Recursion

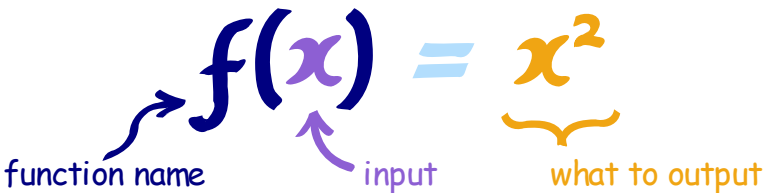
Lecture 06.01

by *Marina Barsky*

Readings:
https://www.cs.hmc.edu/csforall/index.html#functional-programming

You may read the entire Chapter 2 (for a review on functions), concentrate on 2.7 – 2.12

Functions as mathematical concepts (proper functions)



parameter

```
def f(x):
    return x**2
```

return value

```
result = f(4)
```

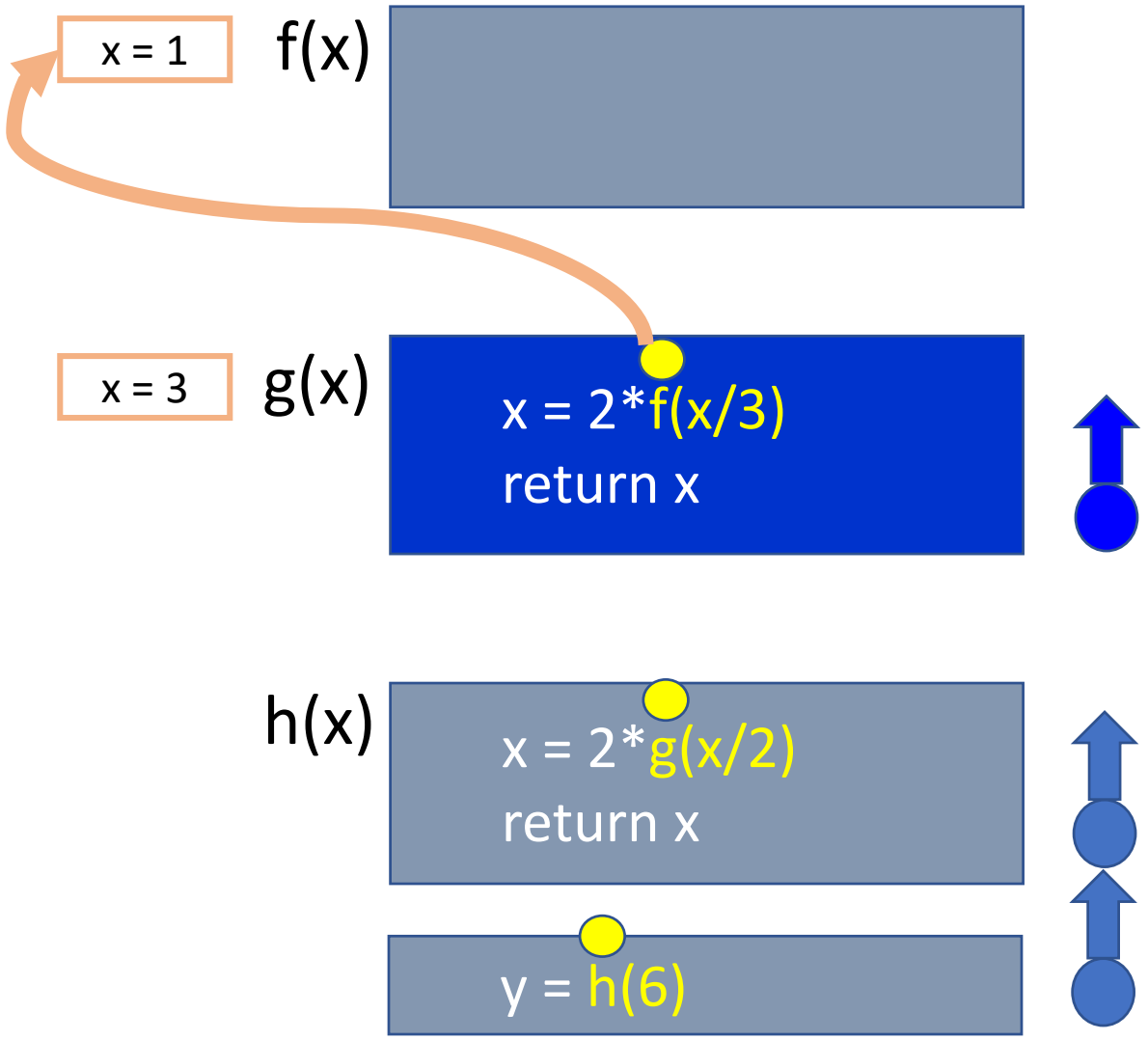
calling function f with argument 4

Recall: functions can call other functions

```
def f (x):
    x = 2*x
    return x

def g (x):
    x = 2*f(x/3)
    return x

def h (x):
    x = 2*g(x/2)
    return x

#function call
h (6)
```

