

Environments

Announcements

Expressions

Types of expressions

Types of expressions

An expression describes a computation and evaluates to a value

Types of expressions

An expression describes a computation and evaluates to a value

$$18 + 69$$

Types of expressions

An expression describes a computation and evaluates to a value

$$18 + 69$$

$$\frac{6}{23}$$

Types of expressions

An expression describes a computation and evaluates to a value

$$18 + 69$$

$$\frac{6}{23}$$

$$\sqrt{3493161}$$

Types of expressions

An expression describes a computation and evaluates to a value

$$18 + 69$$

$$\frac{6}{23}$$

$$\sin \pi$$

$$\sqrt{3493161}$$

Types of expressions

An expression describes a computation and evaluates to a value

$$18 + 69$$

$$\frac{6}{23}$$

$$\sin \pi$$

$$\sqrt{3493161}$$

$$| - 1869 |$$

Types of expressions

An expression describes a computation and evaluates to a value

$$18 + 69$$

$$\frac{6}{23}$$

$$\sin \pi$$

$$\sqrt{3493161}$$

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

$$|-1869|$$

Types of expressions

An expression describes a computation and evaluates to a value

$$18 + 69$$

$$\frac{6}{23}$$

$$\sin \pi$$

$$\sqrt{3493161}$$

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

$$|-1869|$$

$$\binom{69}{18}$$

Types of expressions

An expression describes a computation and evaluates to a value

$$18 + 69$$

$$\frac{6}{23}$$

$$\sin \pi$$

$$f(x)$$

$$\sqrt{3493161}$$

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

$$|-1869|$$

$$\binom{69}{18}$$

Types of expressions

An expression describes a computation and evaluates to a value

$$18 + 69$$

$$\frac{6}{23}$$

$$\sin \pi$$

$$2^{100}$$

$$f(x)$$

$$\sqrt{3493161}$$

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

$$|-1869|$$

$$\binom{69}{18}$$

Types of expressions

An expression describes a computation and evaluates to a value

$$18 + 69$$

$$\frac{6}{23}$$

$$\sin \pi$$

$$\log_2 1024$$

$$2^{100}$$

$$f(x)$$

$$\sqrt{3493161}$$

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

$$|-1869|$$

$$\binom{69}{18}$$

Types of expressions

An expression describes a computation and evaluates to a value

$$18 + 69$$

$$\frac{6}{23}$$

$$\sin \pi$$

$$\log_2 1024$$

$$2^{100}$$

$$f(x)$$

$$\sqrt{3493161}$$

$$7 \bmod 2$$

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

$$\binom{69}{18}$$

$$|-1869|$$

Types of expressions

An expression describes a computation and evaluates to a value

$$18 + 69$$

$$\frac{6}{23}$$

$$\sin \pi$$

$$\log_2 1024$$

$$2^{100}$$

$$f(x)$$

$$\sqrt{3493161}$$

$$7 \bmod 2$$

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

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

$$|-1869|$$

$$\binom{69}{18}$$

Types of expressions

An expression describes a computation and evaluates to a value

$$18 + 69$$

$$\frac{6}{23}$$

$$\sin \pi$$

$$\log_2 1024$$

$$2^{100}$$

$$f(x)$$

$$\sqrt{3493161}$$

$$7 \bmod 2$$

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

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

$$|-1869|$$

$$\binom{69}{18}$$

Types of expressions

An expression describes a computation and evaluates to a value

$$18 + 69$$

$$\frac{6}{23}$$

$$\sin \pi$$

$$\log_2 1024$$

$$2^{100}$$

$$f(x)$$

$$\sqrt{3493161}$$

$$7 \bmod 2$$

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

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

$$|-1869|$$

$$\binom{69}{18}$$

(Demo)

