

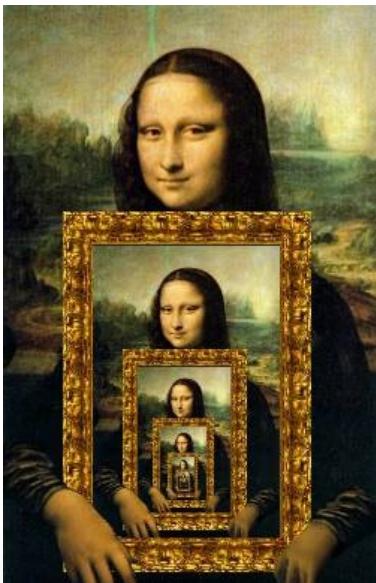
Recursion

a.k.a., CS's version of mathematical induction

As close as CS gets to magic

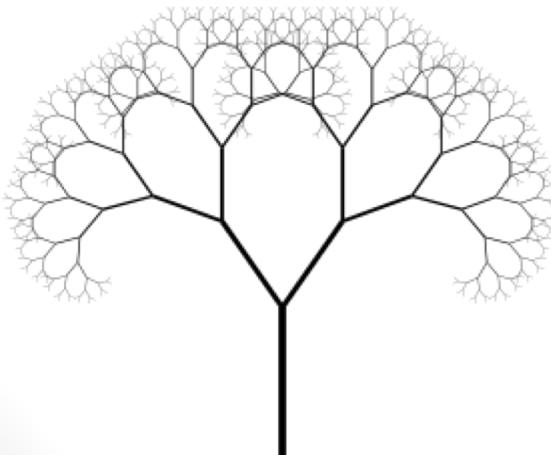
Let recursion draw you in....

- Recursion occurs when something is described in terms of itself
- Describe these pictures

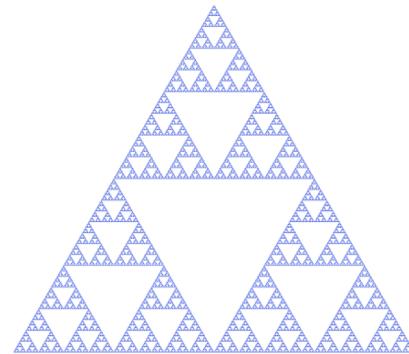


Recursion !

- General idea: Solve problems by describing it in terms of a smaller version of itself
- Applications:
Fractals, advanced data structures, file systems



Tree



Sierpinski triangle



Koch's snowflake

Function *design*

Thinking *sequentially*

factorial

$$5! = 120$$

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

$$N! = N * (N-1) * (N-2) * \dots * 3 * 2 * 1$$

Thinking *recursively*

```
N! = N * (N-1) ! , if N > 1  
= 1, if N <= 1
```

Recursion == **self**-reference!

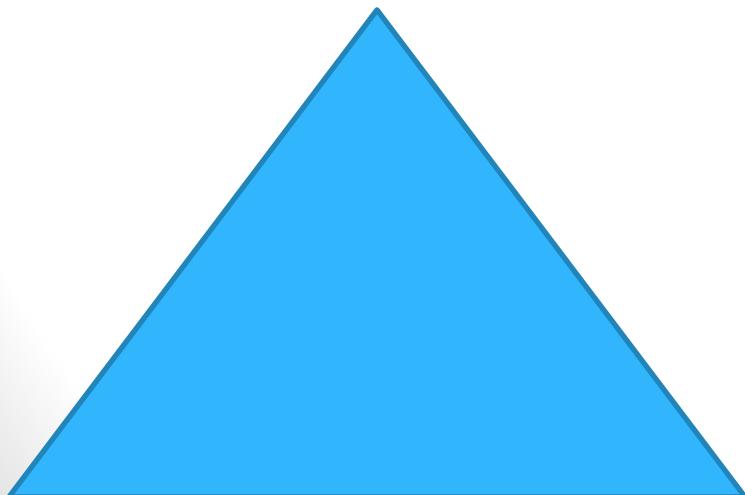
Designing Recursive Functions

```
def fac(N):  
    if N <= 1:  
        return 1 } }
```

