Environments

Announcements

- Hog, HW1, and Lab 1 have been released!
  - Lab 1 is due tomorrow
  - HW 1 is due Thursday
  - Hog Checkpoint is due Friday

- Tutoring section sign ups released!
  - tutorials.cs61a.org

- Regular OH this week!
  - Calendar: https://cs61a.org/office-hours/

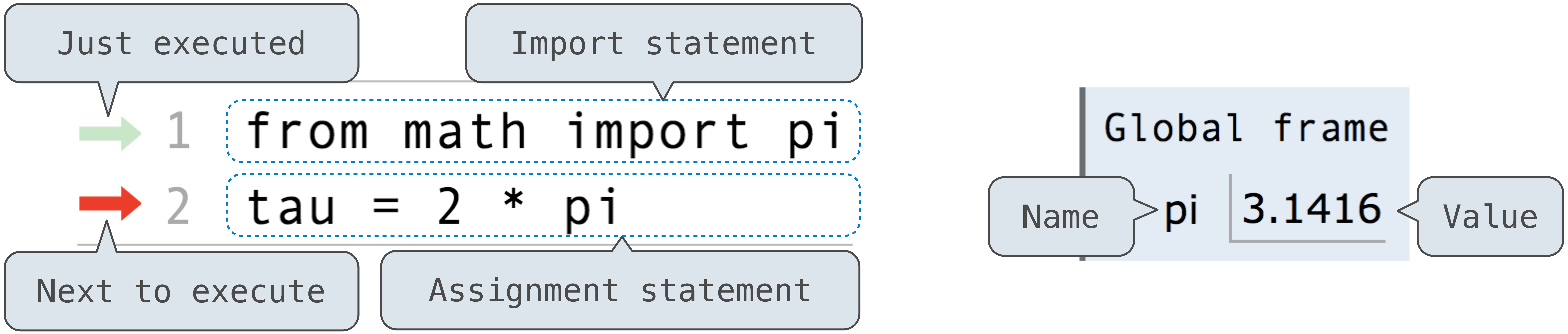
- Instructor OH Schedule in Soda 781
  - Jordan: Mondays, 12:45 - 1:45 pm
  - Noor: Tuesdays, 9:30 - 10:30 am
  - Tim: Thursdays, 12:45 - 1:45 pm

- Sections will be finalized 6/30
  - sections.cs61a.org

Environment Diagrams

Environment Diagrams

Environment diagrams visualize the interpreter's process.



Just executed

Import statement

```
from math import pi
tau = 2 * pi
```

Next to execute

Assignment statement

Global frame

Name

pi 3.1416

Value

Code (left):

Statements and expressions

Arrows indicate evaluation order

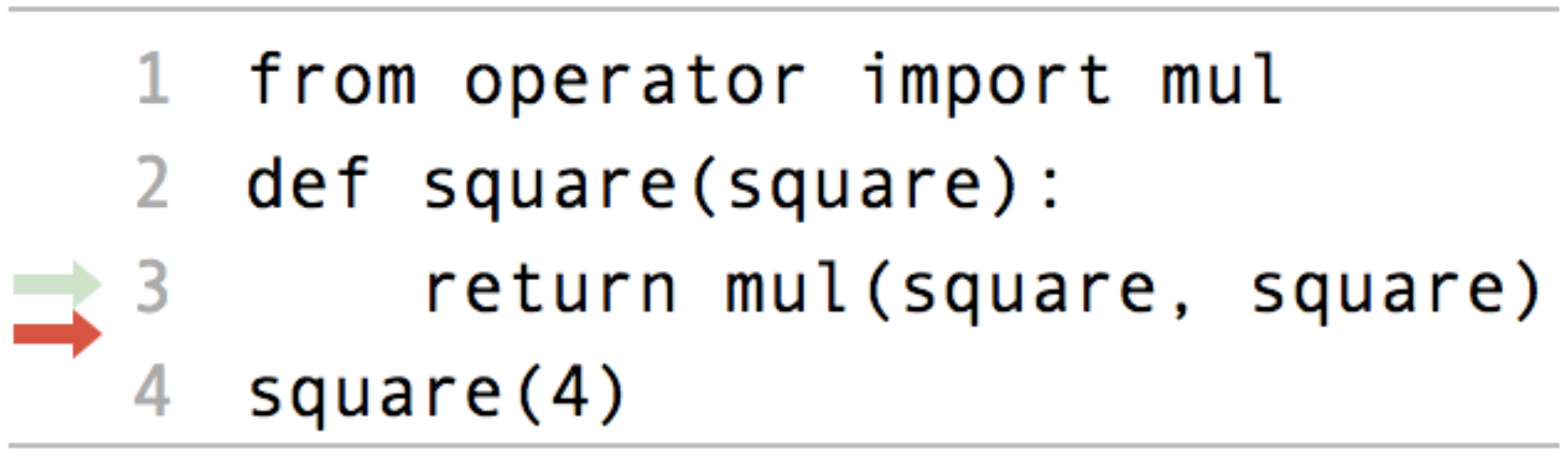
Frames (right):

