Functions

Announcements

We may have activities until Berkeley time (1:10), so enter quietly if one is in progress!

It’s OK to not know things!

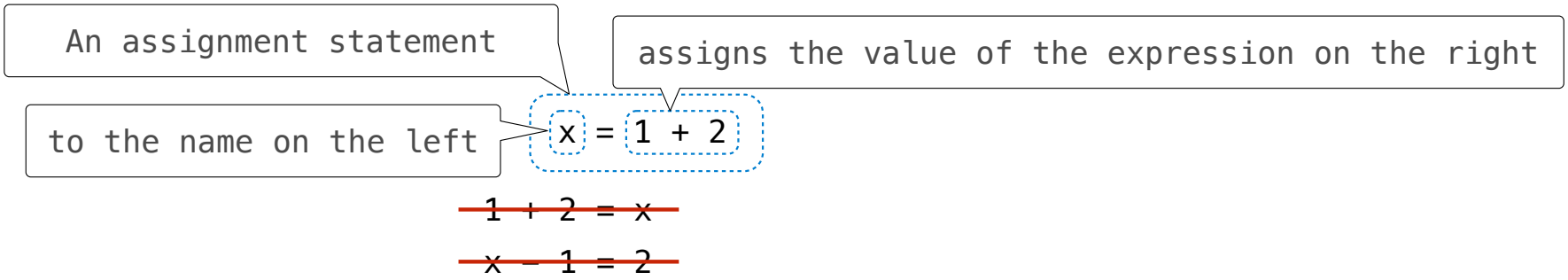
Today

Naming things!

