DS5010

Intro to programming for Data Science

LECTURE 3

TODAY

- review session 2
- learn how to count! (permutations, combinations)
- functions
- modules

REVIEW

- iterations with for/while loops
- tuples, lists, dictionaries
- mutable and immutable objects
- read/write files

HOW TO COUNT?

Problem:

5 houses are competing in DS5010 House Competitions. When the competitions are over, houses will be ranked from 1 to 5 based on their performance. How many different rankings can we have (assuming no two houses will get a same rank)?

PERMUTATIONS

- **permutation** is the act of arranging the members of a set into a **sequence** or **order.**
- So keep in mind, in a permutation always the **order** of elements matter.
- If you change the order of the selected elements you will have a different permutation.

FACTORIAL

$$n! = n \times (n-1) \times (n-2) \times \cdots \times 2 \times 1$$

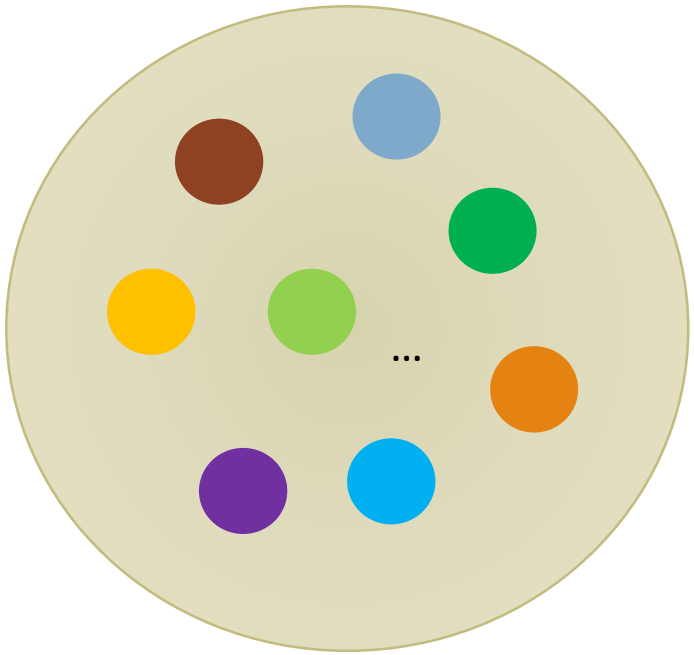
$0! = 1$

by definition!

$61!$ > the number of atoms in the universe!!(10^{82})

PERMUTATIONS with NO REPETITIONS

a set of **n** objects



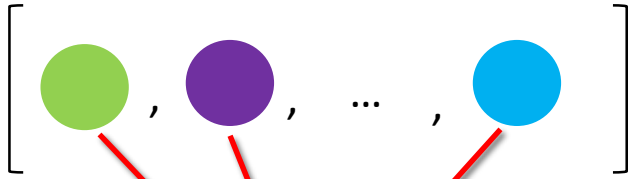
select **r** of them

place them in an **ordered sequence**

the **rule of product**

$$n \times (n-1) \times \cdots \times (n-r+1) = P_r^n = \frac{n!}{(n-r)!}$$

the number of possible permutations



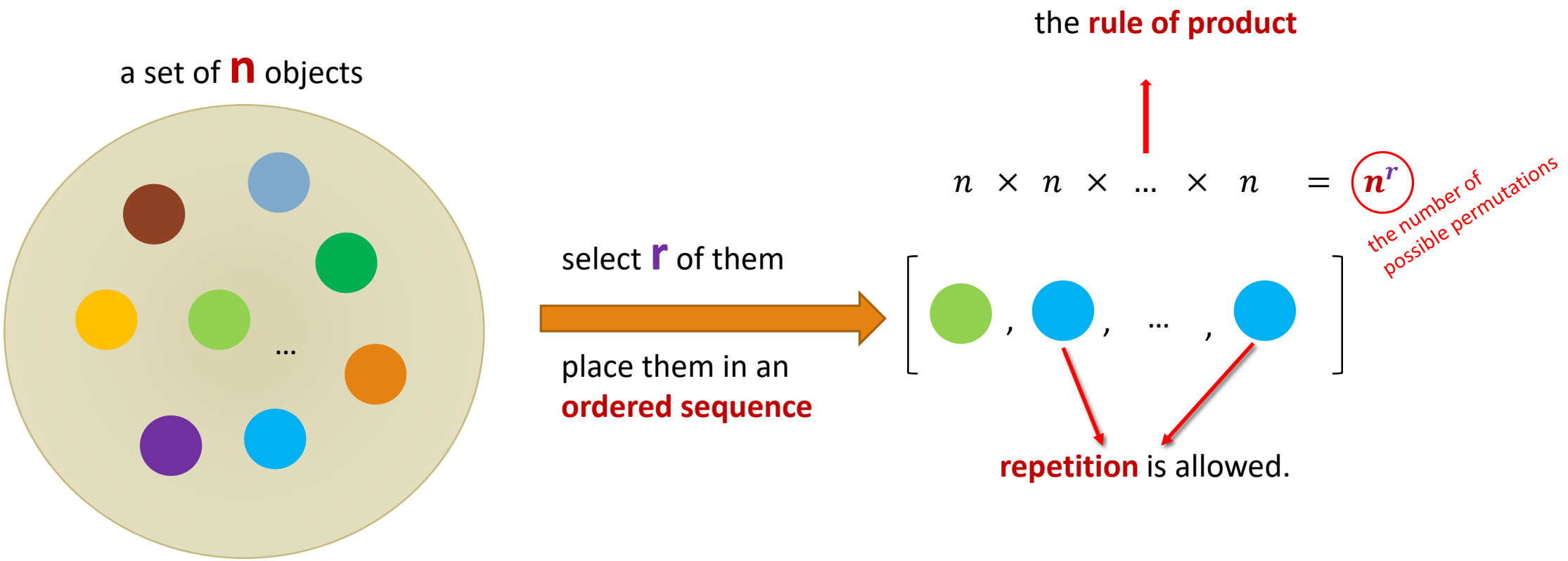
repetition is not allowed.
all elements should be different.

