Topic 4

Functions and Modules

Murdoch
UNIVERSITY

LAST WEEK

Murdoch UNIVERSITY

- Define multi-line string
- String index: positive index and negative index
- Slicing a string
- Strings are immutable, string identity
- Loop over the sequence of characters in a string,
- Formatted strings
- Strings as composite objects
- String methods
- F-strings
- Escape character

TODAY

Murdoch UNIVERSITY

- structuring programs and hiding details (abstraction)
- divide-and-conquer strategy for solving large problems
- defining a function
- calling or invoking a function
- `return` statement
- scope of a function
- global variable and its scope
- local variable and its scope
- nested function
- modules

HOW DO WE WRITE CODE?

Murdoch UNIVERSITY

- so far…
  - covered language mechanisms
  - know how to write different files for each computation
  - each file is some piece of code
  - each code is a sequence of instructions
- problems with this approach
  - easy for small-scale problems
  - messy for larger problems
  - hard to keep track of details
  - how do you know the right info is supplied to the right part of code

GOOD PROGRAMMING

Murdoch UNIVERSITY

- more code not necessarily a good thing
- measure good programmers by the amount of functionality
- introduce **functions**
- mechanism to achieve **decomposition** and **abstraction**

EXAMPLE – PROJECTOR

Murdoch UNIVERSITY

- a projector is a black box
- don't know how it works
- know the interface: input/output
- connect the electronic cable to it that can send the input signals to the black box
- the black box somehow converts the electric signals from the input to image and project the image to the wall
- ABSTRACTION IDEA: do not need to know how projector works to use it

EXAMPLE – PROJECTOR

EXAMPLE – PROJECTOR

Murdoch UNIVERSITY

- projecting large image for Olympics 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: divide a large task into several smaller tasks, solve all small tasks to achieve an end goal
- This strategy is also called "divide-and-conquer"

Murdoch
UNIVERSITY

APPLY THESE CONCEPTS

TO PROGRAMMING!

CREATE STRUCTURE with DECOMPOSITION

Murdoch UNIVERSITY

- in projector example, separate devices
- in programming, divide code into **several chunks**
  - are **self-contained**
  - used to **break up** code
  - intended to be **reusable**
  - keep code **organized**
  - **keep code coherent**
- in this lecture, achieve decomposition with **functions** and **modules**
- in later topics, achieve decomposition with **classes** and **objects**

SUPRESS DETAILS with ABSTRACTION

Murdoch UNIVERSITY

- in projector example, instructions for how to use it are sufficient, no need to know how to build one
- in programming, think of a piece of code as a **black box**
  - cannot see details
  - do not need to see details
  - do not want to see details
  - hide tedious coding details
- achieve abstraction with **function specifications** or **docstrings**

FUNCTIONS

Murdoch UNIVERSITY

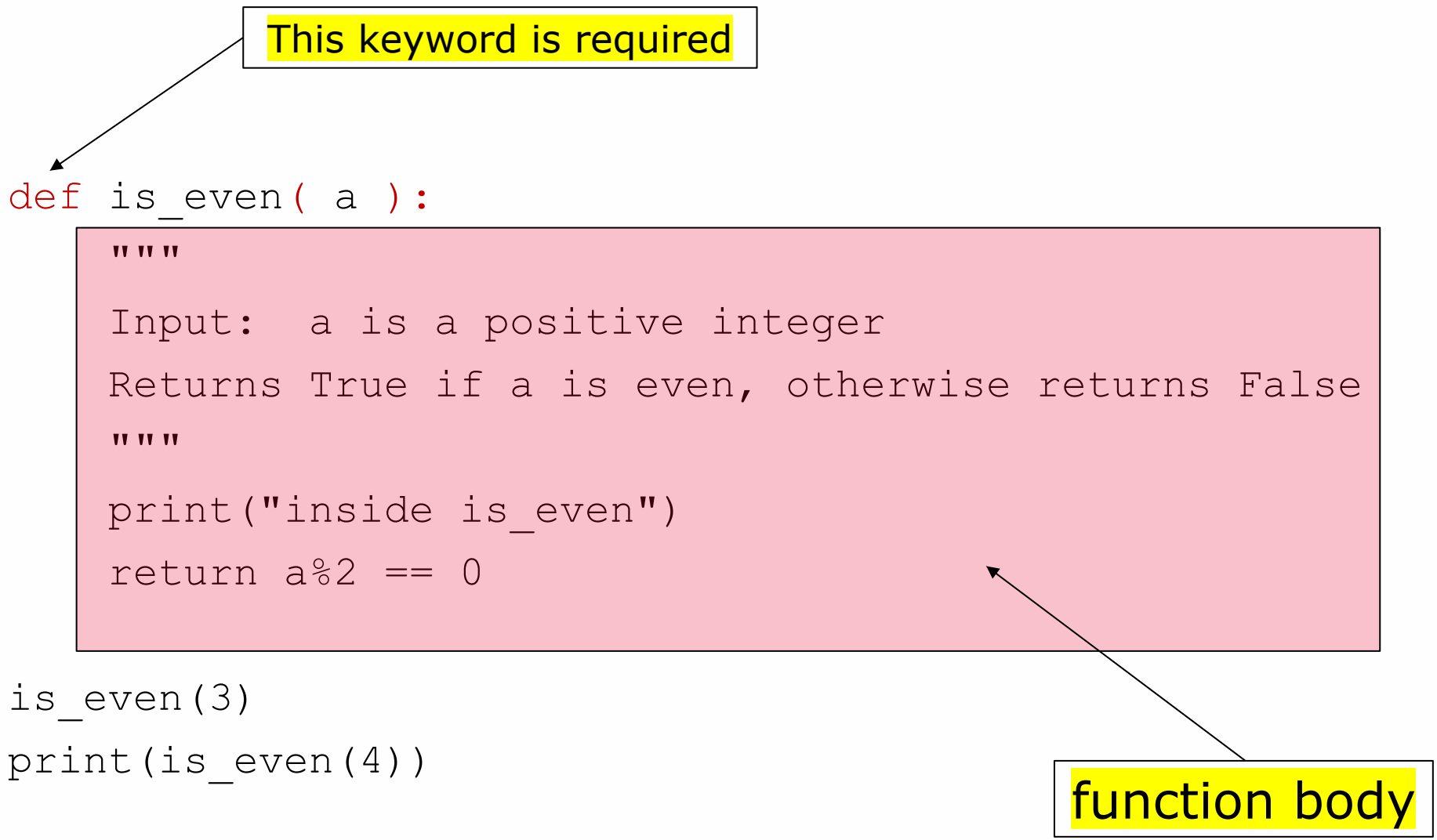
- write reusable pieces/chunks of code, called **functions**
- functions are not run in a program until they are "**called**" or "**invoked**" in a program
- we have already used several functions before, such as
  - `print(…)`
  - `input(…)`
  - `len(…)`
  - `int(…)`
  - `range(…)`
  - `dir(…)`
  - `help(…)`
- These functions are language built-in functions.

FUNCTIONS

Murdoch UNIVERSITY

- in addition to those built-in functions, we can define and use our own functions
- to define a function, we need the following:
  - a **function name**
  - a list of **parameters** (0 or more)
  - a **docstring** (optional but recommended)
  - a **function body**
  - **returns** something (optional)

DEFINE and CALL/INVOKE A FUNCTION



DEFINE and CALL/INVOKE A FUNCTION

Murdoch UNIVERSITY

- A function is defined with keyword `def`, followed by the *function name*. `is_even` is the function name.
- The function name is followed by the *formal parameter list* inside the parentheses : `(a)`. The formal parameter list can be empty.
- The optional string at the beginning of the function body is the *docstring*, it explains what the function does
- The optional `return` statement returns the execution flow back to the caller. It can also provide a value (optional) to the caller of the function: `return a % 2 == 0`
- The group of statements inside the *function body* can be executed (called or invoked) using the function name and a list of *actual arguments* inside paratheses, such as

```
is_even(3)
```

- The function call can also be used where a value is used, eg,

```
print( is_even(3) )
```

VARIABLE SCOPE

Murdoch UNIVERSITY

- A global variable is the one defined outside any function.
- A local variable is the one defined inside a function
- A global is accessible everywhere, including inside a function unless there is a local variable of the same name in that function.
- A local variable is only accessible inside the function in which it is defined.

