Python Programming

Lecture03

Functions

- A Function is a set of instructions that is used to perform a single related operation.
- Functions are:
  - » Organized
  - » Re-usable
  - » Modularized
- Python has many built-in functions like print(), file(), input(), int(), len(), etc. but you can also create your own functions.
- These functions are called **User-defined Functions.**

Syntax:

```
def name( parameter1, parameter2,...,parameterN ):
        Satement1
        Statement2
        return [expression]
```

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

keyword

name

parameters

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

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

```
X=3
is_even(x)
```

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
```

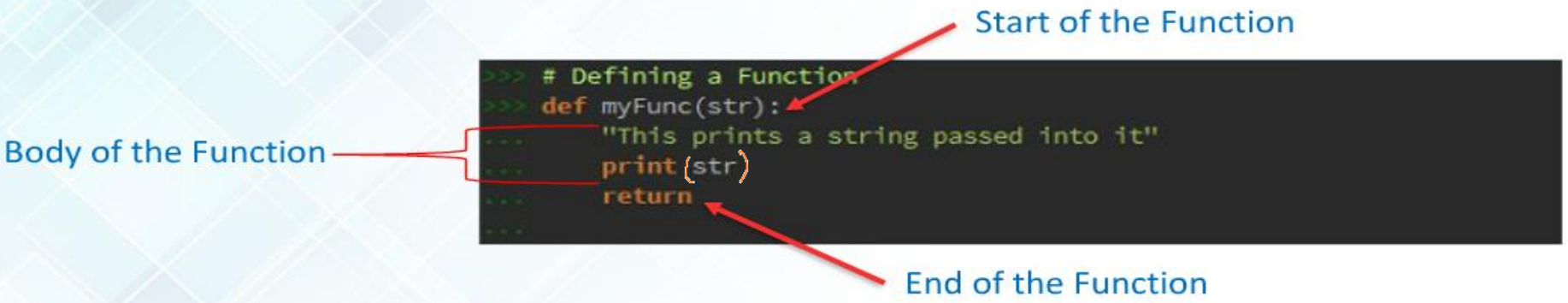
run some commands

keyword

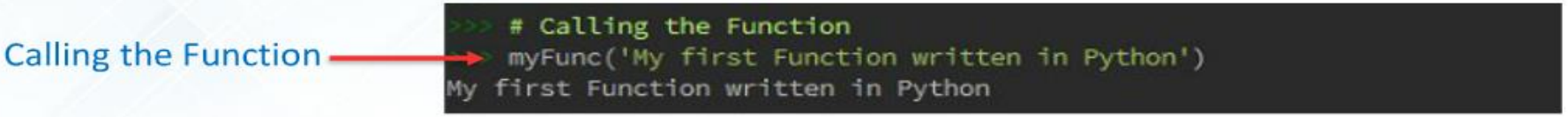
expression to evaluate and return

Functions

- **Defining a Function**



- **Calling the Function**



VARIABLE SCOPE

- **formal parameter/ parameter** gets bound to the value of **actual parameter / argument** 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

ONE WARNING IF NO `return` STATEMENT

```
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

Various Forms of Function Arguments

There are 4 types of actual arguments are allowed in Python.

1) Positional Arguments

2) Keyword Arguments

3) Default Arguments

4) Variable Length Arguments

Python Positional Arguments --Examples

```
def greet(name,msg):

        """This function greets to
        the person with the provided message"""

        print("Hello", name + ', ' + msg)

greet("Monica", "Good morning!")
greet("Good morning!“, ”Monica“)
```

These are the arguments passed to function in correct positional order. If we change the order then result may be changed.

The number of arguments must equal to the number of parameters otherwise error will be generated.

```
def sub(a, b):
    print(a-b)
sub(100, 200)
sub(200, 100)
```

Python Default Arguments--Examples

Sometimes we can provide default values for our positional arguments.
can provide a default value to an argument by using the assignment operator (=).

```
def greet(name, msg = "Good morning!"):
    """
    This function greets to the person with
    the provided message.

    If message is not provided, it defaults to
    "Good morning!"
    """
    print("Hello",name + ', ' + msg)

greet("Kate")
greet("Bruce","How do you do?")
```

In this function, the parameter **name** does not have a default value and is required (mandatory) during a call.

On the other hand, the parameter **msg** has a default value of "Good morning!". So, it is optional during a call. If a value is provided, it will overwrite the default value.

Any number of arguments in a function can have a default value. But once we have a default argument, all the arguments to its right must also have default values.

This means to say, non-default arguments cannot follow default arguments.

Python Keyword Arguments--Examples

- **We can pass argument values by**
- **keyword i.e by parameter name.**

- **Keyword argument in the form of**
- **name= value .**

```
def greet(name, msg = "Good morning!"):
    """
    This function greets to the person with
    the provided message.

    If message is not provided, it defaults to
    "Good morning!"
    """
    print("Hello",name + ', ' + msg)
```

 # 2 keyword arguments

```
greet(name = "Bruce",msg = "How do you do?")
```

 # 2 keyword arguments (out of order)

```
greet(msg = "How do you do? ",name = "Bruce")
```

 # 1 positional, 1 keyword argument

```
greet("Bruce",msg = "How do you do?").
```

Having a positional argument after keyword arguments will result into errors.

```
greet(name="Bruce","How do you do?") //error
```

Python Keyword Arguments--Examples

Here the order of arguments is not important but number of arguments must be matched.

Note: We can use both positional and keyword arguments simultaneously. But first we have to take positional arguments and then keyword arguments, otherwise we will get syntax error.

```
def wish(name,msg):
    print("Hello",name,msg)
    wish("DPSharma","GoodMorning") → Valid
    wish("DPSharma",msg="GoodMorning") → Valid
    wish(name="DPSharma","GoodMorning") → Invalid

SyntaxError: positional argument follows keyword argument
```

Python Arbitrary Arguments --Examples

- Sometimes, we do not know in advance the number of arguments that will be passed into a function. Python allows us to handle this kind of situation through function calls with arbitrary number of arguments.

- function definition use an asterisk (*) before the parameter name to denote this kind of argument

```
def greet(*names):
    """This function greets all
    the person in the names tuple."""

    # names is a tuple with arguments
    for name in names:
        print("Hello", name)

greet("Monica","Luke","Steve","John")
```

Python Arbitrary Arguments --Examples

We can call this function by passing any number of arguments including zero number.

Internally all these values represented in the form of tuple.

```
1)  def sum(*n):
2)      total=0
3)      for n1 in n:
4)          total=total+n1
5)      print("The Sum=",total)
6)
7)  sum()
8)  sum(10)
9)  sum(10,20)
10) sum(10,20,30,40)
```

Output

```
The Sum= 0
The Sum= 10
The Sum= 30
The Sum= 100
```

We can mix variable length arguments with positional arguments.

```
1)  def  f1(n1,*s):
2)      print(n1)
3)      for s1 in s:
4)          print(s1)
5)
6)  f1(10)
7)  f1(10,20,30,40)
8)  f1(10,"A",30,"B")
```

Python Arbitrary Arguments --Examples

Note: After variable length argument, if we are taking any other arguments then we should provide values as keyword arguments.

```
1)  def f1(*s,n1):
2)      for s1 in s:
3)          print(s1)
4)      print(n1)
5)
6)  f1("A","B",n1=10)
```

Output

```
A
B
10
```

f1("A","B",10) → Invalid

TypeError: f1() missing 1 required keyword-only argument: 'n1'

Python Arbitrary Arguments --Examples

Note: We can declare keyword variable length arguments also.

- For this we have to use **. Ex. **def** f1(**n):
- We can call this function by passing any number of keyword arguments.
- Internally these keyword arguments will be stored inside a dictionary.

```
def display(**kwargs):

    for k,v in kwargs.items():
        print(k,"=",v)

display(n1=10,n2=20,n3=30)
display(rno=100,name="DPSharma",marks=70,subject="Java")
```

Output:

```
n1 = 10
n2 = 20
n3 = 30
rno = 100
name = DPSharma
marks = 70
subject = Java
```

Python Functions—Case Study

```
def f(arg1,arg2,arg3=4,arg4=8):
    print(arg1,arg2,arg3,arg4)
```

1) f(3,2) → 3 2 4 8
2) f(10,20,30,40) → 10 20 30 40
3) f(25,50,arg4=100) → 25 50 4 100
4) f(arg4=2,arg1=3,arg2=4) → 3 4 4 2

5) f() → Invalid
TypeError: f() missing 2 required positional arguments: 'arg1' and 'arg2'

6) f(arg3=10, arg4=20, 30, 40) → Invalid
SyntaxError: positional argument follows keyword argument
[After keyword arguments we should not take positional arguments]

7) f(4, 5, arg2 = 6) → Invalid
TypeError: f() got multiple values for argument 'arg2'

8) f(4, 5, arg3 = 5, arg5 = 6) → Invalid
TypeError: f() got an unexpected keyword argument 'arg5'

VARIABLE SCOPE

Local Variables

- The variables defined within the function has a local scope and hence they are called local variables.
- Local scope means they can be accessed within the function only.
- They appear when the function is called and disappear when the function exits.

Global Variables

- The variables defined outside the function has a global scope and hence they are called global variables.
- Global scope means they can be accessed within the function as well as outside the function.
- The value of a global variable can be used by referring the variable as **global** inside a function.

