CS61A

section 2

http://links.cs61a.org/jasonxu

upcoming

lab 2
hw 2
hog

feedback

http://links.cs61a.org/jasonxu-feedback

CS61A

last week…

inputs of functions

internals of functions

outputs of functions

CS61A

review

1. 100 % 10 =

2. 241241//10 =

CS61A

booleans

bools have rules

not > and > or

TRUE	FALSE
1	0
‘non-empty’ values	‘empty’ values
	None

CS61A

boolean operators

`and`	`<a> and <b> …`
`or`	`<a> or <b> …`
`not`	`not <a>`

a	b
1	0
1	1
0	0

CS61A

boolean operators

and	<a> and <b> …
or	<a> or <b> …
not	not <a>

a	b	
1	0	
1	1	**which 1?**
0	0	

CS61A

boolean operators another approach

if it’s sunny and not hot

i will go for a run

only will do so when ‘<True> and <True>’

if it’s sunny or not hot

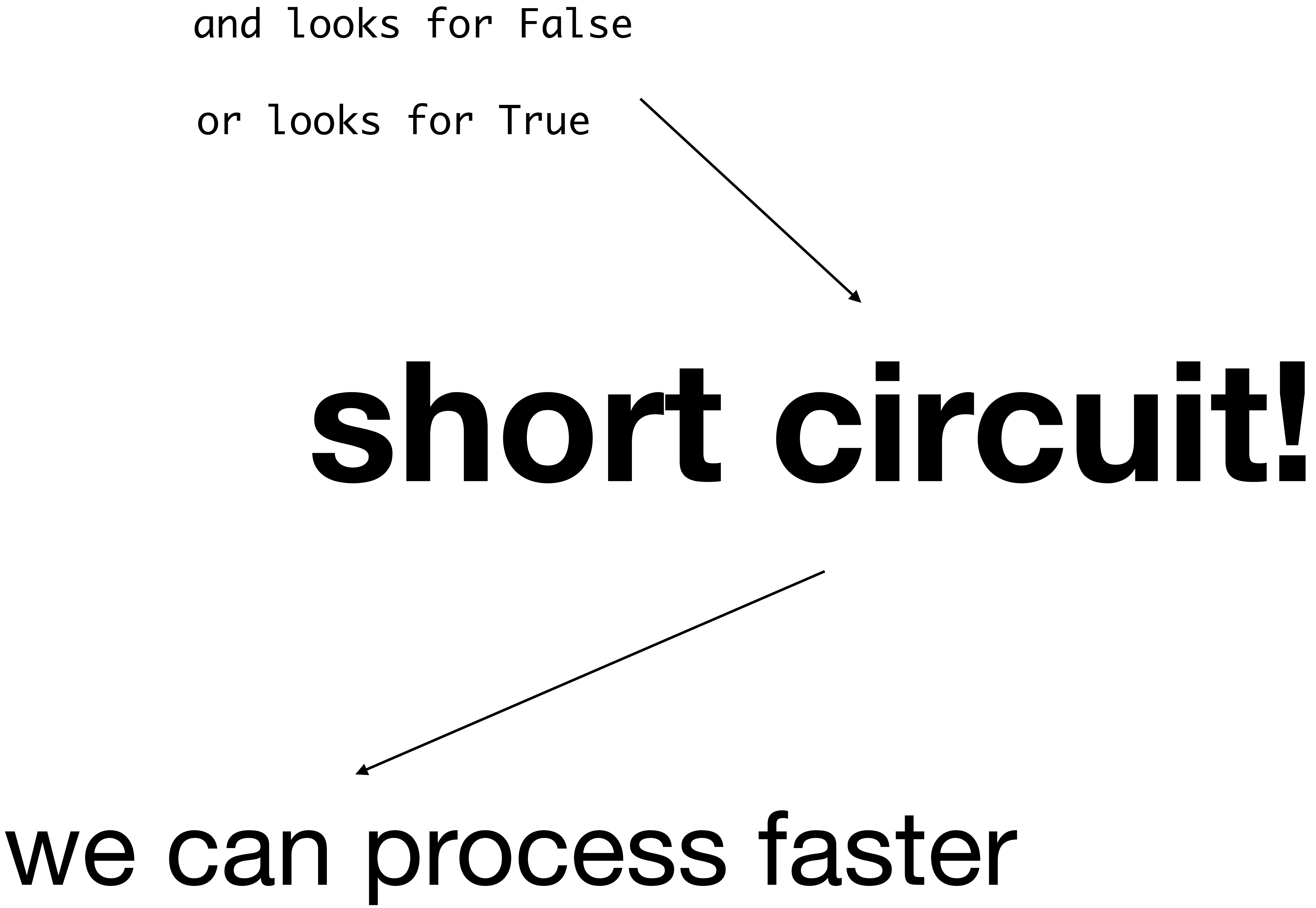
i will go for a run

will do so when either condition is true

CS61A

booleans

bools have rules



`and looks for False`

`or looks for True`

short circuit!

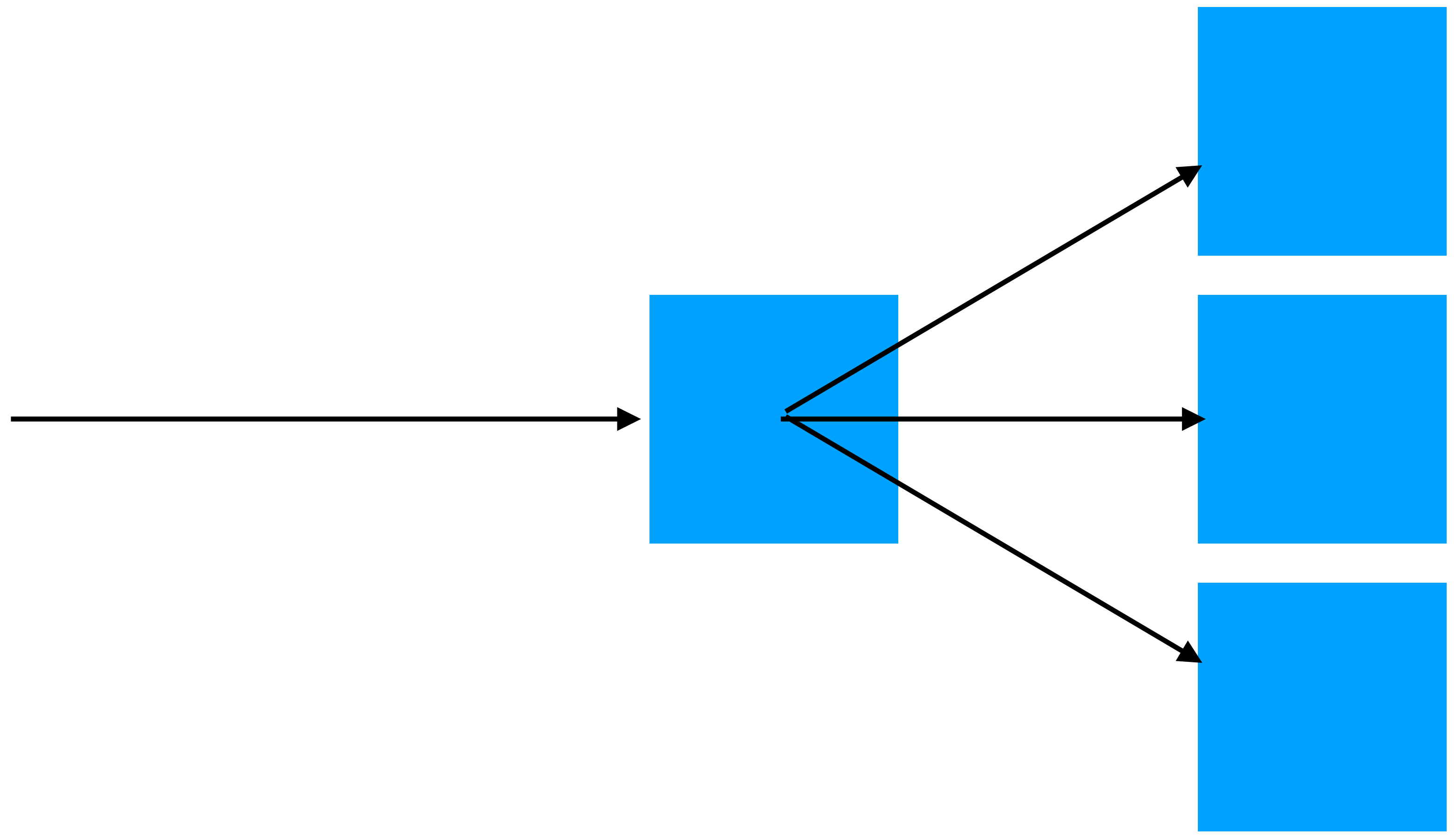
we can process faster

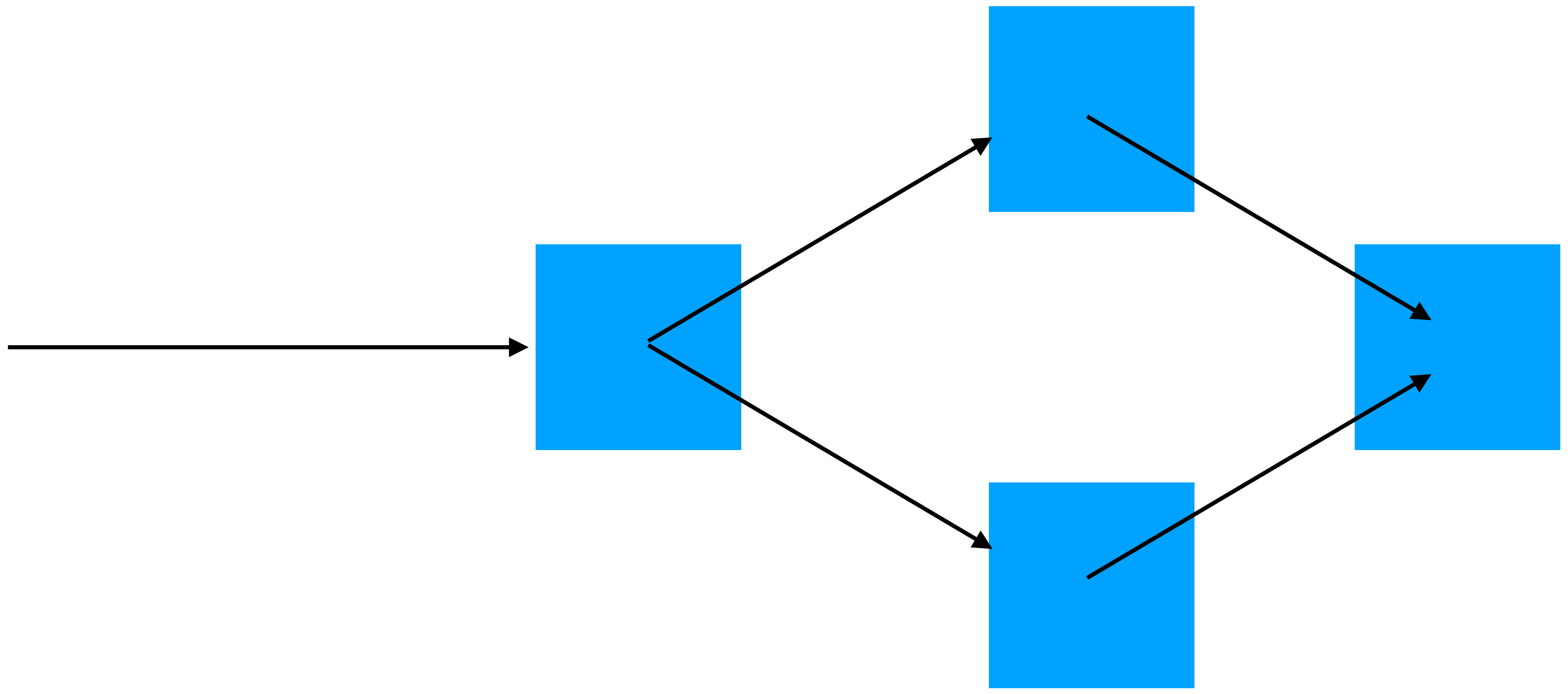
CS61A

controls

```
if <predicate>:
  <do this>
elif <predicate>:
  <do this>
else:
  <do this>
```

```
if <predicate>:
  <do this>
elif <predicate>:
  <do this>
<do this>
```

