

Welcome to Data C88C!

Lecture 02: Functions

Tuesday, June 24th, 2025

Week 1

Summer 2025

Instructor: Eric Kim (ekim555@berkeley.edu)

Announcements

- Lab starts today! "Lab00"
 - Lab Zoom links here: [\[link\]](#)
 - Due: Sun June 29th, 11:59 PM PST
- HW01 released today! [\[link\]](#)
 - Due: Sun June 29th, 11:59 PM PST
 - Tip: ALL assignments this semester are due at 11:59 PM PST
- Submit both Lab and HW on Gradescope
 - Autograders
- **Reminder**: watch YouTube video BEFORE lecture!
 - See course website for video link

Important: watch these videos before lecture to maximize learning!

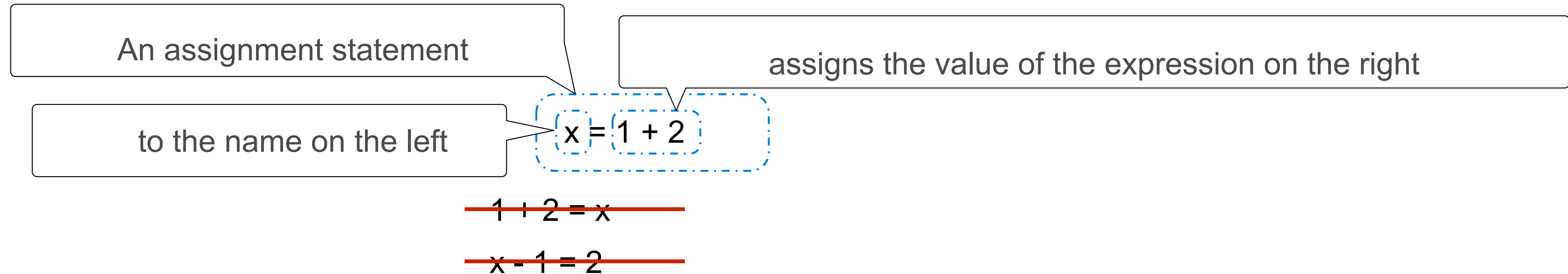
Calendar					
Week	Date	Lecture	Textbook	Lab & Discussion Links	Homework & Project
1	Mon 6/23	Welcome		Disc 00: Getting Started	
	Tu 6/24	Functions Videos	<div>Ch. 1.1</div> <div>Ch. 1.2</div> <div>Ch. 1.3</div>	Disc 01: Functions Lab 00: Getting Started <div>Due Sun 6/29</div>	HW 01: Functions <div>Due Sun 6/29</div>
	Wed 6/25	Control Videos	<div>Ch. 1.4</div> <div>Ch. 1.5</div>	Disc 02: Control, Environment Diagrams	

Lecture Overview

- Functions
- Environment Diagrams ("V1")
- Print and None
- Big Demo: Small Expressions

Assignment Statements

Assignment Statements



The expression (right) is evaluated, and its value is assigned to the name (left).

```
>>> x = 2
>>> y = x + 1
>>> y
3
>>> x = 5
>>> x
5
>>> y
3
```

Note: we can also assign names to functions, not just numbers!

[\[Demo 02.py:Demo00\]](#)

Python Tutor: max vs pow visualized

```
>>> max(2, 10)
10
>>> pow(2, 10)
1024
>>> max = pow
>>> pow(2, 10)
1024
>>> pow = max
>>> pow(2, 10)
1024
>>> max = pow
>>> pow(2, 10)
1024
```

Python 3.6
([known limitations](#))

```
1 # (setup) tell python tutor to show initial max/pow val
2 max = max
3 pow = pow
4 print(max(2, 10))
5 print(pow(2, 10))
6 max = pow
7 print(pow(2, 10))
8 pow = max
9 print(pow(2, 10)) # tricky: is this 10, or 1024?
10 max = pow
11 print(pow(2, 10))
```

[Edit this code](#)

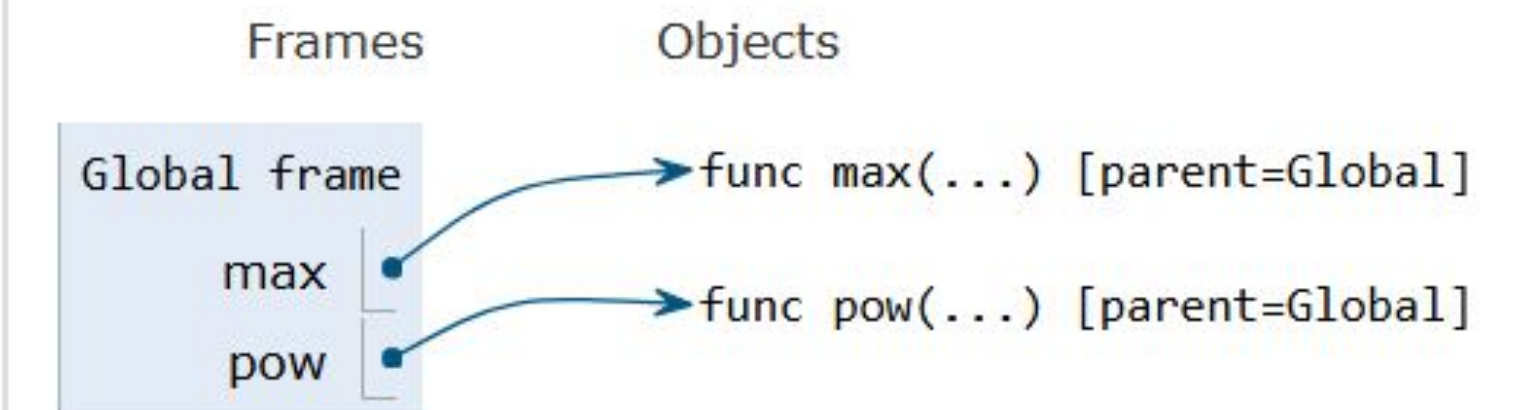
→ line that just executed
→ next line to execute

[Customize visualization](#)

Step 5 of 10

Print output (drag lower right corner to resize)

```
10
1024
```



Python Tutor: <https://pythontutor.com/cp/composingprograms.html#mode=edit>

Very nice web tool for visualizing Python

[Demo] pow vs max example: [\[link\]](#)

Name lookup in user-defined functions

```
→ x = 2  
  
def g(y):  
→   x = 2 * y  
   return x + 1  
→
```

Observation: there are multiple `x` names here! How do we know which `x` refers to which value?

Question: what does `g(x)` return?

Answer: 5

The `x` in `return x + 1` is different than the `x` in the global scope (`x = 2`).
Sound hand-wavy? Let's formalize this a little more, using a tool called **"Environment Diagrams"**

Environment Diagrams

What are Environment Diagrams?