Each name is bound to a value

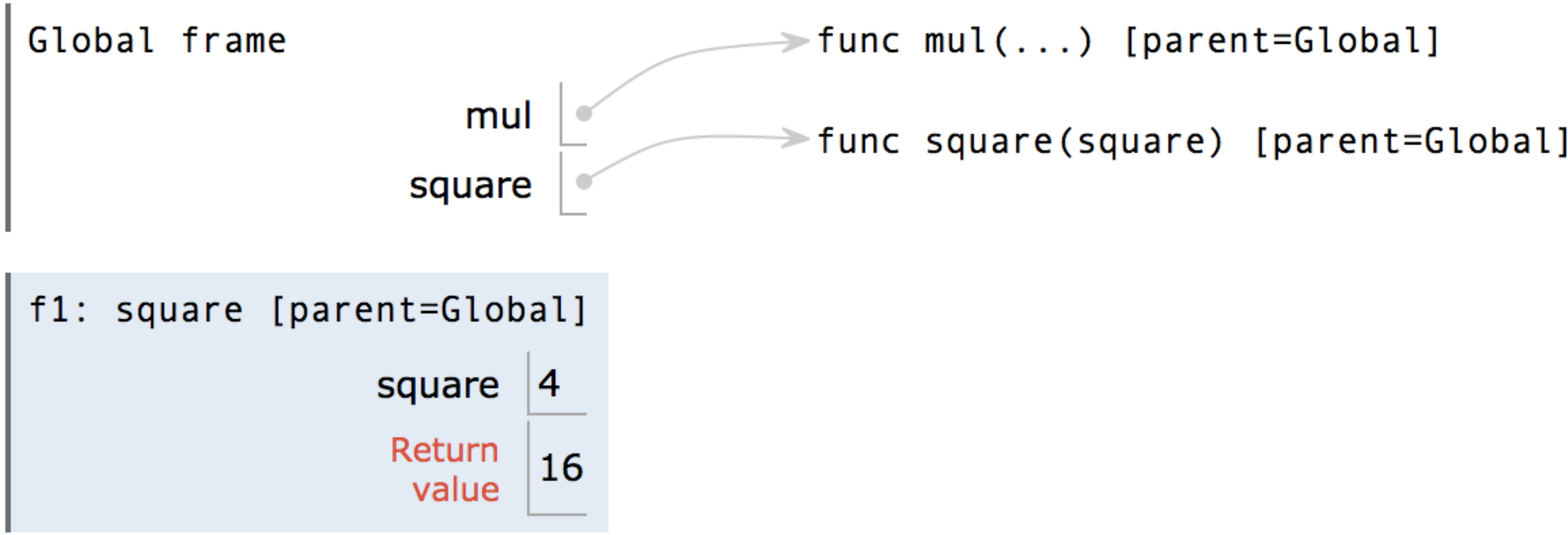
Within a frame, a name cannot be repeated

Why Use Environment Diagrams?

- They help us understand why the programs we design work the way they do!
  - Predict how a program will behave



```
from operator import mul
def square(square):
    return mul(square, square)
square(4)
```



Global frame

mul → func mul(...) [parent=Global]

square → func square(square) [parent=Global]

f1: square [parent=Global]

square 4

Return value 16

- They can also be useful in debugging!
  - When we run into an unexpected error, we can trace back our steps!

