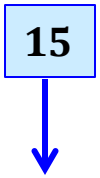
How functions work...



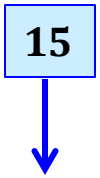

```
def demo(x):
    y = x // 3
    z = g(y)
    return z + y + x

def g(x):
    result = 4*x + 2
    return result
```

"the stack"

*they **stack**.*

How functions work...




```
def demo(x):
    y = x // 3
    z = g(y)
    return z + y + x

def g(x):
    result = 4*x + 2
    return result
```

"the stack"

stack frame

call: `demo(15)`

local variables:

```
x = 15
y = 5
z = ?????
```

*they **stack.***

How functions work...

15

```
def demo(x):
    y = x // 3
    z = g(y)
    return z + y + x

def g(x):
    result = 4*x + 2
    return result
```

"the stack"

call: `demo(15)` — stack frame

local variables:

```
x = 15
y = 5
z = ?????
```

call: `g(5)` — stack frame

local variables:

```
x = 5
result = 22
```

returns 22

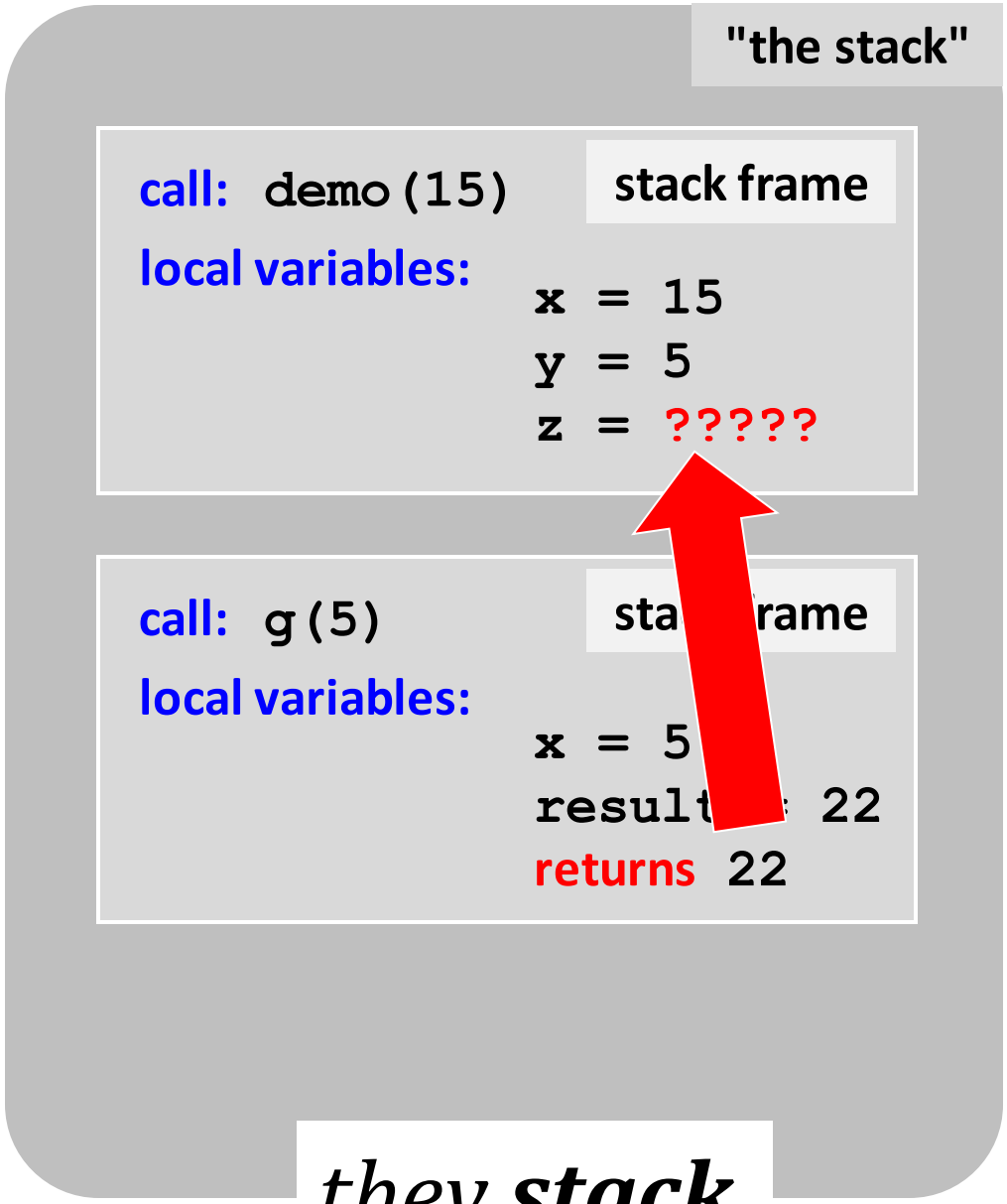
*they **stack.***

How functions work...

15

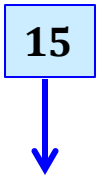
```
def demo(x):
    y = x // 3
    z = g(y)
    return z + y + x

def g(x):
    result = 4*x + 2
    return result
```



*they **stack.***

How functions work...




```
def demo(x):
    y = x // 3
    z = g(y)
    return z + y + x

def g(x):
    result = 4*x + 2
    return result
```

"the stack"