Global Variable

Local Variable

```
>>> # Example of Local and Global variable
>>> var = 10
>>> def fun():
...     # Referring global variable var
...     global var
...     varLocal = var * 2
```

Global Keyword

We can use global keyword for the following 2 purposes:

- To declare global variable inside function
- To make global variable available to the function so that we can perform required modifications

Common points to remember

- global a=10 ✗ global a ✓
  a=10
- before global declaration, we can not use the variable inside the function

```
a=10
def f1():
    print(a)
    global a
```
✗

```
a=10
def f1():
    global a
    print(a)
```
✓

VARIABLE SCOPE

- **formal parameter/ parameter** gets bound to the value of **actual parameter / argument** 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

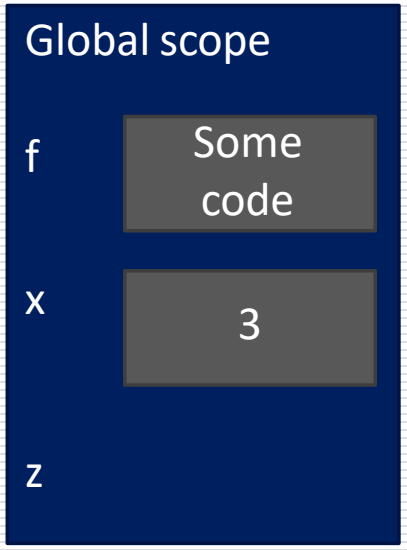
actual parameter

Function definition

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

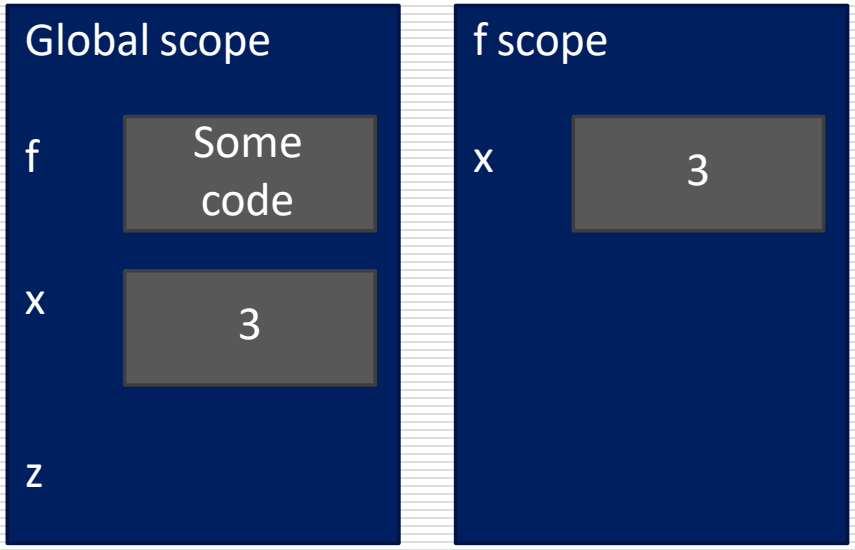
VARIABLE SCOPE

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

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



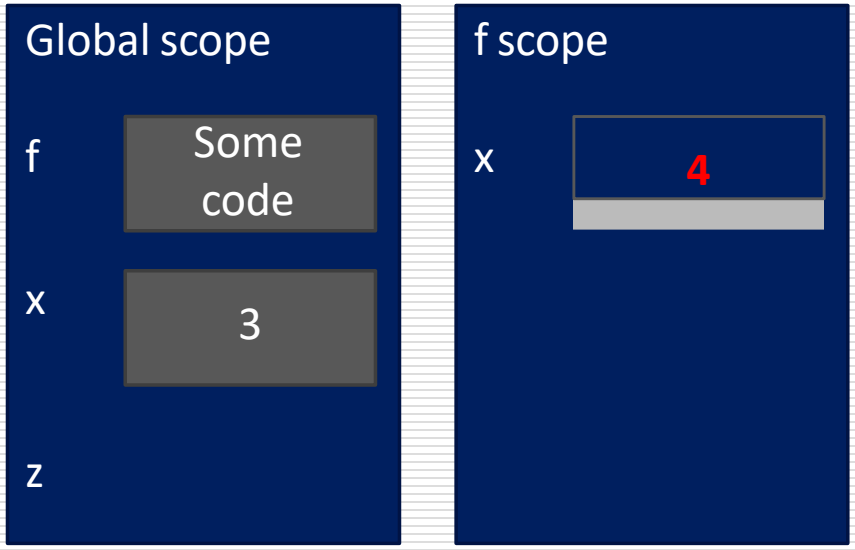
VARIABLE SCOPE

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

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



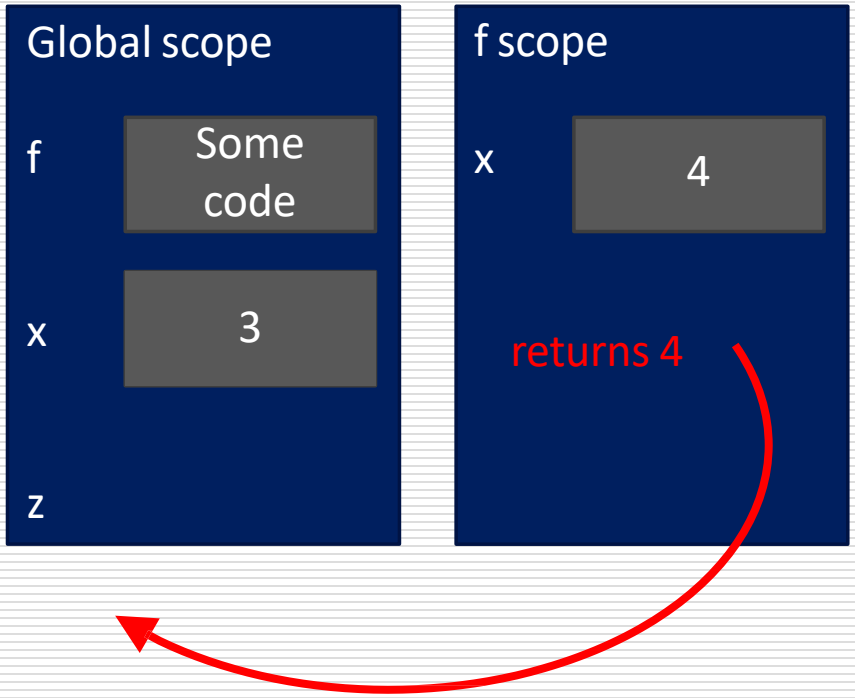
VARIABLE SCOPE

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

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



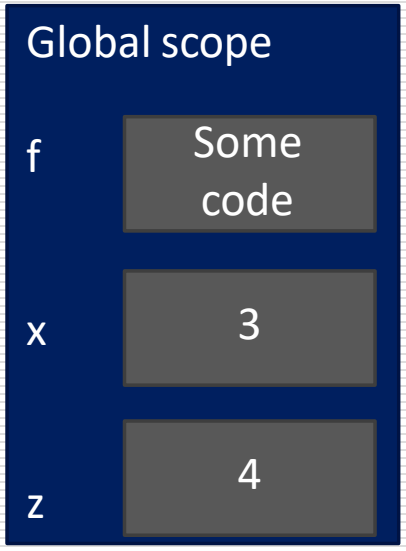
VARIABLE SCOPE

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

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



VARIABLE SCOPE

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

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



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))
```

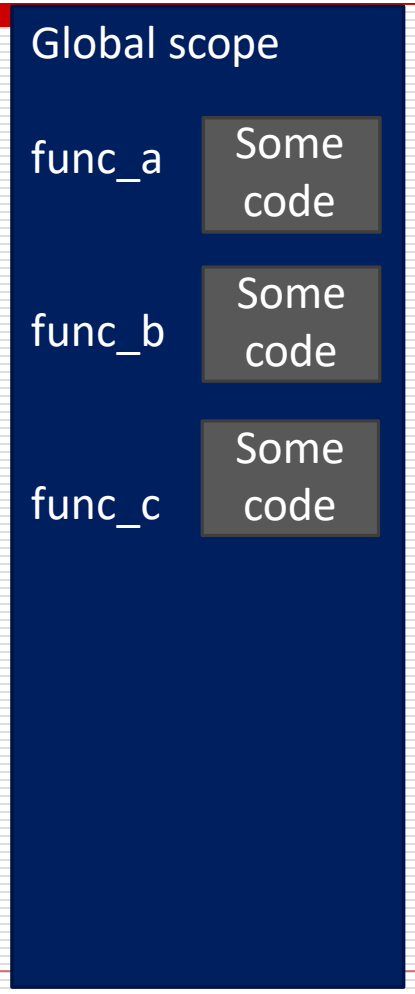
call `func_a`, takes no parameters

call `func_b`, takes one parameter

call `func_c`, takes one parameter, another function

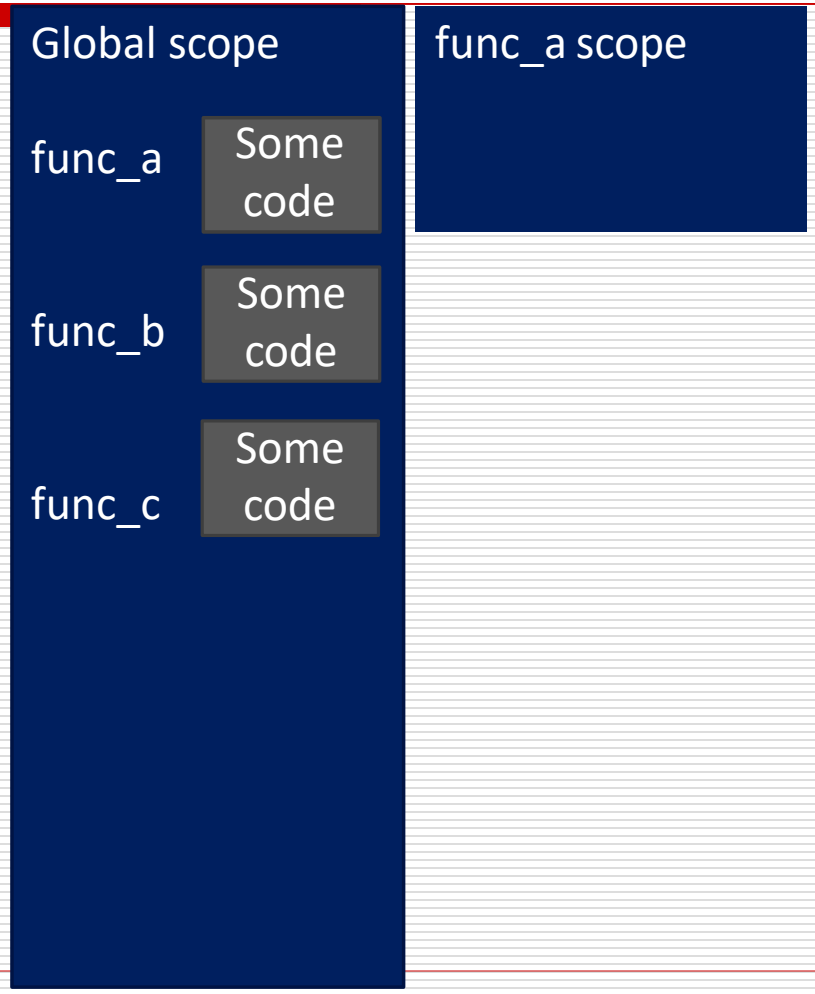
FUNCTIONS AS ARGUMENTS