Anatomy of a Call Expression

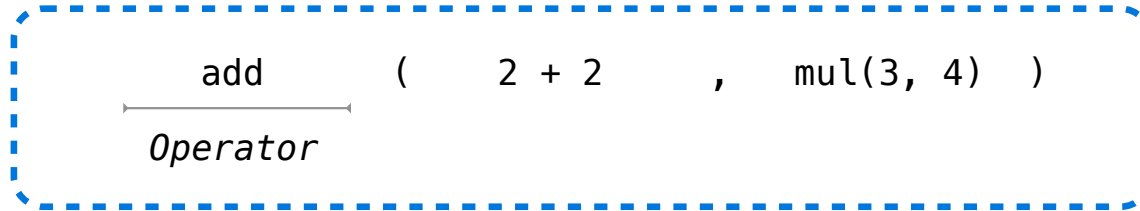
Anatomy of a Call Expression

```
add      (    2 + 2    ,    mul(3, 4)  )
```

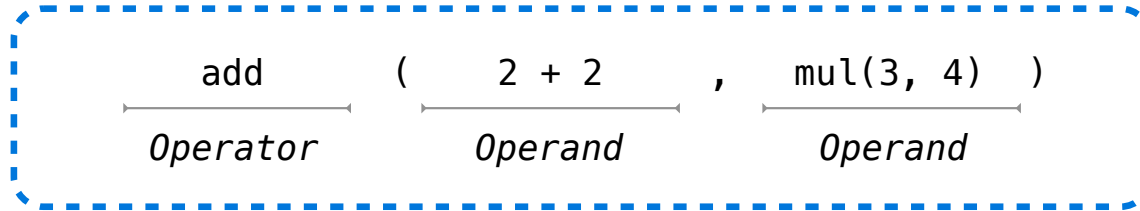
Anatomy of a Call Expression

```
add    (    2 + 2    ,    mul(3, 4)    )
```