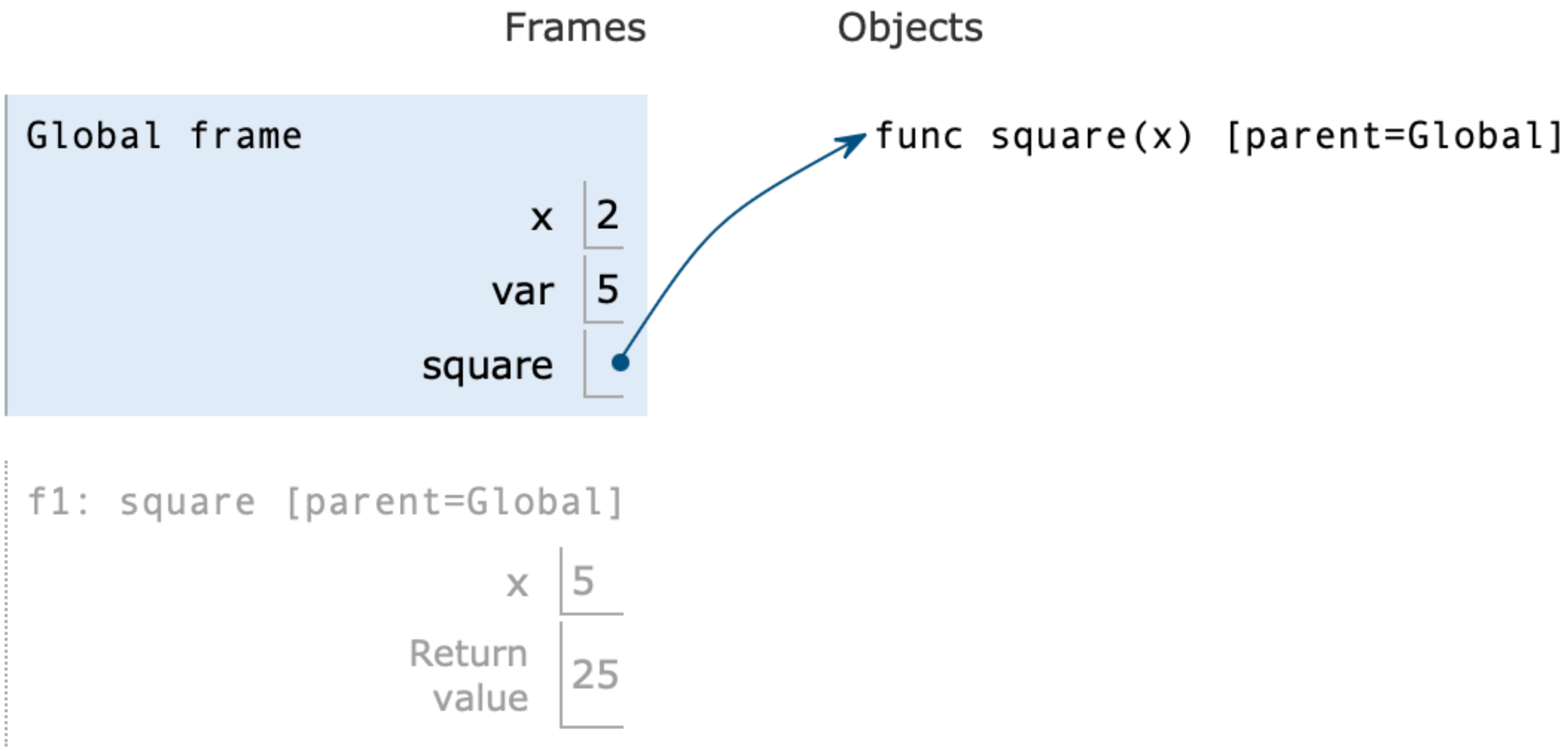
staring at lines of codes

diagramming code

What We Have Seen So Far

- **Assignment Statements**
  - `x = 2`
  - `var = 5`
- **Def Statements**

  -
    ```
    def square(x):
        return x * x
    ```
- **Call Expressions**
  - `square(var)`



Frames

Objects

Global frame

x 2

var 5

square

func square(x) [parent=Global]

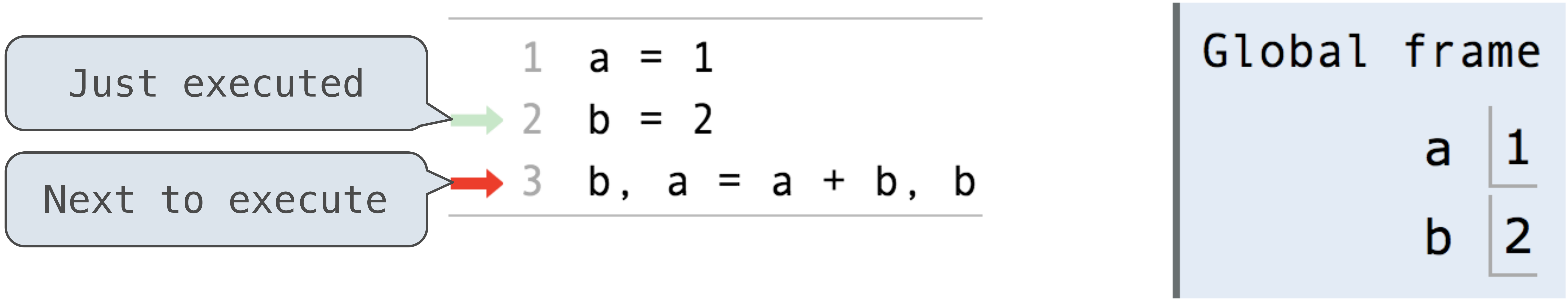
f1: square [parent=Global]

x 5

Return value 25

Environment Diagram

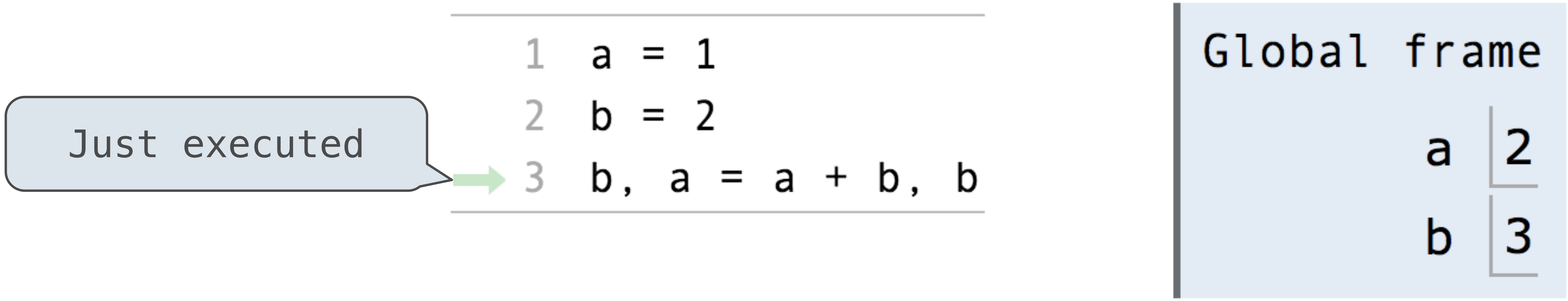
Assignment Statements



