Programming in Python
Lecture 4- Functions

Plan for today

- Functions
  - What are they good for
  - Built-in Functions
  - Defining New Functions
  - Functions call functions
- Lambda
- Recursion

Reminder

- If statements
- For loop
- While loop

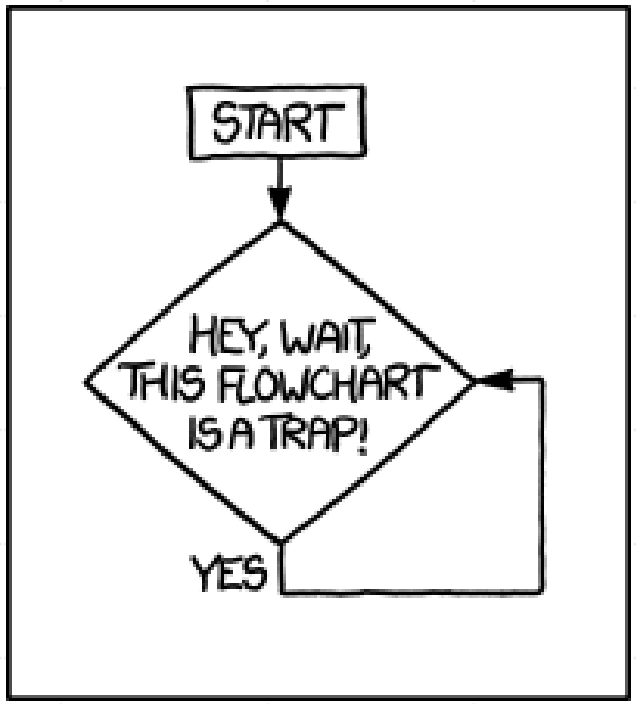
Flow Control

Different inputs →Different execution order

- Computer games
- Illegal input

Control structures

- **if-else**
- **for loop**
- **while** loop



http://xkcd.com/1195/

Conditional Statement: if

Used to execute statements conditionally

Syntax

if *condition:*

statement1

statement2

*If condition is **True**, statements are executed*

***Condition** = expression that evaluates to a Boolean*

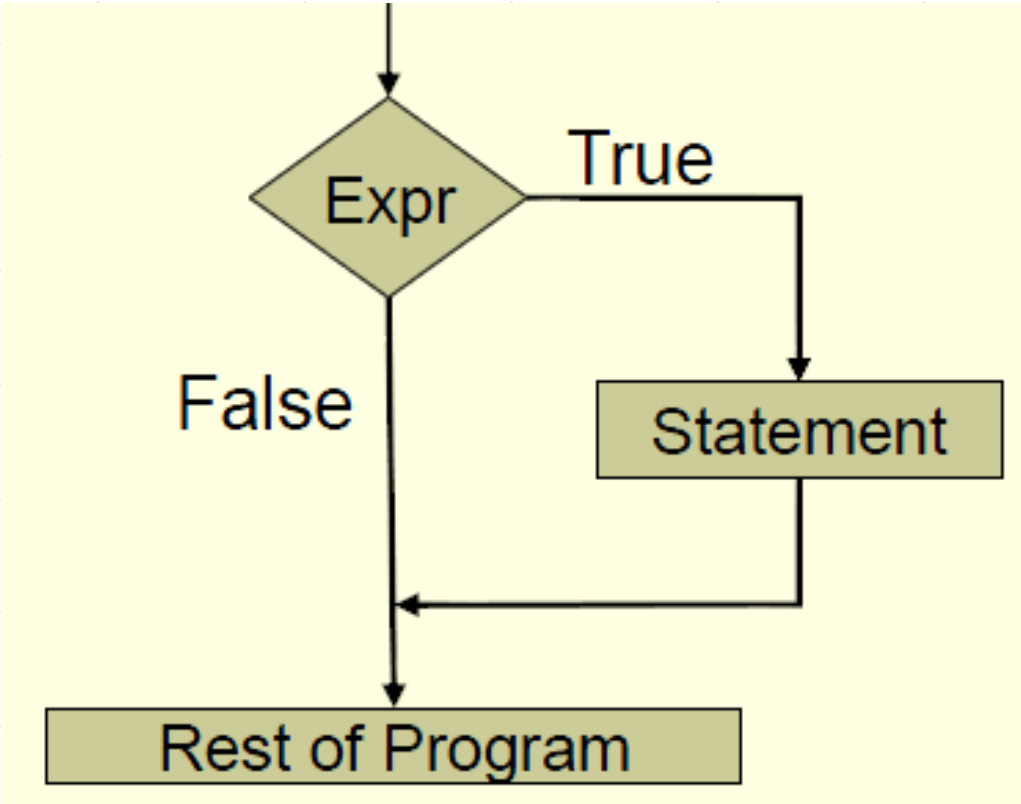
Indentation:

Following the if statement:

Open a new scope = one tab to the right.

Indicates the commands within the scope of this if.

Conditional Statements



elif

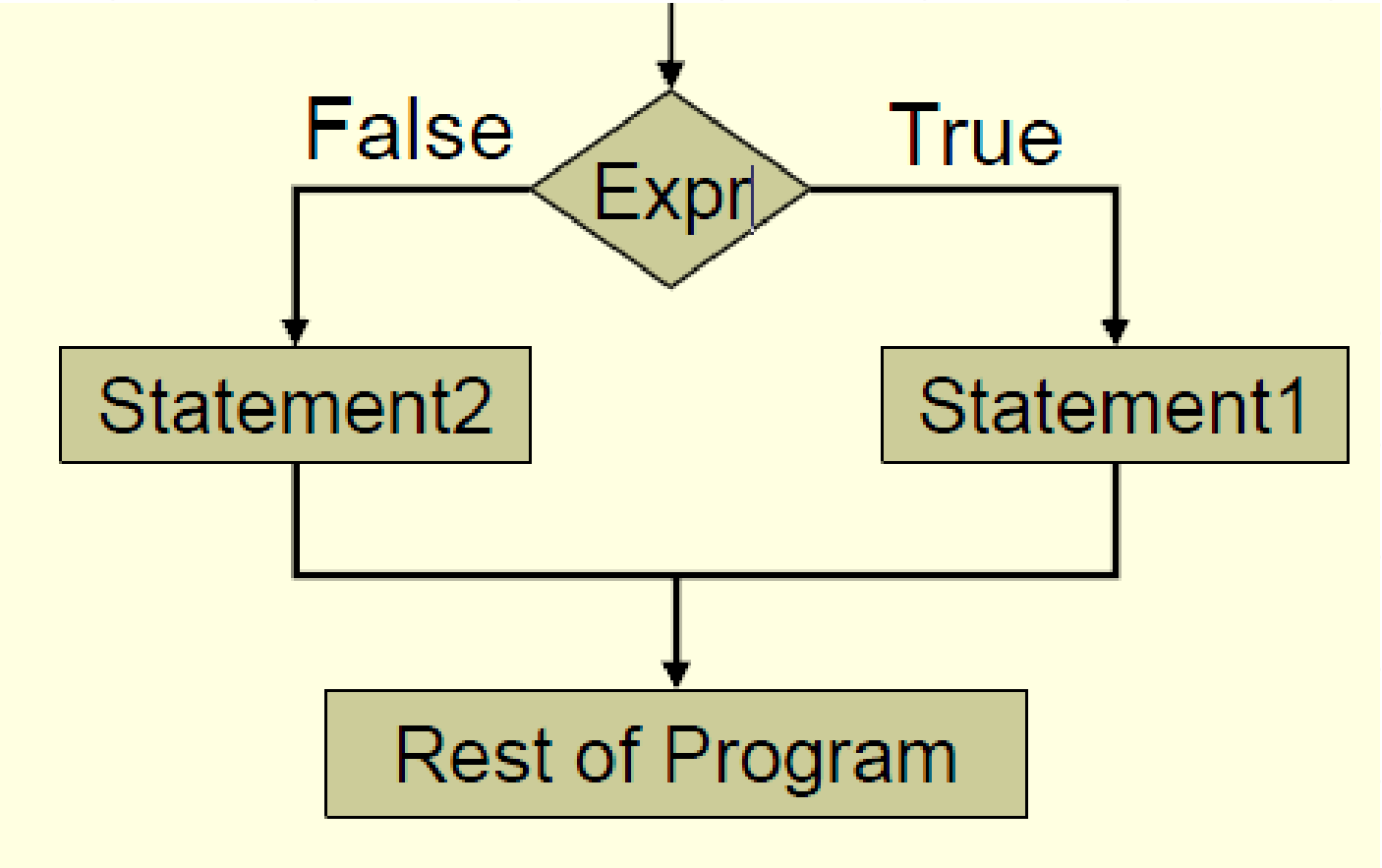
```
if  condition1:
   statement1
elif condition2:
   statement2
else:
   statement3
```

condition1 is true → execute ***statement1***

condition1 false and **condition2** true → execute ***statement2***

both conditions are false → execute ***statement3***

elif



For Loop

```
for element in iterable:
  statement1
  statement2
    ...
```

Run over all elements in the object (list, string, etc.)