Drawing pictures to help us understand those names

Assignment Statements

Assignment Statements

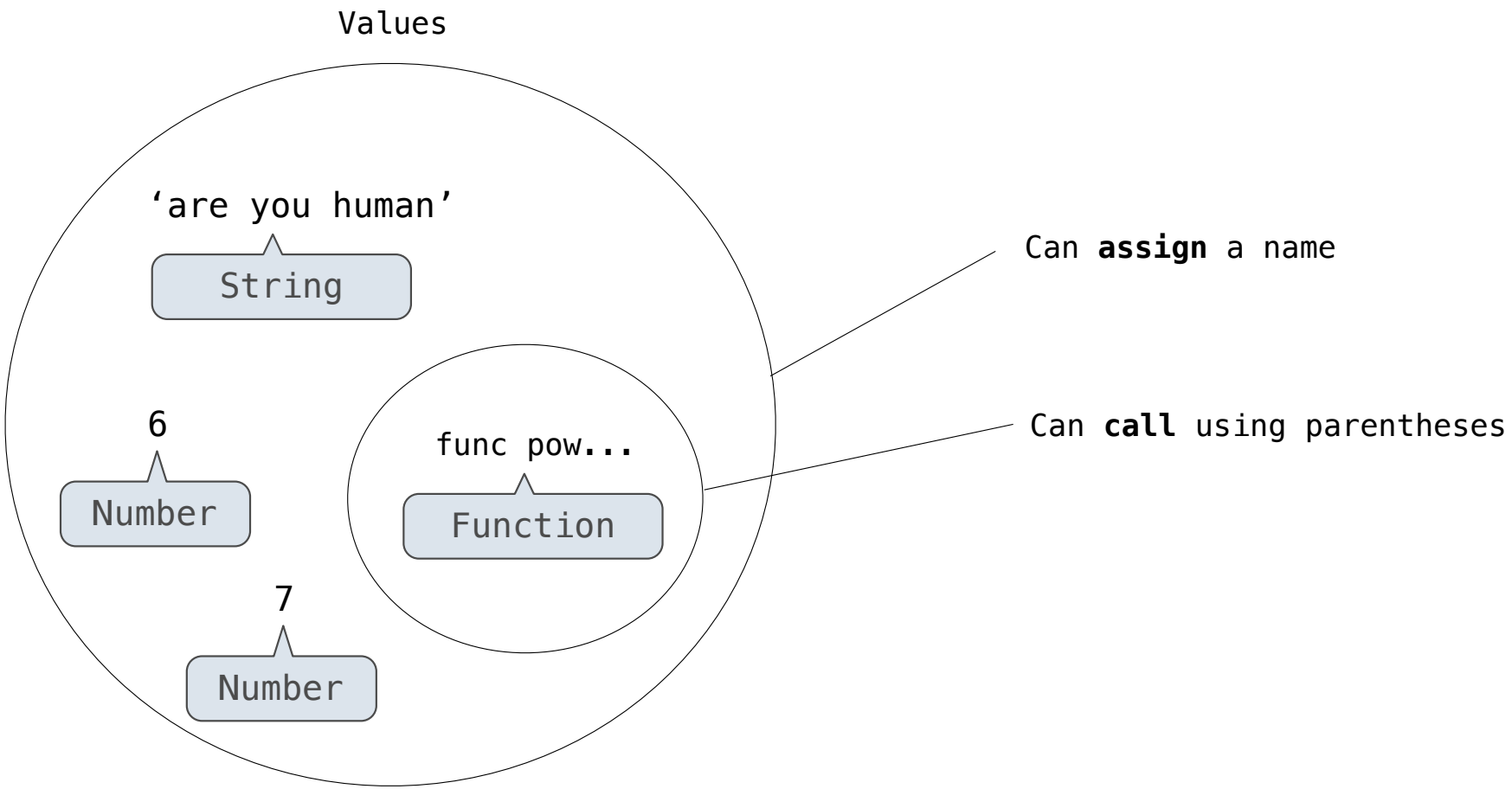


The expression (right) is evaluated, and its value is assigned to the name (left).

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

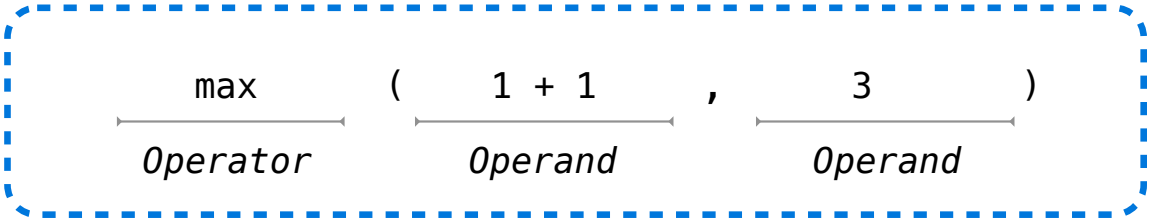
(Demo: Functions)