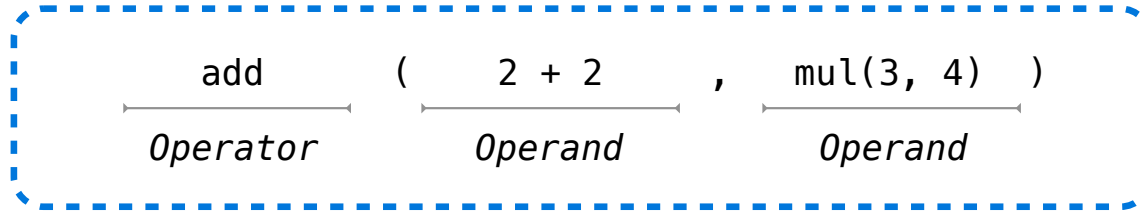
Anatomy of a Call Expression



Anatomy of a Call Expression

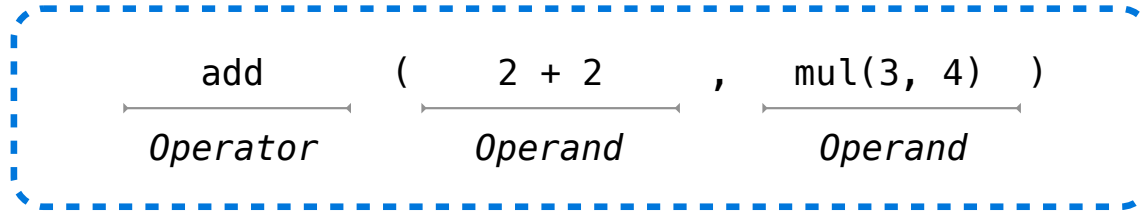


Anatomy of a Call Expression



Operators and operands are also expressions

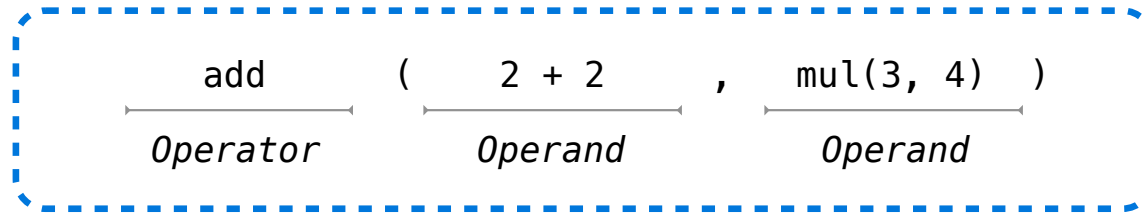
Anatomy of a Call Expression



Operators and operands are also expressions

So they evaluate to values

Anatomy of a Call Expression

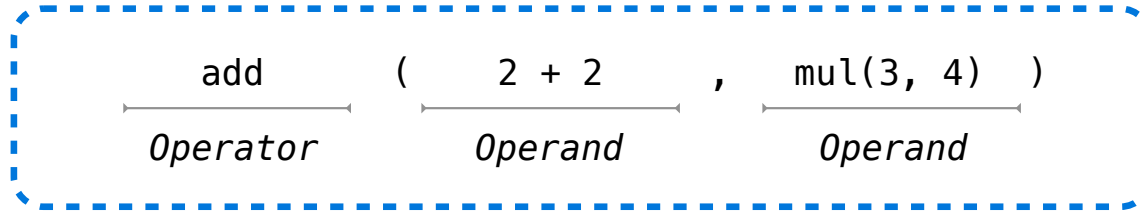


Operators and operands are also expressions

So they evaluate to values

Evaluation procedure for call expressions:

Anatomy of a Call Expression



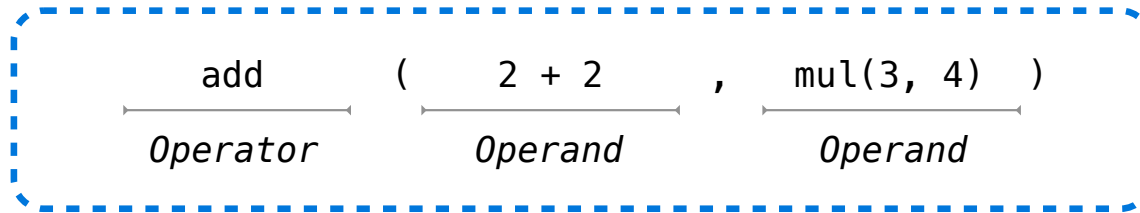
Operators and operands are also expressions

So they evaluate to values

Evaluation procedure for call expressions:

1. Evaluate the operator and then the operand subexpressions

Anatomy of a Call Expression



Operators and operands are also expressions

So they evaluate to values

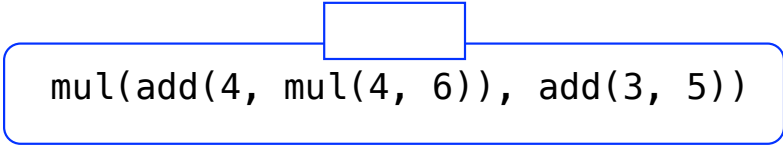
Evaluation procedure for call expressions:

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

```
mul(add(4, mul(4, 6)), add(3, 5))
```

