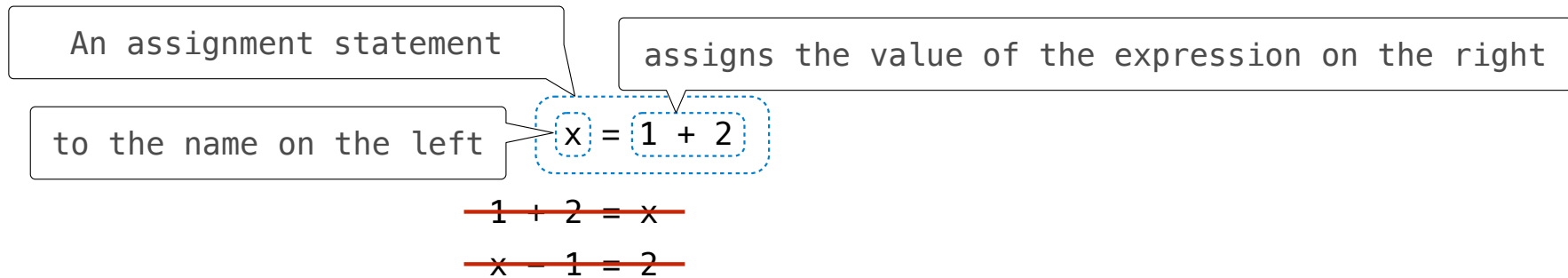


Functions

Announcements

Assignment Statements

Assignment Statements

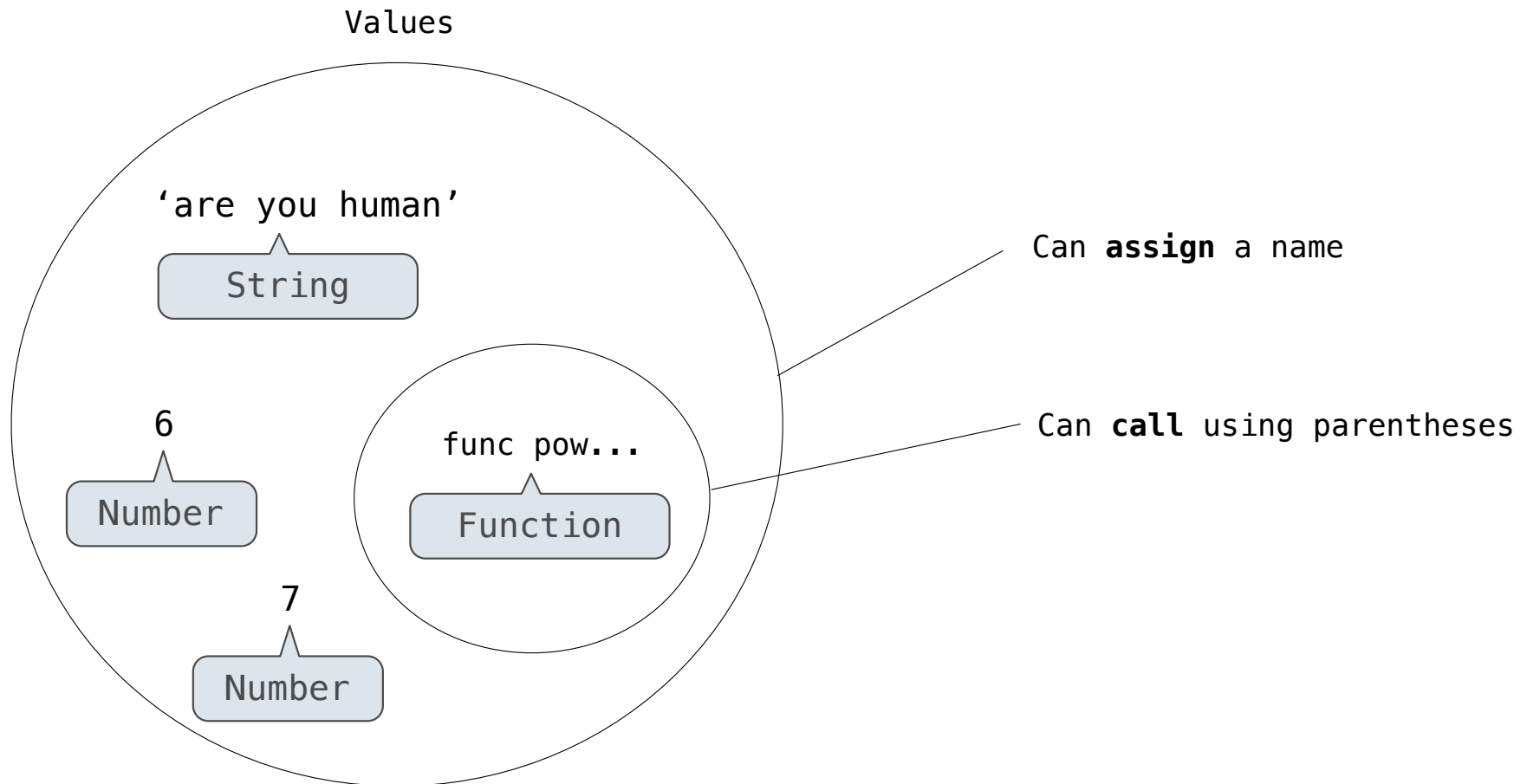


The expression (right) is evaluated, and its value is assigned to the name (left).

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

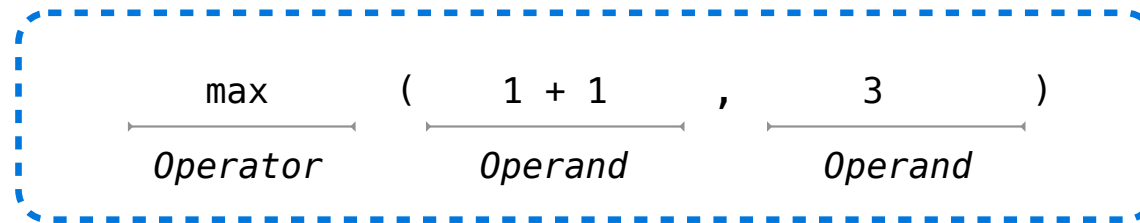
(Demo)

Functions, Values, and Calling



Lecture 1: Anatomy of a Call Expression

Call Expression



Evaluation Procedure for call expressions

(1) Evaluate operator



function

(2) Evaluate each operand



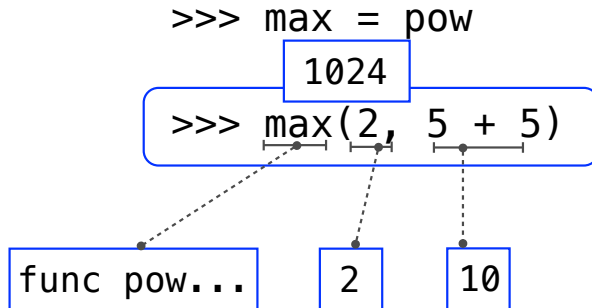
arguments