```
a = 1
b = 2
b, a = a + b, b
```

Global frame

a	1
b	2



```
a = 1
b = 2
b, a = a + b, b
```

Global frame

a	2
b	3

Execution rule for assignment statements:

1. Evaluate all expressions to the right of = from left to right.

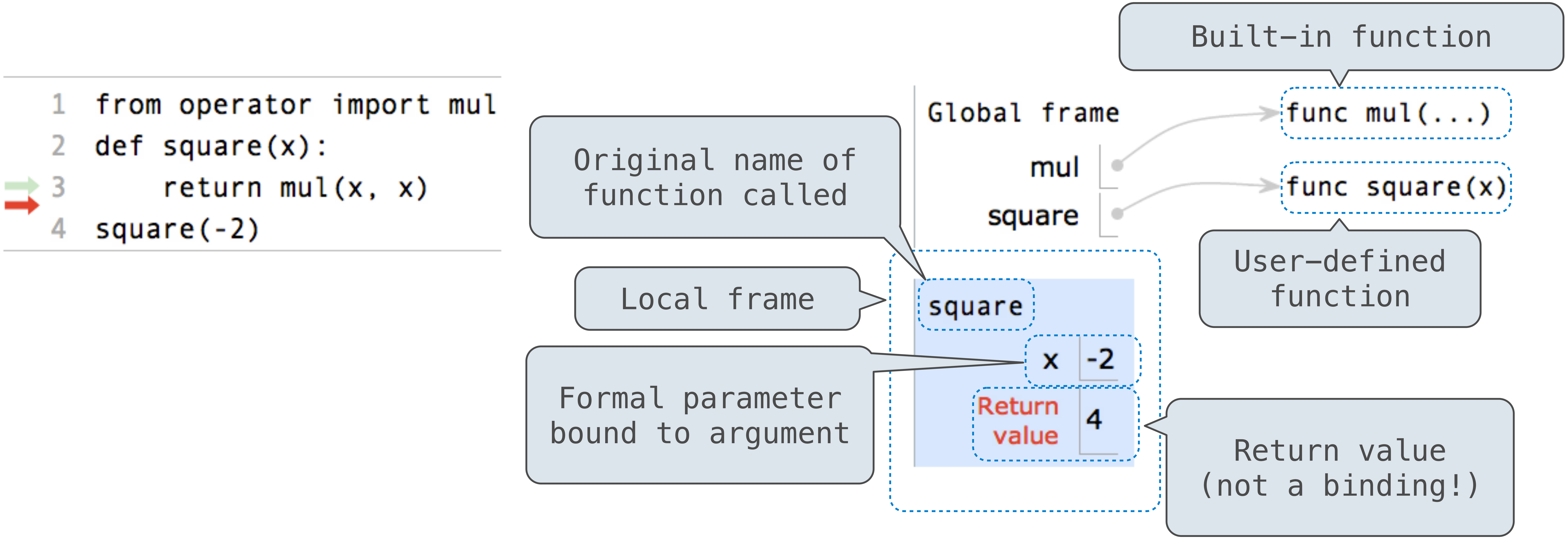
2. Bind all names to the left of = to those resulting values in the current frame.

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

Calling User-Defined Functions

Procedure for calling/applying user-defined functions:

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



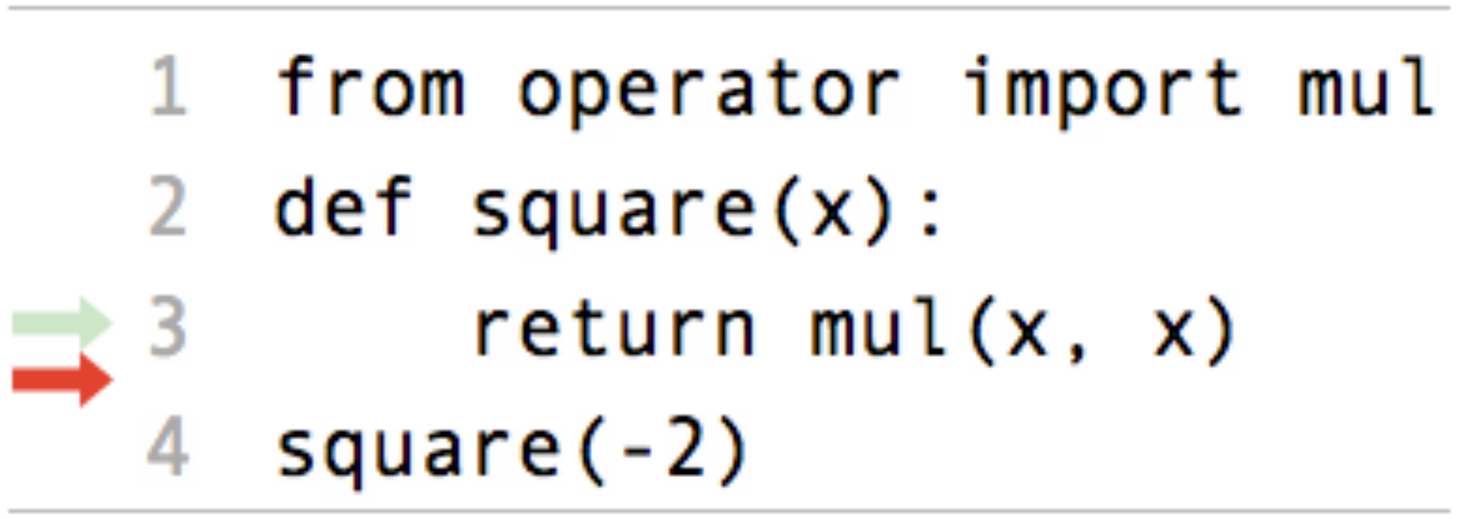
```
from operator import mul
def square(x):
    return mul(x, x)
square(-2)
```

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

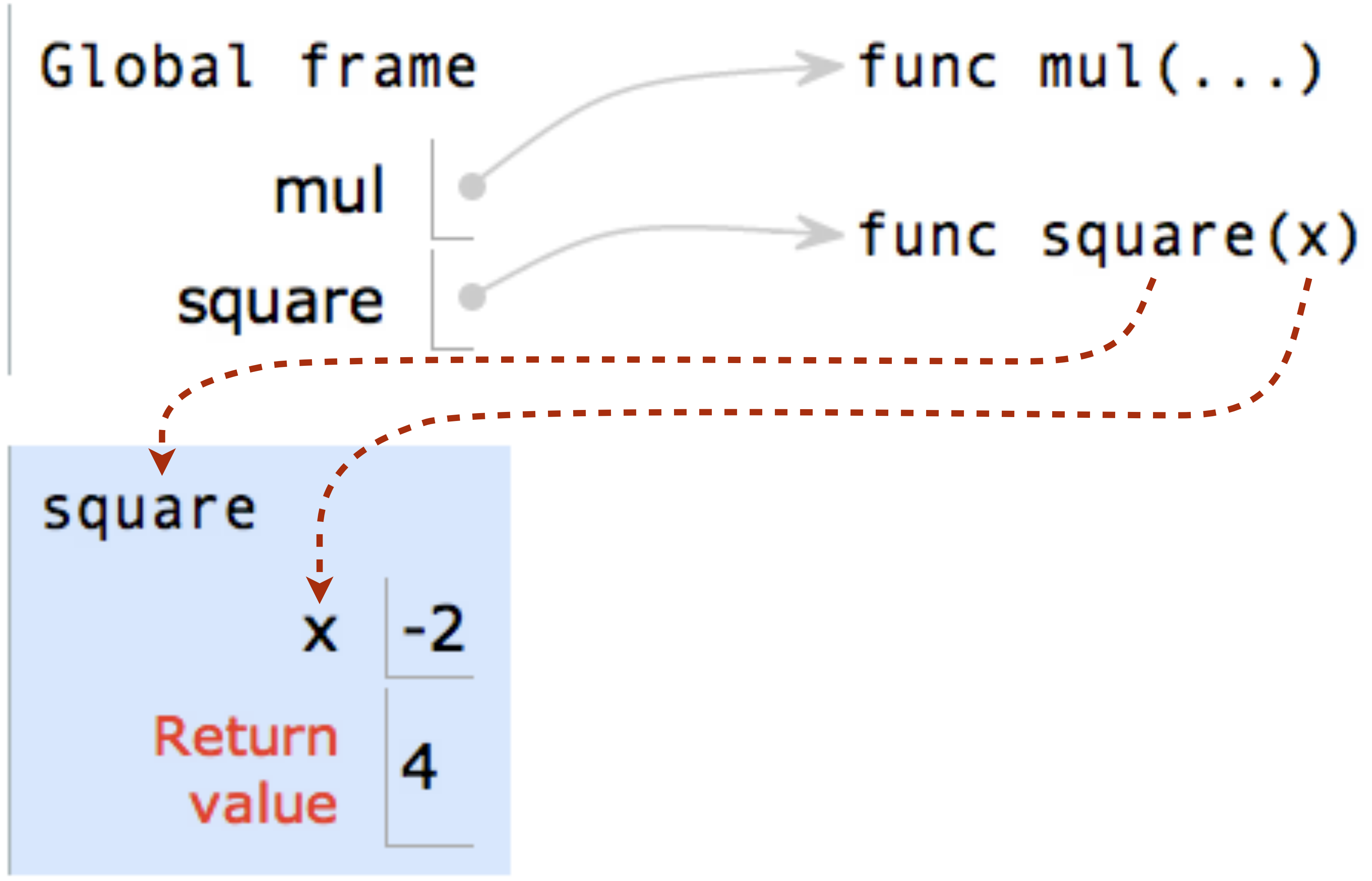
Calling User-Defined Functions

Procedure for calling/applying user-defined functions:

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



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

