

INT 489 Selected Topics IN IT

Dr. Khaled Mostafa Reda

Sinai University (SU)

**Faculty of Information Technology and Computer
Science (FIT)**

E-Mail: khaled.Mostafa@su.edu.eg

Lecture #6

Recursion, Dictionaries

Last Time

- Tuples - immutable
- Lists – mutable
- Aliasing, cloning
- Mutability side effects

TODAY

- Review
- Recursion – divide/decrease and conquer
- Dictionaries – another mutable object type

Question (1)

- Analyze and design an algorithm by drawing flowcharts and writing pseudo-code to find the roots of quadratic equations $ax^2 + bx + c = 0$ and print a message the roots of the equation are different real numbers, equal real numbers, or imaginary numbers using if statement according to the following:

- where $D = b^2 - 4ac$ $r1 = \frac{-b+\sqrt{D}}{2a}$, $r2 = \frac{-b-\sqrt{D}}{2a}$

- $$\left\{ \begin{array}{l} r1, r2 \text{ and the roots are different real numbers if } D > 0 \\ r1, r2 \text{ and the roots are equal real numbers if } D = 0 \\ \text{the roots are imaginary numbers if } D < 0 \end{array} \right.$$

- Finally**, write a Python program to express your design.
Given its 3 coefficients a , b , and c .

Analysis

Input: a, b, c

Output; roots $r1, r2$ of the equation

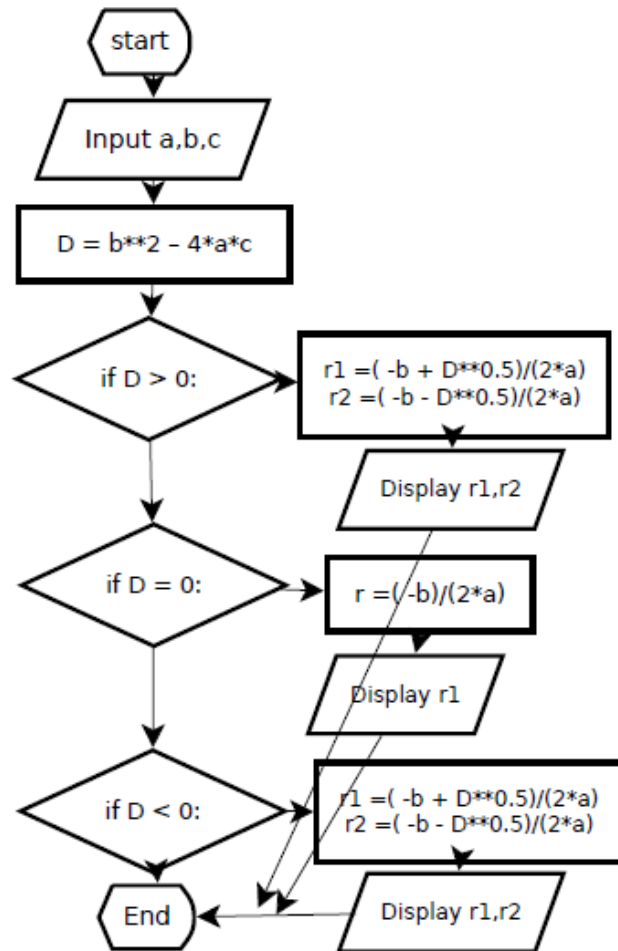
Relation:

$$\text{where } D = b^2 - 4ac \quad r1 = \frac{-b+\sqrt{D}}{2a} \quad , r2 = \frac{-b-\sqrt{D}}{2a}$$

$$\left\{ \begin{array}{l} r1, r2 \text{ and the roots are different real numbers if } D > 0 \\ r1, r2 \text{ and the roots are equal real numbers if } D = 0 \\ \text{the roots are imaginary numbers if } D < 0 \end{array} \right.$$

Question (1), Cont.

Design (flowchart)



Design (pseudo-code)

Start

Input a, b, and c

Calculate discriminant $D = b^2 - 4ac$

if $D > 0$:

$r1 = \frac{-b + D^{0.5}}{2a}$

$r1 = \frac{-b - D^{0.5}}{2a}$

Print ("The roots are two distinct real numbers.")

print("the roots are different", " r1 = ",r1," and
r2 = ",r2)

elif $D = 0$:

$r = \frac{-b}{2a}$

Print ("The roots are two equal real numbers.")

print("the roots are equal", " r = ",r)

else:

$r1 = \frac{-b + (d)^{0.5}}{(4a)}$
 $r2 = \frac{-b - (d)^{0.5}}{(4a)}$

Print ("The roots are two imaginary numbers.")

print("the roots has imagine parts", " r1 = ",r1,"
and r2 = ",r2)

End

Question (1), Cont.