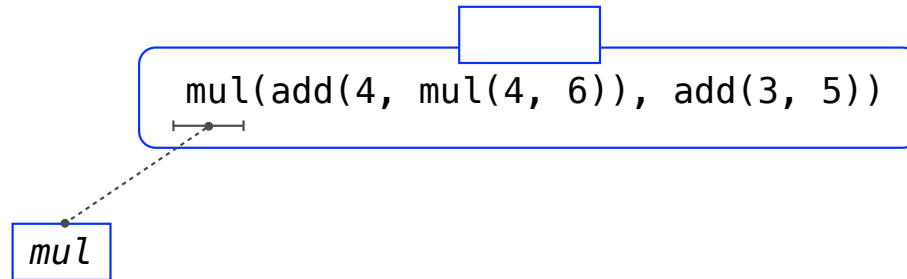
Evaluating Nested Expressions



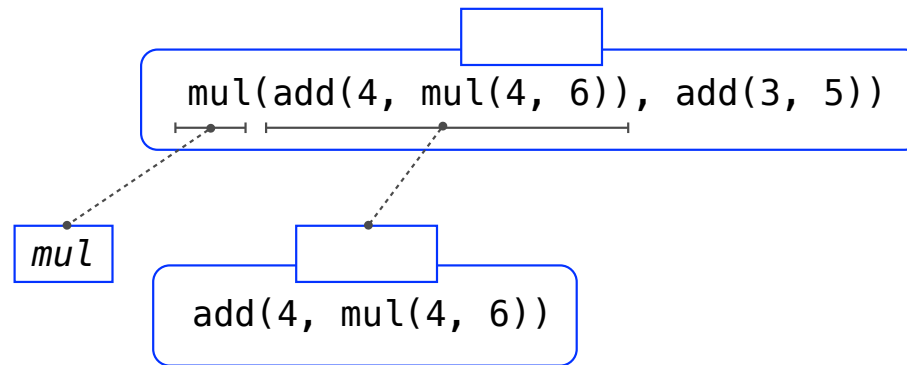
The diagram shows the expression `mul(add(4, mul(4, 6)), add(3, 5))` enclosed in a rounded rectangular box. A smaller, empty rectangular box is positioned above the `mul(4, 6)` sub-expression, indicating the current step in the evaluation process.

```
mul(add(4, mul(4, 6)), add(3, 5))
```

