DS5010

Intro to programming for Data Science

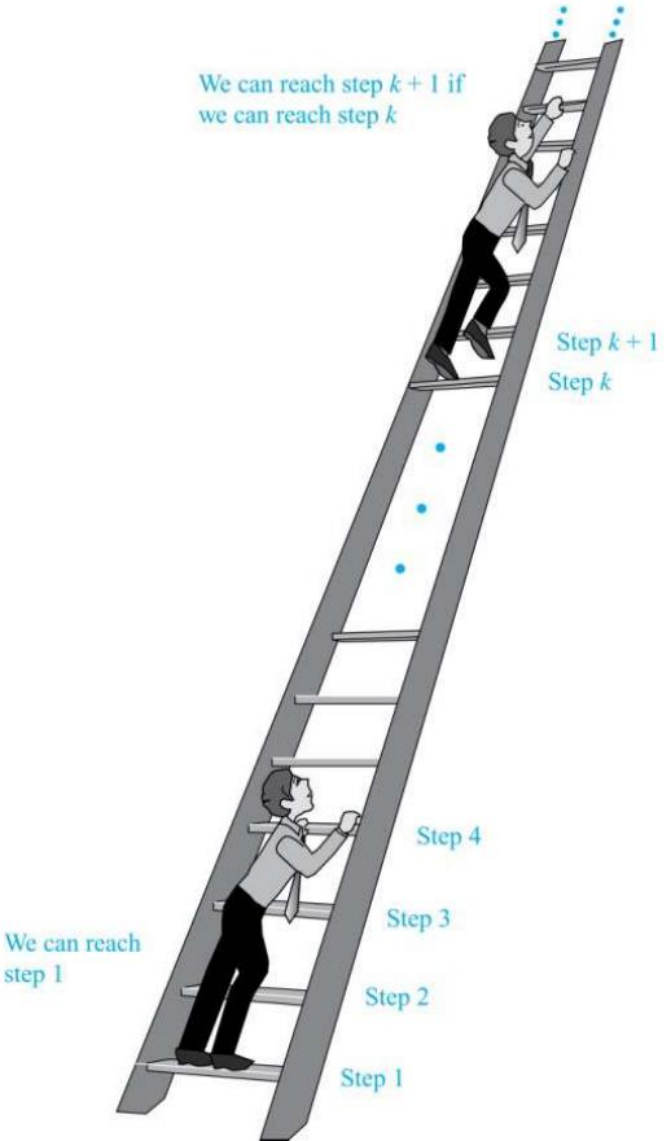
LECTURE 4

TODAY

- review session 3
- talk about induction
- introduce recursion
- solving problems with recursion and divide and conquer technique
- higher order functions and lambda expressions

REVIEW

- permutations, combinations
- functions
- scopes
- modules



MATHEMATICAL INDUCTION

Mathematical induction proves that we can climb as high as we like on a ladder, by proving that we can climb onto the bottom rung (the **basis**) and that from each rung we can climb up to the next one (the **step**).

— *Concrete Mathematics, page 3 margins.*

MATHEMATICAL INDUCTION

Problem:

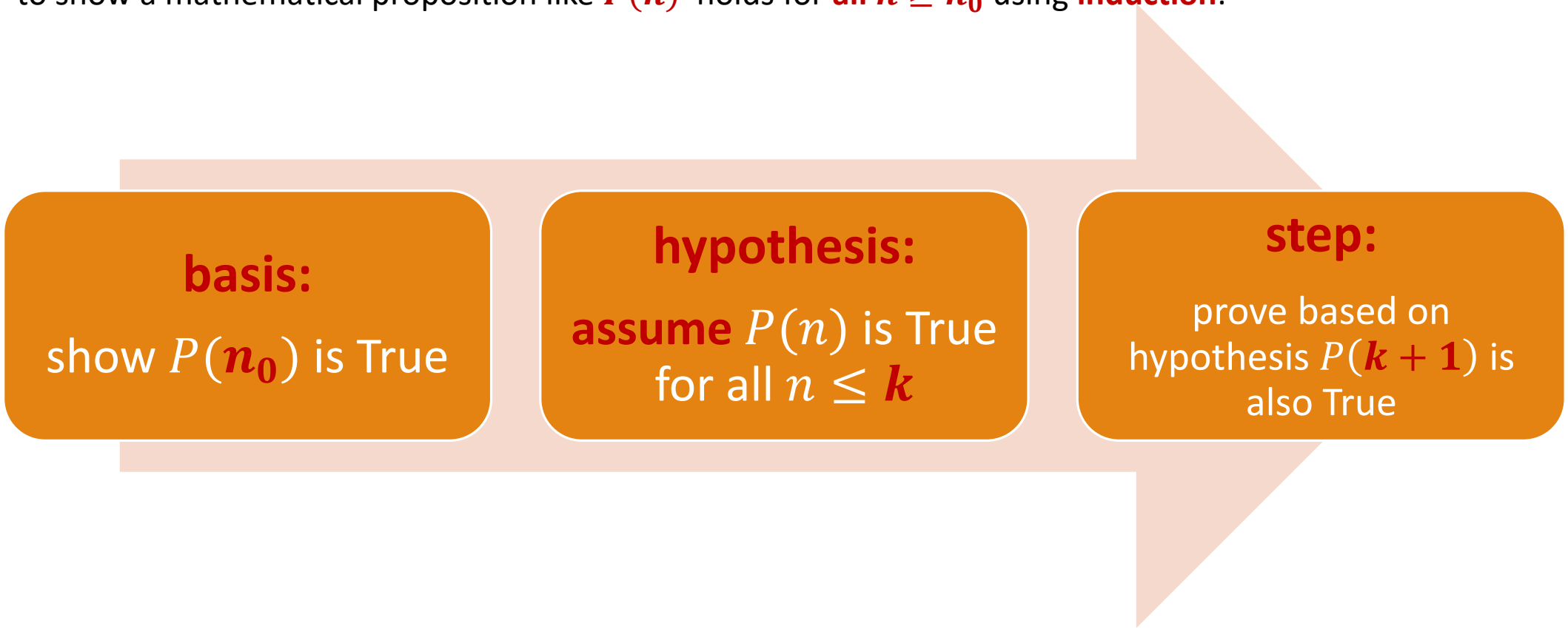
Some one says for all positive integer $\boldsymbol{n}$ we have:

$$1 + 2 + 3 + \ \ldots + n = \frac{n(n+1)}{2}$$

how can you **prove** this equality holds for all positive integers?