- Environment diagrams are a helpful way of visualizing how Python looks up variable values during execution
- Sort of a "simulation" of the Python interpreter
- Tip: Python Tutor [\[link\]](#) will step-by-step draw the environment diagrams for any Python code you give it! Super useful for studying.

Python 3.6
([known limitations](#))

```
1 def cool_function(x):  
2     y = 42  
3     return x * y  
4  
5 print(cool_function(2))
```

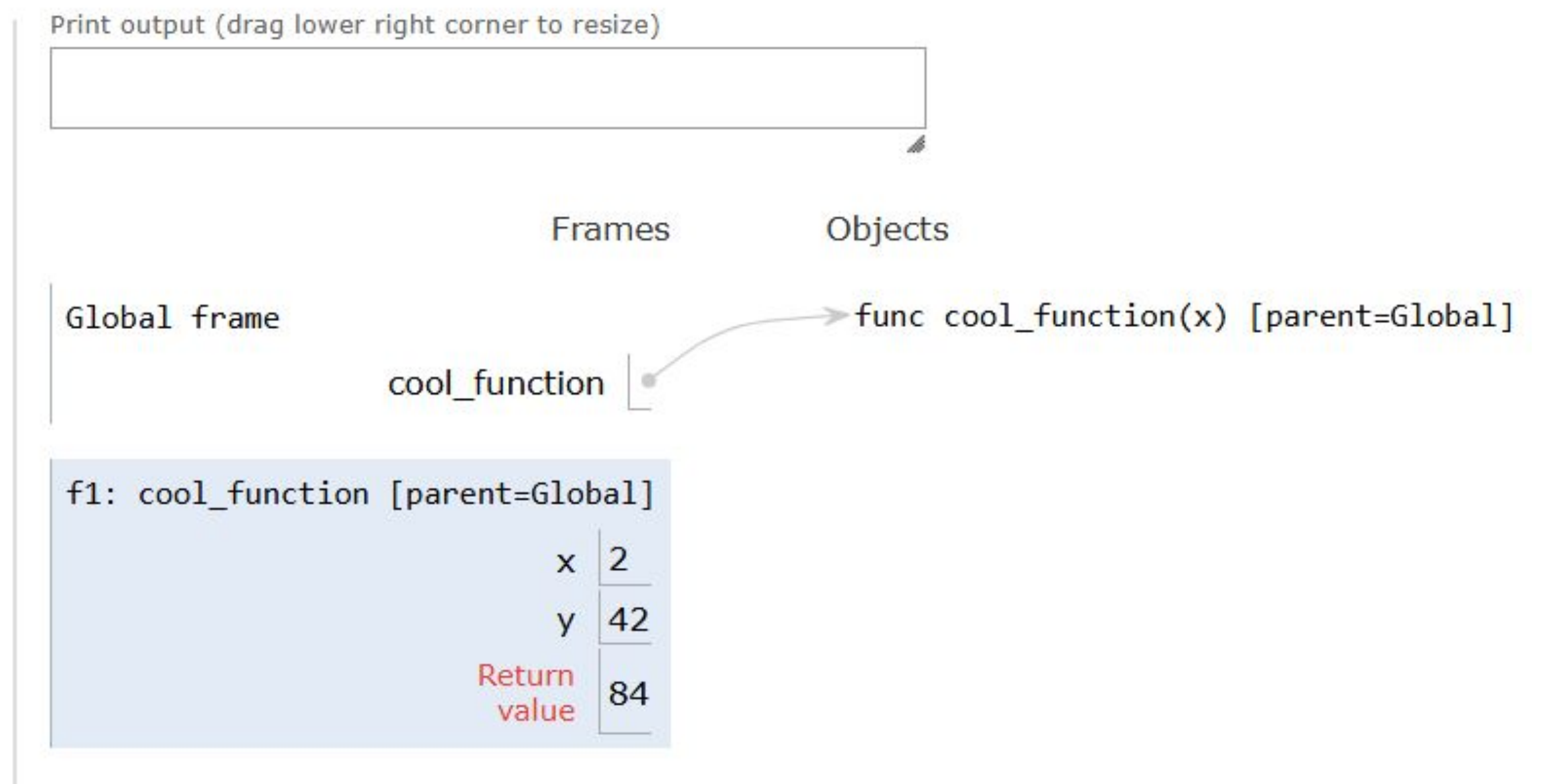
[Edit this code](#)

→ line that just executed
→ next line to execute

<< First < Prev Next > Last >>

Step 6 of 6

[Customize visualization](#)