HOW TO COUNT?

Problem:

We want to generate a secret code of length **5** from a set of **36** characters. How many different secret codes can we generate?

PERMUTATIONS with REPETITIONS ALLOWED



HOW TO COUNT?

Problem:

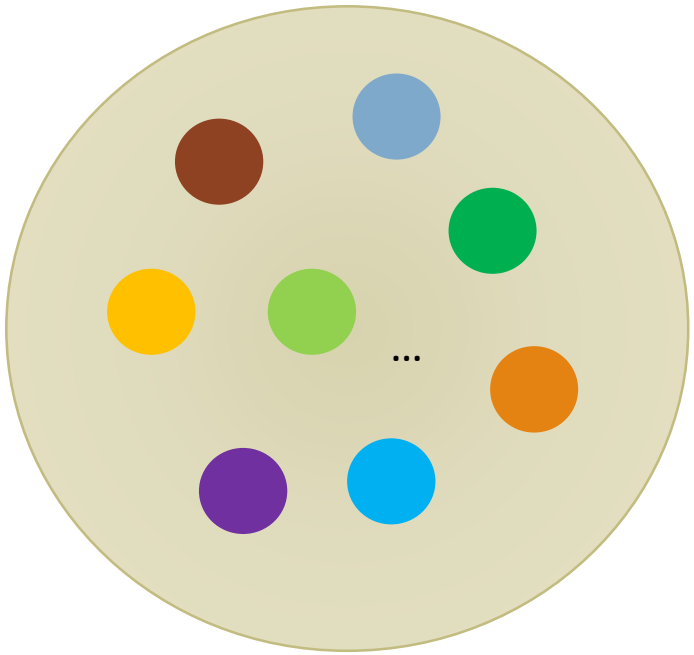
There are **24** students in DS5010 class and I want to choose **4** of them to form a new house which is called Tyrell! How many different shapes can the house of Tyrell have?

COMBINATIONS

- a **combination** is a selection of items from a collection, such that (unlike permutations) the **order** of selection **does not** matter.
- If you change the order of the selected elements you will have the same combination!

COMBINATIONS with NO REPETITIONS

a set of **n** objects



select **r** of them

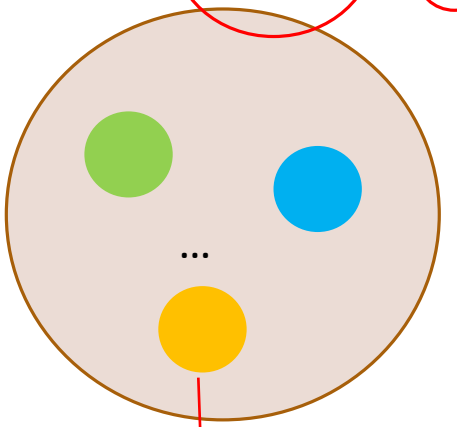
place them in a set were **order** doesn't matter.

the number of possible combinations

$$C_r^n = \frac{n!}{(n-k)! \times k!}$$

the number of **permutations!**

K elements in permutations are repeated **k!** times



repetition is not allowed.
all elements should be different.

HOW TO COUNT?

Problem:

Let us say there are five flavors of ice-cream: **banana**, **chocolate**, **lemon**, **strawberry** and **vanilla**.

We can have **three scoops**. How many variations will there be?

COMBINATIONS with REPETITIONS

- this is out of the scope of this class, but if you are interested, the formula for it is:

$$\binom{r+n-1}{r} = \frac{(r+n-1)!}{(n-1)! \times r!}$$

SOME USEFUL RELATIONS

$$\binom{n}{k} = \binom{n}{n-k}$$

$$\binom{n}{k} = \binom{n-1}{k} + \binom{n-1}{k-1}$$

rule of sum

BINOMIAL EXPANSION

- what is the expanded version of $(x + y)^n$?

the answer is:

$$\sum_{k=0}^{n} \binom{n}{k} x^k y^{n-k}$$

where does this come from?