VARIABLE SCOPE

- **a formal parameter** gets bound to the value of **an actual parameter** when function is called
- an actual parameter is also called **an argument**
- a new **scope** created when enter a function
- a formal parameter acts like a local variable

formal parameter

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

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

actual parameter

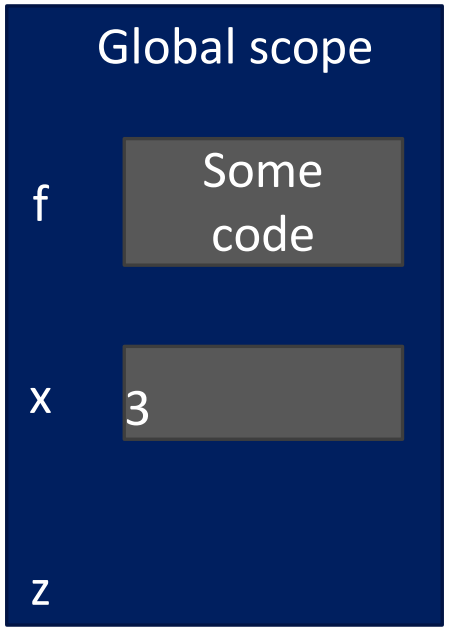
Function definition

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

VARIABLE SCOPE

Murdoch UNIVERSITY

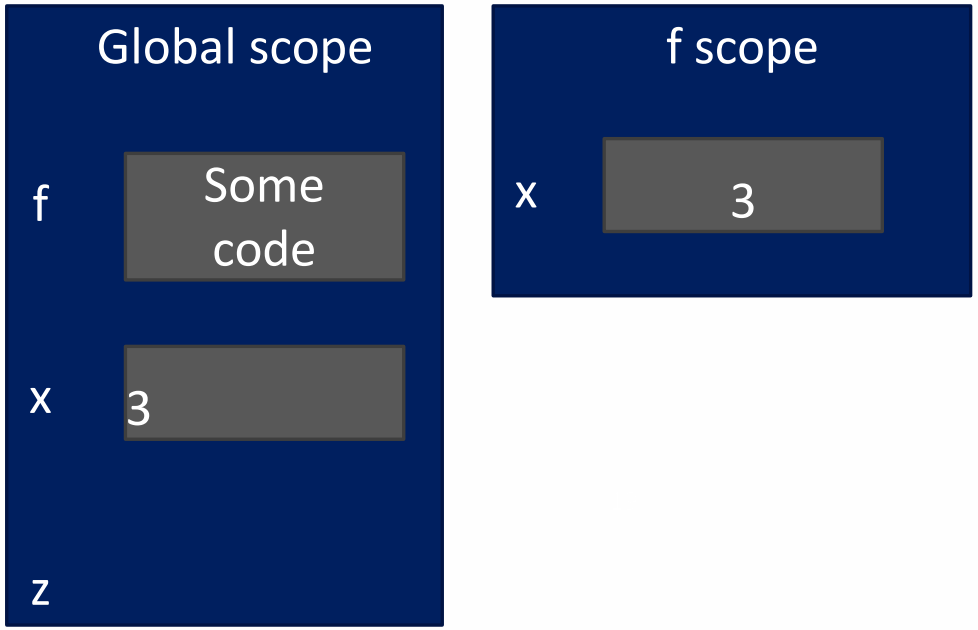
```
def f( x ):
    x = x + 1
    print('in f(x): x =', x)
    return x

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



VARIABLE SCOPE

Murdoch UNIVERSITY

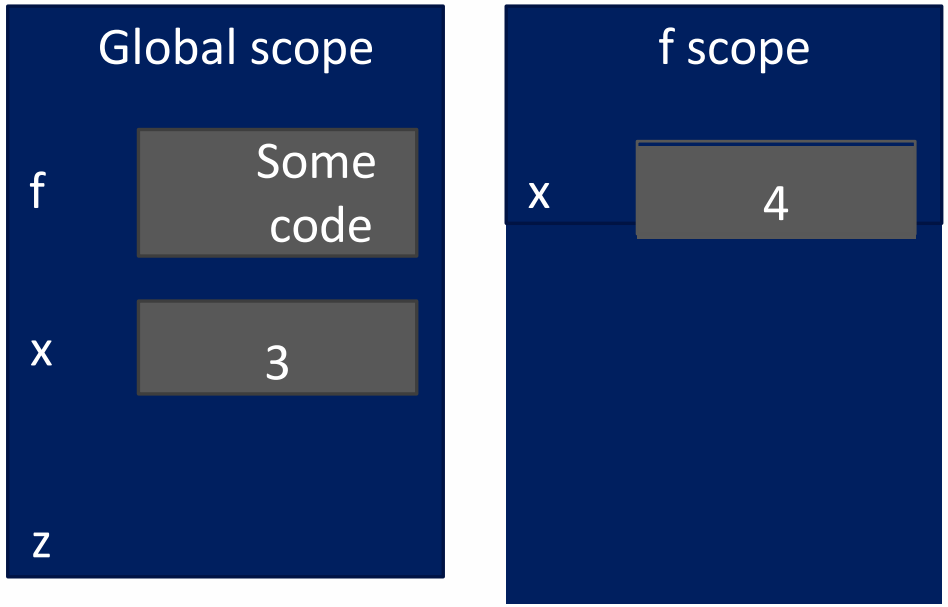
```
def f( x ):
    x = x + 1
    print('in f(x): x =', x)
    return x

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



VARIABLE SCOPE

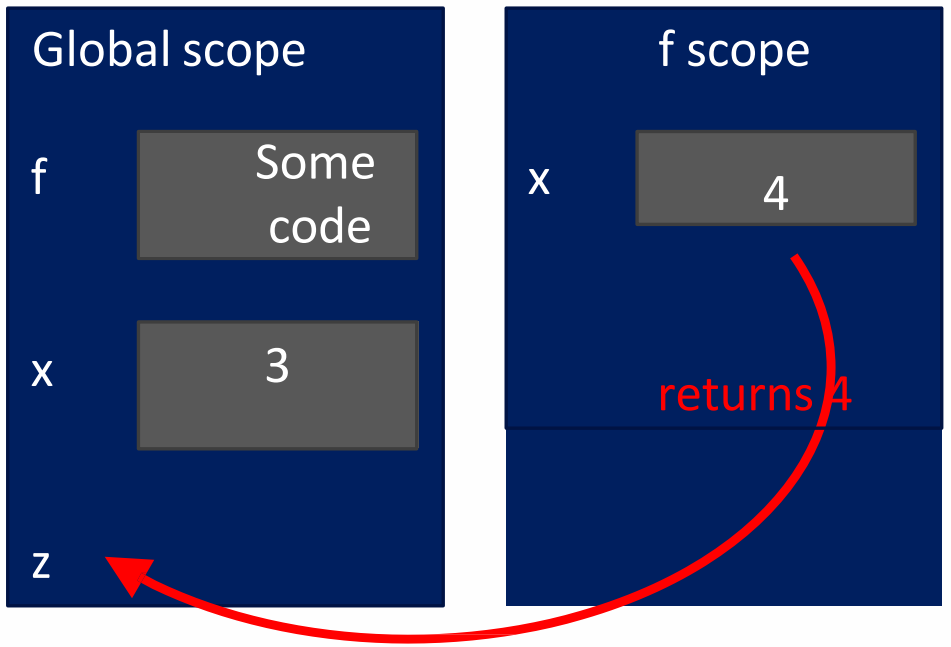
Murdoch UNIVERSITY

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

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



VARIABLE SCOPE

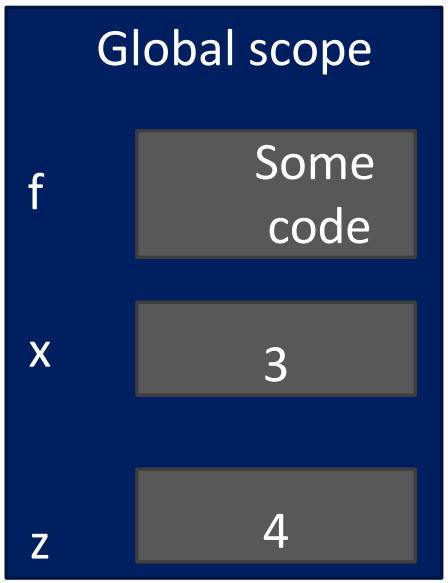
```
def f( x ):
    x = x + 1
    print('in f(x): x =', x)
    return x

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



SCOPE OF VARIABLE AND FUNCTION

Murdoch UNIVERSITY