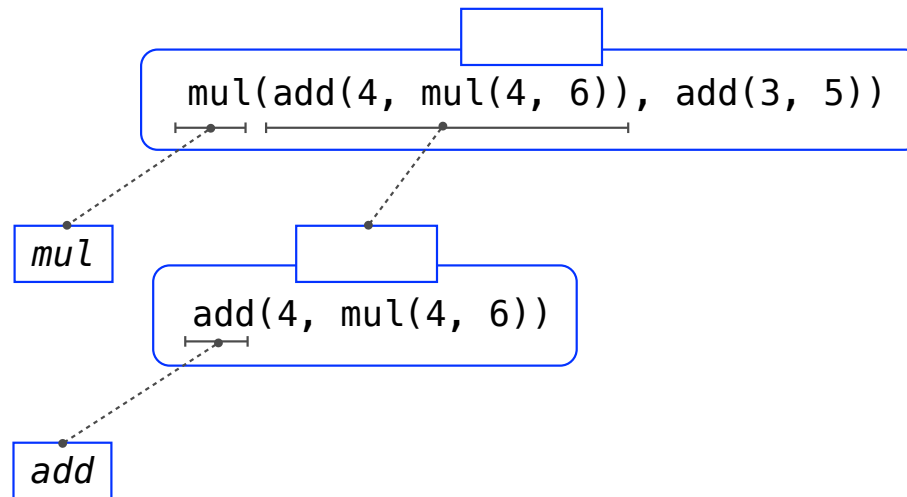
Evaluating Nested Expressions



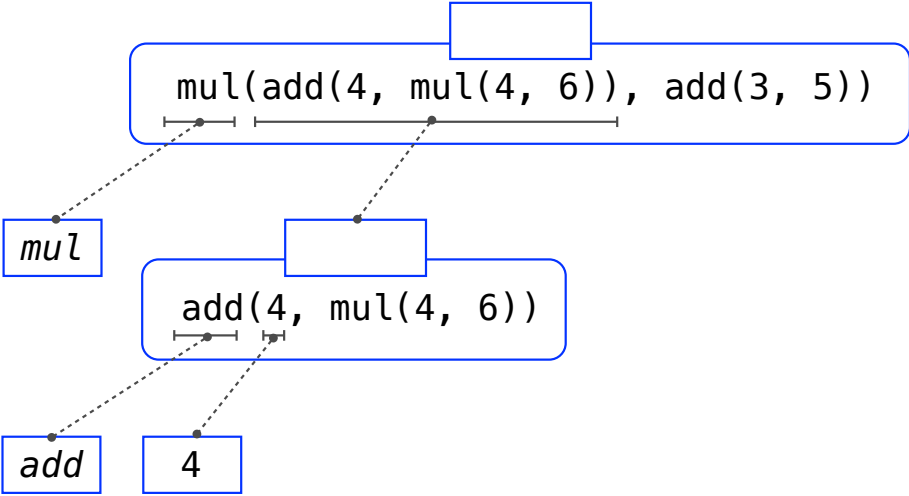
Evaluating Nested Expressions