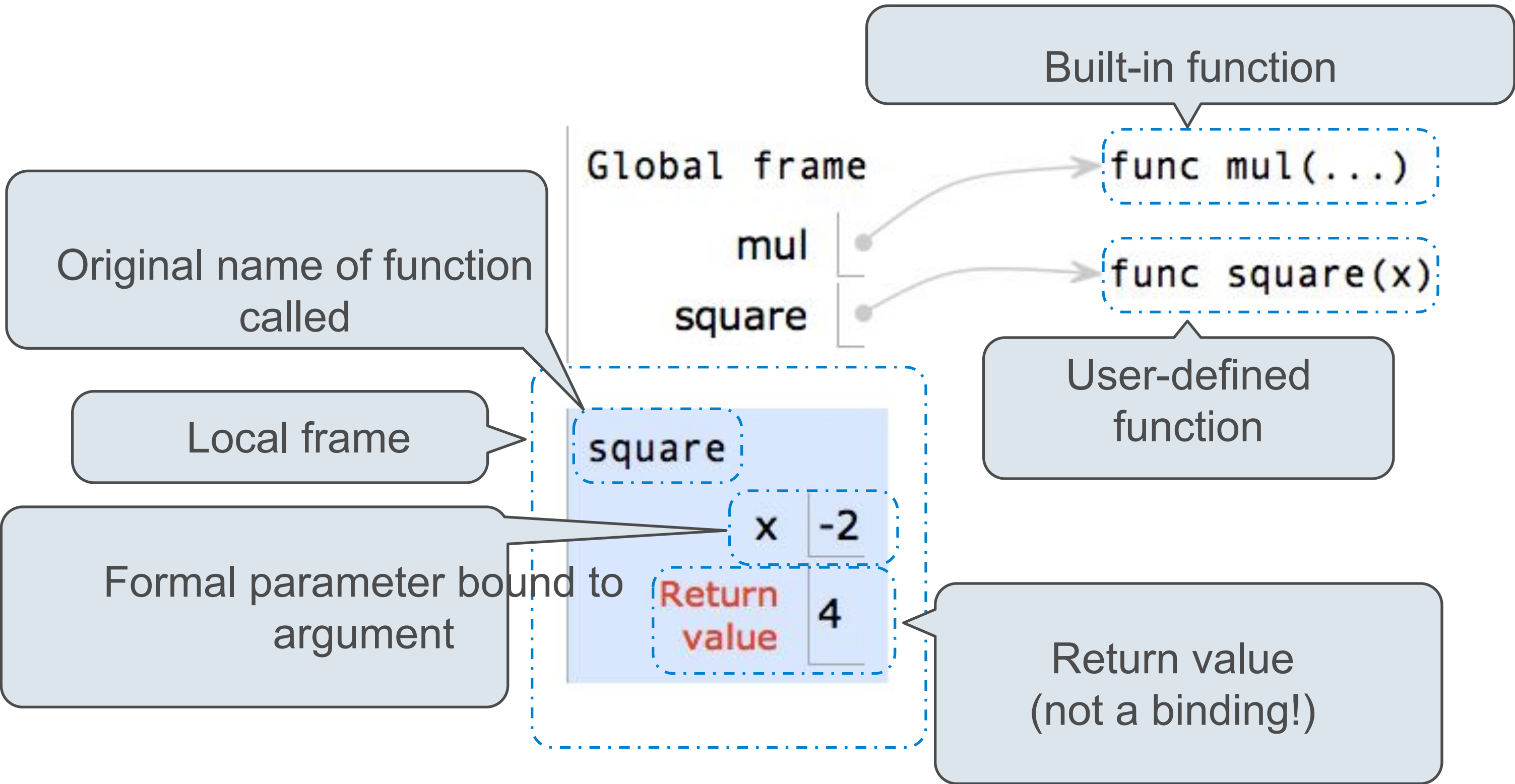


Calling User-Defined Functions

Procedure for calling/applying user-defined functions (version 1):

1. Add a local frame, forming a new environment
2. Bind the function's formal parameters to its arguments in that frame
3. Execute the body of the function in that new environment

```
1 from operator import mul
2 def square(x):
3     return mul(x, x)
4 square(-2)
```



Frames & Environments

Frame: Holds name-value bindings; looks like a box; no repeated names allowed!

Global frame: The frame with built-in names (min, pow, etc.)

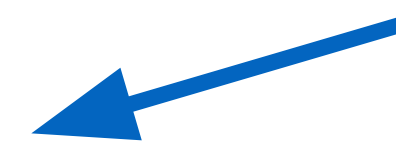
Environment: A sequence of frames that always ends with the global frame

Lookup: Find the value for a name by looking in each frame of an environment

A name (which is a type of expression) such as **x** is evaluated by looking it up

So far, the "sequence of frames" is always very short (length 2): the function's frame, followed by the global frame.

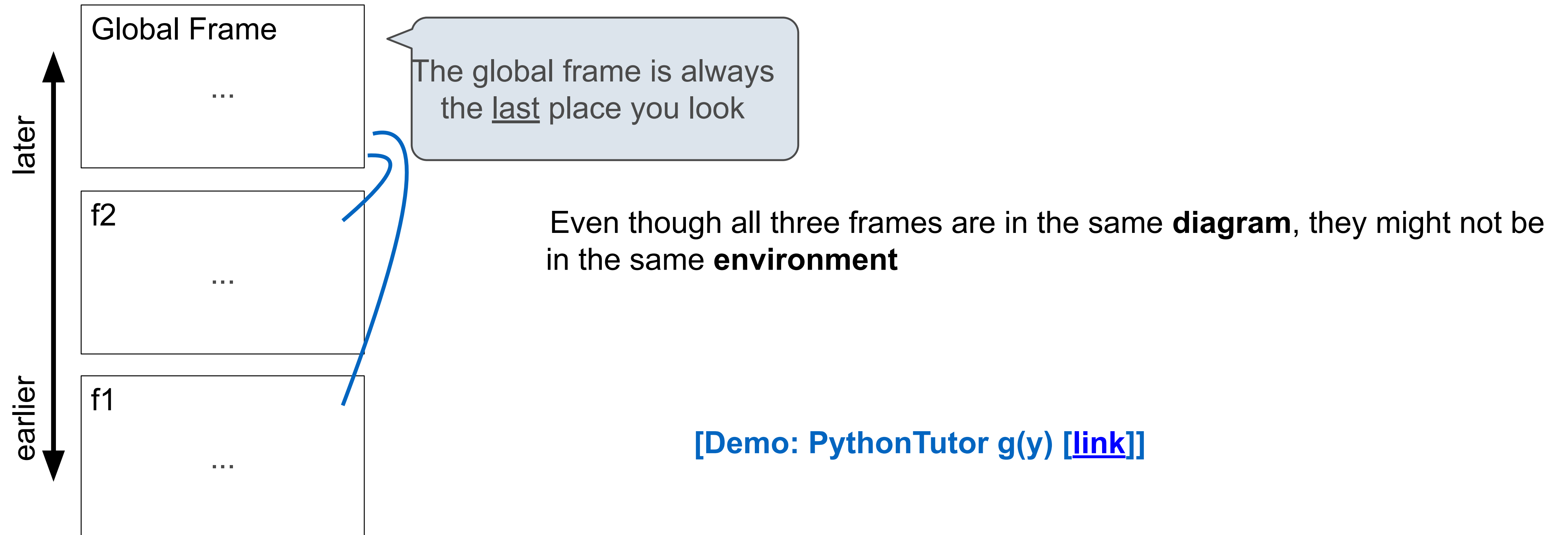
But, later when we add **nested (aka inner) functions**, we will have longer frame sequences!



A Sequence of Frames

An environment is a sequence of frames.

A name evaluates to the value bound to that name in the earliest frame of the current environment in which that name is found.