Functions, Values, and Calling



Lecture 1: Anatomy of a Call Expression

Call Expression



Evaluation Procedure for call expressions

(1) Evaluate operator

function

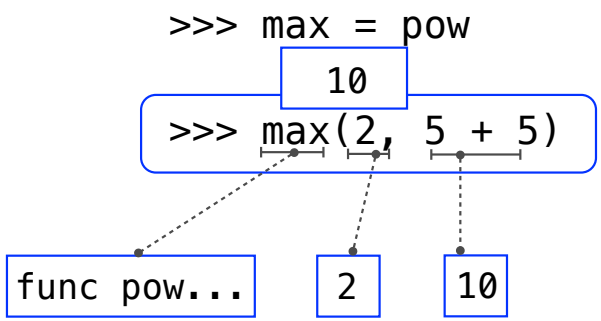
(2) Evaluate each operand

argument

(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
```



(Demo: more assigning functions)

Environment Diagrams

Environment Diagrams

(Demo: Python tutor tutor.cs61a.org, boxes and arrows)

Frames

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

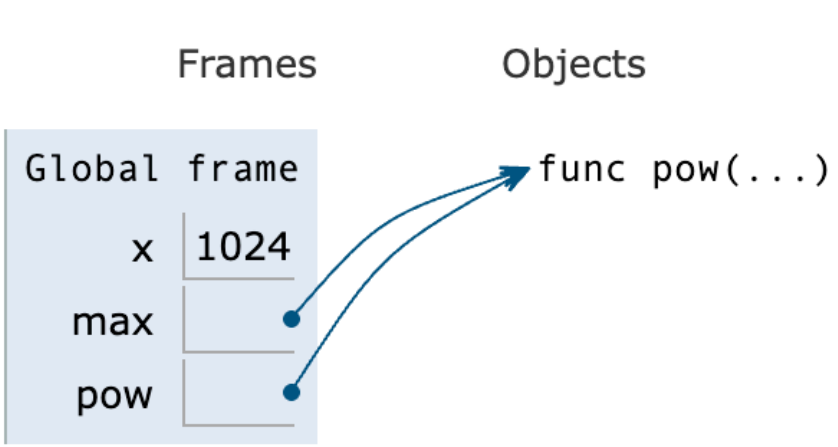
Lookup: Find the value for a name by looking in each frame of an environment

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



Today

Naming values: **Assignment**

Drawing pictures to help us understand those names: **Environment diagrams**

Naming code: **User defined functions**

User defined functions

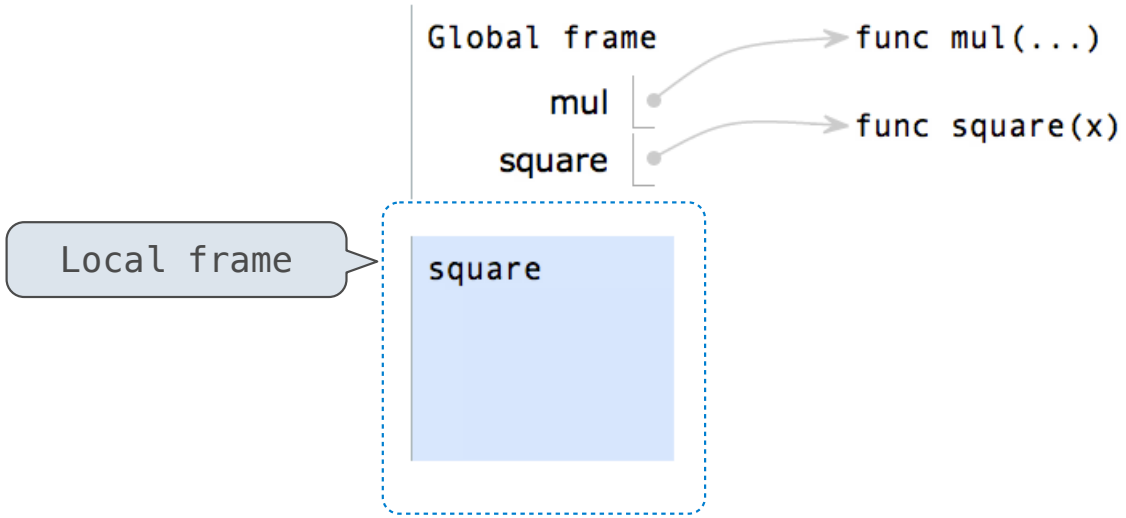
(Demo: Program in a file, defining a function, defining is not calling)

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)
```