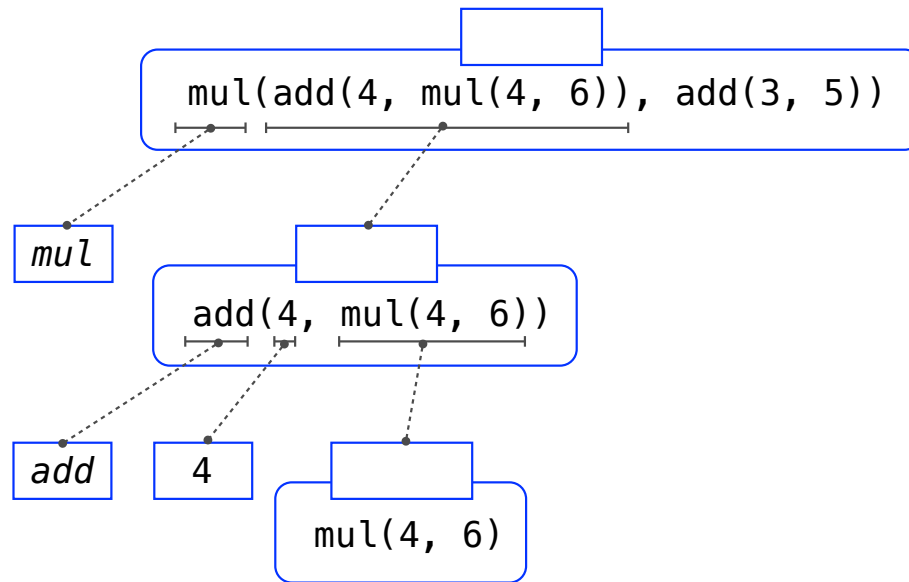
Evaluating Nested Expressions



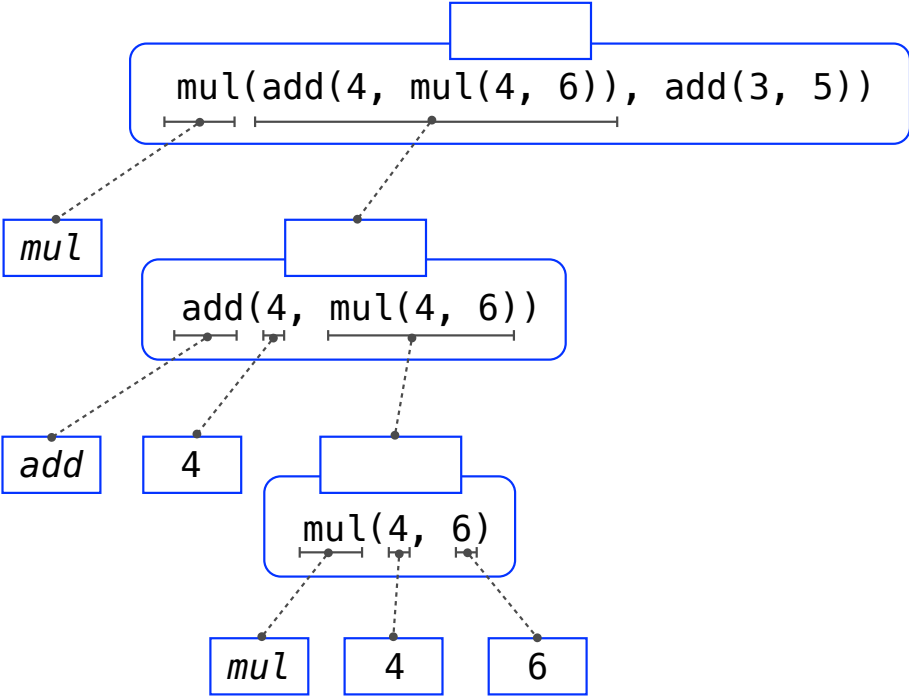
Evaluating Nested Expressions



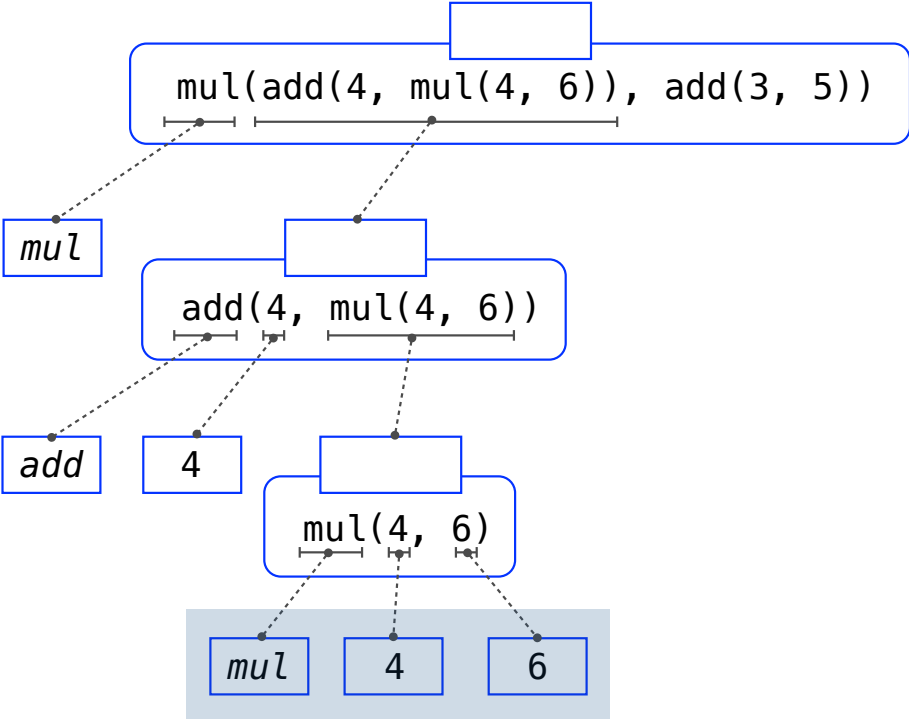