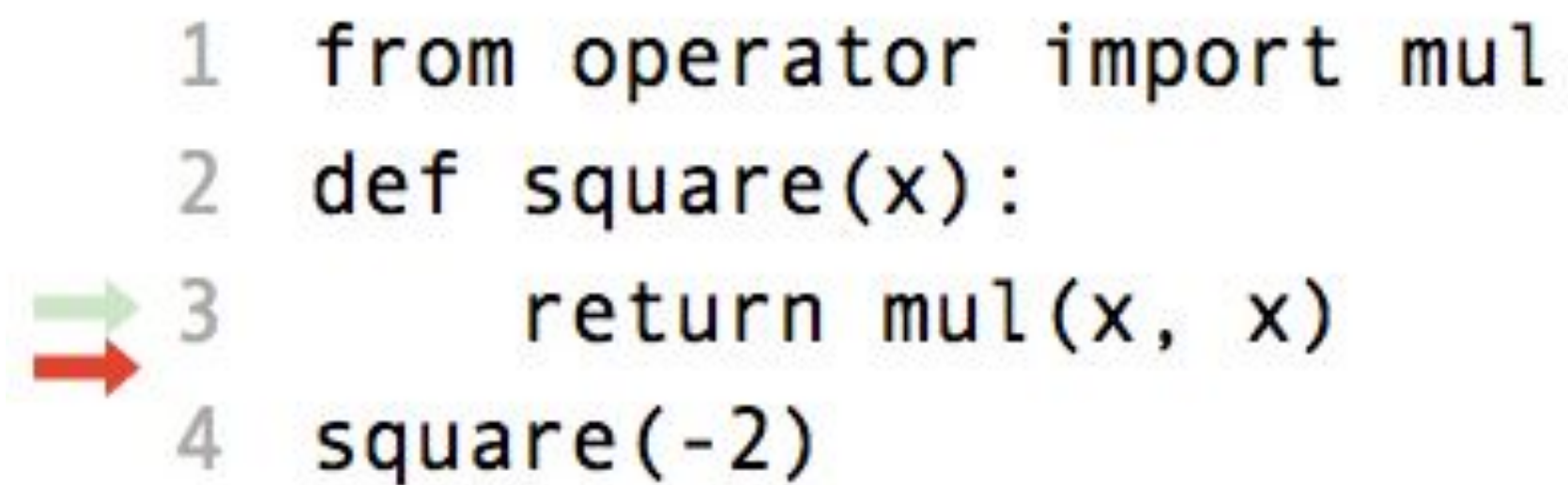


Frames & Environments

Why organize information this way?

- Local context before global context
- Calling or returning changes the local context
- Assignment within a function's local frame doesn't affect other frames

```
1 from operator import mul
2 def square(x):
3     return mul(x, x)
4 square(-2)
```

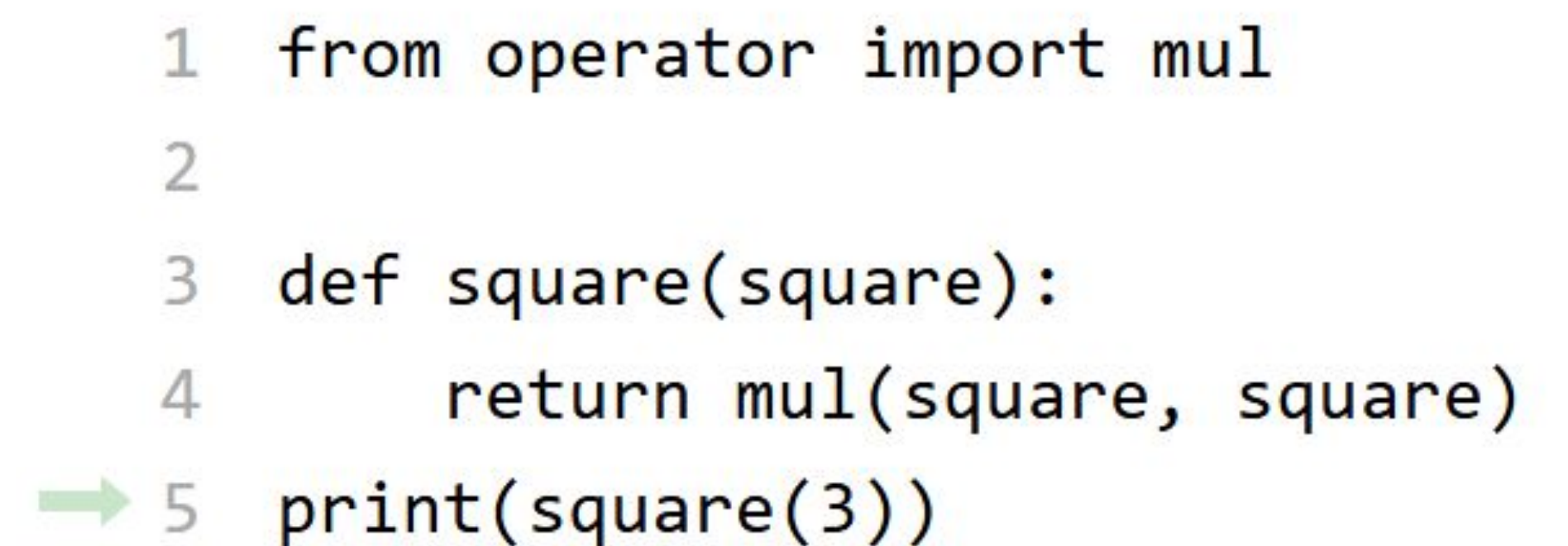


Question: What
Would Python Print?

Answer: 4

[Demo: Python Tutor [\[link\]](#)]

```
1 from operator import mul
2
3 def square(square):
4     return mul(square, square)
5 print(square(3))
```



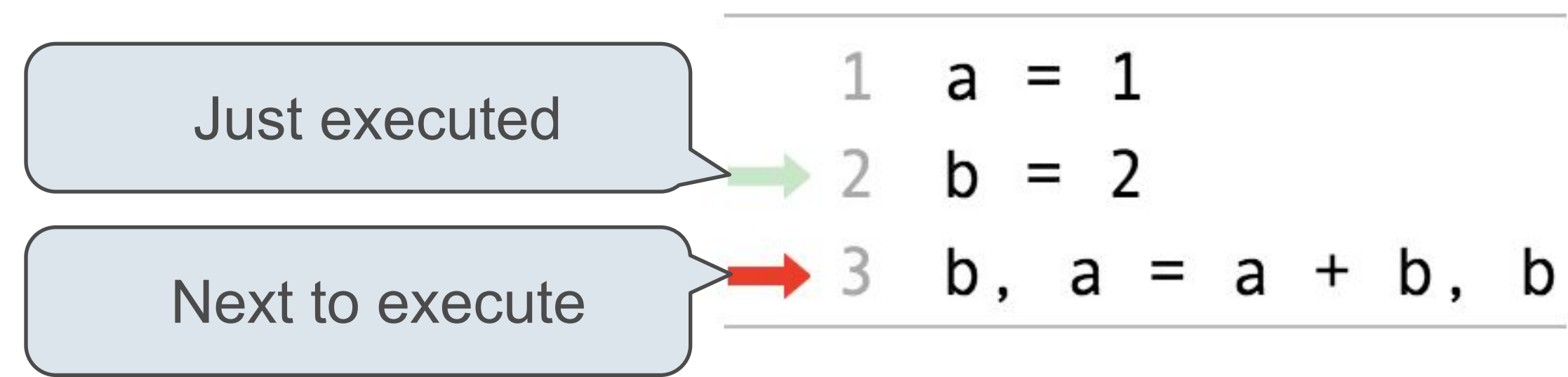
Question: What
Would Python Print?