Iteration 0: Assign element = object[0]

- Execute the statements

Iteration 1: Assign *element* = object*[1]*

- Execute the statements

…

Range

An ordered list of integers in the range.

```
>>> range(10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

range(from, to) contains all integers k satisfying from ≤ k < to.

range(to) is a shorthand for range(0, to).

```
>>> range(2,10)
[ 2, 3, 4, 5, 6, 7, 8, 9]
>>> range(-2,2)
[-2, -1, 0, 1]
>>> range(4,2)
[]
```

Range

```
>>> type(range(3))
<type 'list'>
```

Step size:

`range(from, to, step)` returns:

from, *from*+*step*, *from*+2**step*,…, *from*+i**step*

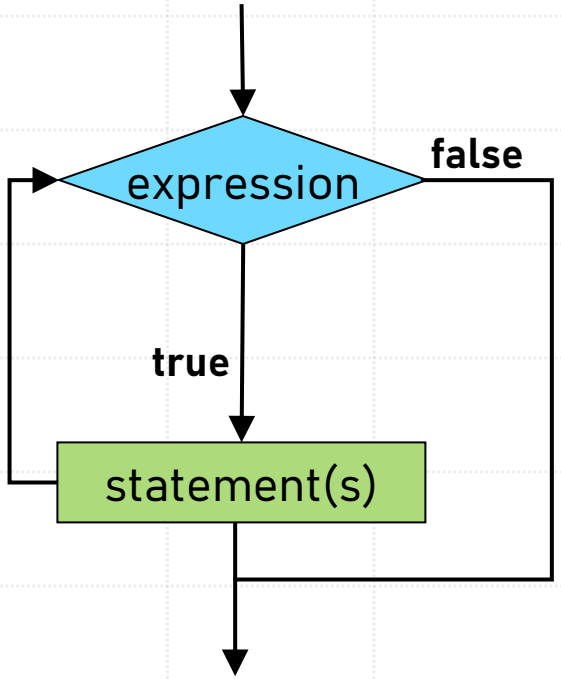
until *to* is reached, not including to itself.

```
>>> range(0,10,2)
[0, 2, 4, 6, 8]
>>> range(10,0,-2)
[10, 8, 6, 4, 2]
```

While Loop

Used to repeat the same instructions until a stop criterion is met

```
while expression:
    statement1
    statement2
    ...
```



Homework

1. Create a Python program to get the largest/smallest number from a list

Input:

```
>>> s=[2,5,7,3,4,6]
```

Output:

```
>>>  The max number is: 7
>>>  The min number is: 2
```

2. Create a Python program to count the number of strings where the string length is 2 or more and the first and last character are same from a given list of strings

Input:

```
>>> words=['drd','1435','savg','11','sys','1321','10934','121']
```

Output:

```
>>> ['drd', 'sys', '1321', '121']
```

Homework

3. Create a Python program to count all the names that start with "M" from a given list

Input:

```
>>>  names = ['Mor','Yuval', 'Many','Eli','Moshe']
```

Output:

```
>>>  3
```

4. Create a Python script to calculate to price of Apple, Milk and Meat

Input:

```
>>>  Supermerket_list={"Apple":10,"Eggs":5,"Milk":5,"Bread":3,"Meat":20}
```

Output:

```
>>>  35
```

Plan for today

- **Functions**
  - **What are they good for**
  - Built-in Functions
  - Defining New Functions
  - Functions call functions
- Lambda
- Recursion

What is a **function** in programming ?

- A function is a block of organized, reusable code that is used to perform a single, well defined action.
- Functions provide better modularity for your application and a high degree of code reusing

How to Calculate 4! + 7! + 9!?

```
factorial4 = 1
factorial7 = 1
factorial9 = 1
for i in range(1, 5):
    factorial4 *= i
for i in range(1, 8):
    factorial7 *= i
for i in range(1, 10):
    factorial9 *= i
print ("4!+7!+9!=", factorial4 + factorial7 + factorial9)
```

Why use functions ?

- **We** wrote code calculating $4! + 7! + 9!$.

To use the code in 3 different calculations, he:

- copy & pasted
- assigned arguments

3 times.

The calculation took 0.0046 microseconds.

- There is a more efficient algorithm to calculate $4! + 7! + 9!$.
- To update the code, he went over the 3 calculations ☹
- After the update, the calculation took only 0.0006 microseconds – over 7 times faster!

→ Don’t duplicate code, use functions !

How to Calculate 4! + 7! + 9!?

```
def factorial(n):

    fact = 1
    for i in range(1,n+1):
        fact *= i
    return fact

print ('4!+7!+9!=', factorial(4) + factorial(7) + factorial(9))
```

Modularity enables code reuse !

Definition

Modularity is the degree to which a system's components may be separated and recombined (Wikipedia).

- Top-down design
- Improves maintainability
- Enforces logical boundaries between components

Scope of a function

- Variables defined within a function are considered **local**, and can be used only within the function's block of code.
- Local variables can mask variables with the same name defined outside of a function (**Global** variables).

So – Why use functions?

- **Modularity** - Break a task into smaller sub-tasks (divide and conquer), enables code reuse
- **Abstraction** – Solve each problem once and wrap it in a function
- **Maintenance** - Solve bugs once
- **Readability** – Main code is shorter, implementation details are hidden within functions
- **Limited Variable Scope** - Temporary variables are restricted to the function's scope

Plan for today

- **Functions**
  - **What are they good for**
  - **Built-in Functions**
  - Defining New Functions
  - Functions call functions
- Lambda
- Recursion

Built-in Functions

http://docs.python.org/library/functions.html

We already used built-in functions:

```
>>> type(5)
<type 'int'>

>>> len(range(8,100, 8))
12
```

Built-in Functions

Conversion functions:

```
>>>5.__str__
'5'
>>> 3.2.__int__
3
>>>’3.14’.__float__
3.14
```

Plan for today

- **Functions**
  - **What are they good for**
  - **Built-in Functions**
  - **Defining New Functions**
  - Functions call functions
- Lambda
- Recursion

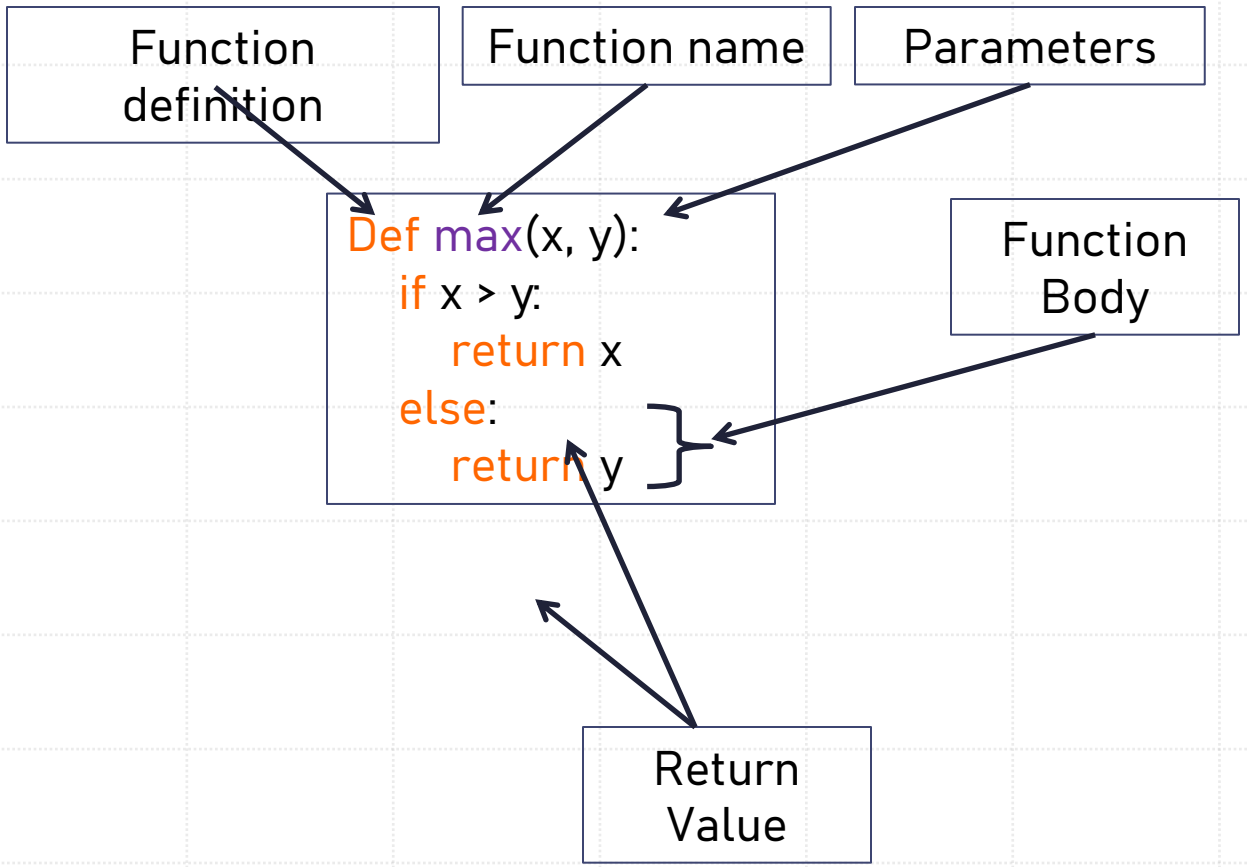
Functions