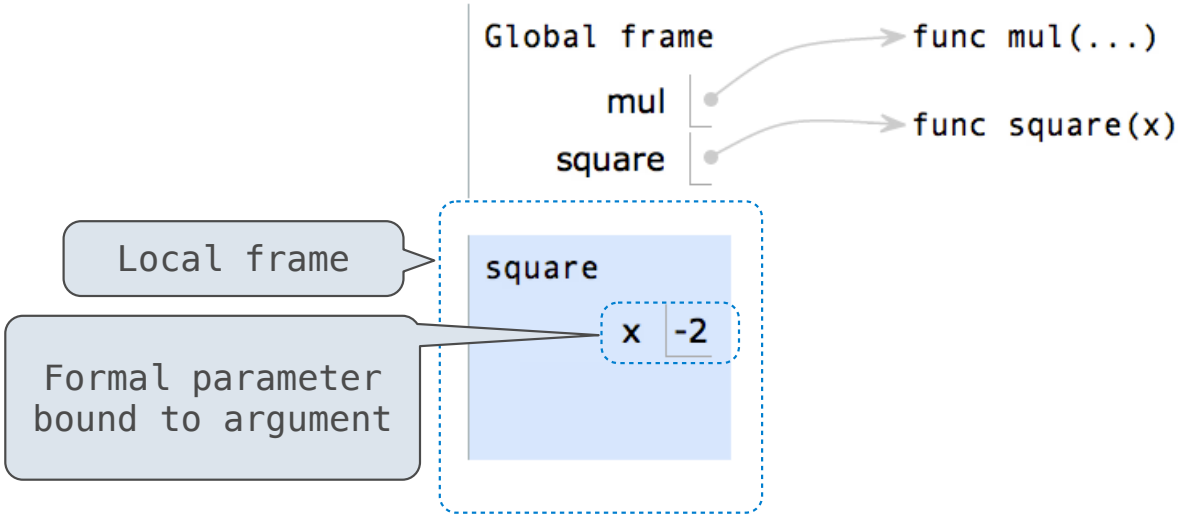


Calling User-Defined Functions

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)
```



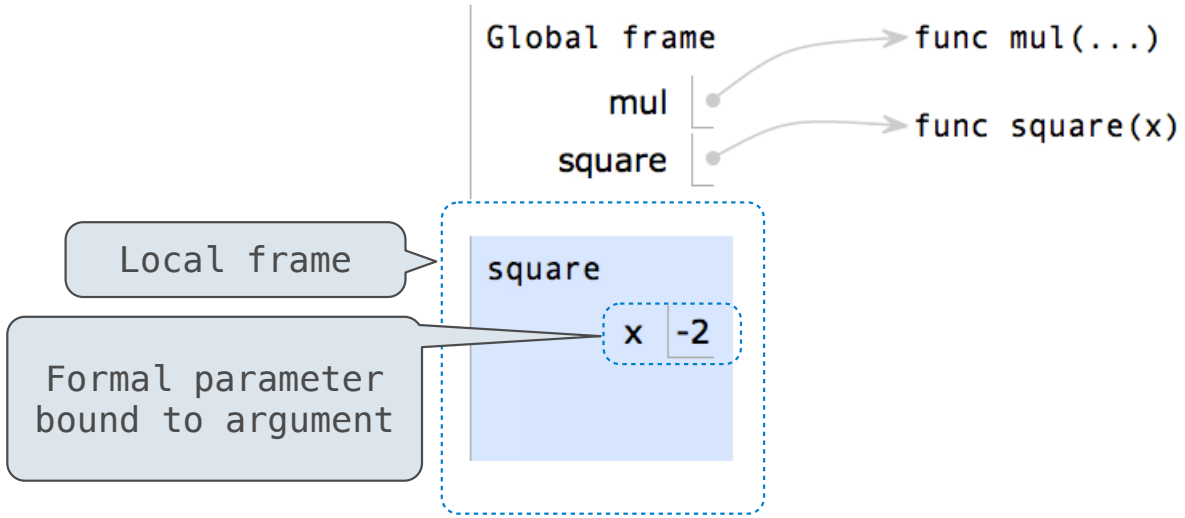
http://pythontutor.com/composingprograms.html#code=from%20operator%20import%20mul%0Adef%20square%28x%29%3A%0A%20%20%20%20return%20mul%28x,%20x%29%0Asquare%28-2%29&cumulative=true&curInstr=0&mode=display&origin=composingprograms.js&py=3&rawInputLstJSON=%5B%5D

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)
```