```
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))
```



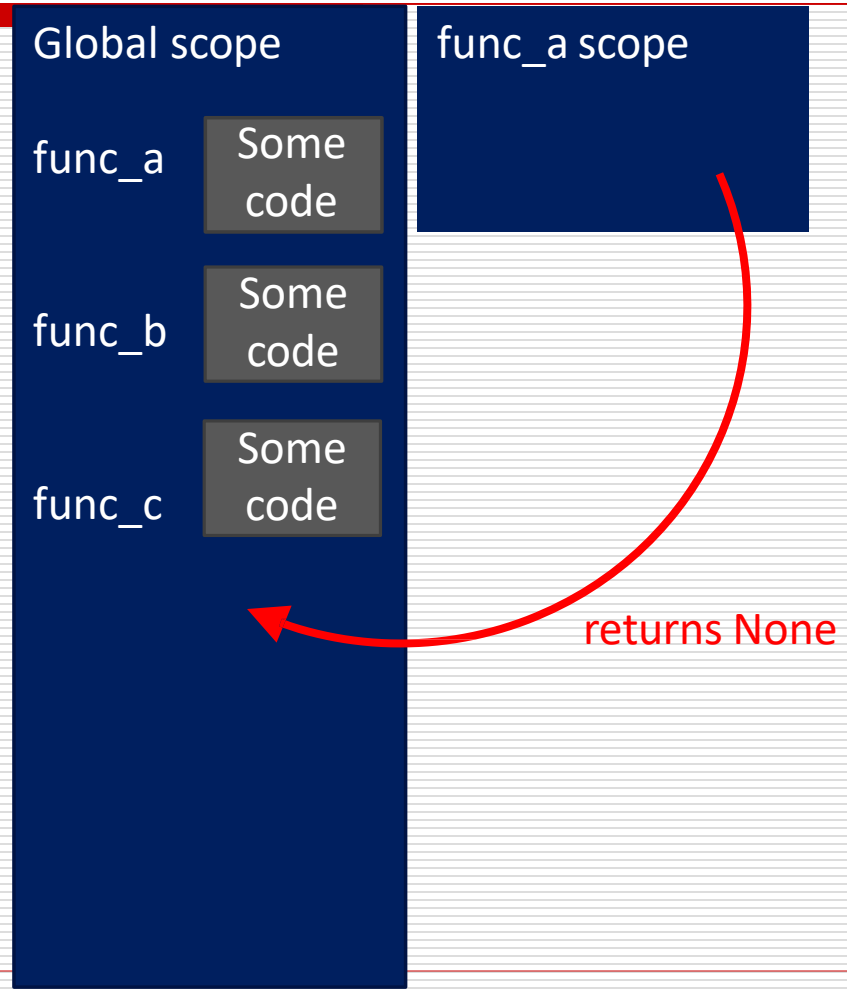
FUNCTIONS AS ARGUMENTS

```
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))
```



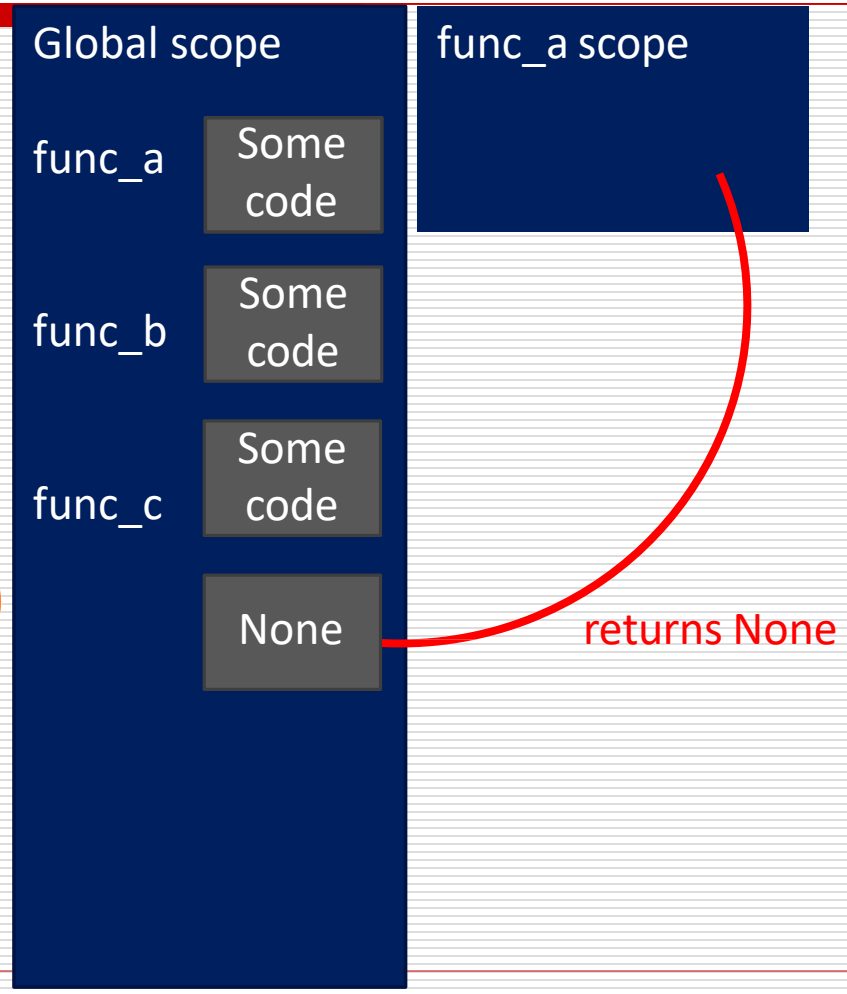
FUNCTIONS AS ARGUMENTS

```
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))
```



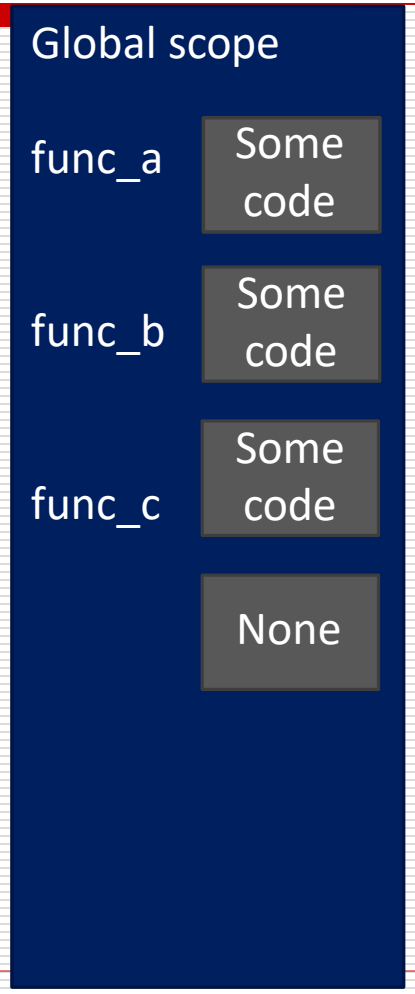
FUNCTIONS AS ARGUMENTS