Base case:

Solution to inputs where
the answer is trivial
(top of the pyramid)

Base case: $N \leq 1$



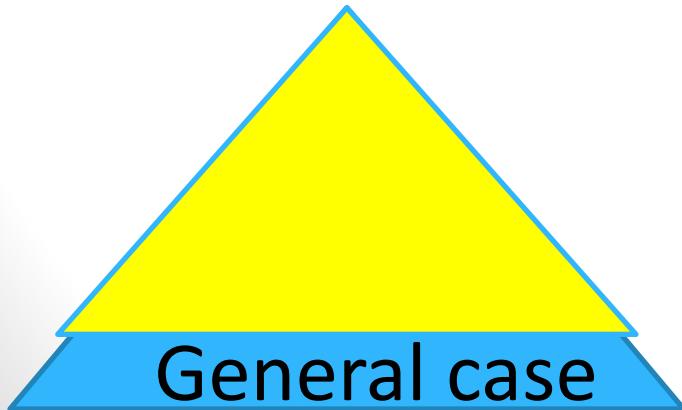
General case: $N > 1$

Designing Recursive Functions

```
def fac(N):  
    if N <= 1: } Base case  
        return 1
```

```
else: # solve for any N  
    rest = fac(N-1)
```

Base case



Designing Recursive Functions

```
def fac(N):
```

```
    if N <= 1:  
        return 1
```

} Base case

```
else:
```

```
    rest = fac(N-1)  
    return rest * N
```

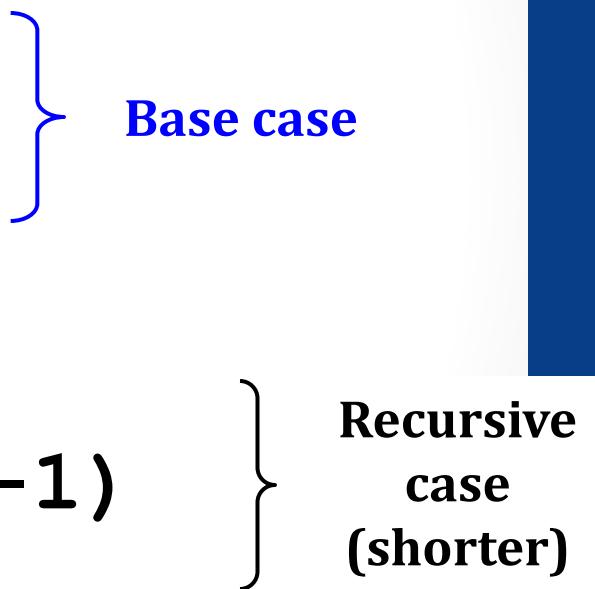
} Recursive
case

Human: Base case and 1 step

Computer: Everything else

Thinking recursively !

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



Base case

Recursive case (shorter)

Human: Base case and 1 step

Computer: Everything else

Warning: *this is legal!*

```
def fac(N) :  
    return N * fac(N-1)
```

legal != *recommended*

```
def fac(N):  
    return N * fac(N-1)
```

No *base case* -- the calls to **fac** will never stop!

Make sure you have a
base case, *then* worry
about the recursion...

How functions *work*...

I might have a
guess...



Three functions:

What is **demo (-4)** ?

```
def demo(x) :  
    return x + f(x)
```

```
def f(x) :  
    return 11*g(x) + g(x/2)
```

```
def g(x) :  
    return -1 * x
```

How functions work...

```
def demo(x):  
    return x + f(x)
```

```
def f(x):  
    return 11*g(x) + g(x/2)
```

```
def g(x):  
    return -1 * x
```

```
>>> demo(-4) ?
```

```
demo  
x = -4  
return -4 + f(-4)
```

How functions work...

```
def demo(x) :  
    return x + f(x)  
  
def f(x) :  
    return 11*g(x)+g(x/2)  
  
def g(x) :  
    return -1 * x  
  
->>> demo(-4) ?
```

demo

x = -4

return -4 + f(-4)

f

x = -4

return 11*g(x) + g(x/2)

How functions work...

```
def demo(x):  
    return x + f(x)
```

```
def f(x):  
    return 11*g(x)+g(x/2)
```

```
def g(x):  
    return -1 * x
```

```
>>> demo(-4) ?
```

demo
x = -4
return -4 + **f(-4)**

f
x = -4
return 11*g(x) + g(x/2)

These are different x's !

