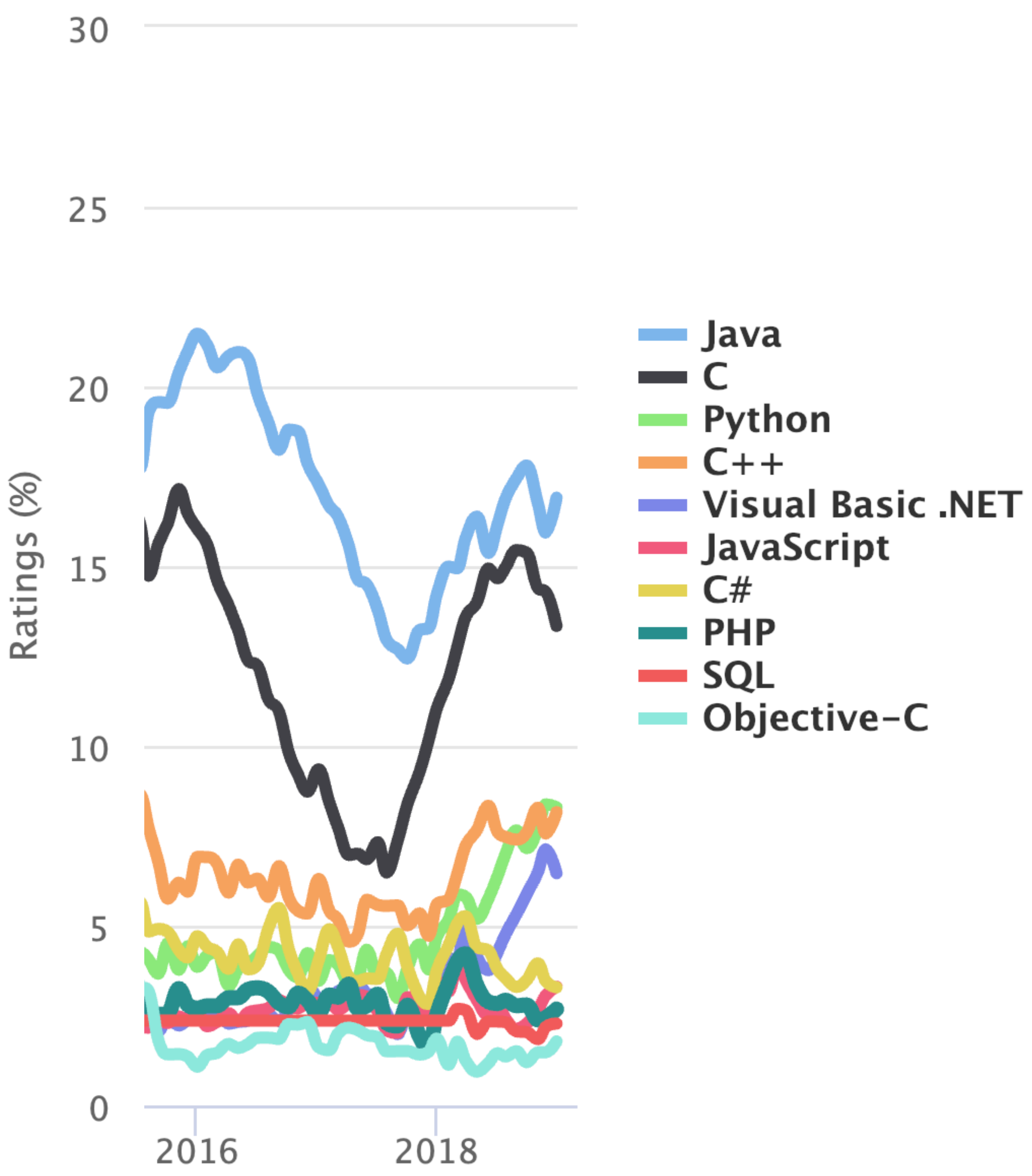
# CS 7 : Introduction to Programming and Computer Science

## Computing in the news

**Python is TIOBE's programming language of the year 2018!**

`www.tiobe.com/tiobe-index`

“The Python programming language has won the title "programming language of the year"! Python has received this title because it has gained most ranking points in 2018 if compared to all other languages. The Python language has won 3.62%, followed by Visual Basic .NET and Java. Python has now definitely become part of the big programming languages. For almost 20 years, C, C++ and Java are consistently in the top 3, far ahead of the rest of the pack. Python is joining these 3 languages now. It is the most frequently taught first language at universities nowadays, it is number one in the statistical domain, number one in AI programming, number one in scripting and number one in writing system tests. Besides this, Python is also leading in web programming and scientific computing (just to name some other domains).” In summary, Python is everywhere.



# Acknowledgements

This material is an adaptation from CS61A material at UC Berkeley.

Credits to Professor John DeNero and the entire CS61A staff.

# Parts of the Course

**Lecture:** Lecture is on Mon and Tues

**Lab section:** The most important part of this course

**Staff office hours:** The most important part of this course

**Online textbook:** [http://composingprograms.com](http://composingprograms.com)

**Optional Discussion section:** The most important part of this course

Weekly lab, homework assignments, three programming projects (hopefully)

Lots of optional special events to help you complete all this work

**Everything is posted to [erickhumalo.com/cs7](erickhumalo.com/cs7)**

# An Introduction to Programming & Computer Science

# What is Computer Science?

The study of

- What problems can be solved using computation,
- How to solve those problems, and
- What techniques lead to effective solutions

Creativity!

Systems

Artificial Intelligence

- Decision Making
- Robotics
- Machine Learning
  - Training Models
  - Classification
  - ...
- ...

Graphics

Security

Networking

Programming Languages

Theory

Scientific Computing

...

# What is This Course About?

- A course about managing complexity
  - Mastering abstraction
  - Programming paradigms
- An introduction to programming
  - Full understanding of Python fundamentals
  - Combining multiple ideas in large projects
  - How computers interpret programming languages
- A challenging course that will demand a lot of you

python

**Hard**

λ

**Fun**

**Worth it**

# Course Policies

# Course Policies

## Uncool

- You don’t know that? Sheesh! (rolls eyes)
- Elitism
- “Me first” attitude
- Making students feel unwelcome

## Learning

## Community

## Cool

- You having trouble? Here, let me help!
- Supporting each other
- “We together” attitude
- Making students feel welcome. We are a CS7 family!

Details...

http://erickhumalo.com/cs7/about.html

# Collaboration

**Asking questions is highly encouraged**

- Discuss everything with each other; learn from your fellow students!
- Some projects can be completed with a partner
- Choose a partner from your discussion section

**The limits of collaboration**

- One simple rule: Don’t share your code, except with your project partner
- Copying project solutions causes people to fail the course

**Build good habits now**

# Announcements

- “Optional” Discussion this week
- Lab this week for setting up your workspace
- Visit the course website and browse through

# Expressions

# Types of expressions

`An expression describes a computation and evaluates to a value`

$18 + 69$

$\frac{6}{23}$

$\sin \pi$

$\log_2 1024$

$2^{100}$

$\boxed{f(x)}$

$\sqrt{3493161}$

$7 \bmod 2$

$\sum_{i=1}^{100} i$

$\lim_{x \to \infty} \frac{1}{x}$

$|-1869|$

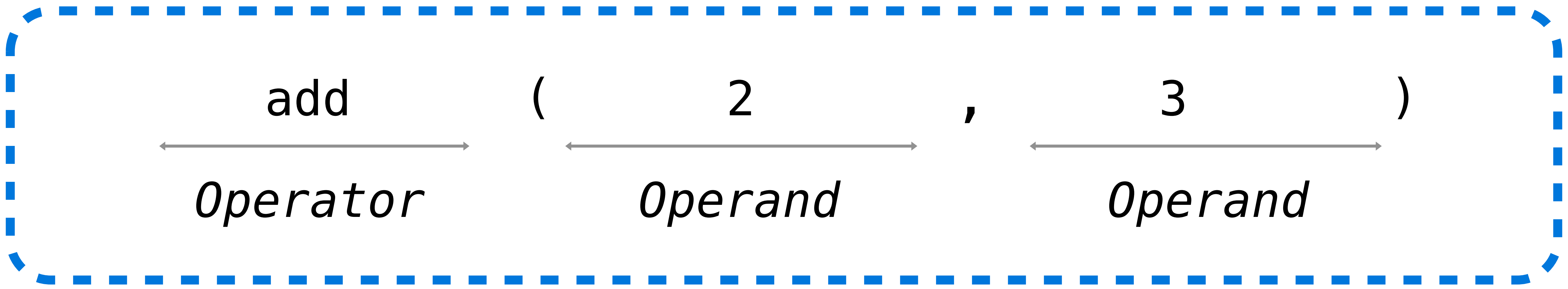
$\binom{69}{18}$

# Call Expressions in Python

**All expressions can use function call notation**

(Demo 1)

# Anatomy of a Call Expression



add ( 2 , 3 )

*Operator* *Operand* *Operand*

Operators and operands are also expressions

So they evaluate to values

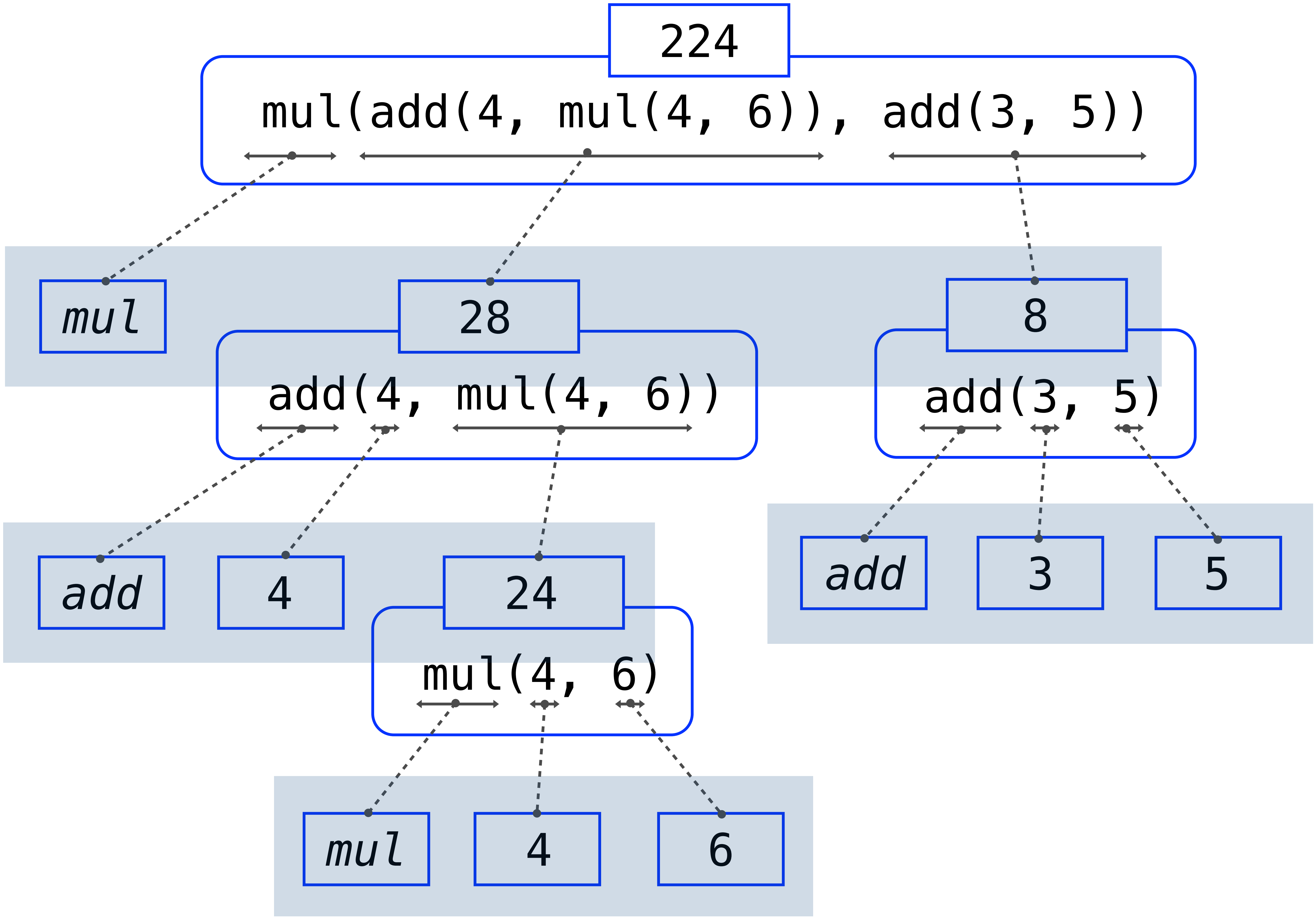
**Evaluation procedure for call expressions:**

1. Evaluate the operator and then the operand subexpressions

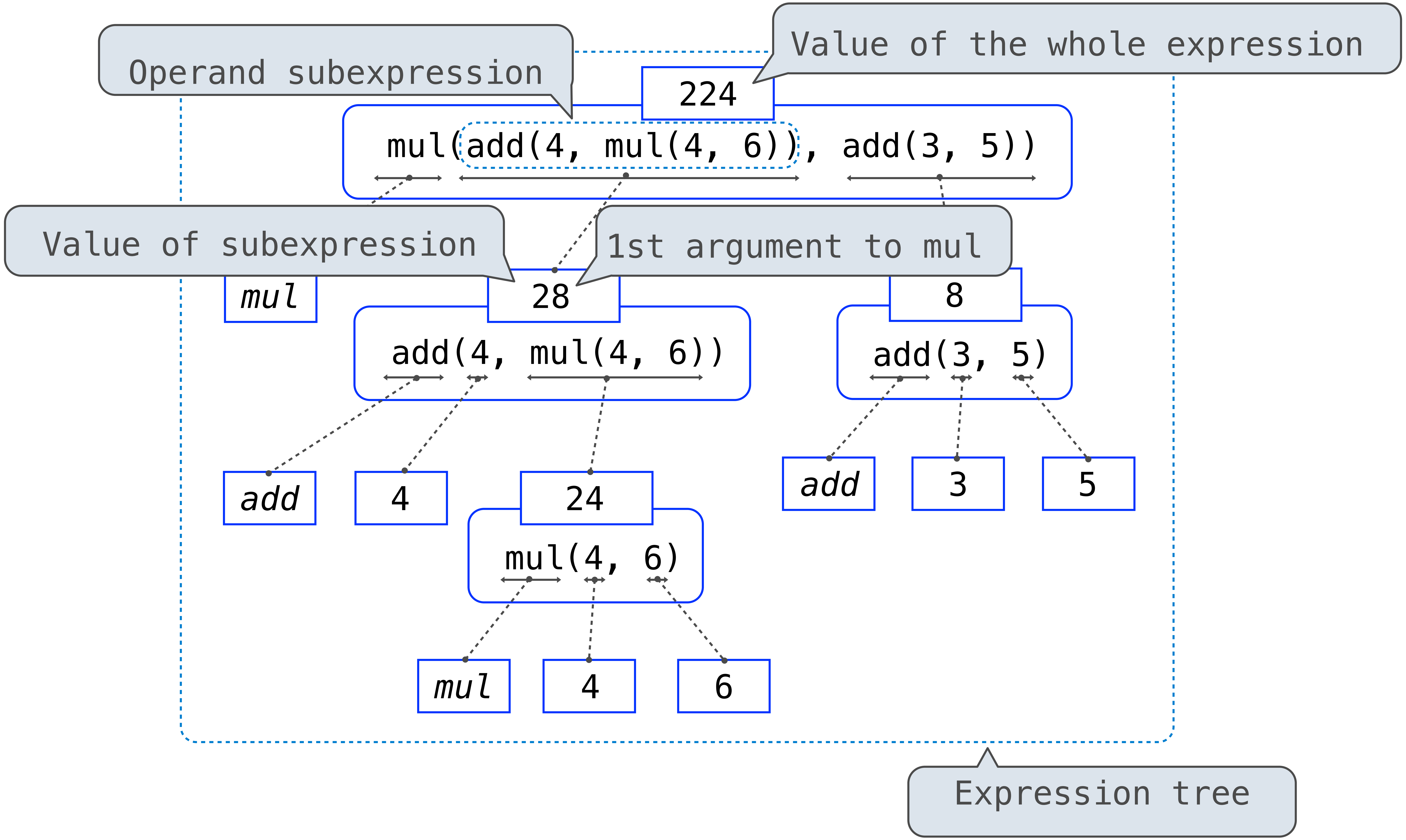
2. Apply the function that is the value of the operator

   to the arguments that are the values of the operands

# Evaluating Nested Expressions



# Evaluating Nested Expressions



# Functions, Values, Objects, Interpreters, and Data

(Demo)