```
def function_name(argument1, argument2,...):
    statement1
    statement2
    …
    return result1, result2, … # optional, returns the
                               constant None if not
                               specified
```

Calling a function:

```
var1, var2,… = function_name(val1, val2,...)
```

Function Definition in Python



Function’s Input / Output

Input: Arguments

Can be of any type - int / float / str / Boolean / list

Output: The *Return* statement

Returns a value to the function caller.

- Returned value can be any Python type
  - Multiple values are wrapped in list
- If no *Return* or no values is specified - returns *None*
- **Different from print**
- ***Return* stops the function’s execution and returns to the caller**

Back to the factorial example…

Avoiding Code Duplication

Calculate 4! + 7! + 9! using a function:

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


print "4!+7!+9!=", factorial(4) + factorial(7) + factorial(9)
```

Defining a new function

Programming Style

- Comments: #
- Meaningful variables names

Why is it important?

Example - Palindromes

Palindromes are read the same way in either direction.

Examples

- 21.11.12
- Alula
- Anna
- Deified

Example - Palindromes

Pseudo-code

Translate a verbal description to code.

For every index in the string:

Check if the i^{th} letter is equal to the $(n-i)^{th}$ letter

If not return False (this is not a palindrome)

If all indexes were checked – this is a palindrome

Palindromes – Code

```
def is_palindrome(text):
    for i in range(len(text) / 2):
        if text[i] != text[- i - 1]:
            return False
    return True


def is_palindrome(text):
    return text == text[::-1]
```

Passing Arguments to Functions

In a function call, **before** execution:

argument **values** are assigned to function arguments **by order**.

```
calculator(2, 3, '*')

def calculator(x, y, op):
    if op == '+':
        return x+y
    elif …
    else:
        return None
```

Questions?

Hands On

Example I

```
def add(a, b):
    print "ADDING %d + %d" % (a, b)
    return a + b

def subtract(a, b):
    print "SUBTRACTING %d - %d" % (a, b)
    return a - b

def multiply(a, b):
    print "MULTIPLYING %d * %d" % (a, b)
    return a * b

def divide(a, b):
    print "DIVIDING %d / %d" % (a, b)
    return a / b

print "Let's do some math with just functions!"

age = add(30, 5)
height = subtract(78, 4)
weight = multiply(90, 2)
iq = divide(100, 2)

print "Age: %d, Height: %d, Weight: %d, IQ: %d" %
(age, height, weight, iq)
```

```
Let's do some math with just functions!
ADDING 30 + 5
SUBTRACTING 78 - 4
MULTIPLYING 90 * 2
DIVIDING 100 / 2
Age: 35, Height: 74, Weight: 180, IQ: 50
```

Example II "The output of one function, is the input of another!"

```
def add(a, b):
    print "ADDING %d + %d" % (a, b)
    return a + b

def subtract(a, b):
    print "SUBTRACTING %d - %d" % (a, b)
    return a - b

def multiply(a, b):
    print "MULTIPLYING %d * %d" % (a, b)
    return a * b

def divide(a, b):
    print "DIVIDING %d / %d" % (a, b)
    return a / b

print "Here is a puzzle."

what = add(age, subtract(height, multiply(weight, divide(iq, 2))))

print "That becomes: ", what
```

Here is a puzzle.
DIVIDING 50 / 2
MULTIPLYING 180 * 25
SUBTRACTING 74 - 4500
ADDING 35 + -4426
That becomes: -4391

Plan for today

- **Functions**
  - **What are they good for**
  - **Built-in Functions**
  - **Defining New Functions**
  - **Functions call functions**
- Lambda
- Recursion

Functions call functions

```
def print_text():
        print “Is this the real life”
        print “Is this the real life or it’s just fantasy”
def text_2():
        print_text()
        print_text()
>>> text_2()
Is this the real life
Is this the real life or it’s just fantasy
Is this the real life
Is this the real life or it’s just fantasy
```

Default Arguments For Functions

- We can specify default values for the arguments of the function.
- The default value will be used only when the function is called **without** specifying a value for the argument.

```
def f1(x, y=1):
    return x+y

>>> f1(1,2)    # In this call: x = 1, y = 2
3
>>> f1(3)    # In this call: x = 3, y = 1 (the default value)
4
>>> f1()    # x doesn't have a default value, it must be specified

Traceback (most recent call last):
  File "<pyshell#12>", line 1, in <module>
    f1()
TypeError: f1() takes at least 1 argument (0 given)
```

More about a function's scope

Consider the following function, operating on two arguments:

```
def linear_combination(x,y):
  y = 2 * y
  return x + y
```

The formal parameters x and y are local within the function's scope, and their "life time" is just the execution of the function. They disappear when the function is returned.

Local vs. Global variables

- A local variable is defined within a function, and exists only within that function.
- A global variable is defined in the main script (outside of any function) and it is known globally (including within functions).

Global variables can be accessed within functions.

Defining a variable by the same name will create a local variable hiding the global variable

Local variables exist only with the function in which they were defined

```
def myFunc():
    localVar = 5
    print localVar
    print globalVar


globalVar = 10
myFunc()
print globalVar+1
# print localVar  ### Error
```

The scope of variable 'localVar'

The scope of variable 'globalVar' is the entire program

```
5
10
11
```

Questions?

Hands On

Plan for today

- **Functions**
  - **What are they good for**
  - **Built-in Functions**
  - **Defining New Functions**
  - **Functions call functions**
- **Lambda**
- Recursion

Lambda

- Lambda is a special command that is used to quickly create in-line functions for specific usecases.
- The lambda operator or lambda function is a way to create small anonymous functions, i.e. functions without a name .
- These functions are throw-away functions, i.e. they are just needed where they have been created.

Lambda

- Example
- a simple function that returns a number multiplied by itself

```
def num_func (num):
    result = num ** 2
    return result
print(num_func(8))
>>>  64
num_lamb = lambda num: num ** 2
print(num_lamb(8))
>>>  64
```

- The basic structure of a lambda statement – lambda input: output

Lambda

- lambda expression can accept multiple parameters

```
bigger = lambda num1, num2: num1 > num2

print(bigger(100,67))

>>>  True

print(bigger(100,670))

>>>  False
```

Lambda

- lambda with if statements, for loop and while loop

```
starts_with = lambda x: True if x[0]=='P' else False

print(starts_with('Python'))

>>>  True

print(starts_with('Java'))

>>>  False
```

Lambda

- lambda with if statements, for loop and while loop

```
starts_with = lambda x: True if x[0]=='P' else False

print(starts_with('Python'))

>>> True

print(starts_with('Java'))

>>> False
```

Lambda

- Filter()-offers an elegant way to filter out all the elements of a sequence "sequence", for which the function function returns True

```
nums = [2, 4, -6, -9, 11, -12, 14, -5, 17]

total_negative_nums = list(filter(lambda n:n<0,nums))
total_positive_nums = list(filter(lambda p:p>0,nums))

print("Sum of the positive numbers: ",sum(total_negative_nums))
print("Sum of the negative numbers: ",sum(total_positive_nums))
```

```
>>> Sum of the positive numbers:  -32
>>> Sum of the negative numbers:  48
```

Questions?

Hands On

Plan for today

- **Functions**
  - **What are they good for**
  - **Built-in Functions**
  - **Defining New Functions**
  - **Functions call functions**
- **Lambda**
- **Recursion**

Recursion

Recursive function:

A function whose implementation calls itself (with different arguements).