How functions work...

```
def demo(x):  
    return x + f(x)
```

```
def f(x):  
    return 11*g(x)+g(x/2)
```

```
def g(x):  
    return -1 * x
```

```
>>> demo(-4) ?
```

```
demo  
x = -4  
return -4 + f(-4)
```

```
f  
x = -4  
return 11*g(-4) + g(-4/2)
```

```
g  
x = -4  
return -1.0 * x
```

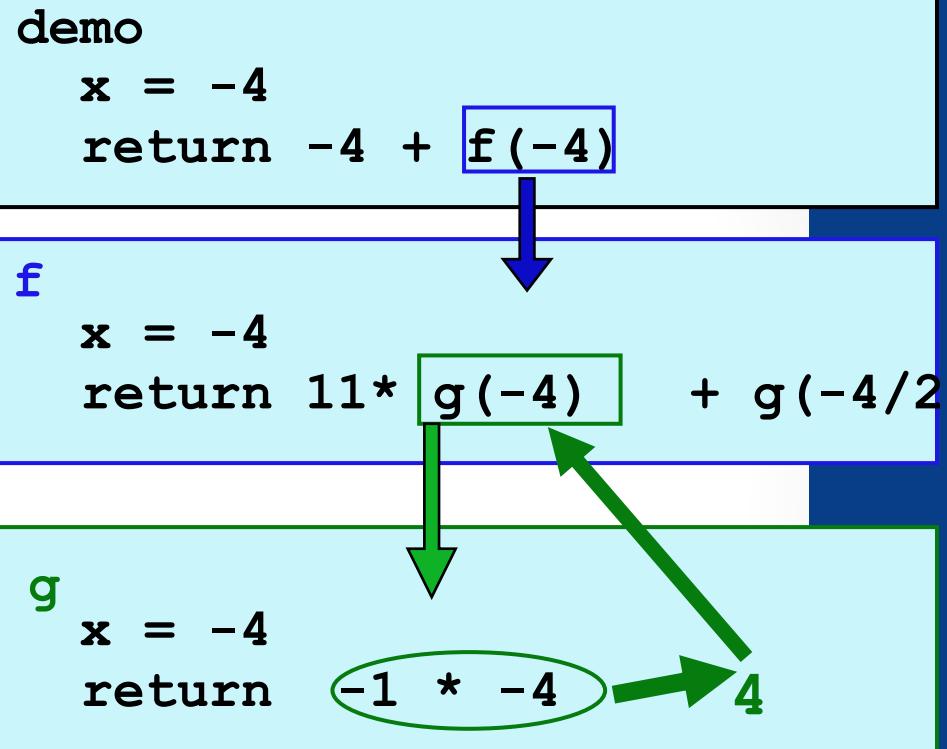
How functions work...

```
def demo(x):  
    return x + f(x)
```

```
def f(x):  
    return 11*g(x)+g(x/2)
```

```
def g(x):  
    return -1 * x
```

```
>>> demo(-4) ?
```



How functions work...

```
def demo(x):  
    return x + f(x)
```

```
def f(x):  
    return 11*g(x)+g(x/2)
```

```
def g(x):  
    return -1 * x
```

```
>>> demo(-4) ?
```

What happens next in program memory?