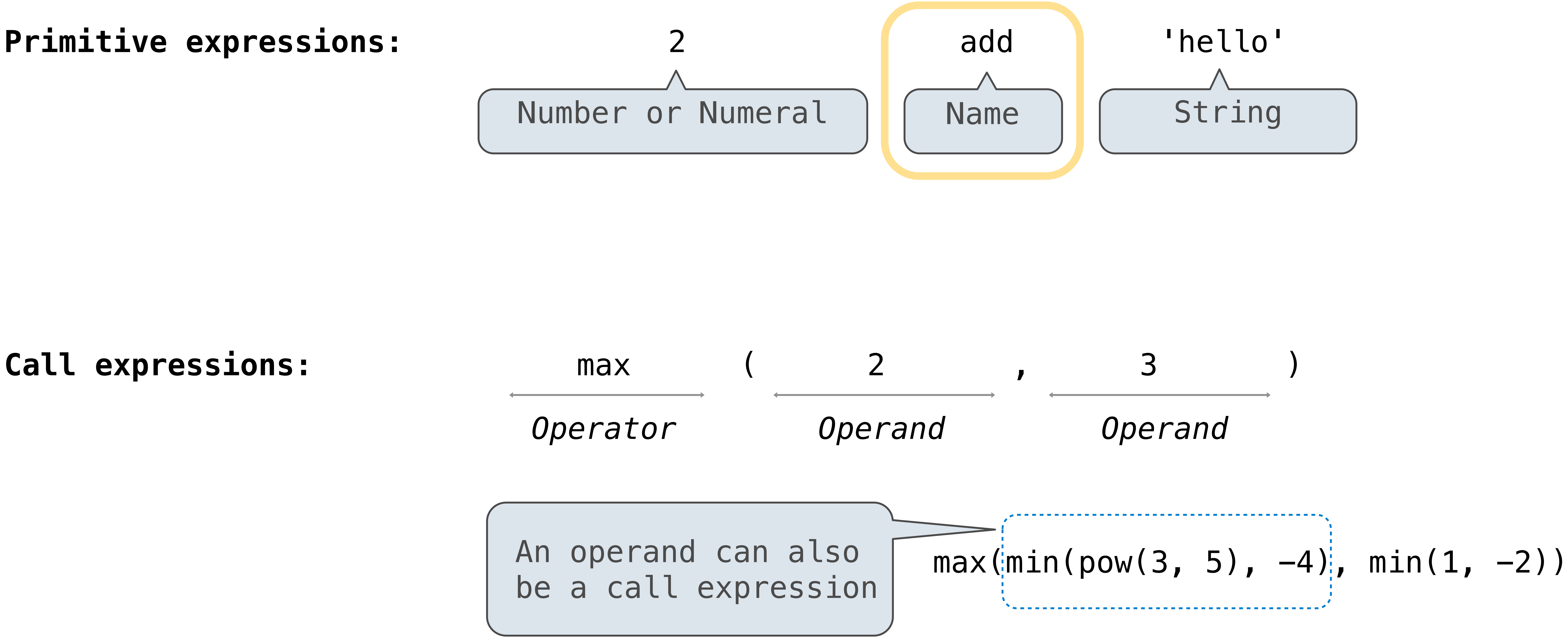
# Names, Assignment, and User-Defined Functions

(Goal: Get you to have a correct understanding of the Notational Machine of Python, the “set of abstractions that define the structure and behavior of a computing device” –Guzdial)

(Demo 2)

computinged.wordpress.com/2016/03/07/notional-machines-and-misconceptions-in-cs-developing-a-research-agenda-at-dagstuhl/

# Types of Expressions



**Primitive expressions:**

`2` — Number or Numeral

`add` — Name

`'hello'` — String

**Call expressions:**

```
max    (    2    ,    3    )
```

*Operator* *Operand* *Operand*

An operand can also be a call expression

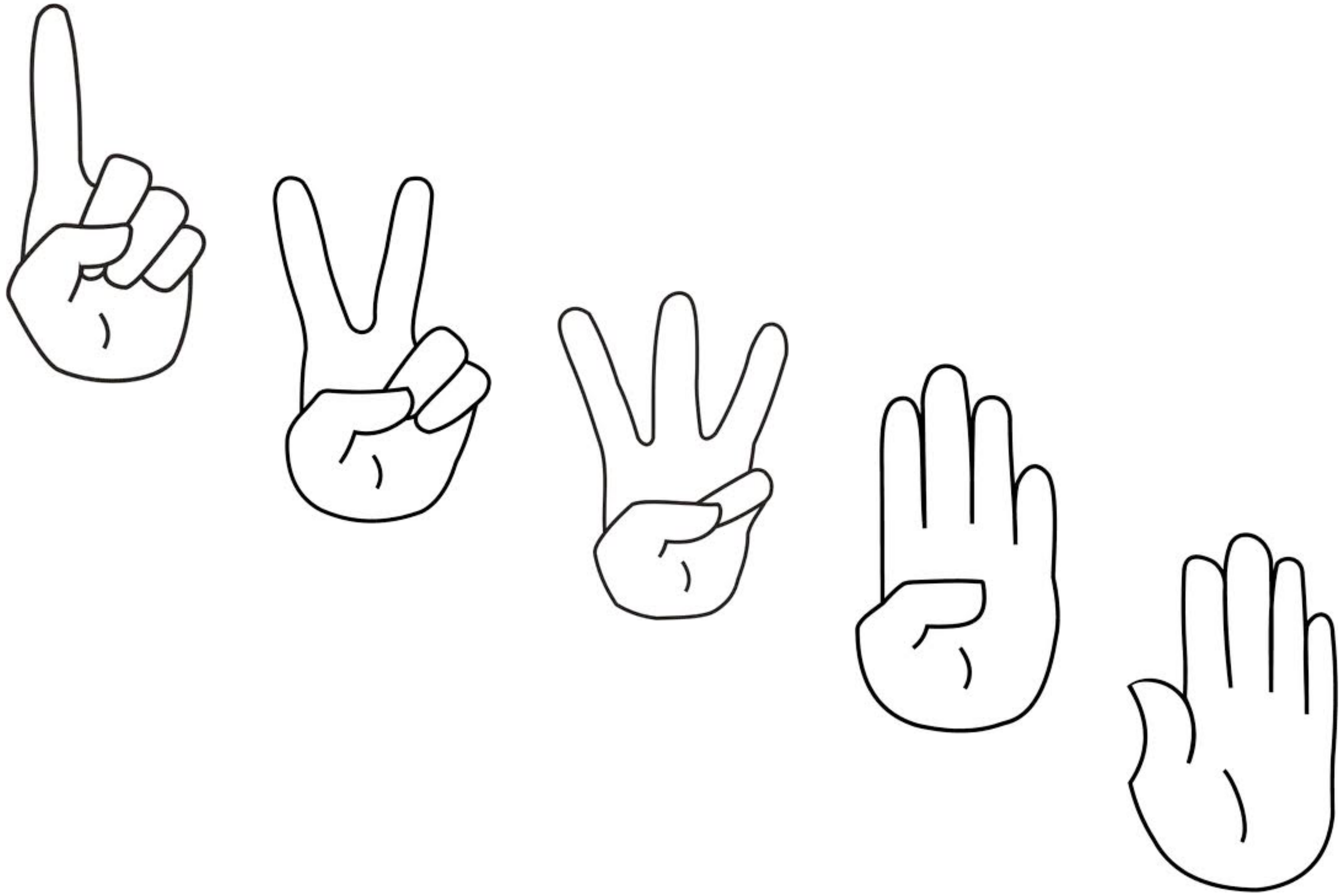
```
max(min(pow(3, 5), -4), min(1, -2))
```

# Discussion Question 1

What is the value of the final expression in this sequence?

```
>>> f = min

>>> f = max

>>> g, h = min, max

>>> max = g

>>> max(f(2, g(h(1, 5), 3)), 4)
```

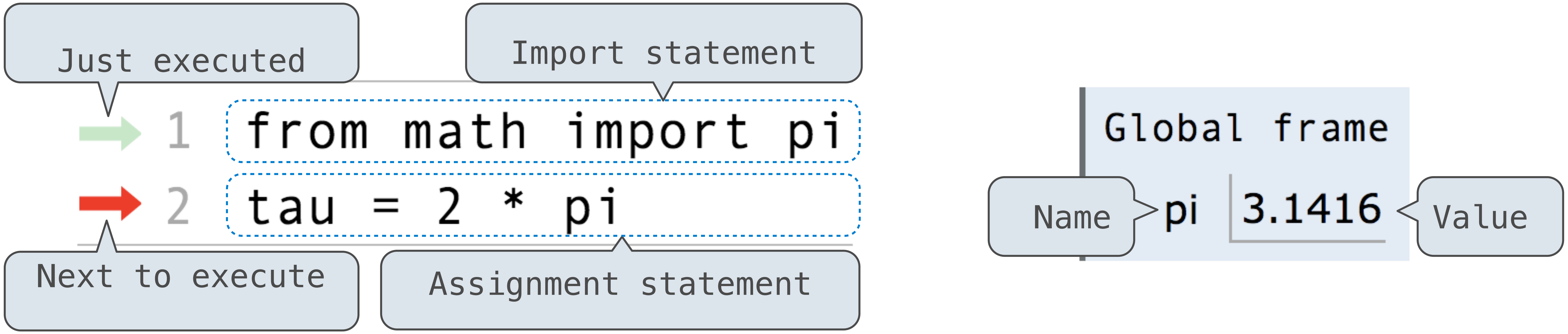
???



# Environment Diagrams

# Environment Diagrams

Environment diagrams visualize the interpreter's process.



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

**Code (left):**

Statements and expressions

Arrows indicate evaluation order

**Frames (right):**

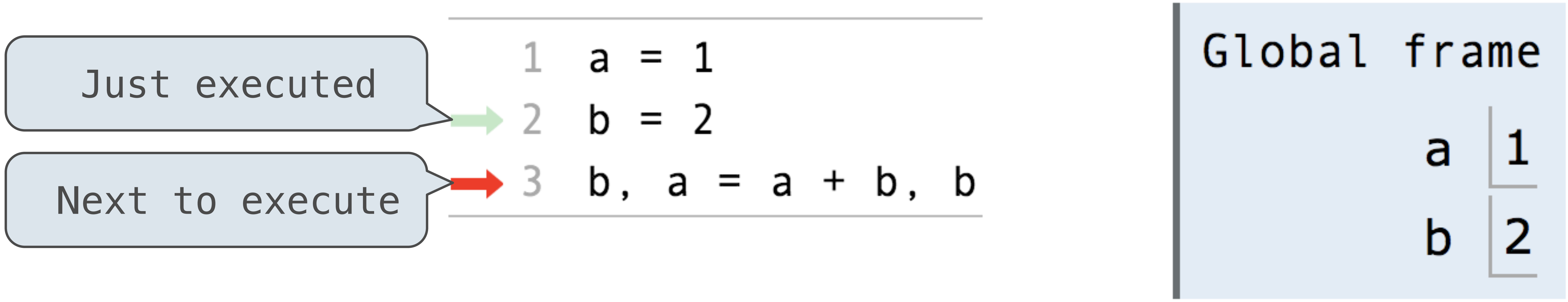
Each name is bound to a value

Within a frame, a name cannot be repeated

(Demo 3)

http://pythontutor.com/composingprograms.html#code=from%20math%20import%20pi%0Atau%20%3D%202%20*%20pi&cumulative=false&curInstr=1&mode=display&origin=composingprograms.js&py=3&rawInputLstJSON=%5B%5D

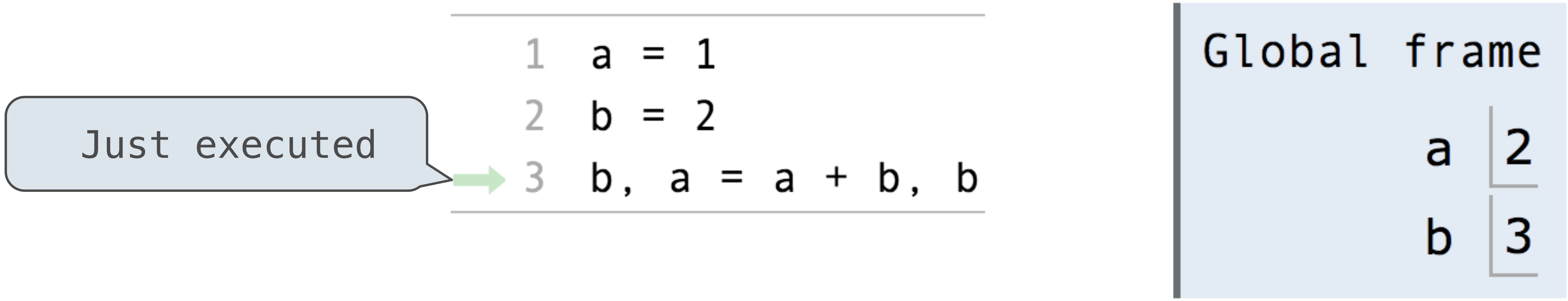
# Assignment Statements



```
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:**

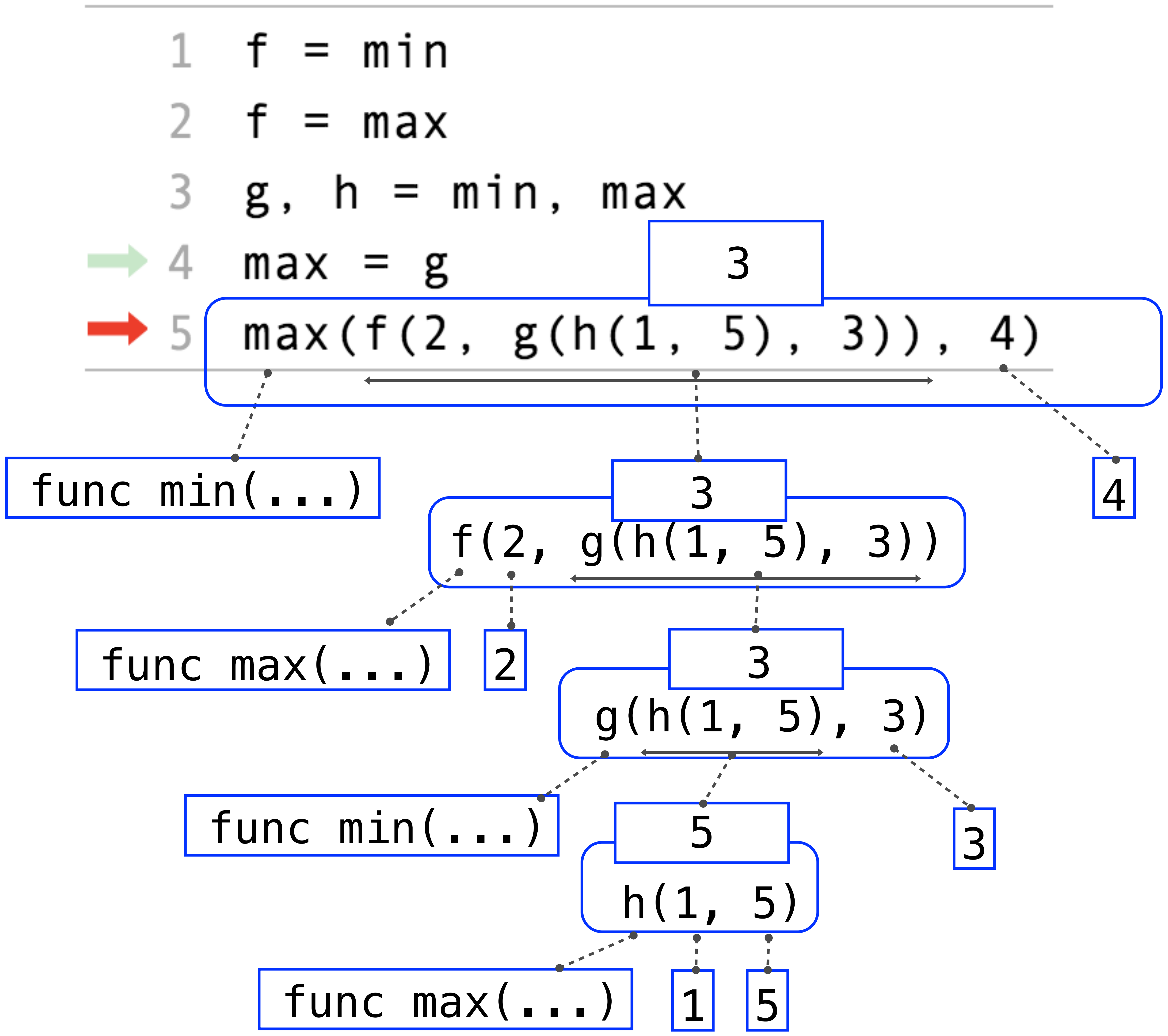
1. Evaluate all expressions to the right of = from left to right.

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

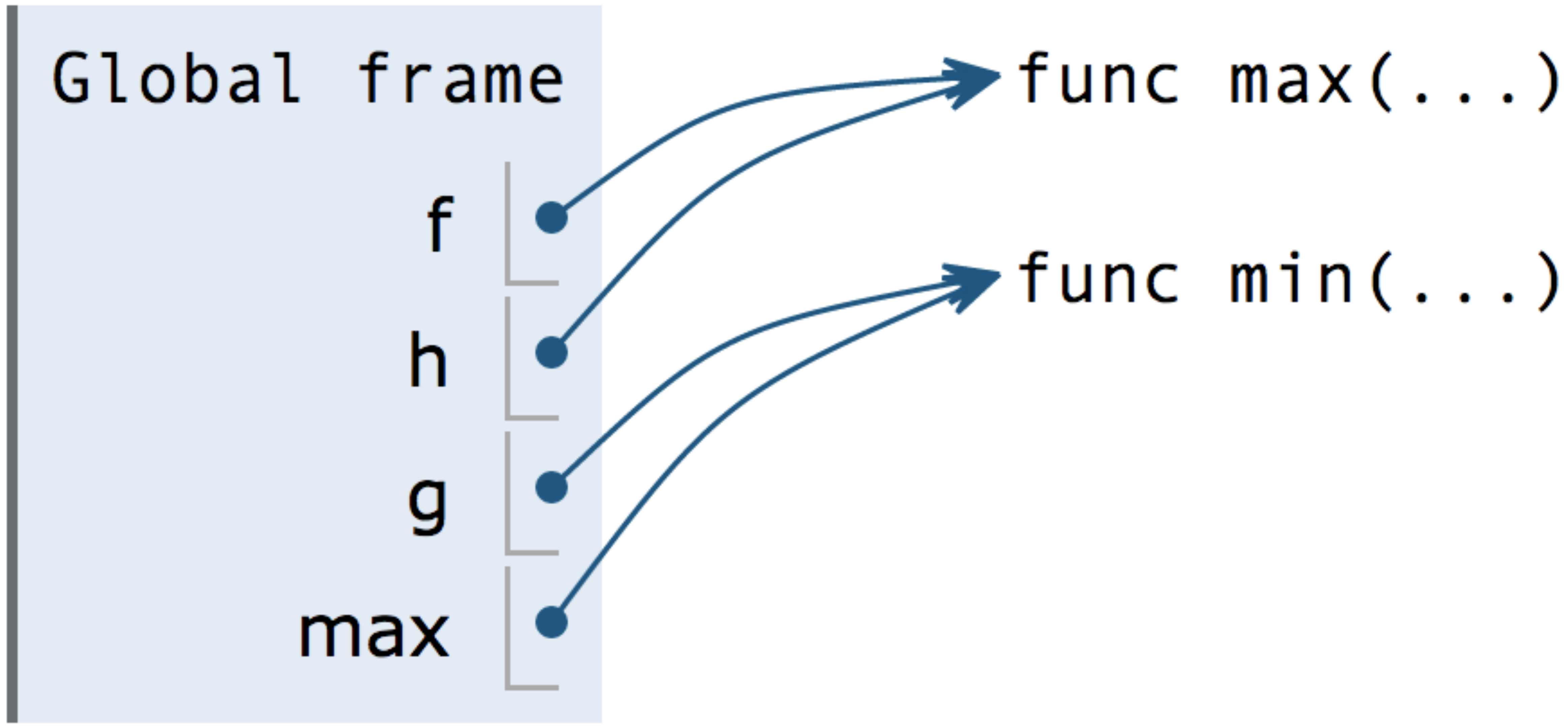
http://pythontutor.com/composingprograms.html#code=a%20%3D%201%0Ab%20%3D%202%0Ab,%20a%20%3D%20a%20%2B%20b,%20b&cumulative=false&curInstr=0&mode=display&origin=composingprograms.js&py=3&rawInputLstJSON=%5B%5D

# Discussion Question 1 Solution

(Demo 4)