stack frame

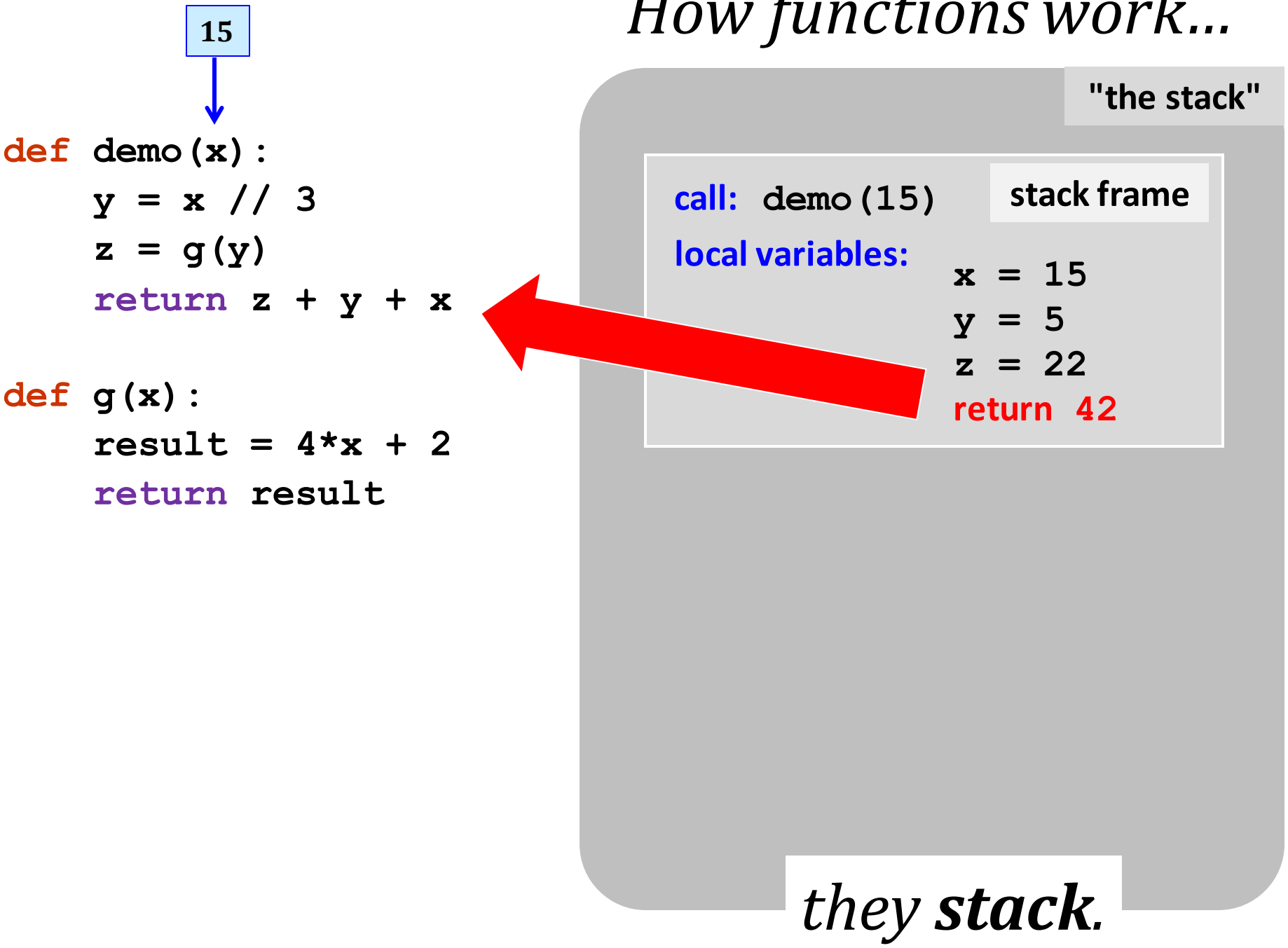
call: `demo(15)`

local variables:

```
x = 15
y = 5
z = 22
```

*they **stack.***

How functions work...



How functions work...

15

```
def demo(x):
    y = x // 3
    z = g(y)
    return z + y + x

def g(x):
    result = 4*x + 2
    return result
```

42

output

"the stack"

afterwards, the stack is empty..., but ready if another function is called

*they **stack**.*

How functions work...

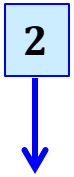
2

what's `f(2)` ?

```
def f(x):
    if x == 0:
        return 12
    else:
        return f(x - 1) + 10*x
```

"the stack"

How functions work...




```
def f(x):
    if x == 0:
        return 12
    else:
        return f(x - 1) + 10*x
```

"the stack"

stack frame

call: `f(2)`

local variables:

`x = 2`

need f(1)

How functions work...

1

```
def f(x):
    if x == 0:
        return 12
    else:
        return f(x - 1) + 10*x
```

"the stack"

stack frame

call: `f(2)`

local variables:

`x = 2`

need f(1)

stack frame

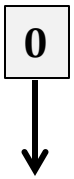
call: `f(1)`

local variables:

`x = 1`

need f(0)

How functions work...




```
def f(x):
    if x == 0:
        return 12
    else:
        return f(x - 1) + 10*x
```

"the stack"

call: `f(2)` — stack frame

local variables:

`x = 2`

need f(1)

call: `f(1)` — stack frame

local variables:

`x = 1`

need f(0)

call: `f(0)` — stack frame

local variables:

`x = 0`

returns 12

How functions work...




```
def f(x):
    if x == 0:
        return 12
    else:
        return f(x - 1) + 10*x
```

"the stack"

call: `f(2)`

local variables:

stack frame

`x = 2`

need f(1)

call: `f(1)`

local variables:

stack frame

`x = 1`

need f(0)

call: `f(0)`

local variables:

sta frame

`x = 0`

returns 12

How functions work…

1

```
def f(x):
    if x == 0:
        return 12
    else:
        return f(x - 1) + 10*x
```

"the stack"

call: `f(2)` — stack frame

local variables:

`x = 2`

need f(1)

call: `f(1)` — stack frame

local variables:

`x = 1`

f(0) = 12

result =

How do we compute the result?

How functions work...

1

```
def f(x):
    if x == 0:
        return 12
    else:
        return f(x - 1) + 10*x
```

"the stack"

call: `f(2)`

stack frame

local variables:

`x = 2`

need f(1)

call: `f(1)`

stack frame

local variables:

`x = 1`

f(0) = 12

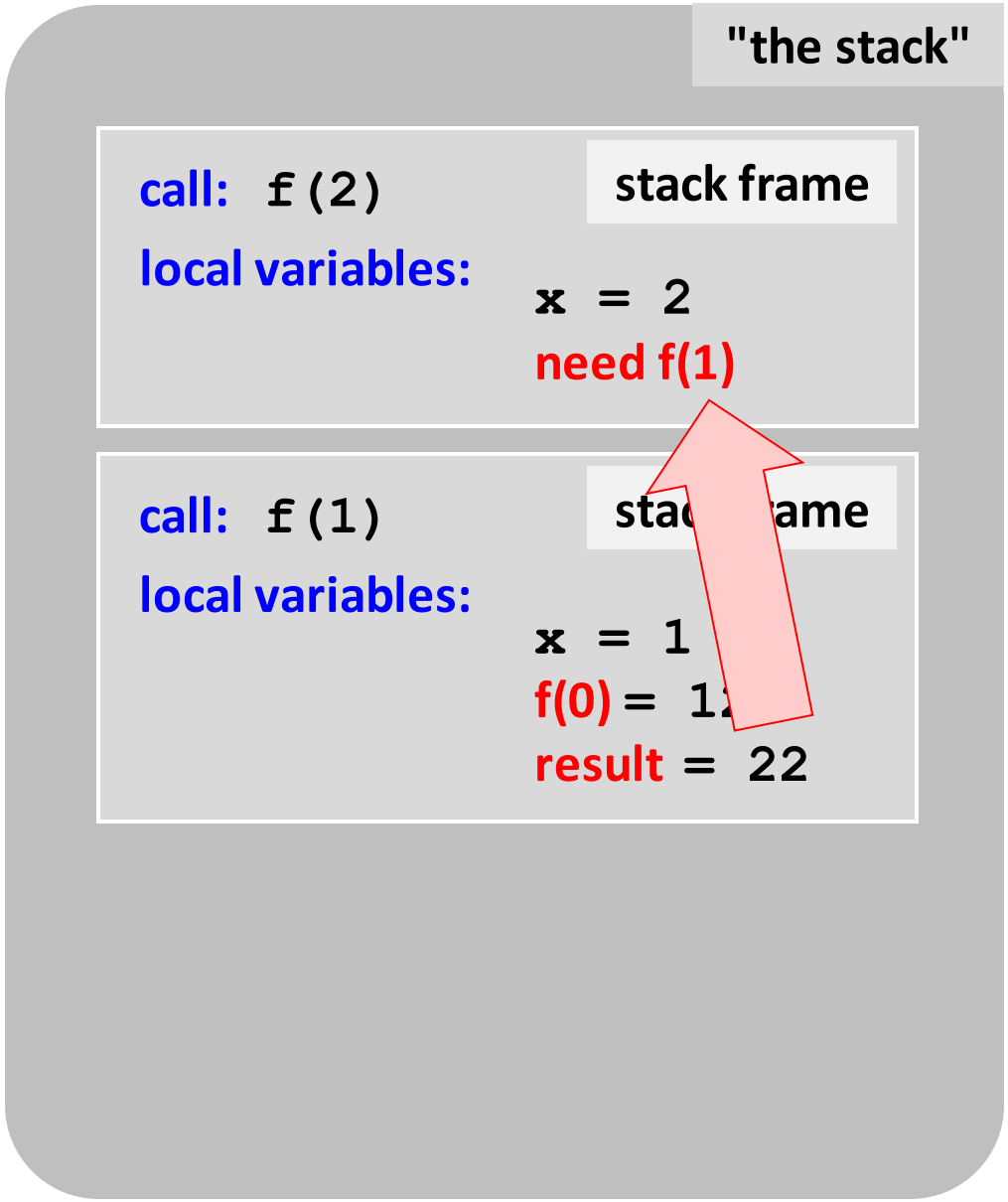
result = 22

Where does that result go?

How functions work...

1

```
def f(x):
    if x == 0:
        return 12
    else:
        return f(x - 1) + 10*x
```



How functions work...

2

```
def f(x):
    if x == 0:
        return 12
    else:
        return f(x - 1) + 10*x
```

"the stack"

stack frame

call: `f(2)`

local variables:

`x = 2`

f(1) = 22

result =

What's *this* return value?

How functions work...

2

```
def f(x):
    if x == 0:
        return 12
    else:
        return f(x - 1) + 10*x
```

"the stack"

stack frame

call: `f(2)`

local variables:

`x = 2`

f(1) = 22

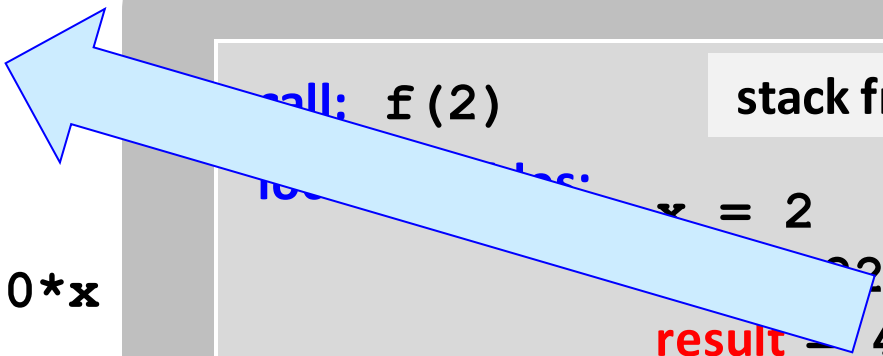
result = 42

which then gets returned...

How functions work...

2

```
def f(x):
    if x == 0:
        return 12
    else:
        return f(x - 1) + 10*x
```



"the stack"

stack frame

f(2)

x = 2

result = 42

the result then gets returned...

How functions work...

2

```
def f(x):
    if x == 0:
        return 12
    else:
        return f(x - 1) + 10*x
```

42

output

"the stack"

again, the stack is empty, but ready if another function is called...

Functions are like cells: software's building blocks...

*... each one, a **self-contained** computational unit!*

*functions **stack**.*

function ***design***

Thinking ***recursively***

factorial

`5!  =  120`

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

`fac(5) =`

can we express **`fac`** *w/ a smaller version of itself?*

`fac(N) = N * (N-1) * … * 3 * 2 * 1`

`fac(N) =`

Thinkin ly

Recursion ~ self-similarity

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

```
fac(5) = 5 * fac(4)
```

can we express **`fac`** *w/ a smaller version of itself?*

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

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

overall goal

output value of
5*4*3*2*1

fac(5)

fac(5)

Recursion,
the process

starting from 5,
decreasing to 1

5 * 4 * 3 * 2 * 1

integers from 5 down to 1,
multiplied together

fac(5)

overall goal

output value of
5*4*3*2*1

fac(5)

is

fac(5)

overall goal

value of
5

one, small piece

*

combined with

output value of
4*3*2*1

self-similar "rest"

fac(4)

Recursion, ***the process***

overall goal

output value of
`N*(N-1)*...*2*1`

fac(N)

is

fac(N)

overall goal

value of
N

one, small piece

*****

combined with

output value of
`(N-1)*...*2*1`

self-similar "rest"

fac(N-1)

Recursion, ***the process***

Recursion's *conceptual* challenge?

You need to see BOTH the ***self-similar pieces*** AND the ***whole thing*** simultaneously!





Nature loves recursion!

overall goal

output value of
`N*(N-1)*...*2*1`

fac(N)

is

fac(N)

overall goal

value of
N

one, small piece

*****

combined with

output value of
`(N-1)*...*2*1`

self-similar "rest"

fac(N-1)

fac(N)

Recursion, ***the code!***

Warning: ***this is legal!***

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

I wonder how this code will **STACK** up!?