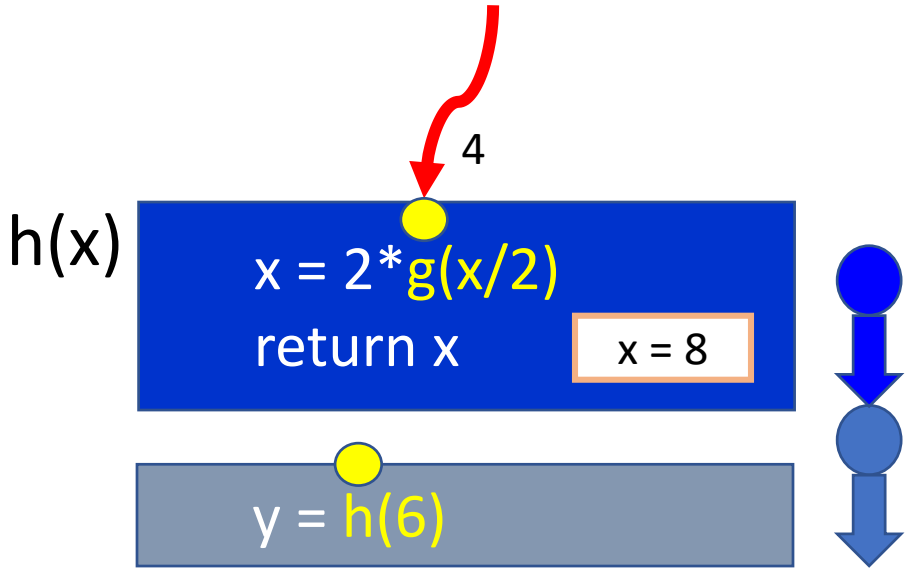
Stack – stacking functions until can compute value

```
def f (x):
    x = 2*x
    return x

def g (x):
    x = 2*f(x/3)
    return x

def h (x):
    x = 2*g(x/2)
    return x

#function call
y  = h (6)
```



Unloading functions from the stack

```
def f (x):
    x = 2*x
    return x

def g (x):
    x = 2*f(x/3)
    return x

def h (x):
    x = 2*g(x/2)
    return x

#function call
y  = h (6)
```



A function can call the same function!

- What will happen if we place call to function f() inside function f()?
- The stack frames will pile up until memory permits and then the program will crash
- We use functions which call the same function inside them if the problem can be broken into smaller problems, which require the same computation
- Such problems are called ***recursive problems***, and the function which contains call to itself is called a ***recursive function***

Example of recursive problem: factorial

5! = 5*4*3*2*1

4!

5! = 5 * (4!)

4! = 4*(3!)

Etc.

$$F(n) = n * F(n-1) \text{ for } n > 1$$
$$F(1) = 1$$

Two important features of a recursive solution

- A recursive solution must have **one or more base cases** (when to stop)

  factorial(1) = 1

- A recursive solution can be expressed through the exact same solution **with a smaller problem size**

  factorial(n) = n * factorial(n-1)

Function *factorial*

$$F(n) = n * F(n-1) \text{ for } n > 0$$
$$F(1) = 1$$

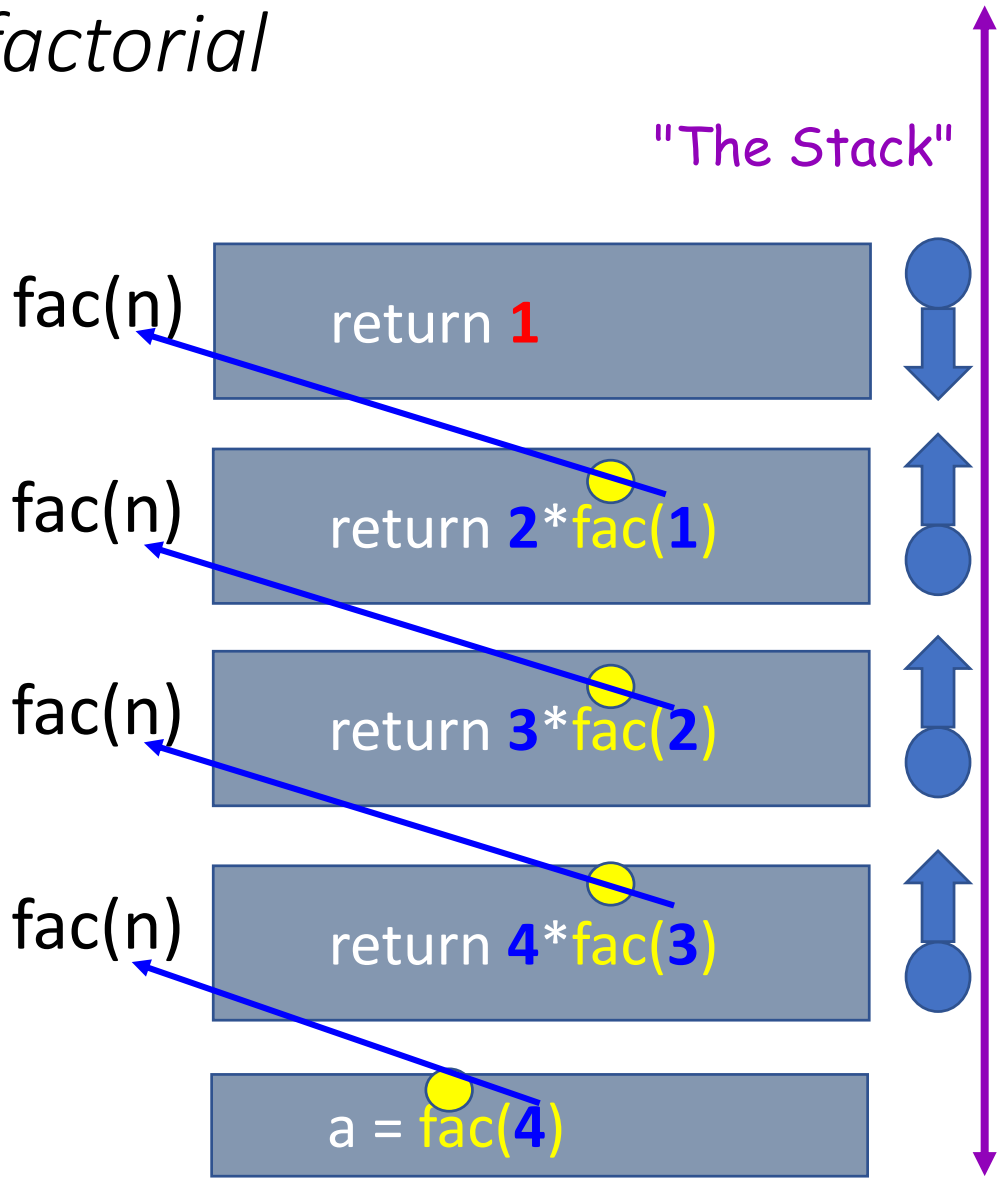
```
def factorial (n):
    ''' (int) -> int

    Computes factorial of
      positive integer n
    >>> factorial (3)
    6
    >>>factorial (7)
    5040
    '''
    if n <= 1:
        return 1
    return n * factorial (n-1)
```