Python Code

```
a = int(input("the parameter of x^2= "))
b = int(input("the parameter of x= "))
c = int(input("the constant  = "))
d=((b**2)-(4*a*c))
if d>0:
    r1=(-b+(d)**0.5)/(4*a)
    r2=(-b-(d)**0.5)/(4*a)
    print("the roots are different", " r1 = ",r1," and r2 = ",r2)
elif d==0:
    r1= -b/(4*a)
    r2= -b/(4*a)
    print("the roots are equals", " r1 = ",r1," and r2 = ",r2)
else:
    r1=(-b+(d)**0.5)/(4*a)
    r2=(-b-(d)**0.5)/(4*a)
    print("the roots has imagine parts", " r1 = ",r1," and r2 = ",r2)
```

Question (2)

- *Analyze and design an algorithm by writing pseudo-code to **use a function** that displays a student's grade (A if the score ≥ 90 , B if the score is between 75:89, C if the score is between 65:74, D if the score is between 50: 64, and F if less than 50).*
- *The program should contain a function called Grade that takes the student's score and returns its grade to the main program. **Finally**, write a Python program to express your design.*

Analysis

- Input: score
- Output: grade

$$\text{Relation: grade} = \begin{cases} A & \text{if score} \geq 90 \\ B & \text{if } 75 \leq \text{score} \leq 89 \\ C & \text{if } 65 \leq \text{score} \leq 74 \\ D & \text{if } 50 \leq \text{score} \leq 64 \\ F & \text{if score} \leq 49 \end{cases}$$

Question (2) Cont.,

Design (pseudo-code)

- Grade(score)
 - If (score \geq 90):
 - grade=A
 - elif ((score \geq 75) and (score \leq 89)):
 - grade=B
 - elif ((score \geq 65) and (score \leq 74)):
 - grade=C
 - elif ((score \geq 50) and (score \leq 64)):
 - grade=D
 - else:
 - grade=F

Ask the user to enter the student's score.

Get a student score.

G=grade(score)

Display: G

Python Code

```
def Grade(x):  
    if (x $\geq$ 90):  
        grd='A'  
    elif ((x $\geq$ 75) and (x $\leq$ 89)):  
        grd='B'  
    elif ((x $\geq$ 65) and (x $\leq$ 74)):  
        grd='C'  
    elif ((x $\geq$ 50) and (x $\leq$ 64)):  
        grd='D'  
    else :  
        grd='F'  
    return grd  
  
score = float(input("Student score = "))  
G=Grade(score)  
print("Student Grade is: ",G)
```

Question (3)

- The form of student ID at Sinai University is **username@domain name**
- Write a Python program to perform the following: -
 - **a)** Read a number ***n*** of students (entered by a user) and store them in a tuple.
 - **b)** From the students' IDs that are entered by a user, create two new tuples, one to store only the usernames and the second to store domain names.
 - **c)** Print all three tuples at the end of the program.

Question (3), Cont.

```
emails = tuple()
username = tuple()
domainname = tuple()
# Create empty tuple 'emails', 'username' and domain-name
n = int(input("How many email ids you want to enter?: "))
for i in range(0,n):
    emid = input("> ")
    #It will assign emailid entered by the user to the tuple 'emails'
    emails = emails +(emid,)
    #This will split the email ID into two parts, username and domain and return a list
    spl = emid.split("@")
    #assigning returned list first part to username and second part to domain name
    username = username + (spl[0],)
    domainname = domainname + (spl[1],)
```

Question (3), Cont.

- `print("\nThe email ids in the tuple are:")`
- `#Printing the list with the email IDs`
- `print(emails)`
- `print("\nThe username in the email ids are:")`
- `#Printing the list with the usernames only`
- `print(username)`
- `print("\nThe domain name in the email ids are:")`
- `#Printing the list with the domain names only`
- `print(domainname)`

Recursion

- **Recursion** is the process of repeating items in a self-similar way.

What is Recursion?

- **Algorithmically:** a way to design solutions to problems by **divide-and-conquer** or **decrease-and-conquer**
 - reduce a problem to simpler versions of the same problem
- **Semantically:** a programming technique where a **function calls itself**
 - In programming, goal is to NOT have infinite recursion
 - Must have **1 or more base cases** that are easy to solve.
 - Must solve the same problem on **some other input** with the goal of simplifying the larger problem input.