Answer: 9

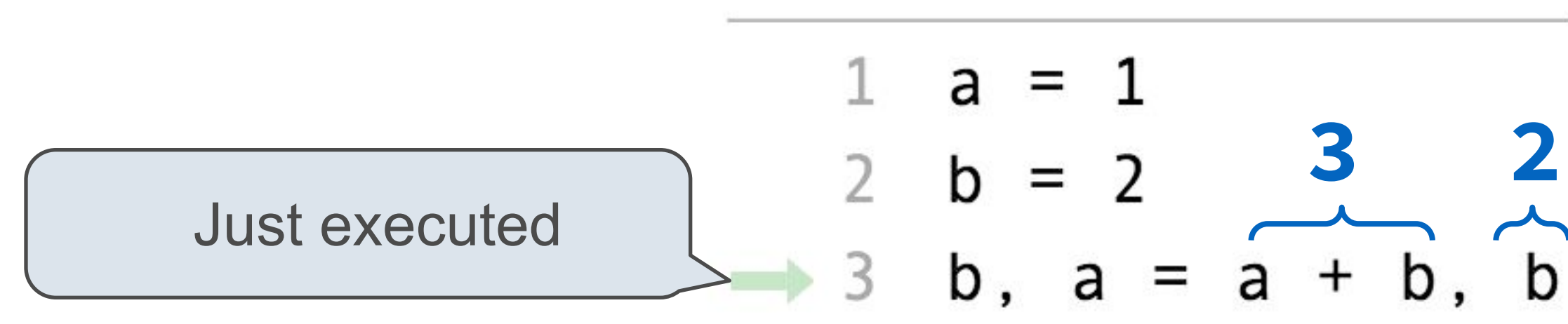
[Demo: Python Tutor [\[link\]](#)]

Multiple Assignment

Multiple Assignment



Global frame	
a	1
b	2



Global frame	
a	2
b	3

b = 3
a = 2

Execution rule for assignment statements:

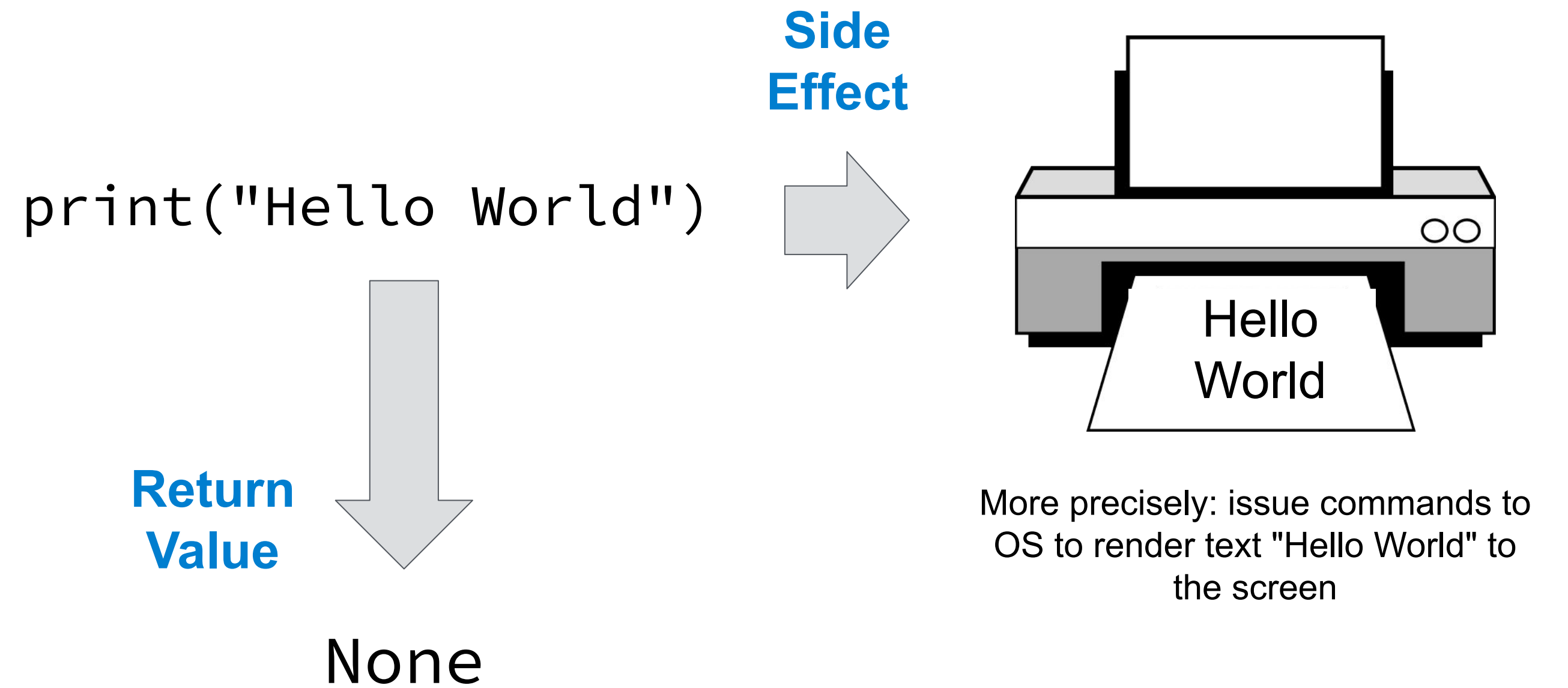
1. Evaluate all expressions to the right of = from left to right.
2. Bind all names to the left of = to those resulting values in the current frame.

Print and None

(Demo: 02.py:Demo01)

Print and None

- `print()` function is "special" in that it has a side-effect
 - **Side-effect**: outputs something to the screen
 - **Return value**: `None`
- In Python, `None` is a special value that means the "nothing" or "empty" value
 - Similar in spirit to `NULL`/nullpointer in other languages



Tip: be sure that you fully understand the difference between function return values and `print()`!

Pro Tip: this often shows up in exams!