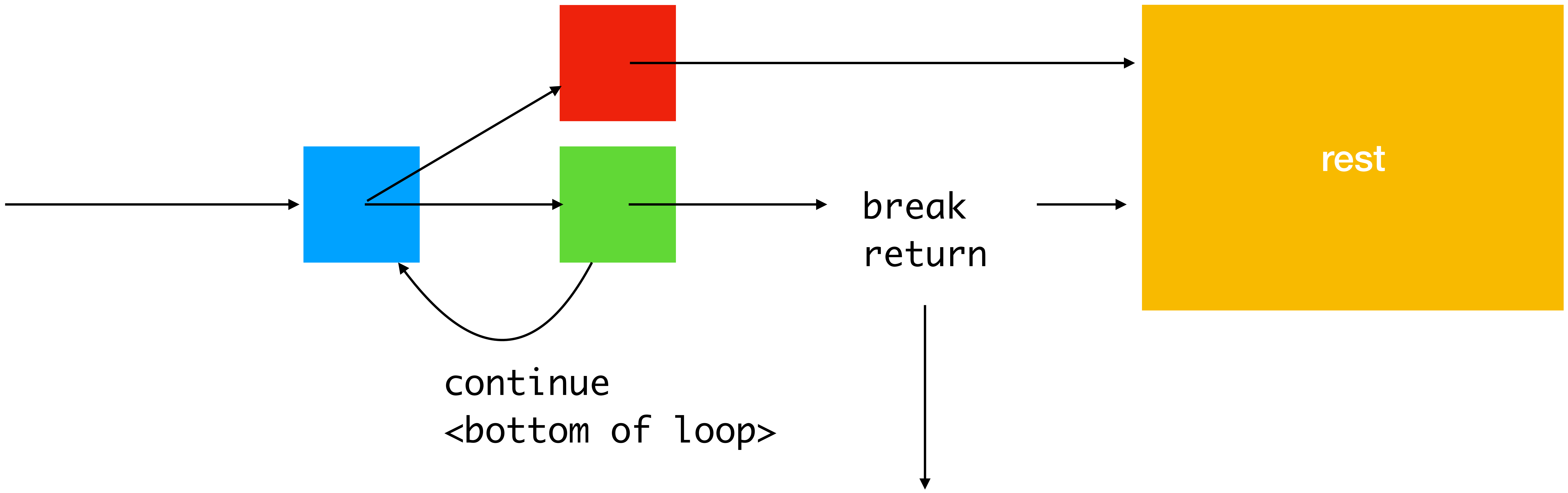




CS61A

controls

```
while <this>:
  <do this>
<rest>
```



CS61A

function calls

$$f(3 \times 2) =$$

CS61A

function calls

$f(3 \times 2) =$ **?**

CS61A

function calls

$f(x) = 2x$ general formula, i can put in any x that is a number

CS61A

function calls

$f(x) = 2x$ general formula, i can put in any x that is a number

$x = 3 \times 2$

CS61A

function calls

$f(x) = 2x$ general formula, i can put in any x that is a number

$x = 3 \times 2$

$f(x) = 12$ i know f, x, can solve!

CS61A

function calls

$f(x) = 2x$

$double(z) = 2z$

$x = 3 \times 2$

$x = 3 \times 2$

$f(x) = 12$

$double(x) = 12$

CS61A

function calls

$f(x) = 2x$

$x = 3 \times 2$

$f(x) = 12$

$double(z) = 2z$

$x = 3 \times 2$

$double(x) = 12$

$double(z) = 2z$

$double(3 \times 2)$

$= 12$

CS61A

functions

There are 2 things to consider for a function

1. Input/Output of function

2. Body of function

CS61A

functions

There are 2 things to consider for a function

1. Input/Output of function

2. Body of function

CS61A

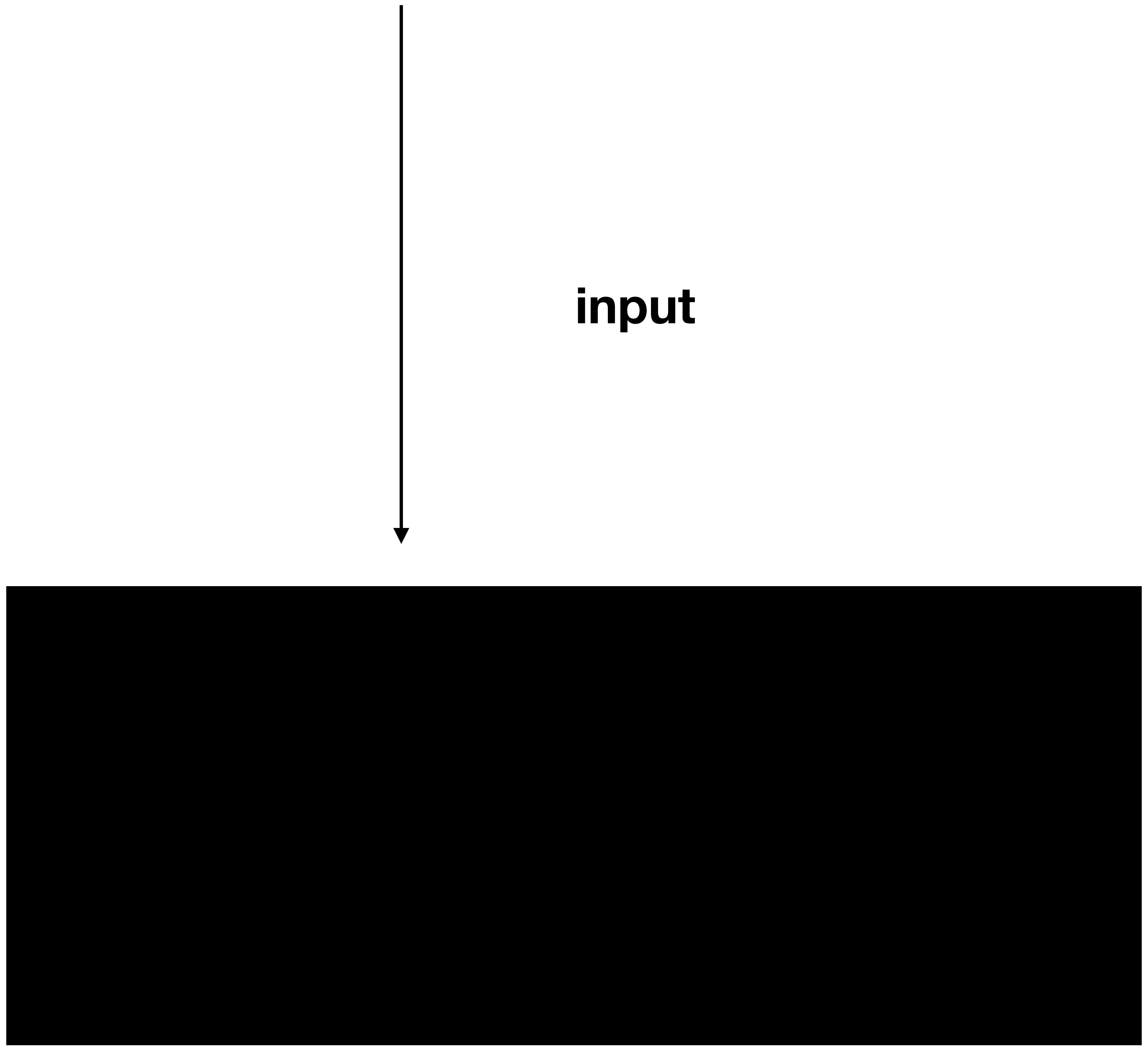
black boxes

we’re going to use these to see an abstract picture of functions

CS61A

black boxes

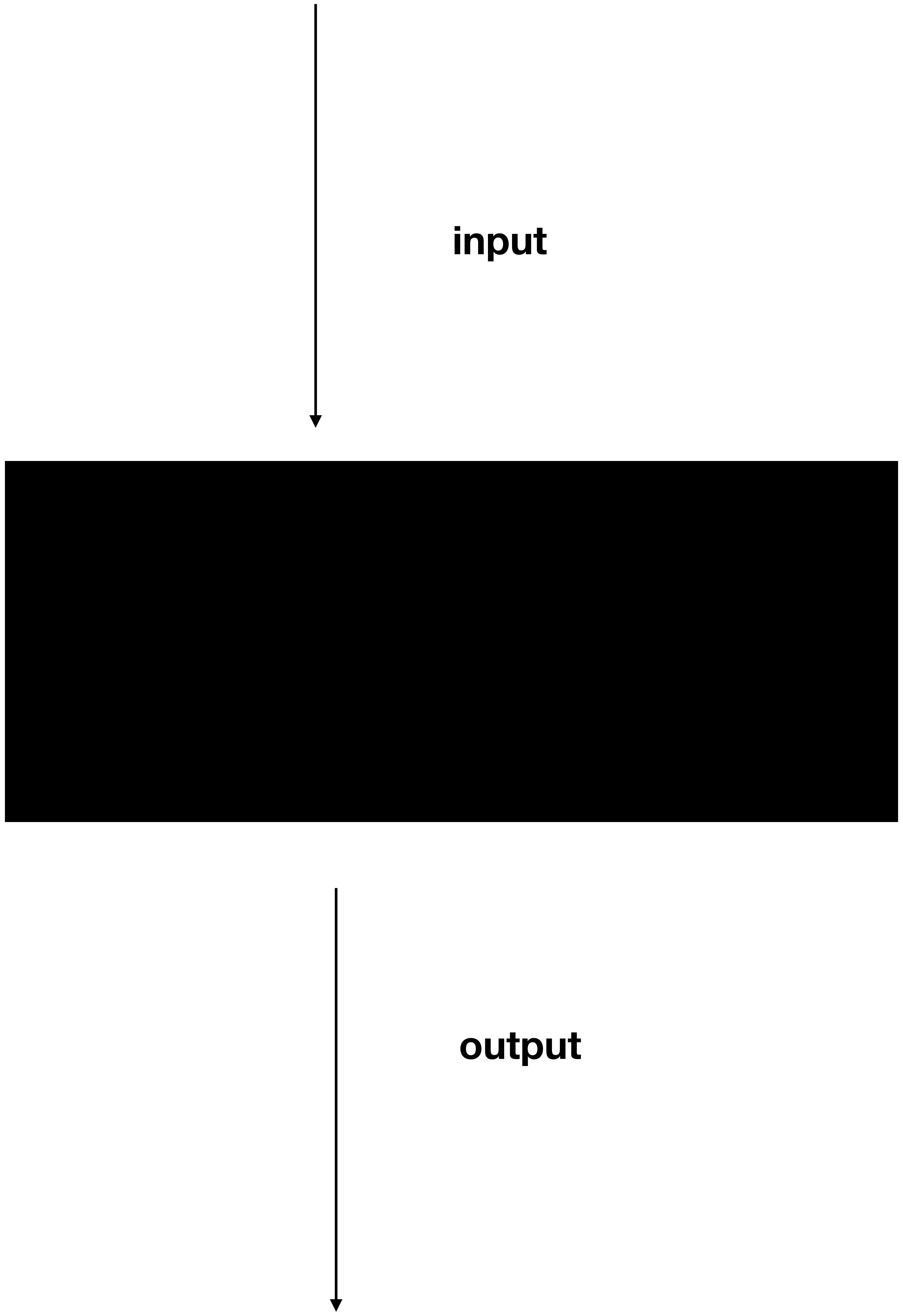
we’re going to use these to see an abstract picture of functions



CS61A

black boxes

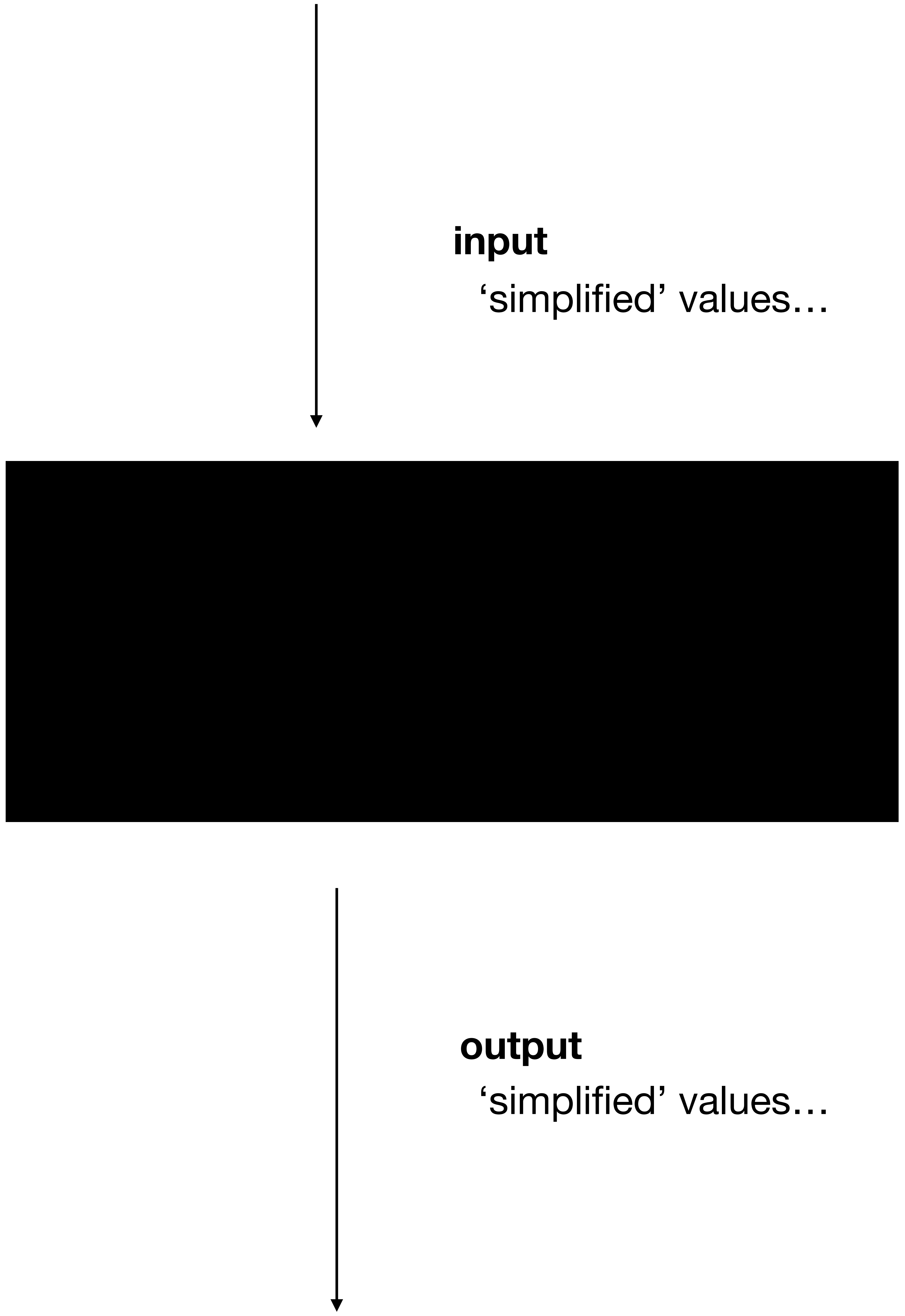
we’re going to use these to see an abstract picture of functions