Iterative Algorithms So Far

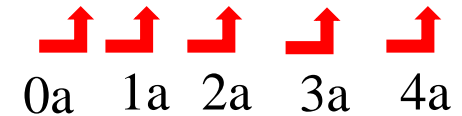
- Looping constructs (while and for loops) lead to **iterative** algorithms.
- Can capture computation in a set of **state variables** that update on each iteration through loop

Multiplication – Iterative Solution

- “Multiply $a * b$ ” is equivalent to “add a to itself b times”.

$a + a + a + a + \dots + a$

- Capture **state** by


0a 1a 2a 3a 4a

—An **iteration** number (i) starts at b

$i \leftarrow i-1$ and stop when 0

—A current **value of computation** (result)

$\text{result} \leftarrow \text{result} + a$

```
def mult_iter(a, b):
```

```
    result = 0
```

```
    while b > 0:
```

```
        result += a
```

```
        b -= 1
```

```
    return result
```

Iteration

Current value of computation, a running sum

Current value of iteration variable

Multiplication – Recursive Solution

- **Recursive step**

—think how to reduce problem to a **simpler/smaller version** of same problem

- **base case**

—keep reducing problem until reach a simple case that can be **solved directly**

—when $b = 1$, $a*b = a$

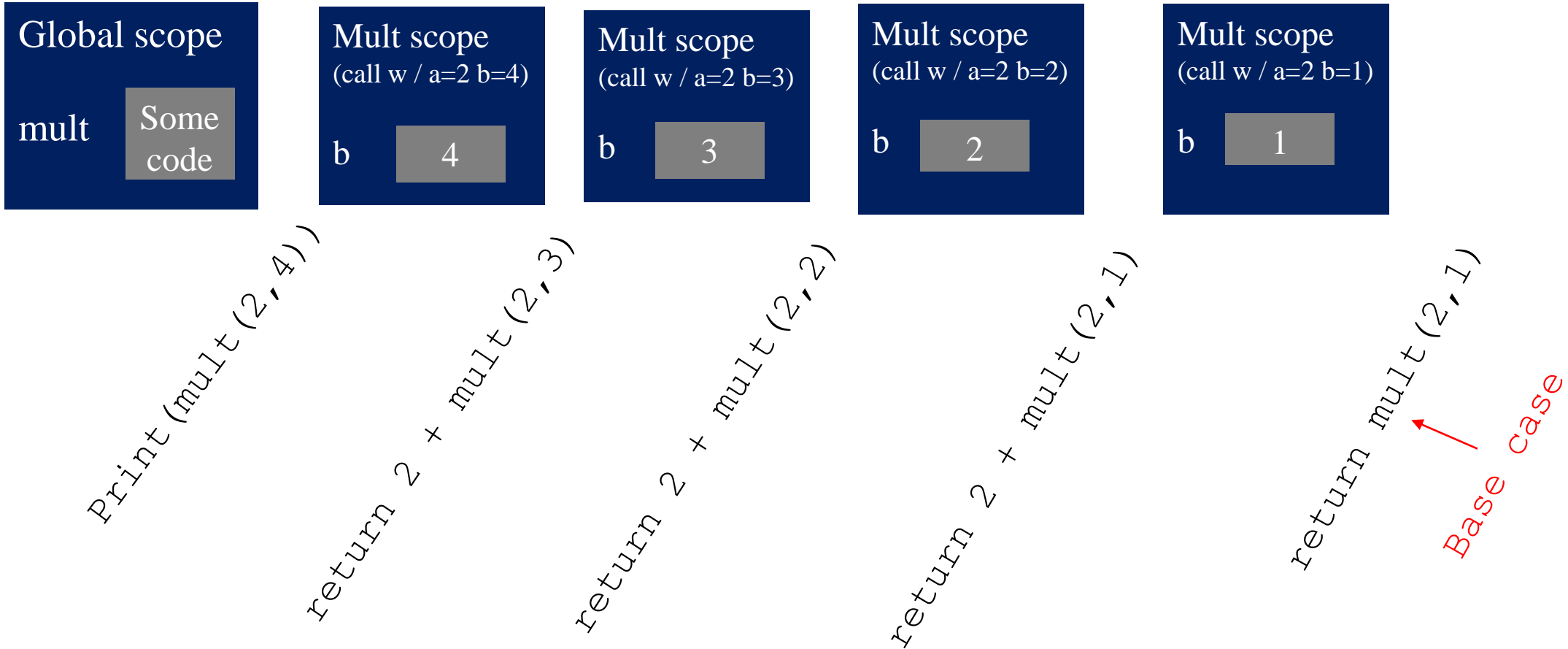
$$\begin{aligned} a*b &= \underbrace{a + a + a + a + \dots + a}_{b \text{ times}} \\ &= a + \underbrace{a + a + a + \dots + a}_{b-1 \text{ times}} \\ &= a + \boxed{a * (b-1)} \quad \text{recursive reduction} \end{aligned}$$

```
def mult(a, b):  
    if b == 1: base case  
        return a  
    else:  
        return a + mult(a, b-1)
```