legal != *recommended*

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

calls to `facBad` will never stop: there's no **BASE CASE**

Make sure you have a
base case

a.k.a. *"escape hatch"*

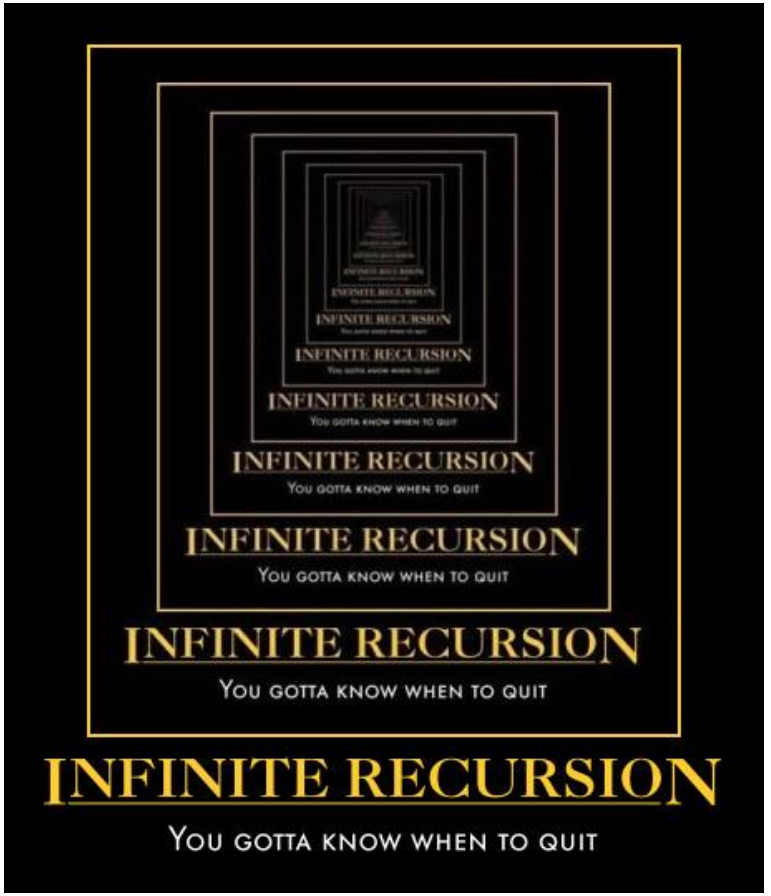


How about an escape
from recursion itself!?!



This "works" ~ *but doesn't work!*

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



Recursion

the dizzying dangers of having no base case!

overall goal

Result of
`N*(N-1)*...*2*1`

fac(N)

is

fac(N)

overall goal

value of
N

one, small piece

*****

combined with

Result of
`(N-1)*...*2*1`

self-similar "rest"

fac(N-1)

fac(N)

base case:

output for fac(`0`) is `1`

smallest possible argument

its result

Thinking recursively...

```
def fac(N):
    if N == 0:
        return 1
    else:
        return N * fac(N-1)
```

Base case

Recursive case

(*too* short?)

Acting recursively

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

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

this recursion happens first!

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

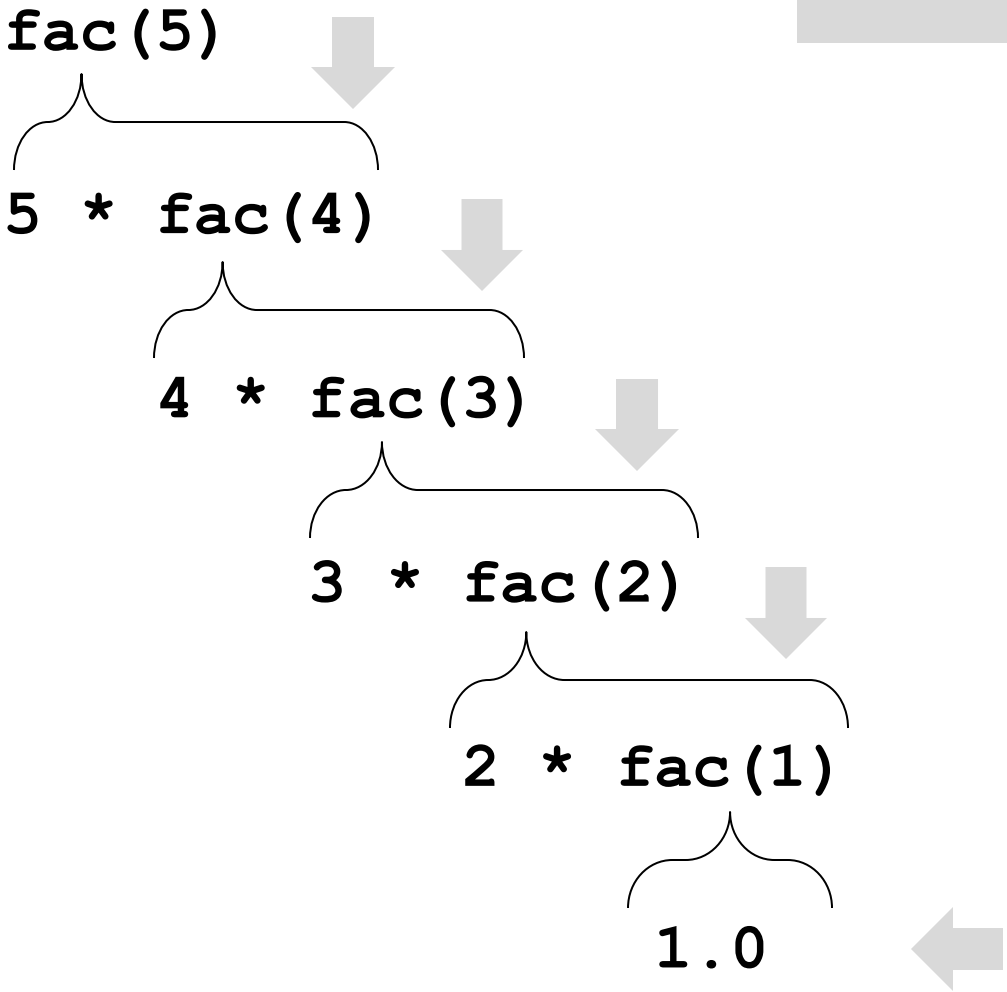
hooray for variables!