BINOMIAL COEFFICIENTS

what will be the coefficient of $\boldsymbol{x^k y^{n-k}}$ in the following multiplication expression?

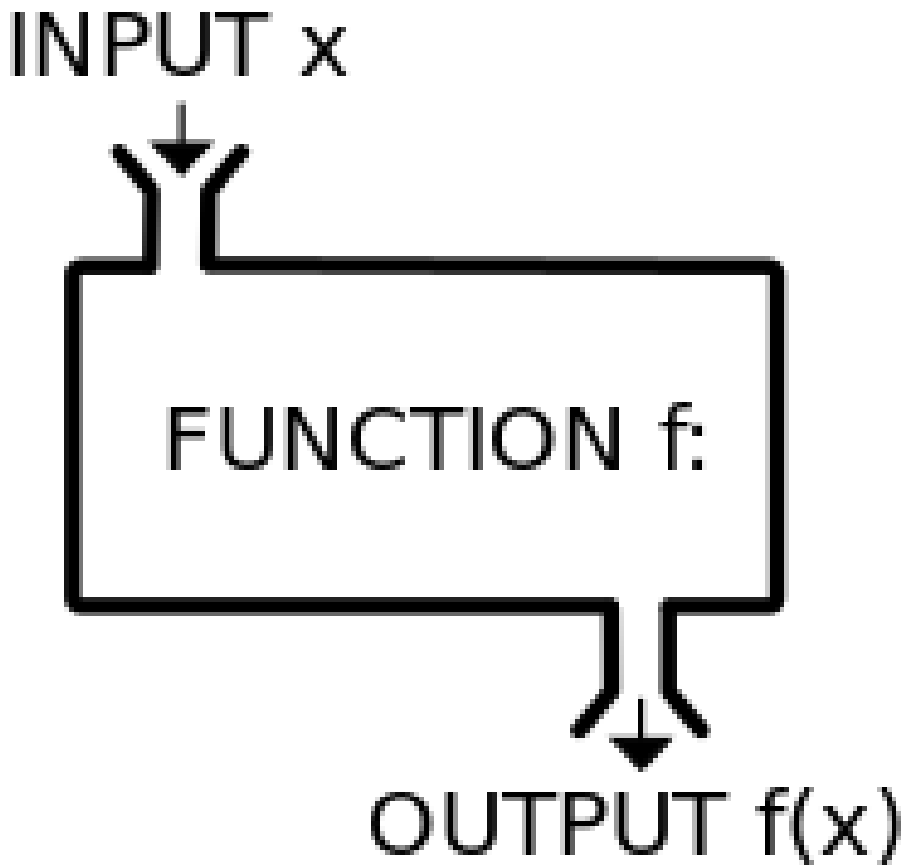
$$(x+y)^n = \overbrace{(x+y)\times(x+y)\times(x+y)\times\cdots\times(x+y)}^{n \text{ times}}$$

In order to have x^k exactly k of the above terms should have been multiplied together.

In how many ways can we select k terms from n terms? $\binom{n}{k}$

so if we add all the $\boldsymbol{x^k y^{n-k}}$ terms together we will have $\binom{n}{k} x^k y^{n-k}$

it is equivalent to $\binom{n}{n-k} x^k y^{n-k}$ (if we considered y instead of x)



FUNCTIONS

HOW DO WE WRITE CODE?

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

- 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

- 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 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**: different devices work together to achieve an end goal

APPLY THESE CONCEPTS

TO PROGRAMMING!

CREATE STRUCTURE with DECOMPOSITION

- in projector example, separate devices
- in programming, divide code into **functions, modules, classes** (covered later).
  - are **self-contained**
  - used to **break up** code
  - intended to be **reusable**
  - keep code **organized**
  - keep code coherent

SUPRESS DETAILS with ABSTRACTION

- 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

- 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 or arguments

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

is_even(3)
```

specifications, docstring

body

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

IN THE FUNCTION BODY

```
def is_even( i ):
    """
    param 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