```
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))
```



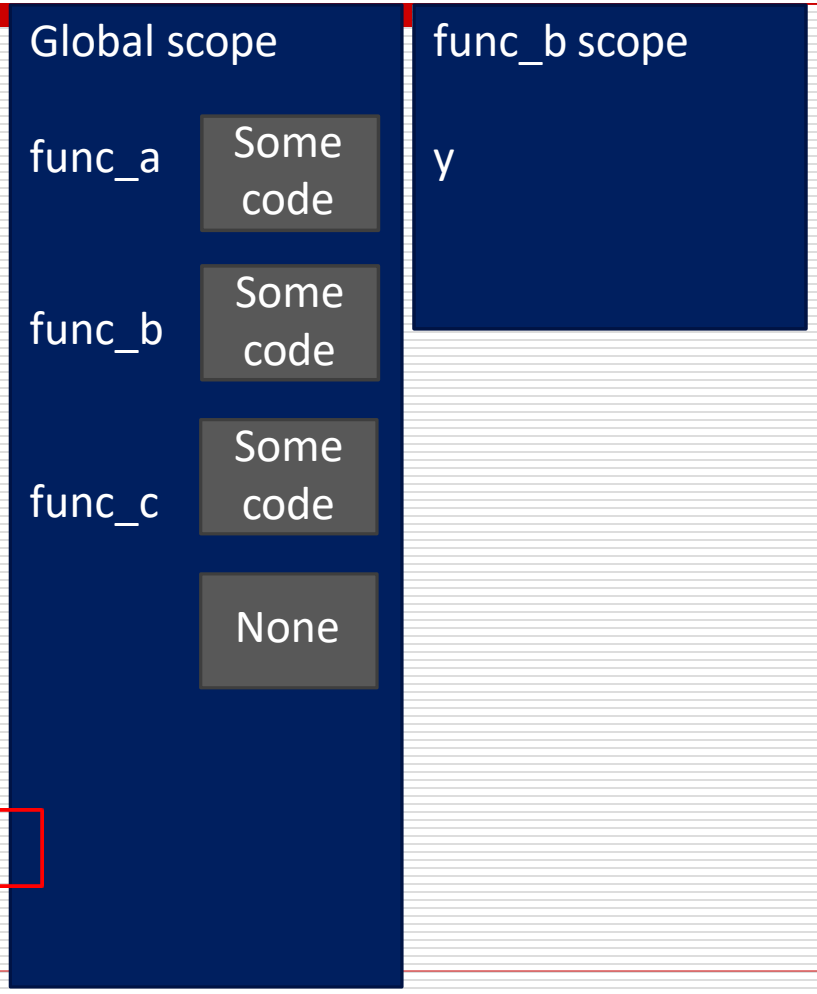
FUNCTIONS AS ARGUMENTS

```
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)
```



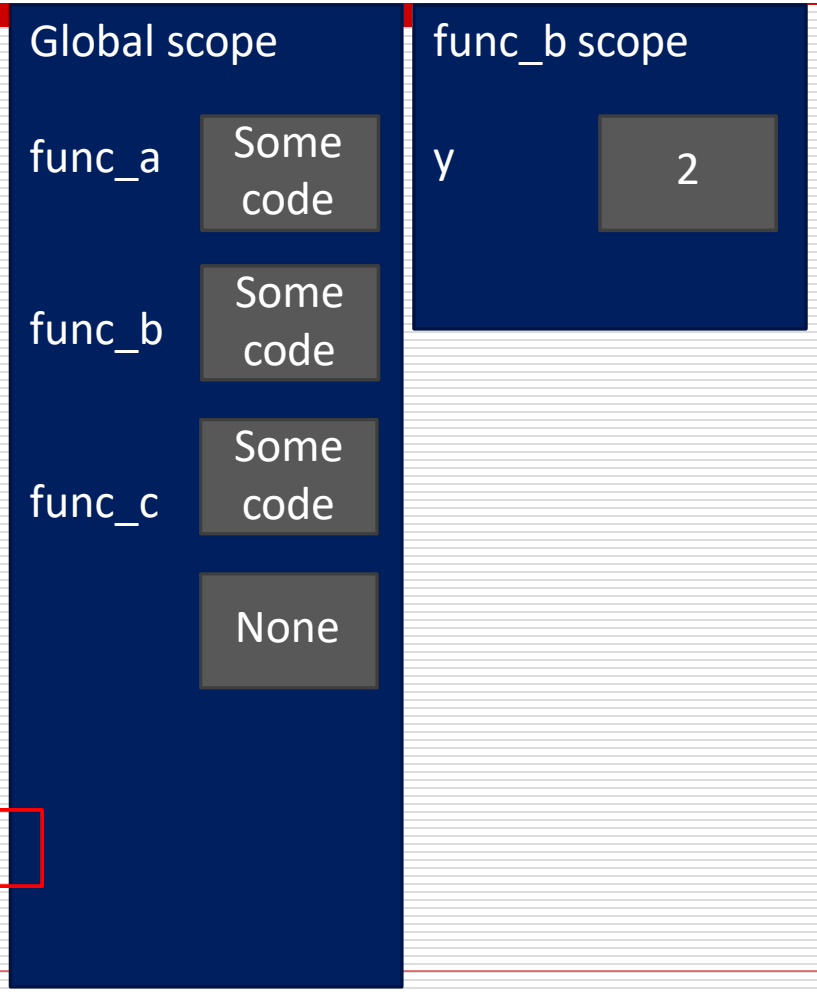
FUNCTIONS AS ARGUMENTS

```
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)
```



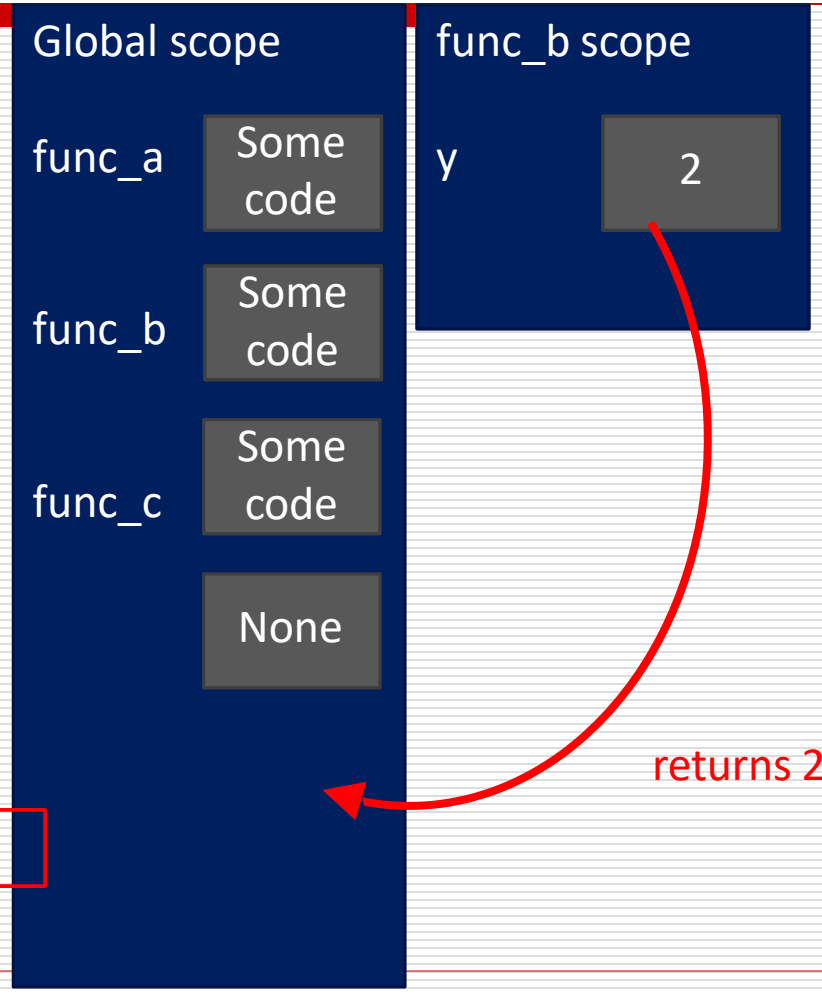
FUNCTIONS AS ARGUMENTS

```
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)
```



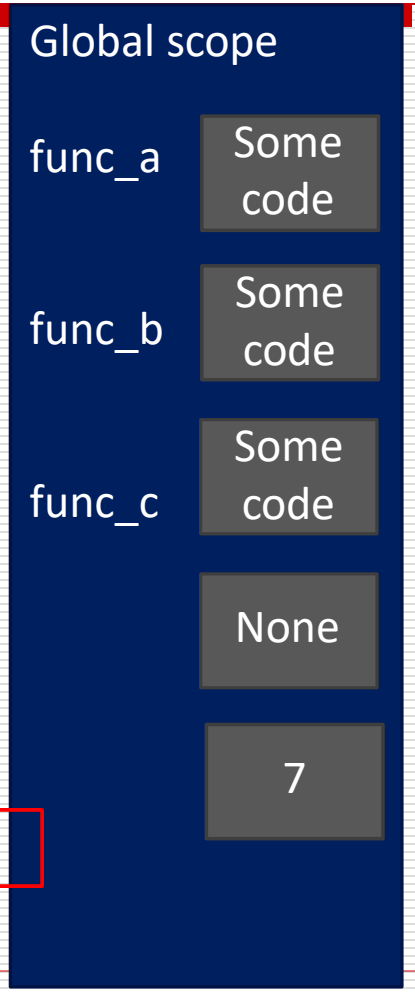
FUNCTIONS AS ARGUMENTS