CS61A

black boxes

we're going to use these to see an abstract picture of functions

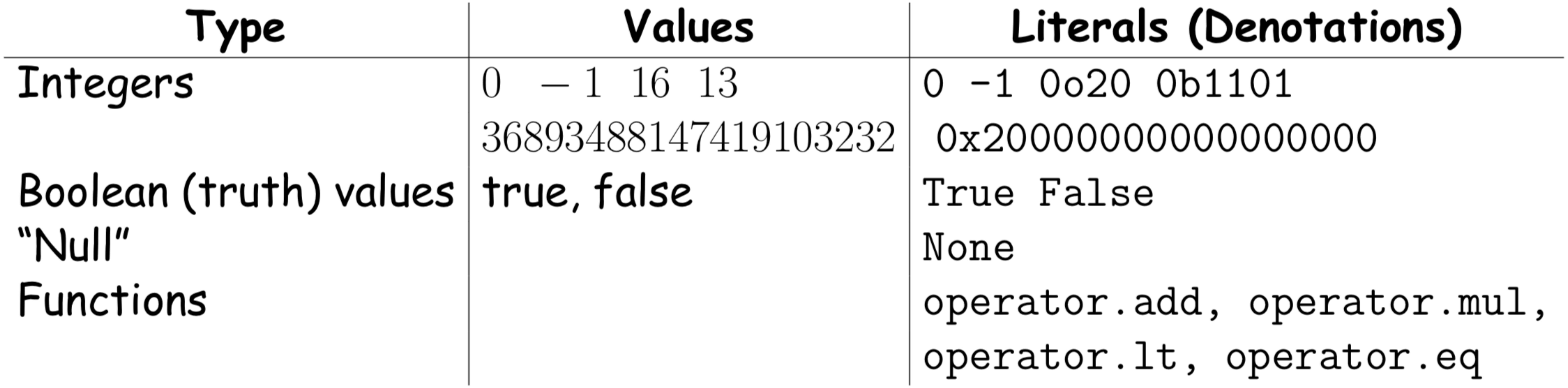


input
'simplified' values…

output
'simplified' values…

CS61A

values



Type	Values	Literals (Denotations)

Integers	$0 \quad -1 \;\; 16 \;\; 13$
	36893488147419103232
		`0 -1 0o20 0b1101`
		`0x20000000000000000`

Boolean (truth) values	true, false	`True False`
"Null"		`None`

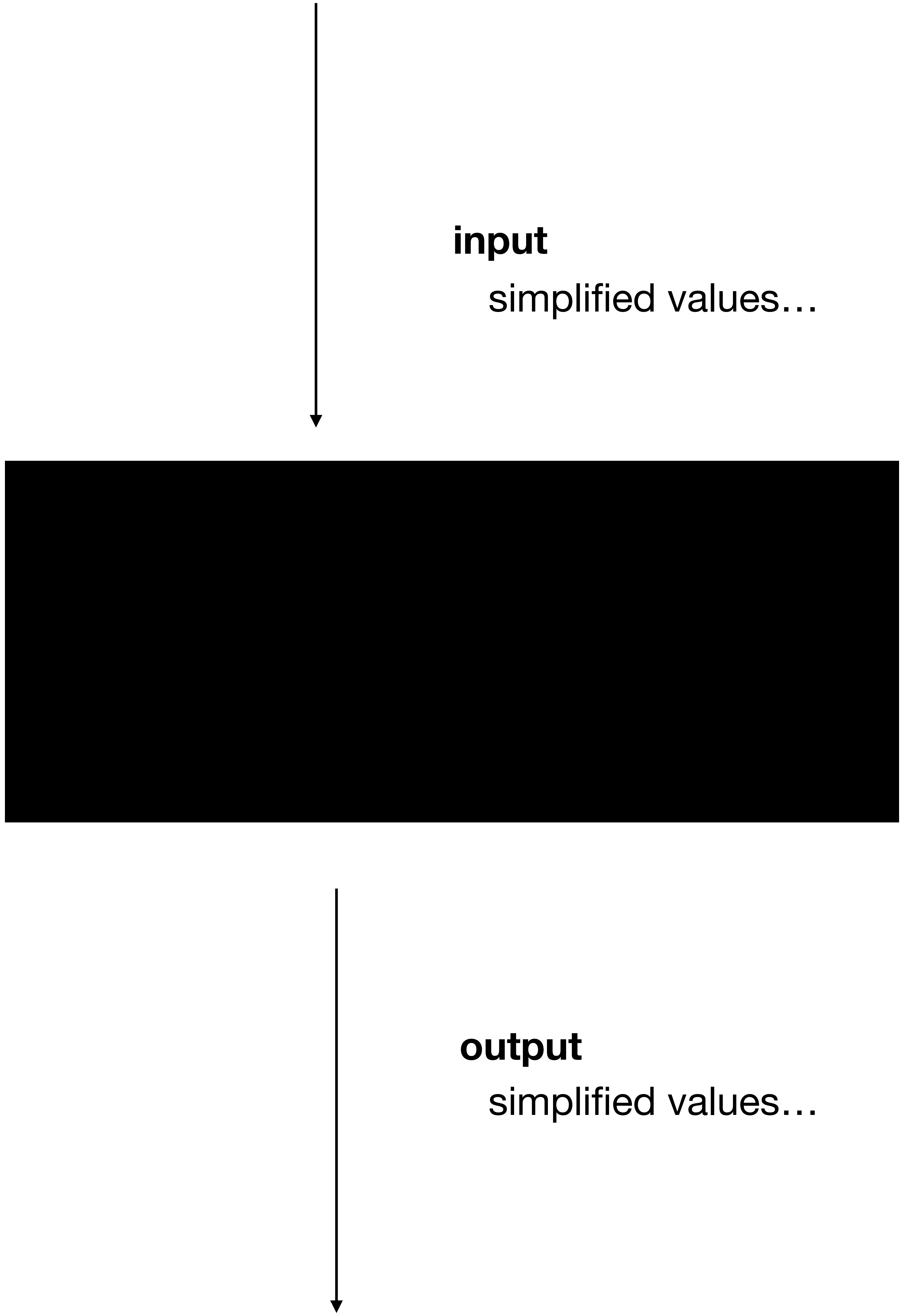
Functions		`operator.add, operator.mul,`
		`operator.lt, operator.eq`

Strings	Say "Hello"	`"Say \"Hello\""`

CS61A

black boxes

we're going to use these to see an abstract picture of functions



input

simplified values…

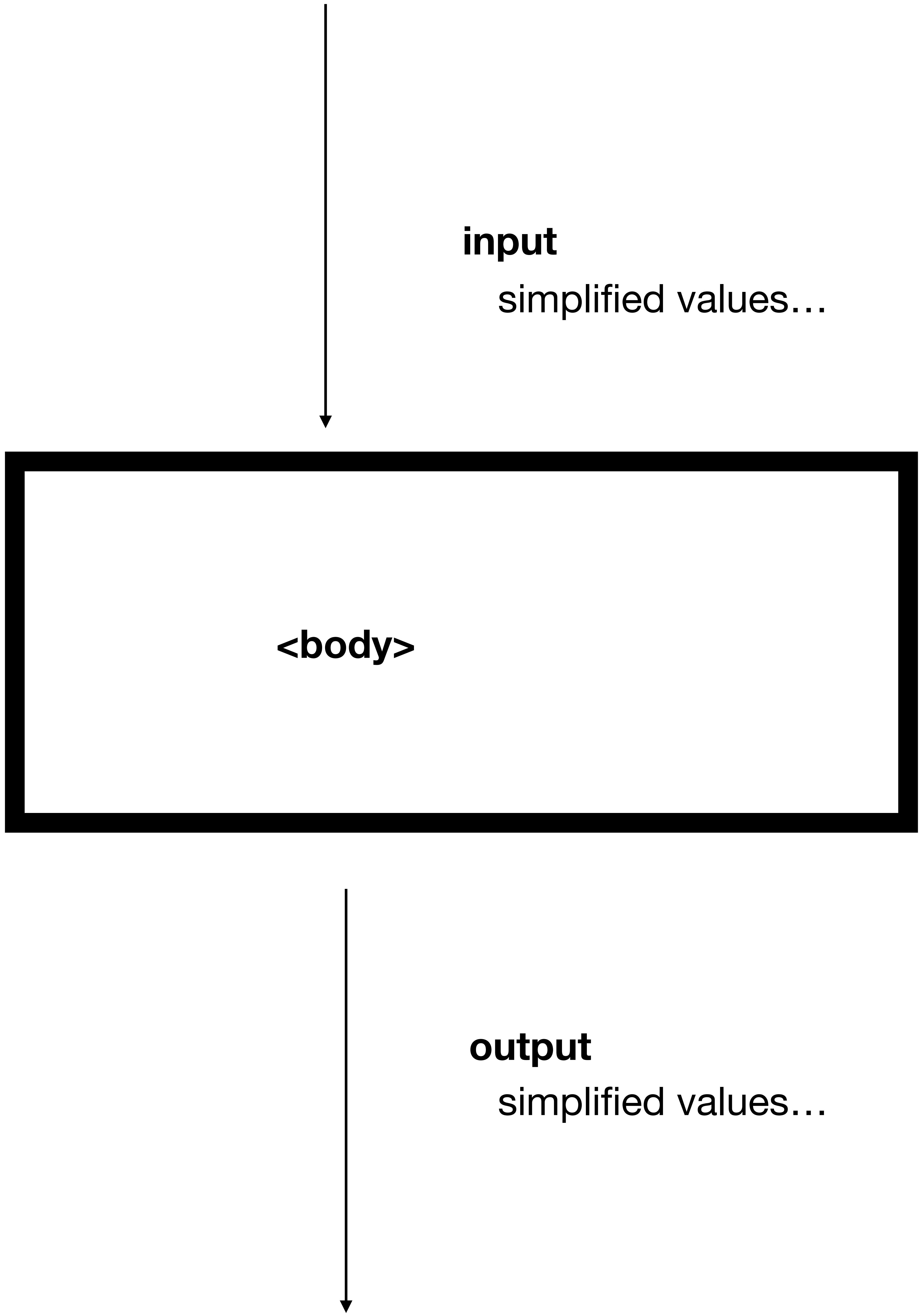
output

simplified values…

CS61A

black boxes

we're going to use these to see an abstract picture of functions



CS61A

returning

return stops procedure and outputs something

print is an action, function

CS61A

returning

return stops procedure and outputs something

print is an action, function

```
def showFivePrint():
   x = 2 + 3
   print(x)
```

```
def showFiveReturn():
   x = 2 + 3
   return x
```