recursive step

Multiplication – Recursive Solution

```
def mult(a, b):  
    if b == 1: base case  
        return a  
    else: recursive step  
        return a + mult(a, b-1)
```



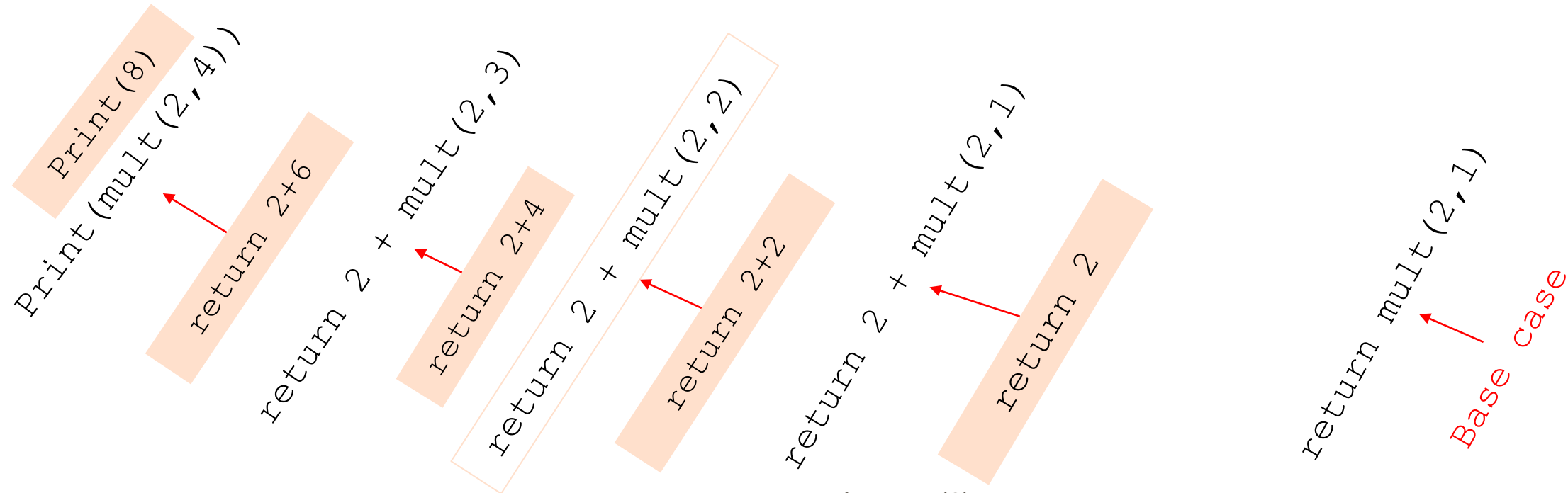
Multiplication – Recursive Solution

```
def mult(a, b):  
    if b == 1: base case  
        return a  
    else: recursive step  
        return a + mult(a, b-1)
```

Global scope

mult

Some
code



Inductive Reasoning

- How do we know that our recursive code will work?
- `mult_iter` terminates because `b` is initially positive, and decreases by 1 each time around loop; thus, must become less than 1
- `mult` called with `b = 1` has no recursive call and stops
- `mult` called with `b > 1` makes a recursive call with a smaller version of `b`; must eventually reach call with `b = 1`

```
def mult_iter (a, b):  
    result = 0  
    while b > 0:  
        result += a  
        b -= 1  
    return result
```

```
def mult (a, b):  
    if b == 1:  
        return a  
    else:  
        return a + mult (a, b-1)
```

Factorial – Recursive Solution

- $n! = n * (n-1) * (n-2) * (n-3) * \dots * 1$
- for what n do we know the factorial?
 - $n = 1 \rightarrow \text{if } n == 1:$