MATHEMATICAL INDUCTION

to show a mathematical proposition like $\boldsymbol{P(n)}$ holds for **all** $\boldsymbol{n \geq n_0}$ using **induction**:



MATHEMATICAL INDUCTION

Problem:

Some one says for all positive integer $\boldsymbol{n \geq 5}$ we have:

$$2^n > n^2$$

use induction to **prove** this inequality holds for integers $n \geq 5$!

MATHEMATICAL INDUCTION

Problem:

Some one says for all positive integer $\boldsymbol{n}$ we have:

A convex $\boldsymbol{n}$–gon has $\dfrac{\boldsymbol{n(n-3)}}{\boldsymbol{2}}$ diagonals

use induction to **prove** this equality holds for all positive integers!

RECURSION

- **Recursion** (adjective: *recursive*) occurs when a thing is defined in terms of **itself** or of its type.
- it is closely tied to induction!

RECURSIVE DEFINTION

- a definition which is defined in terms of itself!!!
- examples:
  - what is **GNU**? **G**NU is **N**ot **U**nix!
  -

```
def f(x):
    y = x + f(x)
    return y
```

we are **recurring** to **f** itself in its very own definition!

WHY DO WE NEED THESE CLUNKY DEFINITIONS?

- recursive definitions are everywhere!

the set of natural numbers $\mathbb{N}$:

1) $1 \in \mathbb{N}$

2) $n \in \mathbb{N}$ if $n - 1 \in \mathbb{N}$

the set of **prime numbers**:

1 is not a **prime** number

n is a **prime** number if and only if it is not divisible by any smaller **prime** numbers.

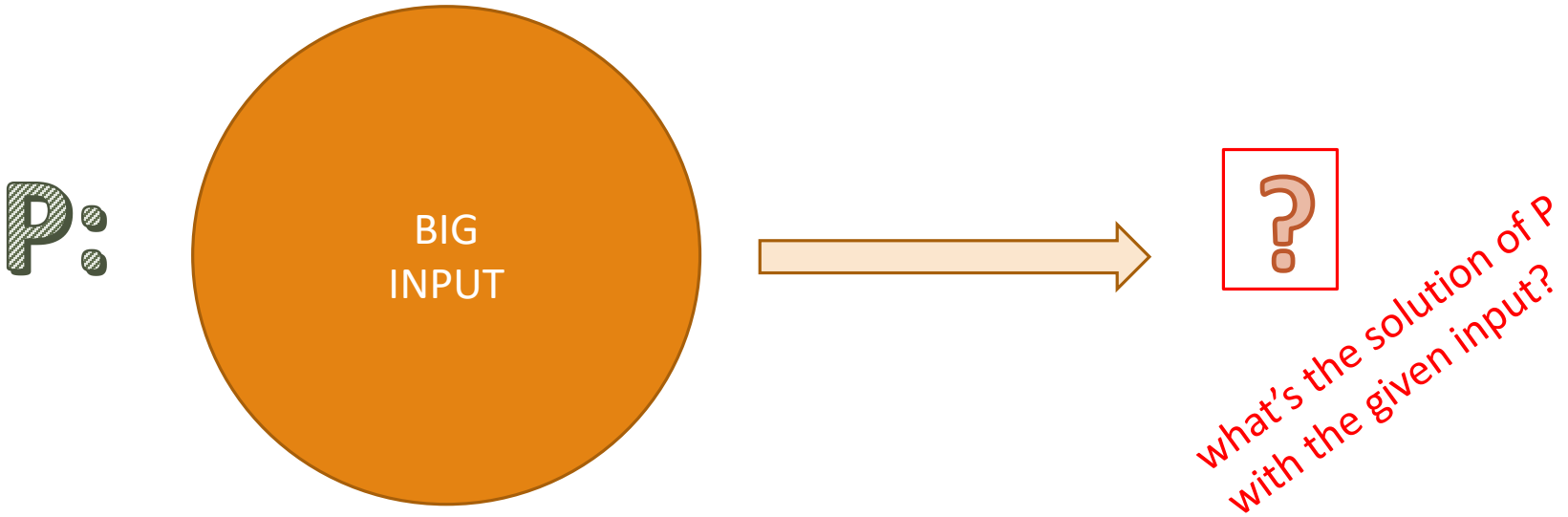
a to the **power** of b (a^b):

if $b = 0$, then it is 1. otherwise it is the multiplication of a to the **power** of $b - 1$ multiplied by a. $(a^{b-1} \times a)$

- recursion usually gives us a **very simpler** way of defining things.