CS61A

returning

return stops procedure and outputs something

print is an action, function

```
def showFivePrint():
    x = 2 + 3
    print(x)
```

```
def showFiveReturn():
    x = 2 + 3
    return x
```

computing 2 + 3 and storing it to x

CS61A

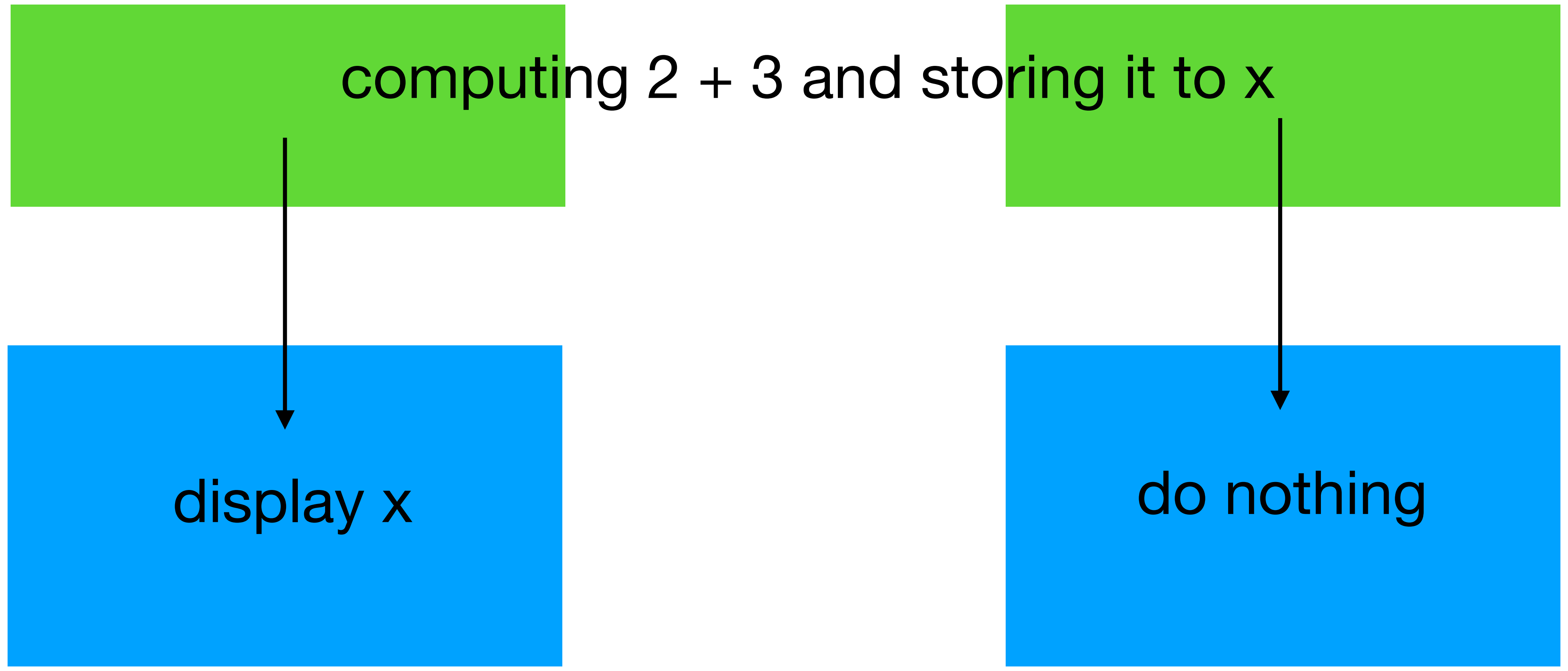
returning

return stops procedure and outputs something

print is an action, function

```
def showFivePrint():
    x = 2 + 3
    print(x)
```

```
def showFiveReturn():
    x = 2 + 3
    return x
```



CS61A

returning

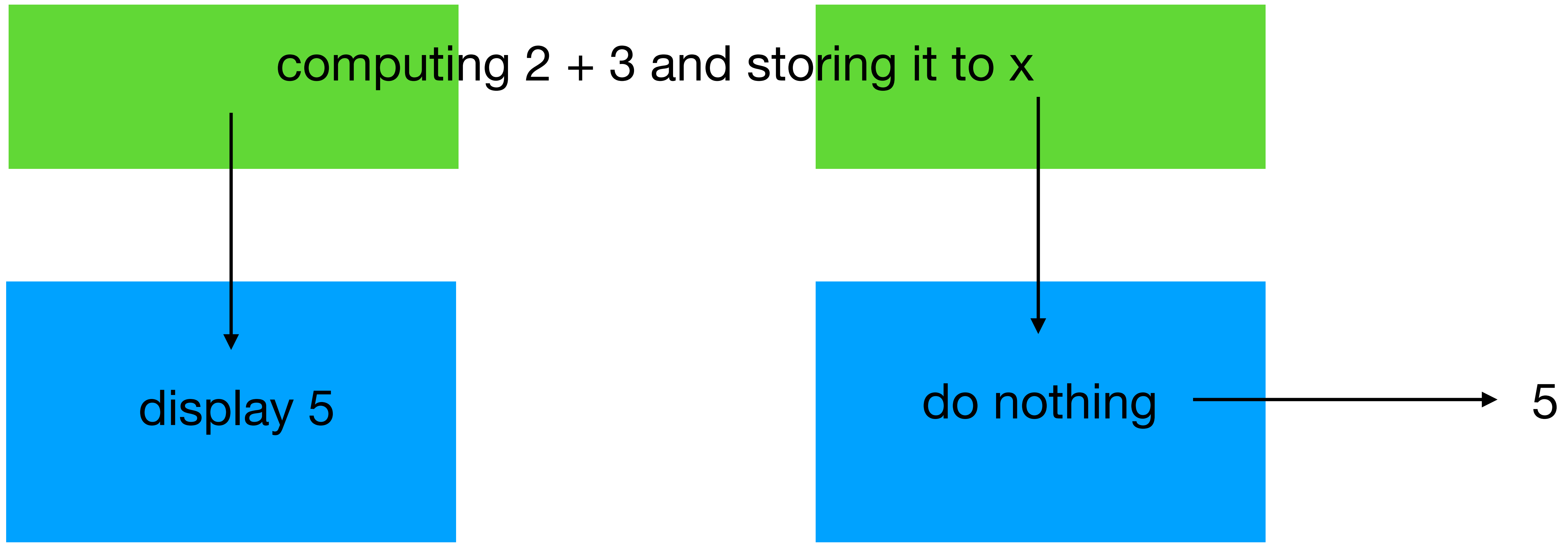
return stops procedure and outputs something

print is an action, function

every function has a return at end of None

```
def showFivePrint():
    x = 2 + 3
    print(x)
```

```
def showFiveReturn():
    x = 2 + 3
    return x
```



CS61A

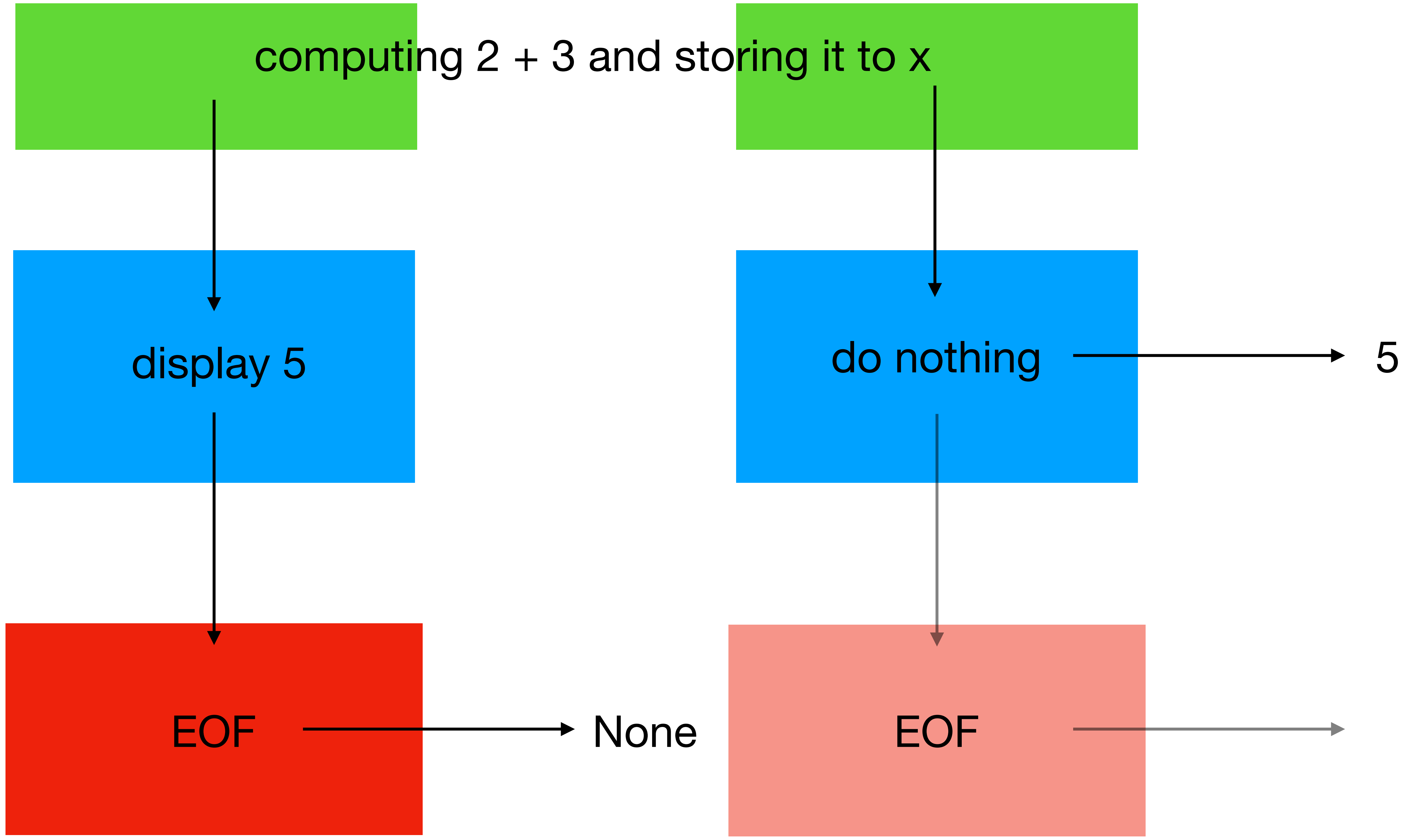
returning

return stops procedure and outputs something

print is an action, function

```
def showFivePrint():
    x = 2 + 3
    print(x)
    return
```

```
def showFiveReturn():
    x = 2 + 3
    return x
```



CS61A

returning

return stops procedure and outputs something

print is an action, function

return only 1 thing but can also return tuples (pair structures)

```
def showFivePrint():
    x = 2 + 3
    print(x)
```

```
> val = showFivePrint()
5
> val
> val is None
True
```

```
def showFiveReturn():
    x = 2 + 3
    return x
```

```
> val = showFiveReturn()
> val
5
```

CS61A

function calls

```
> max(10 + 5, 9, double(18))
36
```

CS61A

function calls

```
> max(10 + 5, 9, double(18))
36
```

```
max
10 + 5 = 15
9 = 9
double(18) = 36
max(15, 9, 36)
= 36
```

CS61A

function calls

```
> max(10 + 5, 9, double(18))
36
```

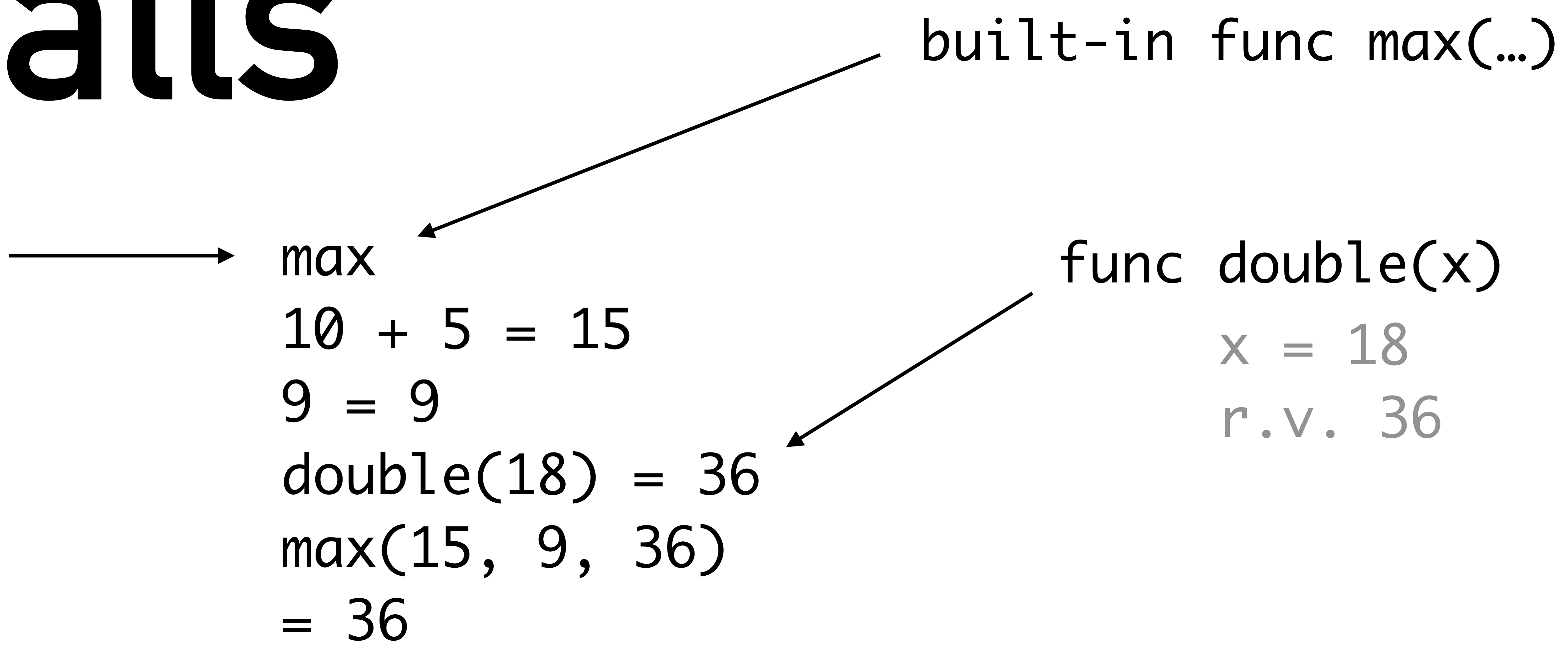
```
max
10 + 5 = 15
9 = 9
double(18)
max(15, 9, ?)
= ?
```

built-in func max(…)

CS61A

function calls

```
> max(10 + 5, 9, double(18))
36
```



built-in func max(…)

```
max
10 + 5 = 15
9 = 9
double(18) = 36
max(15, 9, 36)
= 36
```

func double(x)

```
x = 18
r.v. 36
```

CS61A

function calls

```
> max(10 + 5, 9, double(18))
36
```

```
max
10 + 5 = 15
9 = 9
double(18) = 36
max(15, 9, 36)
= 36
```

if any part of this breaks, you get an error and stops

CS61A

function calls

```
> max(10 + 5, 9, 18 / 0)
error
```

max

10 + 5 = 15

9 = 9

18 / 0 = ?