Small Expressions

Problem Definition

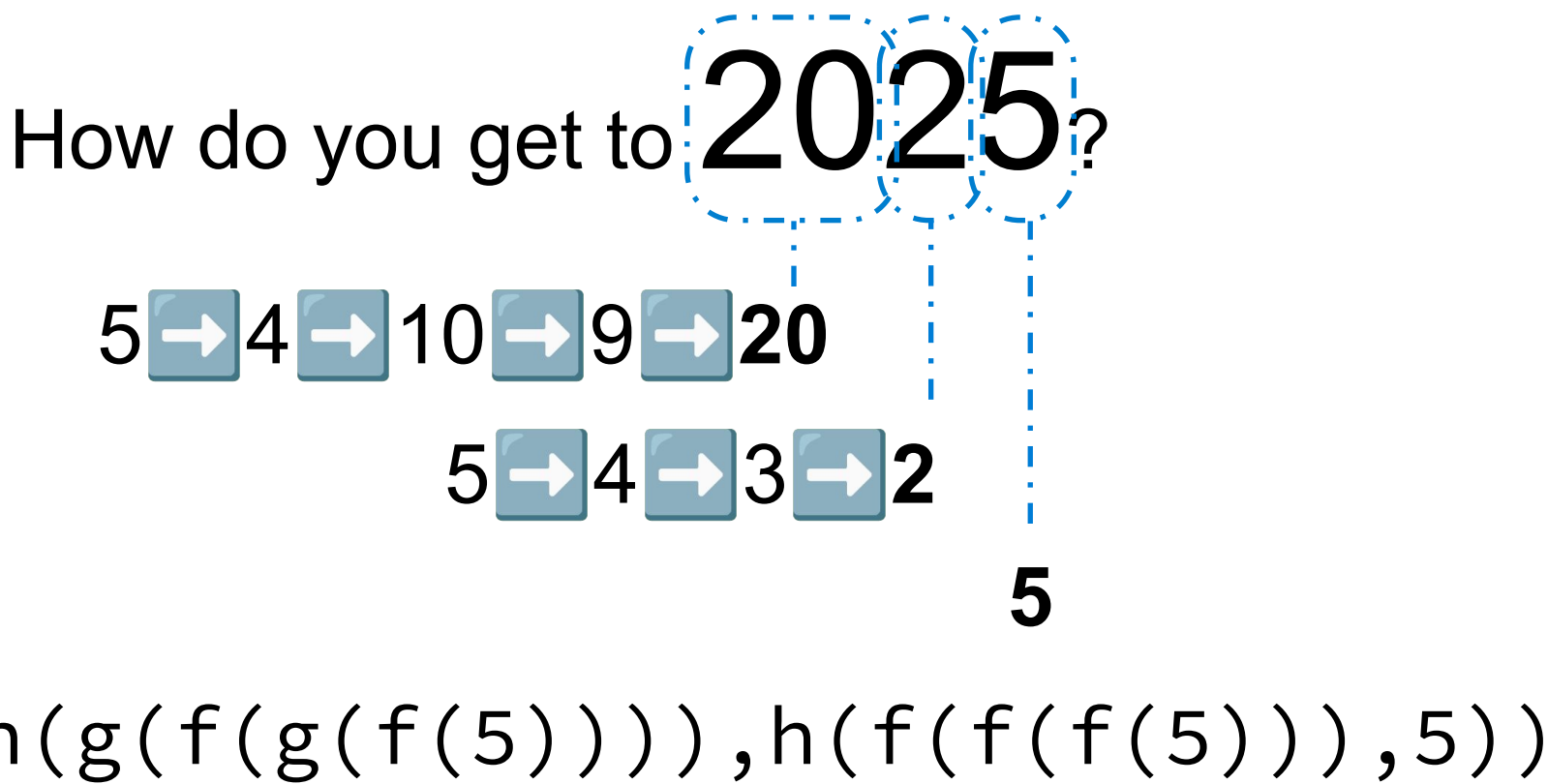
Imagine you can call only the following three functions:

- $f(x)$: decrement an integer x to get $x-1$
- $g(x)$: increment then double an integer x to get $2*(x+1)$
- $h(x, y)$: Concatenates the digits of two different positive integers x and y . For example, $h(789, 12)$ evaluates to 78912 and $h(12, 789)$ evaluates to 12789.

Definition: A *small expression* is a call expression that contains only f , g , h , the number 5, and parentheses. All of these can be repeated. For example, $h(g(5), f(f(5)))$ is a small expression that evaluates to 103.