Base case

Recursion with smaller problem

Behind the curtain: *factorial*

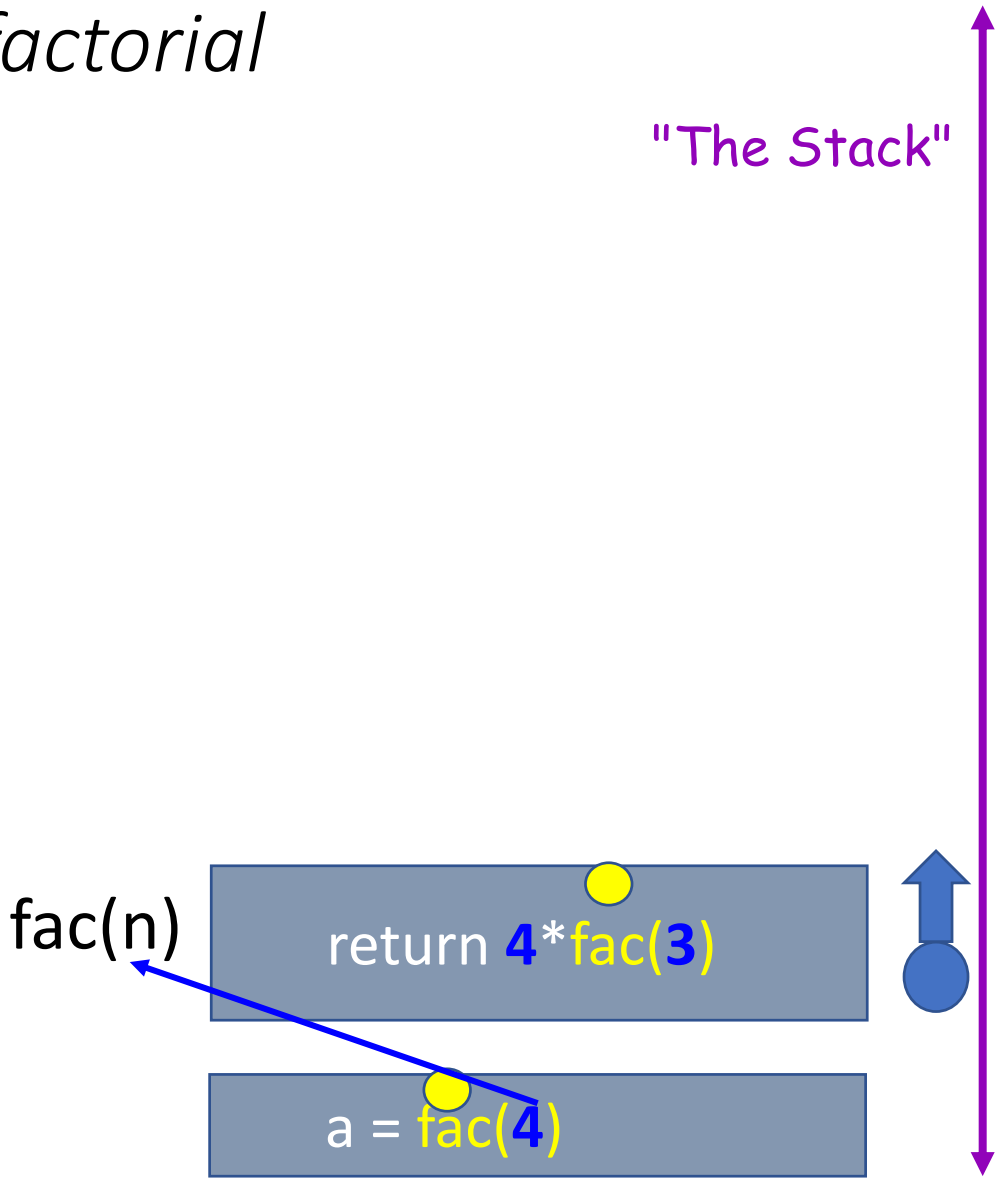
```
def fac (n):
    if n <= 1:
        return 1
    return n * fac (n-1)

a = fac (4)
```