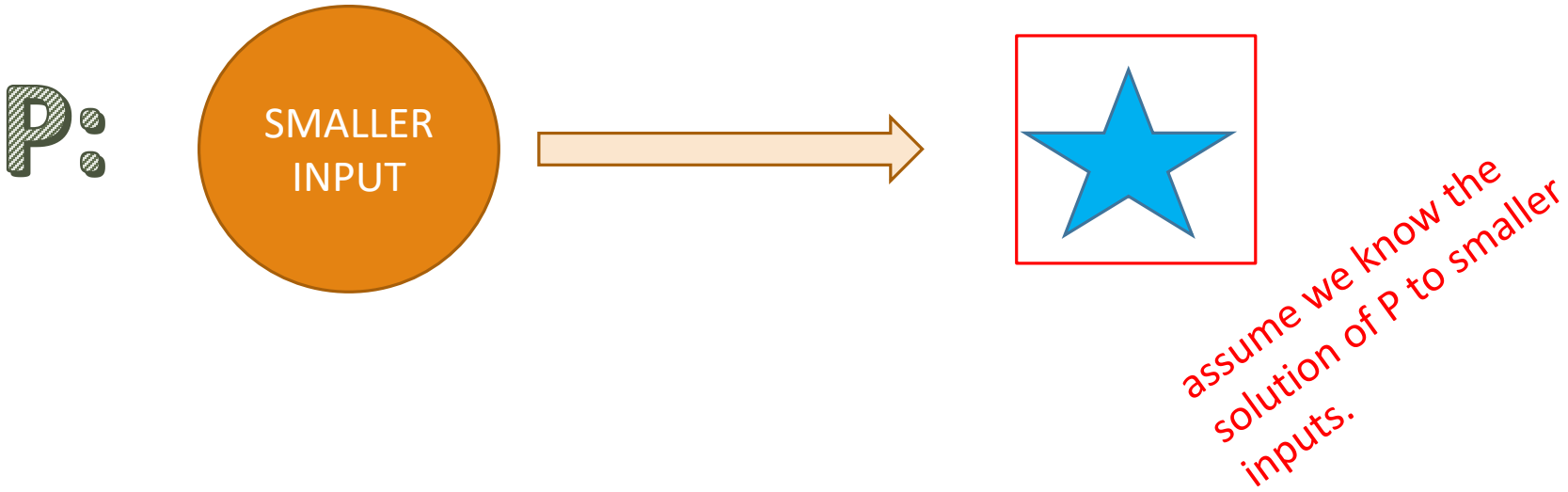
RECURSIVE PROBLEM SOLVING

- a problem like P can be **solved recursively** if:
  - its solution can be defined recursively in terms of **P itself**! (what does this mean?!!)
  - there are some simple inputs for which the solution is straight forward. (**base cases**)
- this is problem P:



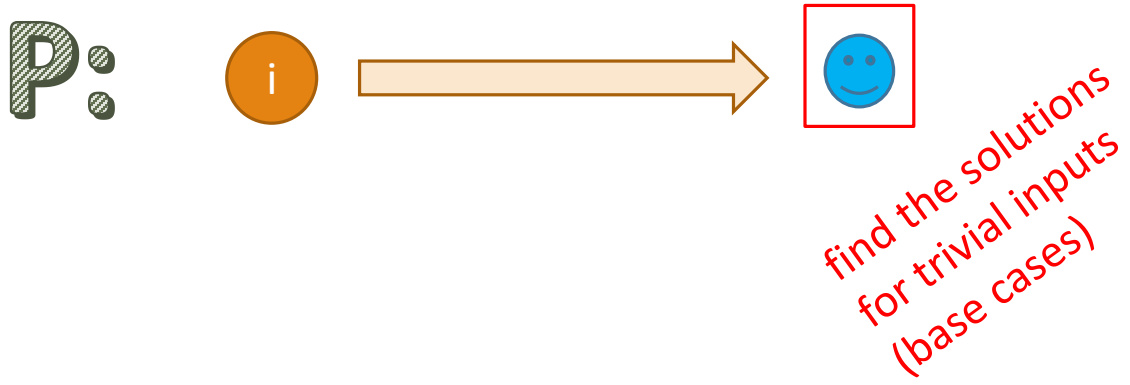
RECURSIVE PROBLEM SOLVING

- a problem like P can be **solved recursively** if:
  - its solution can be defined recursively in terms of **P itself**! (what does this mean?!!)
  - there are some simple inputs for which the solution is straight forward. (**base cases**)
- we **assume** we know all the **solutions to smaller inputs:**



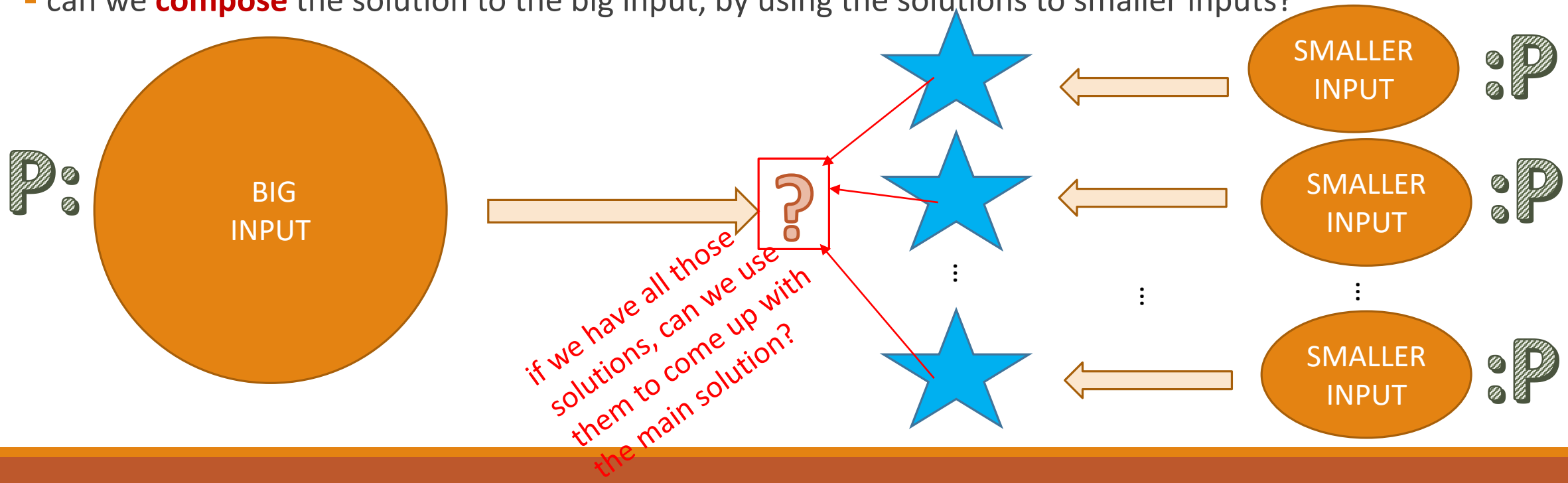
RECURSIVE PROBLEM SOLVING

- a problem like P can be **solved recursively** if:
  - its solution can be defined recursively in terms of **P itself**! (what does this mean?!!)
  - there are some simple inputs for which the solution is straight forward. (**base cases**)
- we **make sure** P has a trivial solution for some very small inputs!