- **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

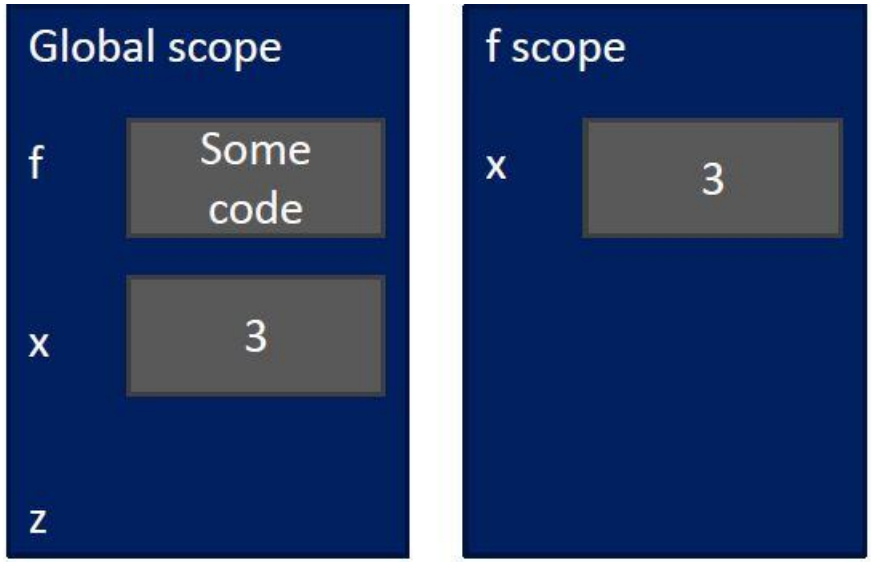
actual parameter

Function definition

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

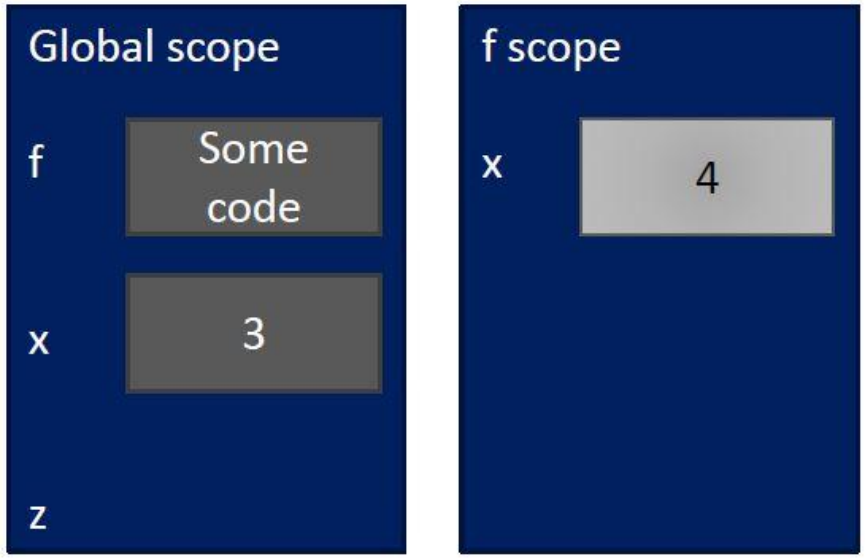
VARIABLE SCOPE

```
def f( x ):
    x = x + 1
    print('in f(x): x =', x)
    return x
x = 3
z = f( x )
```



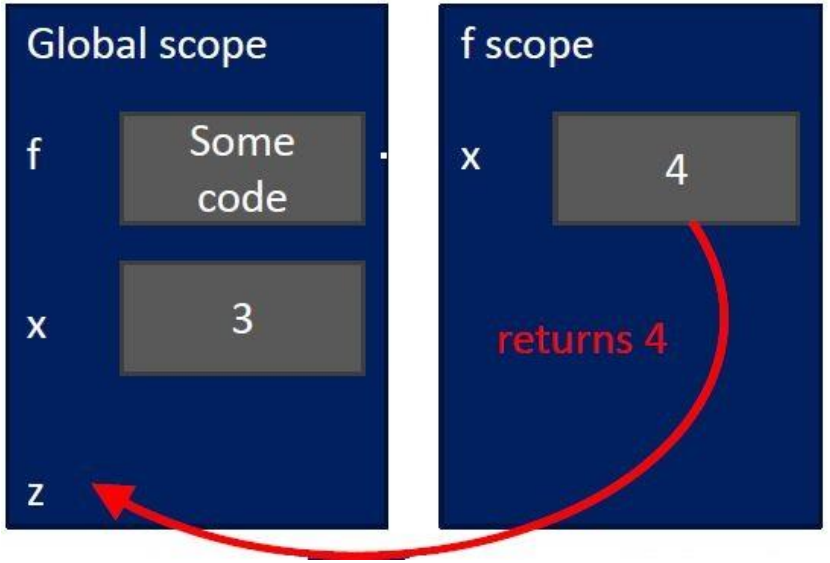
VARIABLE SCOPE

```
def f( x ):
    x = x + 1
    print('in f(x): x =', x)
    return x
x = 3
z = f( x )
```



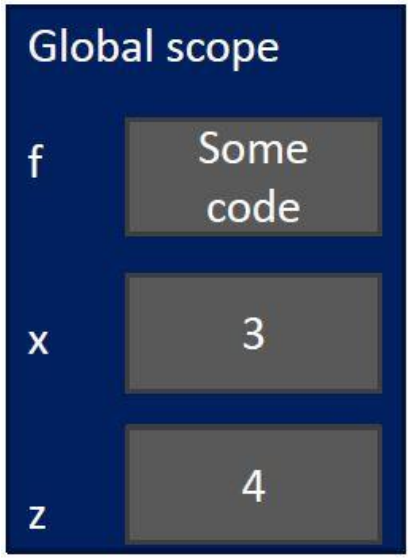
VARIABLE SCOPE

```
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

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

without a return keyword

- 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)
```