```
f = min
f = max
g, h = min, max
max = g
max(f(2, g(h(1, 5), 3)), 4)
```



3

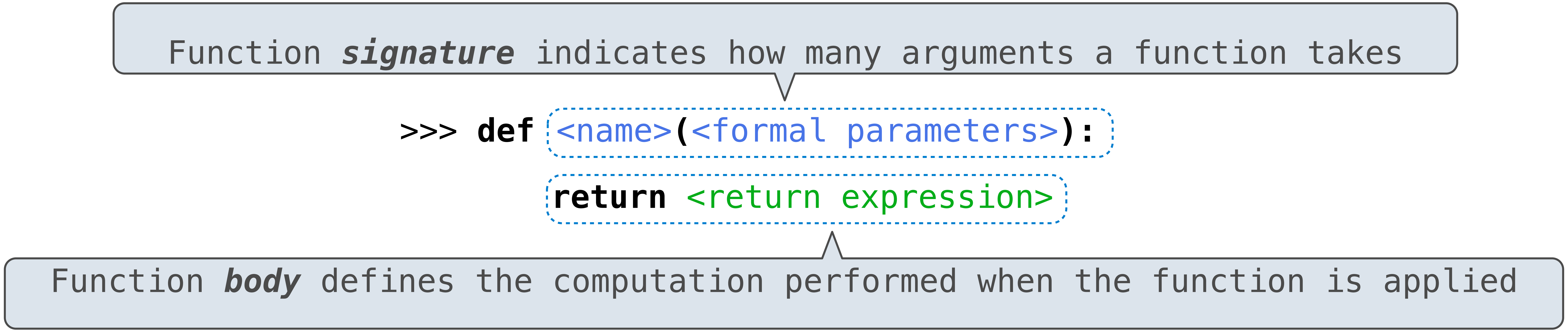
http://pythontutor.com/composingprograms.html#code=f%20%3D%20min%0Af%20%3D%20max%0Ag,%20h%20%3D%20min,%20max%0Amax%20%3D%20g%0Amax%28f%282,%20g%28h%281,%205%29,%203%29%29,%204%29&cumulative=false&curInstr=0&mode=display&origin=composingprograms.js&py=3&rawInputLstJSON=%5B%5D

# Defining Functions

# Defining Functions

Assignment is a simple means of abstraction: binds names to values

Function definition is a more powerful means of abstraction: binds names to expressions



Function ***signature*** indicates how many arguments a function takes

```
>>> def <name>(<formal parameters>):
        return <return expression>
```

Function ***body*** defines the computation performed when the function is applied

**Execution procedure for def statements:**

1. Create a function with signature <name>(<formal parameters>)

2. Set the body of that function to be everything indented after the first line

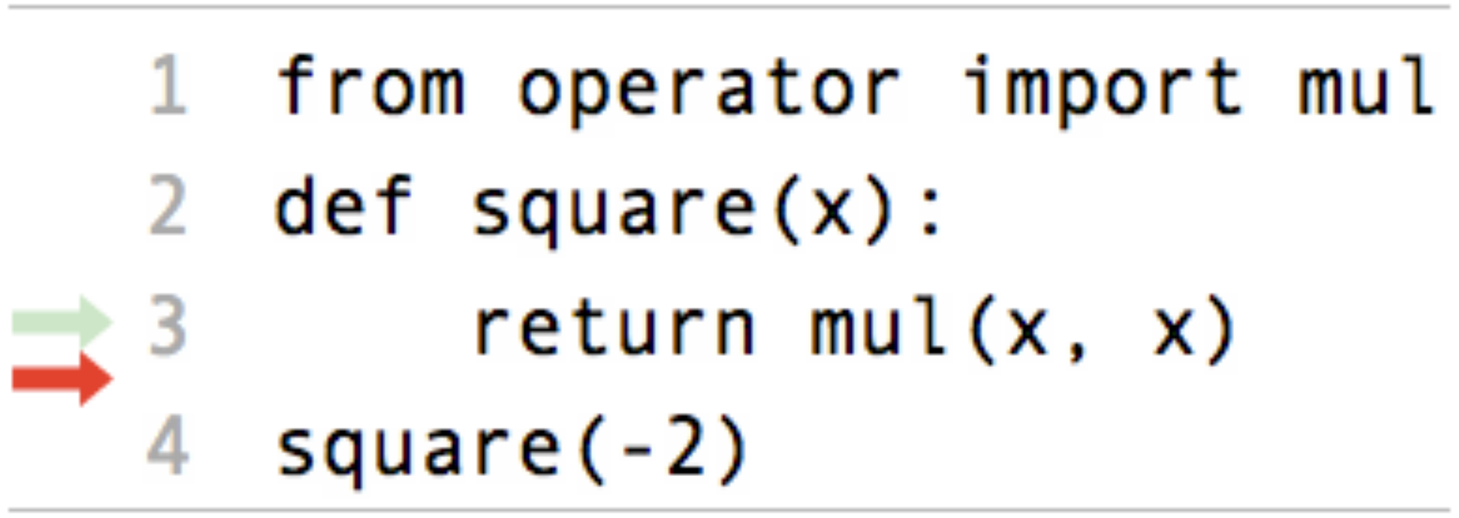
3. Bind <name> to that function in the current frame

# Calling User-Defined Functions

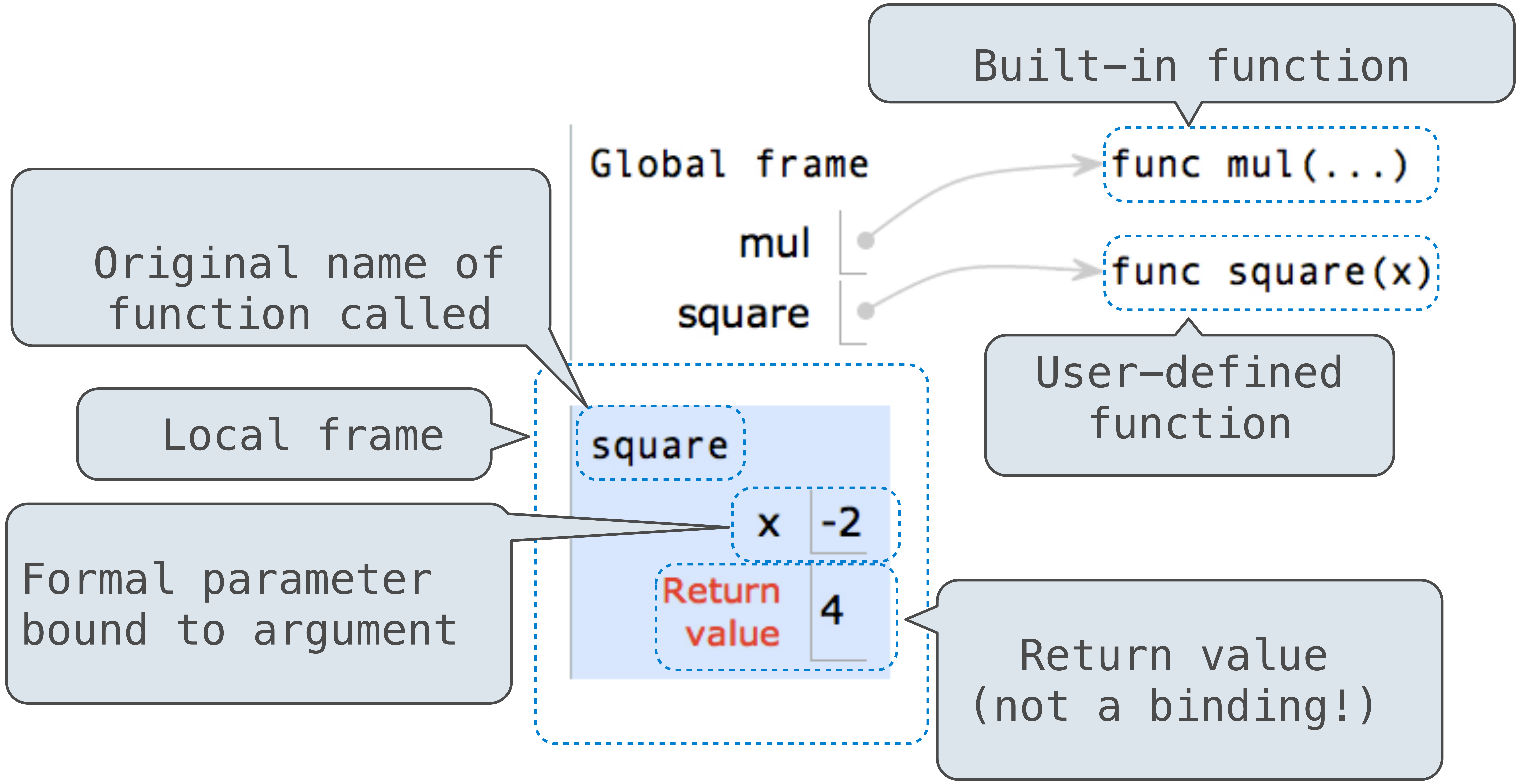
**Procedure for calling/applying user-defined functions (version 1):**

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

(Demo 5)



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



Built-in function

Original name of function called

Local frame

Formal parameter bound to argument

User-defined function

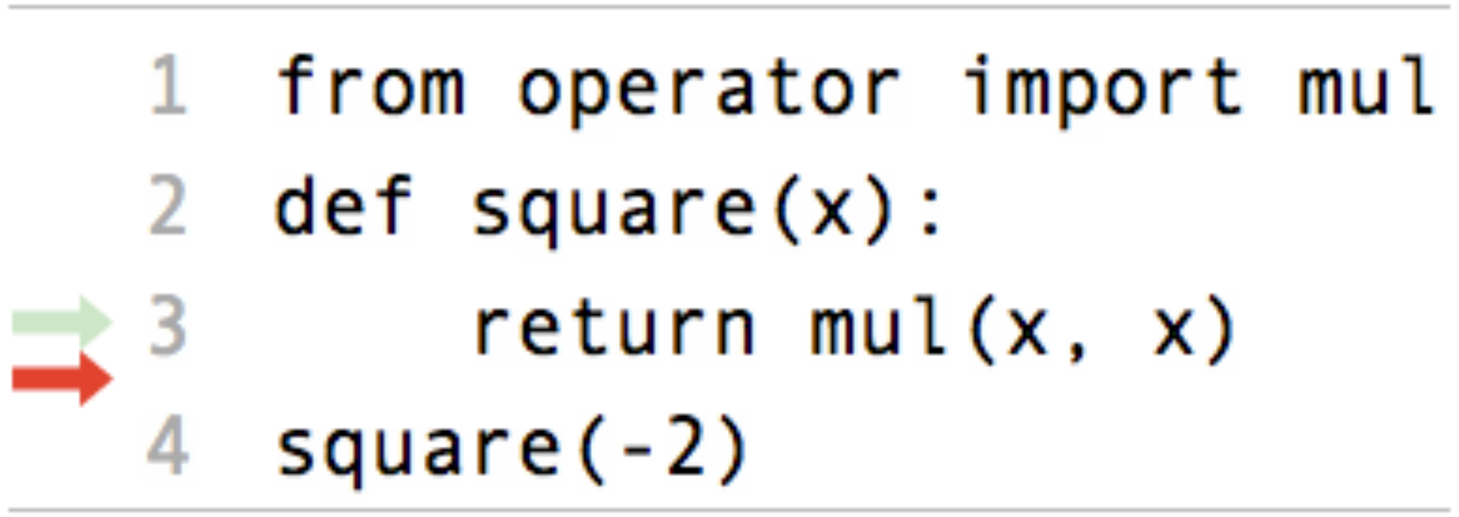
Return value (not a binding!)

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

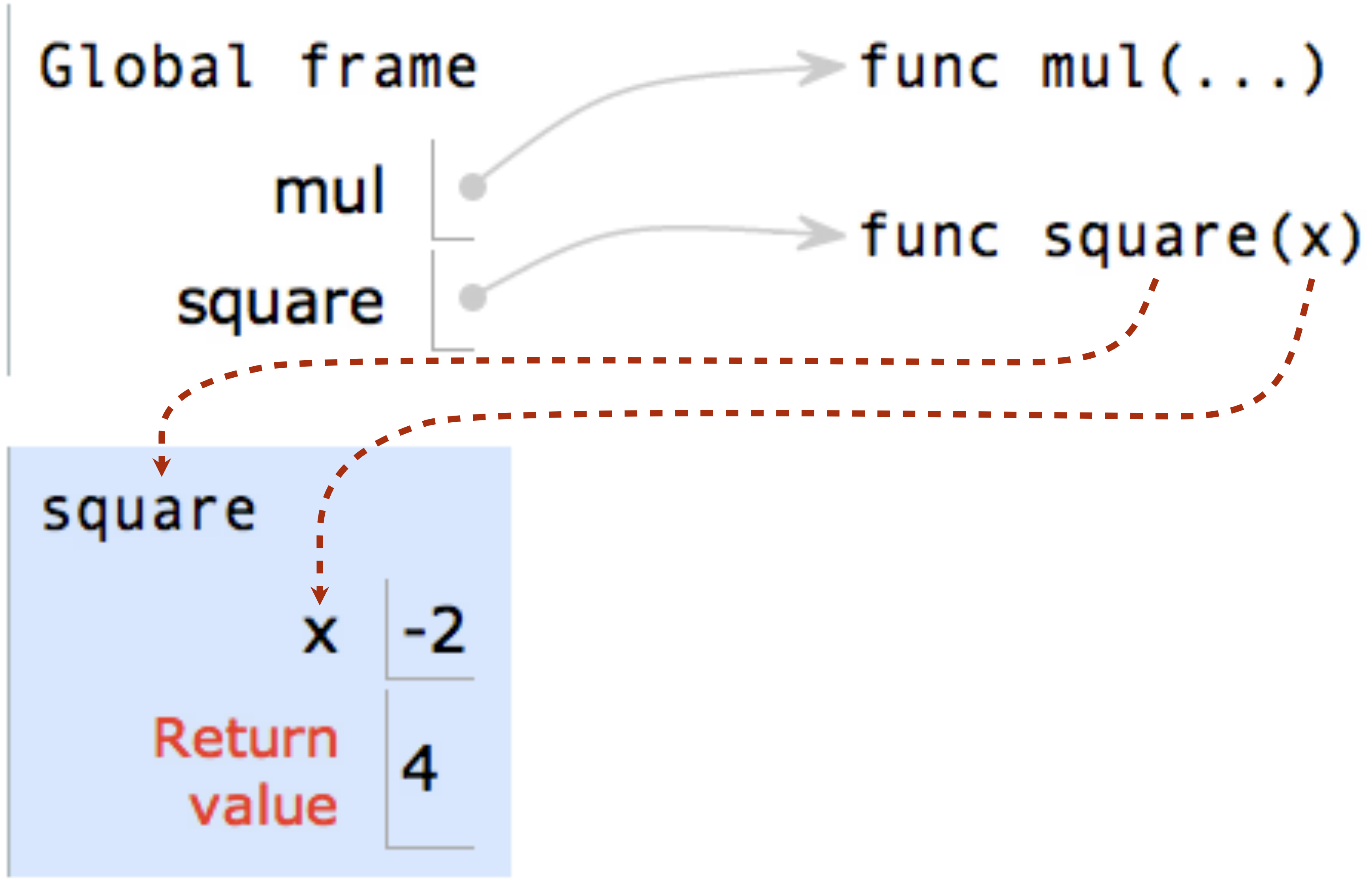
# Calling User-Defined Functions

**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



```
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

# Looking Up Names In Environments

Every expression is evaluated in the context of an environment.