RECURSIVE PROBLEM SOLVING

- a problem like P can be **solved recursively** if:
  - its solution can be defined recursively in terms of **P itself**! (what does this mean?!!)
  - there are some simple inputs for which the solution is straight forward. (**base cases**)
- can we **compose** the solution to the big input, by using the solutions to smaller inputs?



RECURSION vs INDUCTION

- what we just saw was very similar to induction.

➢ find out the solutions to P when input is trivial (**base cases**).	➢ write the **induction basis** for the proposition when $n = n_0$.
➢ **assume** you have the solutions to P for all **smaller inputs**.	➢ **Induction Hypothesis**. Assume the proposition holds for $n \leq k$.
➢ show P can be solved by **composing** the solutions to smaller inputs.	➢ **Induction Step**. Show the proposition holds for $n = k + 1$ by using the induction hypothesis.

RECURSIVE PROBLEM SOLVING

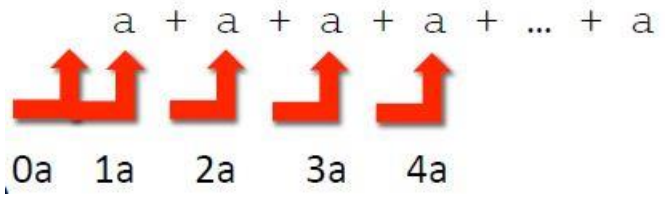
- Algorithmically: a way to design solutions to problems by **divide-and-conquer** or **decrease-and-conquer**
  - reduce a problem to **simpler** versions of the same problem
- Semantically: a programming technique where a **function calls itself**
  - in programming, goal is to NOT have infinite recursion
  - 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 $a \times b$" is equivalent to "add a to itself b times"
- capture **state** by
  - an **iteration** number (`i`) starts at `b`
    `i ← i-1` and stop when 0



- a current **value of computation** (`result`)
  `result ← result + a`

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

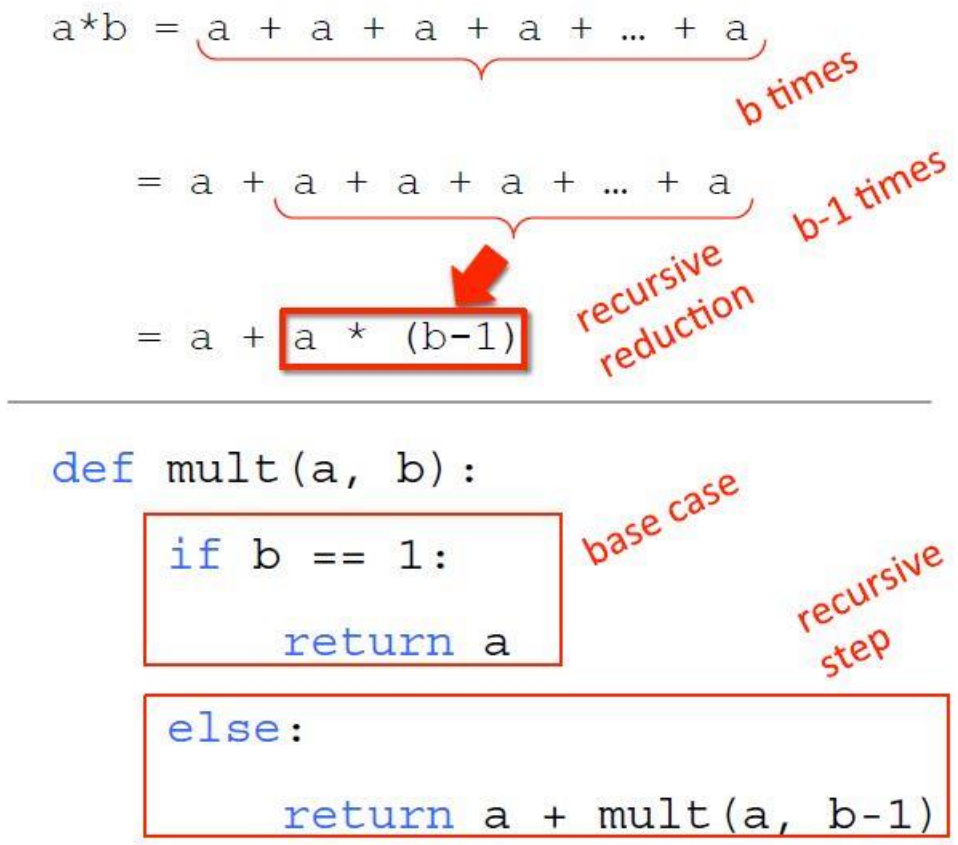
iteration

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



FACTORIAL

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

- for what n do we know the factorial trivially?

```
n = 1        →    if n == 1:
                      return 1
```