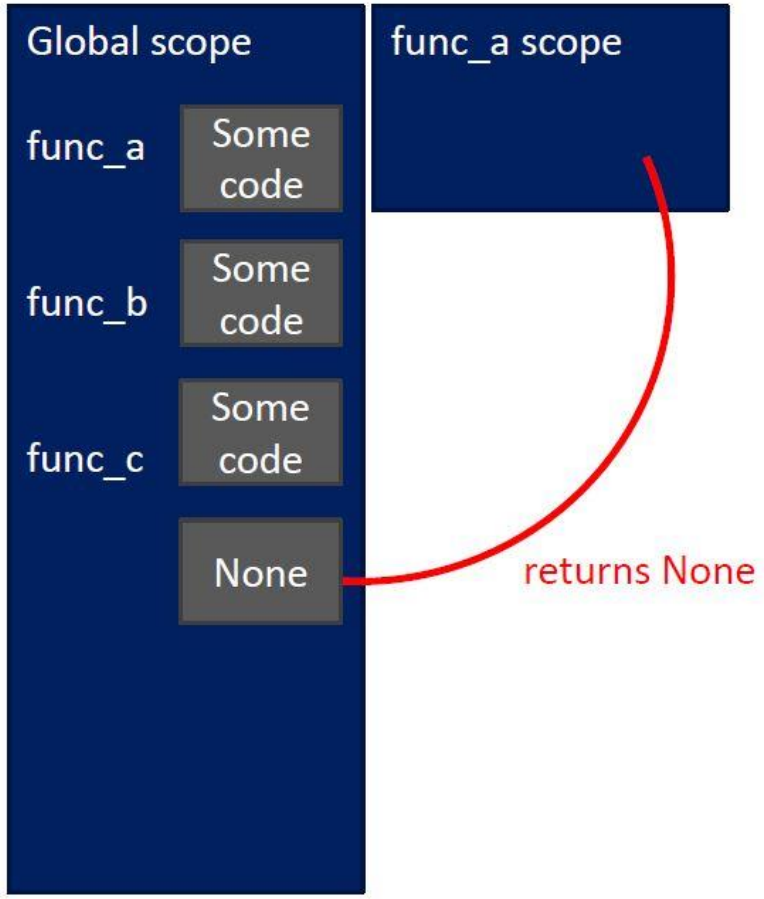
call func_a, takes no parameters

call func_b, takes one parameter

call func_c, takes one parameter, which is another function

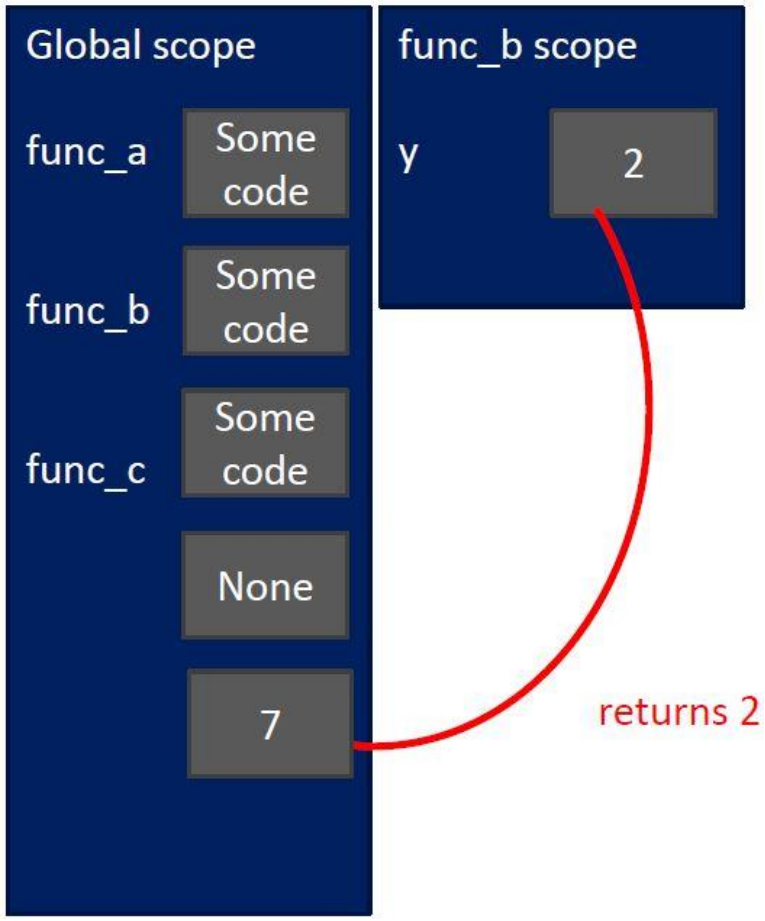
FUNCTIONS AS ARGUMENTS

```
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)
```