```
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

Murdoch UNIVERSITY

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


is_even(3)
print(is_even(4))
```

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

MORE ON `return` STATEMENT

Murdoch UNIVERSITY

- return only has meaning **inside** a function
- only one return executed inside a function
- when a return statement is executed, the execution flow goes back to the caller
  - The next statement would be the one following the function call.
- code following the return statement inside the function will not be executed
- has a value associated with it, given to function caller. If no value is given in the return statement, value `None` is returned.

FUNCTIONS AS ARGUMENTS

Murdoch UNIVERSITY

- 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

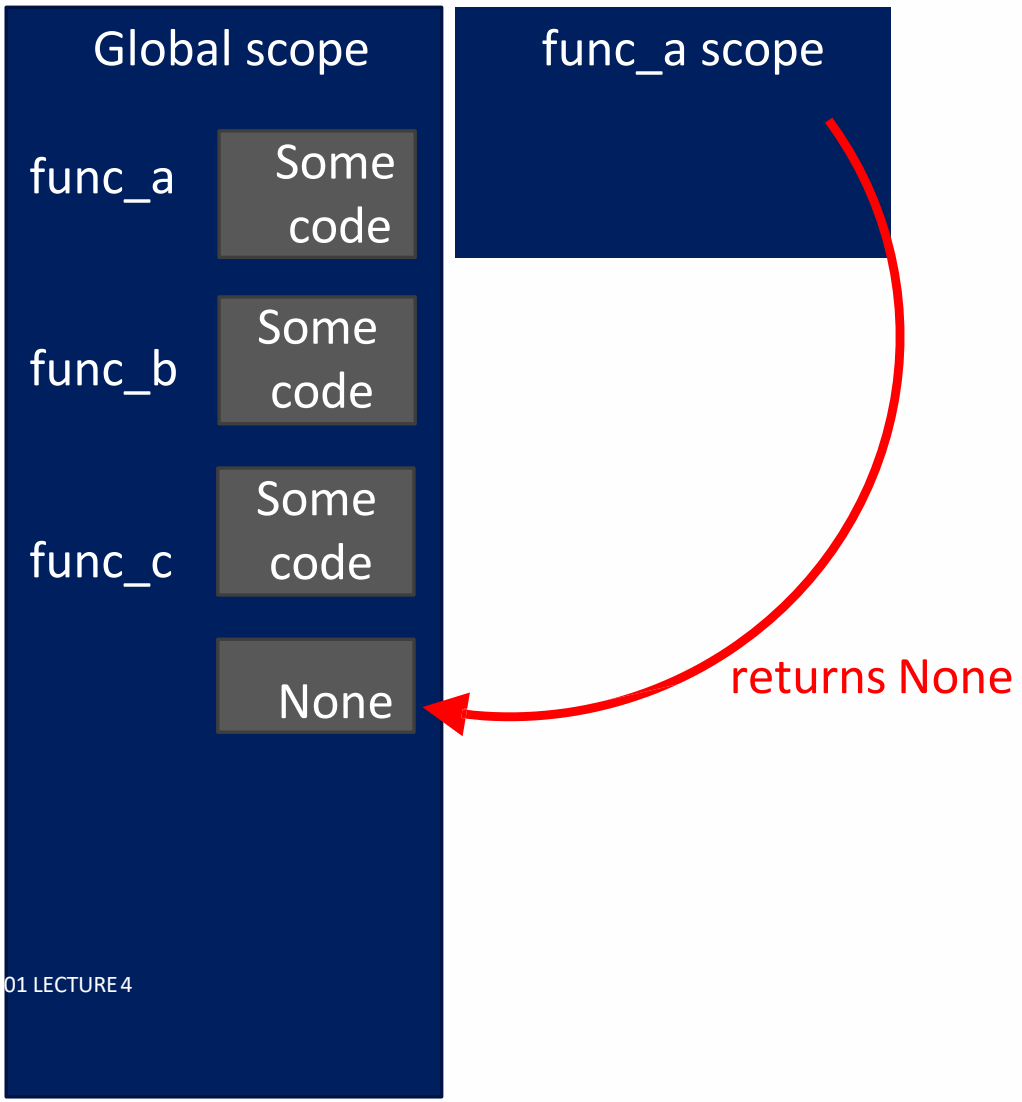
call `func_b`, takes one parameter

call `func_c` , takes one parameter, another function

CALL A FUNCTION WITHOUT ARGUMENTS

Murdoch UNIVERSITY

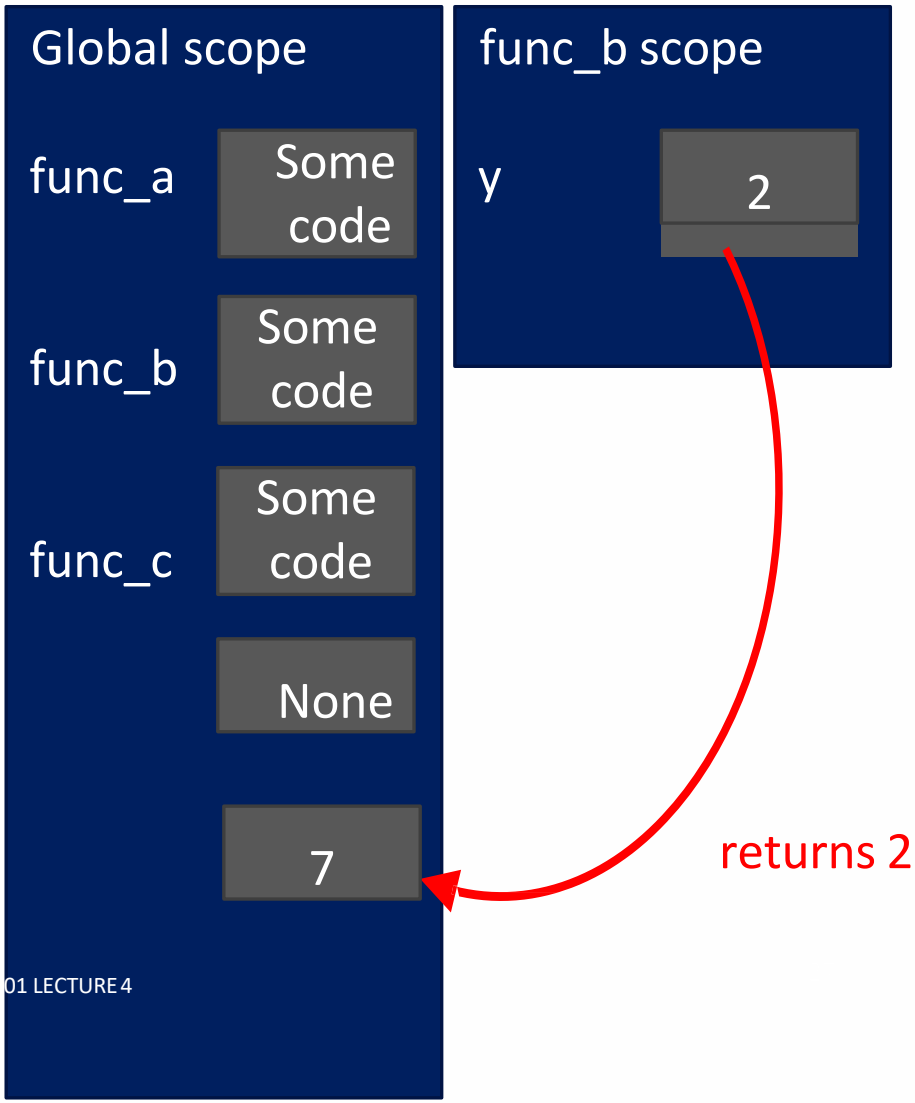
```
def func_a():
    print('inside func_a')
def func_b(y):
    print('inside func_b')
    return y
def func_c(z):
    print('inside func_c')
    return z()