base case

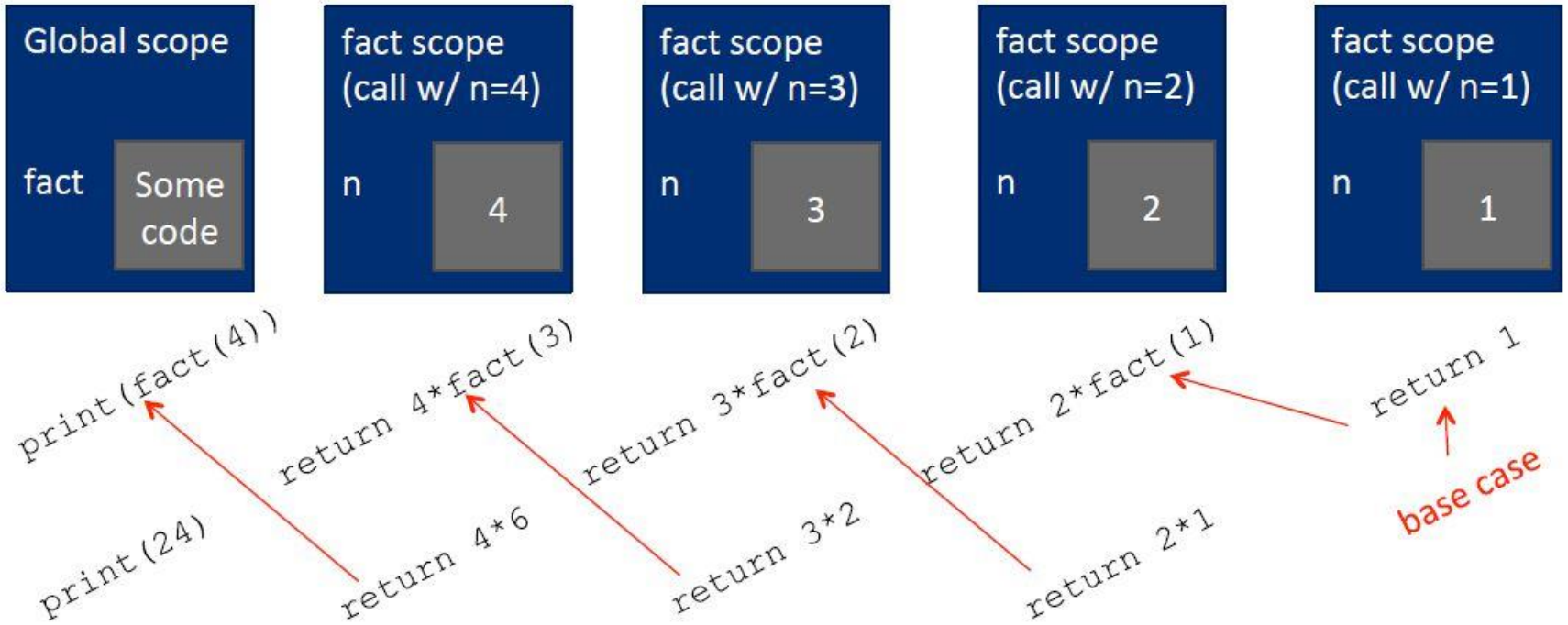
- how to reduce problem? Rewrite in terms of something simpler to reach base case

```
n*(n-1)!     →    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))
```



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

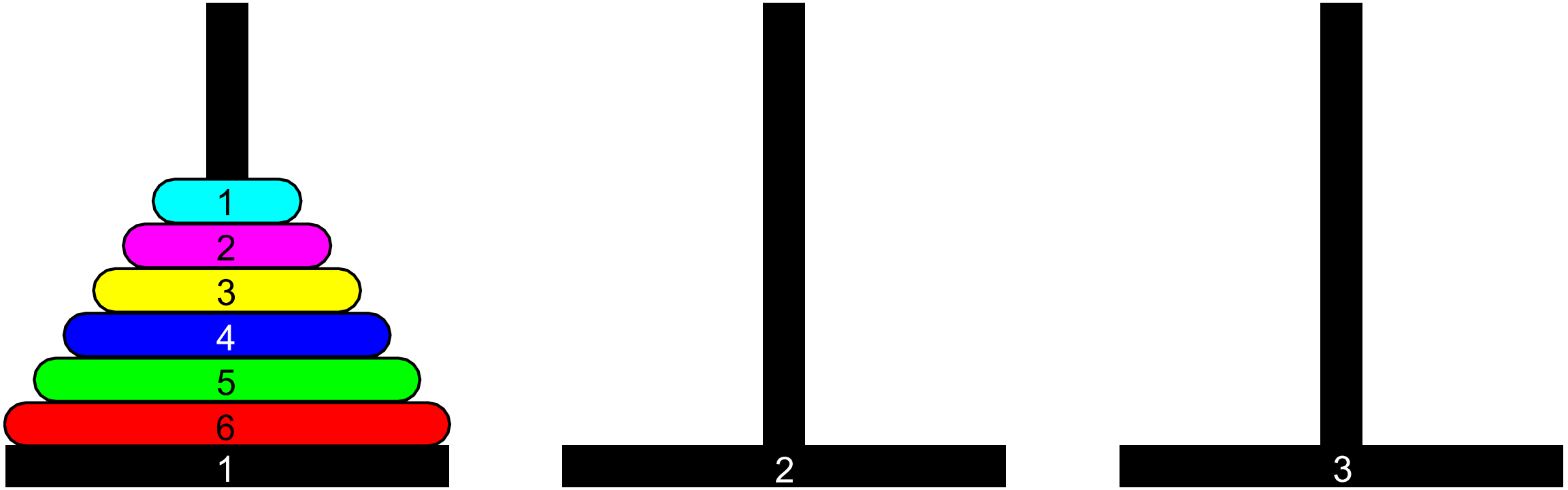
using the same variable names but they are different objects in separate scopes

ITERATION vs. RECURSION

```
def factorial_iter(n):
    prod = 1
    for i in range(1, n + 1):
        prod *= i
    return prod
```

```
def factorial_recur(n):
    if n == 1:
        return 1
    return n* factorial_recur(n-1)
```

- recursion may be simpler, more intuitive
- recursion may be efficient from programmer POV
- recursion may not be efficient from computer POV



TOWERS OF HANOI

TOWERS OF HANOI

The story:

- 3 tall spikes
- Stack of 64 different sized discs – start on one spike
- Need to move stack to third spike (at which point universe ends!!)
- Can only move one disc at a time, and a larger disc can never cover up a small disc

TOWERS OF HANOI

for cool animated examples, refer to http://towersofhanoi.info/Animate.aspx

TOWERS OF HANOI

- Having seen a set of examples of different sized stacks, how would you write a program to print out the right set of moves for **n** disks?
- **Think recursively!**
  - assume you know how to move **n-1** disks correctly.
  - you know the base case where you only have to move **one** disk
  - now how do you compose a solution for **n** disks?