```
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))
```



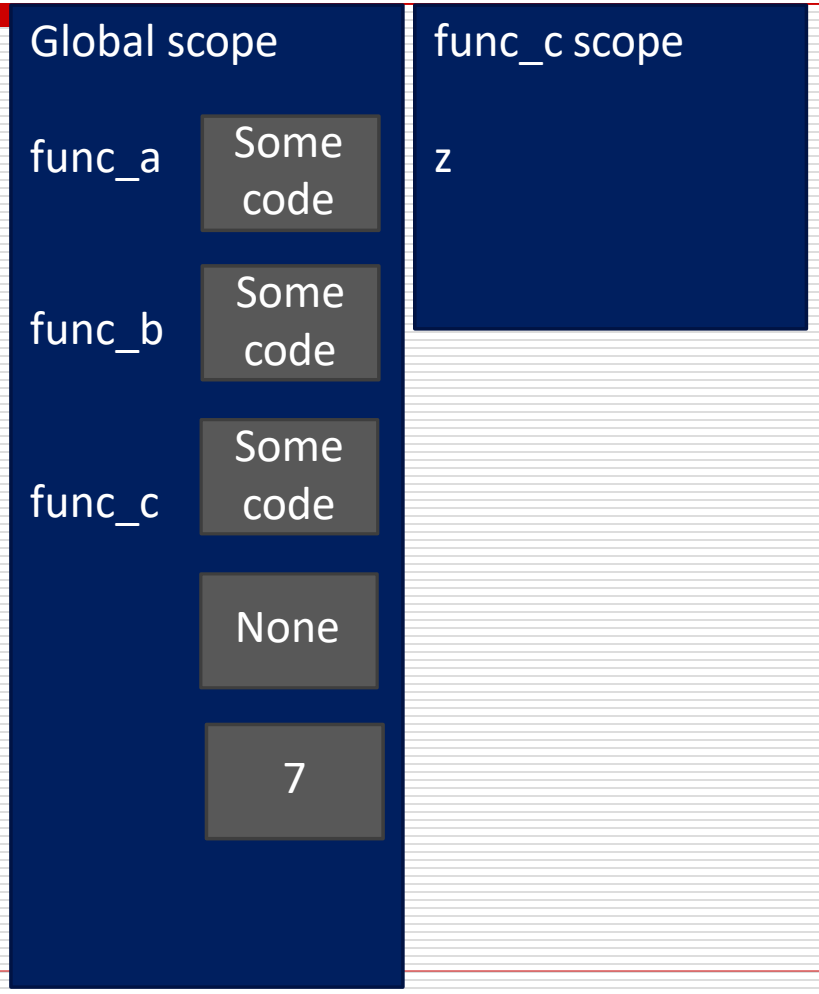
FUNCTIONS AS ARGUMENTS

```
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))
```



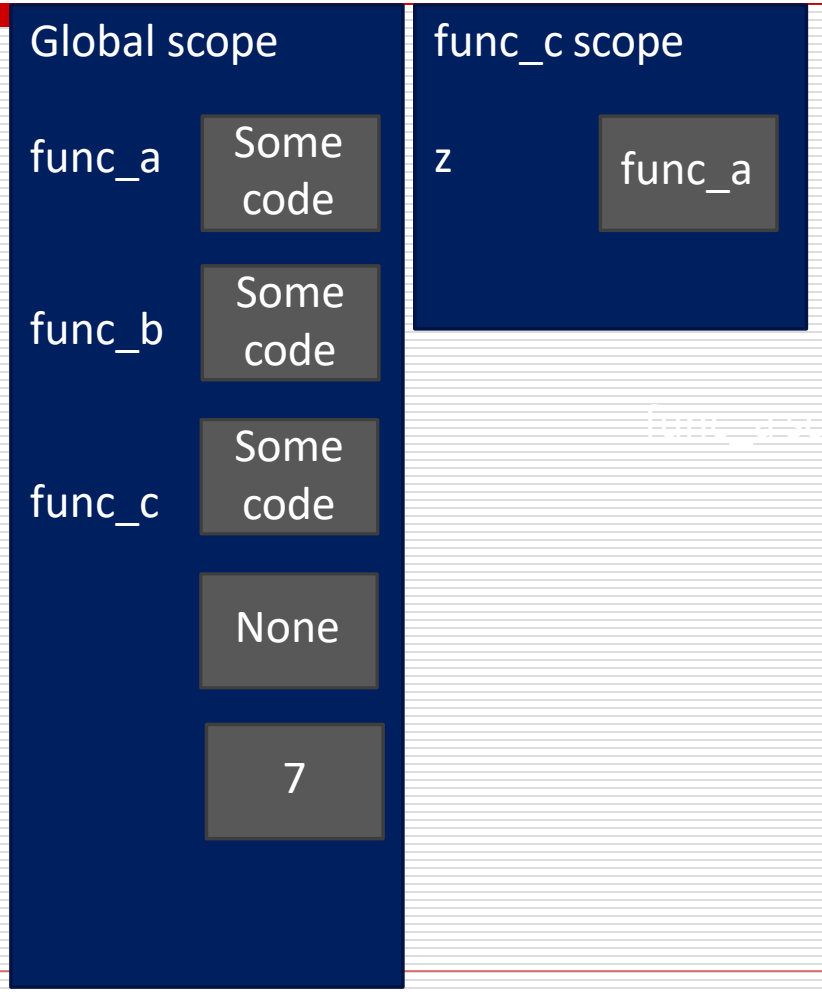
FUNCTIONS AS ARGUMENTS

```
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)
```



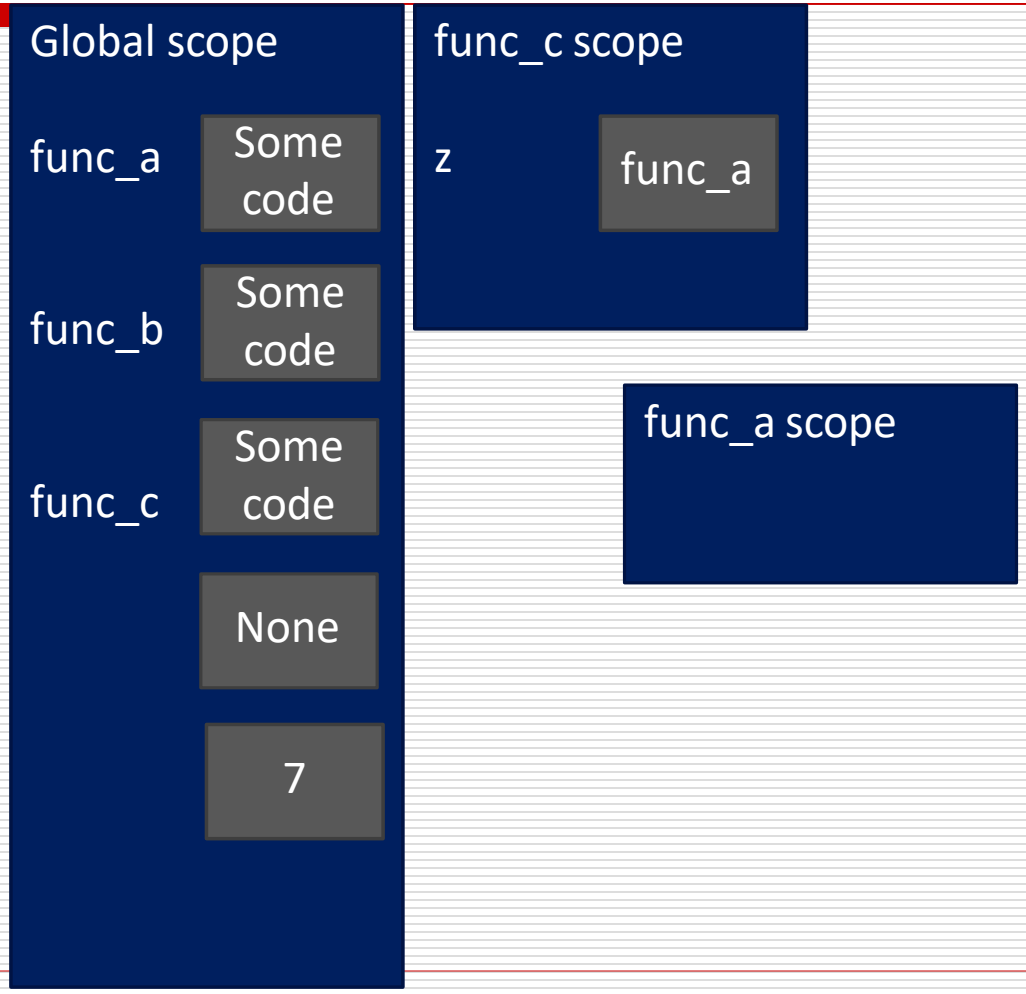
FUNCTIONS AS ARGUMENTS

