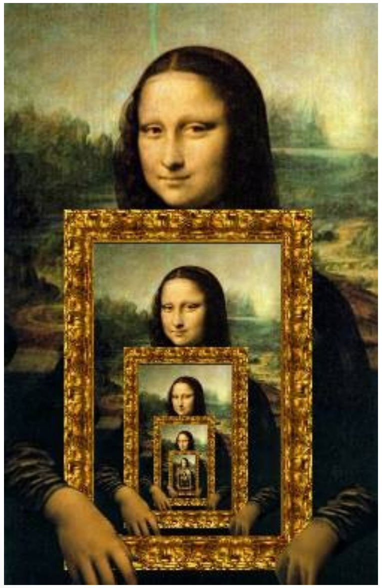
Recursion

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

As close as CS gets to magic

Let recursion draw you in….

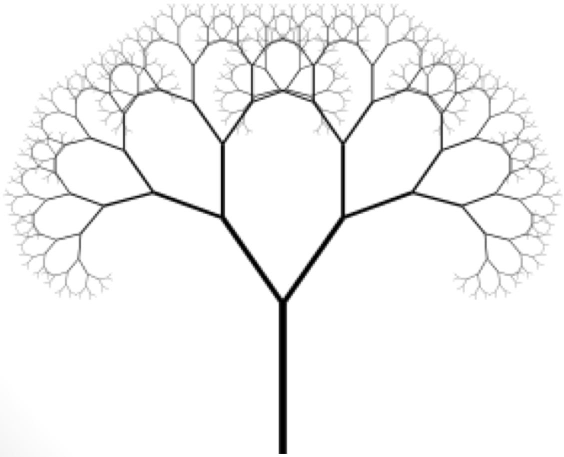
- Recursion occurs when a thing is defined in terms of itself
- Identify the “recursive structure” in these pictures by describing them



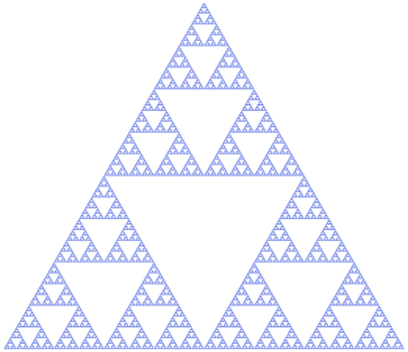
Recursion:
Strategy for solving problems in CS!

- 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

Recursive algorithms: an everyday example

```
To wash the dishes in the sink
    If there are no more dishes:
        you are done!
    else:
        Wash the dish on top of the stack
        Wash the remaining dishes in the sink
```

Function ***design***

Thinking ***sequentially***

factorial

```
5!  =  120

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

```
N!  =  N * (N-1) * (N-2) * … * 3 * 2 * 1
```

Thinking ***recursively***

Recursion == ***self***-reference!

factorial

```
5!   =   120

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

5!   =
```

Thinking ***recursively***

factorial

Recursion == ***self***-reference!

```
N!  =  N * (N-1) * (N-2) * … * 3 * 2 * 1
```

Which of the following is a recursive definition of N!?

A. **N! = N* (N-1)! , for N >0, N! = 1 for N=0**

B. **N! = N* (N-1)! , for N >=0**

C. **N! = N!**

D. **N! = (N+1)! *N**

E. **None of the above**

Thinking *recursively*

`N!  =  N * (N-1) * (N-2) * … * 3 * 2 * 1`

Strategy to implement functions recursively:

- Step 1: Implement the stopping condition (usually solution to smallest inputs)
- Step 2: To solve for a general input
  - Step 2a: Assume your function works for all smaller inputs
  - Step 2b: Recurse: Use (2a) to solve for any general input.

This involves recursive calling the function on arguments that are "closer" to the base cases

Designing Recursive Functions

Step 1: Implement the function for trivial case

```
def fac(N):

    if N <= 1:
        return 1
```

Base case:

Solution to inputs where the answer is trivial

Designing Recursive Functions

Step 2a) Assume your function works for all smaller inputs!