Recursive Solution

A solution to a “large” problem using solutions to “small” problems that assemble it.

Recursion

- In every recursive call the problem is reduced.
- When the problem is small enough - solve directly (base case).

A **divide and conquer**

strategy

Iterative Versus Recursive

Step by step (iteratively):

n! = 1*2*3*….*n

4! = 1*2*3*4 = 2*3*4 = 6*4 = 24

Recursively:

$$\begin{cases} n! = n*(n-1)! \\ 0! = 1 \end{cases}$$

4! = 4*3!

= 4*(3*2!)

= 4*(3*(2*1!))

=4*(3*(2*(1*0!)))

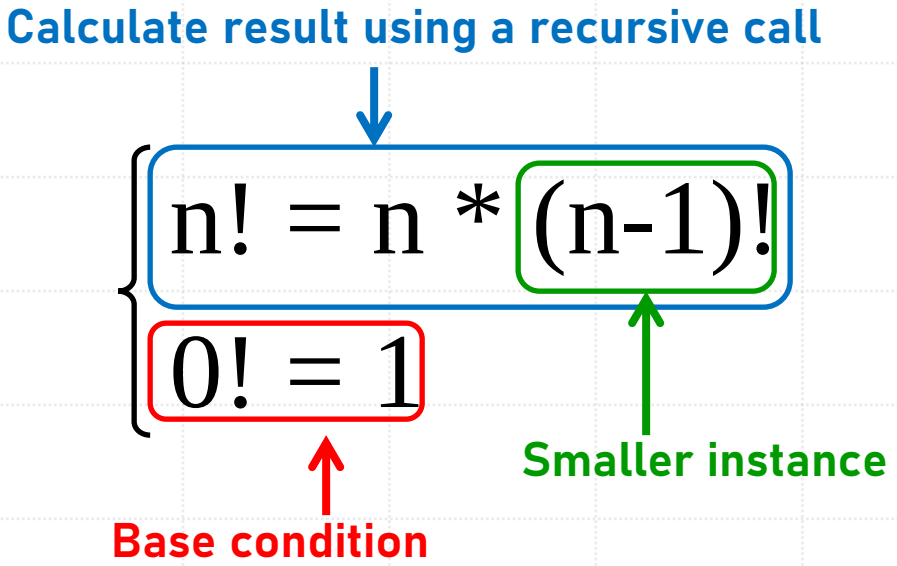
=4*(3*(2*(1*1)))

=4*(3*(2*1))

= 4*(3*2) = 4*6 = 24

Recursive Definition

Factorial



Pros and Cons

Pros

Short

Natural for

some problems

Cons

Computational inefficient

Hard to understand

Recursion Example in Python

```
def factorial(n):
    if n == 0:
        return 1
    return n * factorial(n-1)
```

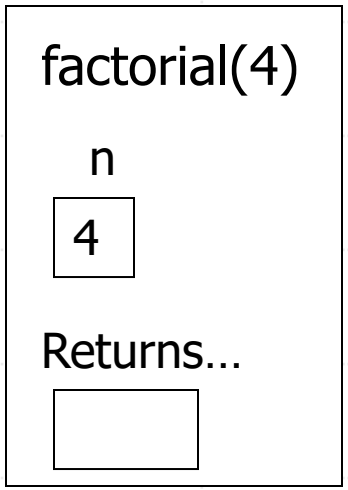
stop condition

calculate the result
using a recursive call

advance towards base case

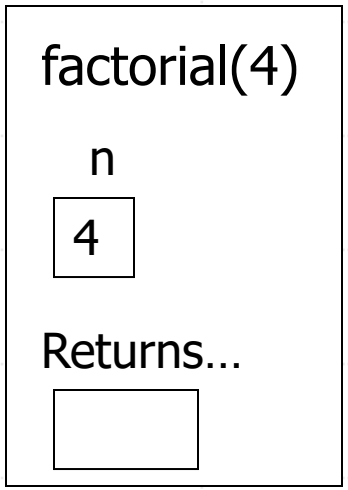
Recursive factorial – step by step

```
def factorial(n):
    if n == 0:
        return 1
    return n * factorial(n-1)
```



Recursive factorial – step by step

```
def factorial(n):
    if n == 0:
        return 1
    return n * factorial(n-1)
```



Recursive factorial – step by step

```
def factorial(n):
    if n == 0:
        return 1
    return n * factorial(n-1)
```

factorial(4)

n

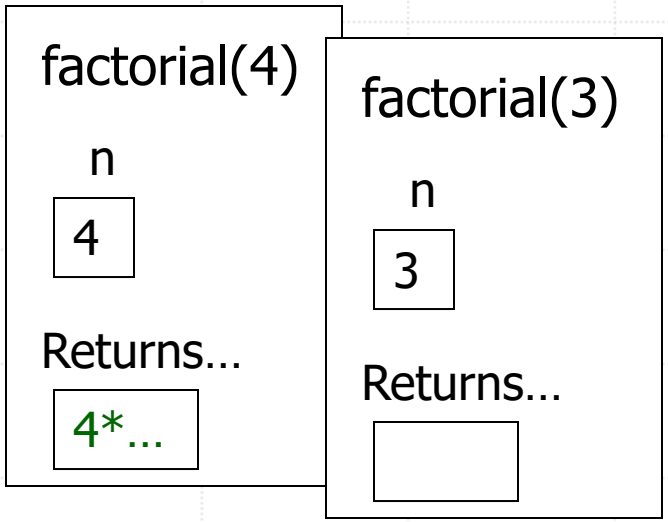
4

Returns…

4*…

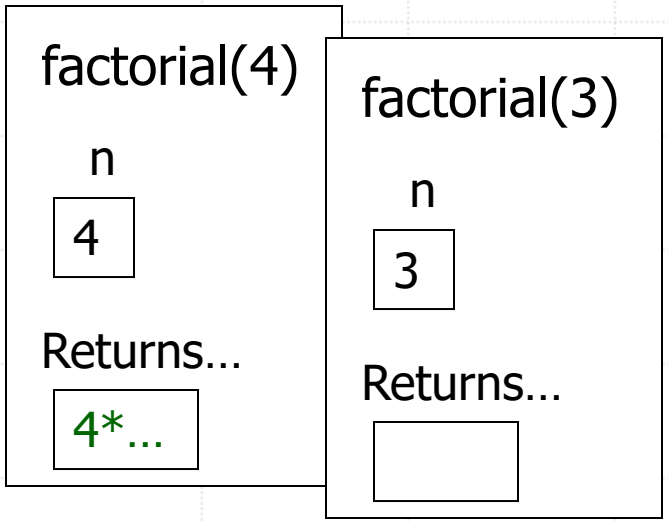
Recursive factorial – step by step

```
def factorial(n):
    if n == 0:
        return 1
    return n * factorial(n-1)
```



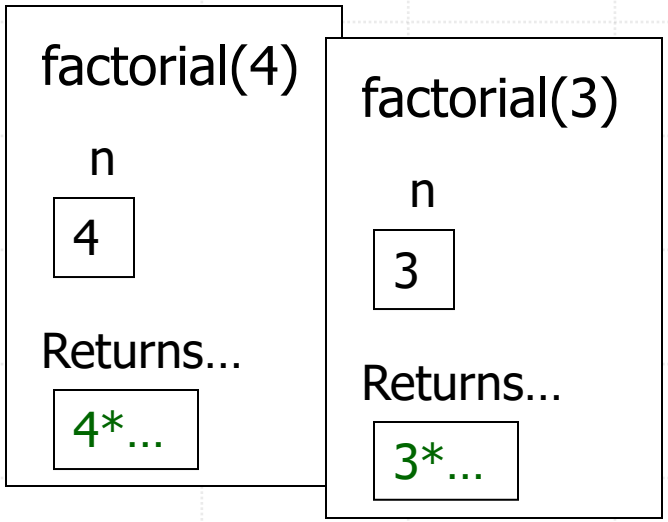
Recursive factorial – step by step

```
def factorial(n):
    if n == 0:
        return 1
    return n * factorial(n-1)
```