```
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)
```



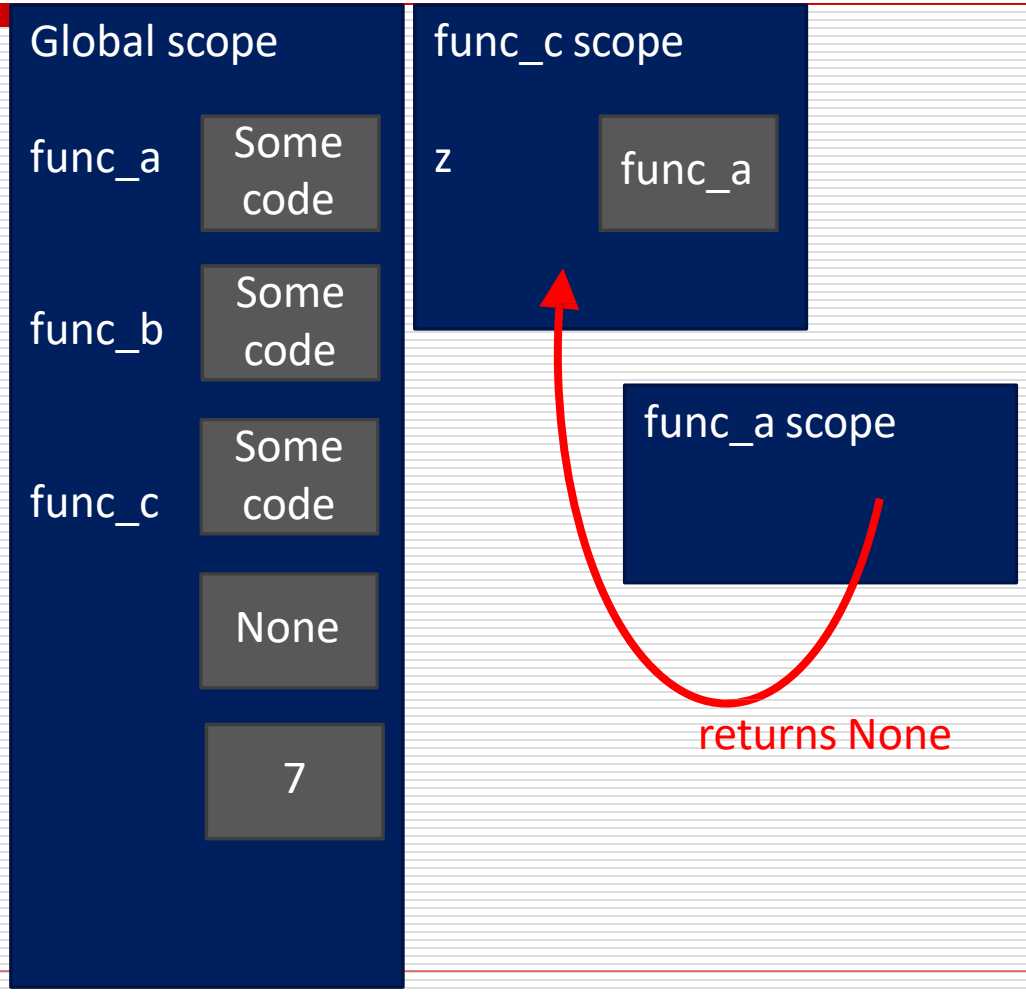
FUNCTIONS AS ARGUMENTS

```
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)
```



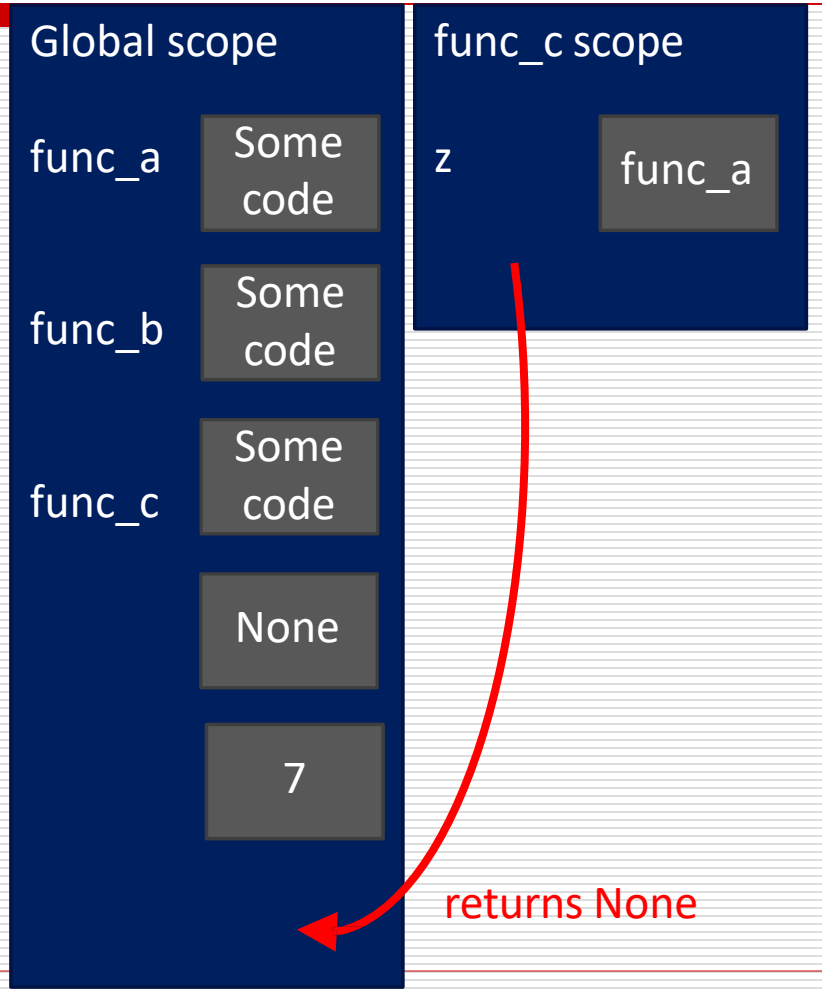
FUNCTIONS AS ARGUMENTS

```
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))
```



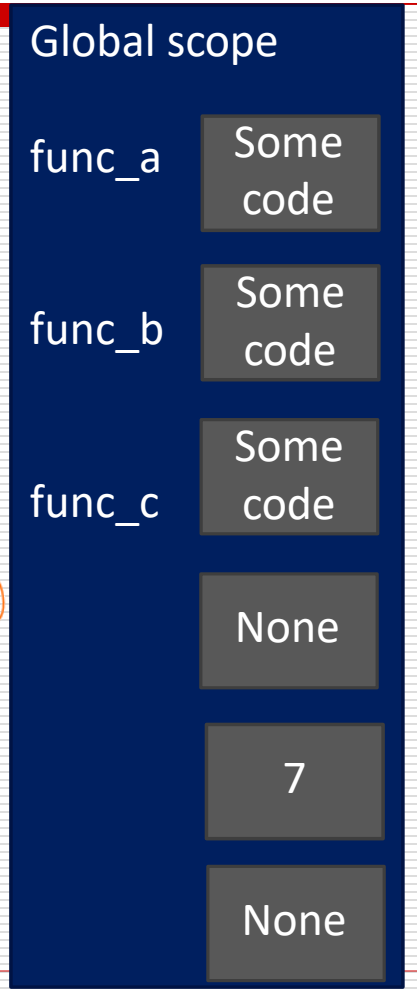
FUNCTIONS AS ARGUMENTS

```
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 -- 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!

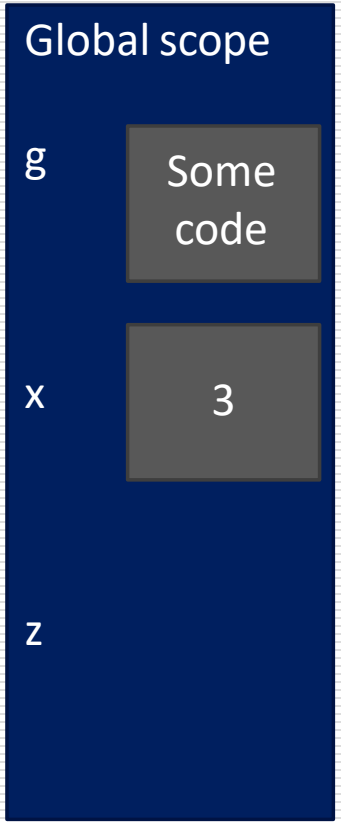
Python Tutor is your best friend to help sort this out!