So far, the current environment is either:

- The global frame alone, or
- A local frame, followed by the global frame.

***Most important two things I'll say all day:***

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.

E.g., to look up some name in the body of the square function:

- Look for that name in the local frame.
- If not found, look for it in the global frame.
  (Built-in names like "max" are in the global frame too,
  but we don't draw them in environment diagrams.)

(Demo5)

# Print and None

`(Demo1)`

# None Indicates that Nothing is Returned

The special value None represents nothing in Python

A function that does not explicitly return a value will return None

*Careful*: None is *not displayed* by the interpreter as the value of an expression

```
>>> def does_not_return_square(x):
...     x * x
...
>>> does_not_return_square(4)
>>> sixteen = does_not_return_square(4)
>>> sixteen + 4
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'NoneType' and 'int'
```

No **return**

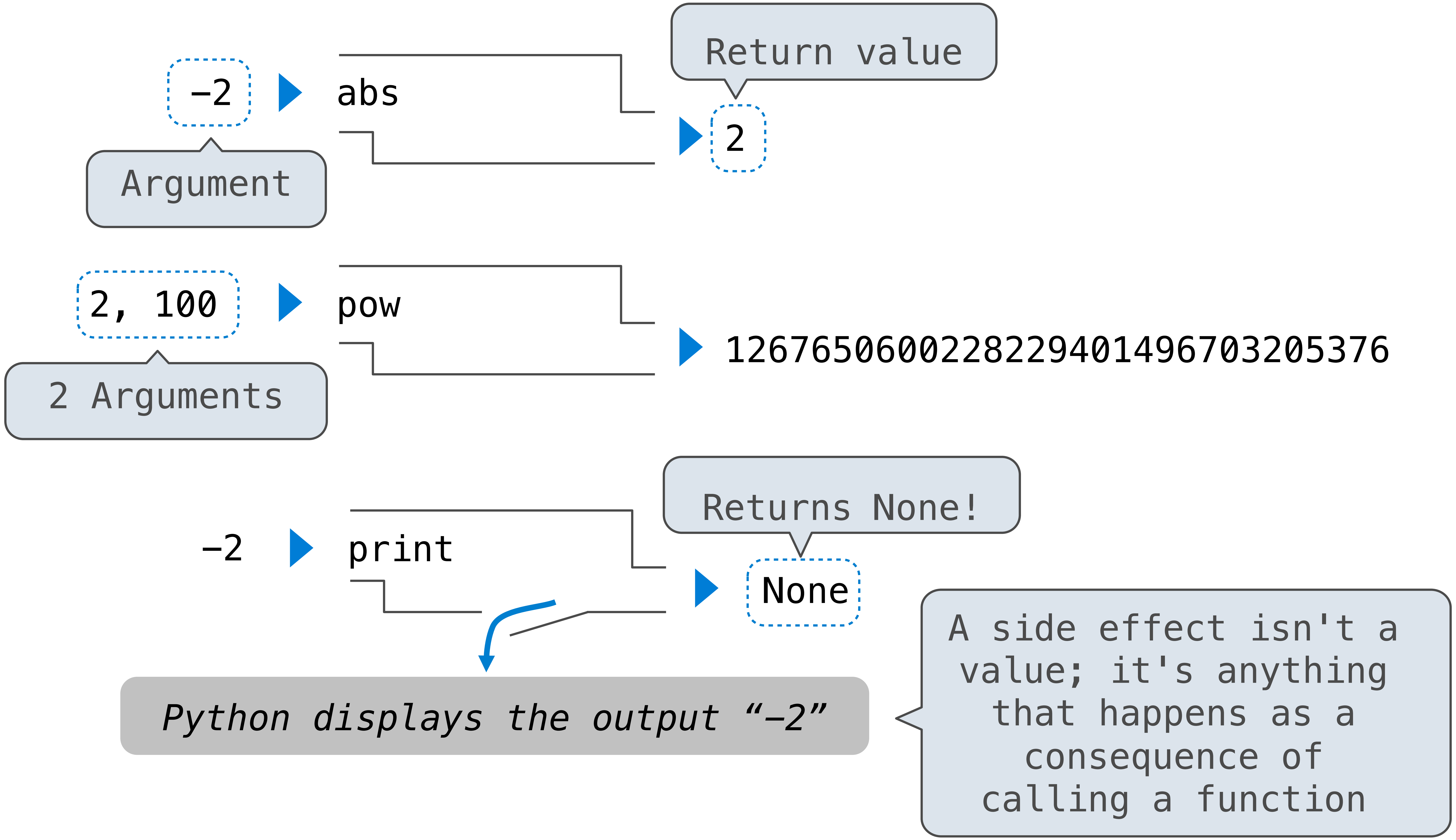
**None** value is not displayed

The name **sixteen** is now bound to the value **None**

# Pure Functions & Non-Pure Functions

**Pure Functions**
*just return values*

**Non-Pure Functions**
*have side effects*



-2 abs 2

Return value

Argument

2, 100 pow 1267650600228229401496703205376

2 Arguments

-2 print None

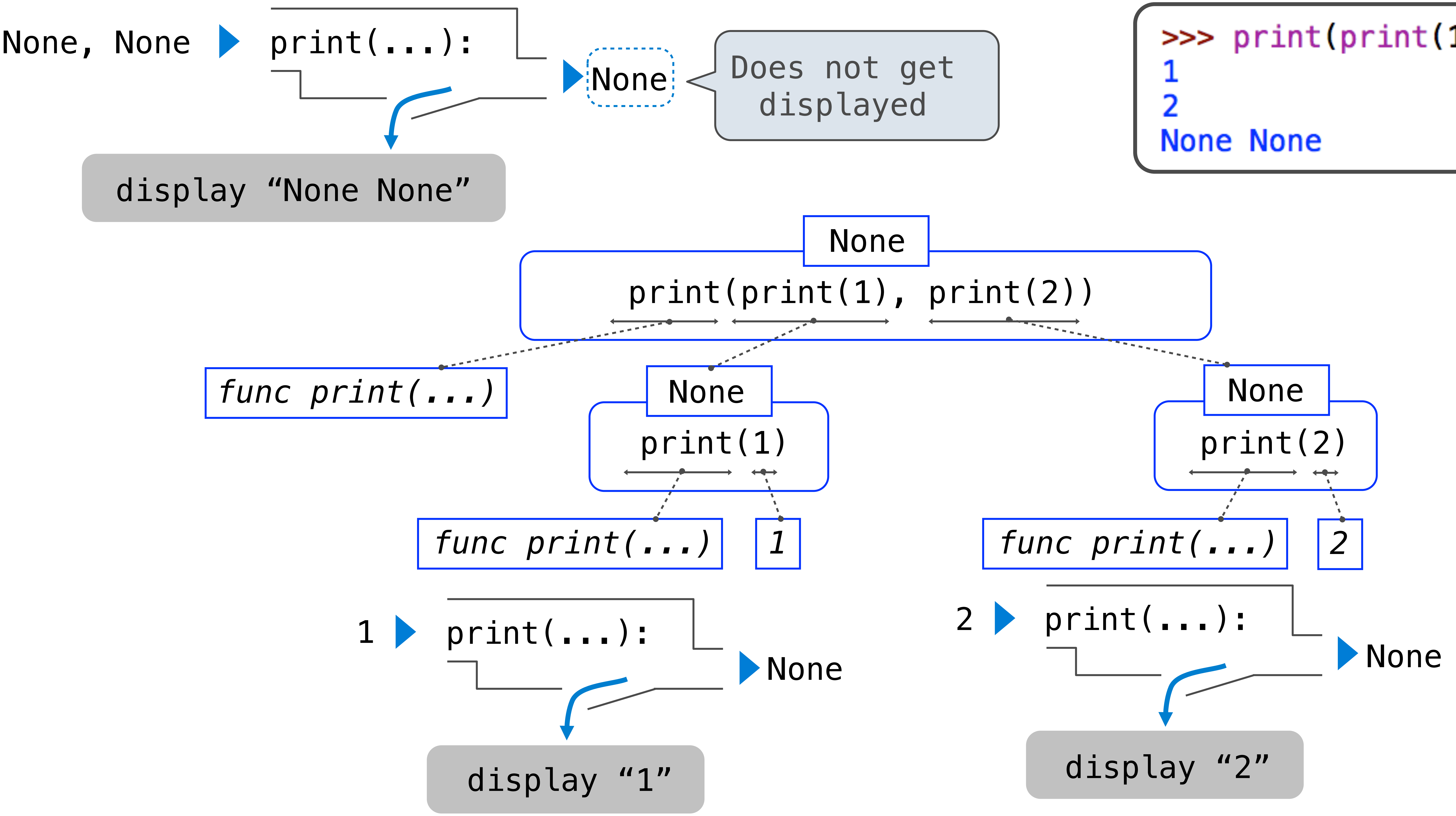
Returns None!

*Python displays the output “-2”*

A side effect isn't a value; it's anything that happens as a consequence of calling a function

# Nested Expressions with Print

```
>>> print(print(1), print(2))
1
2
None None
```

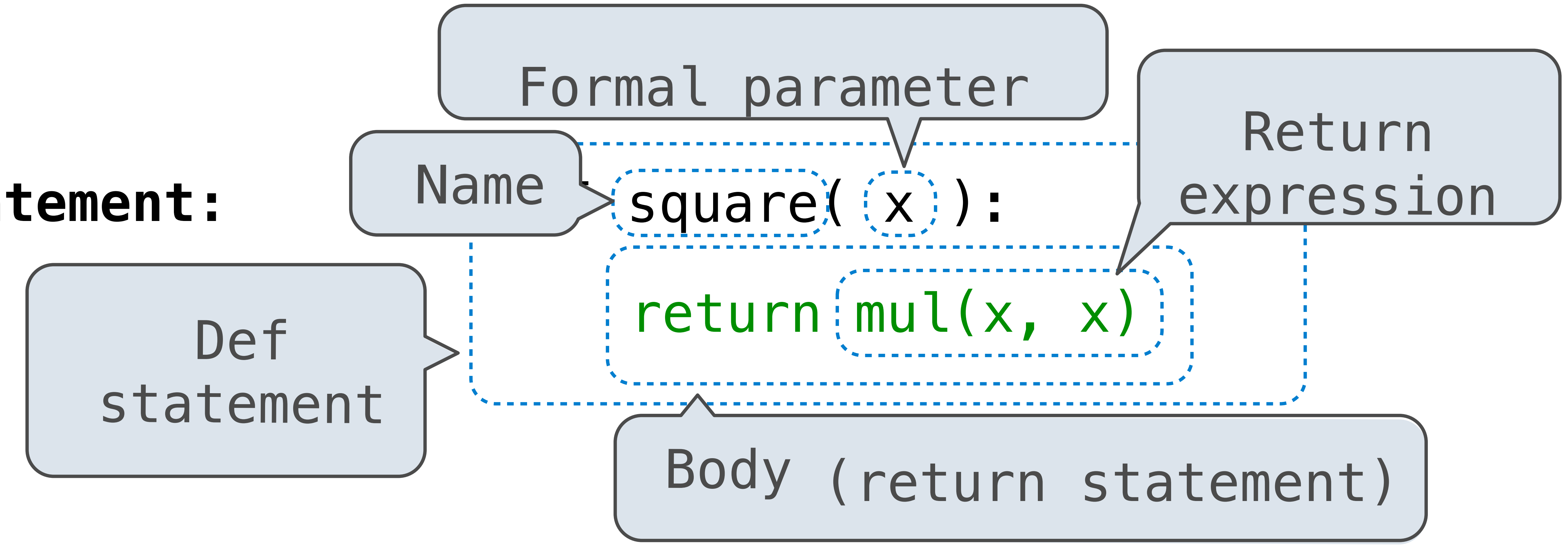


# Multiple Environments

# Life Cycle of a User-Defined Function

**What happens?**

**Def statement:**

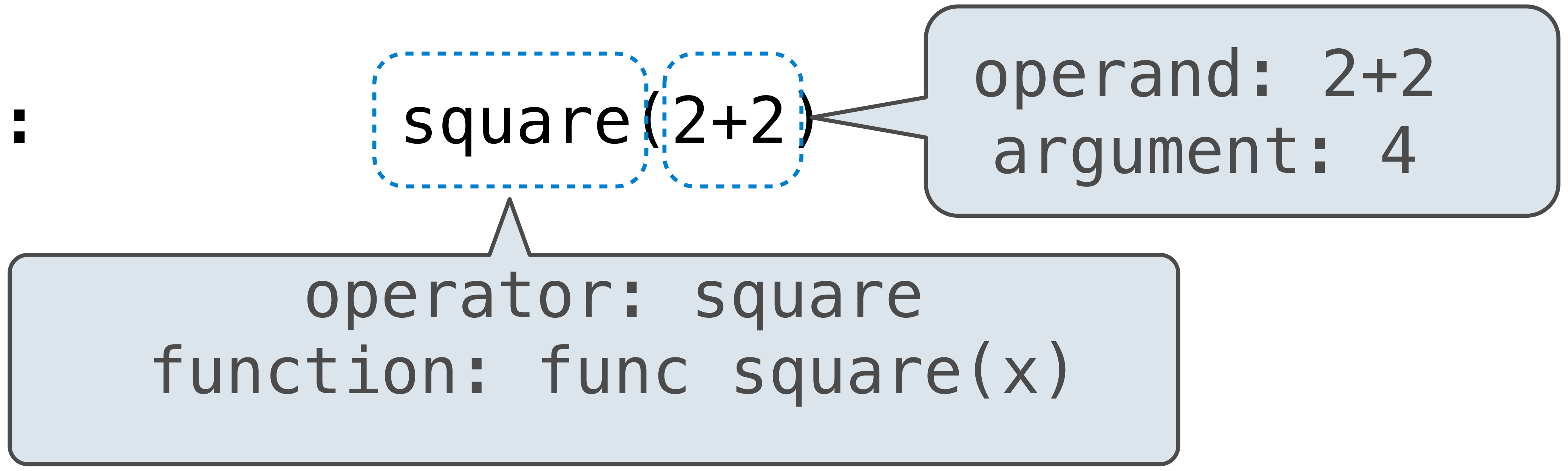


```
square( x ):
    return mul(x, x)
```

A new function is created!