Behind the curtain: *factorial*

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

a = fac (4)
```

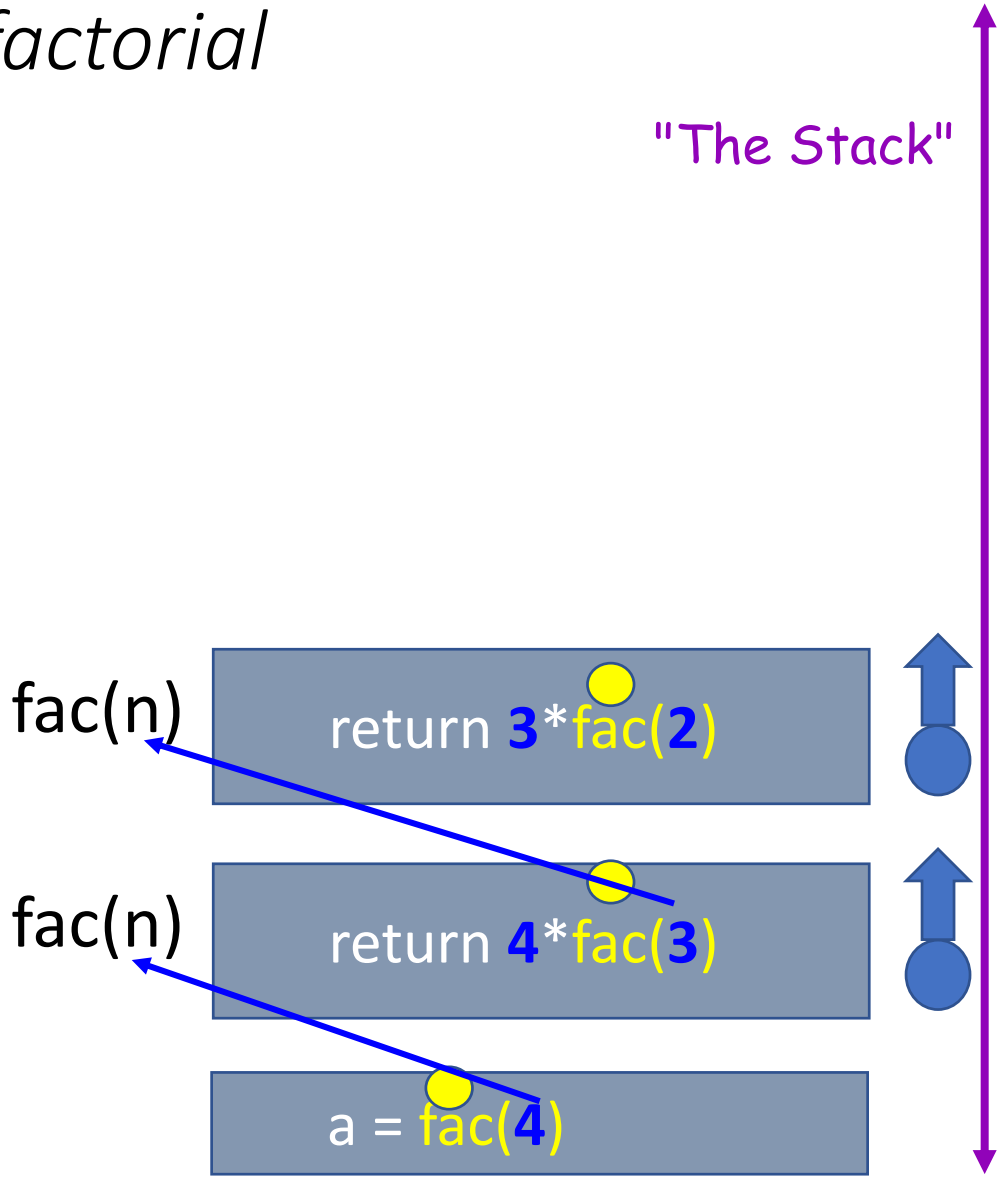
Loaded definition of *fac* to compute *fac(4)*, but cannot compute, needs to compute *fac(3)* first



Behind the curtain: *factorial*

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

a = fac (4)
```