(3) **Apply** the **function** to the **arguments**

Lecture 1: Anatomy of a Call Expression

- (1) Evaluate operator
- (2) Evaluate each operand
- (3) Apply the function to the arguments

What if you start from scratch and ...



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

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

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



pollev.com/denero

(Demo)

Environment Diagrams

(Demo)

Frames

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

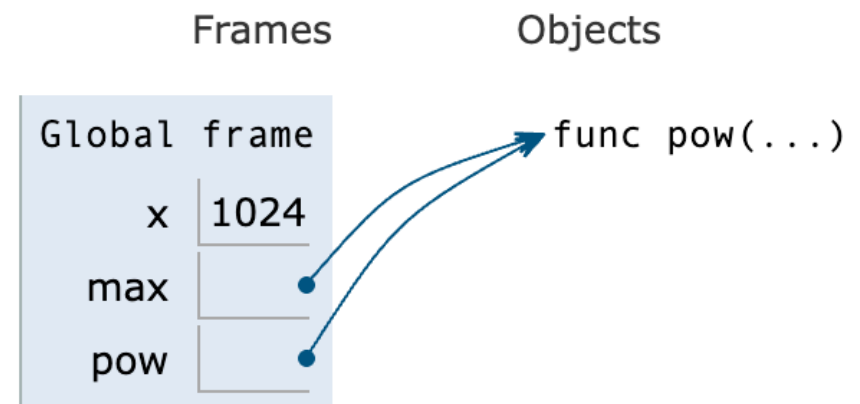
Lookup: Find the value for a name by looking in each frame of an environment