Conceptual

Actual

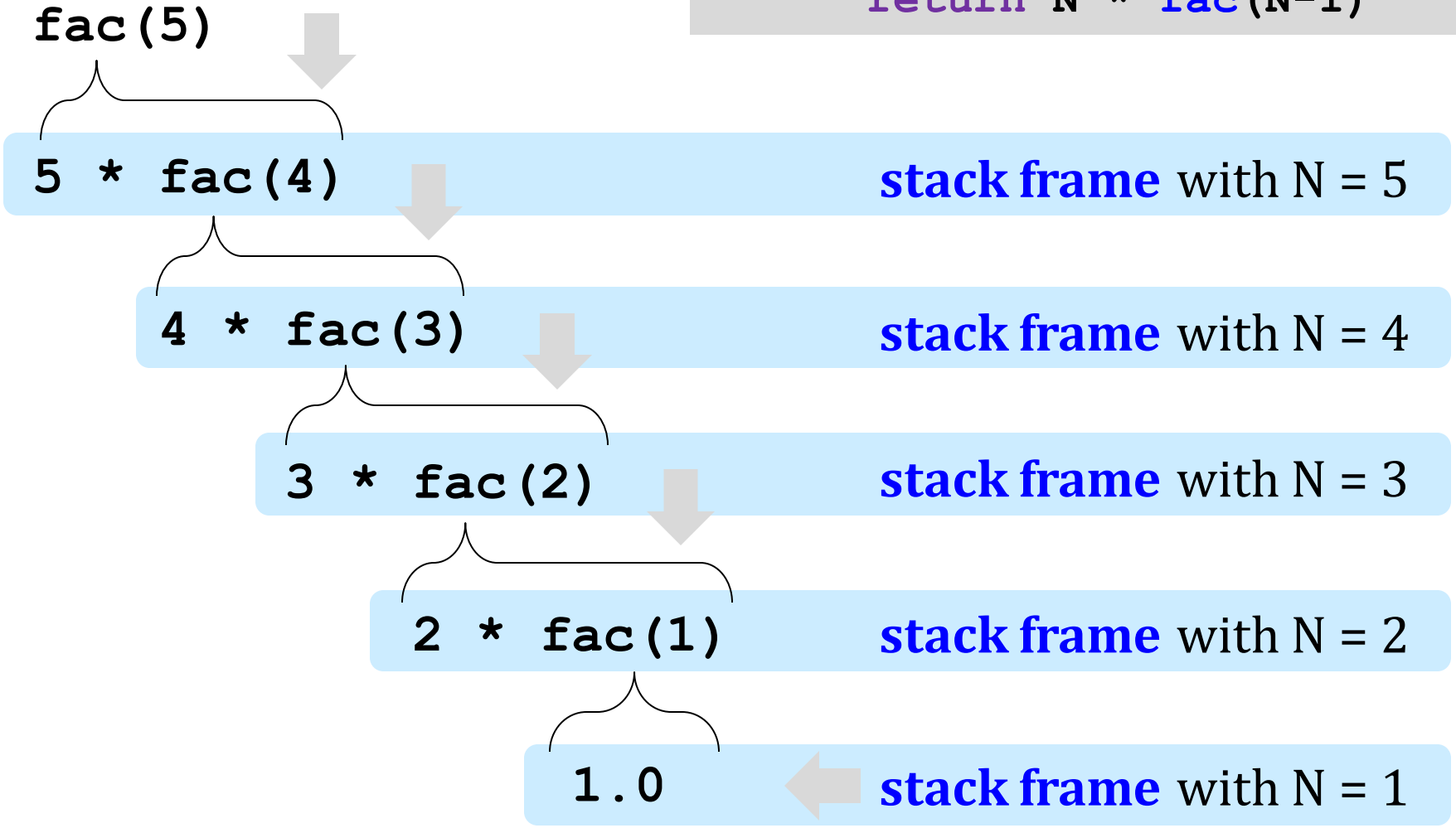
Behind the curtain:
how recursion works...

```
def fac(N):
    if N <= 1:
        return 1.0
    else:
        return N * fac(N-1)
```



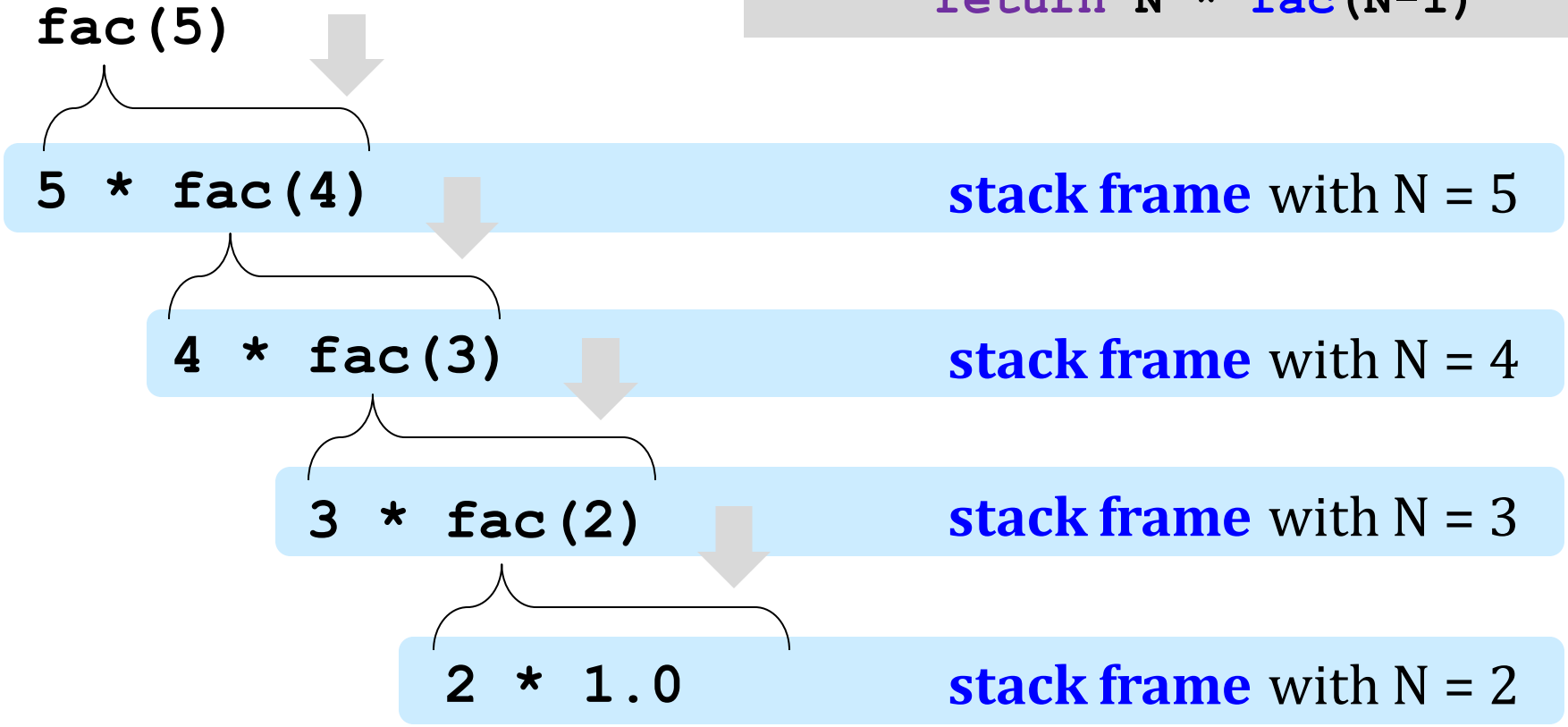
Behind the curtain:
how recursion works...

```
def fac(N):
    if N <= 1:
        return 1.0
    else:
        return N * fac(N-1)
```