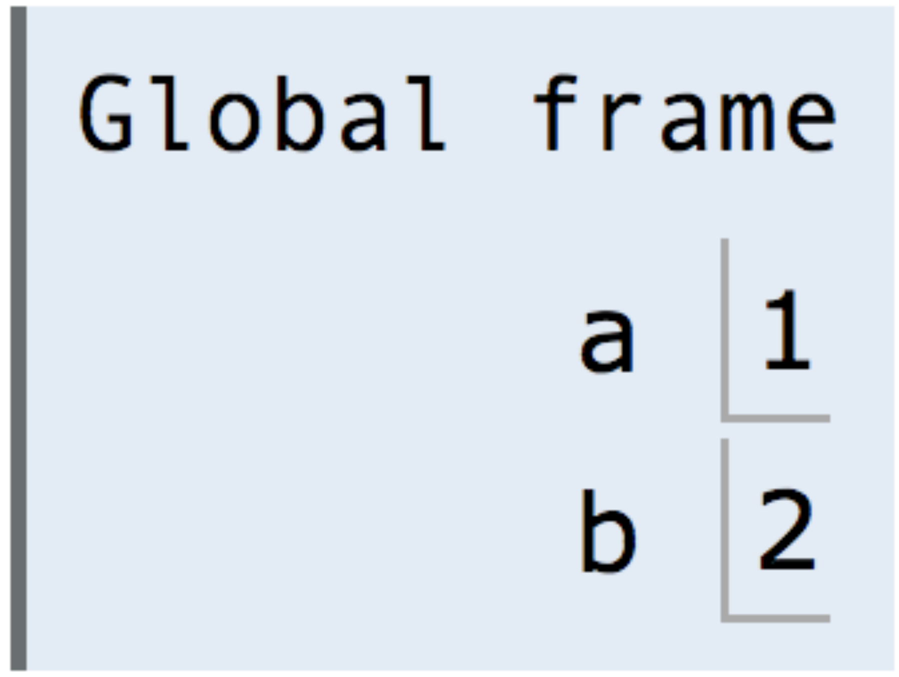


A function's signature has all the information needed to create a local frame

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

- A frame keeps track of variable-to-value bindings



Global frame

a 1

b 2

- By default, the global frame is the starting frame
  - It doesn’t correspond to a specific call expression

- Every call expression has a corresponding frame

- The parent of a function is the frame is which is was **defined** *not called*
  - *Important for variable lookup!*
  - If you cannot find a name in the current frame, you can go up to its parent until you reach the global frame
    - If it is not found, you get a **NameError: name 'x' is not defined**

Demo

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

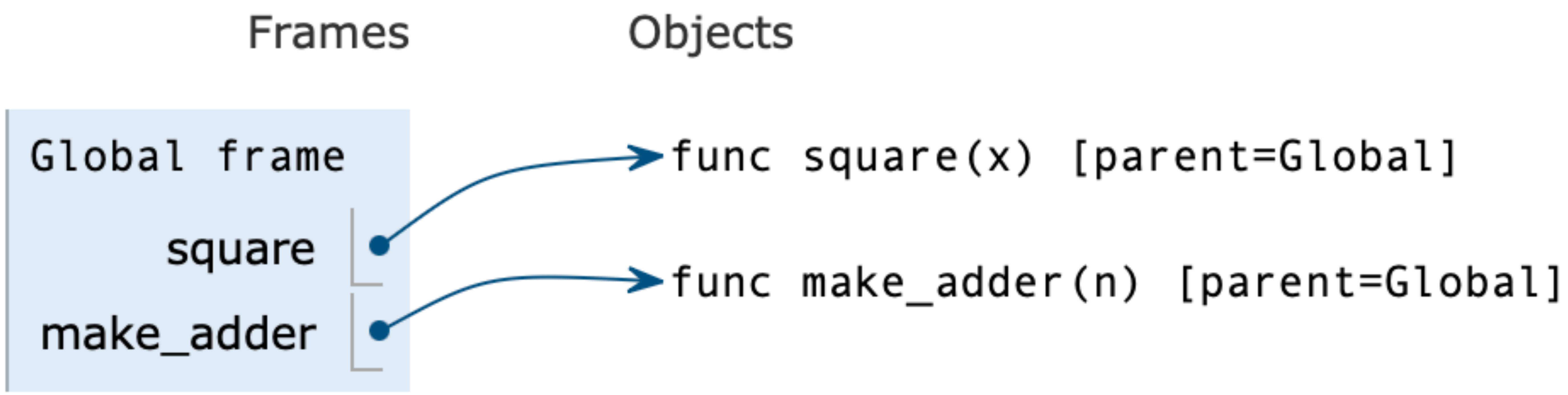
How to Draw an Environment Diagram

When a function is defined:

Create a function value: func <name>(<formal parameters>) [parent=<label>]

Its parent is the current frame.