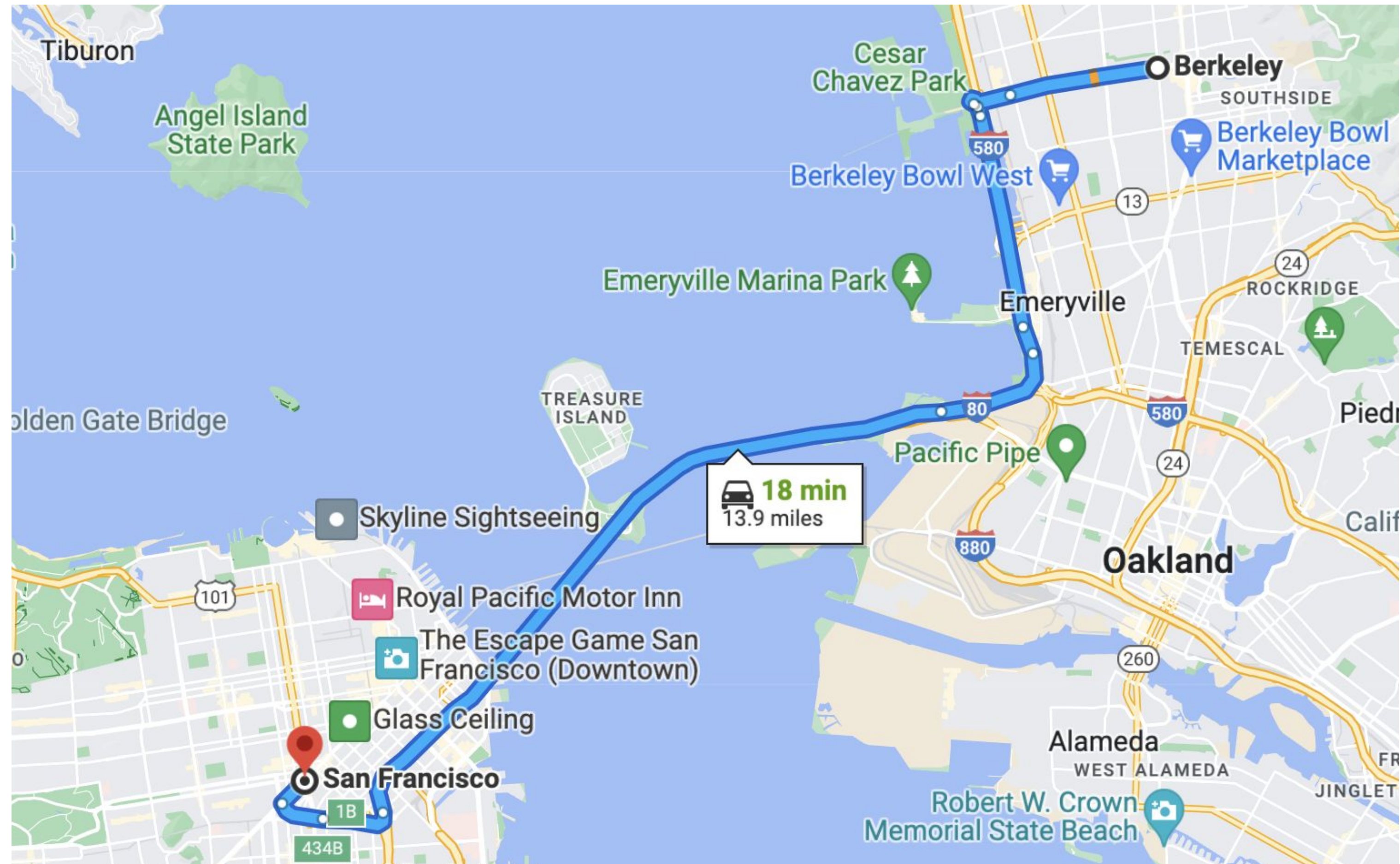
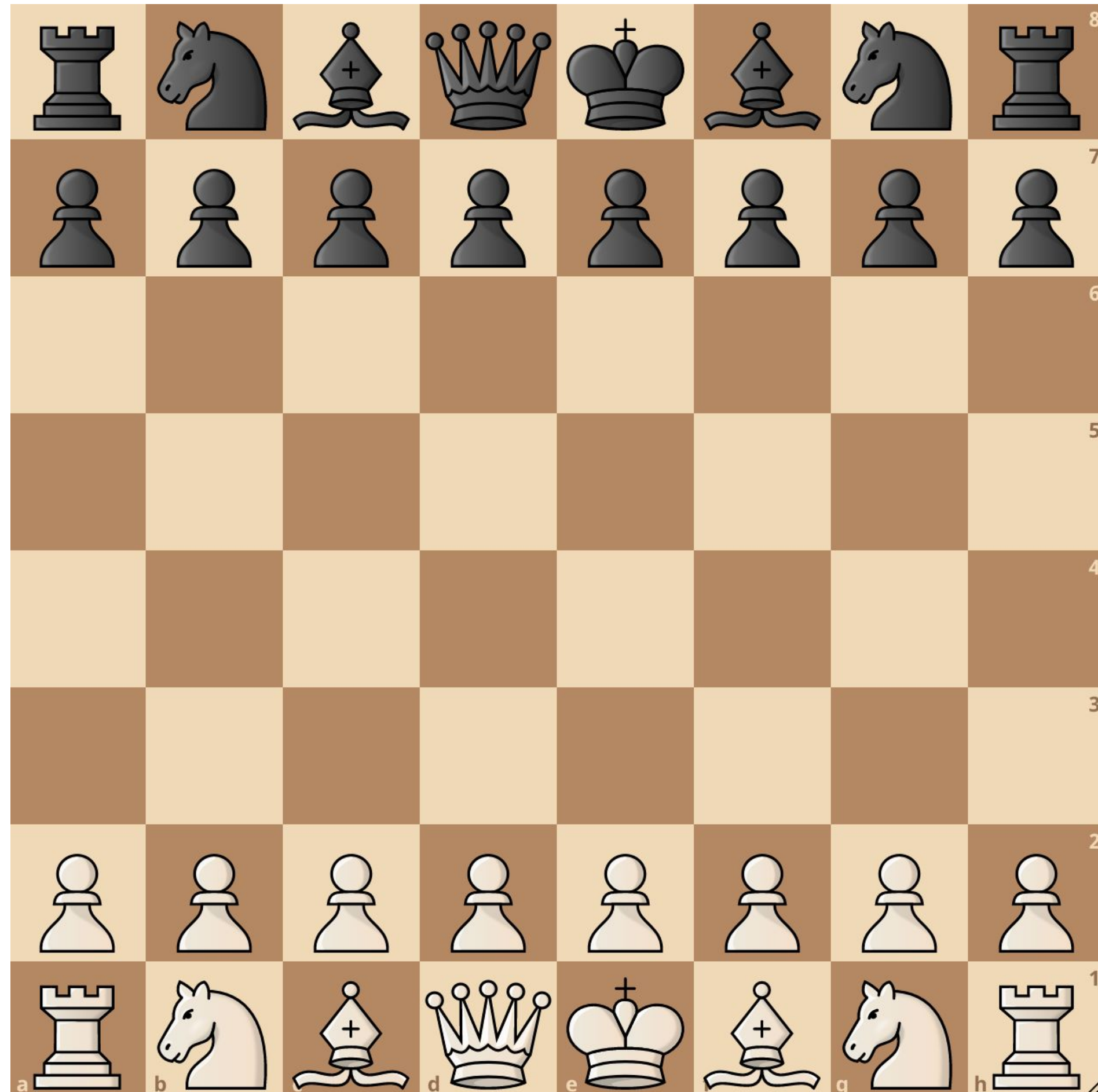
What's the **shortest** *small expression* you can find that evaluates to 2025?

Fewest calls?
Shortest length when written?



- Effective problem solving:**
- Understand the problem
 - Come up with ideas
 - Turn those ideas into solutions

Search



A common strategy: try a bunch of options to see which is best ("**exhaustive/brute-force search**")

Computer programs can evaluate many alternatives by repeating simple operations (Fortunately, computers are fast!)

A Computational Approach

Try all the small expressions with 3 function calls, then 4 calls, then 5 calls, etc.

f(f(f(5))) -> 2
g(f(f(5))) -> 8
f(g(f(5))) -> 9
g(g(f(5))) -> 22
f(f(g(5))) -> 10
g(f(g(5))) -> 24
f(g(g(5))) -> 25
g(g(g(5))) -> 54

f(f(h(5,5))) -> 53
g(f(h(5,5))) -> 110
f(g(h(5,5))) -> 111
g(g(h(5,5))) -> 226
f(h(5,f(5))) -> 53
g(h(5,f(5))) -> 110
f(h(5,g(5))) -> 511
g(h(5,g(5))) -> 1026
f(h(5,h(5,5))) -> 554
g(h(5,h(5,5))) -> 1112
f(h(f(5),5)) -> 44
g(h(f(5),5)) -> 92
f(h(g(5),5)) -> 124
g(h(g(5),5)) -> 252
f(h(h(5,5),5)) -> 554
g(h(h(5,5),5)) -> 1112
...

h(5,f(f(5))) -> 53
h(5,g(f(5))) -> 510
h(5,f(g(5))) -> 511
h(5,g(g(5))) -> 526
h(5,f(h(5,5))) -> 554
h(5,g(h(5,5))) -> 5112
h(5,h(5,f(5))) -> 554
h(5,h(5,g(5))) -> 5512
h(5,h(5,h(5,5))) -> 5555
h(5,h(f(5),5)) -> 545
h(5,h(g(5),5)) -> 5125
h(5,h(h(5,5),5)) -> 5555
h(f(5),f(5)) -> 44
h(f(5),g(5)) -> 412
h(f(5),h(5,5)) -> 455
...

h(g(5),f(5)) -> 124
h(g(5),g(5)) -> 1212
h(g(5),h(5,5)) -> 1255
h(h(5,5),f(5)) -> 554
h(h(5,5),g(5)) -> 5512
h(h(5,5),h(5,5)) -> 5555
h(f(f(5)),5) -> 35
h(g(f(5)),5) -> 105
h(f(g(5)),5) -> 115
h(g(g(5)),5) -> 265
h(f(h(5,5)),5) -> 545
h(g(h(5,5)),5) -> 1125
h(h(5,f(5)),5) -> 545
h(h(5,g(5)),5) -> 5125
h(h(5,h(5,5)),5) -> 5555
h(h(f(5),5),5) -> 455
h(h(g(5),5),5) -> 1255
h(h(h(5,5),5),5) -> 5555
...