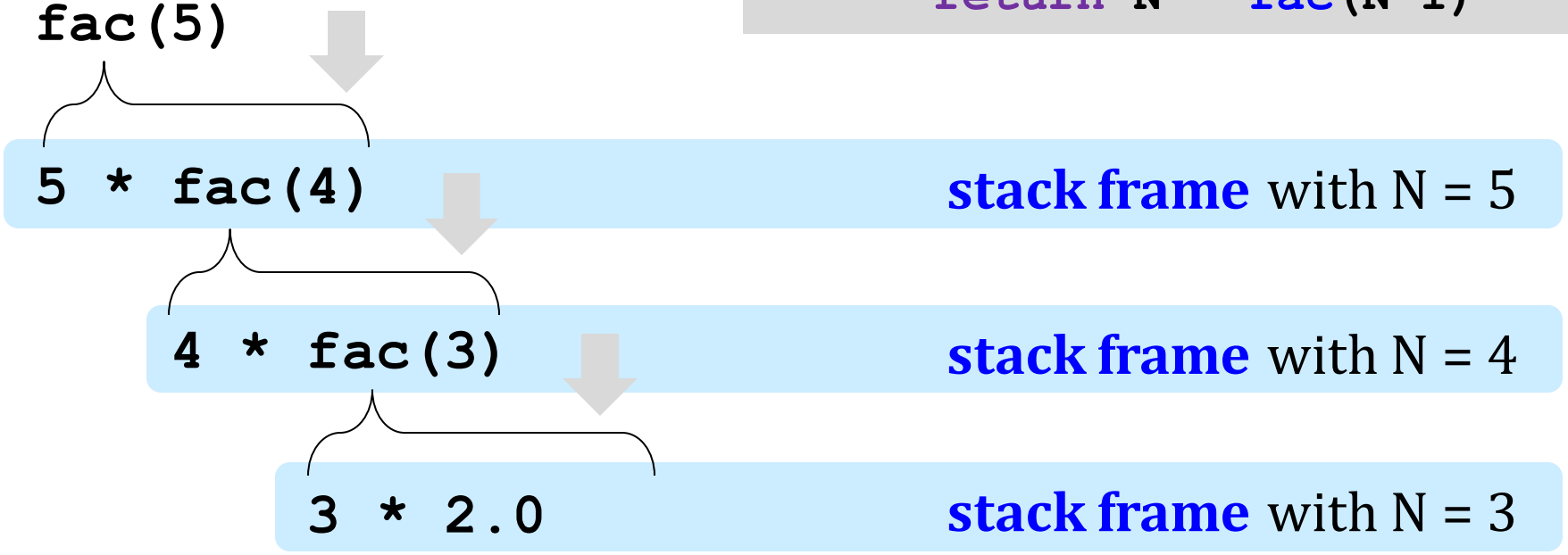
Behind the curtain:
how recursion works...

```
def fac(N):
    if N <= 1:
        return 1.0
    else:
        return N * fac(N-1)
```



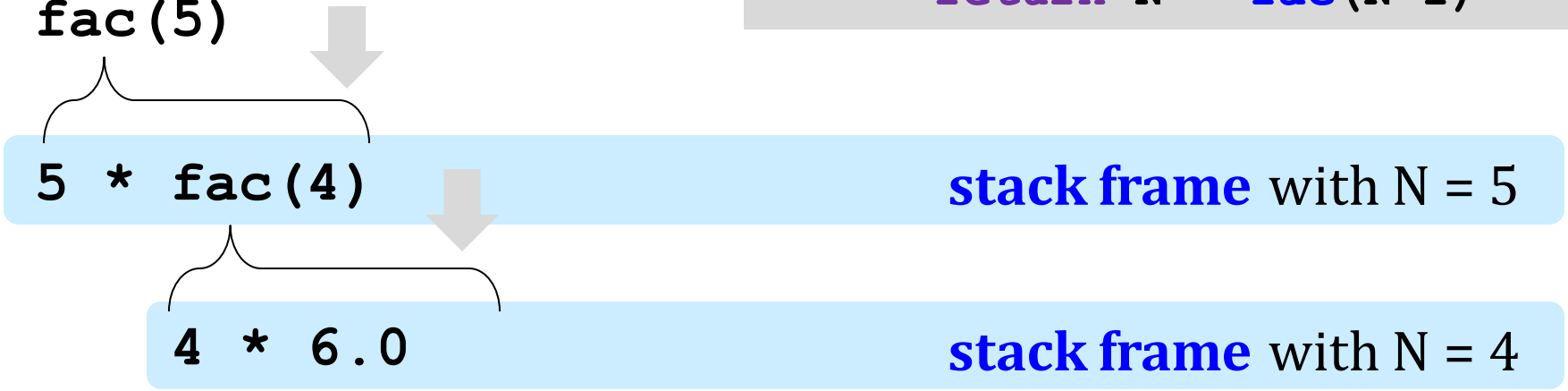
Behind the curtain:
how recursion works...

```
def fac(N):
    if N <= 1:
        return 1.0
    else:
        return N * fac(N-1)
```



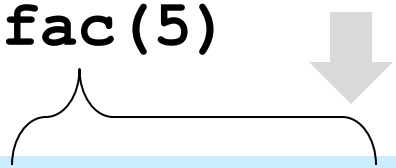
Behind the curtain:
how recursion works...

```
def fac(N):
    if N <= 1:
        return 1.0
    else:
        return N * fac(N-1)
```



Behind the curtain:
how recursion works...

```
def fac(N):
    if N <= 1:
        return 1.0
    else:
        return N * fac(N-1)
```

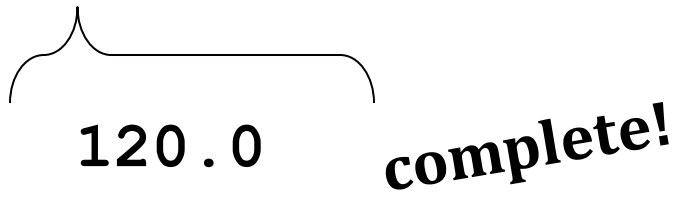



```
5 * 24.0
```

stack frame with N = 5

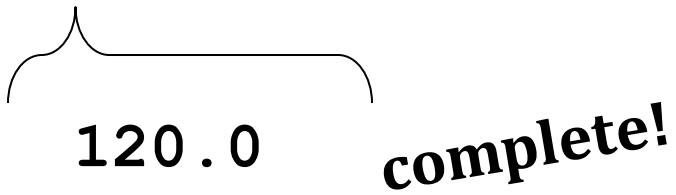
Behind the curtain:
how recursion works...

```
def fac(N):
    if N <= 1:
        return 1.0
    else:
        return N * fac(N-1)
```

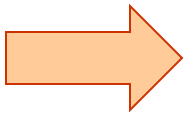


Behind the curtain:
how recursion works...

```
def fac(N):
    if N <= 1:
        return 1.0
    else:
        return N * fac(N-1)
```



Recursion, ***the design process***



Recursive Design: the 3 **i**'s



intuition

identify

self-similarities
& base cases

implement

Recursive Design: the 3 **i**'s



Examples!

intuition

What's the self-similarity?

identify

self-similarities
& base cases

Python...

implement

overall goal

output value of
5*4*3*2*1

fac(5)

fac(5)

Examples!

starting from 5,
decreasing to 1

5*4*3*2*1

integers from 5 down to 1,
multiplied together

fac(5)

overall goal

output value of
5*4*3*2*1

fac(5)

is

fac(5)

overall goal

value of
5

one, small piece

*

combined with

output value of
4*3*2*1

self-similar "rest"

fac(4)

Examples!

What's the self-similarity?

fac(N)

overall goal

output value of
`N*(N-1)*...*2*1`

is

fac(N)

What's the self-similarity?

overall goal

value of
N

one, small piece

*****

combined with

output value of
`(N-1)*...*2*1`

self-similar "rest"

fac(N-1)

What's the base case?

base case: output for fac(`0`) is `1`

smallest possible argument

its result

Python...