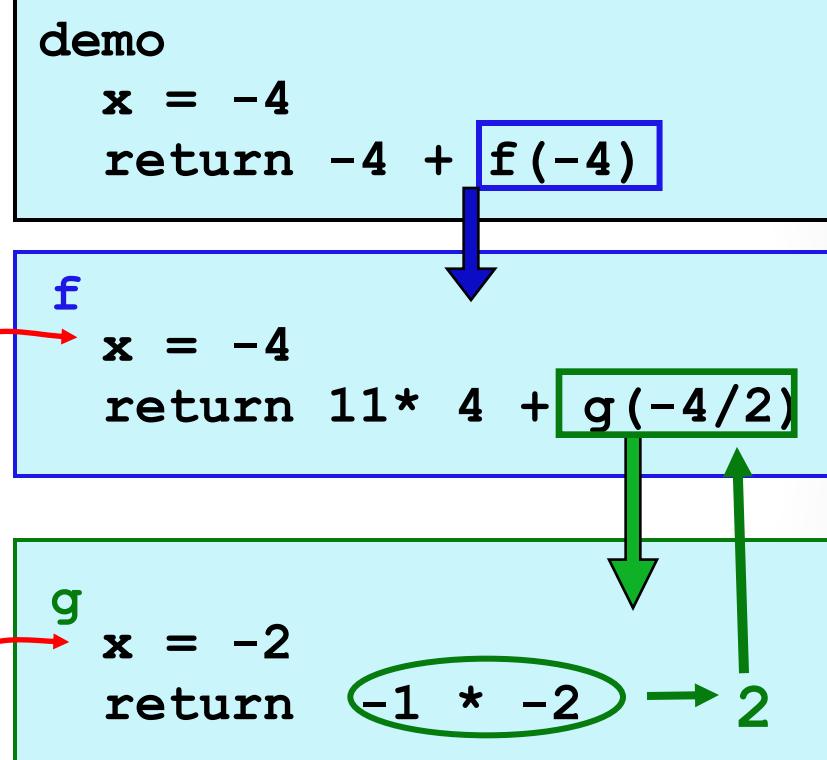
- A. f() returns, its local variables are removed from memory
- B. g() is called, new local variable (x) is created in memory

```
demo  
x = -4  
return -4 + f(-4)
```

```
f  
x = -4  
return 11* 4 + g(-4/2)
```

How functions work...

```
def demo(x):  
    return x + f(x)  
  
def f(x):  
    return 11*g(x) + g(x/2)  
  
def g(x):  
    return -1 * x  
  
">>>> demo(-4) ?
```

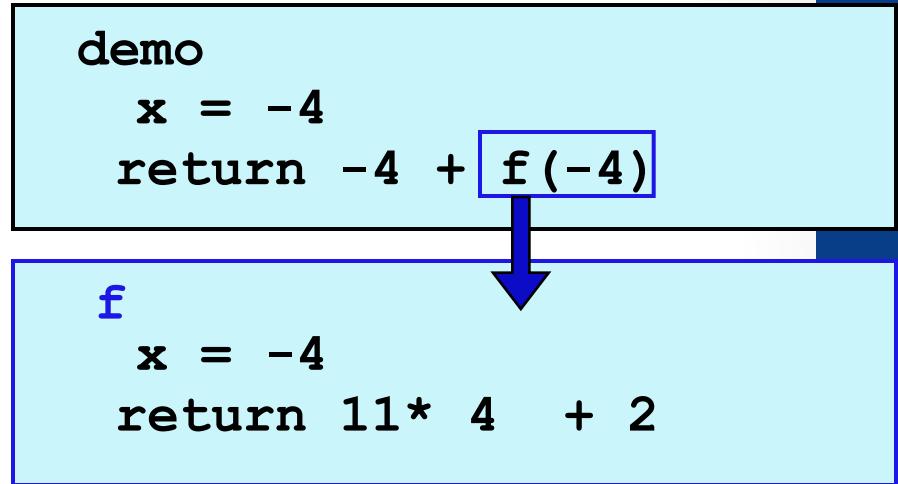


These are *really* different `x`'s !