~~max(15, 9, 18 / 0)~~

error

if any part of this breaks,
you get an error and stops

typing an error != will error

EVALULATE OPERATOR

EVALUATE OPERANDS

APPLY OPERATOR

CS61A

last week...

inputs of functions

internals of functions

outputs of functions

Higher Order Functions

CS61A

env. diagrams 101
on board

CS61A

env. diagrams

when do i open a frame?

what is a function’s parent frame?

do we copy *intrinsically same* functions during assignment?

how do we look up variables?

CS61A

lambdas

```
lambda <arguments>: <return value>

lambda x, y: x + y

lambda: lambda x: x

(lambda x: lambda x: x)(2)(3)
```

which x?

CS61A

recursion

things *defined* by themselves

~~CS61A~~

~~recursion~~

~~things *defined* by themselves~~

NOT ON MT1

yay!

CS61A

recursion

factorial!

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

5! = 5 * 4!

4! = 4 * 3!

3! = 3 * 2!

2! = 2 * 1!

1! = 1 * 0!

CS61A

recursion

factorial!

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

5! = 5 * 4!

4! = 4 * 3!

3! = 3 * 2!

2! = 2 * 1!

1! = 1 * 0!

uhhhhhhhhh

when do i stop?

0! = 0 * -1!

-1! = -1 * -2! ...

CS61A

recursion

factorial!

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

5! = 5 * 4!

4! = 4 * 3!

3! = 3 * 2!

2! = 2 * 1!

1! = 1 * 0!

uhhhhhhhhh

base case!

0! = 0 * -1!

-1! = -1 * -2! ...

CS61A

recursion

factorial!

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

5! = 5 * 4!

4! = 4 * 3!

3! = 3 * 2!

2! = 2 * 1!

1! = 1 * 0!

yay! 😍

base case!

0! = 1

CS61A

recursion

factorial!

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

how do we come up with this

by definition, 😱

by assuming it works, 🤪

CS61A

recursion

factorial!

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

how do we calculate 5!

5! = 5 * 4!

for this to be true, don’t we have to assume that ‘!’ really does what it says

well in code we can’t name a function ‘!’

we assume that (n-1)! works

recursive leap of faith

well… i have to test it by tracing it

well… big headache

CS61A

recursion

strategy

if you capture all the base cases

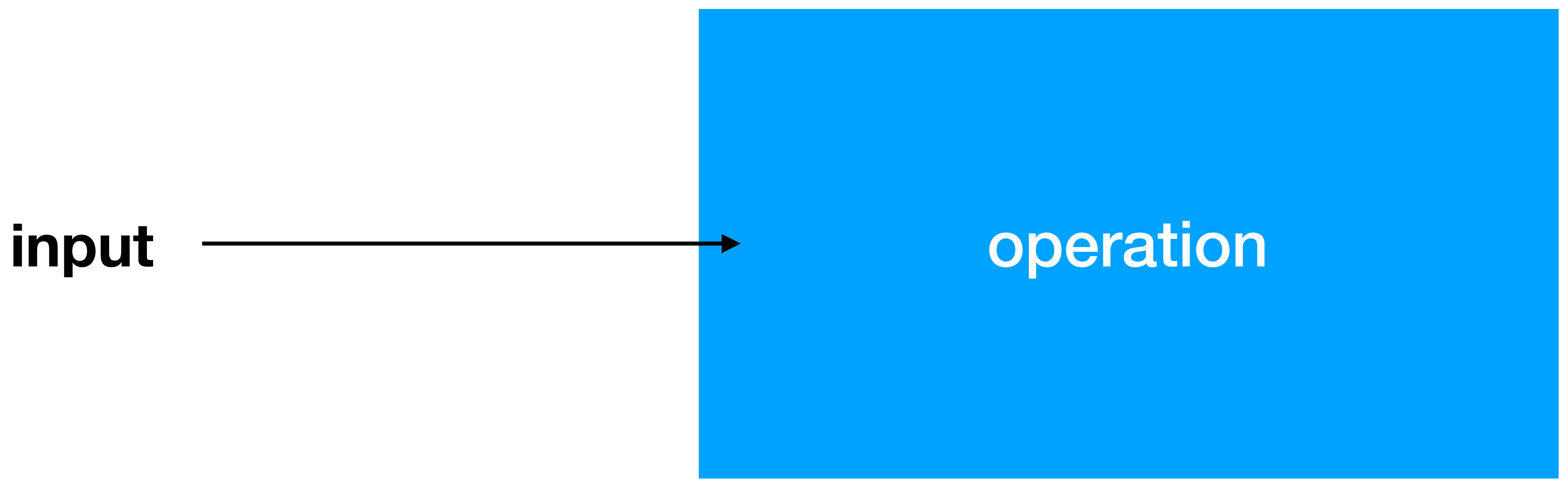
you can assume it works

so you can create the recursive call

CS61A

recursion

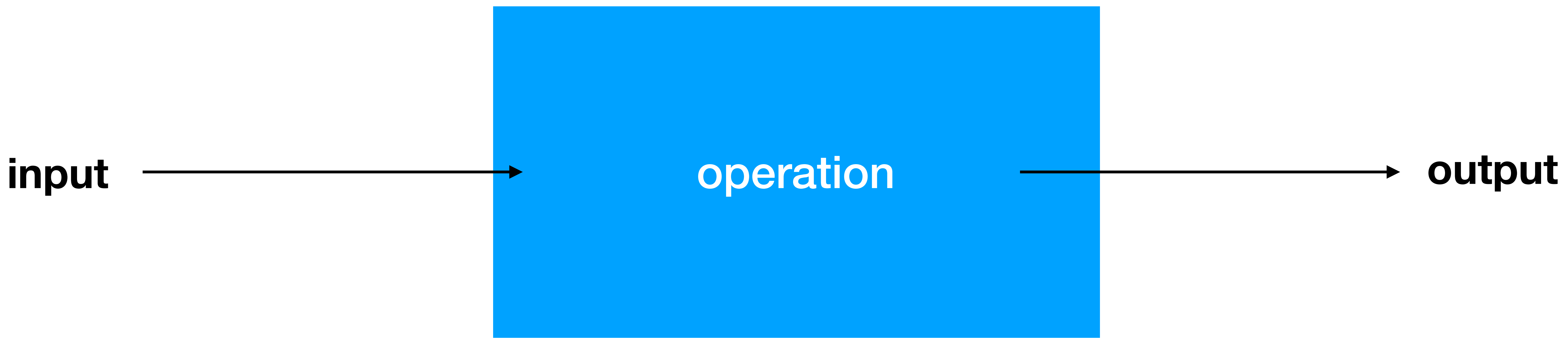
motivation for it



CS61A

recursion

things *defined* by themselves



CS61A

recursion

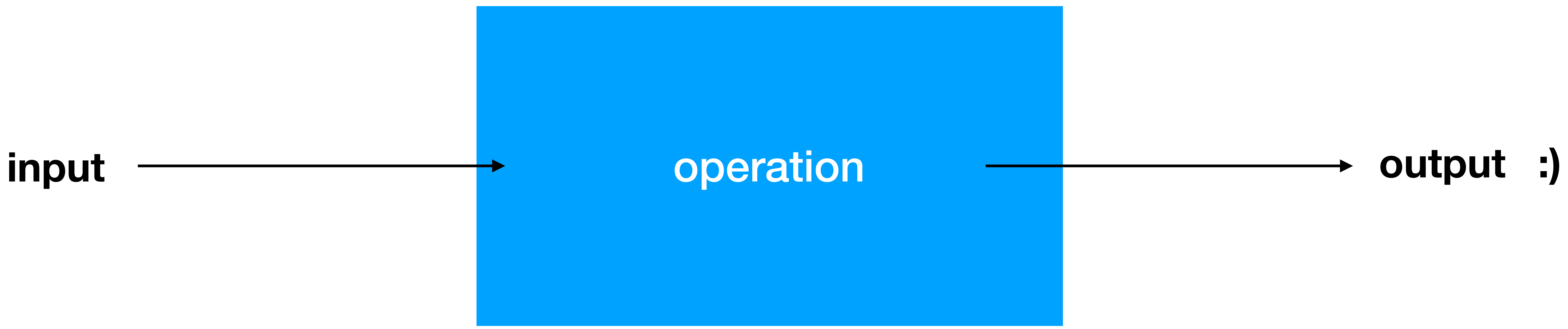
things *defined* by themselves

input
operation
output :)