TOWERS OF HANOI

```
def printMove(fr, to):
    print('move from ' + str(fr) + ' to ' + str(to))


def Towers(n, fr, to, spare):
    if n == 1:
        printMove(fr, to)
    else:
        Towers(n-1, fr, spare, to)
        Towers(1, fr, to, spare)
        Towers(n-1, spare, to, fr)
```

RECURSION WITH MULTIPLE BASE CASES

- Fibonacci numbers

the Fibonacci Sequence is the series of numbers:

1, 1, 2, 3, 5, 8, 13, 21, 34, ...

each number in the sequence drives from the addition of its two predecessor elements.

$$F(n) = F(n-1) + F(n-2)$$

and we have two base cases: $F(0) = F(1) = 1$

FIBONACCI

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

RECURSION ON NON-NUMERICS

problem:

you are given a string like `s`, write a recursive function which returns the reverse of `s`.

ex: `s="abcde" → "edcba"`

REVERSE STRING

```
def rev_recur(s):
    if len(s) <= 1:
        return s

    return s[-1] + rev_recur(s[1:-1]) + s[0]
```

DIVIDE AND CONQUER

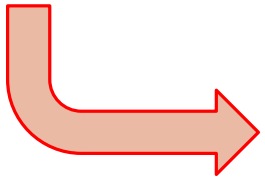
- an example of a "**divide and conquer**" algorithm
- solve a hard problem by **breaking** it into a set of sub-problems such that:
  - sub-problems are easier to solve than the original
  - solutions of the sub-problems can be combined to solve the original

MERGE SORT

Problem:

Given an array of integers, sort the elements in increasing order.

L = [7, 5, 2, 4, 1, 6, 3, 0]



L = [0, 1, 2, 3, 4, 5, 6, 7]

THINK RECURSIVELY

- the **base case** is straight-forward. If our list only has one element, then that list is already sorted.
- assuming to have the solution for smaller sub-problems.
  - if we divide the list in half, then we have two smaller sub-lists (say L1 and L2), for which we assume we know their sorted version.
- how can we use the sorted version of L1 and L2, to come up with the sorted version of L?