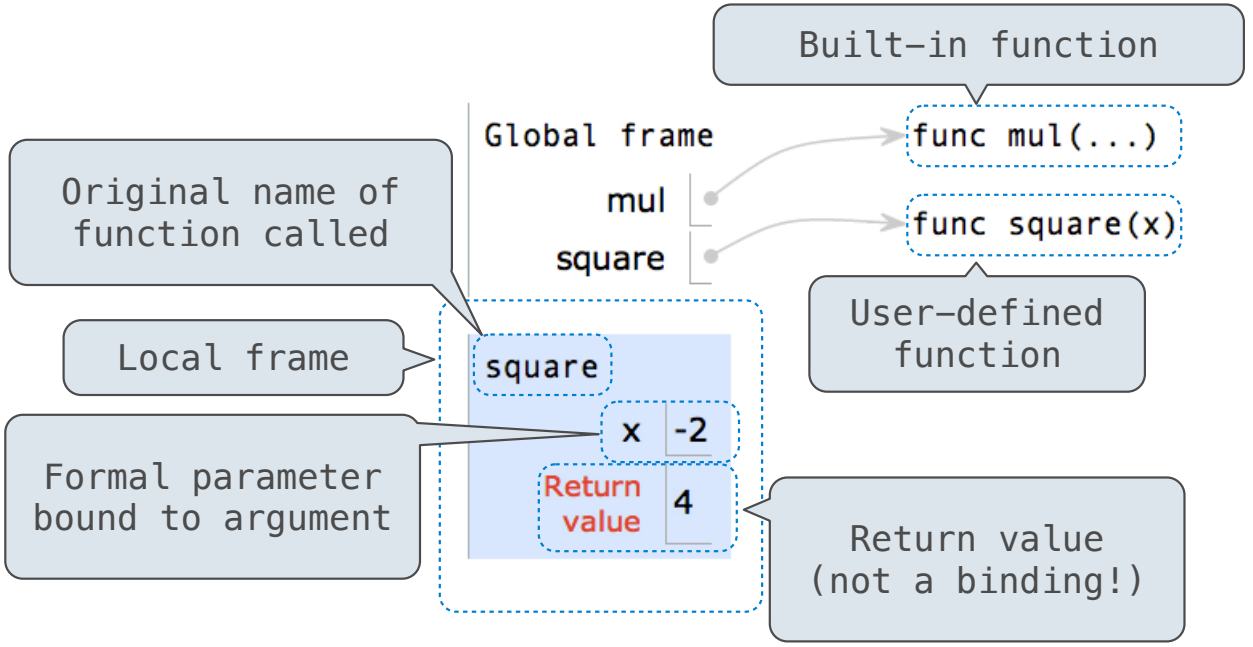
http://pythontutor.com/composingprograms.html#code=from%20operator%20import%20mul%0Adef%20square%28x%29%3A%0A%20%20%20%20return%20mul%28x,%20x%29%0Asquare%28-2%29&cumulative=true&curInstr=0&mode=display&origin=composingprograms.js&py=3&rawInputLstJSON=%5B%5D