Name bound to that function in the current frame

**Call expression:**

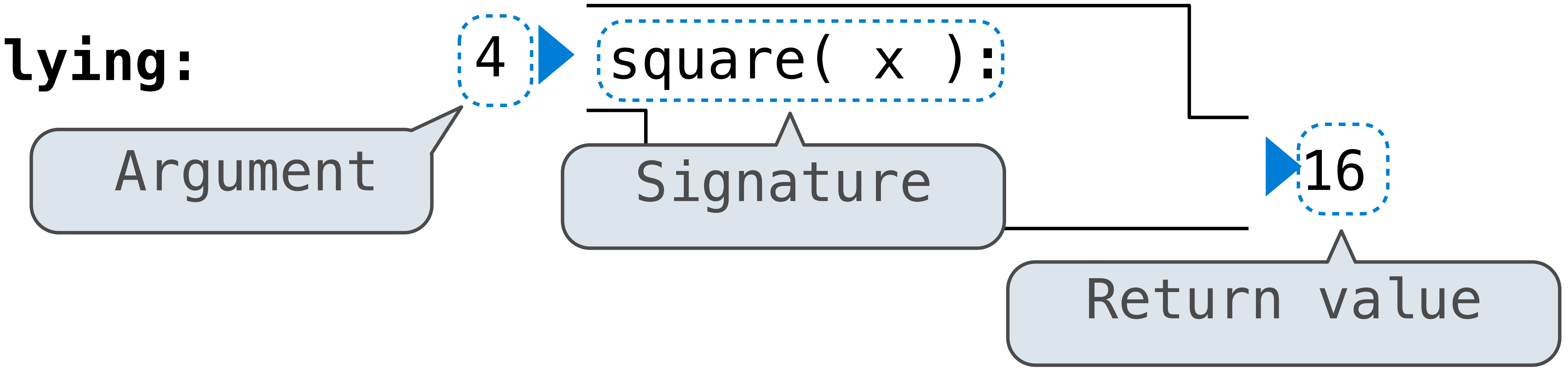


```
square(2+2)
```

Operator & operands evaluated

Function (value of operator) called on arguments (values of operands)

**Calling/Applying:**



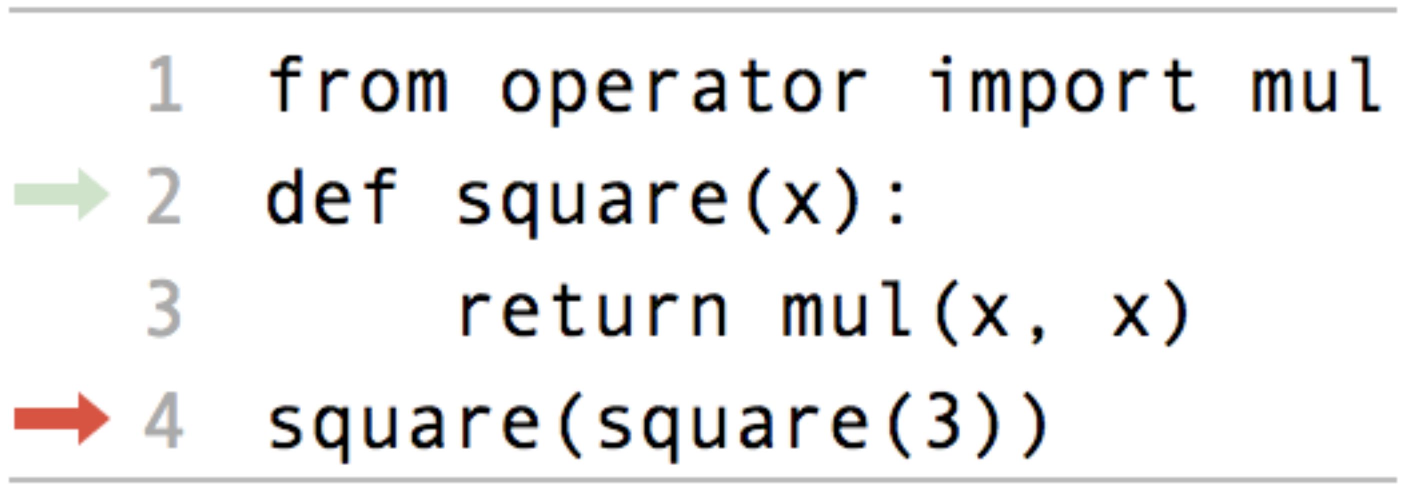
```
4 square( x ): 16
```

A new frame is created!

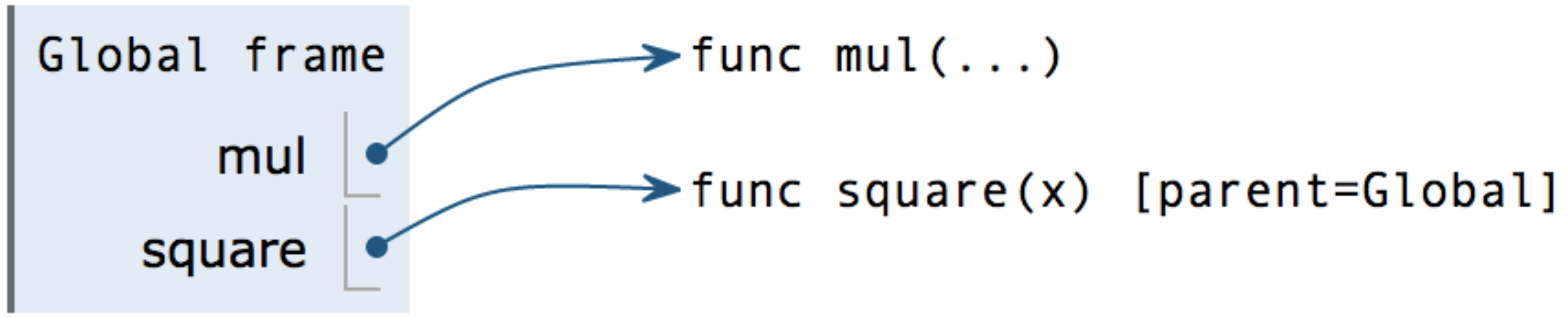
Parameters bound to arguments

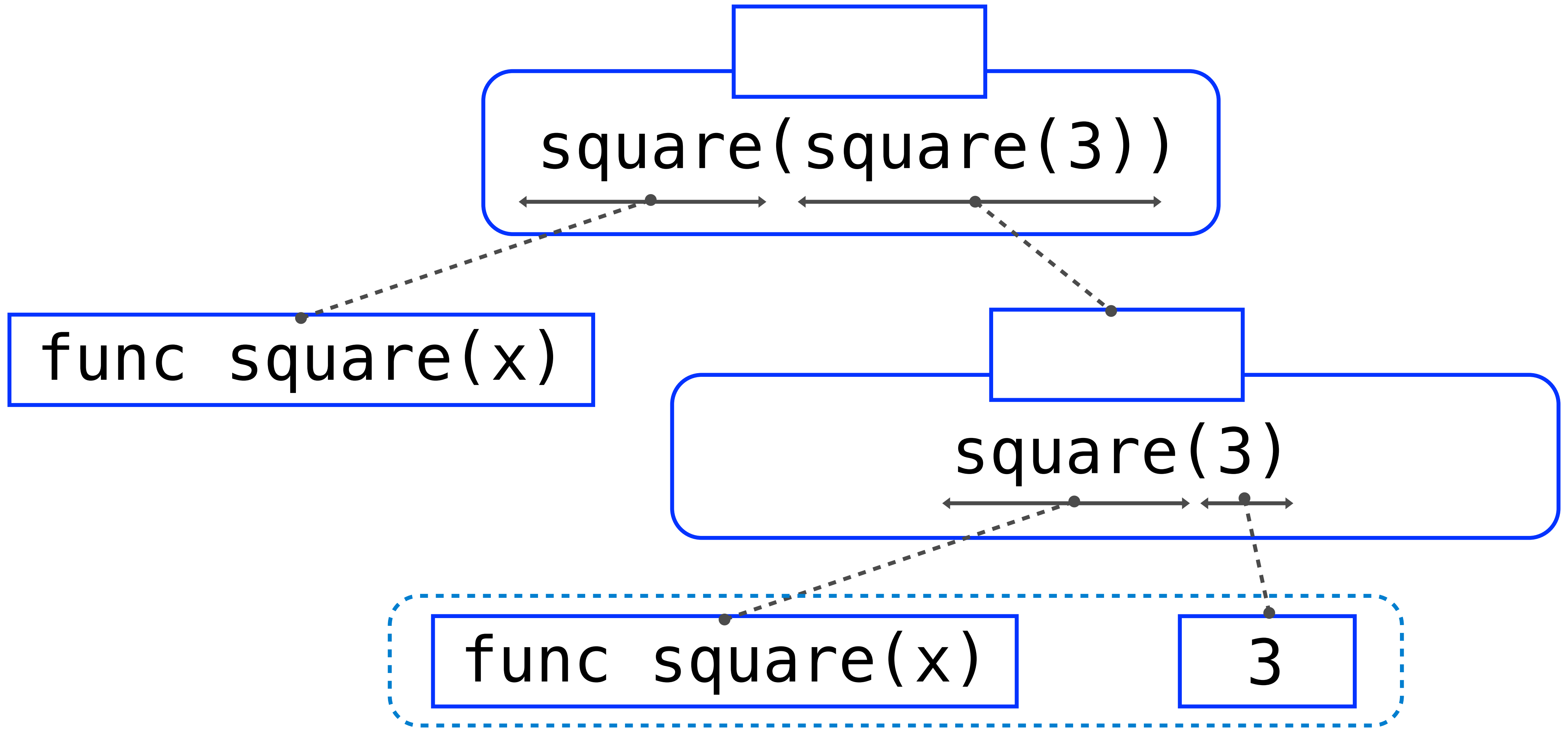
Body is executed in that new environment

# Multiple Environments in One Diagram!



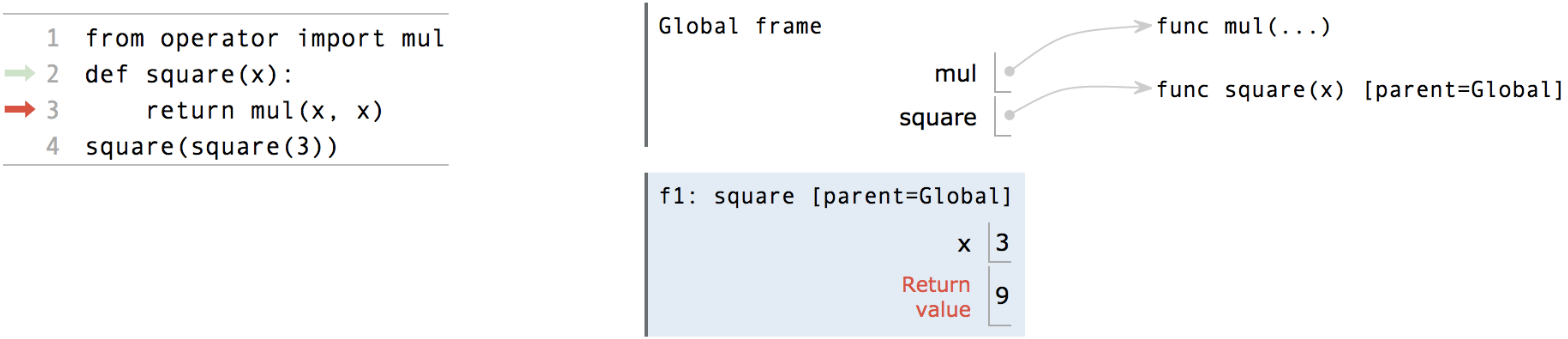
```
from operator import mul
def square(x):
    return mul(x, x)
square(square(3))
```





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%28square%283%29%29&cumulative=true&curInstr=0&mode=display&origin=composingprograms.js&py=3&rawInputLstJSON=%5B%5D

# Multiple Environments in One Diagram!



```
from operator import mul
def square(x):
    return mul(x, x)
square(square(3))
```

Global frame

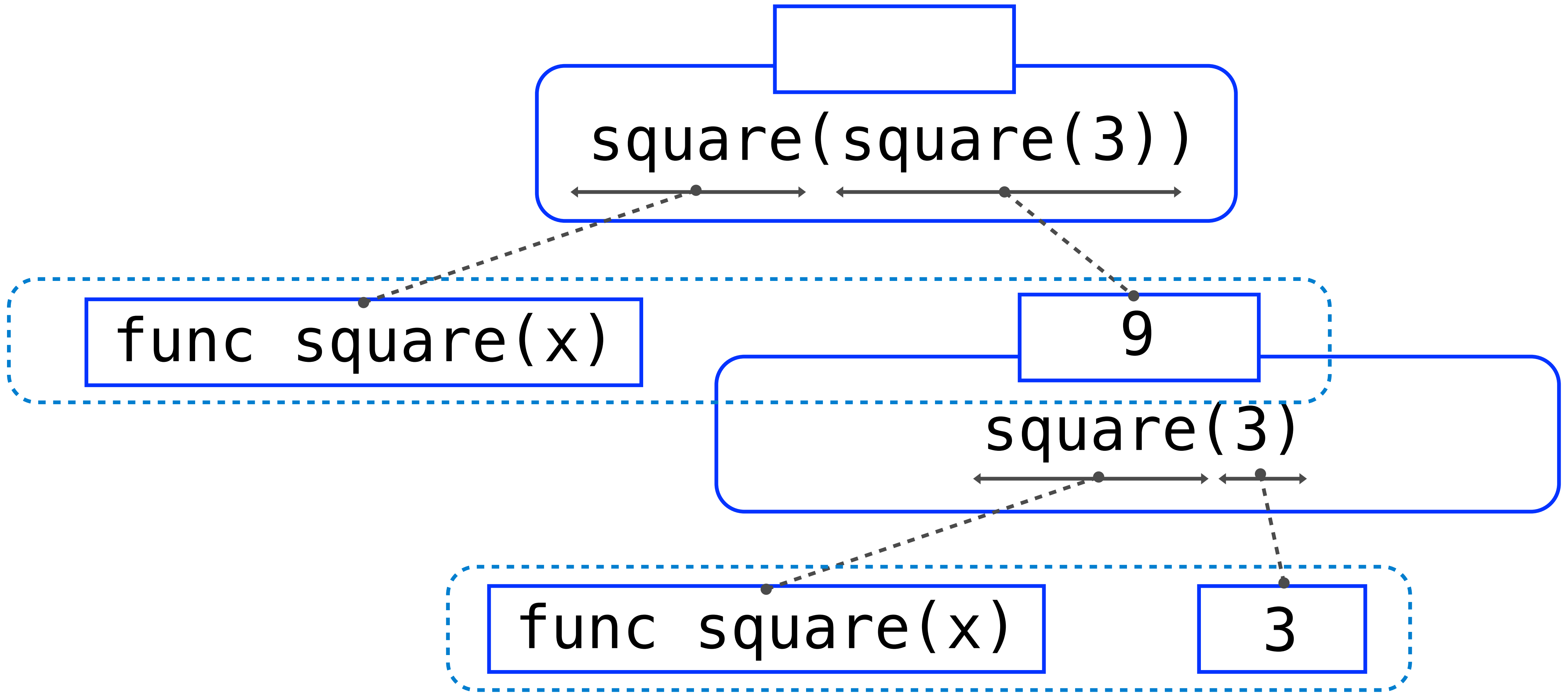
mul → func mul(...)

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

f1: square [parent=Global]