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

Base case

Recursive case

(*too* short?)

Recursive Design: the 3 **i**'s



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

rec.

fac(4) is 4*3*2*1

fac(3) is 3*2*1

fac(2) is 2*1

fac(1) is 1

fac(0) is 1

base case(s)

intuit

rec.

fac(N) is N*fac(N-1)

self-similarities

base cases

base cases

identify

self-similarities & base cases

```
def fac(N):
    if N <= 1:
        return 1.0
    else:
        return N * fac(N-1)
```

base case

rec.

implement

Recursive Design: the 3 **i**'s

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

rec.

fac(4) is

fac(3)

fac(2)

fac(1) = 1

fac(0) is 1

base case(s)

intuit

rec.

fac(N) is N*fac(N-1)

similarities

dentify

elf-similarities
& base cases

Let's try two more examples together – and then try 2-3 on your own...

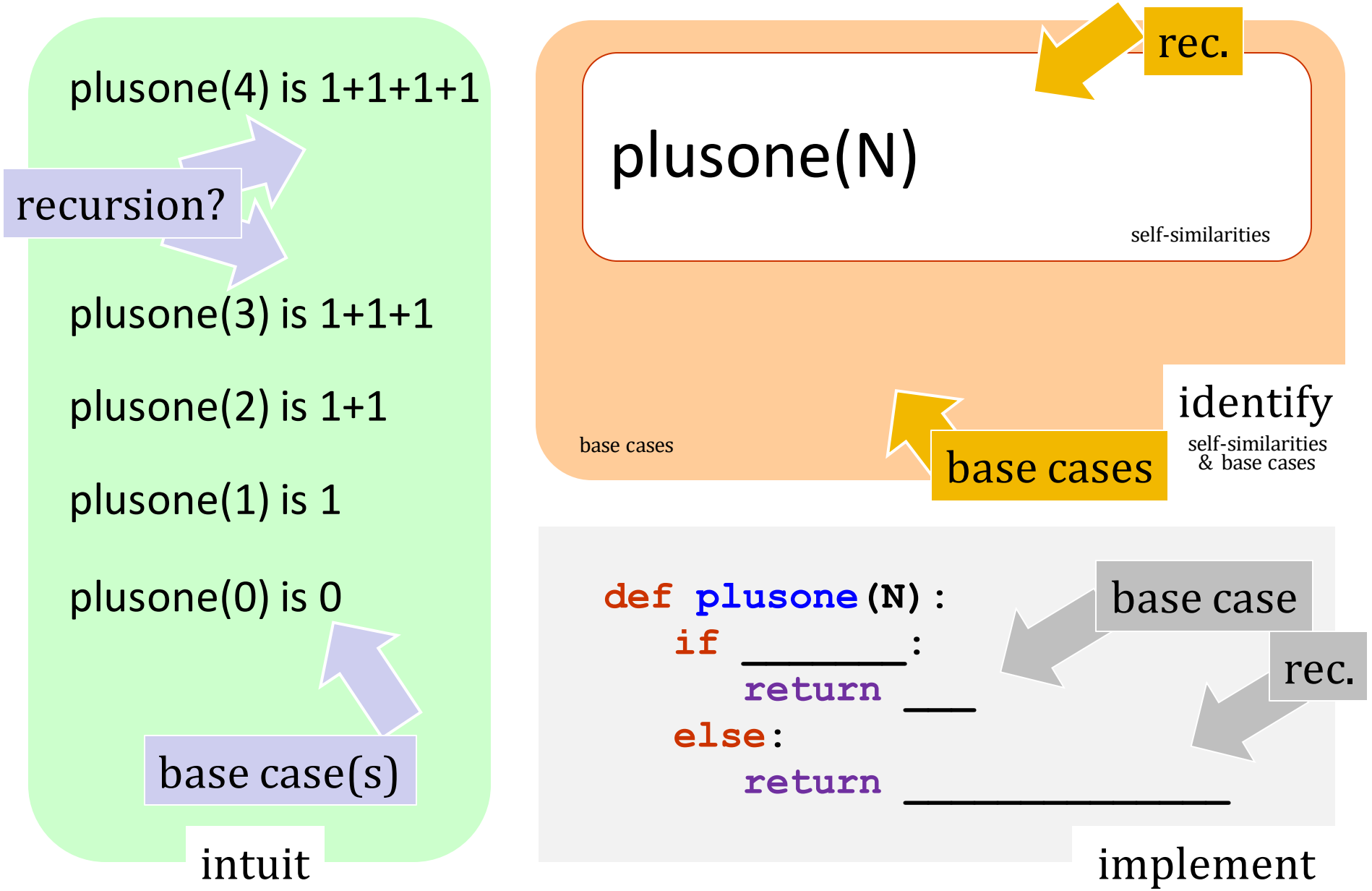
```
def fac(N):
    if N > 0:
        return N * fac(N-1)
    else:
        return 1.0
```

rec.

base case

implement

Recursive Design: the 3 **i**'s



overall goal

output value of

`1+1+1+1`

plusone(4)

plusone(4)

4 times

`1+1+1+1`

+1 added together, 4 times

plusone(4)

plusone(4)

overall goal

output value of
1+1+1+1

plusone(4)

is

overall goal

value of
1

one, small piece

+

combined with

output value of
1+1+1

self-similar "rest"

plusone(3)

example with concrete values

plusone(N)

overall goal

output value of
1+1+. . .+1
plusone(N)