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)
```



http://pythontutor.com/composingprograms.html#code=from%20operator%20import%20mul%0Adef%20square%28x%29%3A%0A%20%20%20%20return%20mul%28x,%20x%29%0Asquare%28-2%29&cumulative=true&curInstr=0&mode=display&origin=composingprograms.js&py=3&rawInputLstJSON=%5B%5D

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 a type of expression) such as **x** is evaluated by looking it up

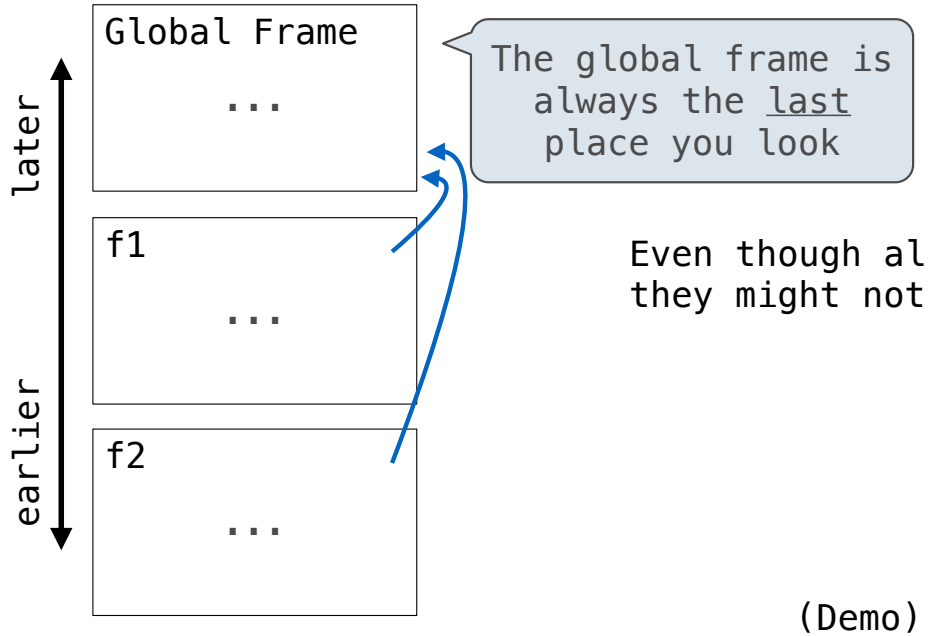
Global frame: The frame with built-in names (min, pow, etc.)

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

A Sequence of Frames

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.



Even though all three frames are in the same **diagram**, they might not be in the same **environment**

(Demo)

Frames & Environments

Why organize information this way?

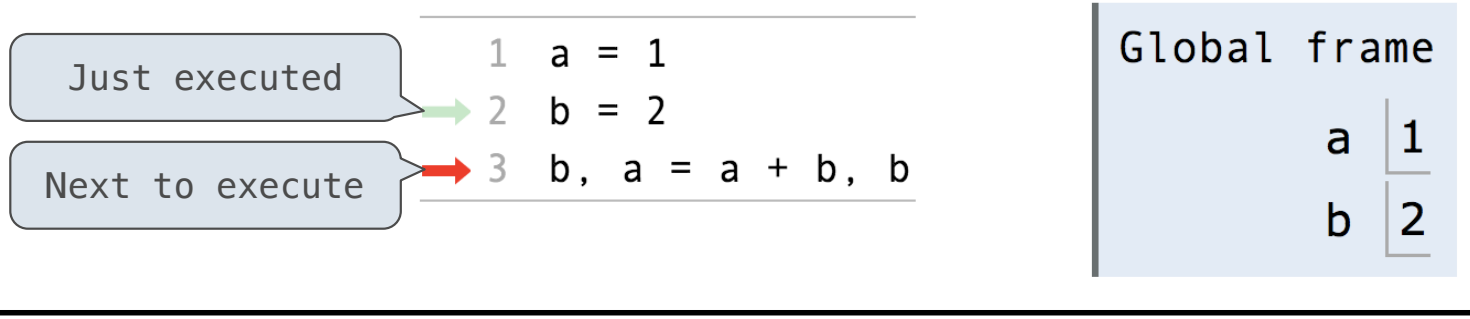
- Local context before global context
- Calling or returning changes the local context
- Assignment within a function's local frame doesn't affect other frames

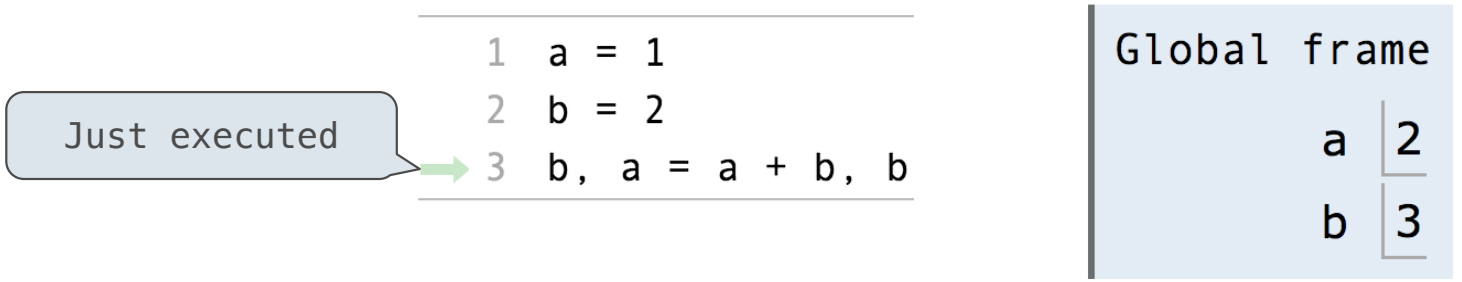
```
1 from operator import mul
2 def square(x):
3     return mul(x, x)
4 square(-2)
```

(Demo: same name in different functions; repeated names)