MERGE SORT

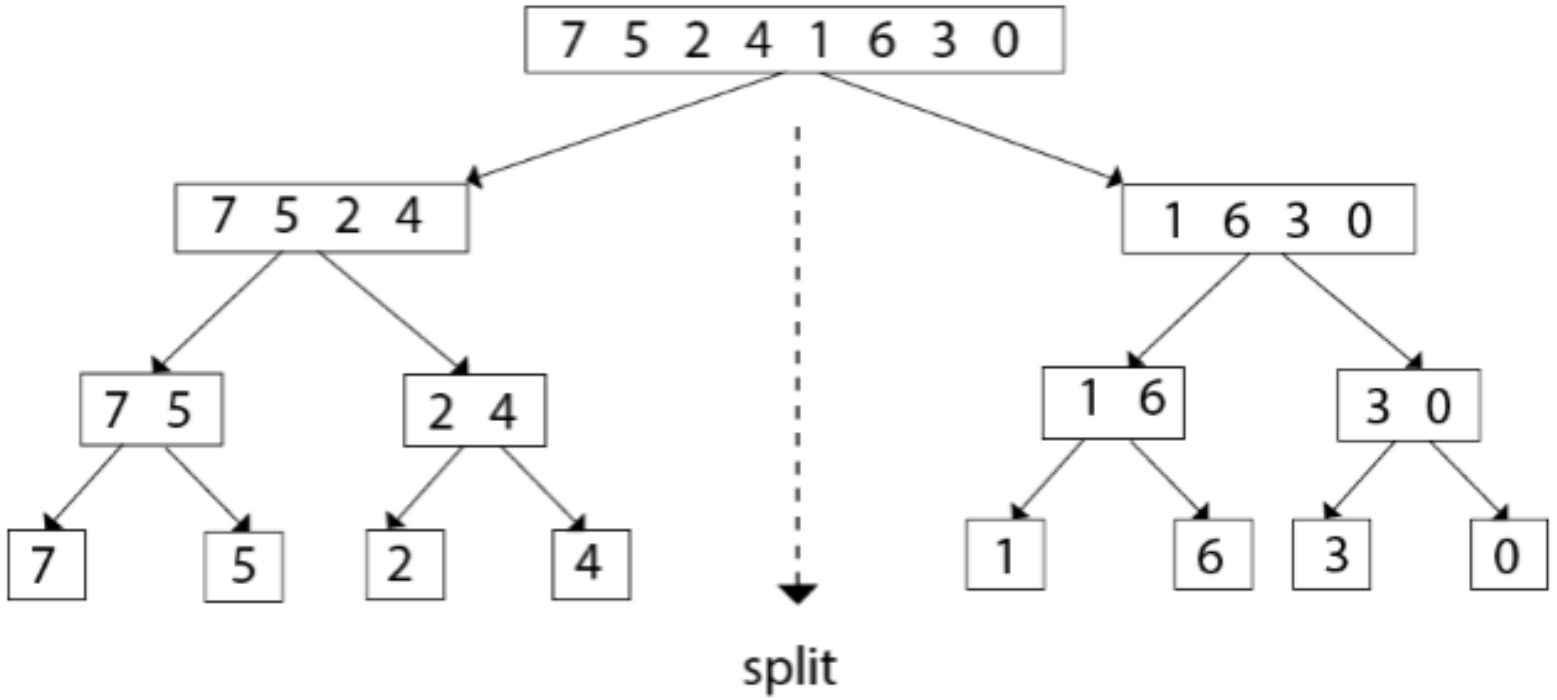


Figure: Merge sort divide phase

MERGE SORT

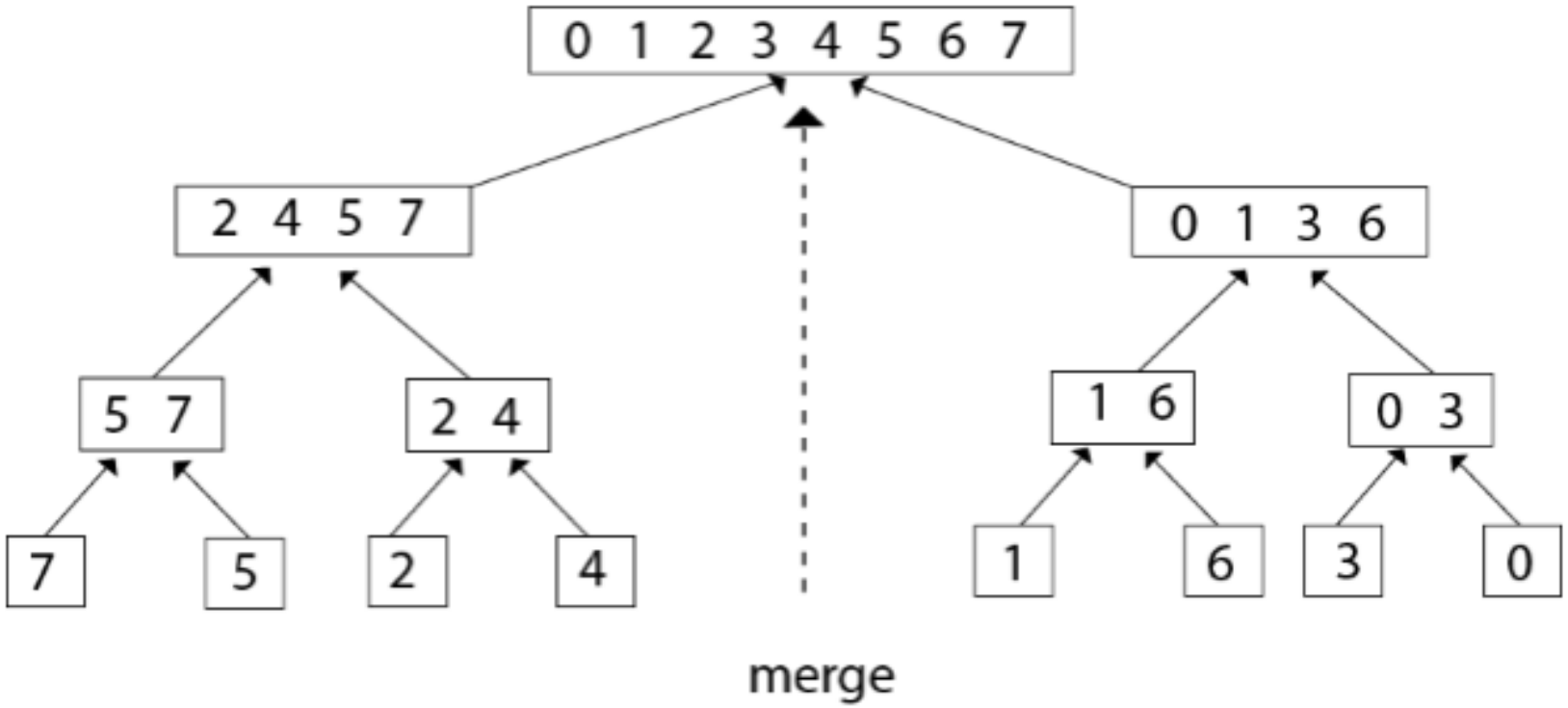


Fig: Merge sort: combine phase

MERGE SORT

```
def merge_sort(L):
    if len(L) <= 1:
        return L

    mid = len(L)//2

    L1 = merge_sort(L[:mid])
    L2 = merge_sort(L[mid:])

    return merge(L1, L2)
```

base case

solving subproblems recursively

combining solutions to subproblem to come up with solution to main problem

MERGE

```
def merge(L1, L2):
    i = j = 0
    m, n = len(L1), len(L2)
    result = []

    while i + j < m + n:
        if i < m and j < n:
            if L1[i] < L2[j]:
                result.append(L1[i])
                i += 1
            else:
                result.append(L2[j])
                j += 1
        elif i < m:
            result.append(L1[i])
            i += 1
        else:
            result.append(L2[j])
            j += 1

    return result
```

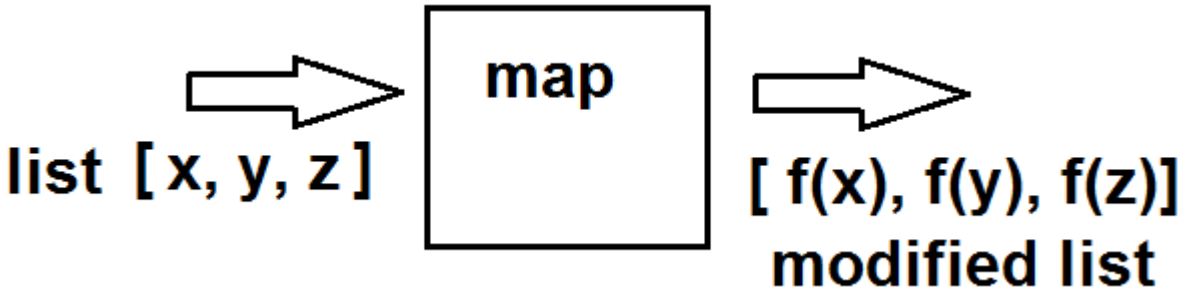
HIGHER ORDER FUNCTIONS

- a higher-order function is a function that takes another function as a parameter
- they are “higher-order” because it’s a function of a function
- Examples
  - map
  - filter
  - reduce

MAP

```
map(function, iterable)
```

- map applies **function** to each element of **iterable** and creates a list of the results
- map returns an iterator which can be cast to list