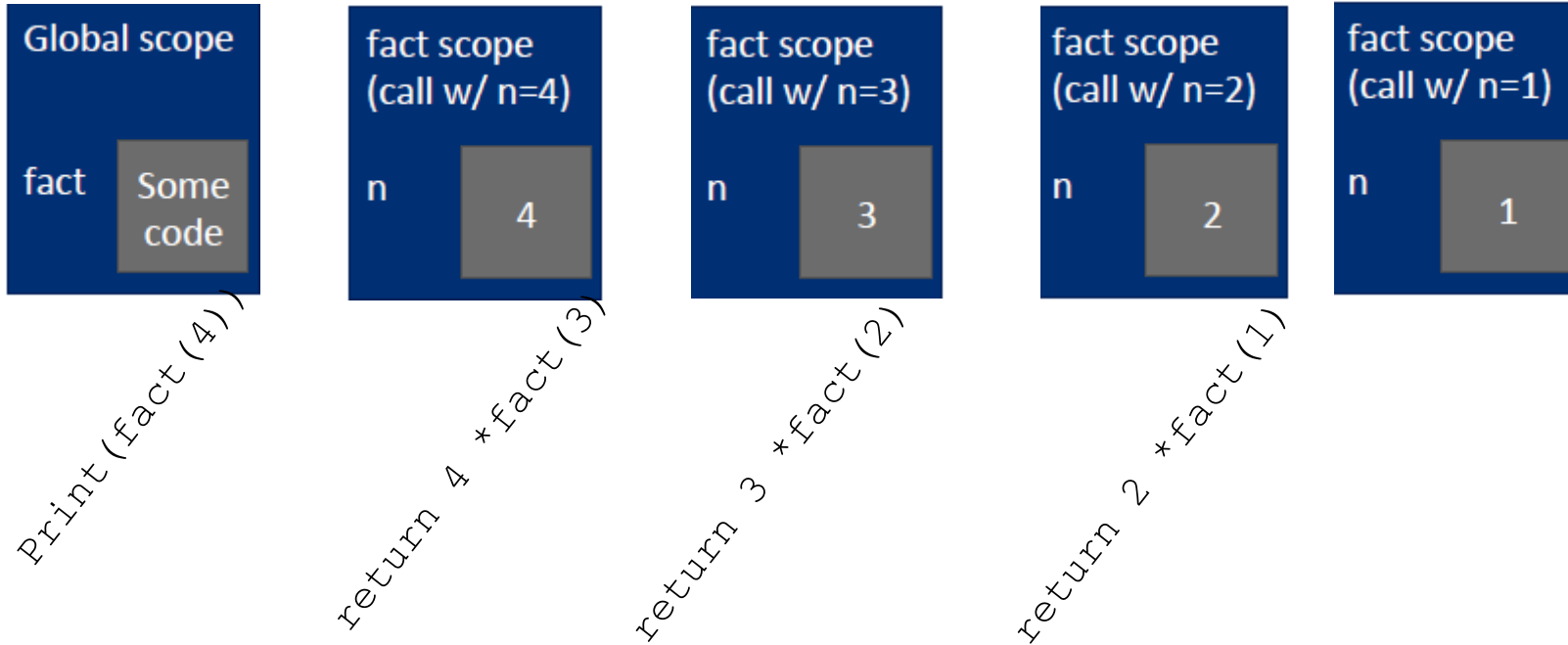
return 1

base case
- How to reduce problem? Rewrite in terms of something simpler to reach base case
 - $n * (n-1)! \rightarrow \text{else:}$

return n*factorial(n-1)