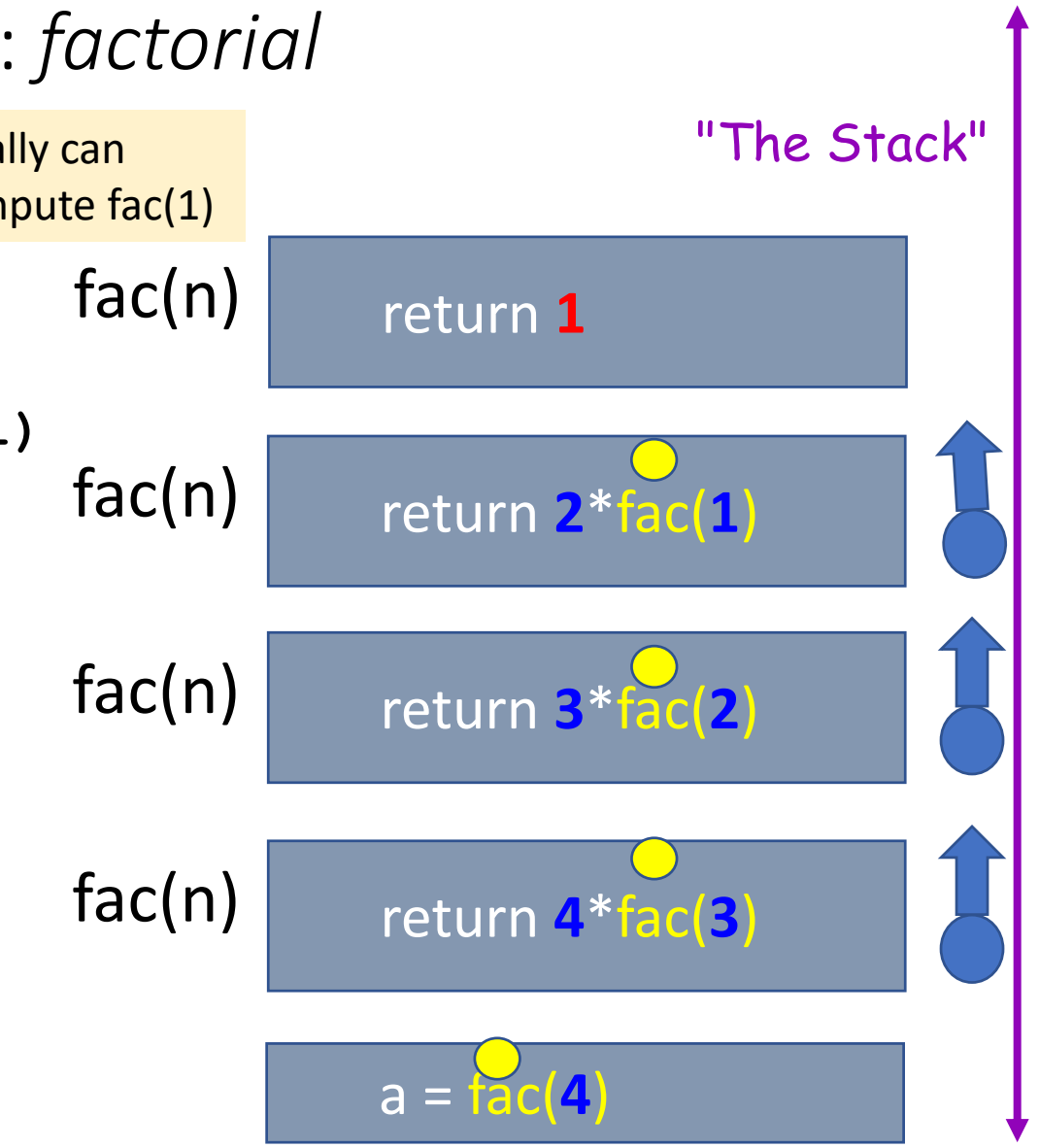
Loaded a **different copy** of *fac*, to compute *fac(3)*