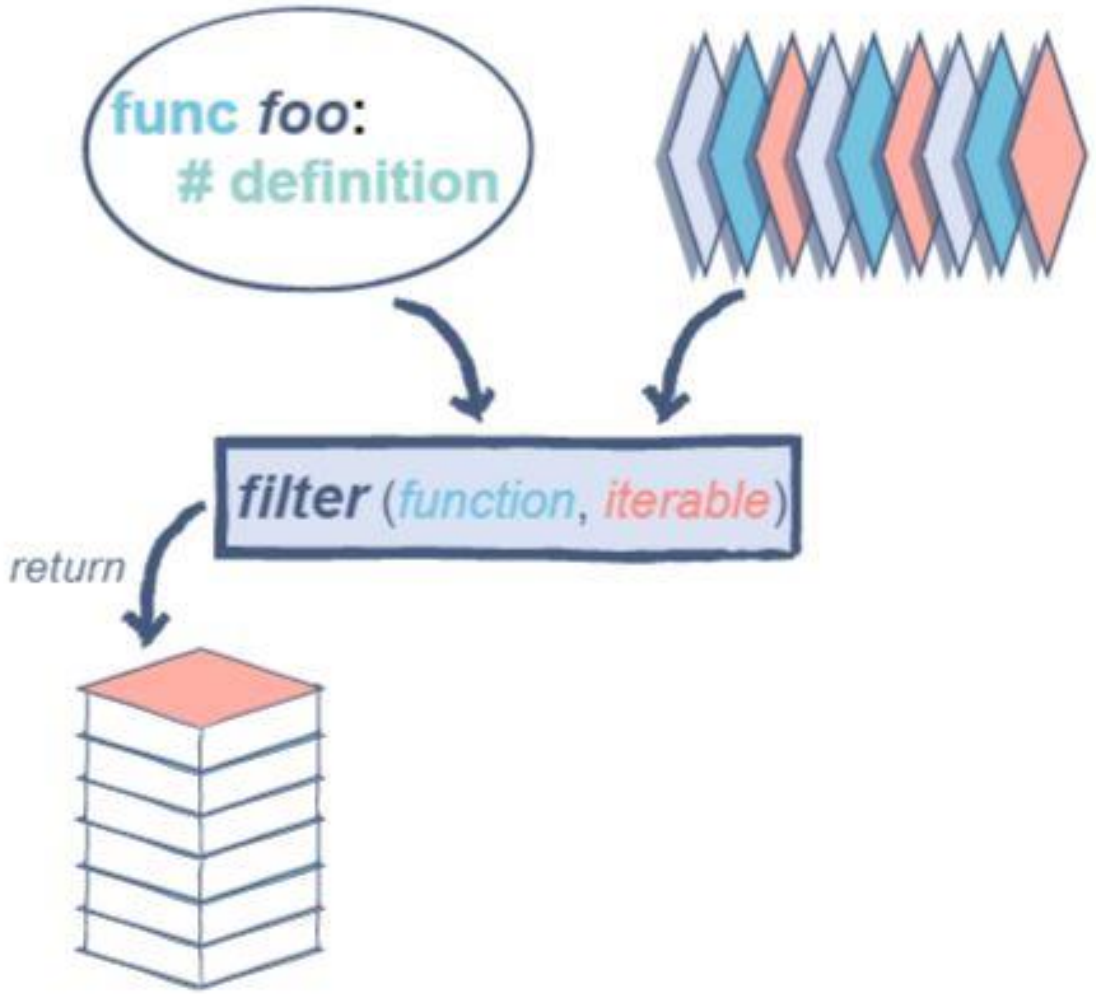


MAP EXAMPLE

```
def func(i):
    return i**2

L = [1, 2, 3]

L_mapped = map(func, L)

print(list(L_mapped))
```

applies the given **func** on each element in the given **iterable**

prints [1, 4, 9]



FILTER

`filter(function, iterable)`

FILTER

- the filter runs through each element of **iterable** (any iterable object such as a `List` or another collection)
- it applies a **function** to each element of **iterable**
- if that **function** returns `True` for that element then the element is put into a `List`
- this list is returned from filter in versions of python under 3
- in python 3, filter returns an iterator which must be cast to type list with list()

FILTER EXAMPLE

```
def func(i):
    return i % 2 == 0

L = [1, 2, 3, 4, 5, 6, 7]

L_filtered = filter(func, L)

print(list(L_filtered))
```

applies the given func on each element. if func returns **True**, the element will be added to output list

prints [2, 4, 6]

LAMBDA EXPRESSIONS

```
lambda <params>: <some_expression>
```

- anonymous functions (functions without names)
- can have any number of parameters, each separated by a comma
- returns the value which is derived from <some_expression>

LAMBDA EXAMPLE

```
f = lambda x: x**2

r = f(4)

print(r)
```

function parameter

an expression which will be evaluated based on the value of the given parameters

prints 16

MAP EXAMPLE with LAMBDA

```
L = [1, 2, 3]

L_mapped = map(lambda x: x**2, L)

print(list(L_mapped))
```

define and pass the mapping function at the same time

prints [1, 4, 9]

used when mapping function is so simple that the return value is just an expression made from given parameters

FILTER EXAMPLE with LAMBDA

```
L = [1, 2, 3, 4, 5, 6, 7]

L_filtered = filter(lambda i: i%2 == 0, L)

print(list(L_filtered))
```

define and pass the filtering function at the same time

prints [2, 4, 6]

used when filtering function is so simple that the return value is just an expression made from given parameters

LAMBDA with MORE THAN ONE PARAMS

```
f = lambda x, y: x + y

print(f(4,5))  # pinrts 9
```

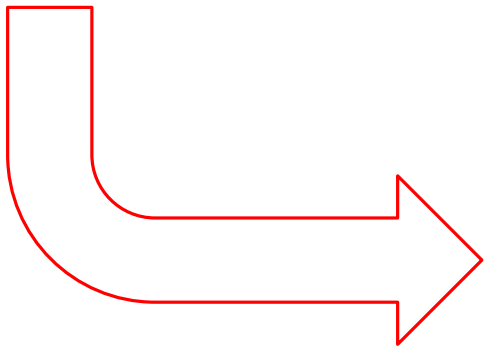
MAP with MORE THAN ONE ITERABLE

- map can get **one or more** iterables as argument
- the mapping function will map the **i'th elements** of the iterables to one element.
- example:

```
map(lambda x, y: x+y, [1,2,3], [10,20,30])
```

first iterable

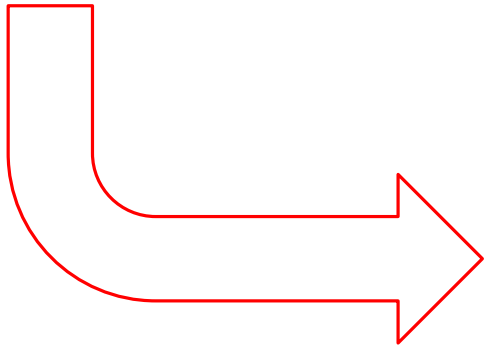
second iterable



maps the two iterables to
[11,22,33]

EXAMPLE: EUCLIDEAN DISTANCE in ONE LINE

```
dist = lambda p, q: math.sqrt(sum(map(lambda x, y: (x-y)**2, p, q)))

p1 = (1, 2, 3)
p2 = (4, 5, 6)

print(dist(p1, p2))
```



prints 5.196152422706632