recursive step

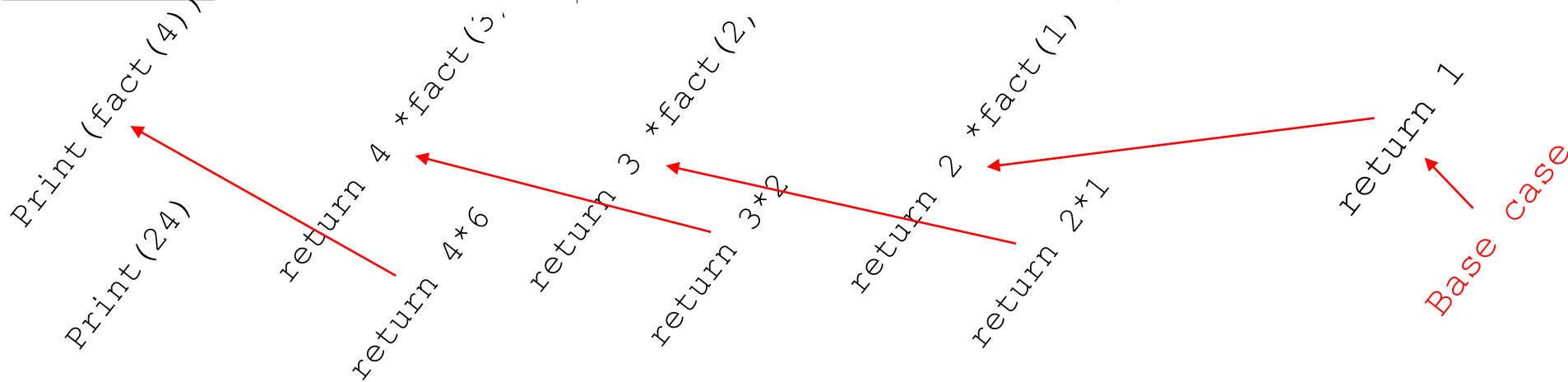
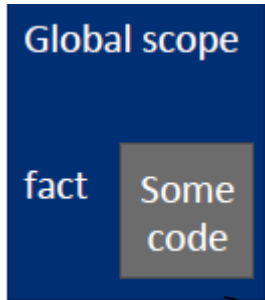
Recursive Function Scope Example



```
def fact(n):  
    if n == 1:  
        return 1  
    else:  
        return  
n*fact(n-1)  
print(fact(4))
```

Recursive Function Scope Example

```
def fact(n):  
    if n == 1:  
        return 1  
    else:  
        return  
n*fact(n-1)  
print(fact(4))
```



Some Observations

- Each recursive call to a function creates its **own scope/environment**
- **Bindings of variables** in a scope are not changed by recursive call
- Flow of control passes back to **previous scope** once function call returns value

Using the same variable names but they are different objects in separate scope

Recursive Vs Iteration

Criteria	Recursive	Iteration
Definition	When a function calls itself directly or indirectly.	When some set of instructions are executed repeatedly.
Implementation	Implemented using function calls.	Implemented using loops.
Format	Base case and recursive relation are specified.	Includes initializing control variable, termination condition, and update of the control variable.
Current State	Defined by the parameters stored in the stack.	Defined by the value of the control variable.
Progression	The function state approaches the base case.	The control variable approaches the termination value.

Recursive Vs Iteration

Criteria	Recursive	Iteration
Memory Usage	Uses stack memory to store local variables and parameters.	Does not use memory except initializing control variables.
Infinite Repetition	It will cause stack overflow error and may crash the system if the base case is not defined or is never reached.	It will cause an infinite loop if the control variable does not reach the termination value.
Code Size	Recursive code is generally smaller and simpler.	Iterative code is generally bigger.
Overhead	Possesses overhead of repeated function calls.	No overhead as there are no function calls.
Speed	Slower in execution.	Faster in execution.

Fibonacci Series

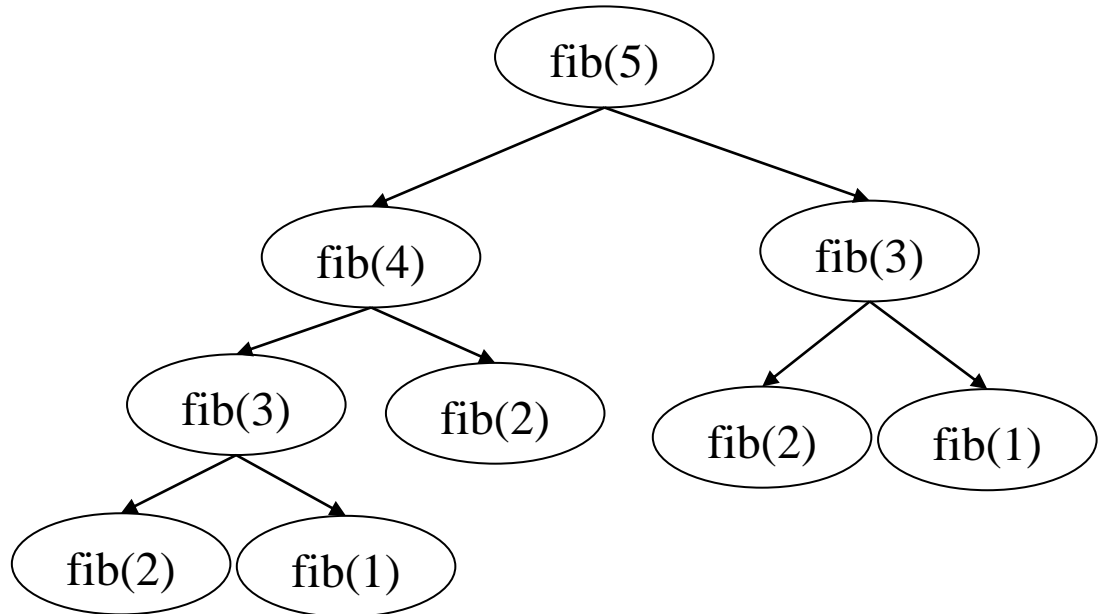
- Fibonacci Series is a pattern of numbers where each number results from adding the last two consecutive numbers.
- The first 2 numbers start with 0 and 1, and the third number in the sequence is $0+1=1$.
- The 4th number is the addition of the 2nd and 3rd number, i.e., $1+1=2$, and so on. The Fibonacci Sequence is the series of numbers:
 $0, 1, 1, 2, 3, 5, 8, 13, 21, 34, \dots$