How functions work...

```
def demo(x):  
    return x + f(x)  
  
def f(x):  
    return 11*g(x) + g(x/2)  
  
def g(x):  
    return -1 * x
```

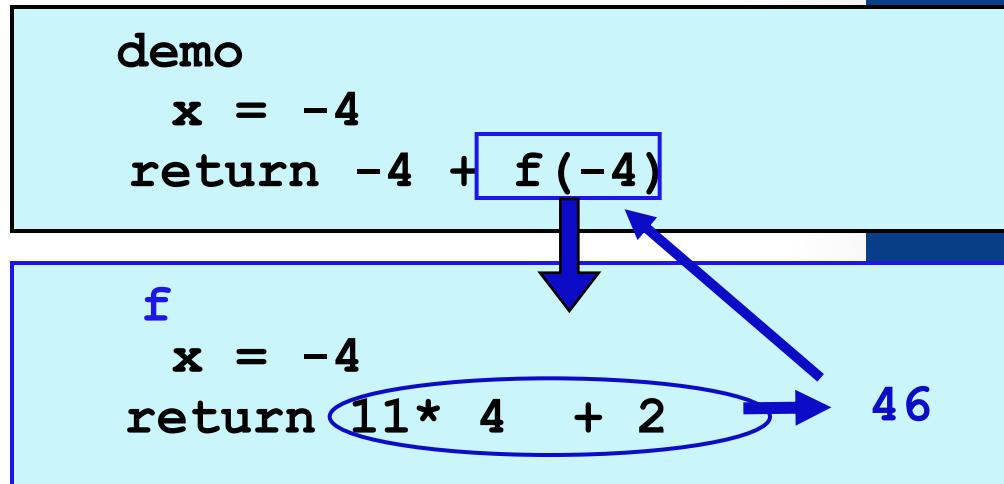
```
>>> demo(-4) ?
```



How functions work...

```
def demo(x):  
    return x + f(x)  
  
def f(x):  
    return 11*g(x) + g(x/2)  
  
def g(x):  
    return -1 * x
```

```
>>> demo(-4) ?
```

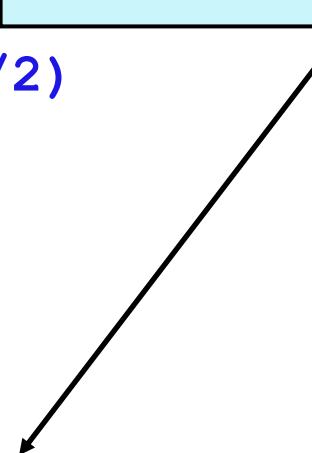


How functions work...

```
def demo(x) :  
    return x + f(x)  
  
def f(x) :  
    return 11*g(x) + g(x/2)  
  
def g(x) :  
    return -1 * x
```

```
>>> demo(-4) ————— 42  
42
```

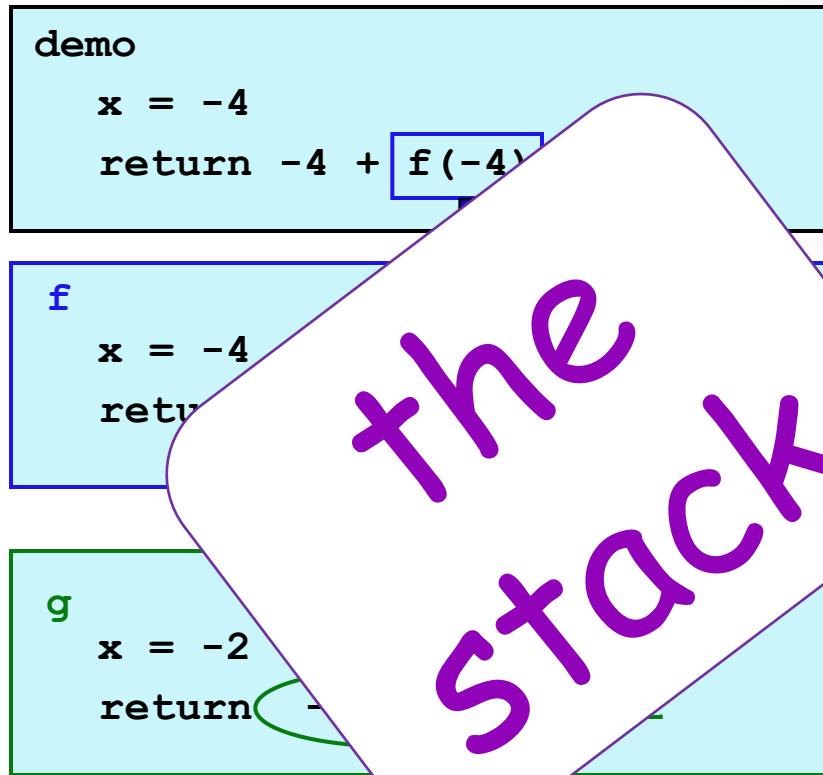
```
demo  
x = -4  
return -4 + 46
```



Function *stacking*

```
def demo(x):  
    return x + f(x)  
  
def f(x):  
    return 11*g(x) + g(x/2)  
  
def g(x):  
    return -1 * x
```

"The stack..."