CS61A

recursion

things *defined* by themselves

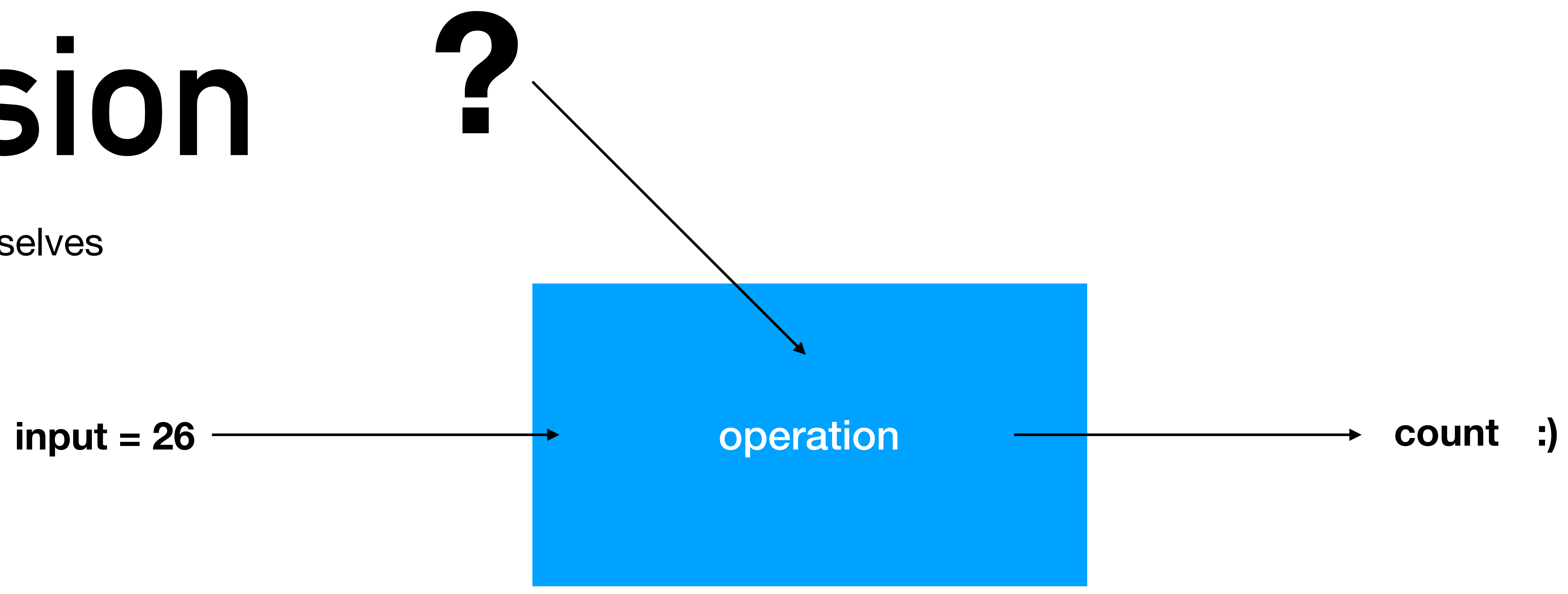


tell me the number of ways to line $26

CS61A

recursion

things *defined* by themselves



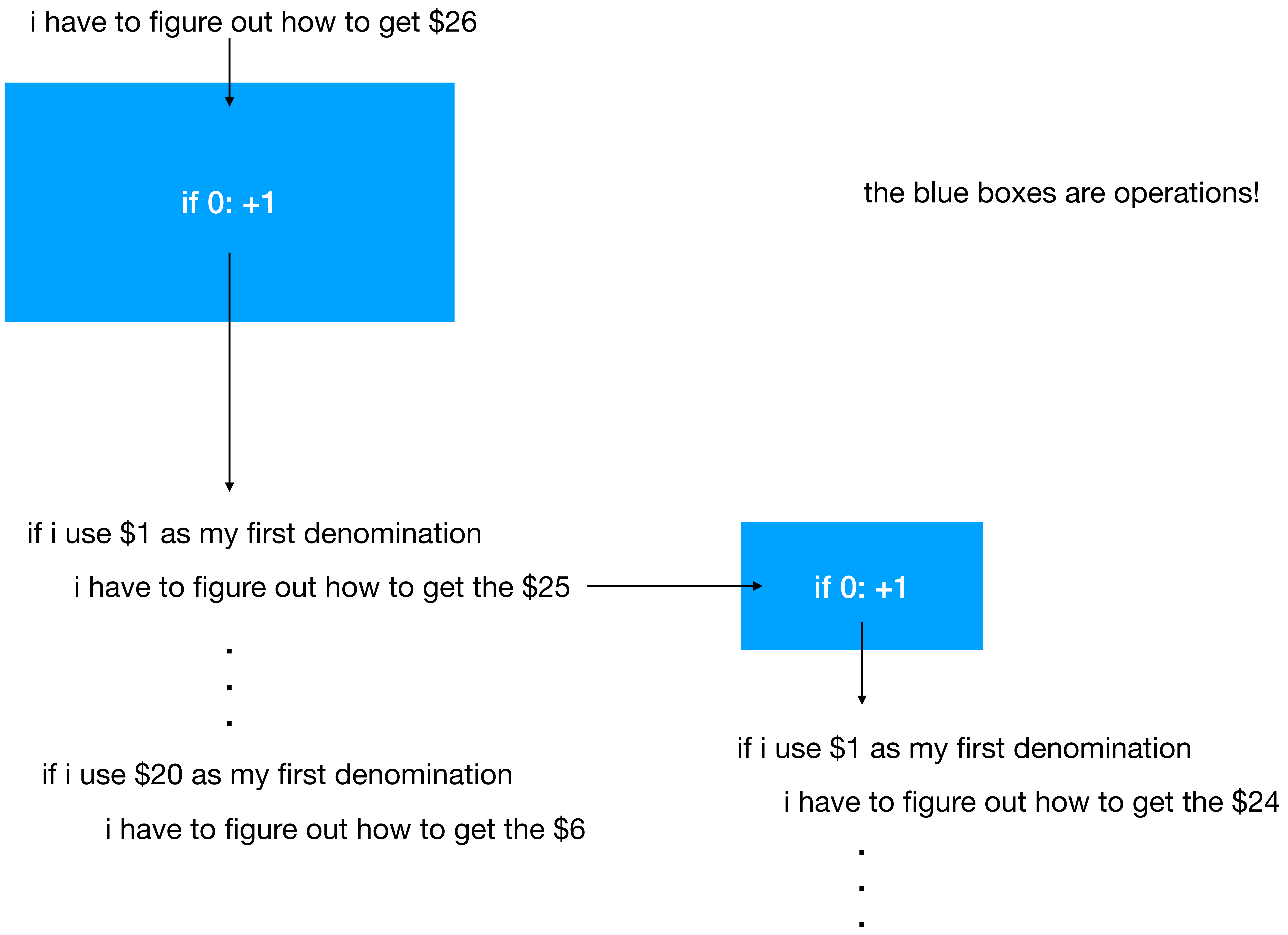
?

input = 26

operation

count :)

tell me the number of ways to line $26



i have to figure out how to get $26

if 0: +1

if i use $1 as my first denomination

i have to figure out how to get the $25

.
.
.

if i use $20 as my first denomination

i have to figure out how to get the $6

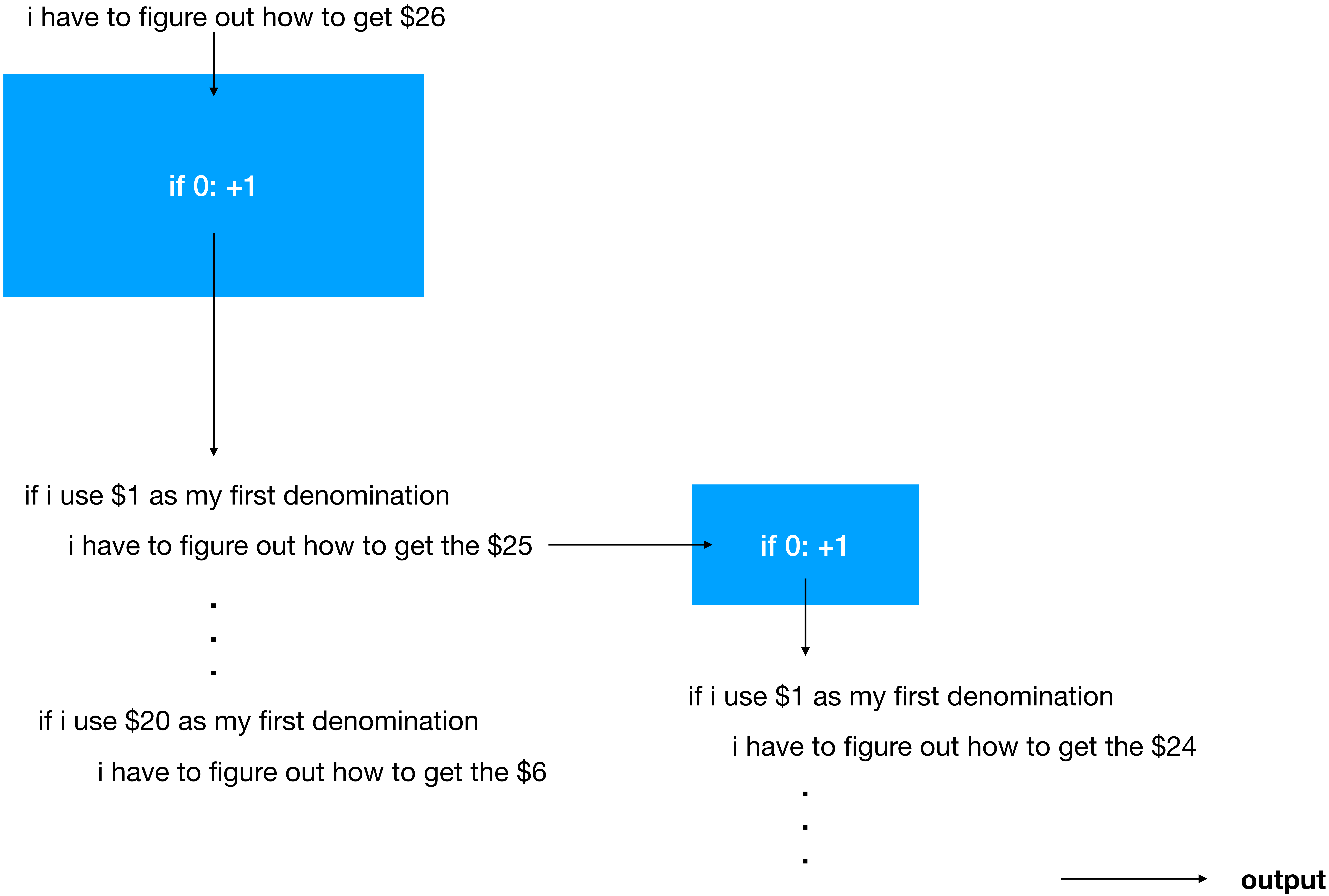
the blue boxes are operations!

if 0: +1

if i use $1 as my first denomination

i have to figure out how to get the $24

.
.
.



i have to figure out how to get $26

if 0: +1

if i use $1 as my first denomination

i have to figure out how to get the $25

if 0: +1

if i use $1 as my first denomination

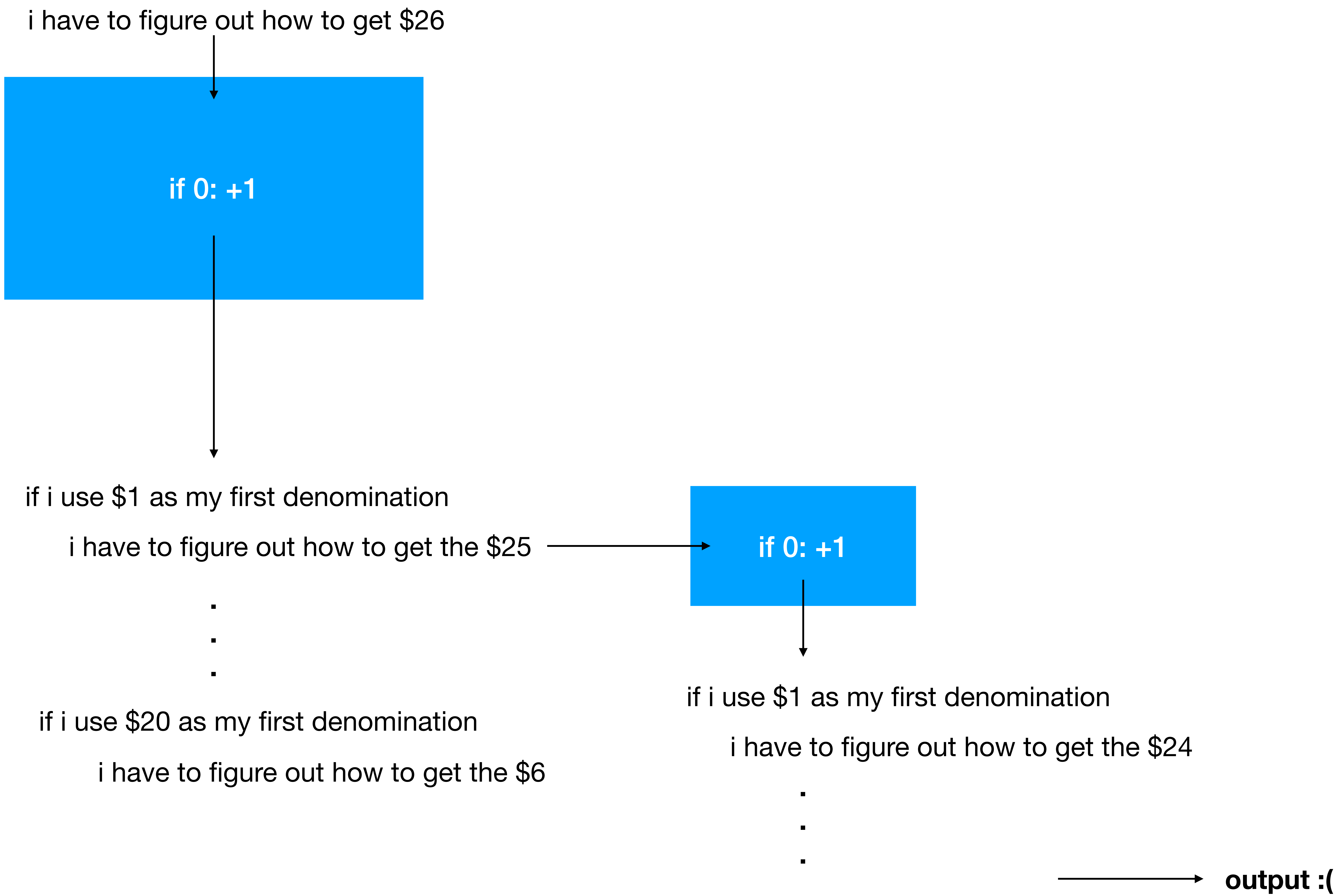
i have to figure out how to get the $24

.
.
.

if i use $20 as my first denomination

i have to figure out how to get the $6

output



i have to figure out how to get $26

if 0: +1

if i use $1 as my first denomination

i have to figure out how to get the $25

.
.
.

if i use $20 as my first denomination

i have to figure out how to get the $6

if 0: +1

if i use $1 as my first denomination

i have to figure out how to get the $24

.
.
.

output :(

```
def count(n):
    total = 0
    options = [n]
    while len(options) > 0:
        curr = options.pop(0)
        for change in [1, 5, 10, 20]:
            val = curr - change
            if val == 0:
                total += 1
            elif val > 0:
                options.append(val)
    return total
```

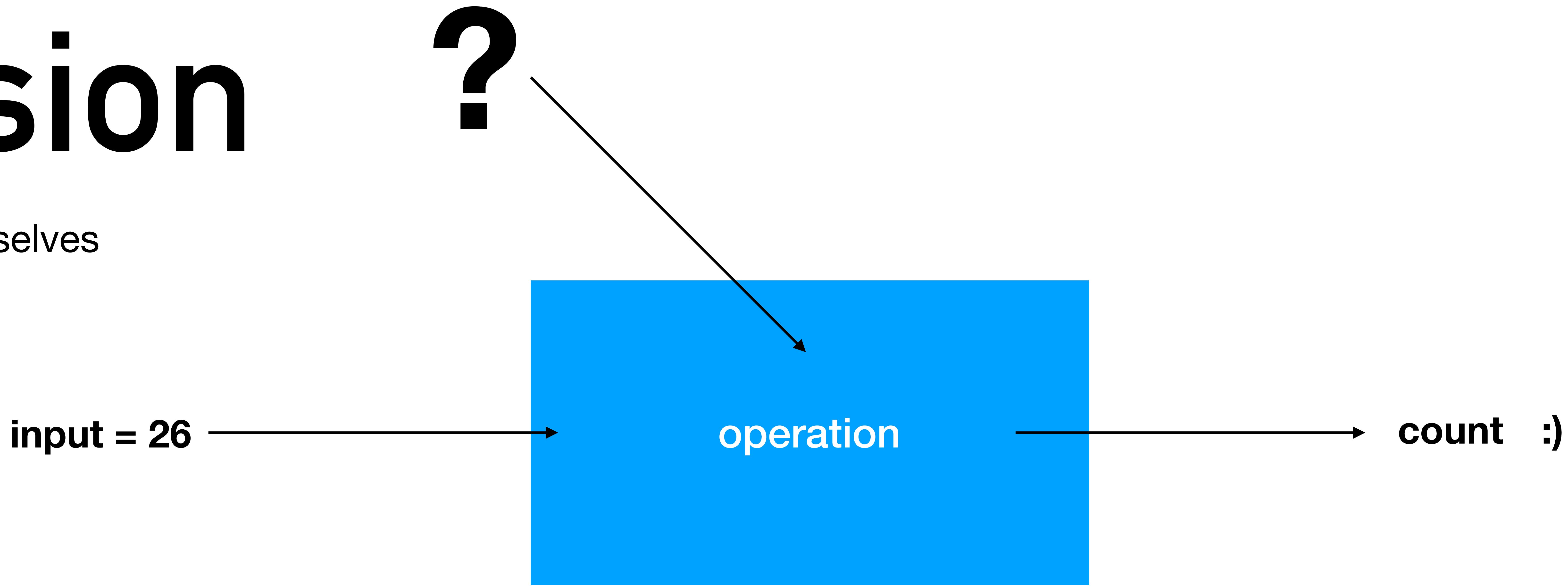
```
def count(n):
    total = 0
    options = [n]
    while len(options) > 0:
        curr = options.pop(0)
        for change in [1, 5, 10, 20]:
            val = curr - change
            if val == 0:
                total += 1
            elif val > 0:
                options.append(val)
    return total
```