print(func_a())

print(5 + func_b(2)

print(func_c(func_a))
```



CALL A FUNCTIONS WITH AN ARGUMENT

Murdoch UNIVERSITY

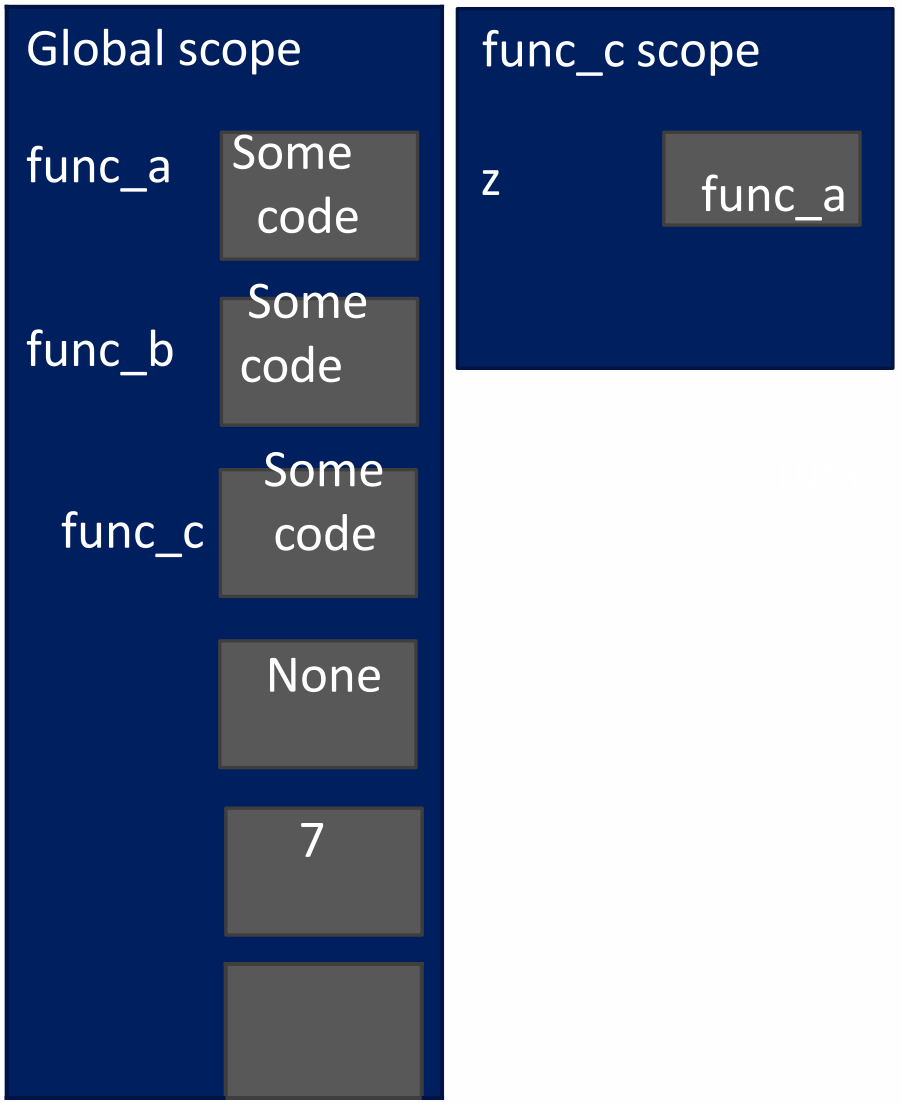
```
def func_a():
    print('inside func_a')

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 A FUNCTION WIT AN ARGUMENT THAT IS A FUNCTION

Murdoch UNIVERSITY

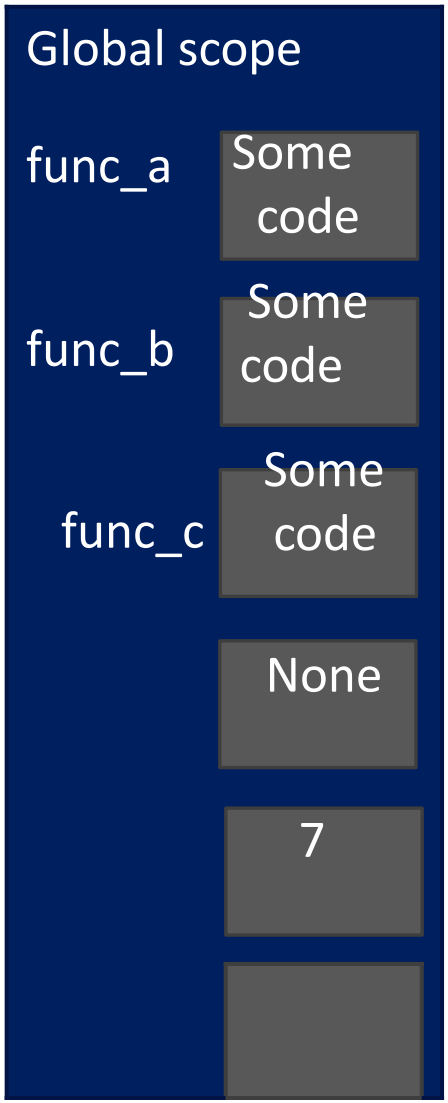
```
def func_a():
    print('inside func_a')

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))
```

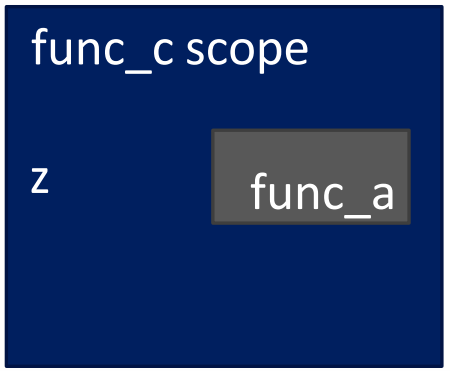


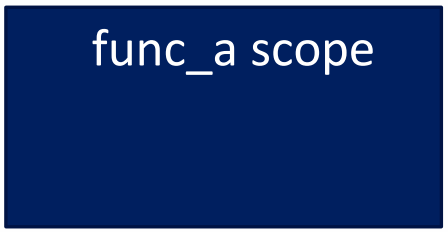
CALL A FUNCTION WIT AN ARGUMENT THAT IS A FUNCTION

Murdoch UNIVERSITY

```
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))
```