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

Python 3.6
([known limitations](#))

```
1 x = pow(2, 10)
2 max = pow
3 x = pow(2, 10)
4 pow = max
➔ 5 x = pow(2, 10)
```

[Edit this code](#)



User defined functions

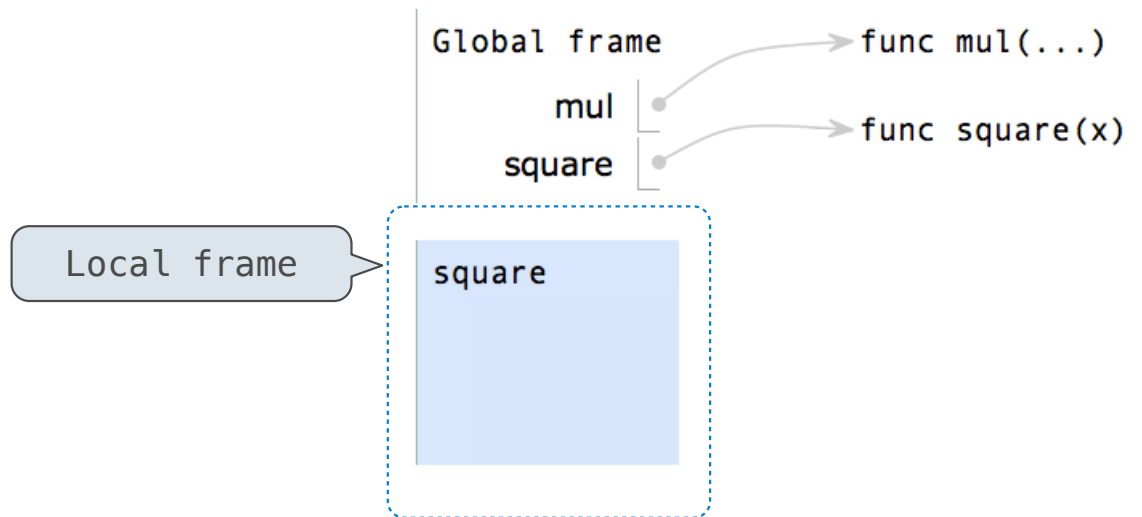
(Demo)