Behind the curtain: *factorial*

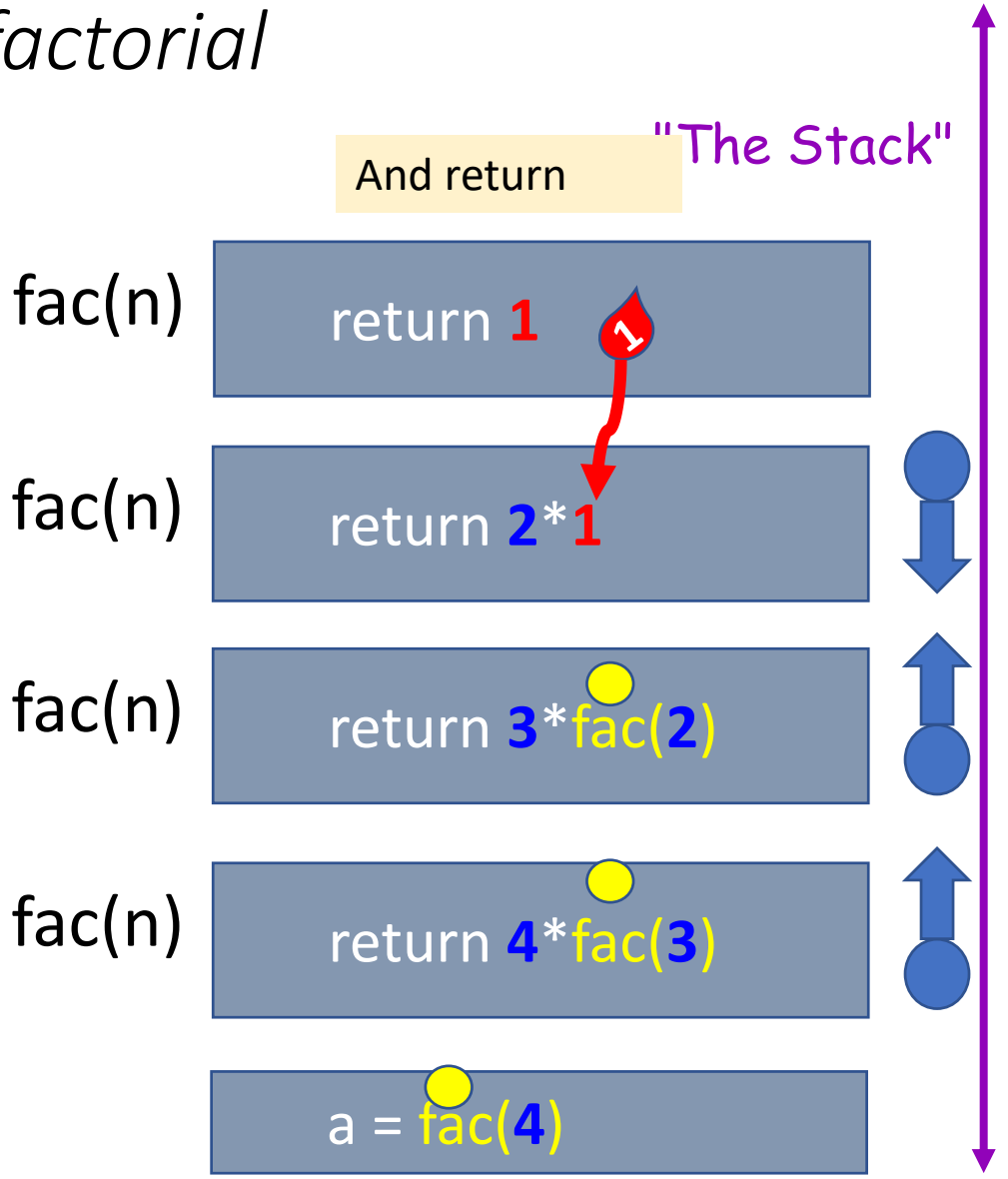
```
def fac (n):
    if n <= 1:
        return 1
    return n * fac (n-1)

a = fac (4)
```

Finally can compute fac(1)