Reminder: f(x) decrements; g(x) increments then doubles; h(x, y) concatenates

A Computational Approach

Try all the small expressions with 3 function calls, then 4 calls, then 5 calls, etc.

Filter for just the attempts that result in desired output: 2025.

$f(g(h(g(f(5)),g(5)))) \rightarrow$ 2025 has 6 calls

$f(g(h(f(f(g(5))),g(5)))) \rightarrow$ 2025 has 7 calls

$f(g(g(f(g(g(h(g(5),5)))))) \rightarrow$ 2025 has 8 calls

$f(g(g(h(g(g(f(g(5))))),5))) \rightarrow$ 2025 has 8 calls

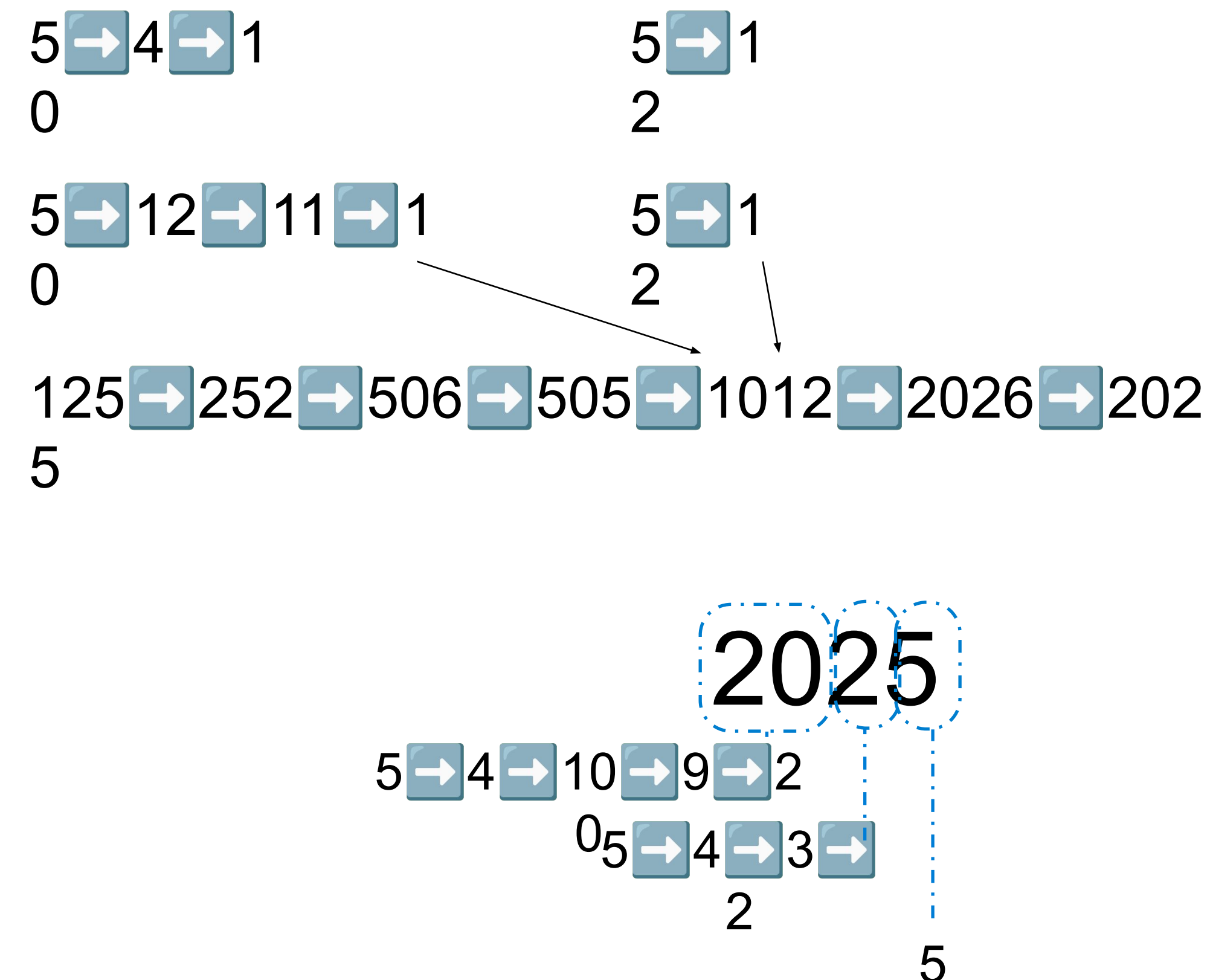
$f(h(g(f(g(f(5)))),g(g(5)))) \rightarrow$ 2025 has 8 calls

$h(g(f(g(f(5))),f(g(g(5)))) \rightarrow$ 2025 has 8 calls

$h(g(g(f(g(g(f(g(5)))))),5) \rightarrow$ 2025 has 8 calls

$h(g(g(h(f(5),f(g(f(5))))),5) \rightarrow$ 2025 has 8 calls

$h(g(f(g(f(5))),h(f(f(f(5))),5)) \rightarrow$ 2025 has 9 calls



Reminder: $f(x)$ decrements; $g(x)$ increments then doubles; $h(x, y)$ concatenates

A Computational Approach

(Demo: 02.py:Demo02)

Try all the small expressions with 3 function calls, then 4 calls, then 5 calls, etc.

```
def f(x):  
    return x - 1
```

Functions

```
def g(x):  
    return 2 * (x + 1)
```

```
def h(x, y):  
    return int(str(x) + str(y))
```

Containers

```
class Number:  
    def __init__(self, value):  
        self.value = value
```

Objects

```
    def __str__(self):  
        return str(self.value)
```

Representation

```
    def calls(self):  
        return 0
```

```
class Call:  
    """A call expression."""  
    def __init__(self, f, operands):  
        self.f = f  
        self.operands = operands  
        self.value = f(*[e.value for e in operands])
```

Sequences

```
    def __str__(self):  
        return f'{self.f.__name__}({",".join(map(str, self.operands))})'
```

```
    def calls(self):  
        return 1 + sum(o.calls() for o in self.operands)
```

```
def smalls(n):  
    if n == 0:  
        return [Number(5)]  
    else:
```

Recursion

```
        results = []  
        for operand in smalls(n-1):  
            results.append(Call(f, [operand]))  
            results.append(Call(g, [operand]))
```

Tree
Recursion

```
        for k in range(n):  
            for first in smalls(k):  
                for second in smalls(n-k-1):  
                    if first.value > 0 and second.value > 0:  
                        results.append(Call(h, [first, second]))  
        return results
```

Control

Mutability

```
def sol():  
    for i in range(9):  
        r = [e for e in smalls(i) if e.value == 2025]  
        for e in r:  
            print(e, '->', e.value, 'has', e.calls(), 'calls')
```

Higher-Order Functions

Note: you aren't expected to understand this slide yet. But, by the Midterm you will be able to!