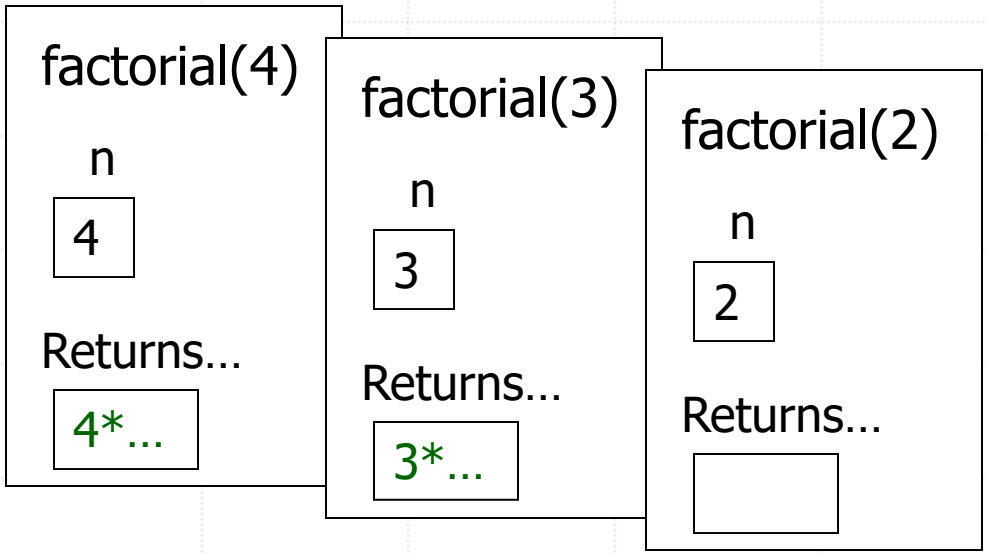
Recursive factorial – step by step

```
def factorial(n):
    if n == 0:
        return 1
    return n * factorial(n-1)
```



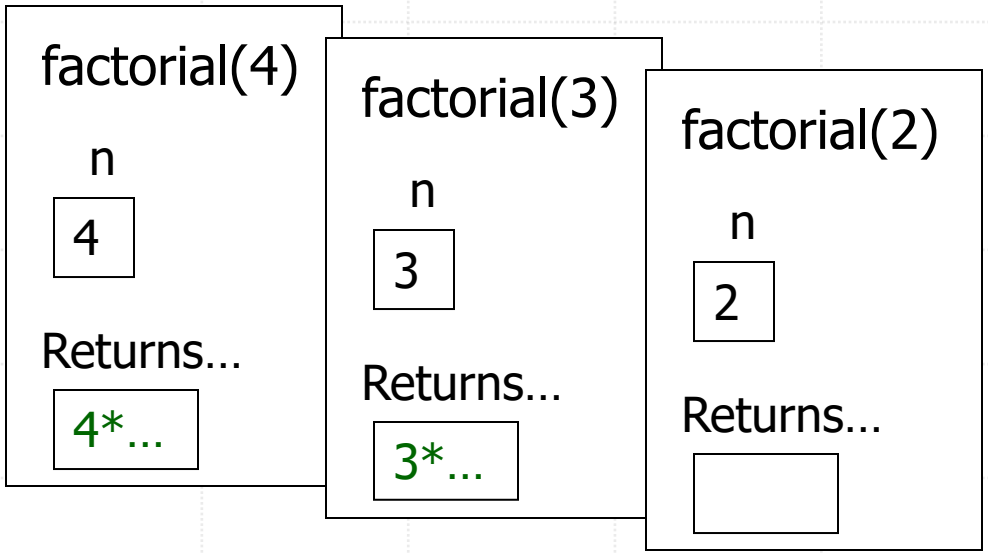
Recursive factorial – step by step

```
def factorial(n):
    if n == 0:
        return 1
    return n * factorial(n-1)
```



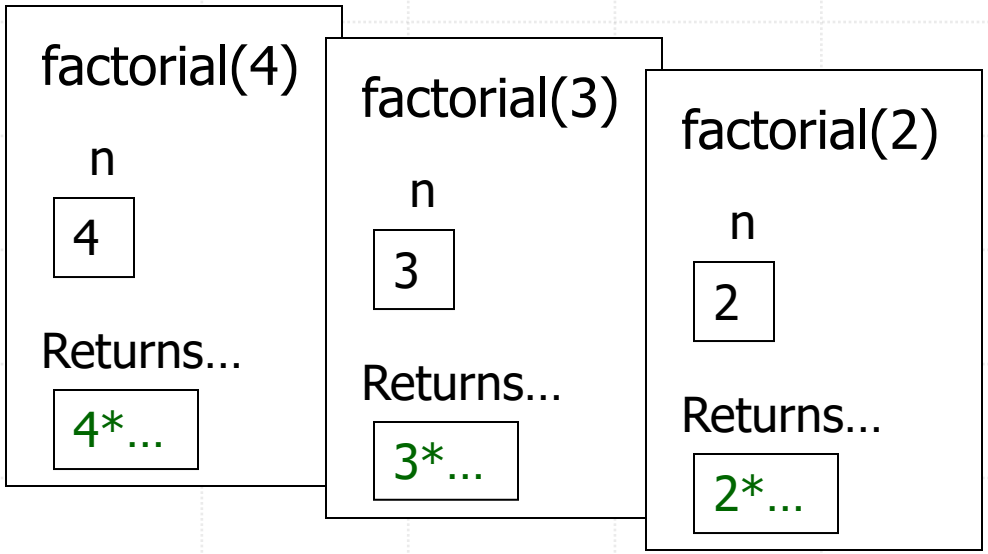
Recursive factorial – step by step

```
def factorial(n):
    if n == 0:
        return 1
    return n * factorial(n-1)
```



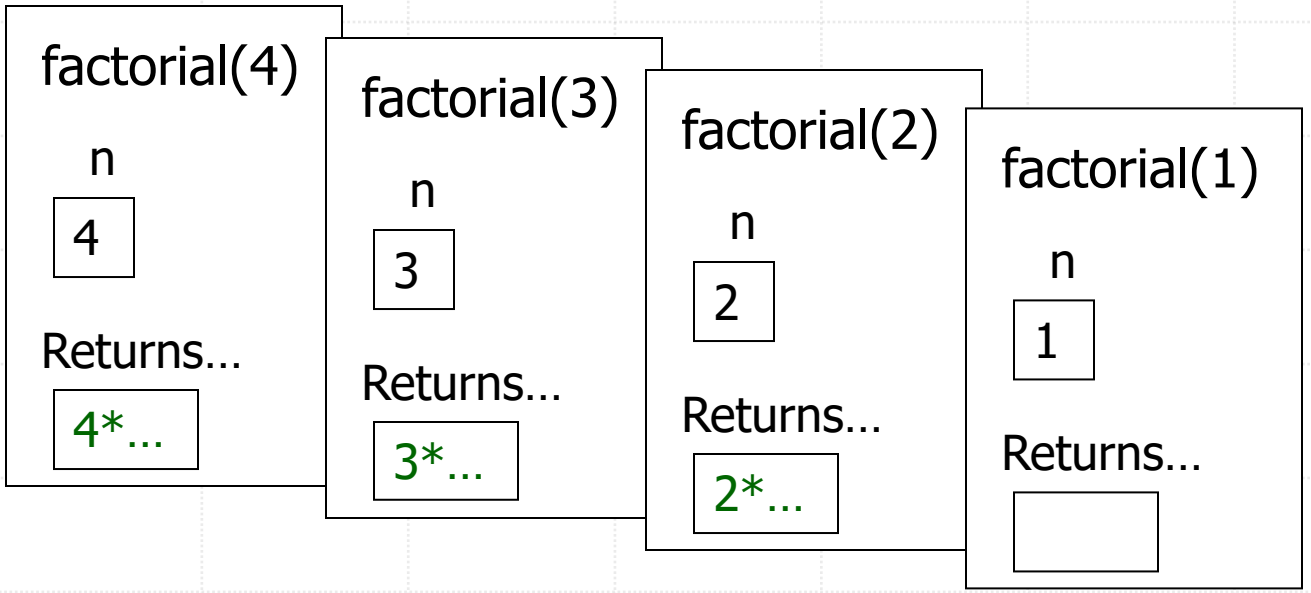
Recursive factorial – step by step

```
def factorial(n):
    if n == 0:
        return 1
    return n * factorial(n-1)
```