```
def fac(N):

    if N <= 1:
        return 1

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

Base case

Designing Recursive Functions

Step 2b) Recurse: Use that solution to solve your original problem

```
def fac(N):

    if N <= 1:
        return 1

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

Base case

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

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

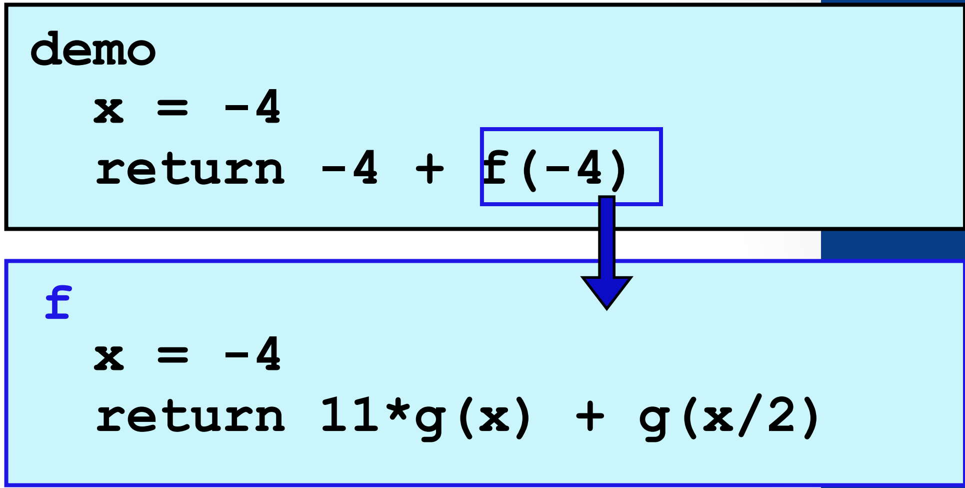
How functions work…

```
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)  ?
```



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

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

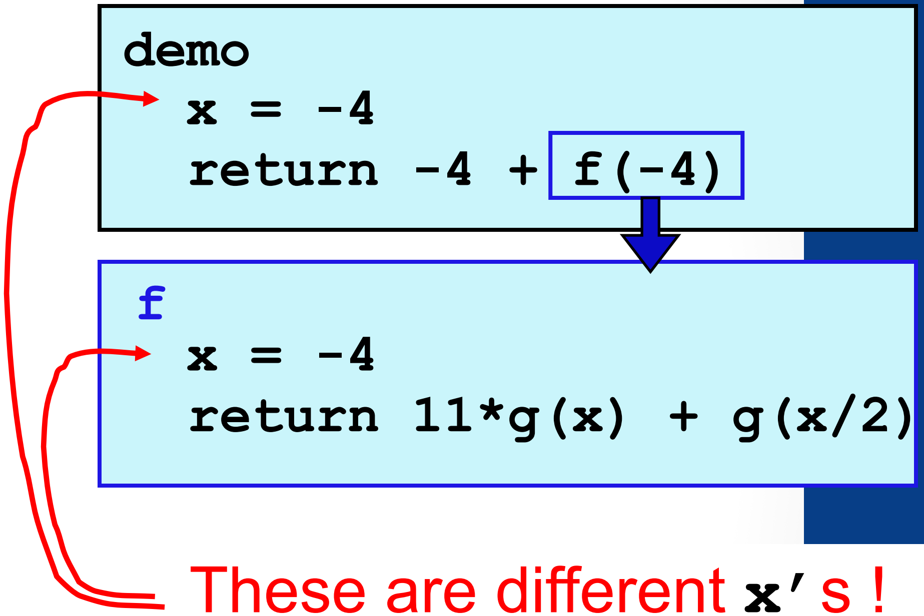
How functions work…

```
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 !

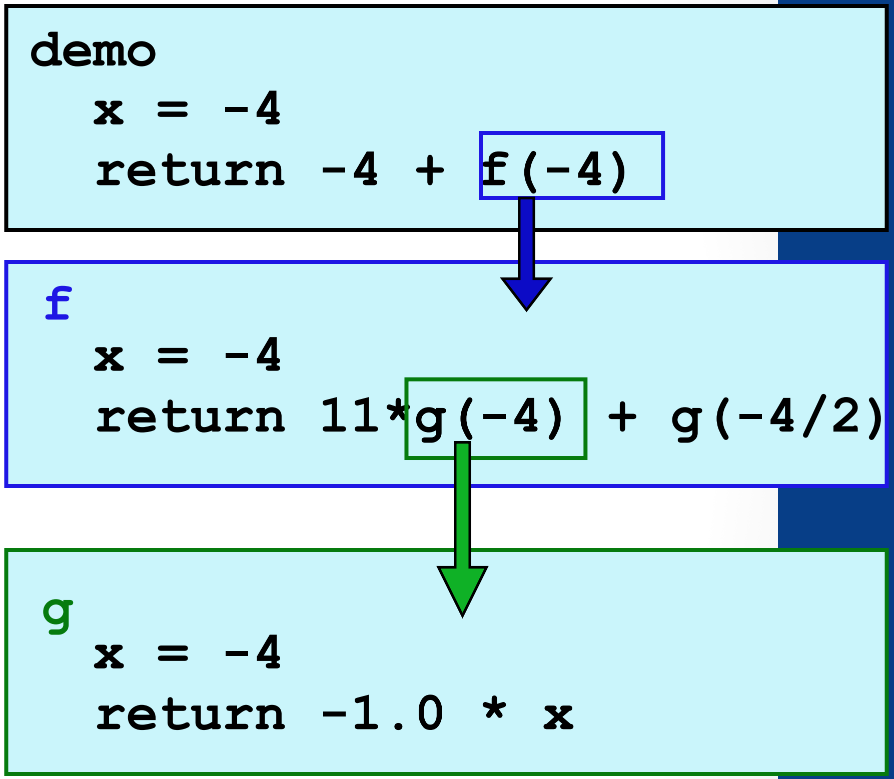
How functions work…

```
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
```