Calling User-Defined Functions

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

1. Add a local frame, forming a new environment

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

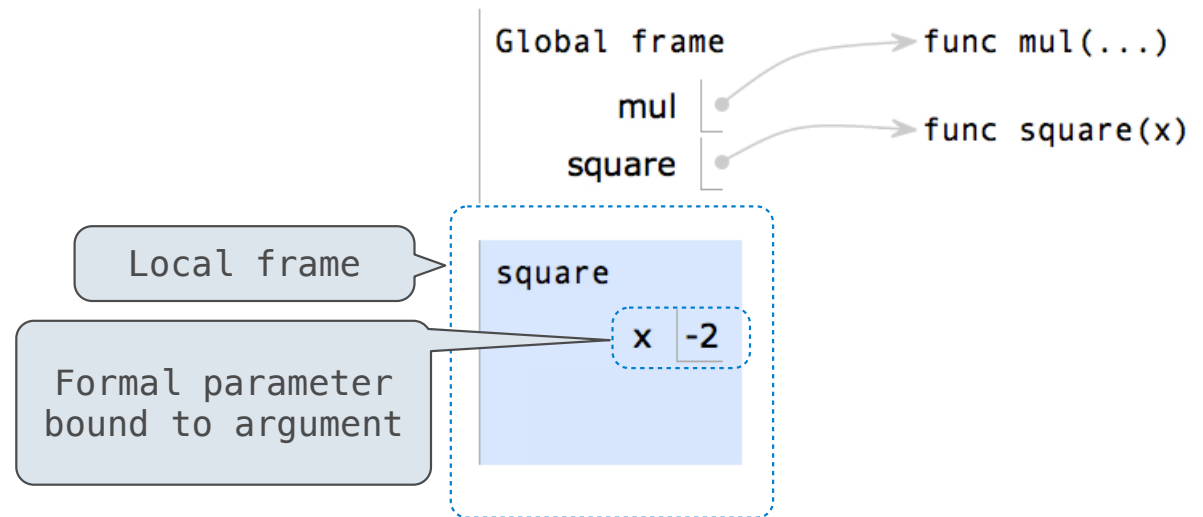


Calling User-Defined Functions

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

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

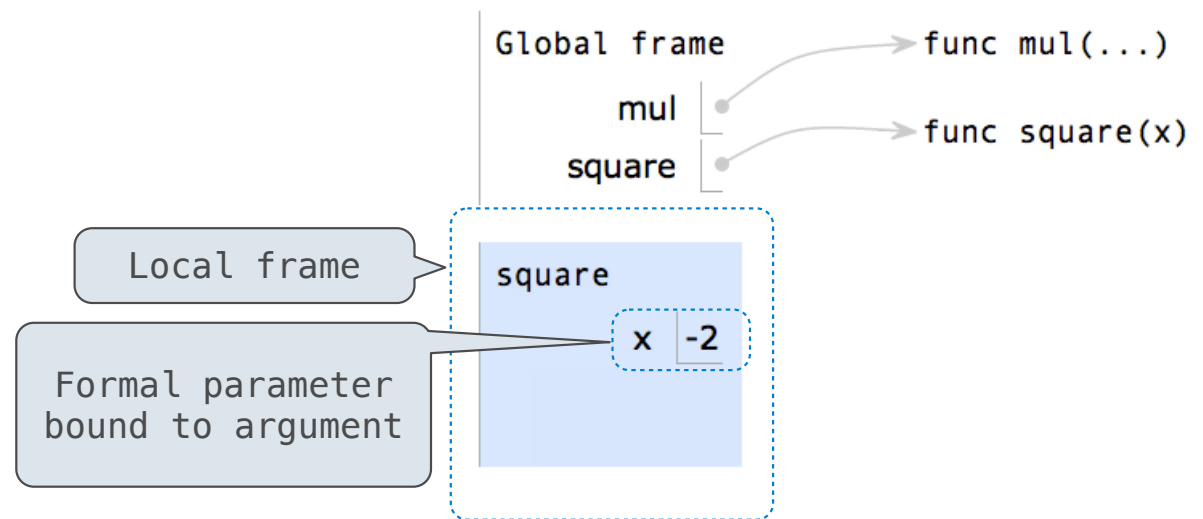


Calling User-Defined Functions

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)
```

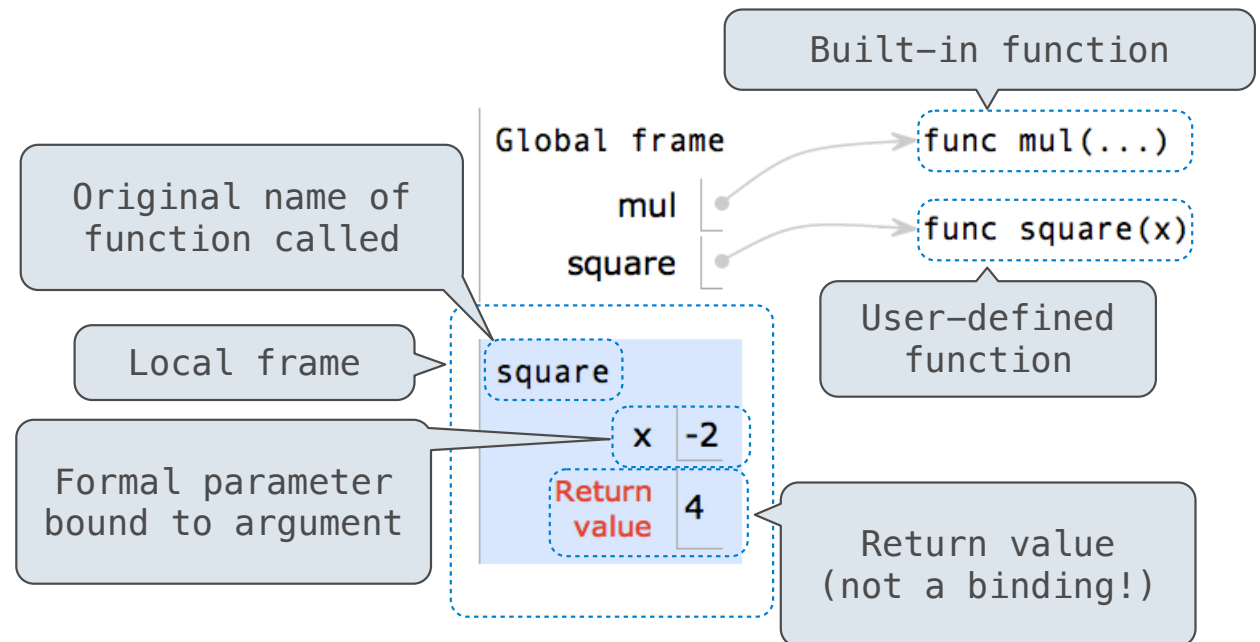


Calling User-Defined Functions

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!