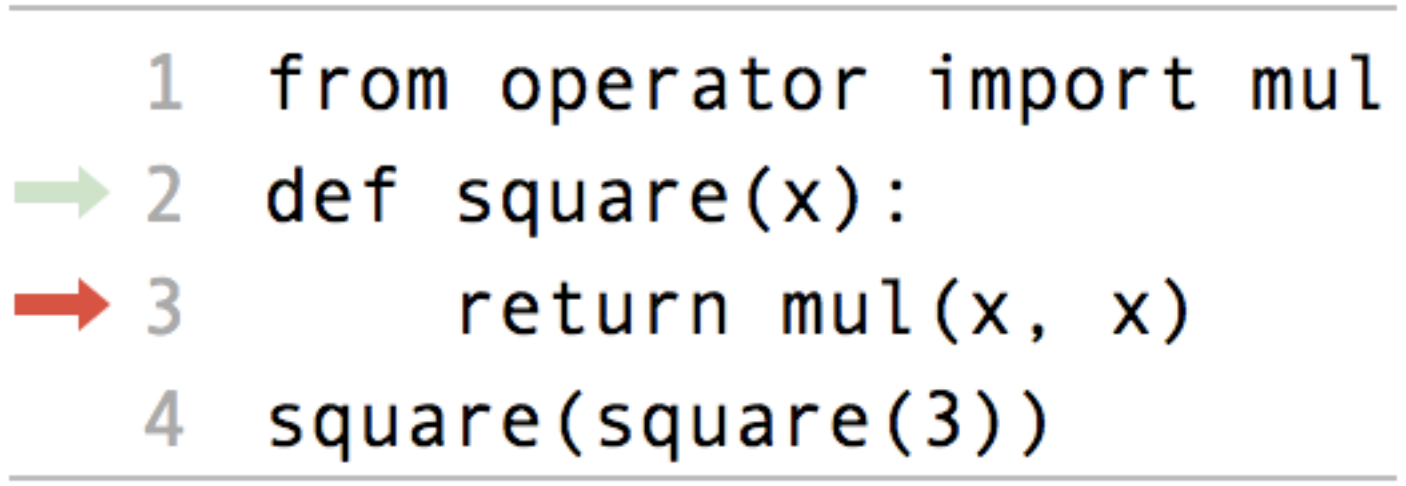
x 3

Return value 9

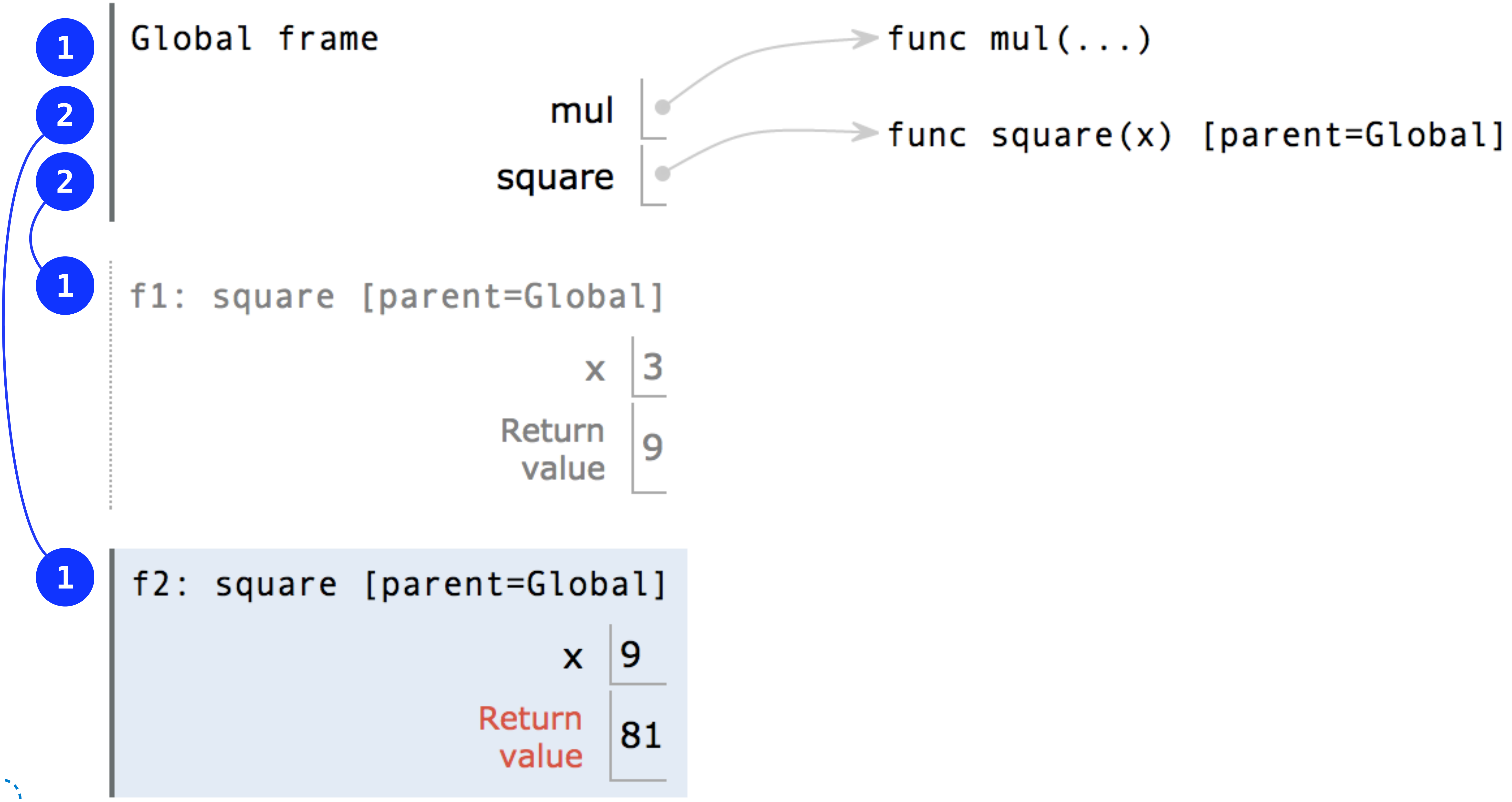


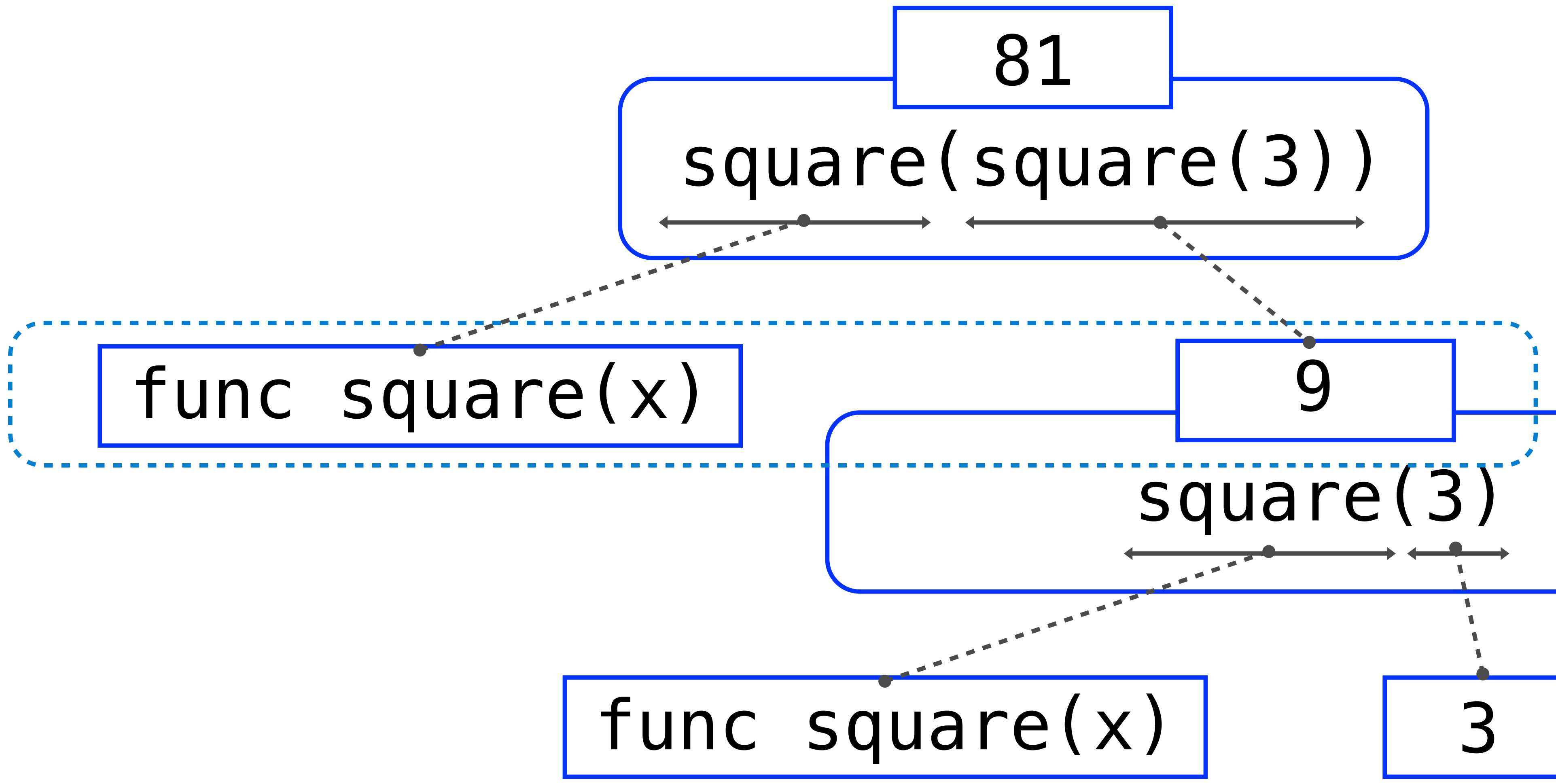
http://pythontutor.com/composingprograms.html#code=from%20operator%20import%20mul%0Adef%20square%28x%29%3A%0A%20%20%20%20return%20mul%28x,%20x%29%0Asquare%28square%283%29%29&cumulative=true&curInstr=0&mode=display&origin=composingprograms.js&py=3&rawInputLstJSON=%5B%5D

# Multiple Environments in One Diagram!



```python
from operator import mul
def square(x):
    return mul(x, x)
square(square(3))
```


1
2
2
1
1
Global frame
mul
square
func mul(...)
func square(x) [parent=Global]
f1: square [parent=Global]
x
3
Return value
9
f2: square [parent=Global]
x
9
Return value
81



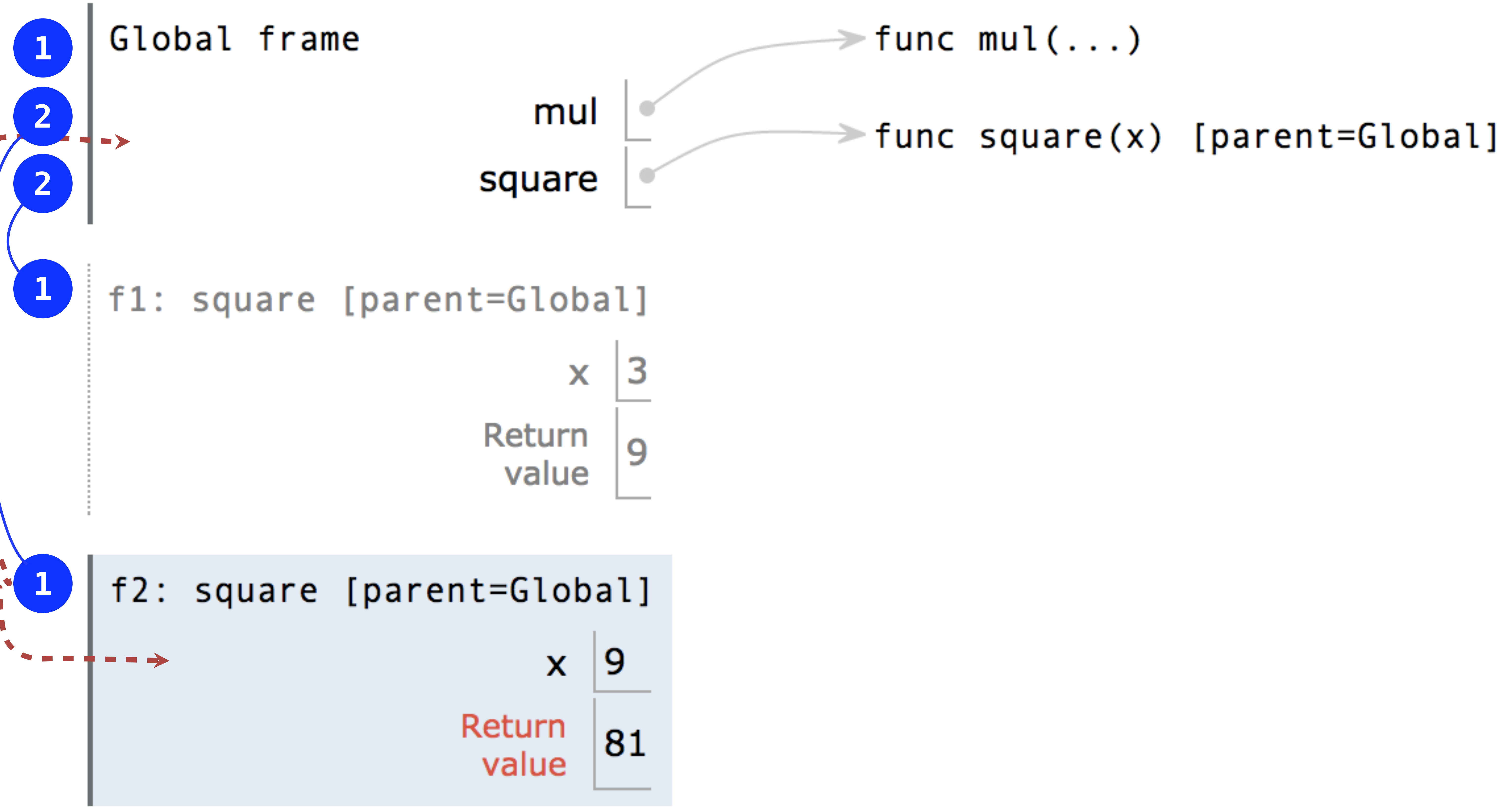
An environment is a sequence of frames.

- The global frame alone
- A local, then the global 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%28square%283%29%29&cumulative=true&curInstr=0&mode=display&origin=composingprograms.js&py=3&rawInputLstJSON=%5B%5D

# Names Have No Meaning Without Environments

```
from operator import mul
def square(x):
    return mul(x, x)
square(square(3))
```


1
2
2
1
1
Global frame
mul
square
func mul(...)
func square(x) [parent=Global]
f1: square [parent=Global]
x
3
Return value
9
f2: square [parent=Global]
x
9
Return value
81

Every expression is evaluated in the context of an environment.

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

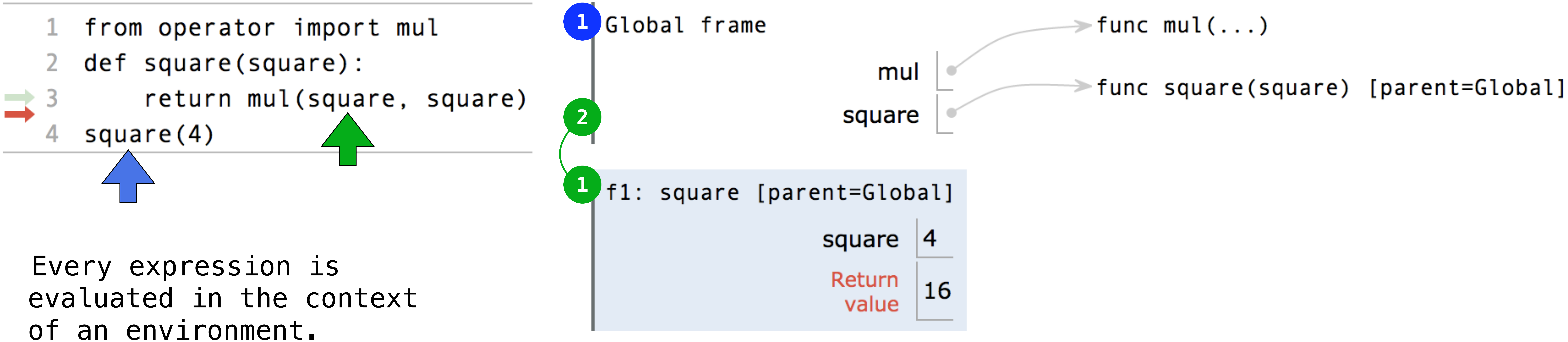
An environment is a sequence of frames.

- The global frame alone
- A local, then the global 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%28square%283%29%29&cumulative=true&curInstr=0&mode=display&origin=composingprograms.js&py=3&rawInputLstJSON=%5B%5D

# Names Have Different Meanings in Different Environments

A call expression and the body of the function being called are evaluated in different environments



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

1
Global frame
mul
square
func mul(...)
func square(square) [parent=Global]
2
1
f1: square [parent=Global]
square
4
Return value
16

Every expression is evaluated in the context of an environment.

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

http://pythontutor.com/composingprograms.html#code=from%20operator%20import%20mul%0Adef%20square%28square%29%3A%0A%20%20%20%20return%20mul%28square,%20square%29%0Asquare%284%29&cumulative=true&curInstr=0&mode=display&origin=composingprograms.js&py=3&rawInputLstJSON=%5B%5D

# Miscellaneous Python Features

Division

Multiple Return Values

Source Files

Doctests

Default Arguments

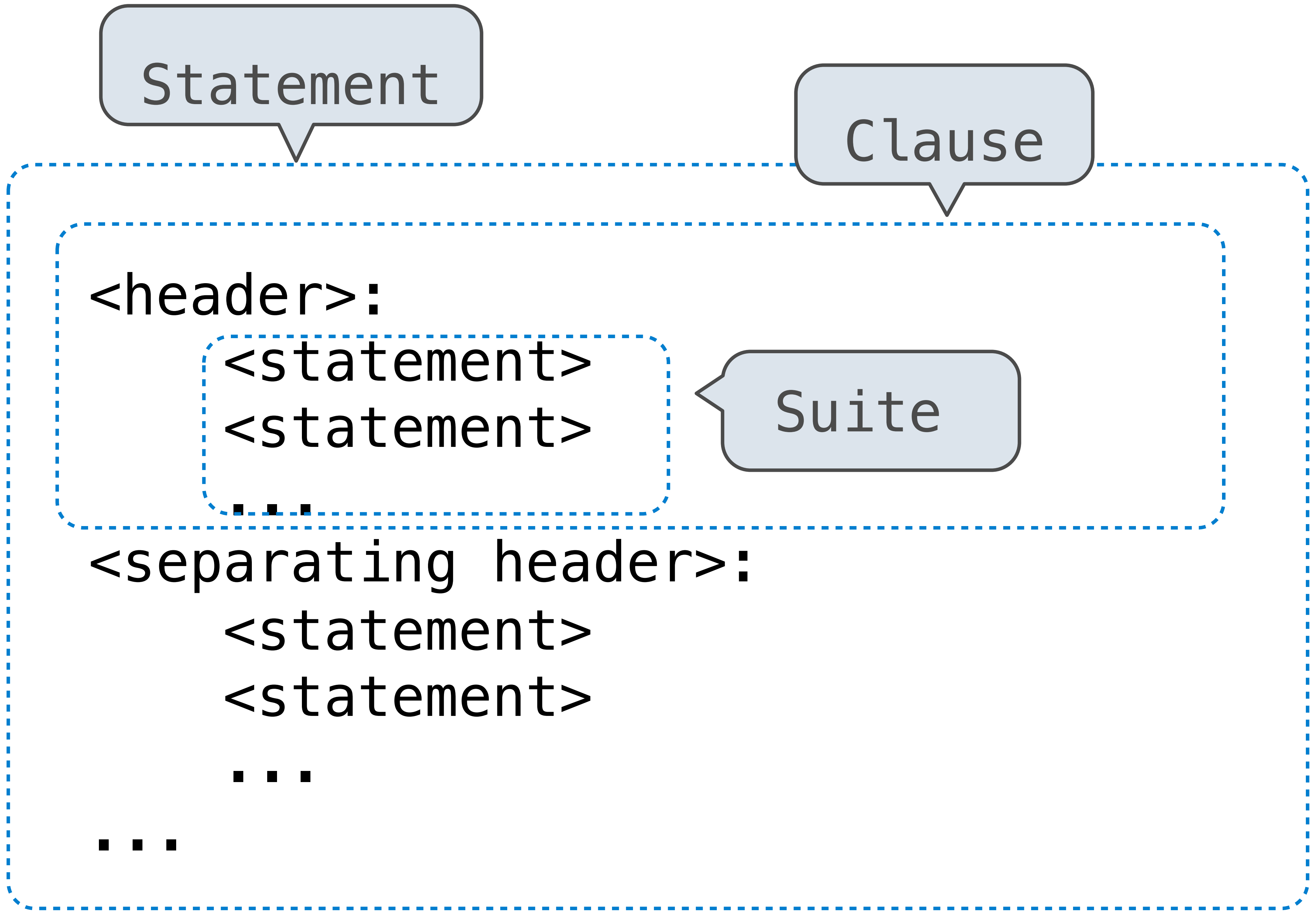
(Demo2)