Multiple Assignment

Multiple Assignment





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.

(Demo)

http://pythontutor.com/composingprograms.html#code=a%20%3D%201%0Ab%20%3D%202%0Ab,%20a%20%3D%20a%20%2B%20b,%20b&cumulative=false&curInstr=0&mode=display&origin=composingprograms.js&py=3&rawInputLstJSON=%5B%5D

Today

Naming things: **Assignment, Multiple Assignment, User Defined Functions**

Drawing pictures to help us understand those names: **Environment diagrams**

Print and None

(Demo)

Small Expressions

Problem Definition

From Discussion 0:

Imagine you can call only the following three functions:

- f(x): decrement an integer x to get x-1

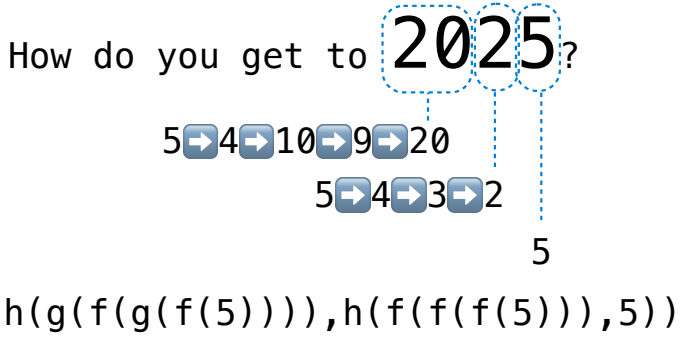
- g(x): increment then double an integer x to get 2*(x+1)

- 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.

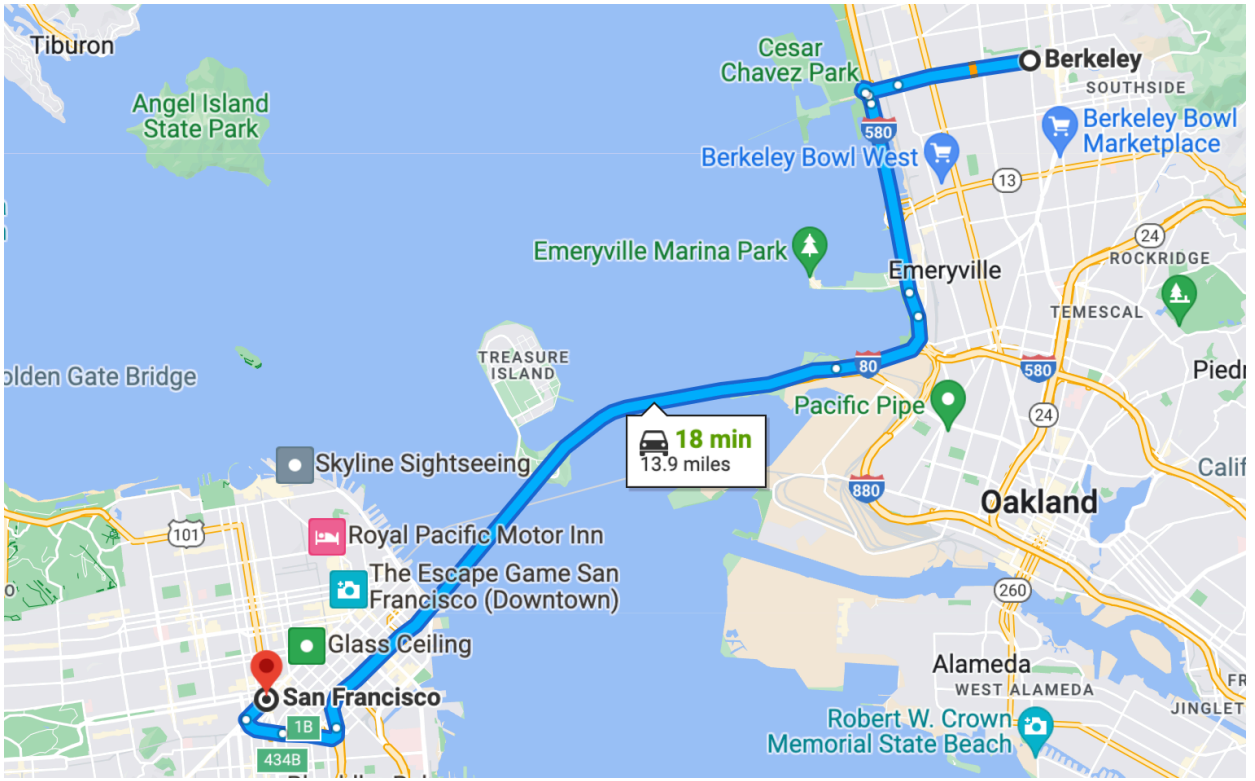
What's the shortest *small expression* you can find that evaluates to 2025?

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

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))) -> 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

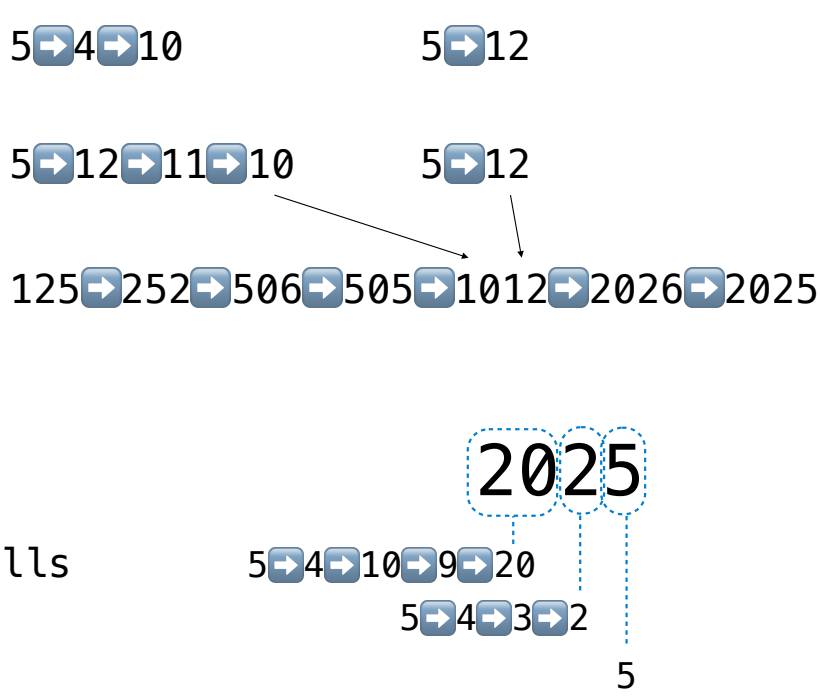
A Computational Approach

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