```
def square(x):
    return x * x

def make_adder(n):
    def adder(k):
        return n + k
    return adder
```

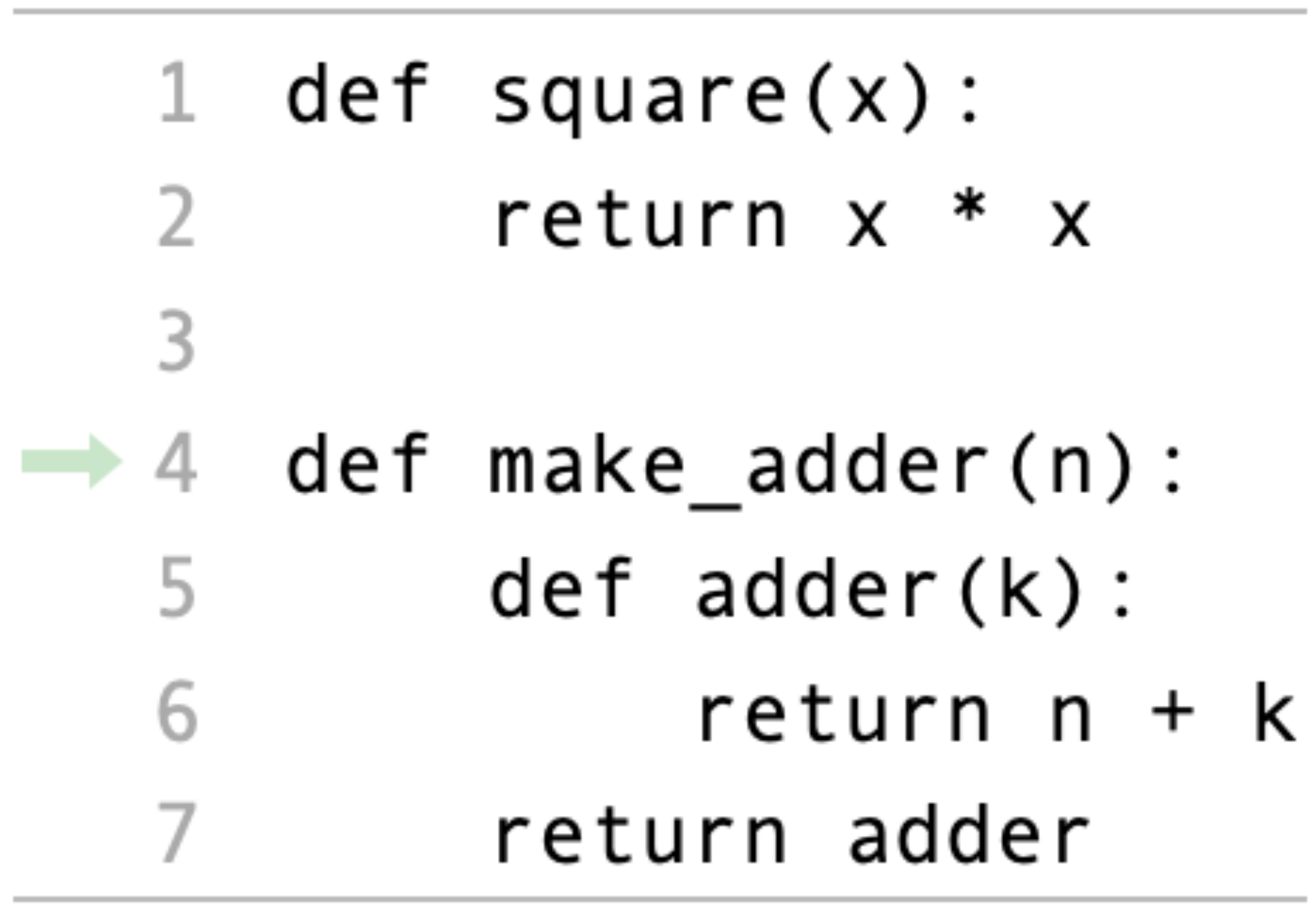


Frames
Objects
Global frame
square
make_adder
func square(x) [parent=Global]
func make_adder(n) [parent=Global]

How to Draw an Environment Diagram

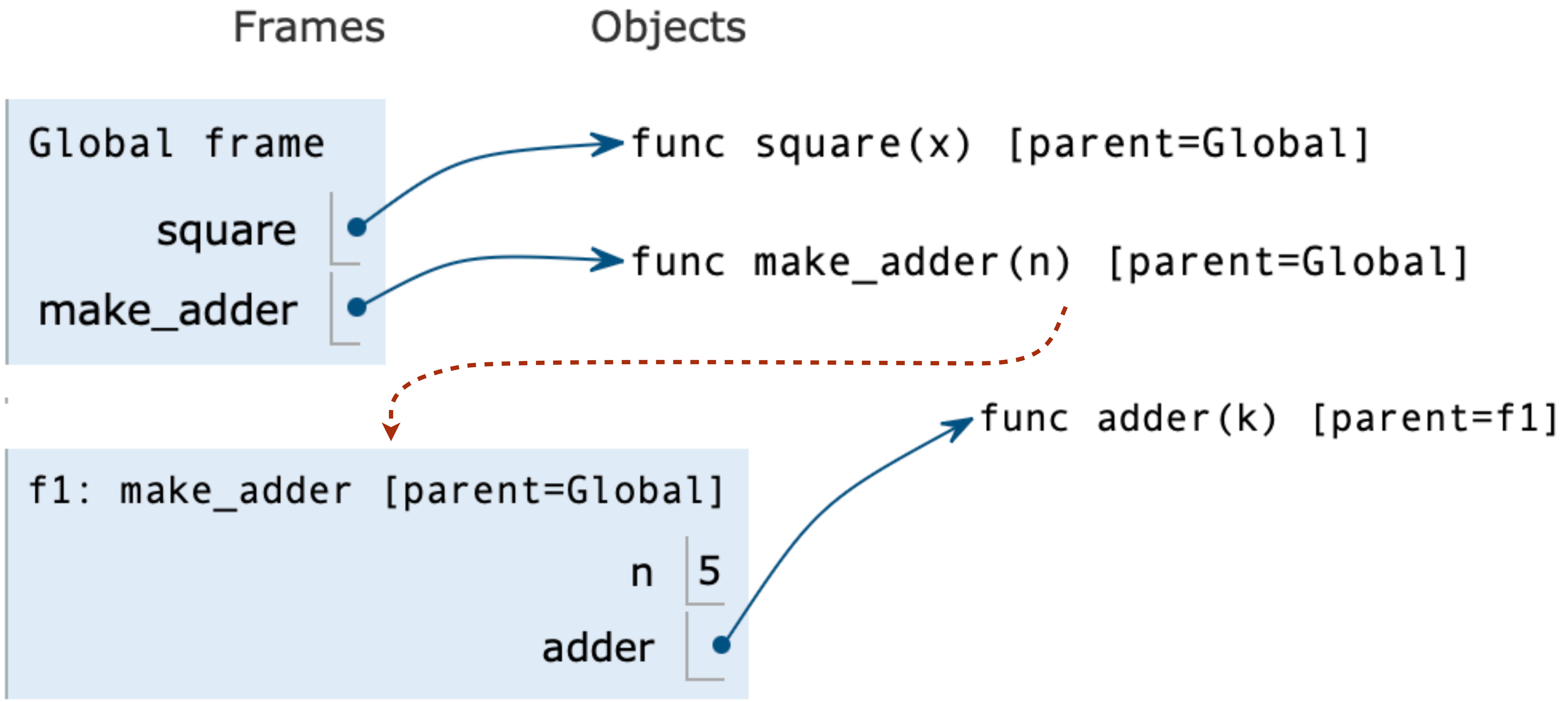
When a function is called:

1. Add a local frame, titled with the <name> of the function being called.
2. Copy the parent of the function to the local frame: [parent=<label>]
3. Bind the <formal parameters> to the arguments in the local frame.
4. Execute the body of the function in the environment that starts with the local frame.

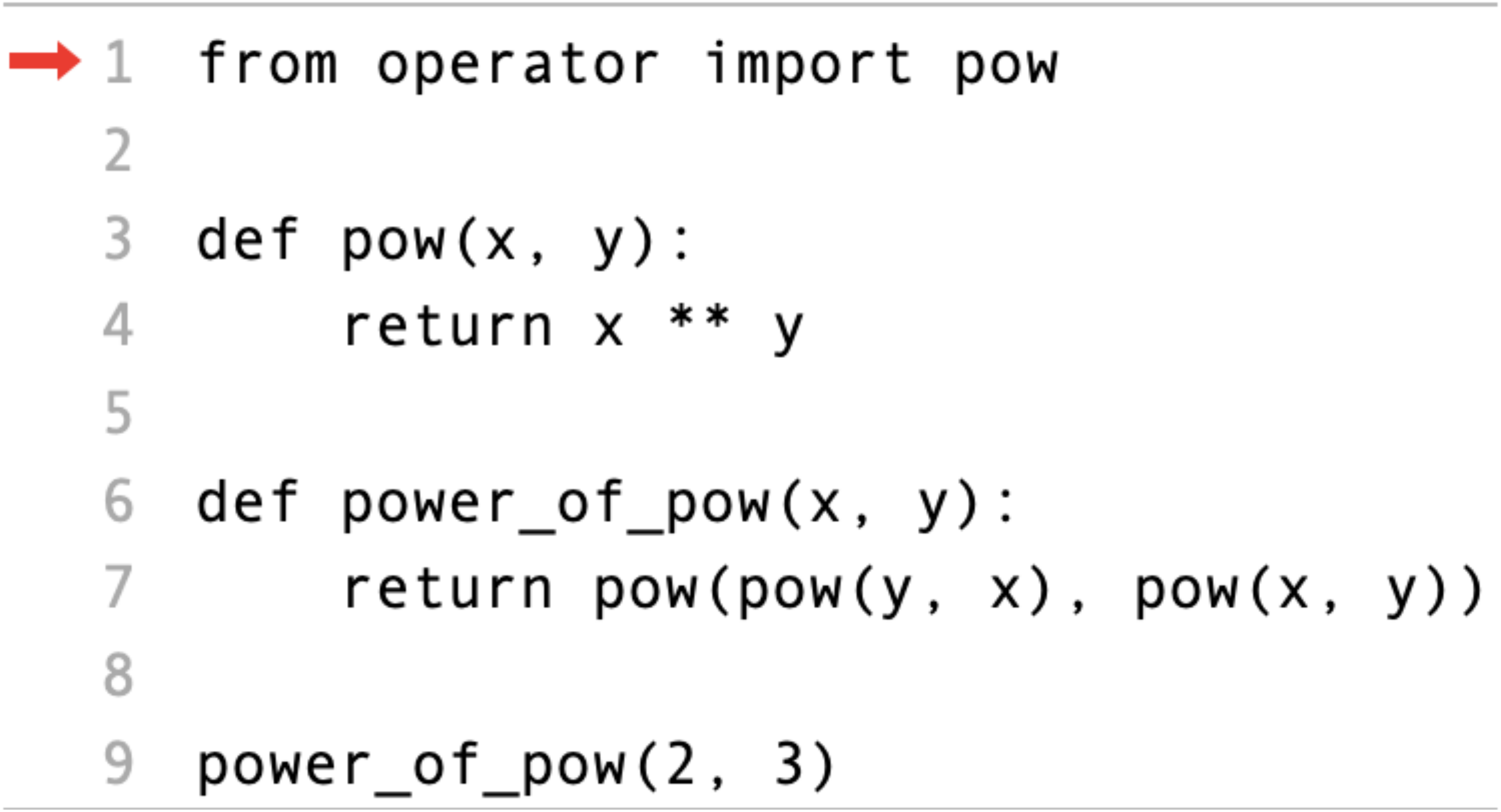