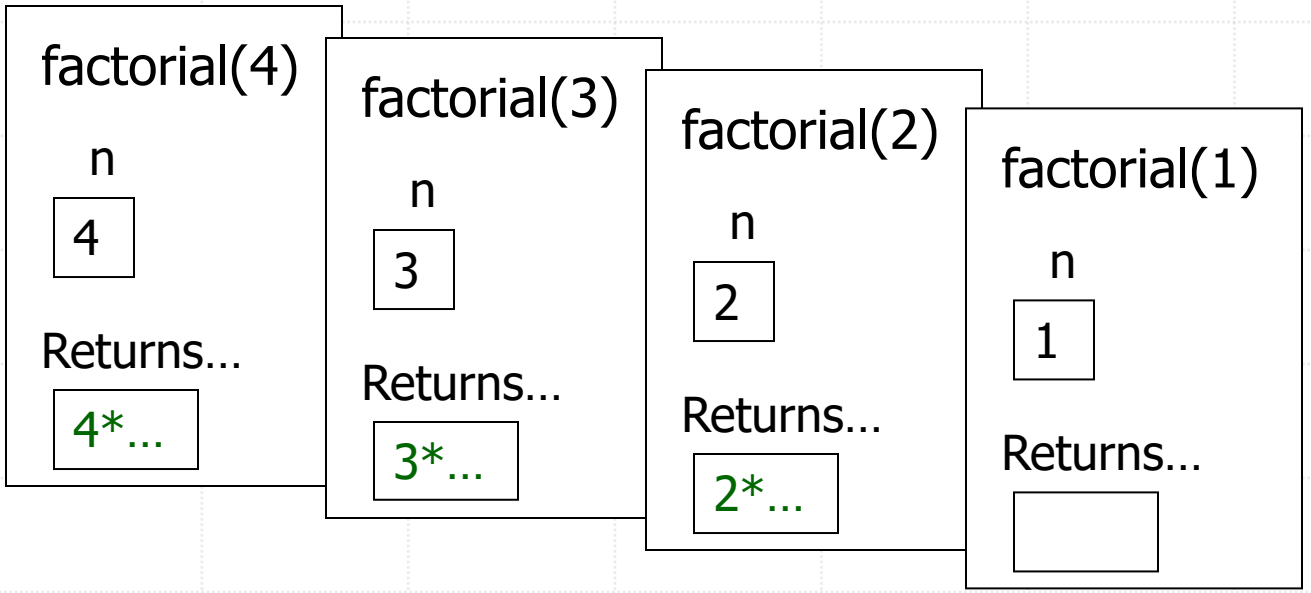
Recursive factorial – step by step

```
def factorial(n):
    if n == 0:
        return 1
    return n * factorial(n-1)
```



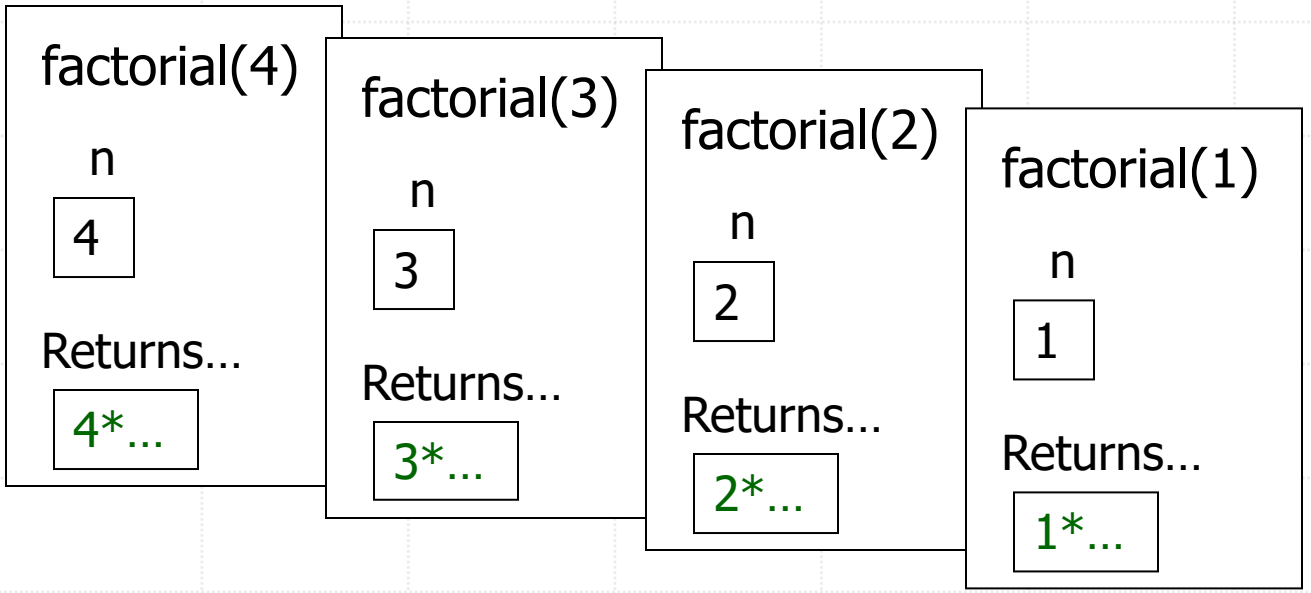
Recursive factorial – step by step

```
def factorial(n):
    if n == 0:
        return 1
    return n * factorial(n-1)
```



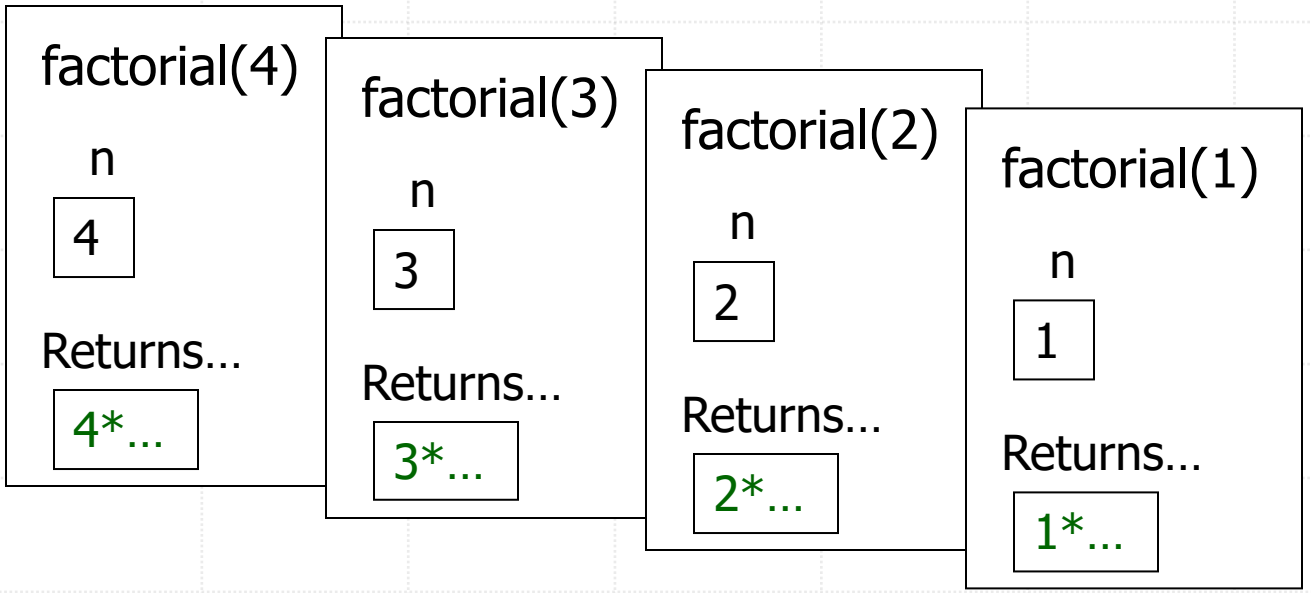
Recursive factorial – step by step

```
def factorial(n):
    if n == 0:
        return 1
    return n * factorial(n-1)
```