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

A name (which is an expression) such as `x` is evaluated by looking it up

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

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

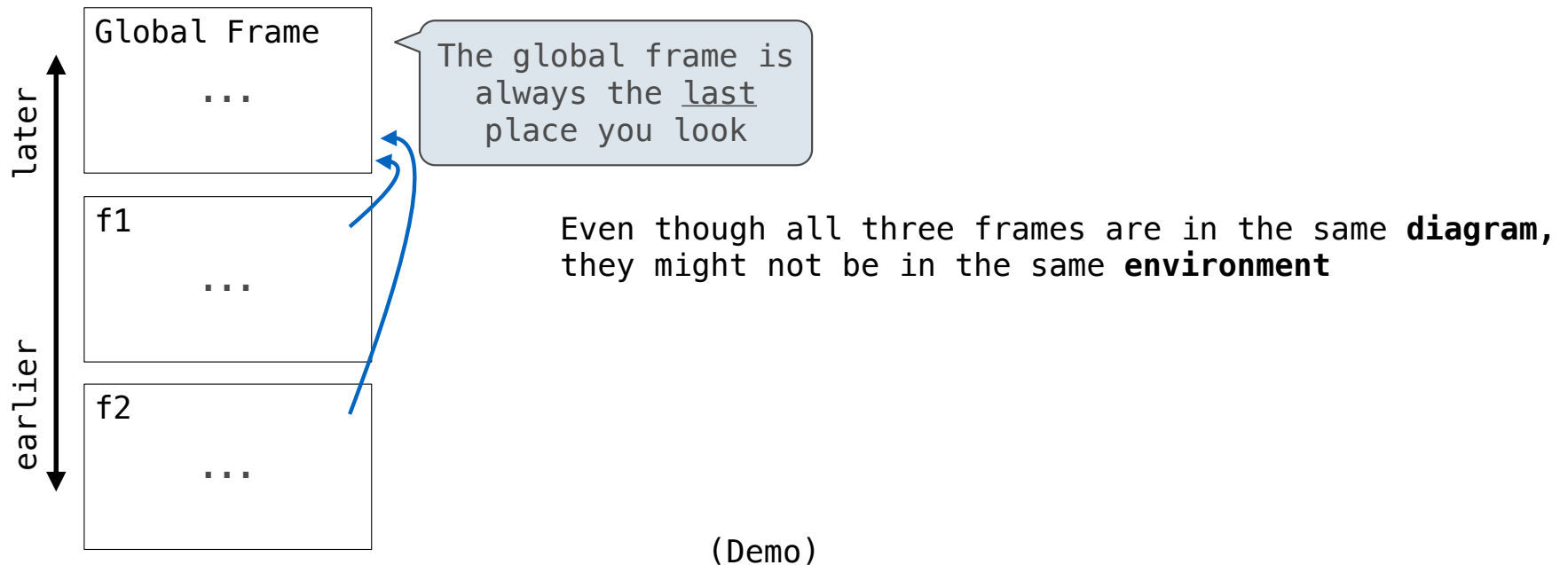
A sequence of frames always has a first frame:

- If the first frame is the global frame, that's the only frame in the environment
- Otherwise, the first frame is local and is followed by the rest of the frames
- Every frame is the first frame of its environment (therefore: one environment per frame)

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)
```