# Conditional Statements

# Statements

A ***statement*** is executed by the interpreter to perform an action

**Compound statements:**



```
<header>:
    <statement>
    <statement>
    ...
<separating header>:
    <statement>
    <statement>
    ...
...
```

The first header determines a statement’s type

The header of a clause “controls” the suite that follows

def statements are compound statements

# Compound Statements

**Compound statements:**

```
<header>:
    <statement>
    <statement>
    ...
<separating header>:
    <statement>
    <statement>
    ...
...
```

Suite

A suite is a sequence of statements

To “execute” a suite means to execute its sequence of statements, in order

**Execution Rule for a sequence of statements:**

- Execute the first statement
- Unless directed otherwise, execute the rest

# Conditional Statements

(Demo3)

```
def my_abs(x):
    """Return the absolute value of x."""
    if x < 0:
        return -x
    elif x == 0:
        return 0
    else:
        return x
```

1 statement,
3 clauses,
3 headers,
3 suites

**Execution Rule for Conditional Statements:**

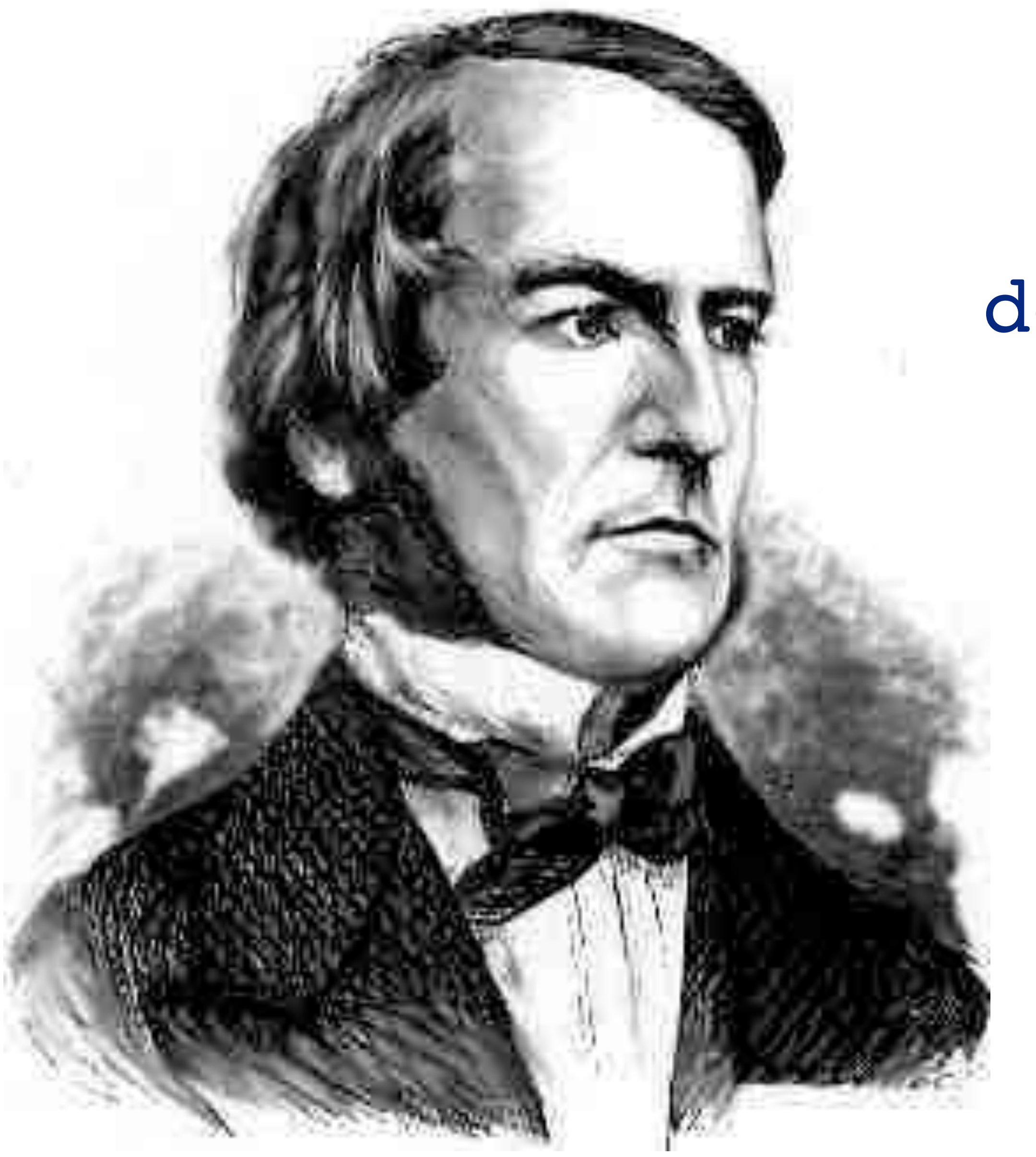
Each clause is considered in order.

1. Evaluate the header's expression.
2. If it is a true value, execute the suite & skip the remaining clauses.

**Syntax Tips:**

1. Always starts with "if" clause.
2. Zero or more "elif" clauses.
3. Zero or one "else" clause, always at the end.

# Boolean Contexts



*George Boole*

```
def my_abs(x):
    """Return the absolute value of x."""
    if x < 0:
        return -x
    elif x == 0:
        return 0
    else:
        return x
```

Two boolean contexts

False values in Python: False, 0, '', None *(more to come)*

True values in Python: Anything else (True)

**Read Section 1.5.4!**

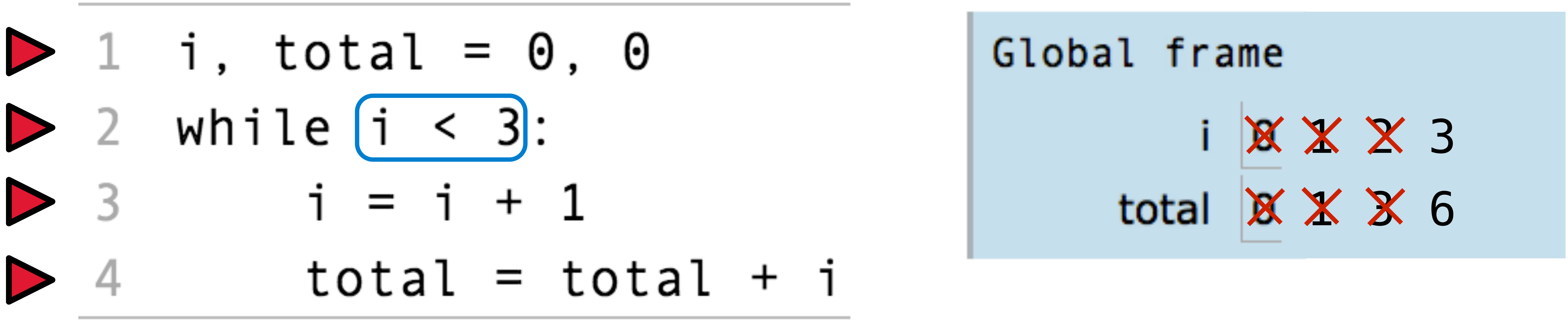
Reading: http://composingprograms.com/pages/15-control.html#conditional-statements

# Iteration

# Iteration: While Statements

*George Boole*

(Demo4)



```
i, total = 0, 0
while i < 3:
    i = i + 1
    total = total + i
```

Global frame

i ~~0~~ ~~1~~ ~~2~~ 3

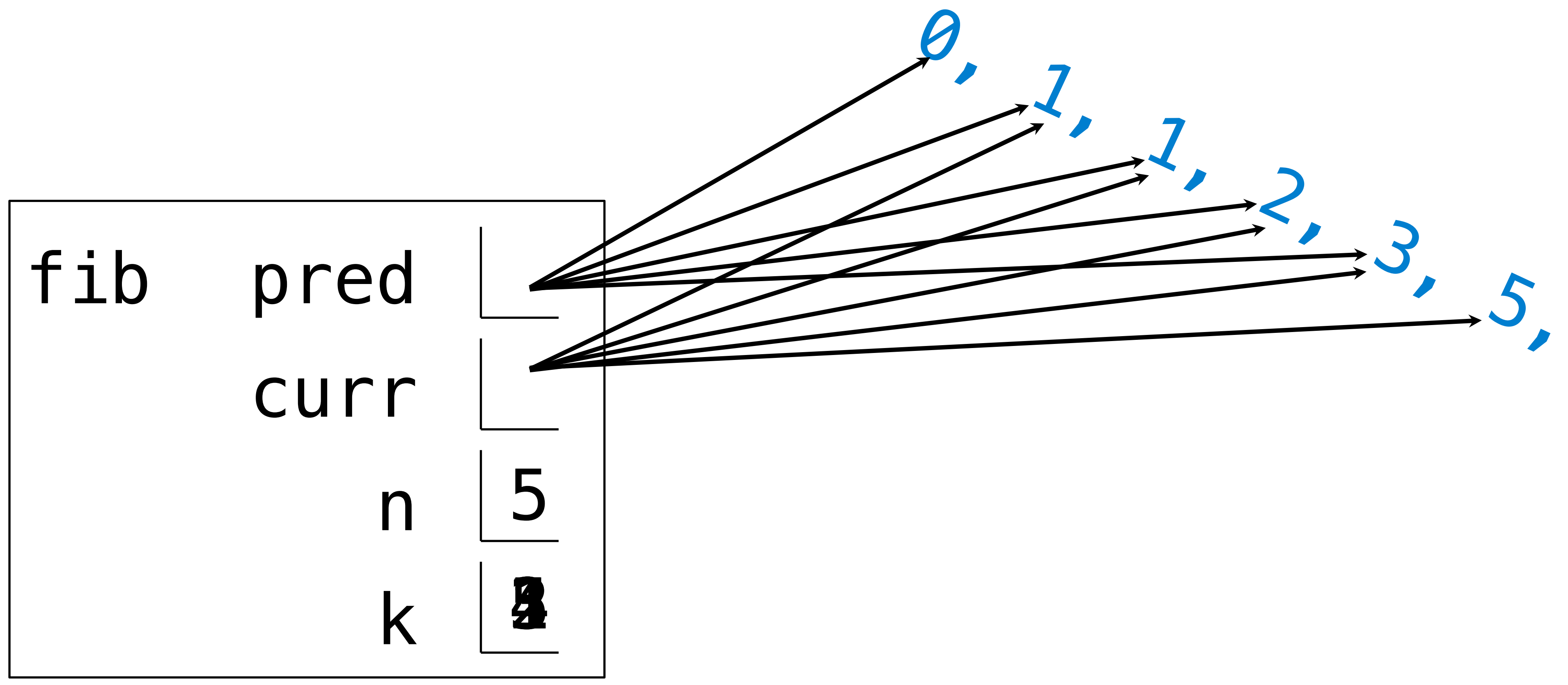
total ~~0~~ ~~1~~ ~~3~~ 6

**Execution Rule for While Statements:**

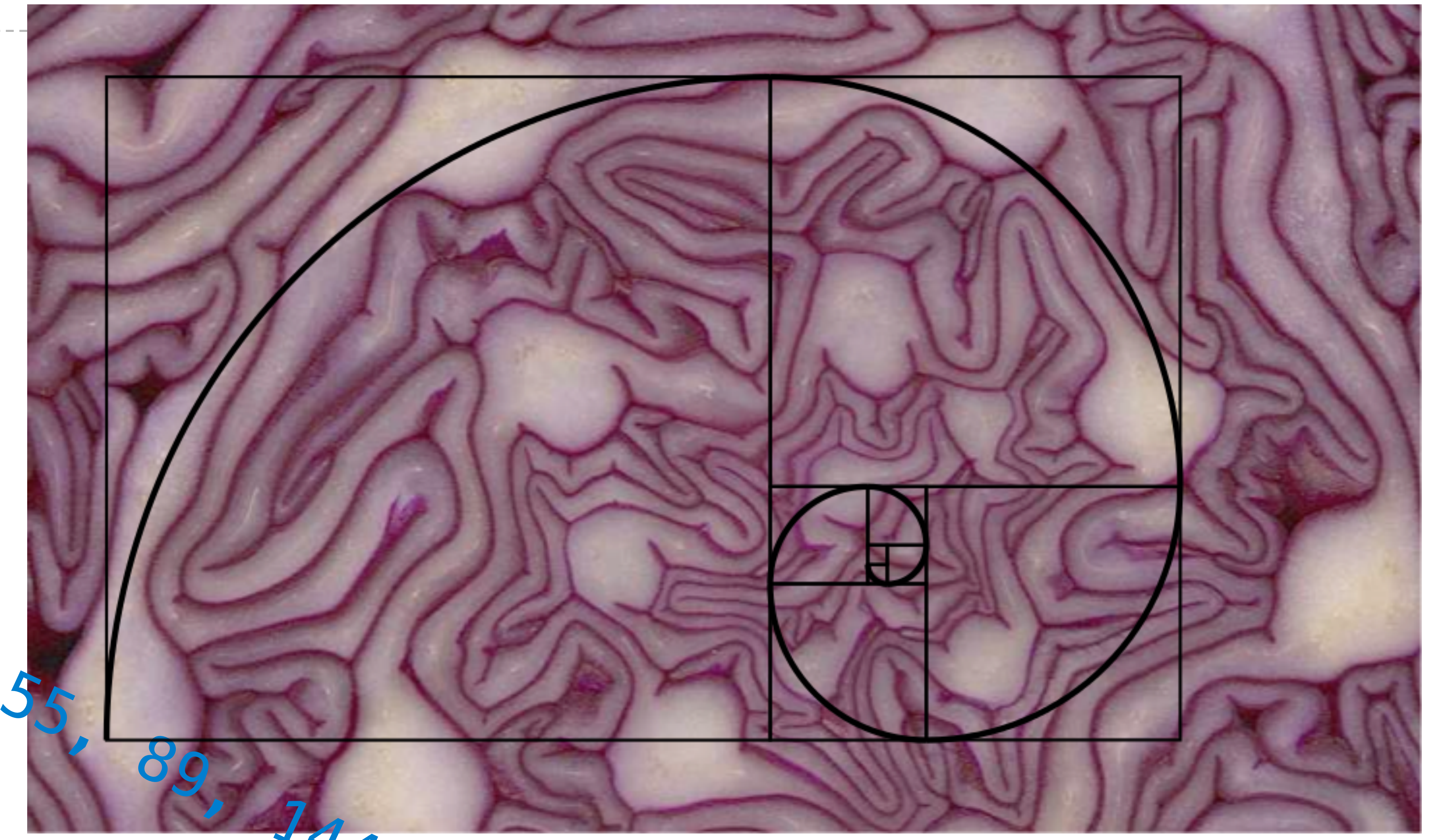
1. Evaluate the header’s expression.

2. If it is a true value,
   execute the (whole) suite,
   then return to step 1.

# The Fibonacci Sequence



0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987



```
def fib(n):
    """Compute the nth Fibonacci number, for N >= 1."""
    pred, curr = 0, 1   # 0th and 1st Fibonacci numbers
    k = 1               # curr is the kth Fibonacci number
    while k < n:
        pred, curr = curr, pred + curr
        k = k + 1
    return curr
```

The next Fibonacci number is the sum of the current one and its predecessor

# Designing Functions

# Describing Functions

```
def square(x):
    """Return X * X."""
```

A function's *domain* is the set of all inputs it might possibly take as arguments.

*x is a number*

A function's *range* is the set of output values it might possibly return.

*square returns a non-negative real number*

A pure function's *behavior* is the relationship it creates between input and output.