🙃

CS61A

recursion

things *defined* by themselves



?

input = 26

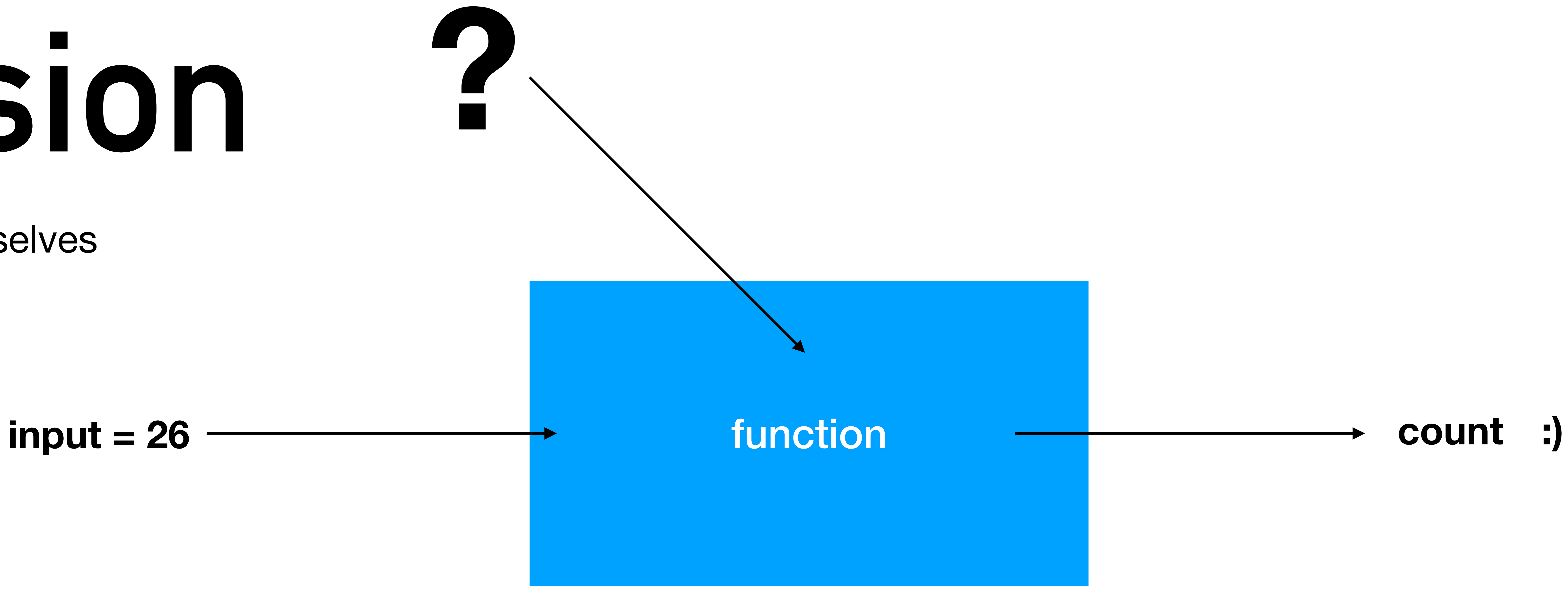
operation

count :)

CS61A

recursion

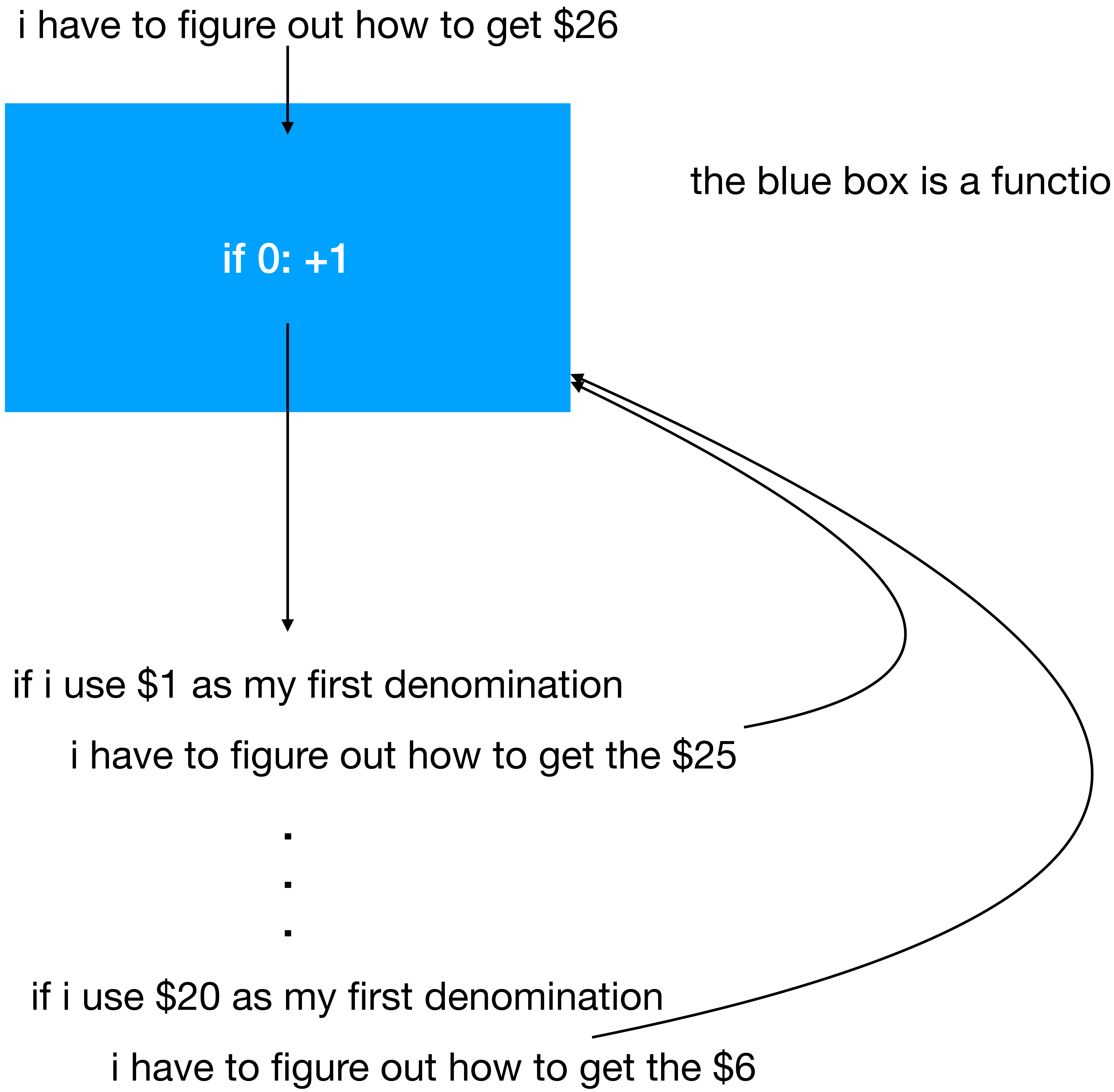
things *defined* by themselves



CS61A

recursion

things *defined* by themselves



i have to figure out how to get $26

if 0: +1

if i use $1 as my first denomination

i have to figure out how to get the $25

.
.
.

if i use $20 as my first denomination

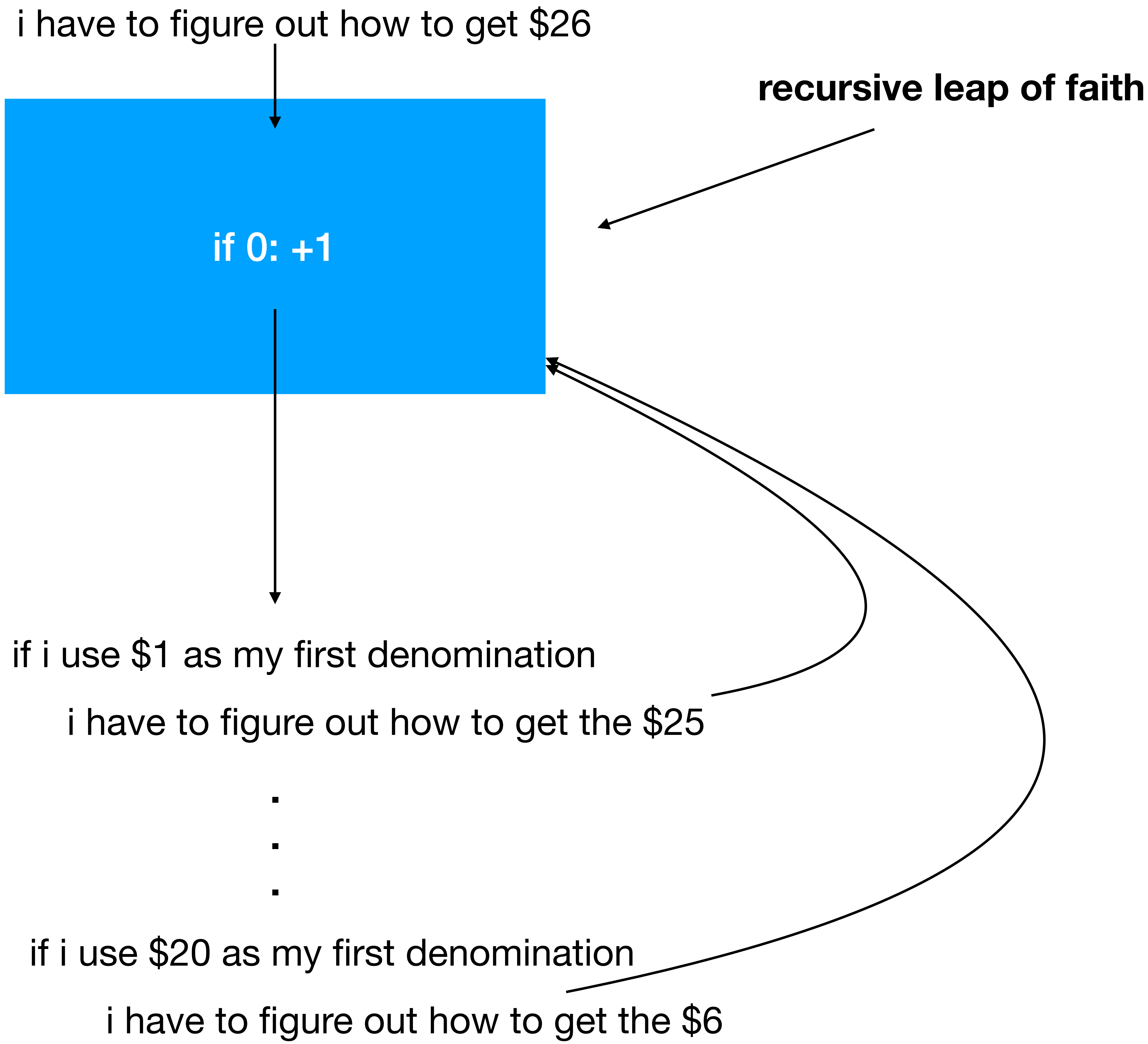
i have to figure out how to get the $6

the blue box is a function!

CS61A

recursion

things *defined* by themselves



i have to figure out how to get $26

if 0: +1

recursive leap of faith

if i use $1 as my first denomination

i have to figure out how to get the $25

.
.
.