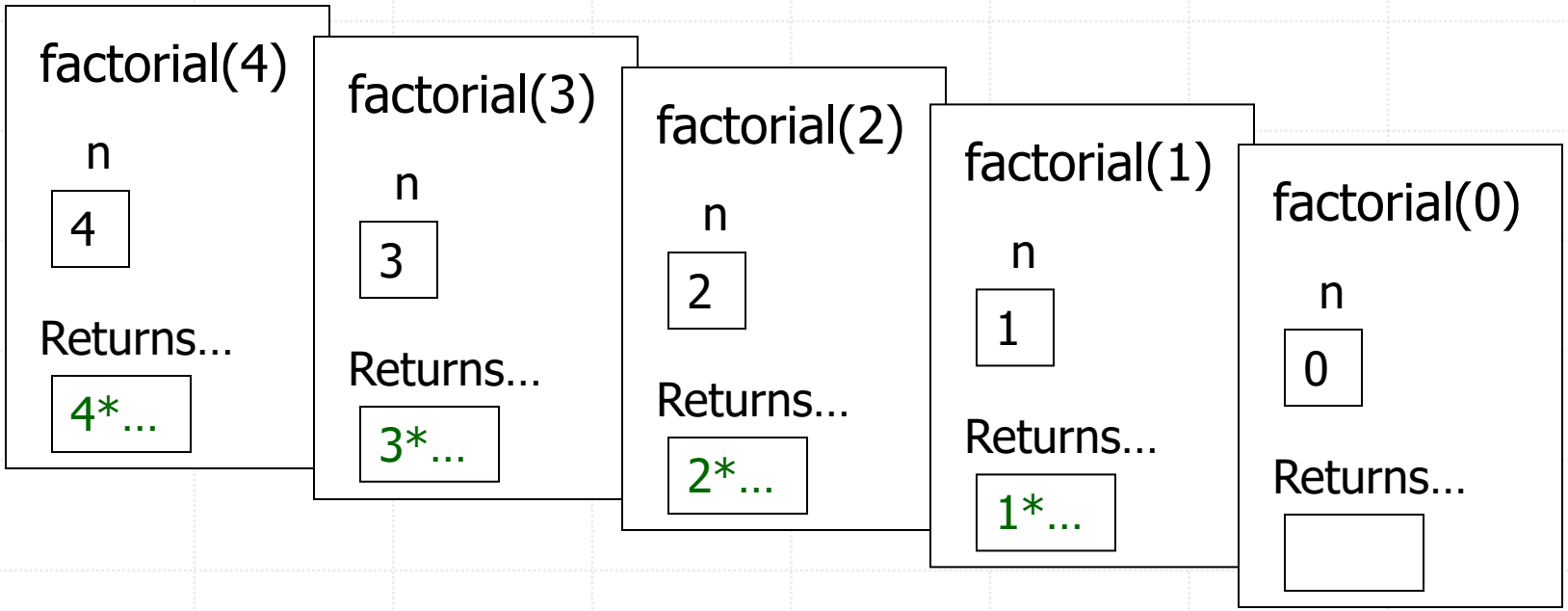
Recursive factorial – step by step

```
def factorial(n):
    if n == 0:
        return 1
    return n * factorial(n-1)
```



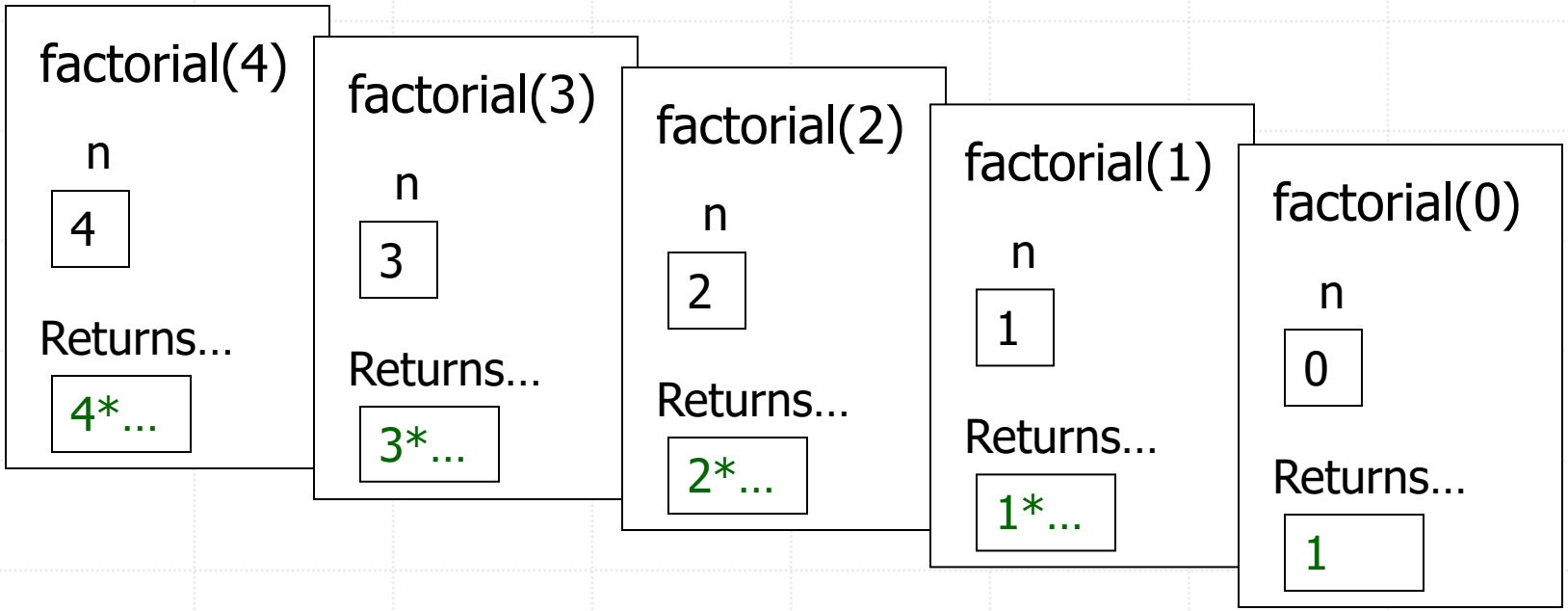
Recursive factorial – step by step

```
def factorial(n):
    if n == 0:
        return 1
    return n * factorial(n-1)
```



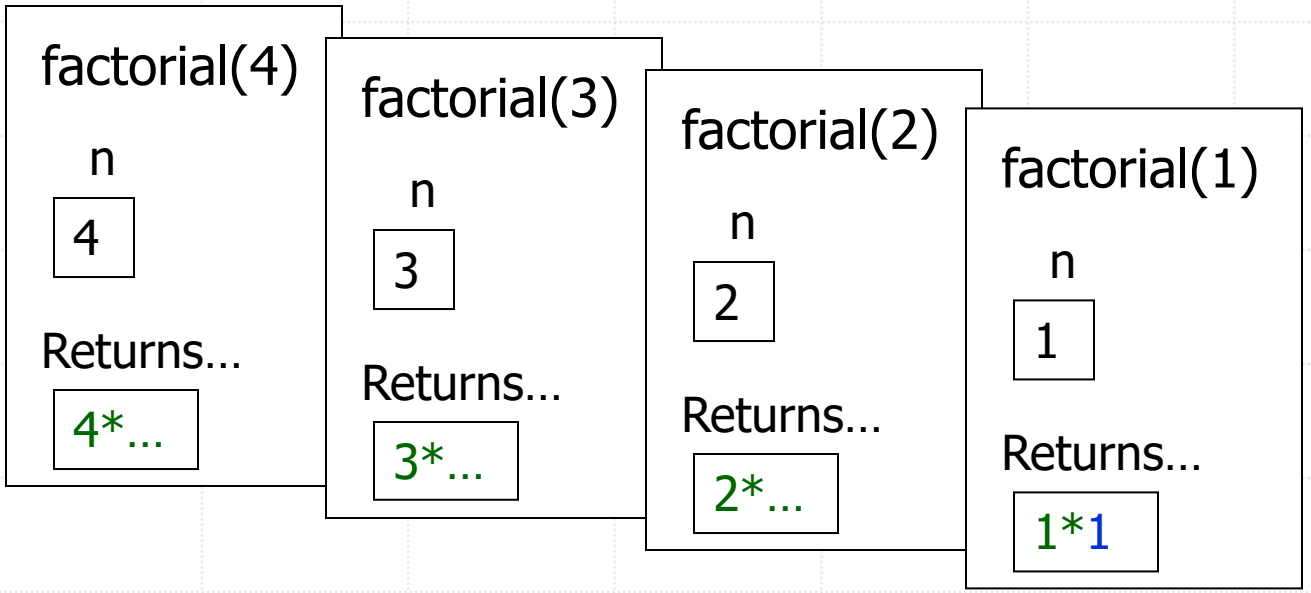
Recursive factorial – step by step

```
def factorial(n):
    if n == 0:
        return 1
    return n * factorial(n-1)
```