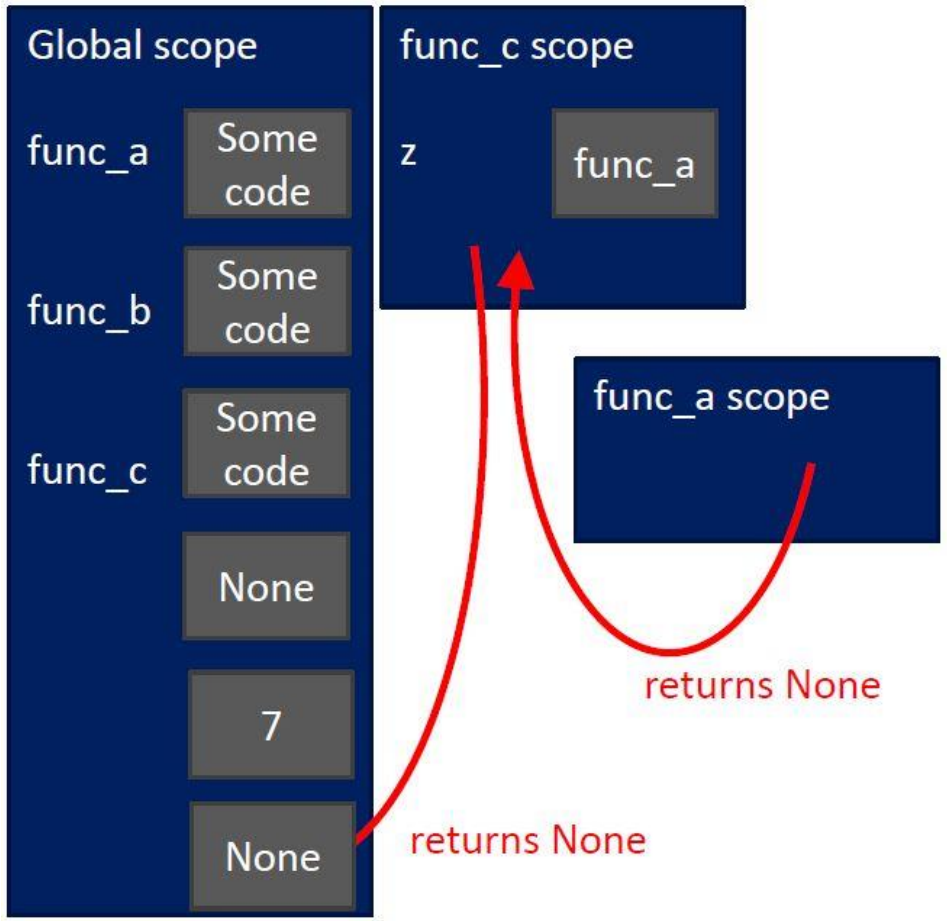
FUNCTIONS AS ARGUMENTS

```
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)
```



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 it is discouraged.

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

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

x is redefined in scope of f

```
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 it is discouraged.

```
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 from global/main program scope