Evaluating Nested Expressions



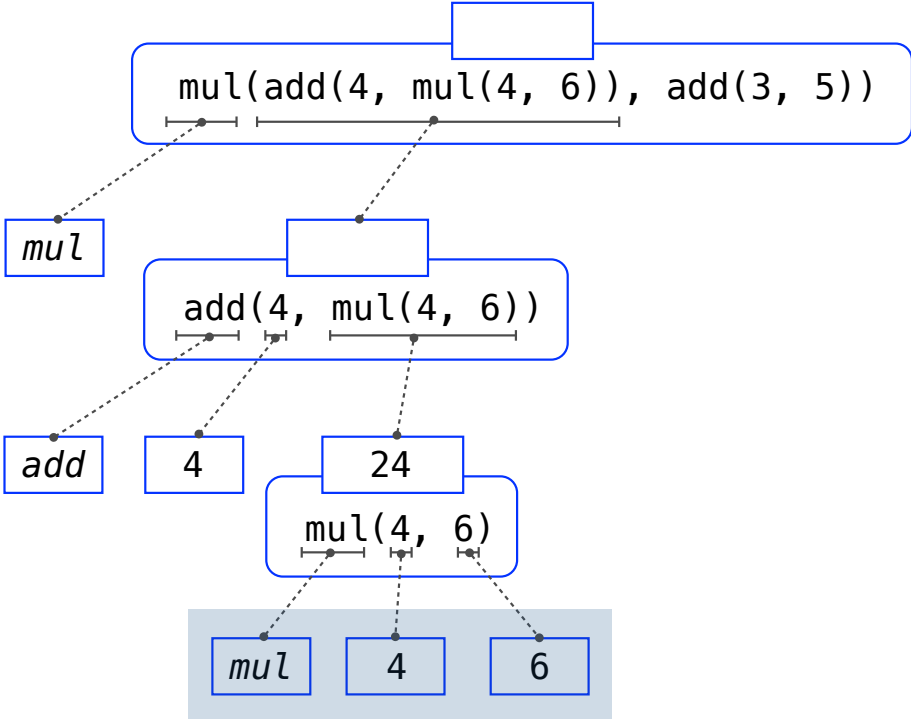
Evaluating Nested Expressions



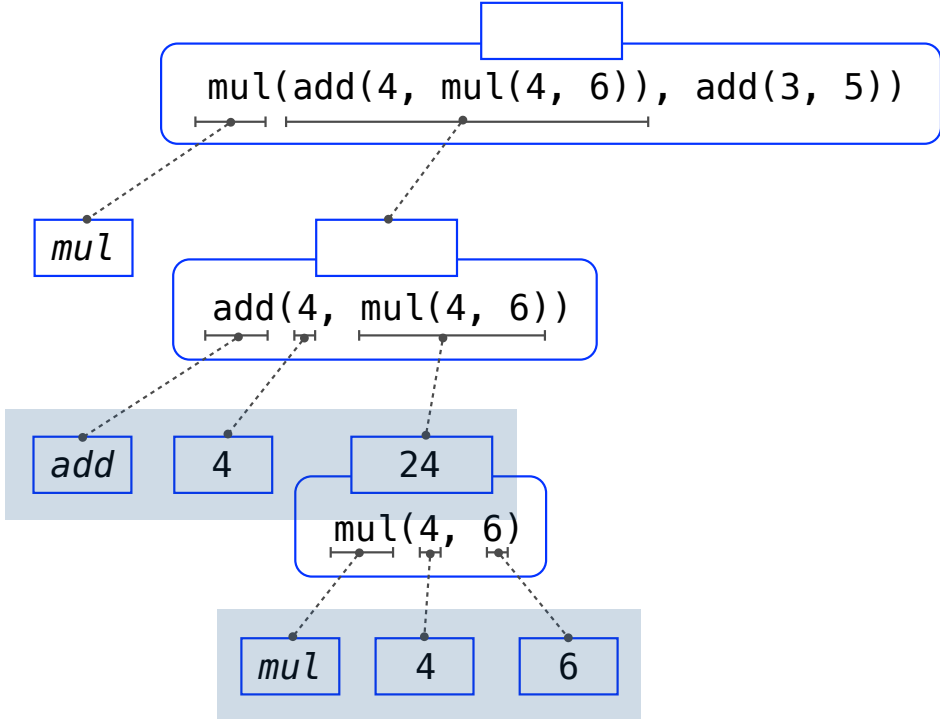