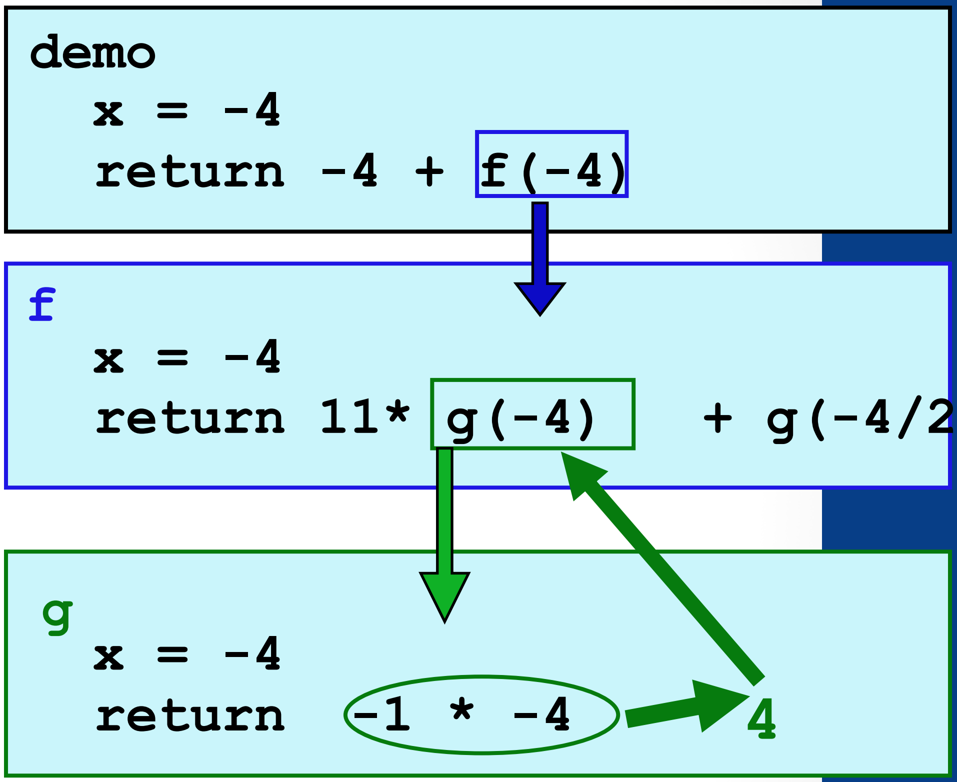
How functions work…

```
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) ?
```



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

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

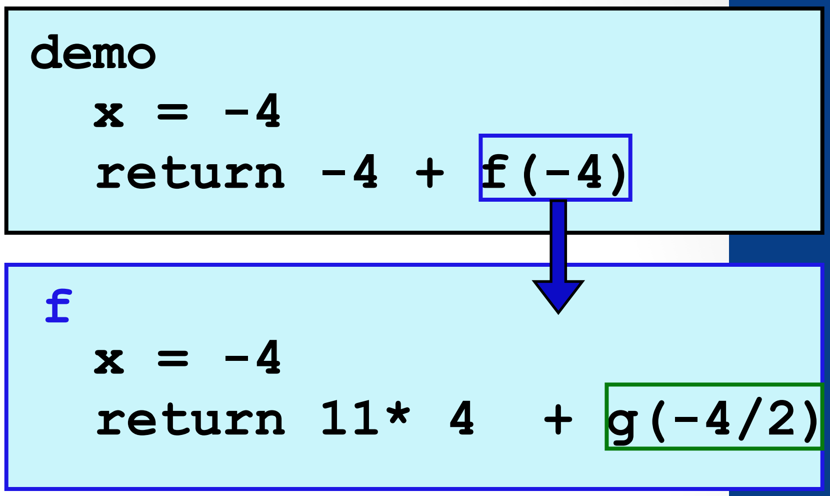
```
g
   x = -4
   return  -1 * -4     4
```