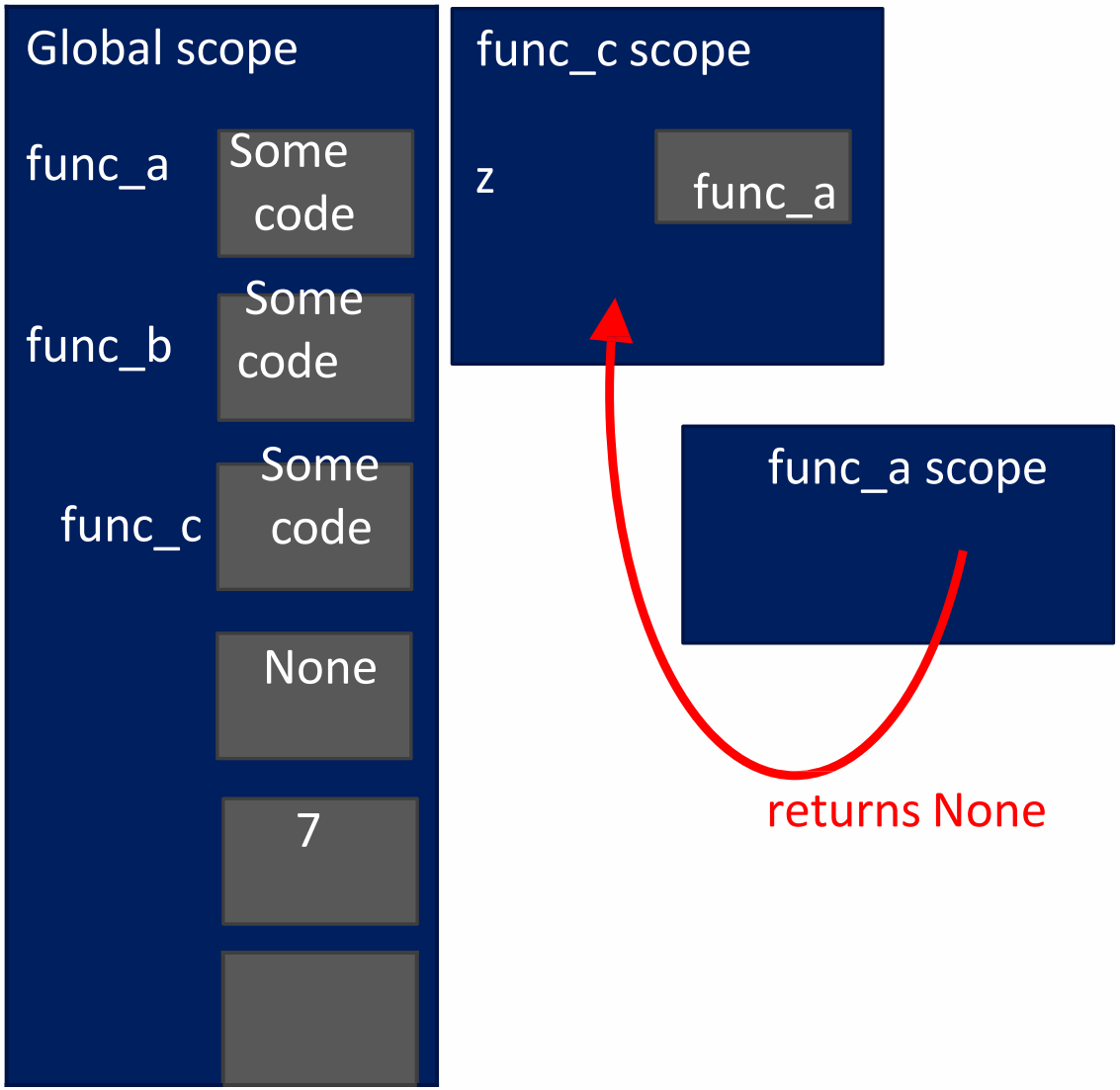
returns None

returns None

CALL A FUNCTION WIT AN ARGUMENT THAT IS A FUNCTION

Murdoch UNIVERSITY

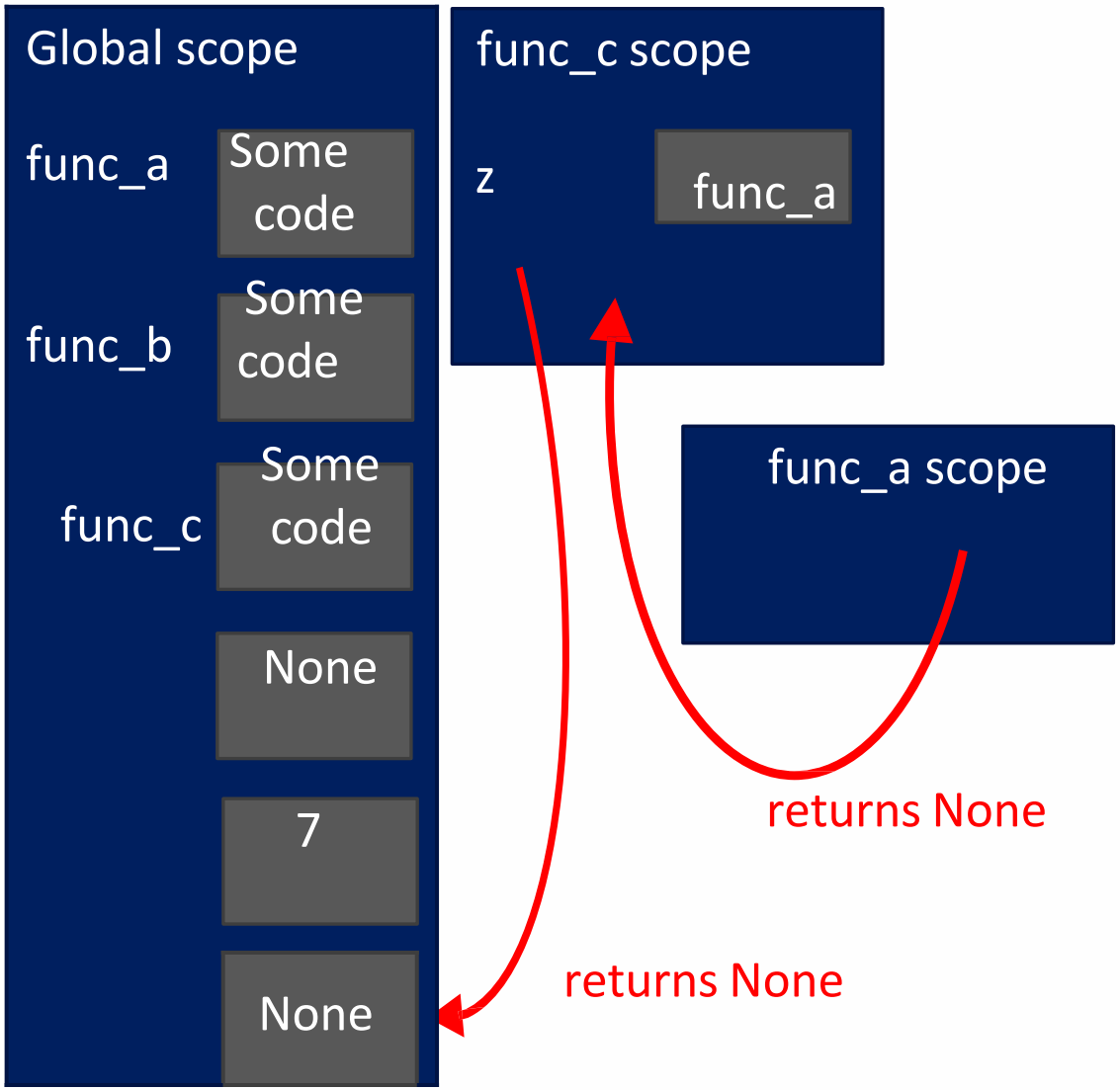
```
def func_a():
    print('inside func_a')

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 A FUNCTION WIT AN ARGUMENT THAT IS A FUNCTION

Murdoch UNIVERSITY

```
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

Murdoch UNIVERSITY

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

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

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

x is re-defined in scope of f

different x objects

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

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

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

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

SCOPE EXAMPLE

Murdoch UNIVERSITY

- 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
    y = x + y
    print(x)
    print(y)

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

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

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 = x + 1

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

SCOPE EXAMPLE

Murdoch UNIVERSITY

- 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
    y = x + y
    print(x)
    print(y)

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

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

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

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

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

UnboundLocalError: local variable 'x' referenced before assignment

SCOPE EXAMPLE

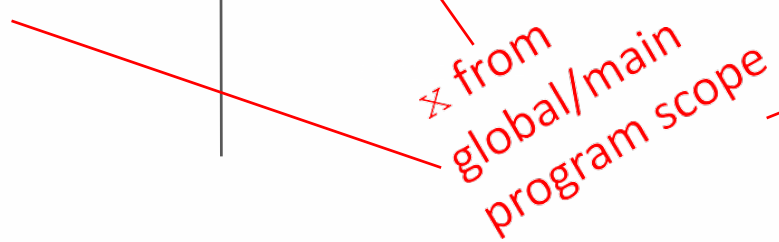
Murdoch UNIVERSITY

- 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
    y = x + y
    print(x)
    print(y)