(Demo)

Print and None

(Demo)

Small Expressions

Problem Definition

From Discussion 0:

Imagine you can call only the following three functions:

- $f(x)$: increment an integer x to get $x+1$
- $g(x)$: double then decrement 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, $h(f(g(5)), f(f(5)))$ evaluates to 107.

What's the **shortest** *small expression* you can find that evaluates to 2026?

Fewest calls?
Shortest length when written?

How do you get to **2026**?

5 → 9 → 10 → 19 → 20

$f(g(f(g(5))))$

5 → 6 → 7 → 13 → 25 → 26

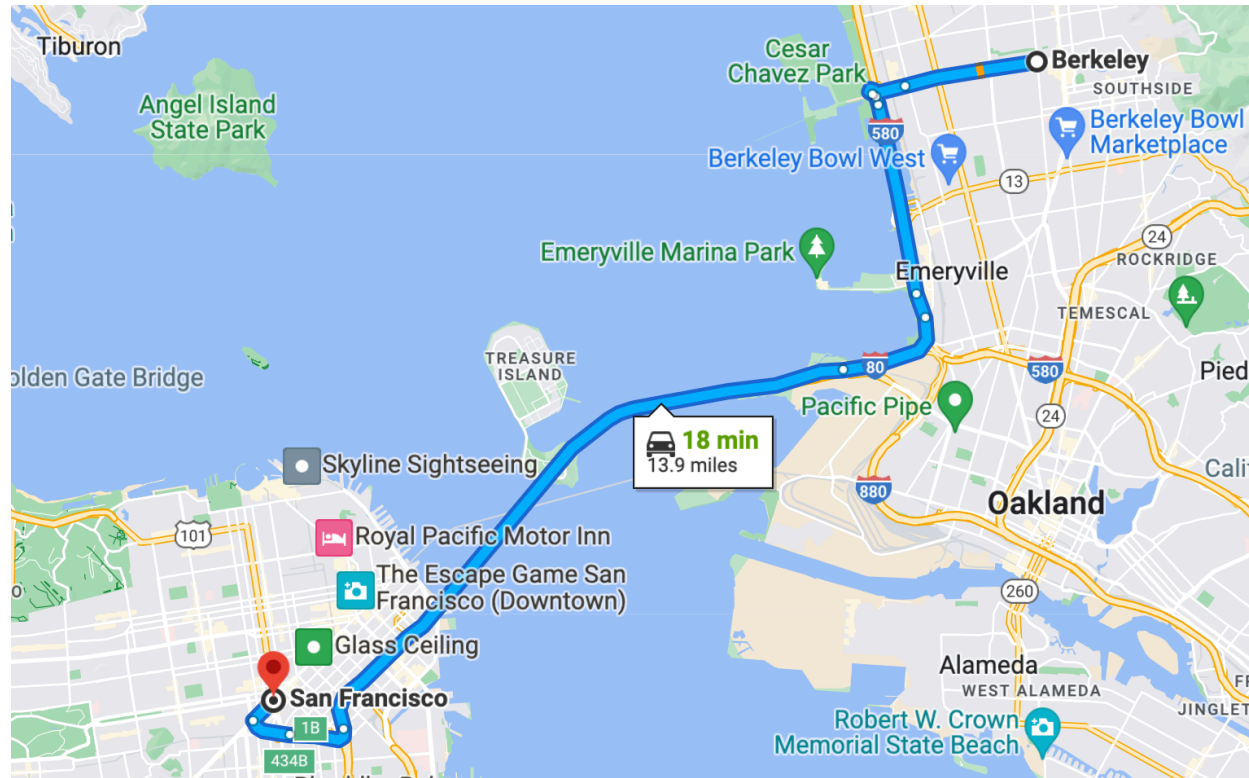
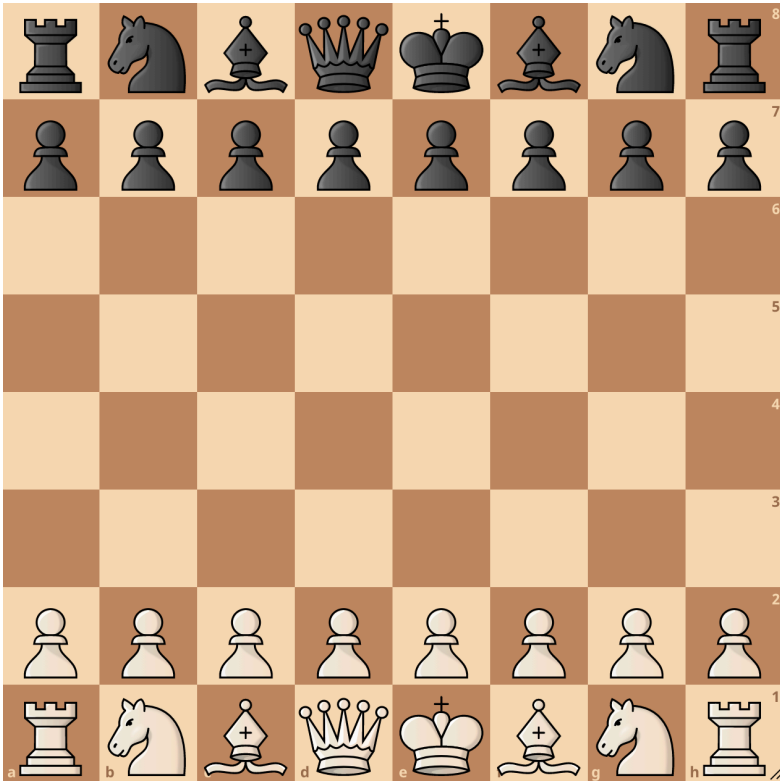