f(g(h(g(f(5)),g(5)))) -> 2025 has 6 calls

f(g(h(f(f(g(5))),g(5)))) -> 2025 has 7 calls

f(g(g(f(g(g(h(g(5),5)))))))) -> 2025 has 8 calls
f(g(g(h(g(g(f(g(5)))),5)))) -> 2025 has 8 calls
f(h(g(f(g(f(5)))),g(g(5)))) -> 2025 has 8 calls
h(g(f(g(f(5)))),f(g(g(5)))) -> 2025 has 8 calls
h(g(g(f(g(g(f(g(5)))))))),5) -> 2025 has 8 calls
h(g(g(h(f(5),f(g(f(5))))))),5) -> 2025 has 8 calls
h(g(f(g(f(5)))),h(f(f(f(5))),5)) -> 2025 has 9 calls



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.

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

    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])

    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)
```

Functions

Containers

Objects

Representation

Sequences

Iterators

```
def smalls(n):
    if n == 0:
        yield Number(5)
    else:
        for operand in smalls(n-1):
            yield Call(f, [operand])
            yield 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:
                        yield Call(h, [first, second])

result = []
for i in range(9):
    result.extend([e for e in smalls(i) if e.value == 2025])
```

Generators

Recursion

Tree Recursion

Control

Mutability

Higher-Order Functions

By Midterm 2, you can do this.