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

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

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

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

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



ACCESS GLOBAL VARIABLES INSIDE FUNCTIONS

Murdoch UNIVERSITY

- As the previous examples show, you can use the value of a global variable inside a function, eg, use global variable `x` inside function `g`
- However, you cannot change the value of a global variable variable by assigning a new value inside a function.
- assign a value to a global variable inside a function, as shown in function `f`.
  - when you assign a value to `x`, you are effectively defining a new local variable `x` inside function `f`.
  - After that, there will be a global variable x outside the function and a new variable inside the function

ACCESS GLOBAL VARIABLES INSIDE FUNCTIONS

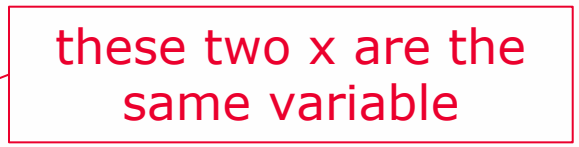
Murdoch UNIVERSITY

- Why the statement x = x + 1 inside the function produces an error?
  - because we try to assign a value to x, that means x on the left is *a new local variable*
  - But to produce a new value, x must have a value. Only the global variable x has a value at this moment, so to do so require *the x to be the global variable*.
  - This leads to a conflict, ie, one name for two different variable in the same scope!
- How do you change the value of a global variable inside a function?
- You declare the variable as `global`, then its scope is global even if it is declared inside a function!

ACCESS GLOBAL VARIABLES INSIDE FUNCTIONS

Murdoch UNIVERSITY

```
def f(a):
    global x
    x += a
    return x

x = 10
print("f(2) = %d" %f(2))
print("x = %d" %x)
```



EXAMPLE: NESTED FUNCTION

Murdoch UNIVERSITY

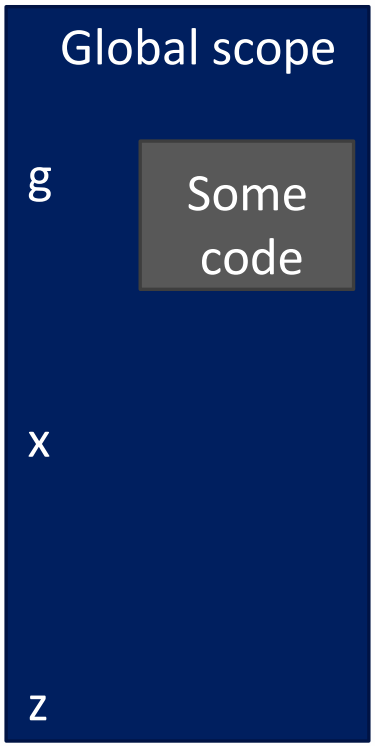
```
def g(x):
    def h():
        x = 'abc'
    x = x + 1
    print('g: x =', x)
    h()
    return x

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

EXAMPLE: NESTED FUNCTION

Murdoch UNIVERSITY

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

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

Some code

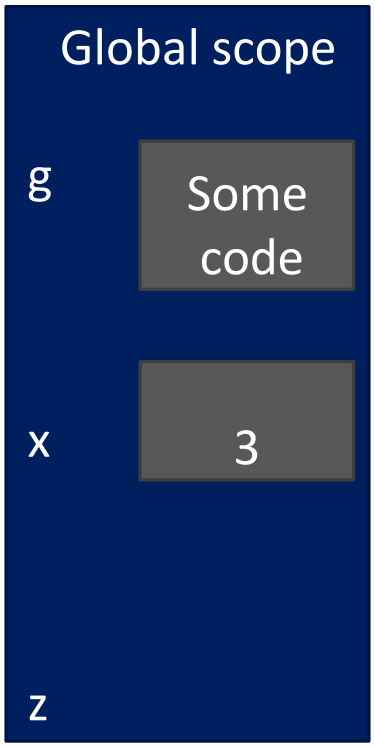


EXAMPLE: NESTED FUNCTION

Murdoch UNIVERSITY

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

Some code

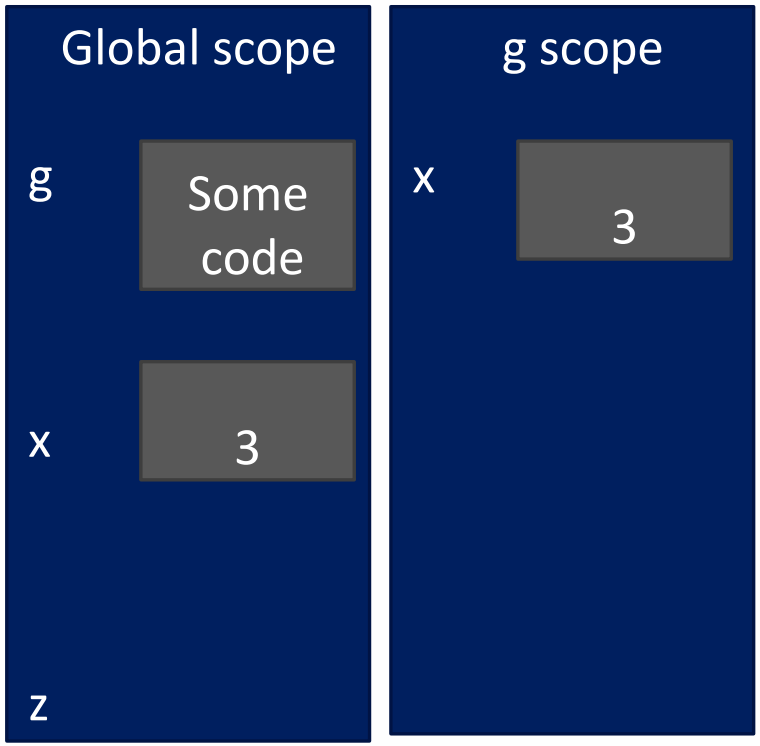