1 1

1 1 2 3 5 8 13 21

Fibonacci - Recursion with Multiple Base Cases

- Base cases:
 - $\text{fib}(1) = 1$
 - $\text{fib}(2) = 1$
- Recursive case
 - $\text{fib}(n) = \text{fib}(n-1) + \text{fib}(n-2)$



Fibonacci - Recursion with Multiple Base Cases

```
def fib(x):  
    """assumes x an int >= 0 returns Fibonacci of x"""  
    if x == 1 or x == 2:  
        return 1  
    else:  
        return fib(x-1) + fib(x-2)
```

Divide and Conquer

- an example of a “divide and conquer” algorithm
- solve a hard problem by breaking it into a set of sub-problems such that:
 - sub-problems are easier to solve than the original
 - solutions of the sub-problems can be combined to solve the original

Dictionary

How to Store Student Info

- So far, can store using separate lists for every info

```
names = [ 'Hosam', 'Mona', 'Ali', 'lila' ]
```

```
grade = [ 'B', 'A+', 'A', 'A' ]
```

```
course = [CS301, CS303, CS312, CS3014]
```

- a **separate list** for each item
- each list must have the **same length**
- info stored across lists at **same index**, each index refers to info for a different person

How to Update/Retrieve Student Info

```
def get_grade(student, name_list, grade_list, course_list):  
    i = name_list.index(student)  
    grade = grade_list[i]  
    course = course_list[i]  
    return (course, grade)
```

- **Messy** if have a lot of different info to keep track of
- Must maintain **many lists** and pass them as arguments
- Must **always index** using integers
- Must remember to change multiple lists

A better and Cleaner Way – A Dictionary

- nice to **index item of interest directly** (not always int).
- nice to use **one data structure**, no separate lists.

A list

0	Elem 1
1	Elem 2
2	Elem 3
3	Elem 4
...	...

index element

A dictionary

Key 1	Val 1
Key 2	Val 2
Key 3	Val 3
Key 4	Val 4
...	...

custom
index by
label element

A Python Dictionary

- store pairs of data

—key

—value

'Hosam'	'B'
'Mona'	'A+'
'Ali'	'A'
'Lila'	'A-'

*Custom
by index
label*

element

```
my_dict = {}
```

Empty Dictionary

```
grades = {'Hosam': 'B', 'Mona': 'A+', 'Ali': 'A', 'lila': 'A-'}
```

↑
Key 1

↑
Val 1

↑
Key 2

↑
Val 2

↑
Key 3

↑
Val 3

↑
Key 4

↑
Val 4

Dictionary Lookup

- similar to indexing into a list
- **looks up** the **key**
- **returns** the **value** associated with the key
- if key isn't found, get an error

'Hosam'	'B'
'Mona'	'A+'
'Ali'	'A'
'Lila'	'A-'

```
grades = {'Hosam': 'B', 'Mona': 'A+', 'Ali': 'A', 'lila': 'A-'}
```

```
grades[ 'Mona' ] → evaluates to 'A+'
```

```
grades[ 'Mohab' ] → gives a KeyError
```

Dictionary Operations

```
grades = { 'Hosam': 'B', 'Mona': 'A+', 'Ali': 'A', 'Lila': 'A' }
```

- **add** an entry
 - `grades['Mohab'] = 'A'`
- **test** if key in dictionary
 - `'Ali' in grades` → returns `True`
 - `'Esam' in grades` → returns `False`
- **delete** entry
 - `del(grades['Hosam'])`