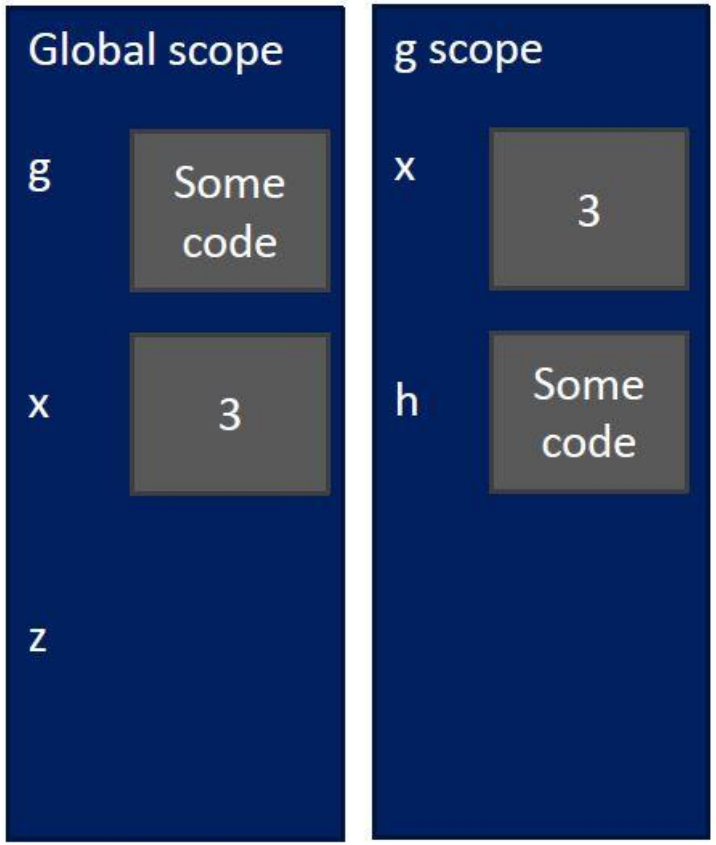
SCOPE DETAILS

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

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

some code

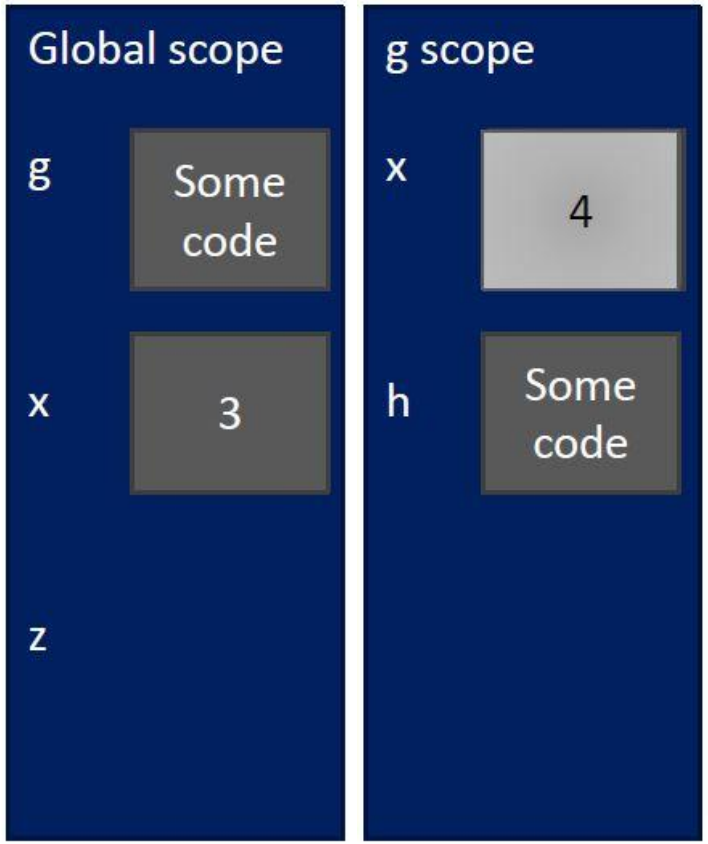
SCOPE DETAILS

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

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



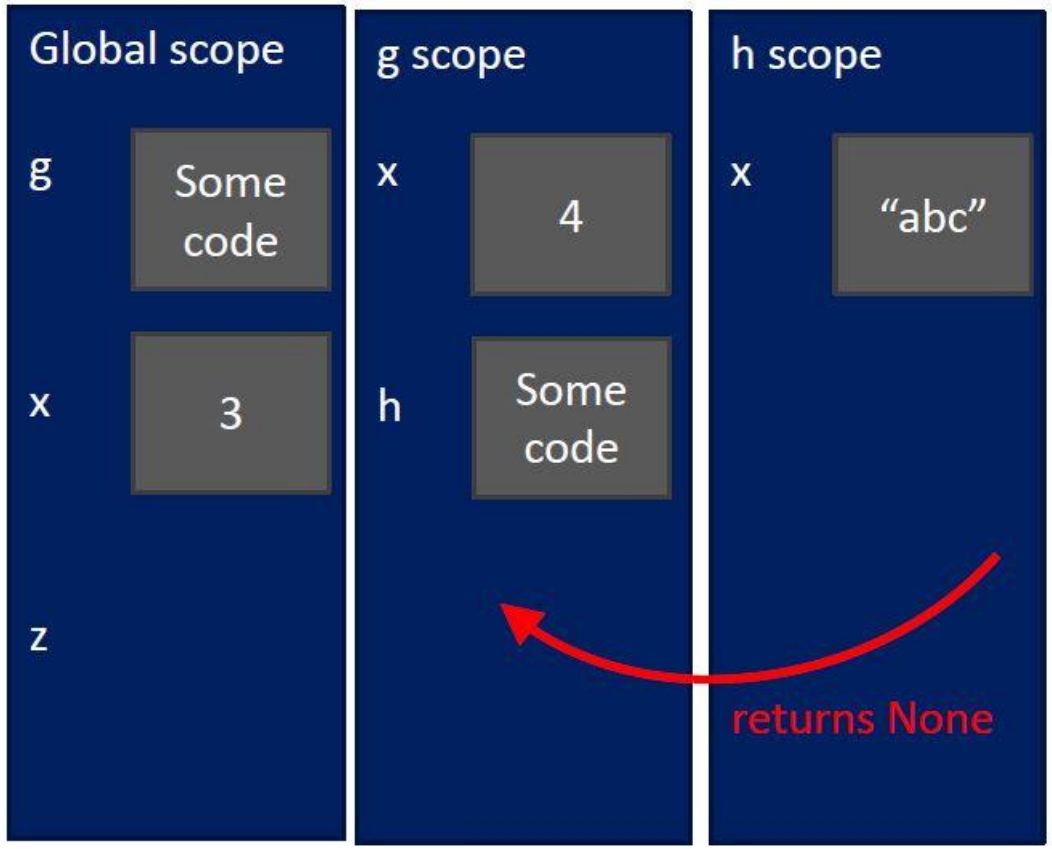
SCOPE DETAILS

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

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



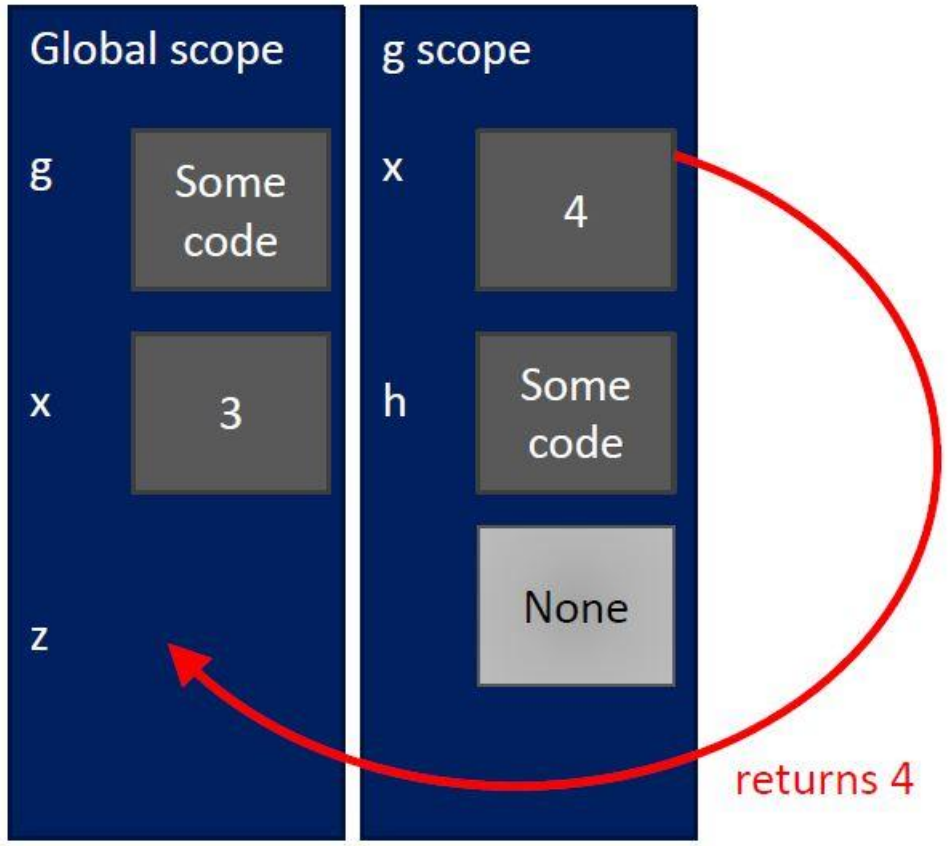
SCOPE DETAILS

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

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



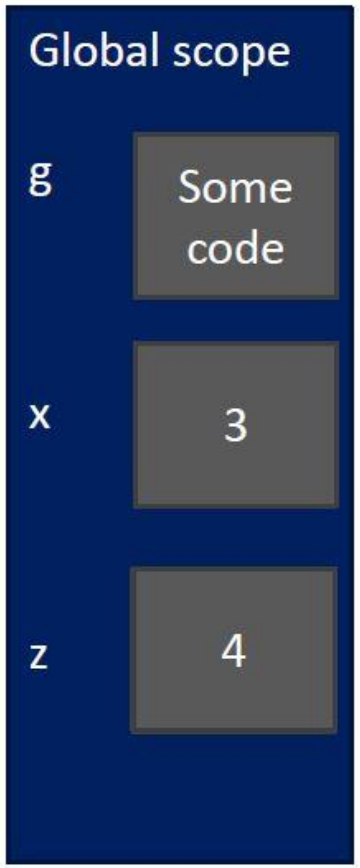
SCOPE DETAILS

```
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)
```



DECOMPOSITION & ABSTRACTION

- powerful together
- code can be used many times but only has to be debugged once!

MODULES

- each python file which ends with `.py` is called a **module**.
- a module is **a collection** of Python definitions and statements.
- when your program gets longer, **decompose** it into several modules for **easier maintenance**.
- the definitions and statements in each module should **serve a same purpose**.

MODULES

example:

- assume we are writing a registration system for NEU
- we want to divide our program into two modules, one for students, and another one for faculties.
- we use the definitions of one module in another one by **importing** it.
- visit –link– to access the example code.

IMPORTING MODULES

- each module has its **own private symbol table** used as the global symbol table by all functions in the module
- each module is imported **once** per interpreter session

```
import some_module
```

- can import names from a module into the importing module's symbol table:

```
from some_module import m1, m2 (or *)
```

avoid * as much as possible as it may shadow other names

MODULE ALIAS NAME

- you can give shorter names to modules with long names, to be easier to use.

```
import my_long_named_module as my_mod
```

BUILTIN STANDARD LIBRATIES

- math
- random
- itertools
- string
- datetime
- statistics
- os
- sys
- and many more…