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

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

`>>> 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

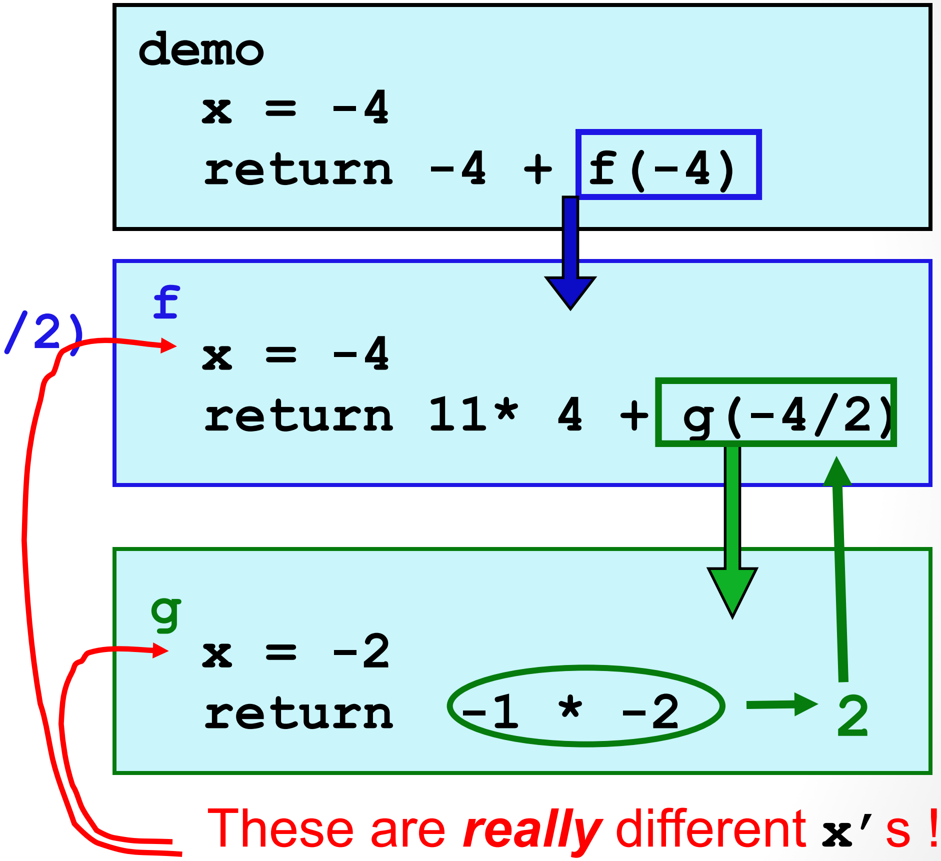
How functions work…

```
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* 4 + g(-4/2)

g
   x = -2
   return  -1 * -2 → 2
```

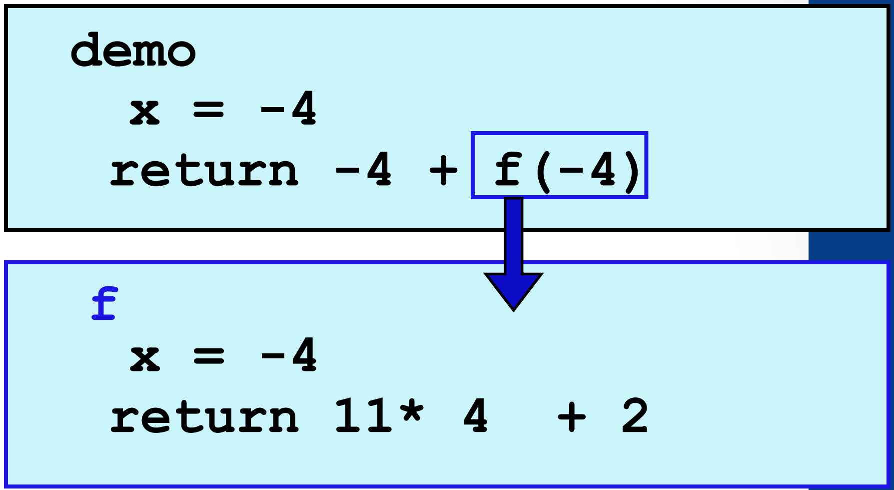
These are ***really*** different `x`’s !