- (1) keeps separate variables for each function call...
- (2) remembers where to send results back to...



```
def fac(N) :  
    if N <=1 :  
        return 1  
    return fac(N)
```

Roadsigns and recursion

examples of self-fulfilling danger

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

```
>>> fac(1)
```

Result: 1

The base case is **No Problem!**

Behind the curtain...

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

fac(5)

Behind the curtain...

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

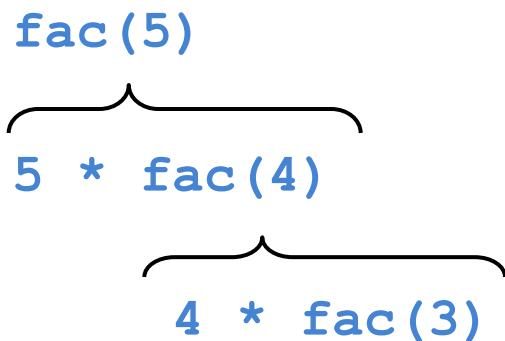
Behind the curtain...

$\text{fac}(5)$

 $5 * \text{fac}(4)$

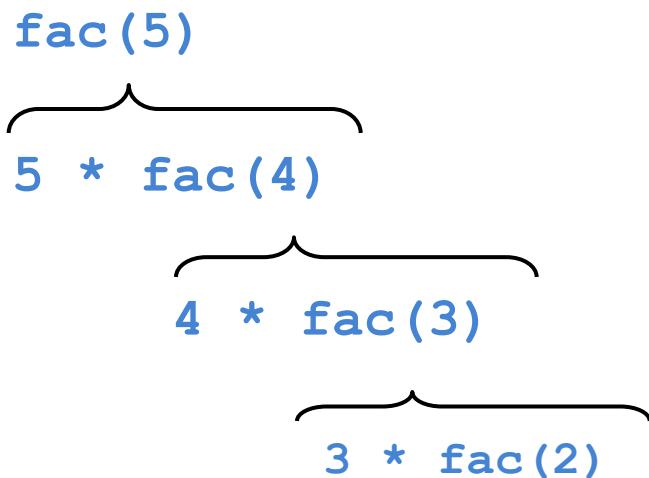
```
def fac(N):  
    if N <= 1:  
        return 1  
  
    else:  
        return N * fac(N-1)
```

Behind the curtain...



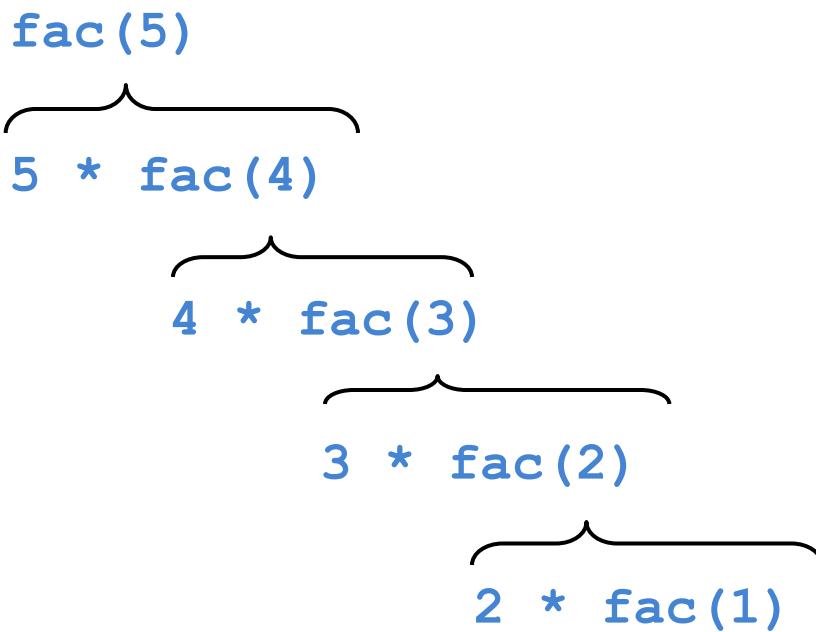
```
def fac(N):  
    if N <= 1:  
        return 1  
  
    else:  
        return N * fac(N-1)
```

Behind the curtain...