'Hosam'	'B'
'Mona'	'A+'
'Ali'	'A'
'Lila'	'A-'

'Hosam'	'B'
'Mona'	'A+'
'Ali'	'A'
'Lila'	'A-'
'Mohab'	'A'

Dictionary Operations

```
grades = {'Hosam': 'B', 'Mona': 'A+', 'Ali': 'A', 'lila': 'A-'}
```

'Hosam'	'B'
'Mona'	'A+'
'Ali'	'A'
'Lila'	'A-'

no guaranteed order

- get **iterable that acts like a tuple of all keys**
- `print (grades.keys())` → returns `(['Hosam', 'Mona', 'Ali', 'lila'])`
- get **iterable that acts like a tuple of all values**
- `print (grades.values())` → returns `(['B', 'A+', 'A', 'A-'])`

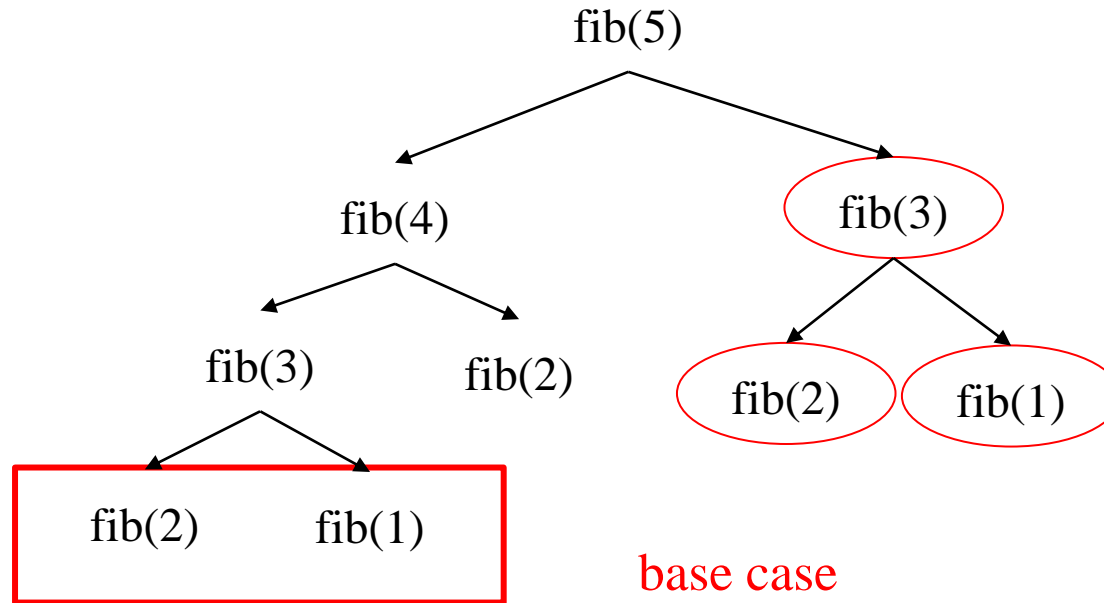
no guaranteed order

Dictionary Keys and Values

- values
 - any type (**immutable and mutable**)
 - can be duplicates
 - dictionary values can be lists, even other dictionaries!
- Keys
 - must be **unique**
 - **immutable** type (int, float, string, tuple, bool)
 - careful with float type as a key
- **no order** to keys or values!

```
d = {4:{1:0}, (1,3):"twelve", 'const':[3.14,2.7,8.44]}
```

Inefficient Fibonacci $\text{fib}(n) = \text{fib}(n-1) + \text{fib}(n-2)$



base case

Already
known
these value

- **recalculating** the same values many times!
- could keep **track** of already calculated values

Fibonacci With A Dictionary

```
def fib_efficient(n, d):  
    if n in d:  
        return d[n]  
    else:  
        ans = fib_efficient(n-1, d) + fib_efficient(n-2, d)  
        d[n] = ans  
        return ans  
d = {1:1, 2:2}  
print(fib_efficient(6, d))
```

Method sometimes called memorization

Initialize dictionary with base cases

- Do a **lookup first** in case already calculated the value
- **Modify dictionary** as progress through function calls

Efficiency Gains

- Calling `fib(34)` results in 11,405,773 recursive calls to the procedure.
- Calling `fib_efficient(34)` results in 87 recursive calls to the procedure.
- Using dictionaries to capture intermediate results can be very efficient.

Thank You