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

```
f
  x = -4
  return 11* 4  + 2
```

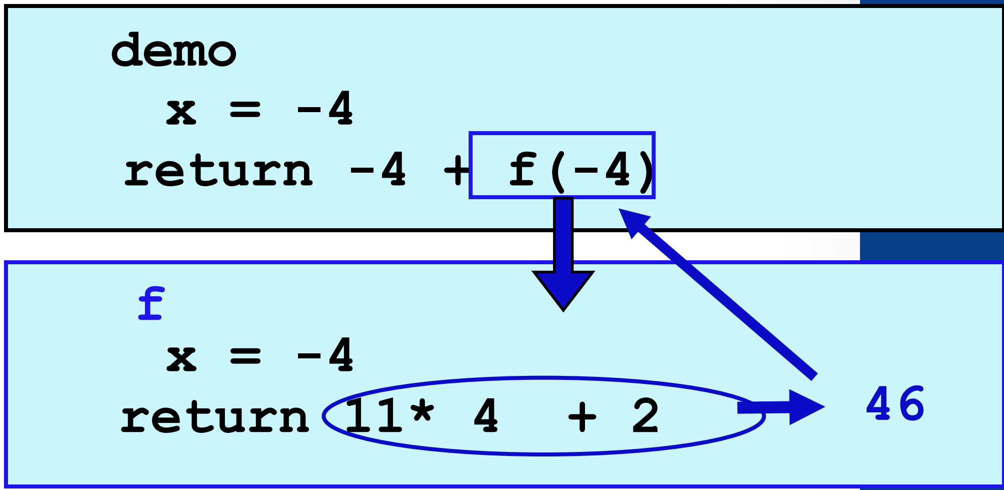
```
>>> demo(-4)  ?
```

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

```
f
  x = -4
 return 11* 4  + 2    46
```