```
def square(x):
    return x * x

def make_adder(n):
    def adder(k):
        return n + k
    return adder
```

make_adder(5)


Frames
Objects
Global frame
square
make_adder
func square(x) [parent=Global]
func make_adder(n) [parent=Global]
func adder(k) [parent=f1]
f1: make_adder [parent=Global]
n
5
adder

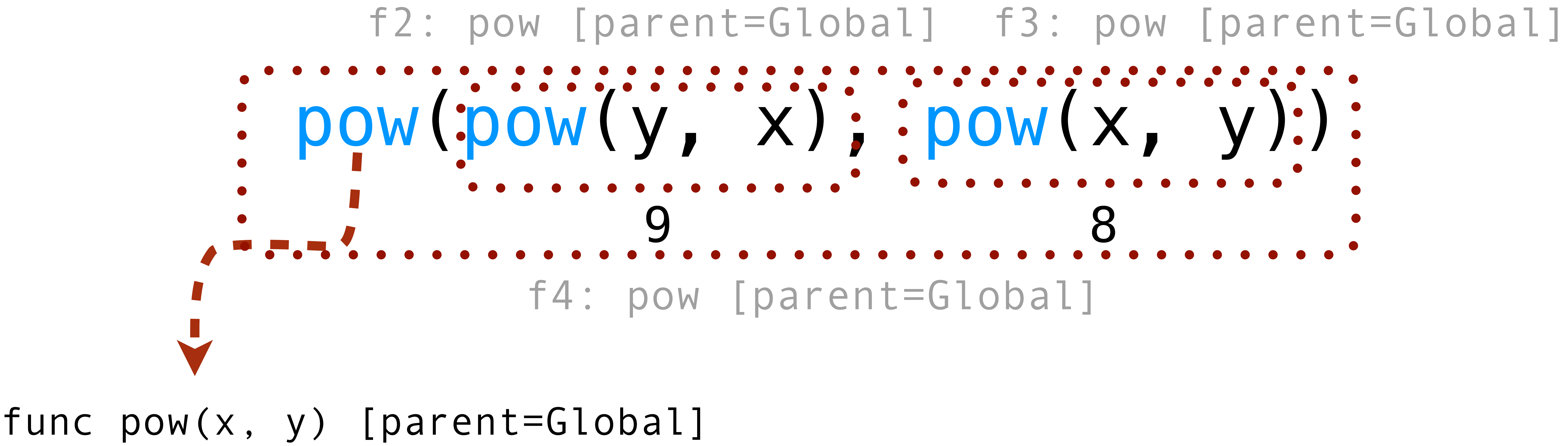
Check Your Understanding: Calling Functions