pollev.com/denero

$f(g(g(f(f(5)))))$

$h(f(g(f(g(5)))) , f(g(g(f(f(5))))))$

Search



A common strategy: try a bunch of options to see which is best

Computer programs can evaluate many alternatives by repeating simple operations

A Computational Approach

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

$f(f(f(5))) \rightarrow 8$	$f(f(h(5,5))) \rightarrow 57$	$h(5,f(f(5))) \rightarrow 57$	$h(g(5),f(5)) \rightarrow 96$
$g(f(f(5))) \rightarrow 13$	$g(f(h(5,5))) \rightarrow 111$	$h(5,g(f(5))) \rightarrow 511$	$h(g(5),g(5)) \rightarrow 99$
$f(g(f(5))) \rightarrow 12$	$f(g(h(5,5))) \rightarrow 110$	$h(5,f(g(5))) \rightarrow 510$	$h(g(5),h(5,5)) \rightarrow 955$
$g(g(f(5))) \rightarrow 21$	$g(g(h(5,5))) \rightarrow 217$	$h(5,g(g(5))) \rightarrow 517$	$h(h(5,5),f(5)) \rightarrow 556$
$f(f(g(5))) \rightarrow 11$	$f(h(5,f(5))) \rightarrow 57$	$h(5,f(h(5,5))) \rightarrow 556$	$h(h(5,5),g(5)) \rightarrow 559$
$g(f(g(5))) \rightarrow 19$	$g(h(5,f(5))) \rightarrow 111$	$h(5,g(h(5,5))) \rightarrow 5109$	$h(h(5,5),h(5,5)) \rightarrow 5555$
$f(g(g(5))) \rightarrow 18$	$f(h(5,g(5))) \rightarrow 60$	$h(5,h(5,f(5))) \rightarrow 556$	$h(f(f(5)),5) \rightarrow 75$
$g(g(g(5))) \rightarrow 33$	$g(h(5,g(5))) \rightarrow 117$	$h(5,h(5,g(5))) \rightarrow 559$	$h(g(f(5)),5) \rightarrow 115$
	$f(h(5,h(5,5))) \rightarrow 556$	$h(5,h(5,h(5,5))) \rightarrow 5555$	$h(f(g(5)),5) \rightarrow 105$
	$g(h(5,h(5,5))) \rightarrow 1109$	$h(5,h(f(5),5)) \rightarrow 565$	$h(g(g(5)),5) \rightarrow 175$
	$f(h(f(5),5)) \rightarrow 66$	$h(5,h(g(5),5)) \rightarrow 595$	$h(f(h(5,5)),5) \rightarrow 565$
	$g(h(f(5),5)) \rightarrow 129$	$h(5,h(h(5,5),5)) \rightarrow 5555$	$h(g(h(5,5)),5) \rightarrow 1095$
	$f(h(g(5),5)) \rightarrow 96$	$h(f(5),f(5)) \rightarrow 66$	$h(h(5,f(5)),5) \rightarrow 565$
	$g(h(g(5),5)) \rightarrow 189$	$h(f(5),g(5)) \rightarrow 69$	$h(h(5,g(5)),5) \rightarrow 595$
	$f(h(h(5,5),5)) \rightarrow 556$	$h(f(5),h(5,5)) \rightarrow 655$	$h(h(5,h(5,5)),5) \rightarrow 5555$
	$g(h(h(5,5),5)) \rightarrow 1109$		$h(h(f(5),5),5) \rightarrow 655$
			$h(h(g(5),5),5) \rightarrow 955$
			$h(h(h(5,5),5),5) \rightarrow 5555$