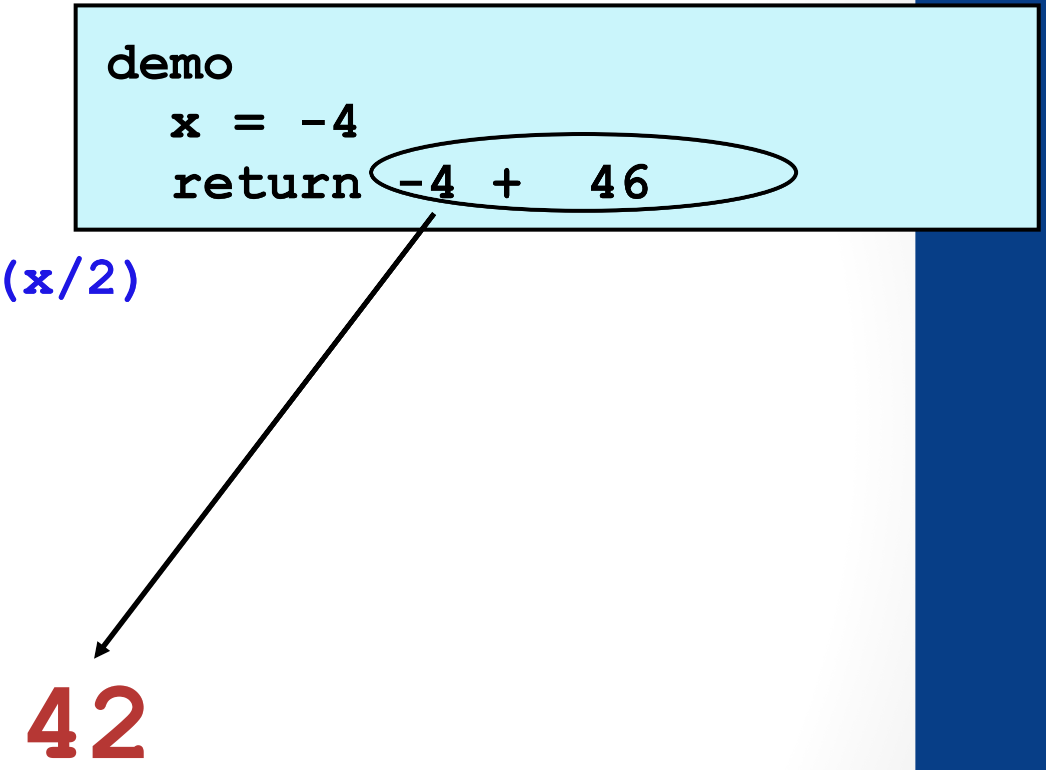
```
>>> demo(-4)  ?
```

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

```
>>> demo(-4)
42
```

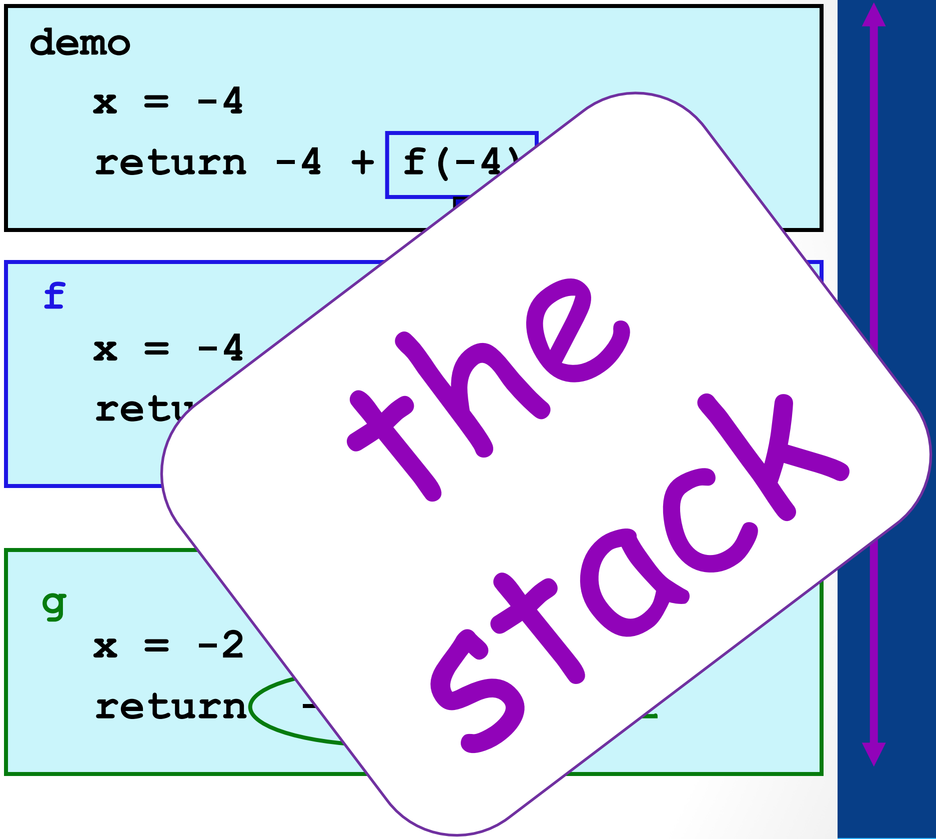
42

Function ***stack***ing

```
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

Behind the curtain…

```
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)
```

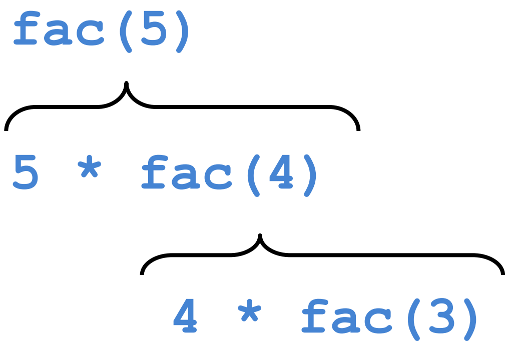
$$\overbrace{\texttt{5 * fac(4)}}^{\texttt{fac(5)}}$$

Behind the curtain…

```
def fac(N):

    if N <= 1:
        return 1

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



fac(5)

5 * fac(4)

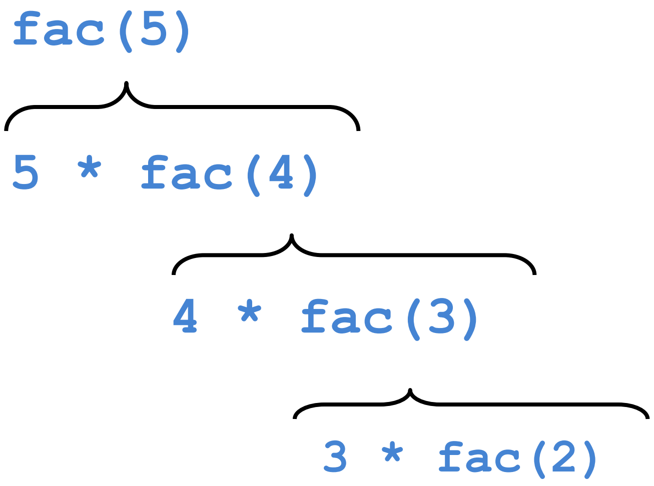
4 * fac(3)