http://www.pythontutor.com/

SCOPE DETAILS

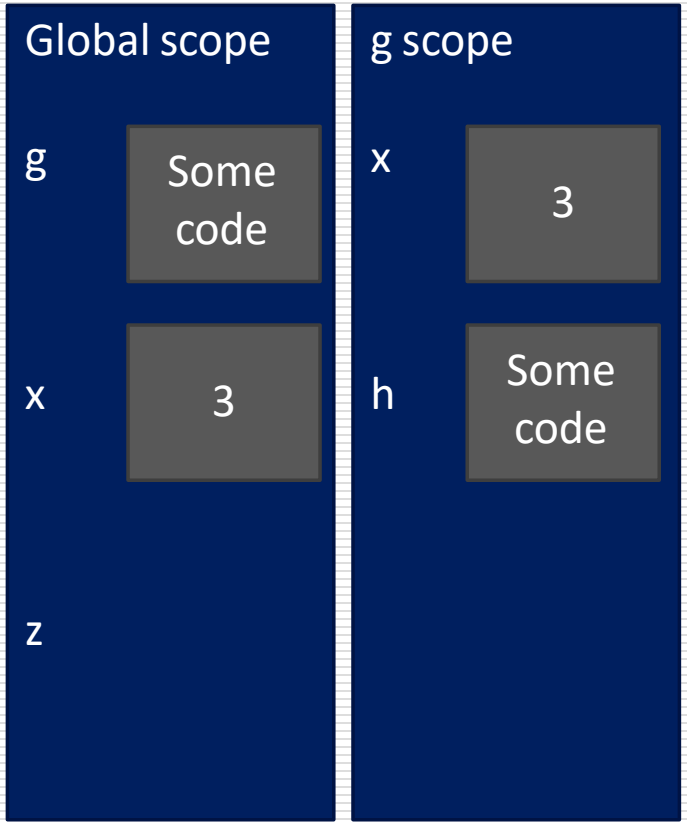
```
def g(x):
    def h():
        x = 'abc'
    x = x + 1
    print('g: x =', x)
    h()
    return x

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

Some code



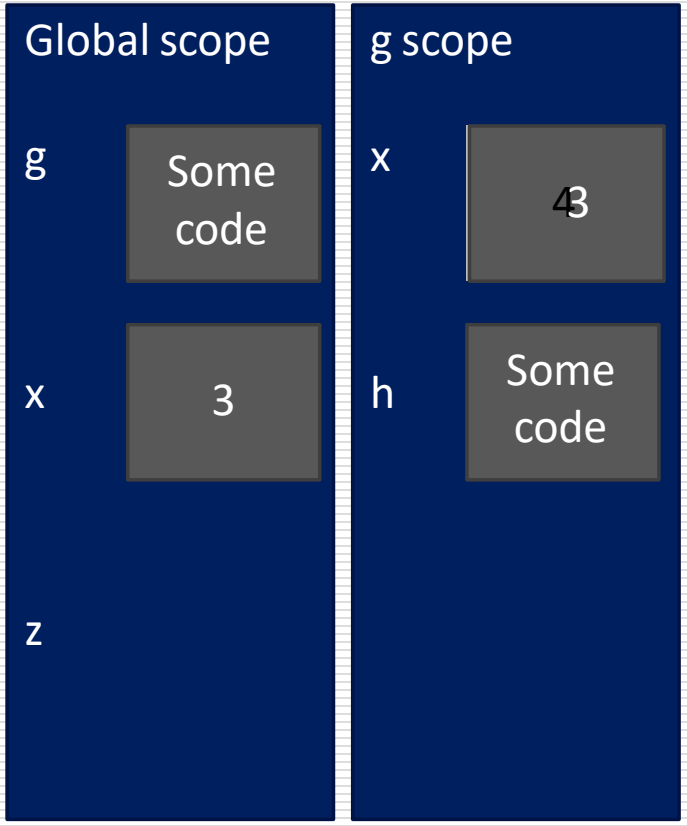
SCOPE DETAILS

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

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



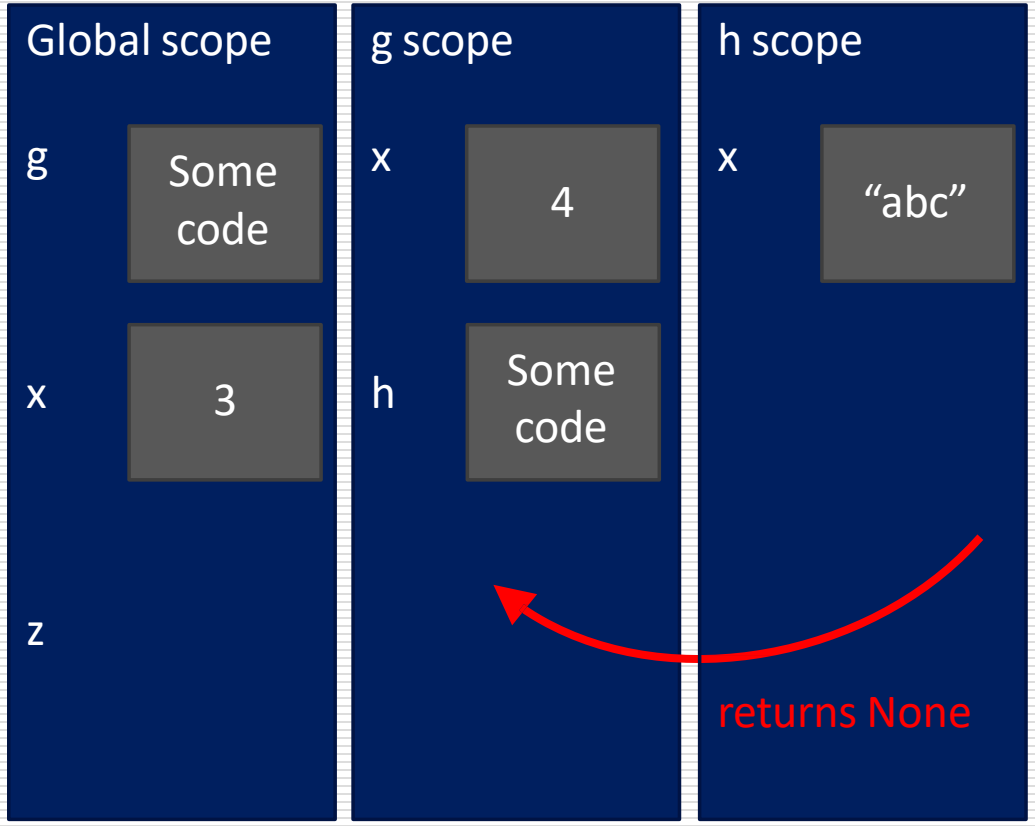
SCOPE DETAILS

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

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



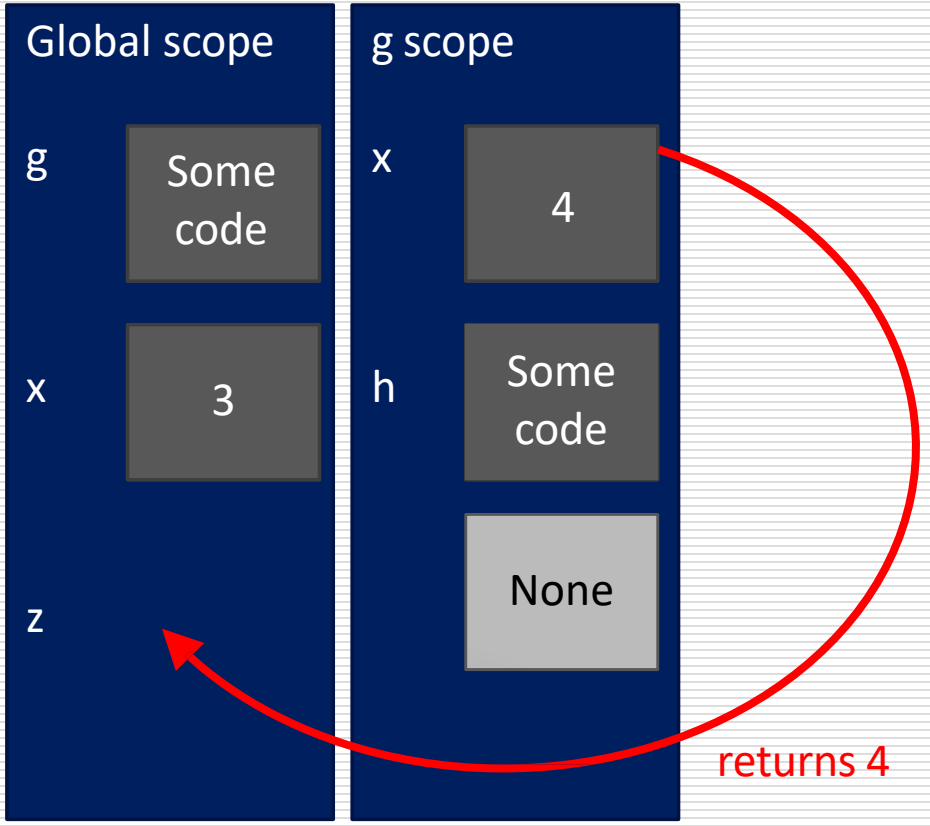
SCOPE DETAILS

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

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



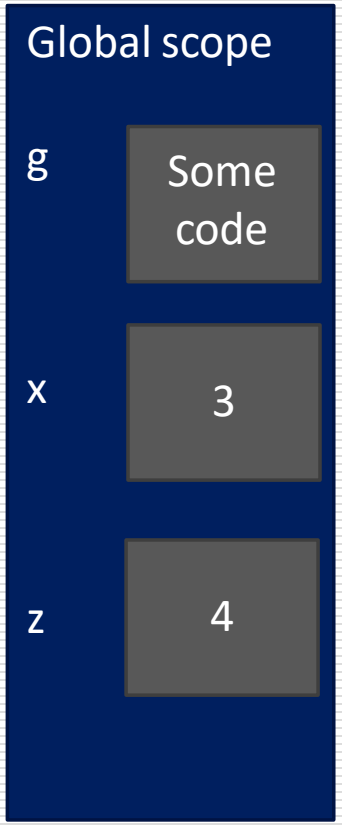
SCOPE DETAILS

```
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)
```



Python Anonymous/Lambda Function

- Anonymous function is a **function** that is defined without a name.
- While normal functions are defined using the **def** keyword, anonymous functions are defined using the **lambda** keyword.
- **Syntax of Lambda Function**
  - **lambda** arguments: expression
- Lambda functions can have any number of arguments but only one expression. The expression is evaluated and returned. The main purpose of anonymous function is just for instant use(i.e for one time usage)
- Sometimes we can pass function as argument to another function. In such cases lambda functions are best choice.
- In Python, we generally use it as an argument to a higher-order function.

Python Anonymous/Lambda Function

lambda

- Lambda function
  - A simple 1-line function
  - Do not use def or return keywords. These are implicit

Python Anonymous/Lambda Function

lambda

```
# double x
def double (x):
    return x * 2
```

```
lambda x: 2 * x
```

Parameter(s) Return

Python Anonymous/Lambda Function

lambda

```
# add x and y

def add (x, y):
    return x + y
```

```
lambda x, y: x + y
```

Python Anonymous/Lambda Function

lambda

```
# max of x, y

def mx(x, y):
    if x > y:
        return x
    else:
        return y
print (mx(8, 5))
```