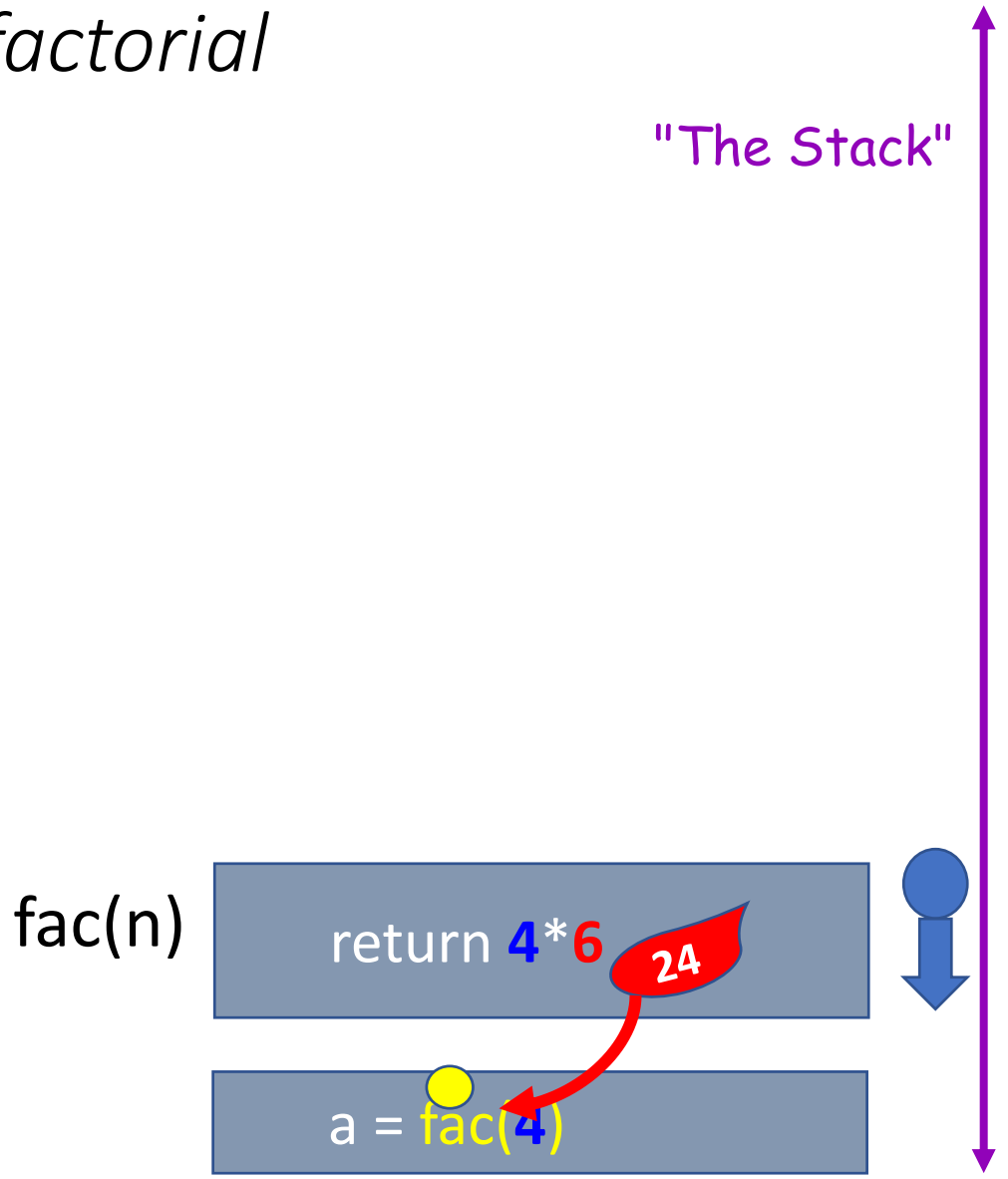
Behind the curtain: *factorial*

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

a = fac (4)
```



Behind the curtain: *factorial*

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

a = fac (4)
```



Behind the curtain: *factorial*

"The Stack"

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

a = fac (4)
```

a = **24**

Let recursion do the work for you!

Exploit self-similarity

Produce short, elegant code

} **Less work !**

```
def factorial(n):

    if n <= 1:
        return 1
    else:
        return n * factorial(n-1)
```

You handle the base case – the easiest possible case to think of!

Recursion does <u>almost</u> all of the rest of the problem!

Always a “smaller” problem!

Exercise 1: recursive *sum*

- How to modify *factorial* function to give us a sum of integers from 1 to n?

```
def factorial(n):

    if n <= 1:
        return 1
    else:
        return n * factorial(n-1)
```

Exercise 1: recursive *sum* solution

- How to modify *factorial* function to give us a sum of integers from 1 to n?

```
def sum(n):

    if n <= 1:
        return 1
    else:
        return n + sum(n-1)
```

How to read recursive functions

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

- Take an example, say, factorial(3)
- Draw separate copies of the same *factorial* function for 3, 2, 1.
- Pile them up in stack frames, and follow the logic of returns

Reading exercise: what is computed?

```
def func(s):
    if s == '':
        return 0
    elif s[0] in 'aeiou':
        return 1 + func(s[1:])
    else:
        return 0 + func(s[1:])
```

Reading exercise: what is computed?

```
def func(s):
    if s == '':
        return 0
    elif s[0] in 'aeiou':
        return 1 + func(s[1:])
    else:
        return 0 + func(s[1:])
```

func counts vowels in *s*

How to write recursive functions

- Start from the **base case**: teach computer how to compute factorial(1)
- If I want to compute factorial(2), I need to multiply 2 by factorial(1) – I will reuse factorial(1)
- Now I know how to compute factorial(2). To compute factorial(3), I just multiply 3 by the value computed in factorial(2).

- Now I see **the general pattern**!