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

Behind the curtain...

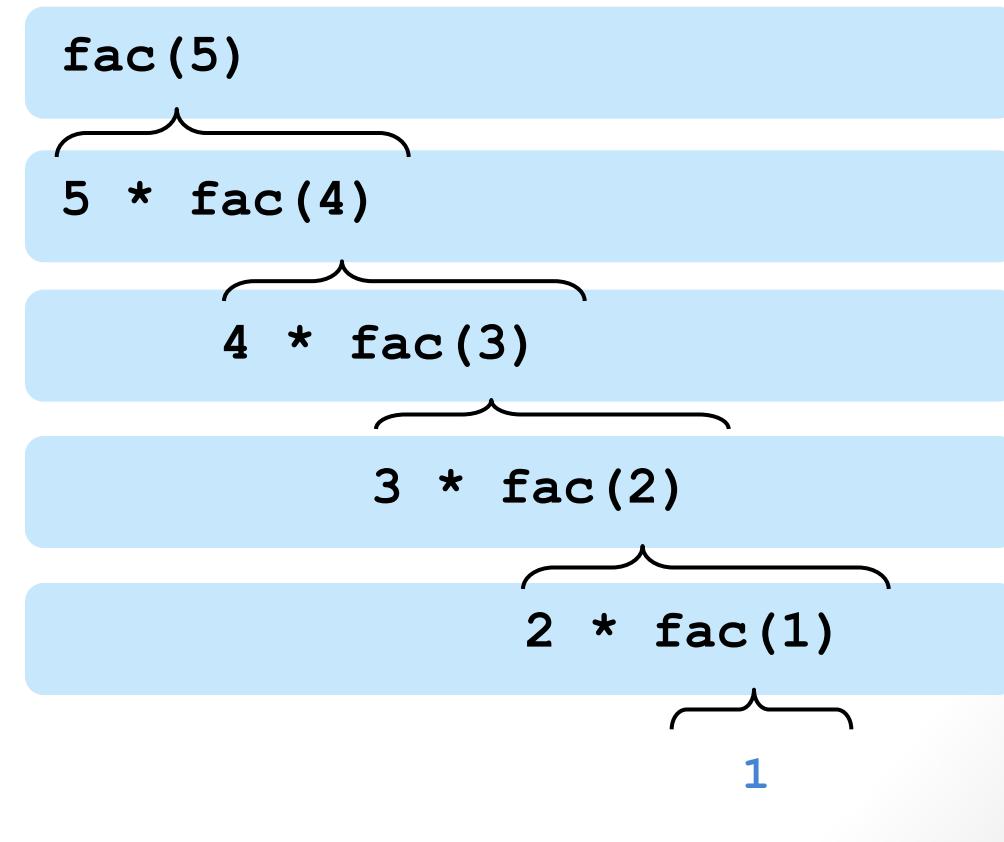


```
def fac(N):  
    if N <= 1:  
        return 1  
  
    else:  
        return N * fac(N-1)
```

Behind the curtain...