```
mx = lambda x, y: x if x > y else y

print (mx(8, 5))
```

Python Anonymous/Lambda Function--uses

We can use lambda functions very commonly with filter(), map() and reduce() functions, because these functions expect function as argument.

Syntax :

map(function, sequence)

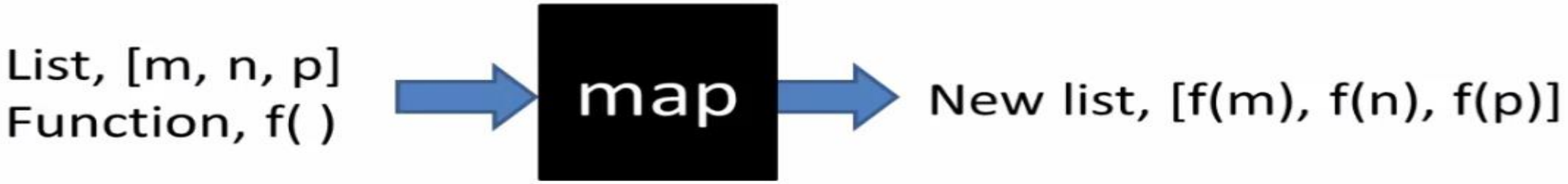
filter(function, sequence)

reduce(function, sequence)

Python Map Function

map

- Apply same function to each element of a sequence
- Return the modified list



Python Anonymous/Lambda-Map Function

map

```
# prints [16, 9, 4, 1]

def square (lst1):
    lst2 = []
    for num in lst1:
        lst2.append(num ** 2)
    return lst2

print square([4,3,2,1])
```

```
n = [4, 3, 2, 1]
print (list(map(lambda x: x**2, n)))
```

Function — List

Python Anonymous/Lambda-Map Function

map

```
# prints [16, 9, 4, 1]

def square (lst1):
    lst2 = []
    for num in lst1:
        lst2.append(num ** 2)
    return lst2

print square([4,3,2,1])
```

```
n = [4, 3, 2, 1]
print (list(map(lambda x: x**2, n)))
```

Function — List

Needn't necessarily use a lambda function:

```
print (list(map(square, n)))
```

Python Anonymous/Lambda-MapFunction

map

```
# prints [16, 9, 4, 1]

def square (lst1):
    lst2 = []
    for num in lst1:
        lst2.append(num ** 2)
    return lst2

print square([4,3,2,1])
```

```
n = [4, 3, 2, 1]
print (list(map(lambda x: x**2, n)))
```

Function List

Needn't necessarily use a lambda function:

```
print (list(map(square, n)))
```

List comprehension solution:

```
print ([x**2 for x in n])
```

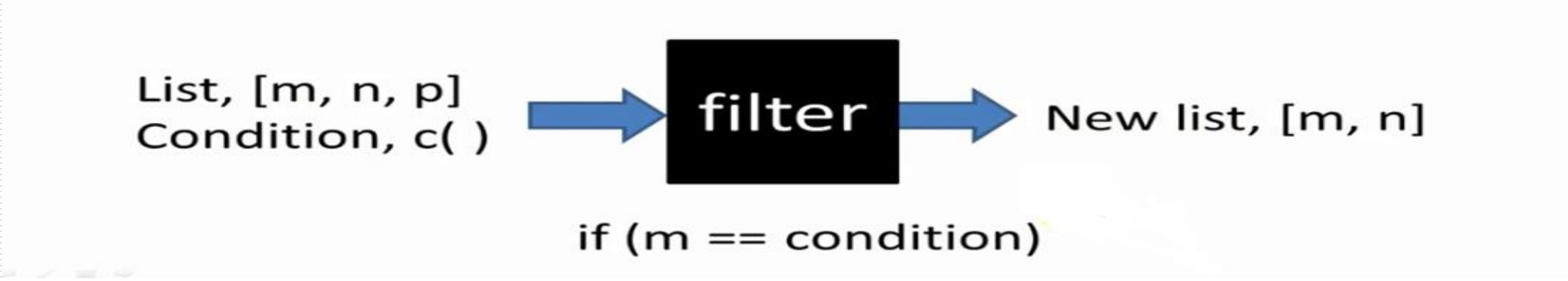
Python Anonymous/Lambda-Map Function

 # Program to double each item in a list using map()

```
my_list = [1, 5, 4, 6, 8, 11, 3, 12]
new_list = list(map(lambda x: x * 2 , my_list))

# Output: [2, 10, 8, 12, 16, 22, 6, 24]
print(new_list)
```

Python Anonymous/Lambda-Filter Function

- The filter() function in Python takes two arguments:
  - Function
  - Sequence as arguments.
- The function is called with all the items in the list and a new list is returned which contains items for which the function evaluates to True.



Python Anonymous/Lambda-Filter Function

filter

```
# prints [4, 3]
```

```
def over_two (lst1):
    lst2 = [x for x in lst1 if x>2]
    return lst2

print over_two([4,3,2,1])
```

```
n = [4, 3, 2, 1]

print (list(filter(lambda x: x>2, n)))
```

Condition List

Python Anonymous/Lambda-Filter Function

filter

```
# prints [4, 3]

def over_two (lst1):
    lst2 = [x for x in lst1 if x>2]
    return lst2

print over_two([4,3,2,1])
```

```
n = [4, 3, 2, 1]
print (list(filter(lambda x: x>2, n)))
```

Condition — List

List comprehension solution:

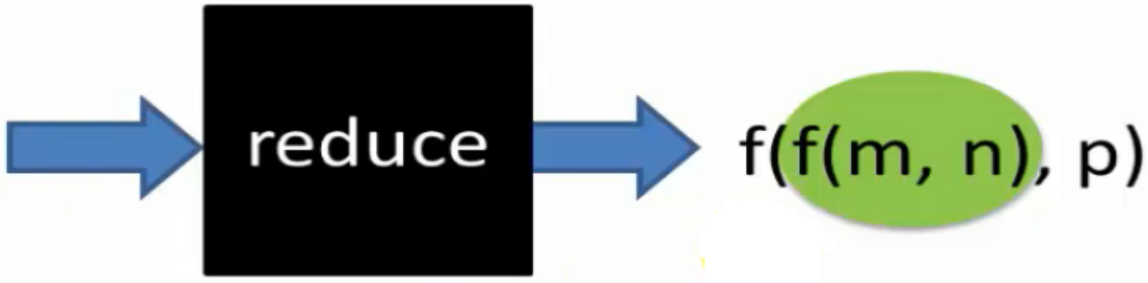
```
print ([x for x in n if x>2])
```

Python Anonymous/Lambda-Reduce Function

reduce

- Applies same operation to items of a sequence
- Uses result of operation as first param of next operation
- Returns an item, not a list

List, [m, n, p]
Function, f( )



Python Anonymous/Lambda-Reduce Function

reduce

```
# prints 24

def mult (lst1):
    prod = lst1[0]
    for i in range(1, len(lst1)):
        prod *= lst1[i]
    return prod

print mult([4,3,2,1])
```

```
n = [4, 3, 2, 1]
print (reduce(lambda x,y: x*y, n))
```

Function List

4 * 3 = 12

12 * 2 = 24

24 * 1 = 24

Python Anonymous/Lambda-Reduce Function

```
# python code to demonstrate working of reduce()

# importing functools for reduce()
import functools

# initializing list
lis = [ 1 , 3, 5, 6, 2, ]

# using reduce to compute sum of list
print ("The sum of the list elements is : ",end="")
print (functools.reduce(lambda a,b : a+b,lis))

# using reduce to compute maximum element from list
print ("The maximum element of the list is : ",end="")
print (functools.reduce(lambda a,b : a if a > b else b,lis))
```

Everything is an Object:

- In Python every thing is treated as object.
- Even functions also internally treated as objects only.

```
1)  def f1():
2)
2)     print("Hello")
3) print(f1)
4) print(id(f1))
```

Output:

```
<function f1 at 0x00419618>
4298264
```

Function Aliasing:

For the existing function we can give another name, which is nothing but function aliasing.

```
def wish(name):
    print("Good Morning:",name)

greeting=wish
print(id(wish))
print(id(greeting))
greeting('DPSharma')
wish('DPSharma')
```

Output:

```
4429336
4429336
Good Morning: DPSharma
Good Morning: DPSharma
```

Function Aliasing:

Note:

- In the example (last slide) only one function is available but we can call that function by using either wish name or greeting name.
- If we delete one name still we can access that function by using alias name.

```
1)  def wish(name):
2)      print("Good Morning:",name)
3)
4)  greeting=wish
5)
6)  greeting('DPSharma')
7)  wish('DPSharma')
8)
9)  del wish
10) #wish('Anaika') → NameError: name 'wish' is not defined
11) greeting('Anaika')
```

Output:

```
Good Morning: DPSharma
Good Morning: DPSharma
Good Morning: Anaika
```

Nested Functions:

- We can declare a function inside another function, such type of functions are called Nested functions.

```
1)  def outer():
2)      print("outer function started")
3)      def inner():
4)          print("inner function execution")
5)      print("outer function calling inner function")
6)      inner()
7)  outer()
8)  #inner() → NameError: name 'inner' is not defined
```

Output:

```
outer function started
outer function calling inner function
inner function execution
```

Note:
In the above example inner() function is local to outer() function and hence it is not possible to call directly from outside of outer() function.

function can return another function

```
1)  def outer():
2)      print("outer function started")
3)      def inner():
4)          print("inner function execution")
5)      print("outer function returning inner function")
6)      return inner
7)  f1=outer()
8)  f1()
9)  f1()
10) f1()
```

Output:

```
outer function started
outer function returning inner function
inner function execution
inner function execution
inner function execution
```