*square returns the square of x*

# A Guide to Designing Function… Generalization!

Give each function exactly one job, but make it apply to many related situations

```
>>> round(1.23)      >>> round(1.23, 1)      >>> round(1.23, 0)      >>> round(1.23, 5)
1                    1.2                     1                       1.23
```

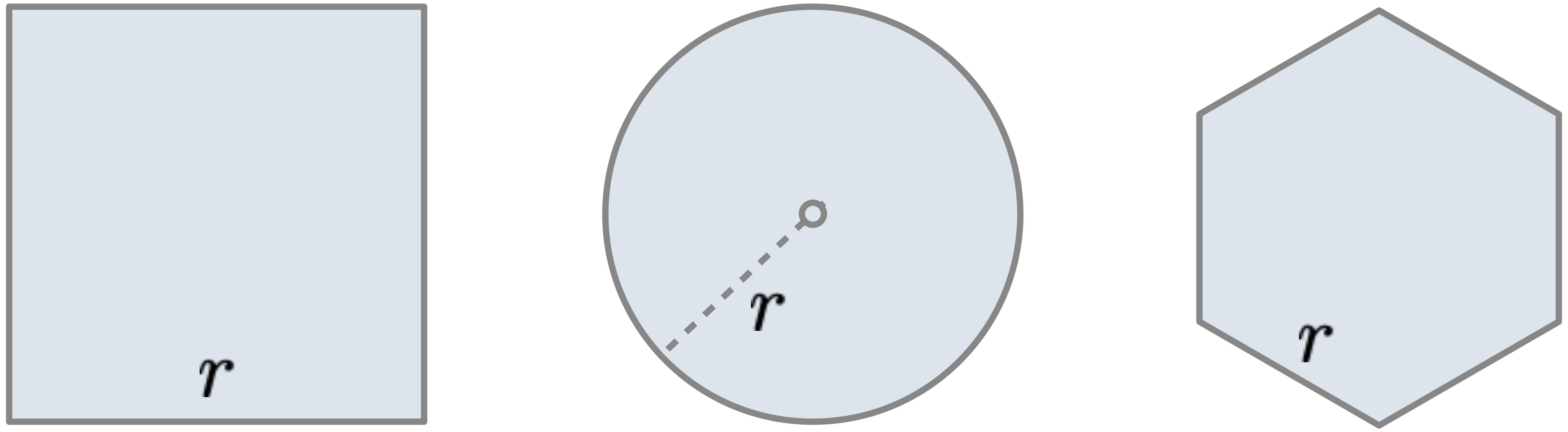
Don’t repeat yourself (DRY).  Implement a process just once, but execute it many times.

# Generalization

# Generalizing Patterns with Arguments

Regular geometric shapes relate length and area.

**Shape:**



**Area:** $\boxed{1} \cdot r^2$ $\boxed{\pi} \cdot r^2$ $\boxed{\frac{3\sqrt{3}}{2}} \cdot r^2$

Finding common structure allows for shared implementation

(Demo1)

# Higher-Order Functions

# Generalizing Over Computational Processes

The common structure among functions may be a computational process, rather than a number.

$$\sum_{k=1}^{5} \boxed{k} = 1 + 2 + 3 + 4 + 5 \qquad = 15$$

$$\sum_{k=1}^{5} \boxed{k^3} = 1^3 + 2^3 + 3^3 + 4^3 + 5^3 \qquad = 225$$

$$\sum_{k=1}^{5} \boxed{\frac{8}{(4k-3)\cdot(4k-1)}} = \frac{8}{3} + \frac{8}{35} + \frac{8}{99} + \frac{8}{195} + \frac{8}{323} \qquad = 3.04$$

(Demo2)

# Summation Example

```
def cube(k):
    return pow(k, 3)

def summation(n, term):
    """Sum the first n terms of a sequence.

    >>> summation(5, cube)
    225
    """
    total, k = 0, 1
    while k <= n:
        total, k = total + term(k), k + 1
    return total
```

Function of a single argument
(*not called "term"*)

A formal parameter that will
be bound to a function

The cube function is passed
as an argument value

0 + 1 + 8 + 27 + 64 + 125

The function bound to term
gets called here

# Functions as Return Values

```
(Demo3)
```

# Locally Defined Functions

Functions defined within other function bodies are bound to names in a local frame

A function that returns a function

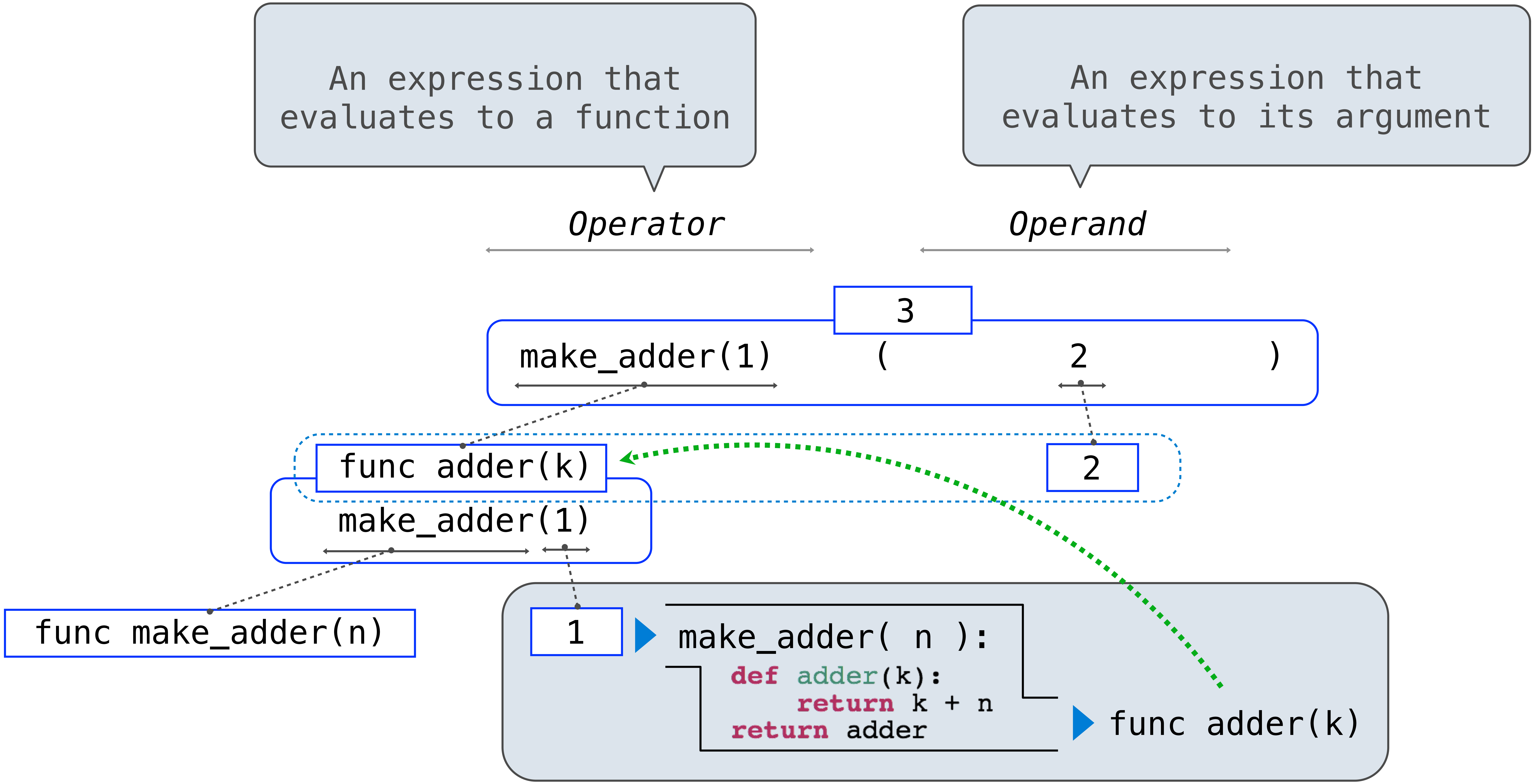
```
def make_adder(n):
    """Return a function that takes one argument k and returns k + n.

    >>> add_three = make_adder(3)
    >>> add_three(4)
    7
    """
    def adder(k):
        return k + n
    return adder
```

The name add_three is bound to a function

A def statement within another def statement

Can refer to names in the enclosing function

# Call Expressions as Operator Expressions



An expression that evaluates to a function

An expression that evaluates to its argument

*Operator*

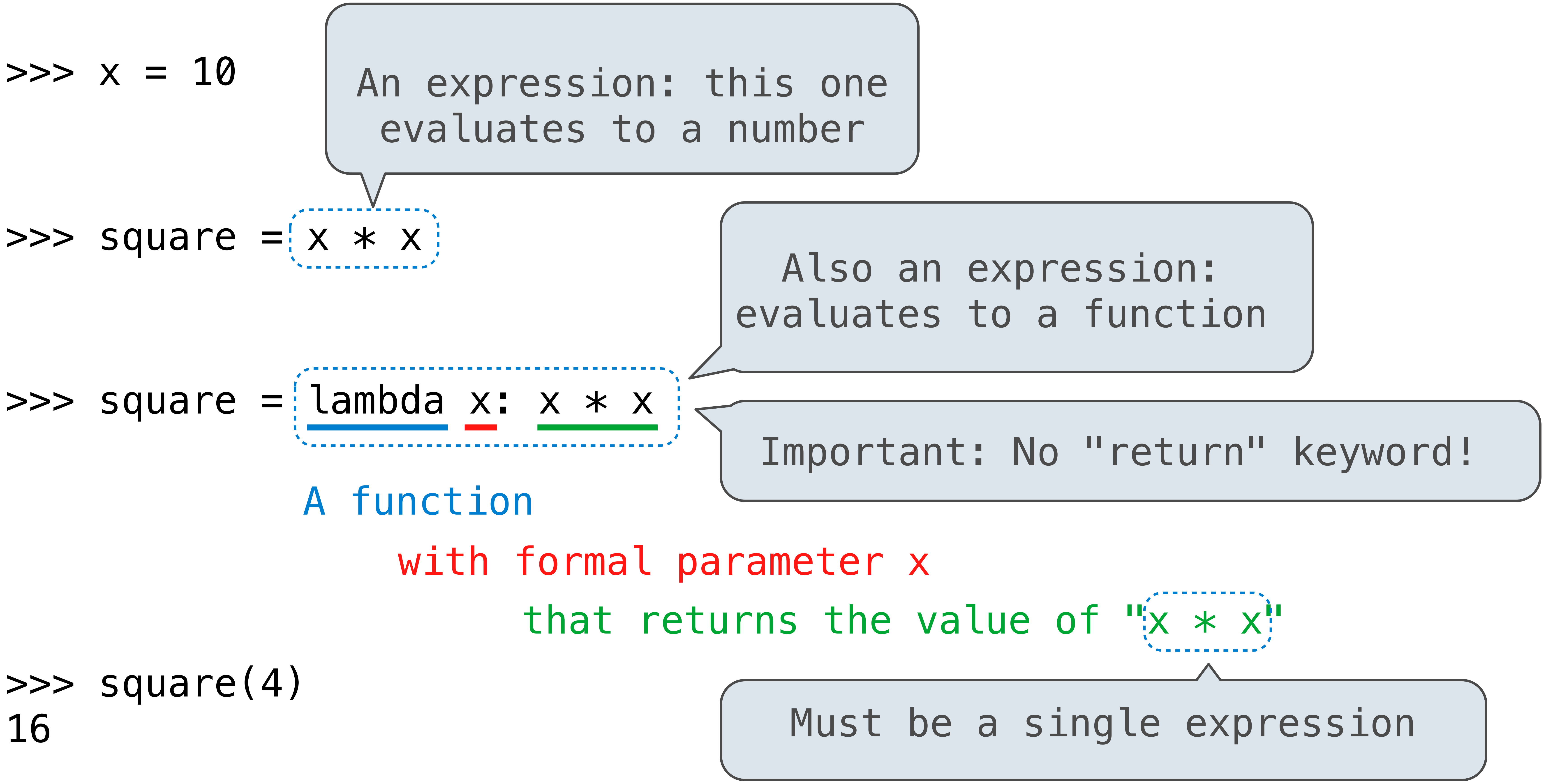
*Operand*

# Lambda Expressions

λ

(Demo4)

# Lambda Expressions



```
>>> x = 10

>>> square = x * x

>>> square = lambda x: x * x

>>> square(4)
16
```

An expression: this one evaluates to a number

Also an expression: evaluates to a function

Important: No "return" keyword!

A function

with formal parameter x

that returns the value of "x * x"

Must be a single expression

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

Lambda expressions in Python cannot contain statements at all!

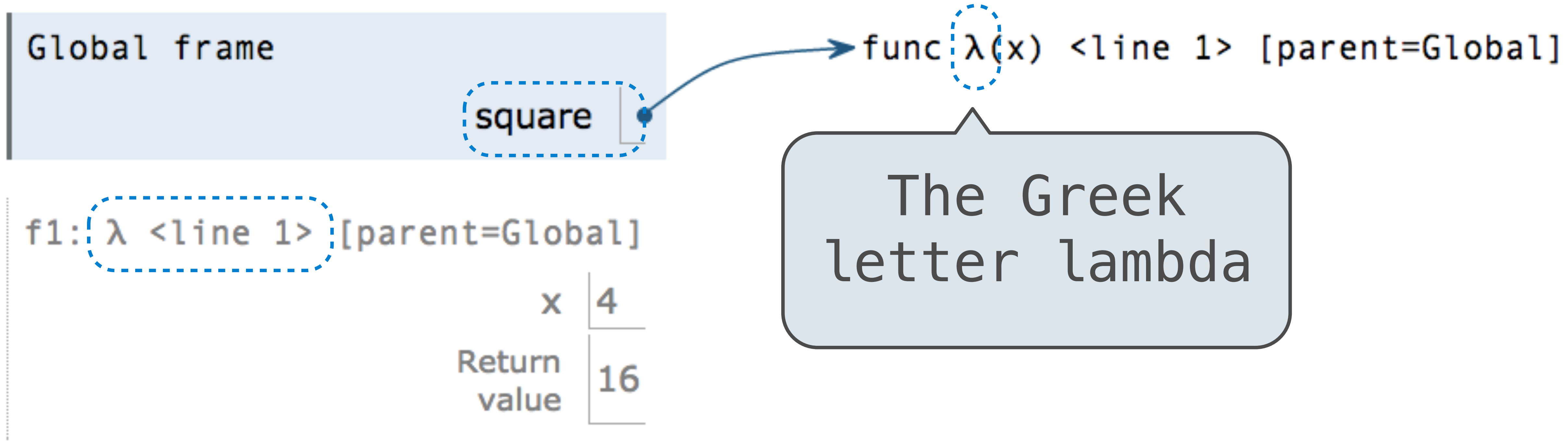
# Lambda Expressions Versus Def Statements

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

**VS**

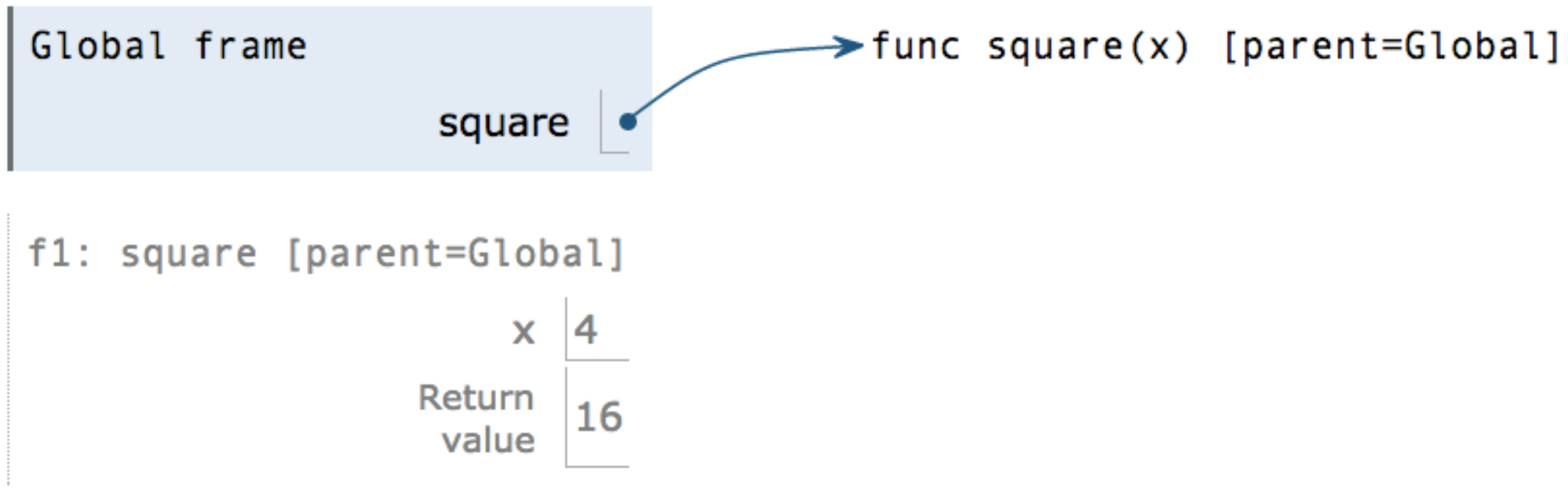
```
def square(x):
    return x * x
```

- Both create a function with the same domain, range, and behavior.
- Both bind that function to the name square.
- Only the def statement gives the function an intrinsic name, which shows up in environment diagrams but doesn't affect execution (unless the function is printed).



```
Global frame                 func λ(x) <line 1> [parent=Global]
                  square

f1: λ <line 1> [parent=Global]
                       x  4
                  Return
                   value  16
```

The Greek letter lambda



```
Global frame                 func square(x) [parent=Global]
                  square

f1: square [parent=Global]
                       x  4
                  Return
                   value  16
```