Behind the curtain…

```
def fac(N):

    if N <= 1:
        return 1

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



fac(5)

5 * fac(4)

4 * fac(3)

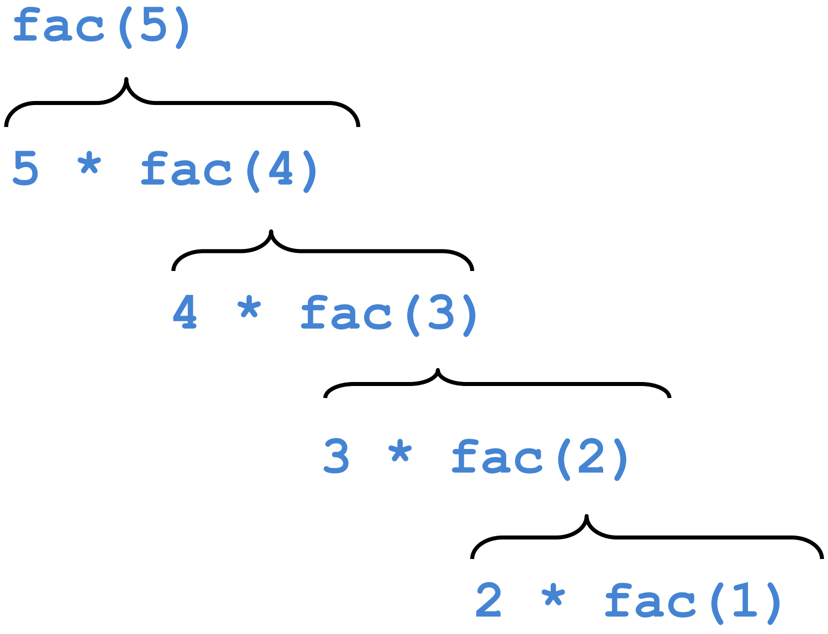
3 * fac(2)

Behind the curtain…

```
def fac(N):

    if N <= 1:
        return 1

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



fac(5)

5 * fac(4)

4 * fac(3)

3 * fac(2)

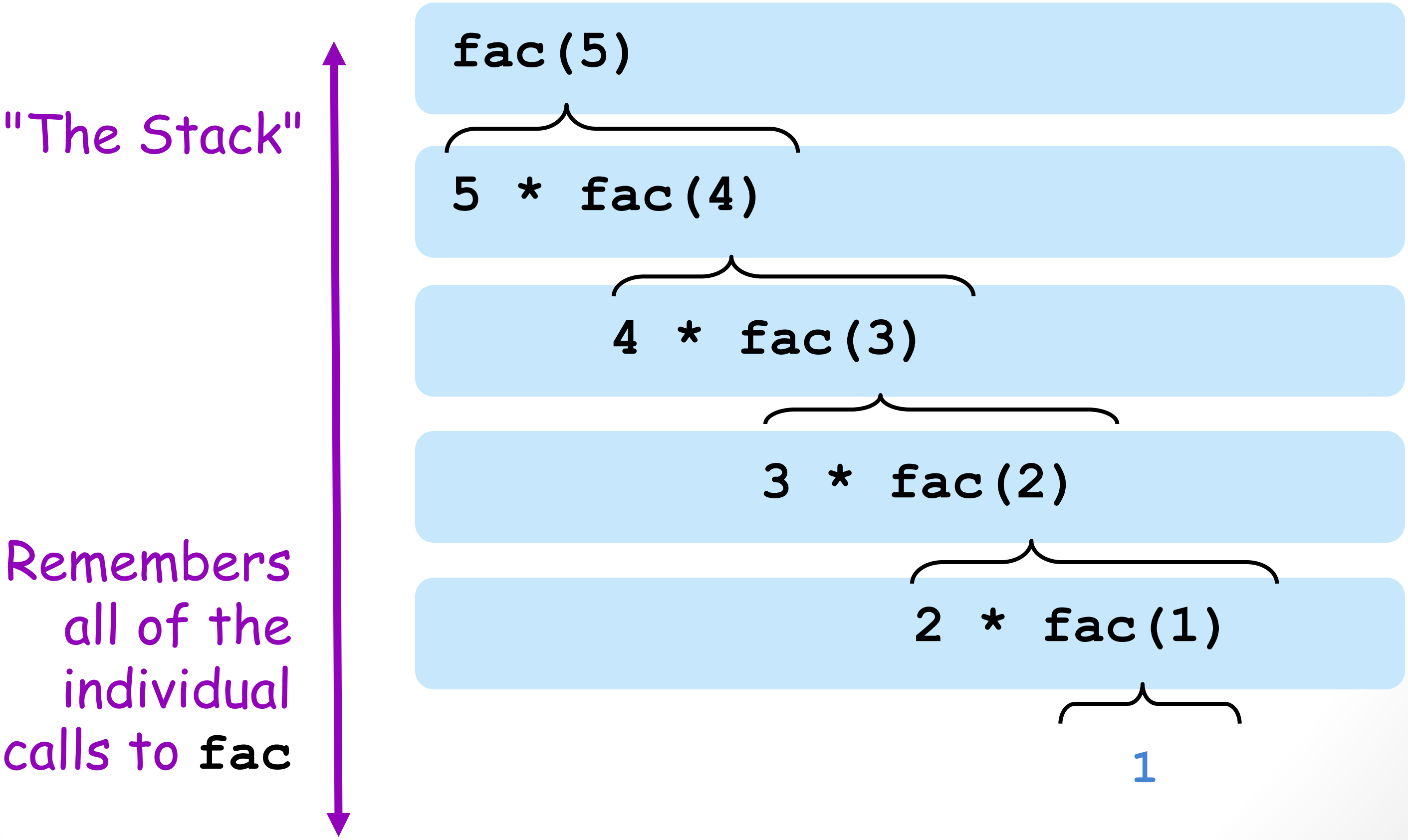
2 * fac(1)