"The Stack"

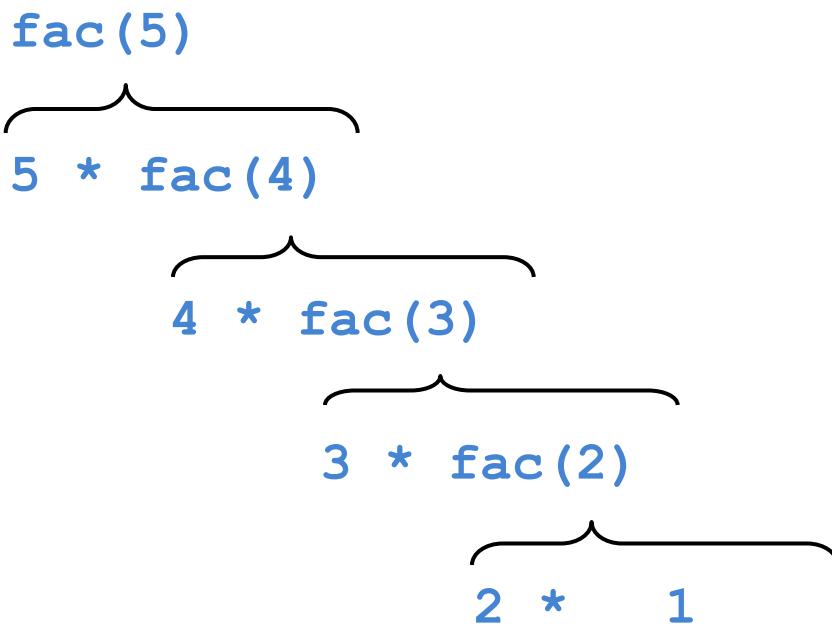
Remembers
all of the
individual
calls to `fac`



1

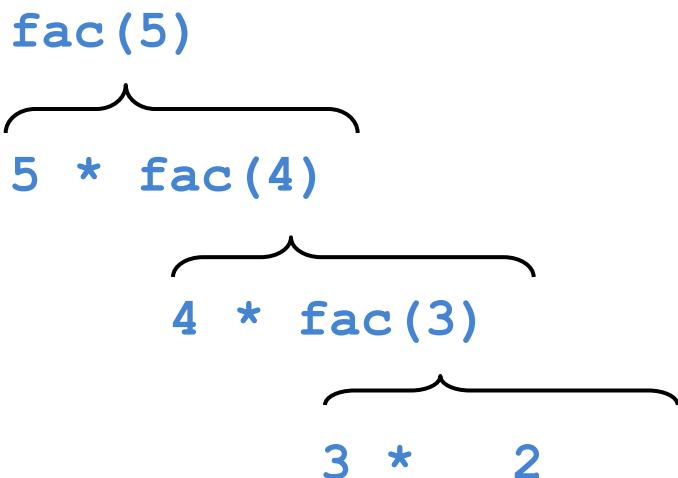
```
def fac(N):  
    if N <= 1:  
        return 1  
  
    else:  
        return N * fac(N-1)
```

Behind the curtain...



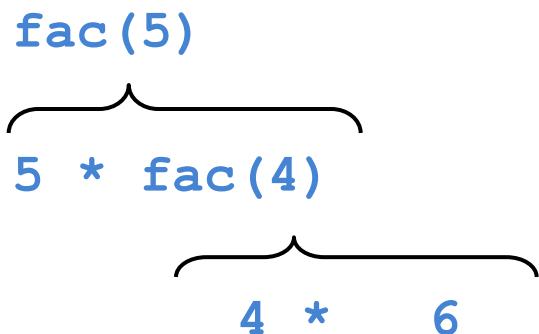
```
def fac(N):  
    if N <= 1:  
        return 1  
  
    else:  
        return N * fac(N-1)
```

Behind the curtain...



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

Behind the curtain...



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

Behind the curtain...

$$\overbrace{5 \ * \ 24}^{\text{fac}(5)}$$

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

fac(5)

Result: 120

Behind the curtain...

Let recursion do the work for you.

Exploit self-similarity
Produce short, elegant code

} **Less work !**

Let recursion do the work for you.

Exploit self-similarity
Produce short, elegant code } Less work !

```
def fac(N):  
    if N <= 1:  
        return 1  
    else:  
        rest = fac(N-1)  
        return rest * N
```

You handle the base case – the easiest case!
Recursion does almost all of the rest of the problem!
You specify one step progress towards the base case

But you *do* need to do one step yourself...

```
def fac(N):  
  
    if N <= 1:  
        return 1  
    else:  
        return fac(N)
```

This will not work !