```
def fac(n):
    if n == 1:
        return 1
    if n == 2:
        return 2 * fac(1)
    if n == 3:
        return 3 * fac(2)
```

How to write recursive functions: generalizing

- To compute *factorial* for any *n*, I multiply *n* by *factorial* of *n*-1

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

Exercise 2: string length

- Write a recursive function called ***my_len*** that computes the length of a string

Example:

my_len("aliens") → 6

- What is the base case?

  Empty string

  my_len('') → 0

- Recursive call:

  1 + my_len(s[1:])

Exercise 2: string length solution (stop and try)

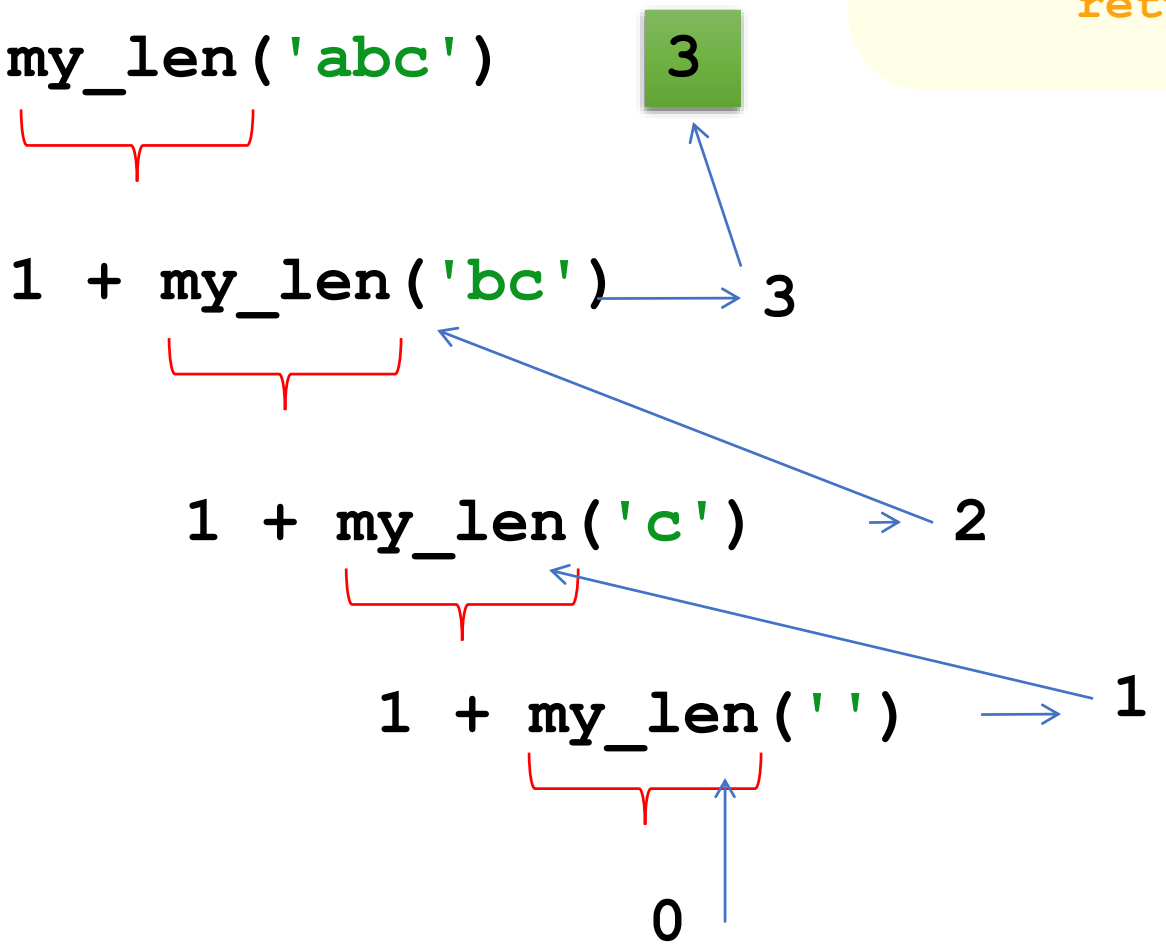
```
def my_len(s):
    """ input: any string, s
        output: the number of characters in s
    """
    if


    else:
```

Exercise 2: string length solution

```
def my_len(s):
    """ input: any string, s
        output: the number of characters in s
    """
    if s == '':
        return 0
    else:
        rest = s[1:]
        return 1 + my_len( rest )
```

Behind the curtain: *string length*...

```
def my_len(s):
    if s == '':
        return 0
    else:
        return 1 + my_len(s[1:])
```



Exercise 3: sum of digits (try it out!)

```
def sum_digits(s):
    """ input: a string s of int numbers
    '252674'
    output: the sum of the numbers
    >>> sum_digits ('1231')
    7
    """
     if


     else:
```

Exercise 4: find_list_max

- Write a recursive function called *find_list_max* that returns the maximum value in a list

- Examples:

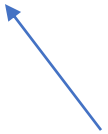
```
>>> find_list_max ([4, 13, 21, 5, 2])
21
>>> find_list_max ([1, -3, 8, -5, 12])
12
```

Exercise 4: find_list_max

```
def find_list_max(t):
    """ input: a NONEMPTY list, t
        output: t's maximum element
    """
    if              :


    elif            :


    else:
```

find_list_max

```
def find_list_max(t):
    """ input: a NONEMPTY list, t
        output: t's maximum element
    """
    if len(t) == 1:
        return t[0]
    elif t[0] < t[1]:    # t[0] can't be the max, remove it
        return find_list_max(t[1:])
    else:                # t[1] can't be the max, remove it
        return find_list_max(t[0:1] + t[2:])
```



t[0:1] returns list with a single element
We concatenate lists with lists!

Exercise 5: extract a sub-list

- Write a recursive function called *extract_list* that returns a sub-list for a given range of indexes
- Examples:

```
>>> extract_list([4, 13, 21, 5, 2],  2,  5)
[21, 5, 2]
>>> extract_list(['hello', 'world', 'how', 'are', 'you', '?'],  2,  5)
['how', 'are', 'you']
```

extract_list

```
def extract_list(t, low, hi):
    """ input: list t, two ints, low and hi
        output: list from low up to, not
                including hi
    """
    if hi <= low:     # base case
        return []
    else:
        return
```

extract_list

```
def extract_list(t, low, hi):
    """ input: list t, two ints, low and hi
        output: list from low up to, not
                    including hi
    """
    if hi <= low:
        return []
    else:
        return [t[low]] + extract_list(t, low+1, hi)
```