```python
from operator import pow

def pow(x, y):
    return x ** y

def power_of_pow(x, y):
    return pow(pow(y, x), pow(x, y))

power_of_pow(2, 3)
```

Evaluation Order

- An environment diagram reflects Python evaluation order
  - Evaluate the operator, then the operands, finally apply the operator to the operands



Lambda Expressions

Lambda Expressions

```
>>> x = 10
```

An expression: this one evaluates to a number

```
>>> square = x * x
```

Also an expression: evaluates to a function

```
>>> square = lambda x: x * x
```

Important: No "return" keyword!

A function

with formal parameter x

that returns the value of "x * x"

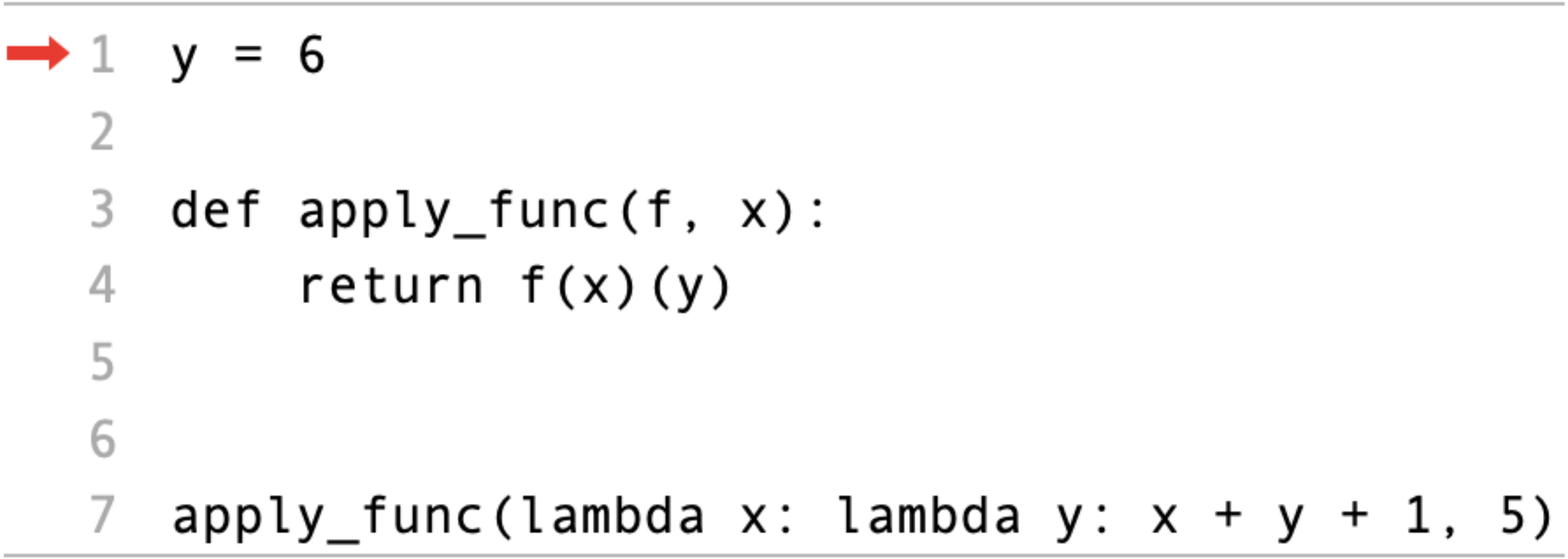
Must be a single expression

```
>>> square(4)
16
```

Lambda expressions are not common in Python, but important in general

Lambda expressions in Python cannot contain statements at all!

Check Your Understanding: Calling Lambda



```python
y = 6

def apply_func(f, x):
    return f(x)(y)


apply_func(lambda x: lambda y: x + y + 1, 5)
```

Environments for Higher-Order Functions

Environments Enable Higher-Order Functions

Functions are first-class: Functions are values in our programming language

Higher-order function: A function that takes a function as an argument value **or**
A function that returns a function as a return value

Environment diagrams describe how higher-order functions work!

Revisiting Evaluation Order

- Even with higher-order function, the rules remain the same and the environment diagram reflects Python evaluation order!
  - Evaluate the operator, then the operands, finally apply the operator to the operands

f2: adder [parent=f1]

f1: make_adder [parent=Global]

```
def make_adder(n):
    def adder(k):
        return n + k
    return adder


make_adder(3)(5)
```

func adder(k) [p=f1](5)

8

func make_adder(n) [parent=Global]

func adder(k) [parent=f1]

(Demo)

Currying

Function Currying

```
def make_adder(n):
    return lambda k: n + k
```

```
>>> make_adder(2)(3)
5
>>> add(2, 3)
5
```

There's a general relationship between these functions

(Demo)

Curry: Transform a multi-argument function into a single-argument, higher-order function

Summary

- Using **environment diagrams** to visualize and understand programming
  - Diagramming follow the evaluation procedure for Python
  - Think deeply about how the code you write actually works

- **Lambda expressions**
  - Similar to user-defined functions but are anonymous
  - They are simple and can be created for one-time use or stored by assigning it to a variable

- The same rules of diagramming apply to **HOFs**, which take in a function as an input to return a function as an output

- To **curry** a multi-argument function is to transform it into a single-argument, multi-nested HOF