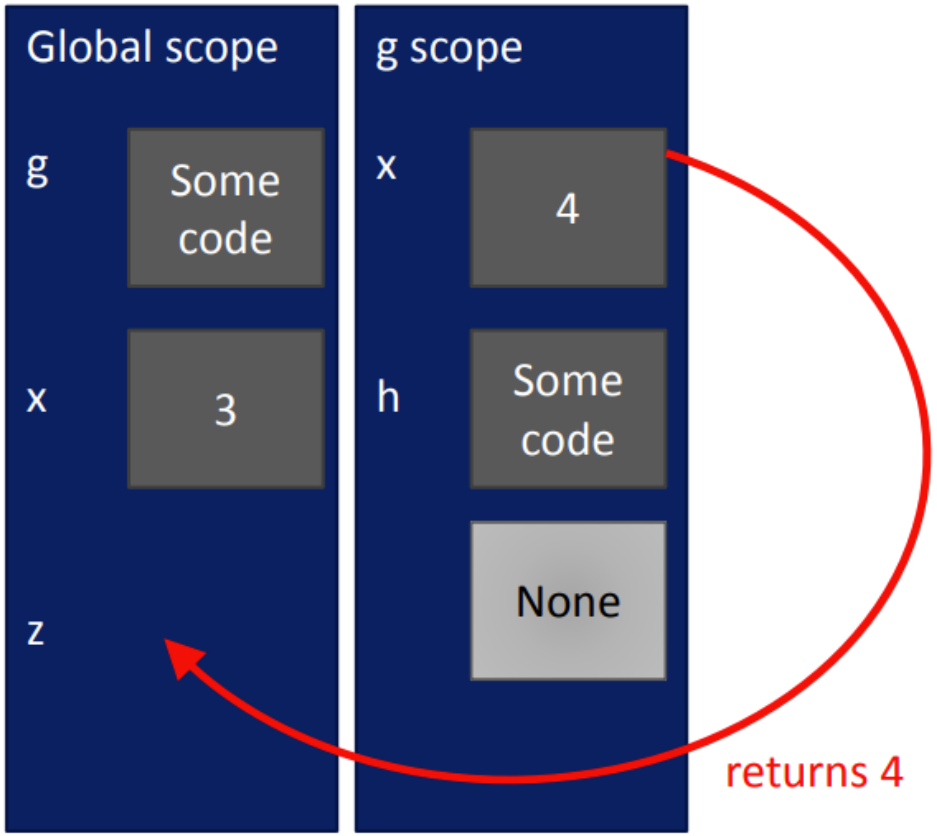
SCOPE DETAILS

```
def g(x):
  def h():
    x = 'abc'
    x = x + 1
    print('g: x =', x)
    h()
    return x
x = 3
z = g(x)
```



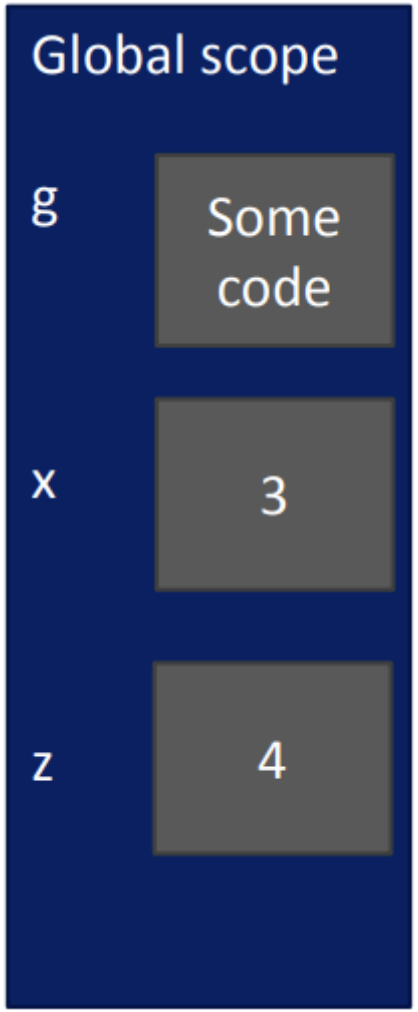
SCOPE DETAILS

```python
def g(x):
  def h():
    x = 'abc'
    x = x + 1
    print('g: x =', x)
    h()
    return x
x = 3
z = g(x)
```



SCOPE DETAILS

```python
def g(x):
  def h():
    x = 'abc'
    x = x + 1
    print('g: x =', x)
    h()
    return x
x = 3
z = g(x)
```



THANK YOU!

Any Question!