Behind the curtain…

```
def fac(N):

    if N <= 1:
        return 1

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



"The Stack"

Remembers all of the individual calls to `fac`

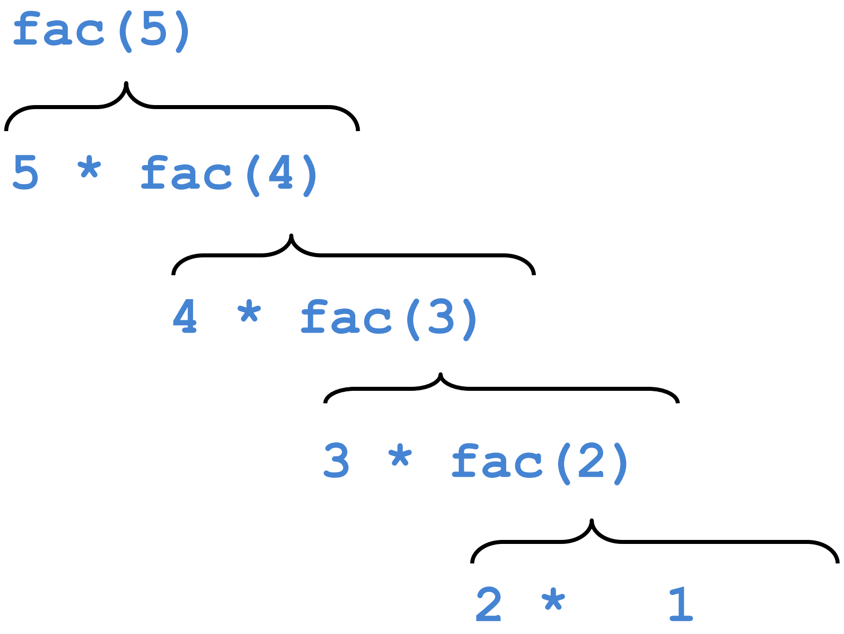
```
fac(5)
5 * fac(4)
    4 * fac(3)
        3 * fac(2)
            2 * fac(1)
                  1
```

Behind the curtain…

```
def fac(N):

    if N <= 1:
        return 1

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



```
fac(5)

5 * fac(4)

     4 * fac(3)

          3 * fac(2)

               2 *   1
```

Behind the curtain…

```
def fac(N):

    if N <= 1:
        return 1

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

```
                fac(5)

                5 * fac(4)

                    4 * fac(3)

                        3 *   2
```

Behind the curtain…

```
def fac(N):

    if N <= 1:
        return 1

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

```
fac(5)

5 * fac(4)

       4 *   6
```

Behind the curtain…

```
def fac(N):

    if N <= 1:
        return 1

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

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

Behind the curtain…

```
def fac(N):

    if N <= 1:
        return 1

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

```
fac(5)
```

Result: `120`

Let recursion do the work for you.

Exploit self-similarity

Produce short, elegant code

$\Bigg\}$ **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 !