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

A Computational Approach

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

$f(g(h(f(g(5)),g(f(f(5)))))) \rightarrow 2026$ has 8 calls
10 13

$f(g(f(f(h(f(g(5)),g(f(5))))))) \rightarrow 2026$ has 9 calls

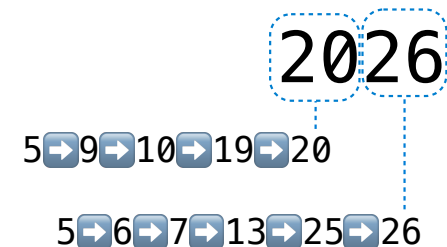
$f(g(f(h(f(g(5)),f(g(f(5))))))) \rightarrow 2026$ has 9 calls

$f(g(h(f(g(5)),f(f(g(f(5))))))) \rightarrow 2026$ has 9 calls
10 13

$f(h(f(g(f(f(h(g(5),g(5)))))),5)) \rightarrow 2026$ has 9 calls

$h(f(g(f(f(h(g(5),g(5))))),f(5)) \rightarrow 2026$ has 9 calls
202 6

$h(f(g(f(g(5))),f(g(g(f(f(5)))))) \rightarrow 2026$ has 10 calls
20 26



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

A Computational Approach

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

```
def f(x):  
    return x + 1  
def g(x):  
    return 2 * x - 1  
def h(x, y):  
    return int(str(x) + str(y))  
  
class Number:  
    def __init__(self, value):  
        self.value = value  
  
    def calls(self):  
        return 0  
  
    def __str__(self):  
        return str(self.value)  
  
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])  
  
    def calls(self):  
        return 1 + sum(o.calls() for o in self.operands)  
  
    def __str__(self):  
        return f'{self.f.__name__}({",".join([str(o) for o in self.operands])})'
```

Functions

Containers

Classes

Objects

Higher-Order
Functions

Sequences

Lazy Evaluation

```
def smalls(n):  
    if n == 0:  
        return [Number(5)]  
    else:  
        results = []  
        for operand in smalls(n-1):  
            results.append(Call(f, [operand]))  
            results.append(Call(g, [operand]))  
        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  
  
def print_smallest():  
    result = []  
    for i in range(10):  
        result.extend([e for e in smalls(i) if e.value == 2026])  
  
    for e in result:  
        print(e, '->', e.value, 'has', e.calls(), 'calls')
```

Recursion

Tree
Recursion

Control

Mutation

By Midterm 2, you can do this.