is

overall goal

value of
1
one, small piece

+
combined with

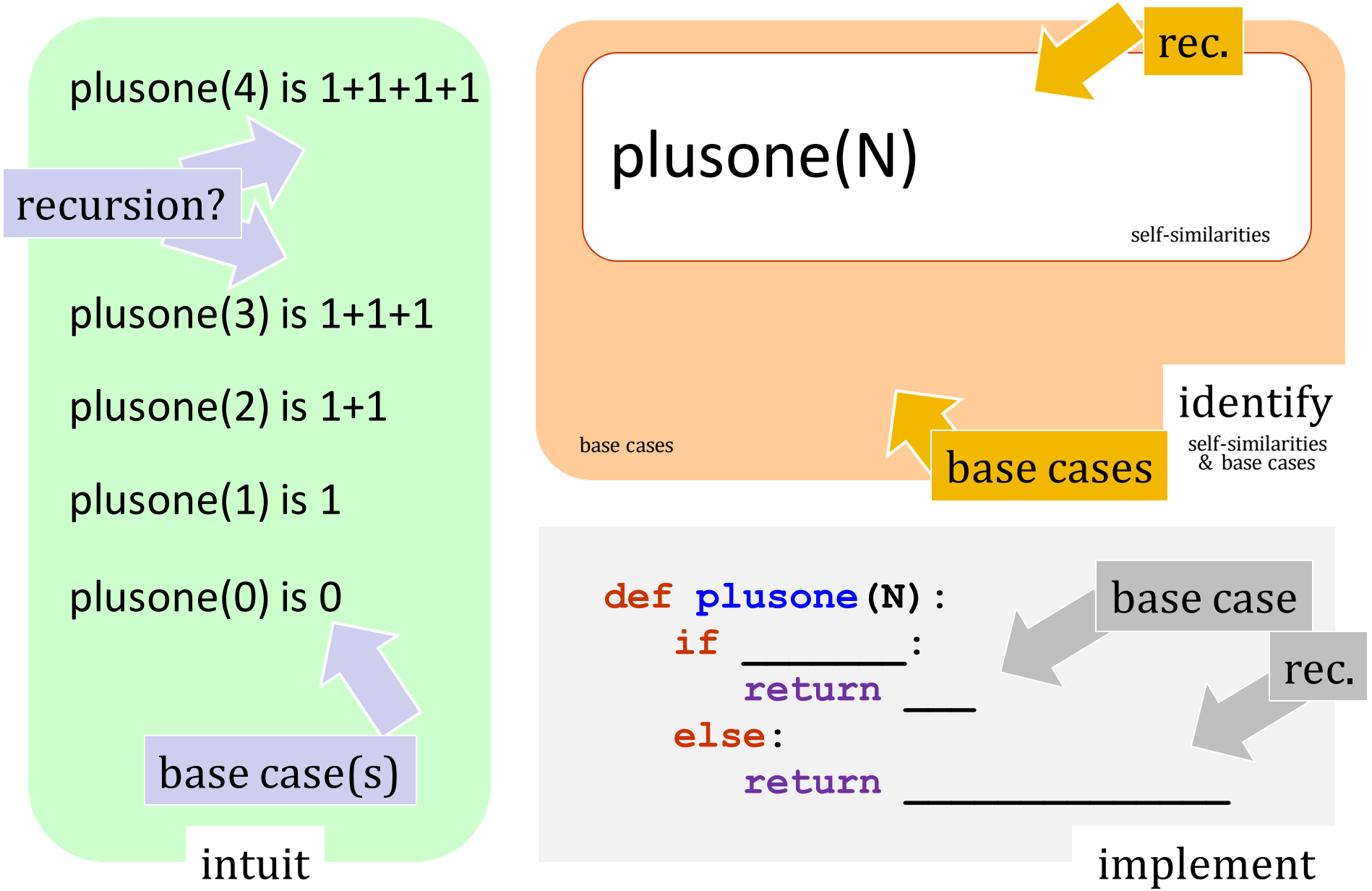
output value of
1+. . .+1
self-similar "rest"
plusone(N-1)

base case: output of plusone(0) is 0

smallest possible argument

its result

Recursive Design: the 3 **i**'s



Recursive Design: the 3 **i**'s

plusone(4) is 1+1+1+1

recursion?

plusone(3) is 1+1+1

plusone(2) is 1+1

plusone(1) is 1

plusone(0) is 0

base case(s)

intuit

rec.

plusone(N) is 1 + plusone(N-1)

self-similarities

plusone(0) is 0

base cases

base cases

identify

self-similarities & base cases

```
def plusone(N):
    if N == 0:
        return 0
    else:
        return 1 + plusone(N-1)
```

base case

rec.

implement

Recursive Design: leng(L)

leng([5, 42, 7]) is 3

recursion?

leng([42, 7]) is 2

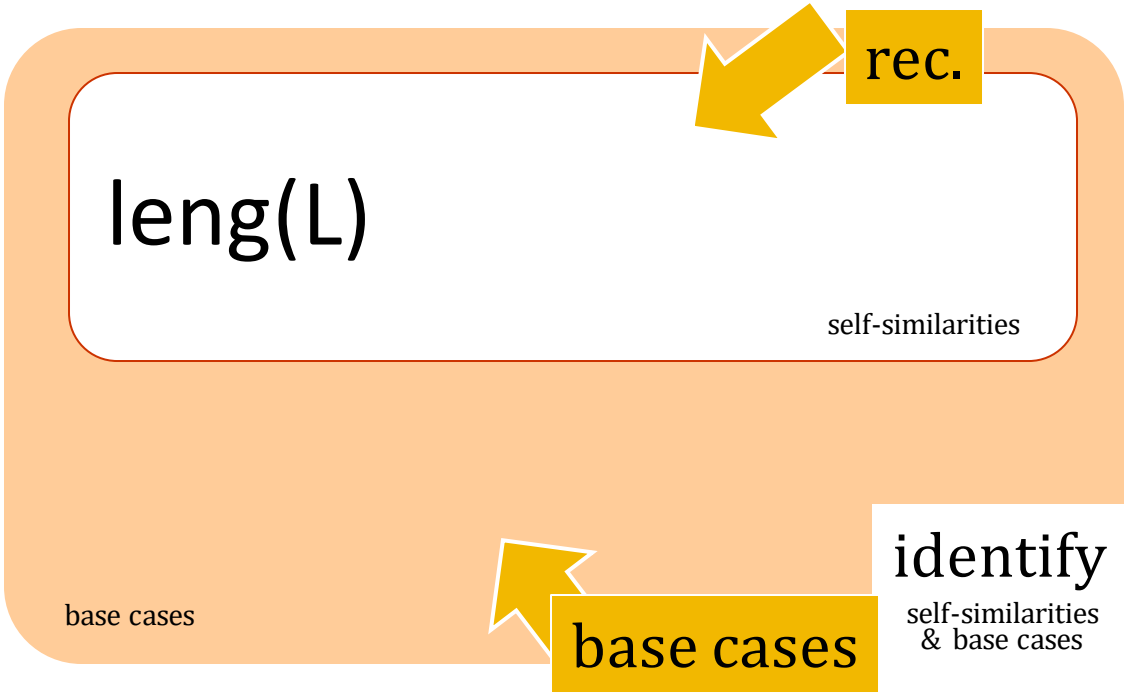
leng([7]) is 1

leng([ ]) is 0

base case(s)

leng('and me?')

intuit




```
def leng(L):
    if _____________:
        return ___
    else:
        return _____________
```

base case

rec.

implement

Recursive Design: leng(L)

leng([5, 42, 7]) is 3

recursion?

leng([42, 7]) is 2

leng([7]) is 1

leng([ ]) is 0

base case(s)

leng('and me?')

intuit

rec.

leng(L) is 1 + leng(L[1:])

self-similarities

leng([ ]) is 0

leng('') is 0

base cases

base cases

identify

self-similarities & base cases

base case

```
def leng(L):
    if L == [] or L == '':
        return 0
    else:
        return 1 + leng(L[1:])
```

rec.

implement

Design patterns…

Recursion's a design—not a formula,
***BUT**, these pieces are common:*

- *Handle base cases, with* **`if`** …
- *Do one piece of work:* **`L[0]`** or **`s[0]`**
- *Recur with the rest:* **`L[1:]`** or **`s[1:]`**
- *Combine & make sure the types match!*