```
x = 3
z = g(x)
```

EXAMPLE: NESTED FUNCTION

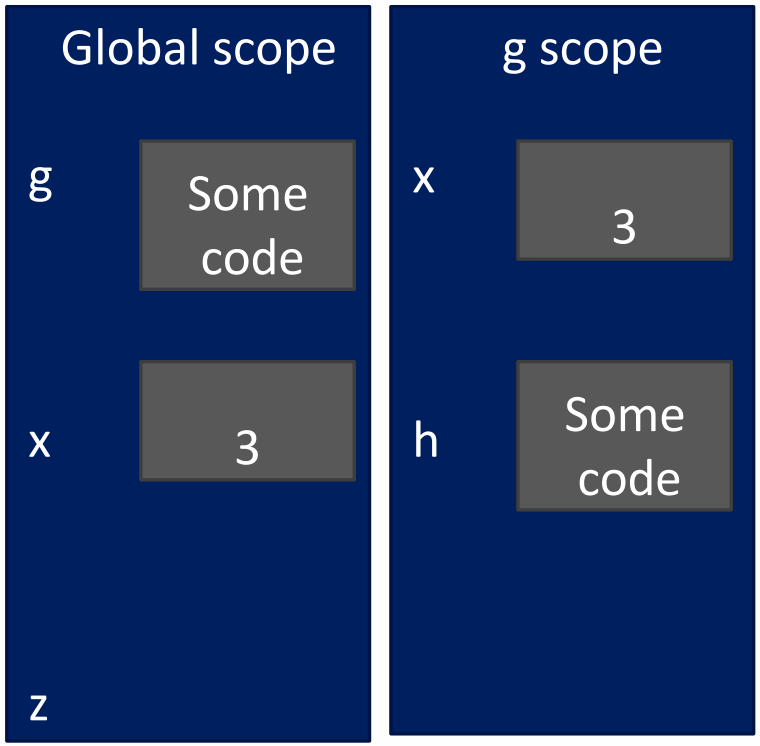
```
def g(x):
    def h():
        x = 'abc'
    x = x + 1
    print('g: x =', x)
    h()
    return x
```

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



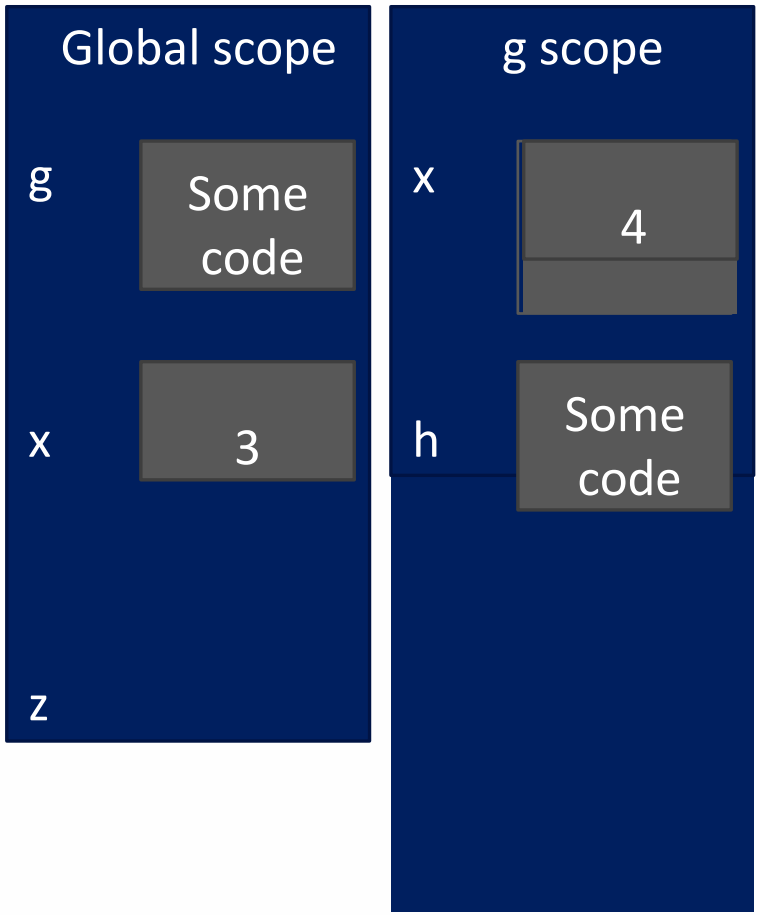
EXAMPLE: NESTED FUNCTION

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

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



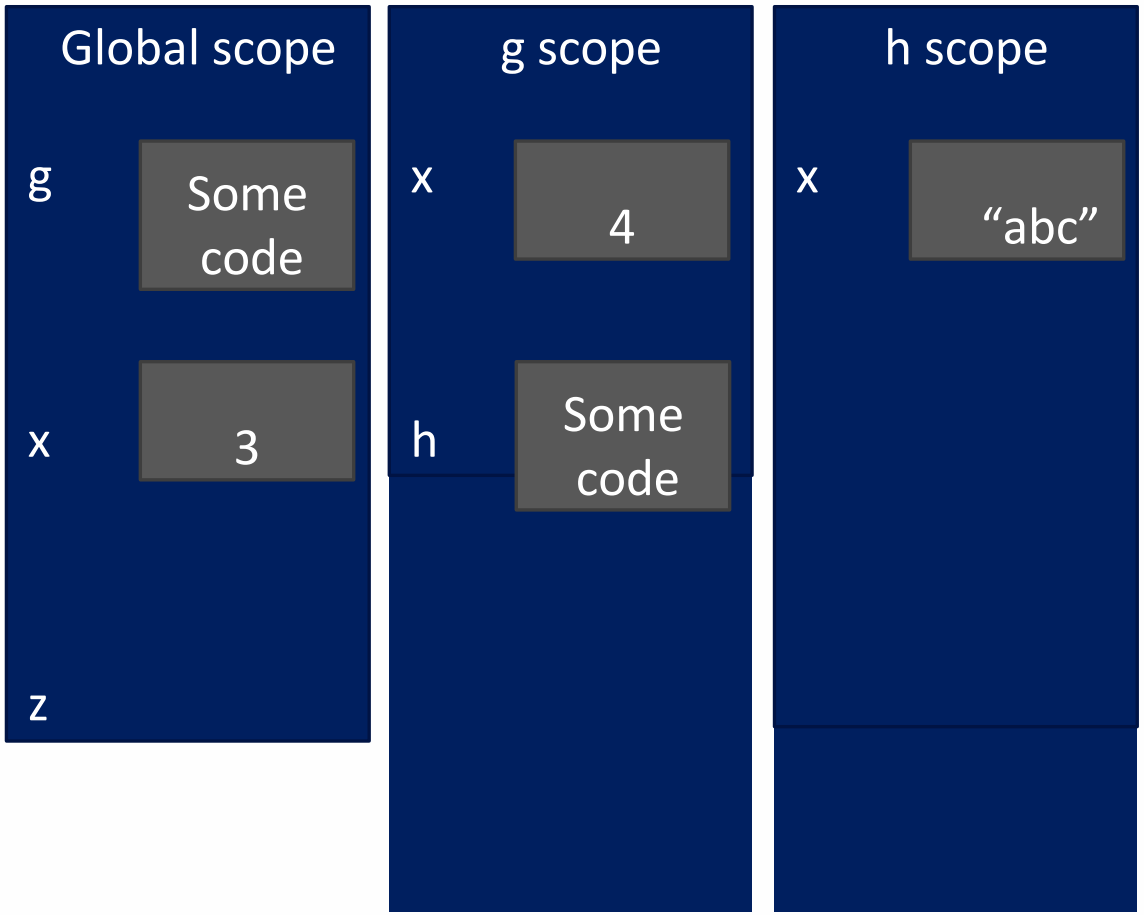
EXAMPLE: NESTED FUNCTION

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

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



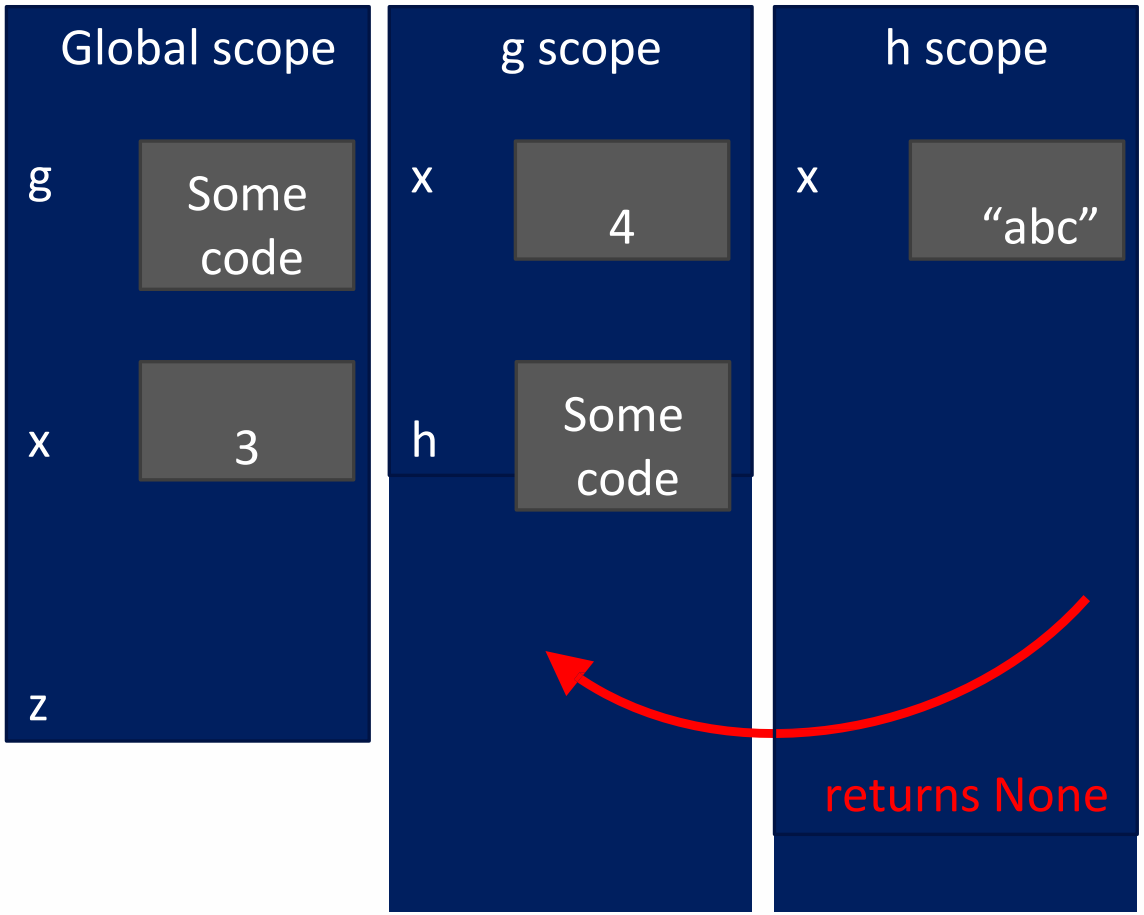
EXAMPLE: NESTED FUNCTION

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

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



EXAMPLE: NESTED FUNCTION

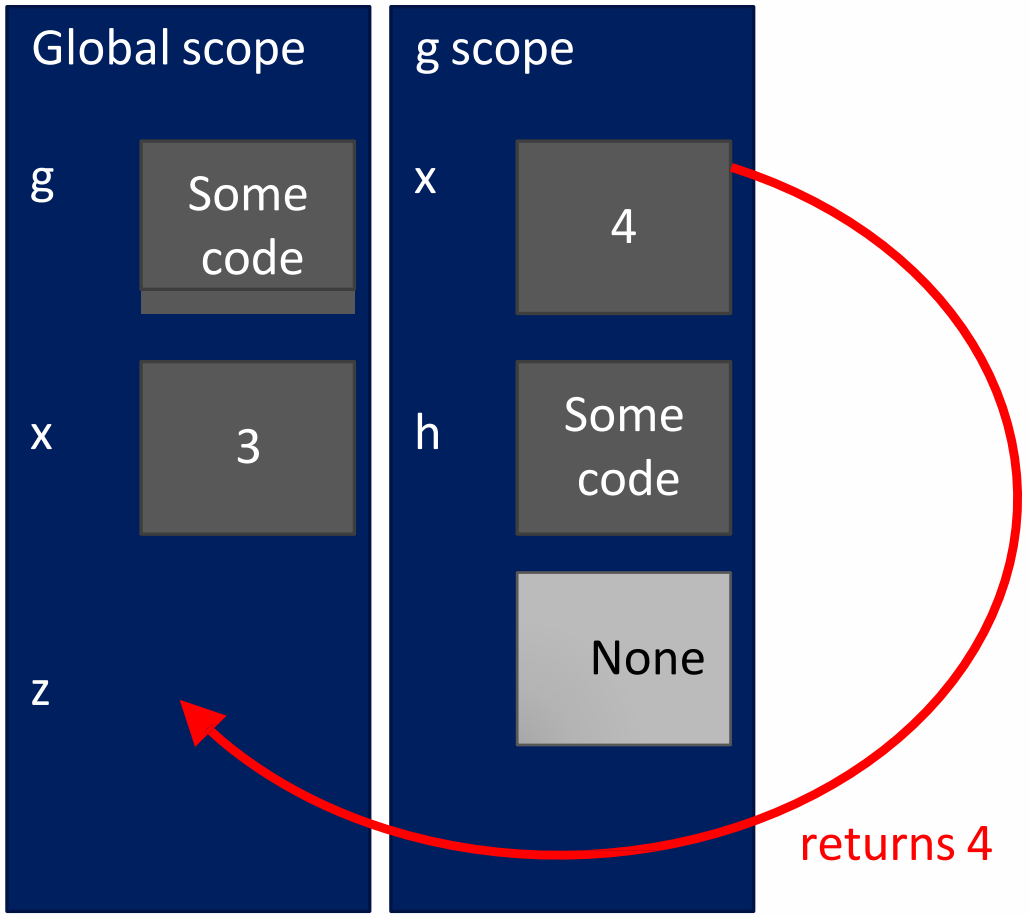
```
def g(x):
    def h():
        x = 'abc'
    x = x + 1
    print('g: x =', x)
    h()
    return x