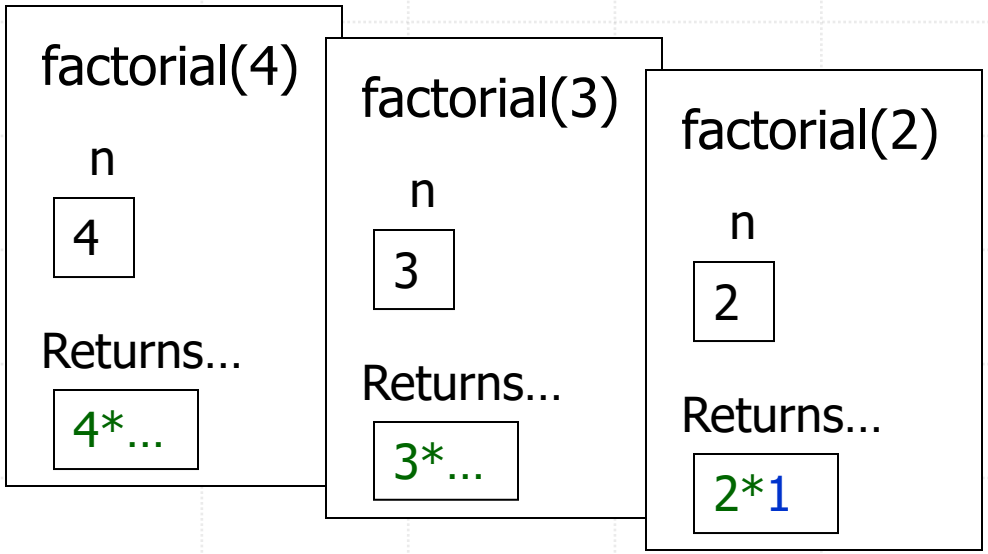
Recursive factorial – step by step

```
def factorial(n):
    if n == 0:
        return 1
    return n * factorial(n-1)
```



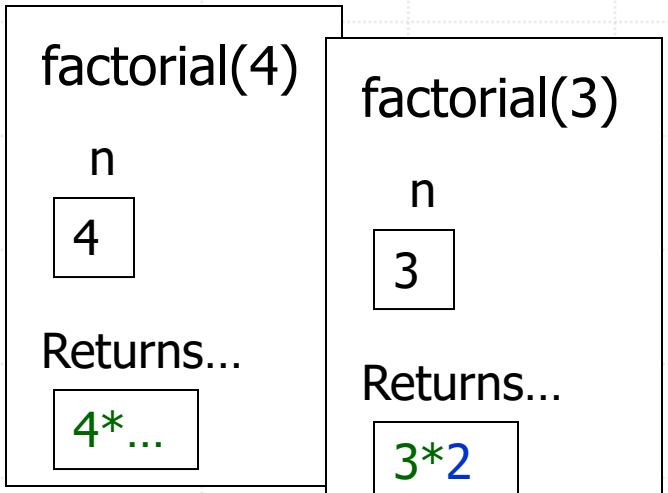
Recursive factorial – step by step

```
def factorial(n):
    if n == 0:
        return 1
    return n * factorial(n-1)
```



Recursive factorial – step by step

```
def factorial(n):
    if n == 0:
        return 1
    return n * factorial(n-1)
```



Recursive factorial – step by step

```
def factorial(n):
    if n == 0:
        return 1
    return n * factorial(n-1)
```

factorial(4)

n

4

Returns…

4*6

General Form of Recursive Algorithms

```
def factorial(n):
    if n == 0:
        return 1
    return n * factorial(n-1)
```

Recursive case - decomposable problem.

Must be:

- at least one base case (the stop condition)
- at least one recursive call.

Short Summary

Design a recursive algorithm by

1. Breaking of big problems to smaller problems
2. Solving base cases directly.

Recursive algorithms have

1. Stopping criteria
2. Recursive call(s)
3. A solution that uses solutions of smaller cases

Example: Fibonacci Series

- Fibonacci series

  0, 1, 1, 2, 3, 5, 8, 13, 21, 34

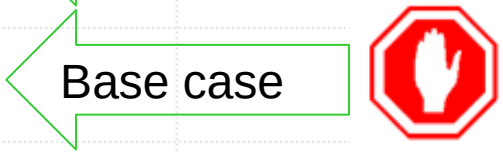
- Definition:
  - fib(0) = 0
  - fib(1) = 1
  - fib(n) = fib(n-1) + fib(n-2)

Fibonacci

"Naturally" recursive

Therefore, the recursive definition is:

- fib(0) = 0 

- fib(1) = 1 

- fib(n) = fib(n-1) + fib(n-2) 

Summary

- Functions
  - What are they good for
  - Built-in Functions
  - Defining New Functions
  - Functions call functions
- Lambda
- Recursion

Homework

1. Create a Python script to create and print a list where the values are square of numbers between 1 and n (both included)

Input:

>>> 5

Output:

>>> [1, 4, 9, 16, 25]

2. -Create a Python script to find all keys in the provided dictionary that have the given value.

Input:

>>> students = {'Theodore': 19,'Roxanne': 20,'Mathew': 21,'Betty': 20}

Output:

>>> ['Roxanne', 'Betty']

Homework

3. Create a Python script to calculate the value of the following expression by using lambda function.(x*10+(y/2)*z)

Input:

>>> x = 5, y = 5, z = 5

Output:

>>> 62.5

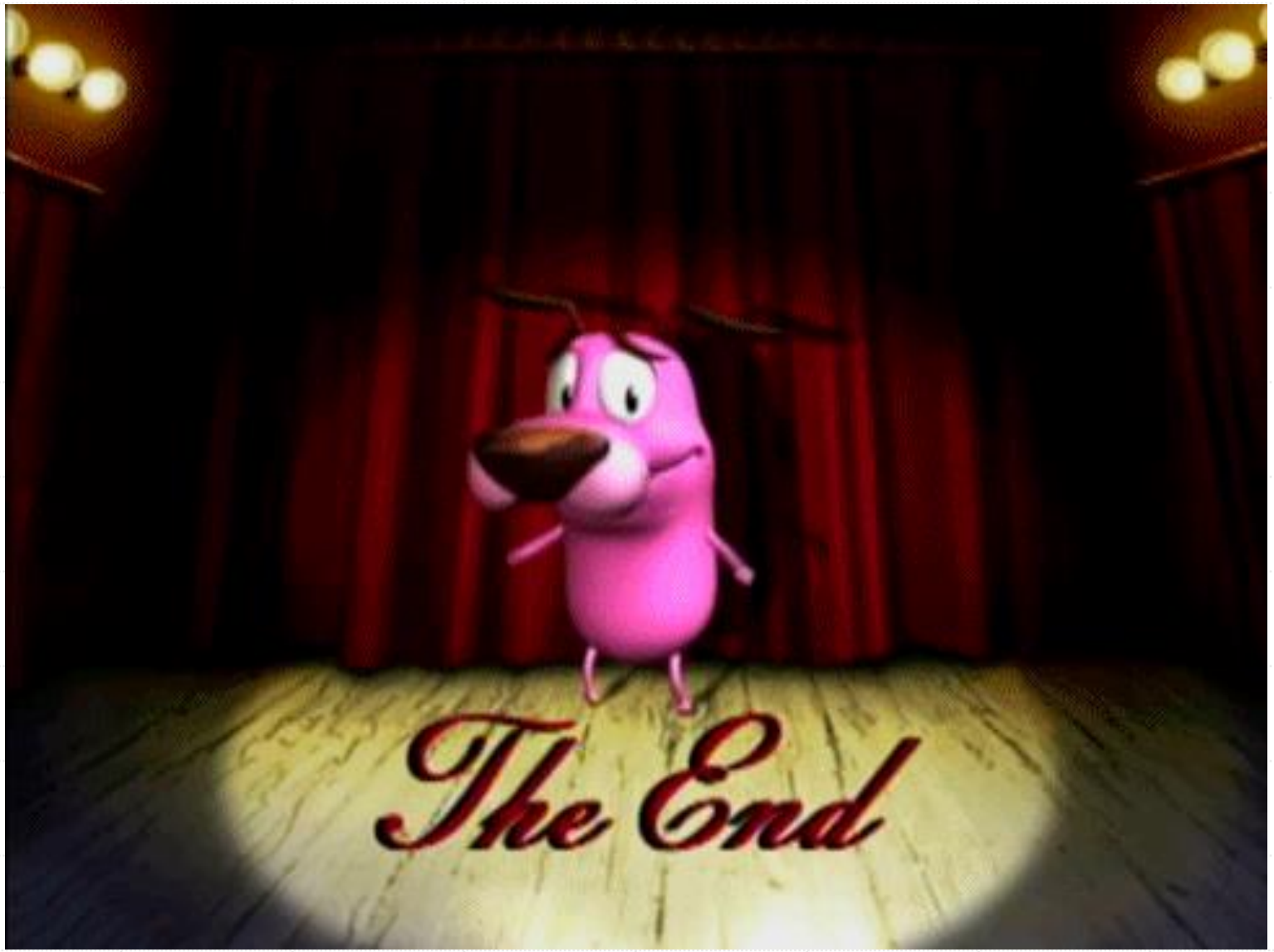
4. Create a Python script to create a lambda function that multiplies argument x with argument y and print the result

Input:

>>> a=4, b=12

Output:

>>> 48


The End