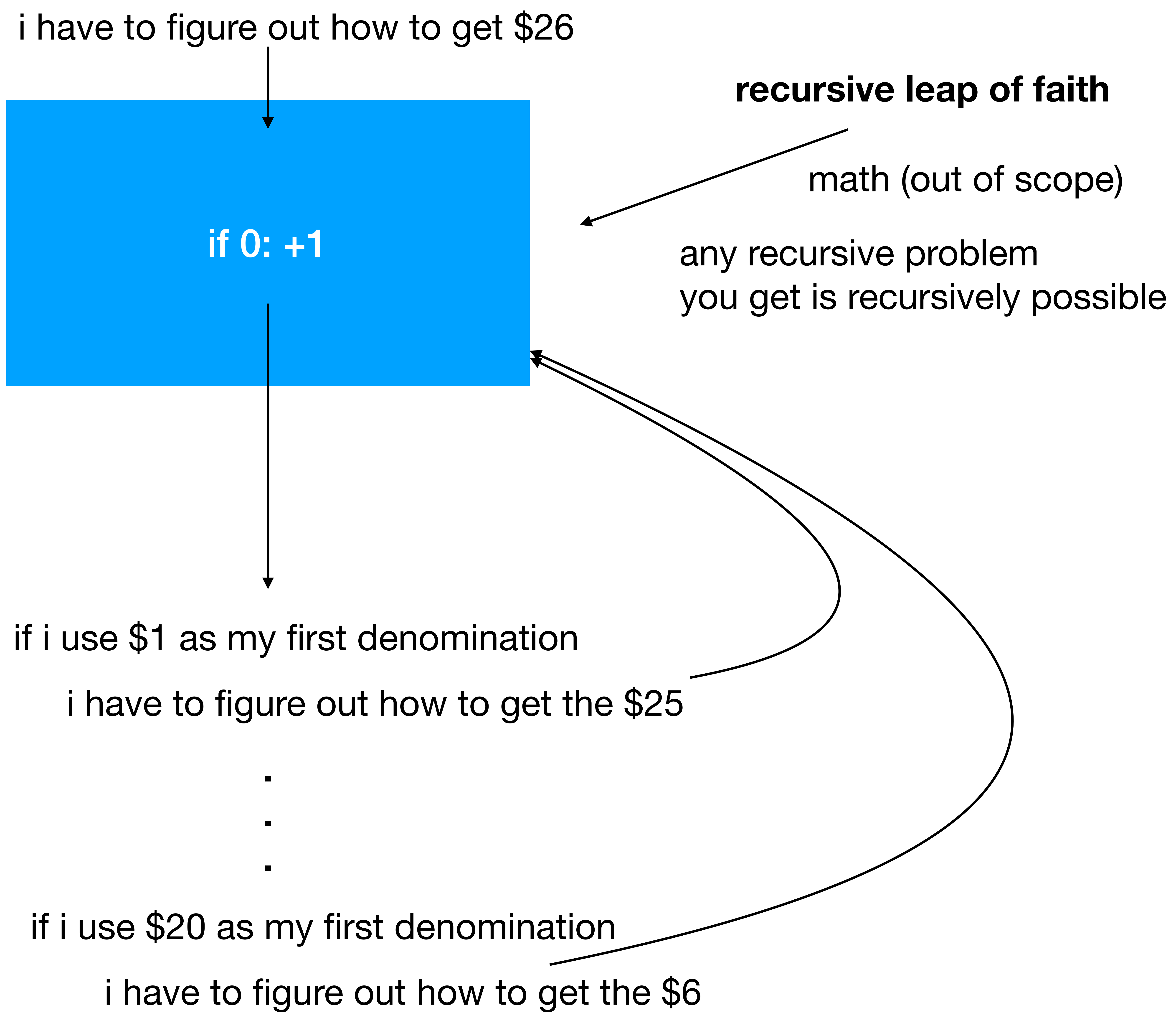
if i use $20 as my first denomination

i have to figure out how to get the $6

CS61A

recursion

things *defined* by themselves



i have to figure out how to get $26

if 0: +1

recursive leap of faith

math (out of scope)

any recursive problem
you get is recursively possible

if i use $1 as my first denomination

i have to figure out how to get the $25

.
.
.

if i use $20 as my first denomination

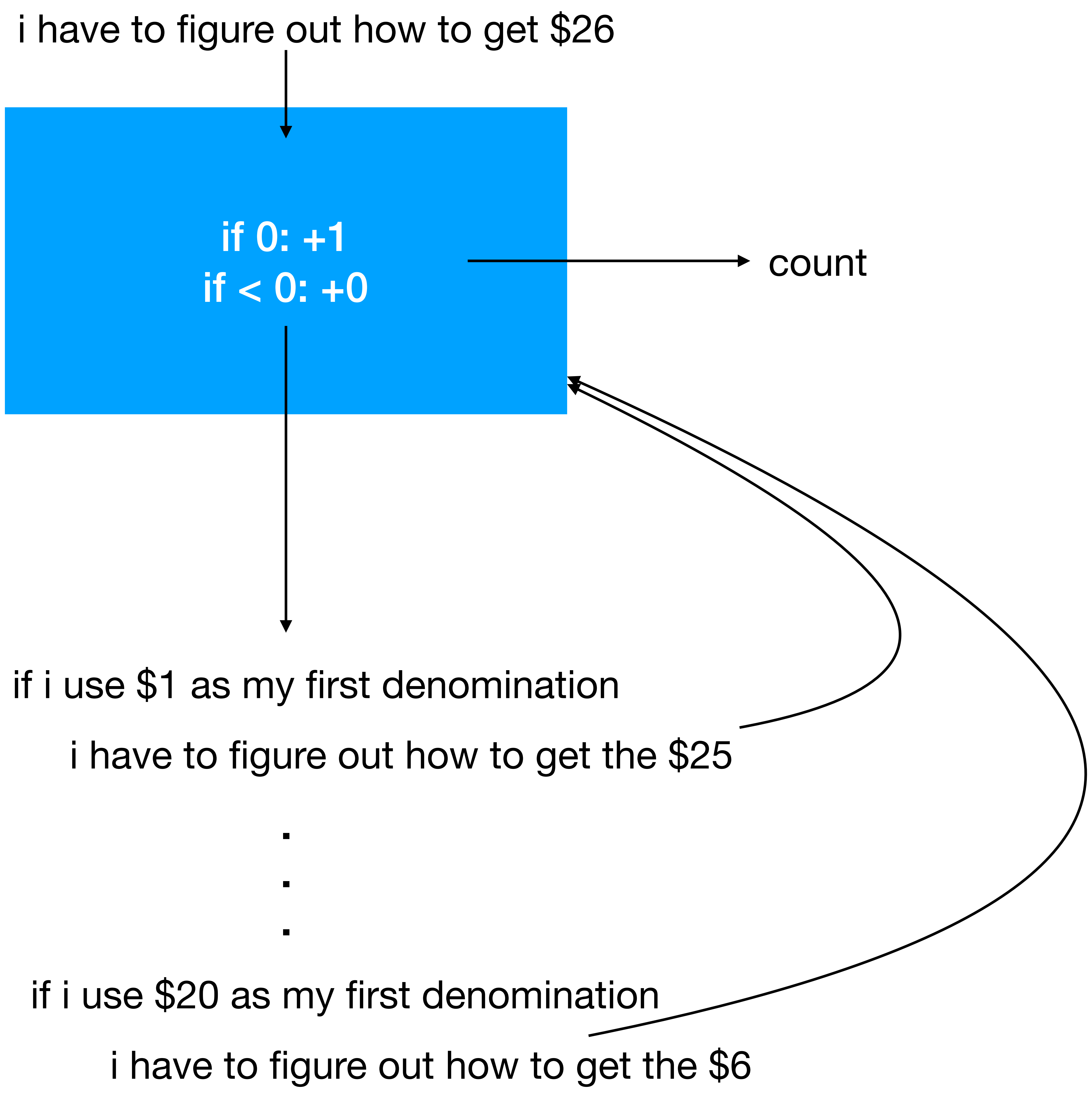
i have to figure out how to get the $6

CS61A

recursion

things *defined* by themselves

```
def count_recurse(n):
     if n < 0:
         return 0
     elif n == 0:
         return 1
     else:
         return count_recurse(n - 1)
            + count_recurse(n - 5)
            + count_recurse(n - 10)
            + count_recurse(n - 20)
```



i have to figure out how to get $26

if 0: +1
if < 0: +0

count

if i use $1 as my first denomination

i have to figure out how to get the $25

.
.
.

if i use $20 as my first denomination

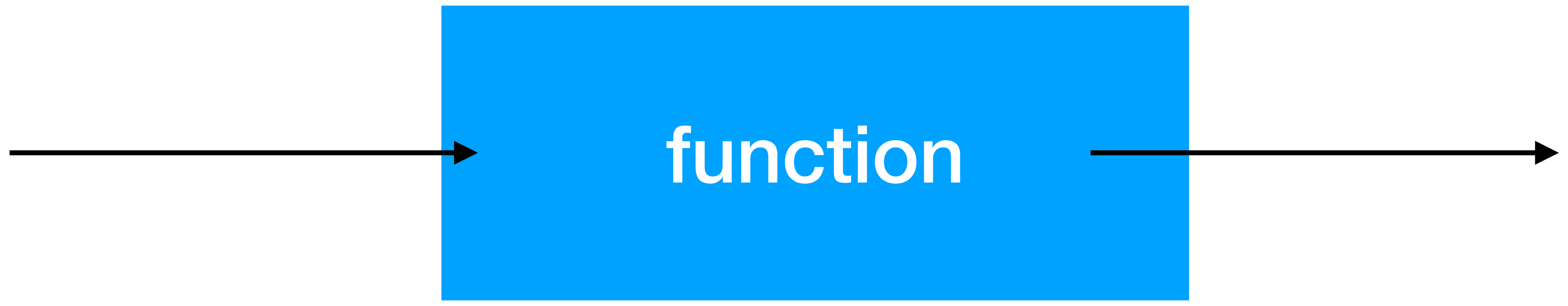
i have to figure out how to get the $6

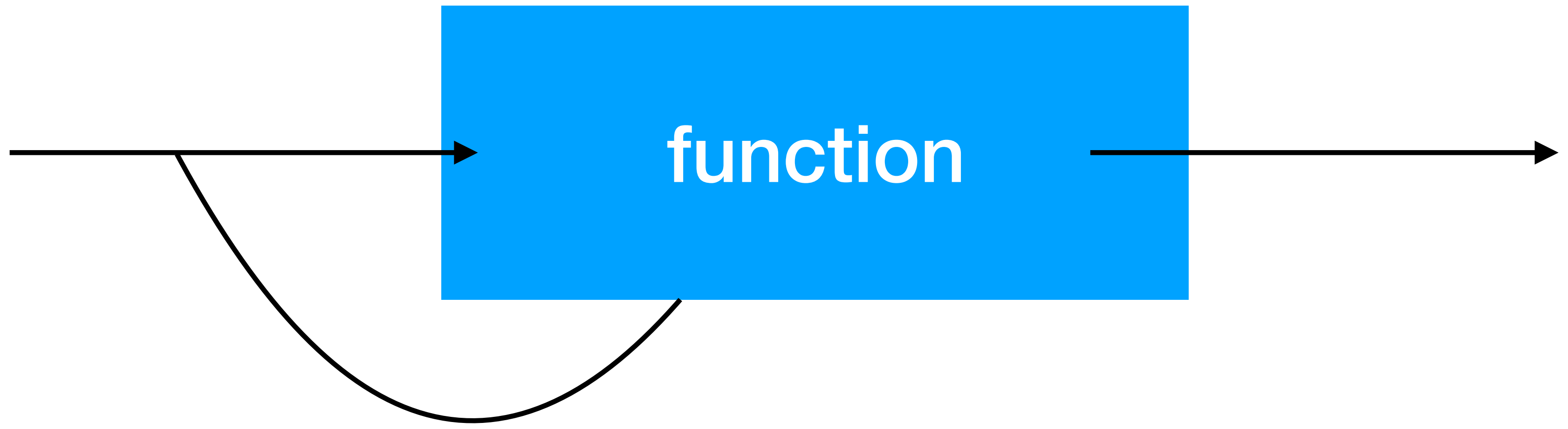
```
def count(n):
     total = 0
     options = [n]
     while len(options) > 0:
         curr = options.pop(0)
         for change in [1, 5, 10, 20]:
            val = curr - change
             if val == 0:
                 total += 1
             elif val > 0:
                 options.append(val)
     return total
```

```
def count_recurse(n):
     if n < 0:
         return 0
     elif n == 0:
         return 1
     else:
         return count_recurse(n - 1)
            + count_recurse(n - 5)
            + count_recurse(n - 10)
            + count_recurse(n - 20)
```

```
def count(n):
     total = 0
     options = [n]
     while len(options) > 0:
         curr = options.pop(0)
         for change in [1, 5, 10, 20]:
            val = curr - change
             if val == 0:
                 total += 1
             elif val > 0:
                 options.append(val)
     return total
```

```
def count_recurse(n):
     if n < 0:
         return 0
     elif n == 0:
         return 1
     else:
         return count_recurse(n - 1)
            + count_recurse(n - 5)
            + count_recurse(n - 10)
            + count_recurse(n - 20)
```





```
def co
     to
     op              ]
     whi
         c                 (0)
         fo              , 5, 10, 20]:
                         change



                              d(val)
               al
```

```
def count_recurse(n):
     if n < 0:
         return 0
     elif n == 0:
         return 1
     else:
         return count_recurse(n - 1)
            + count_recurse(n - 5)
            + count_recurse(n - 10)
            + count_recurse(n - 20)
```

example too complex… for now

so what does this mean

we have a strategy on how to create recursive functions

we can see that recursion isn’t pointless…
at least for more complex problems

CS61A

midterm 1

10% of your grade

there’s extra credit on every project

epa points

CS61A

midterm 1

10% of your grade

there’s extra credit on every project

epa points

one test isn’t going to determine
if you should be a computer scientist