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



EXAMPLE: NESTED FUNCTION

Murdoch UNIVERSITY

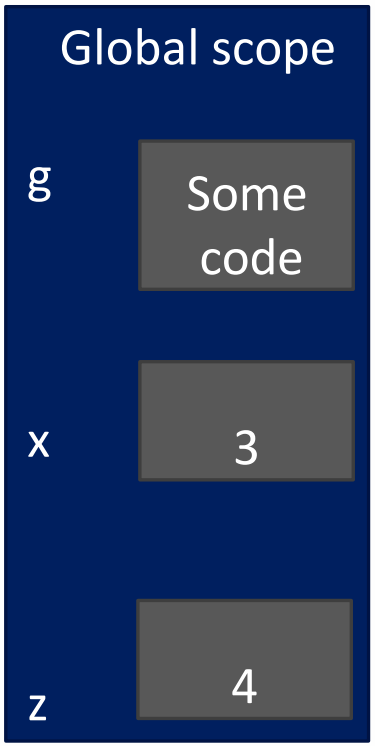
```
def g(x):
    def h():
        x = 'abc'
    x = x + 1
    print('g: x =', x)
    h()
    return x

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



EXAMPLE: NESTED FUNCTION

Murdoch UNIVERSITY

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




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

MODULES

Murdoch UNIVERSITY

- Modules are another way to achieve decomposition
- To put simply, a module is simply a Python code file containing related functions, variables and/or classes that will be used elsewhere
- The file name of the module also serves as the module's name. The file must use extension `.py`, eg, `Example.py`
- You may place executed statements in a module, but it is not recommended

IMPORT MODULES

Murdoch UNIVERSITY

- To use the functions and variables defined in a module, you must import the module into your program

  `import module_name`

- When referring to a function (or variable) defined in a module in your program (another Python file), you must proceed the function (or variable) name with the module name

  `module_name.function_name` *or*

  `module_name.variable_name`

EXAMPLE: A MATH MODULE

Murdoch UNIVERSITY

- The file `circle.py` contains the following code

```
Pi = 3.1415926

def area (radius):
    return Pi * radius * radius

def circumference(radius):
    return Pi * radius * 2.0
```

- In your main program, `main.py`, you may use function `area` from module `circle`

```
import circle
print(circle.area(15.3))
```

ABSTRACTION

Murdoch UNIVERSITY

- Abstraction in computer programming is a fundamental concept that involves simplifying complex systems or processes by breaking them down into manageable and understandable components.
- It allows programmers to focus on the essential aspects of a system while hiding unnecessary details.
- At its core, abstraction helps to create higher-level, more generalized concepts that can be used to represent real-world objects, ideas, or actions in a simplified manner.
- It provides a level of separation between the implementation details and the usage or interaction with those details.

DECOMPOSITION & ABSTRACTION

Murdoch UNIVERSITY

- Abstraction and decomposition, when used together, become a powerful tool in software design
- The code can be used many times but only has to be written and debugged once!
- Python provides several tools for using this powerful idea: functions, modules and classes.

Summary

Murdoch UNIVERSITY

- structuring programs and hiding details
- functions
- Specification of a function
- `return` statement in a function
- scope of variable
- modules
- import module
- abstraction and decomposition as a powerful tool for software design

Acknowledgement

- Sources used in this presentation include:
  - Programiz.com
  - MIT OCW