Evaluating Nested Expressions



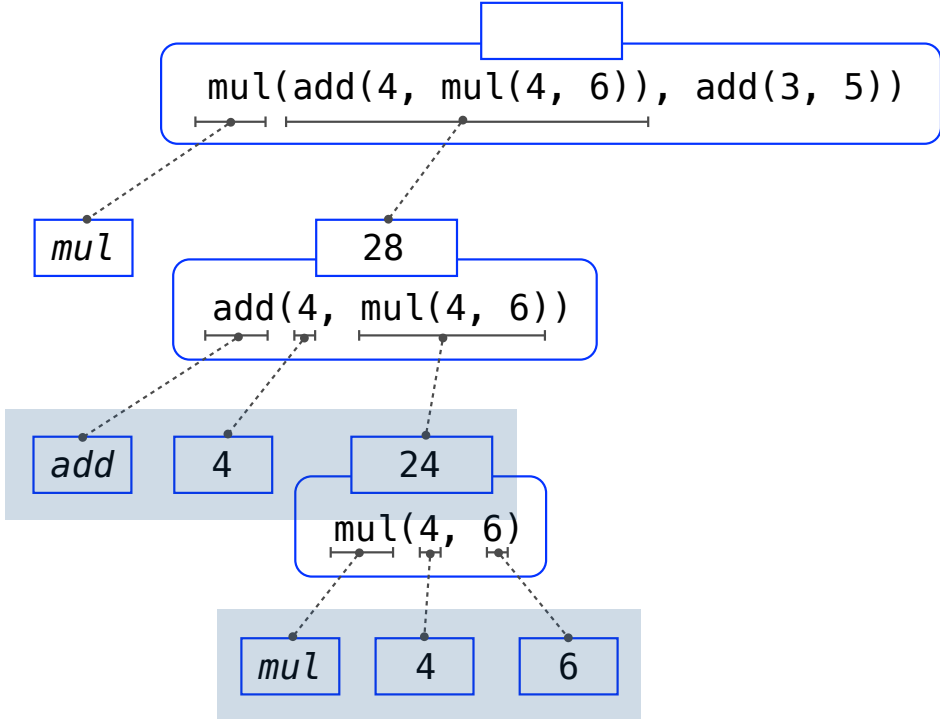
Evaluating Nested Expressions



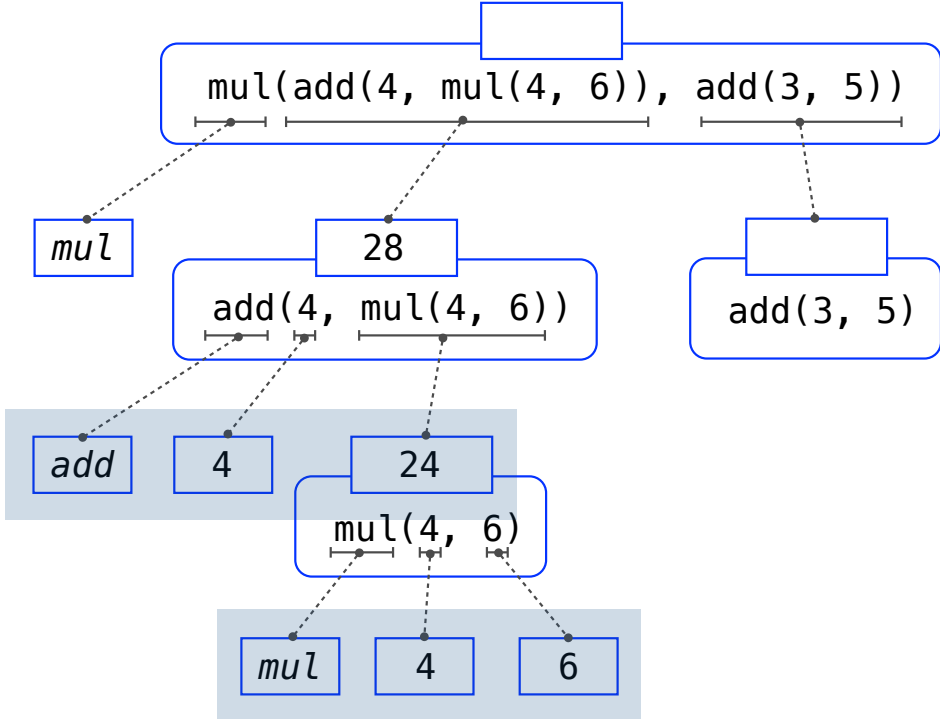
Evaluating Nested Expressions



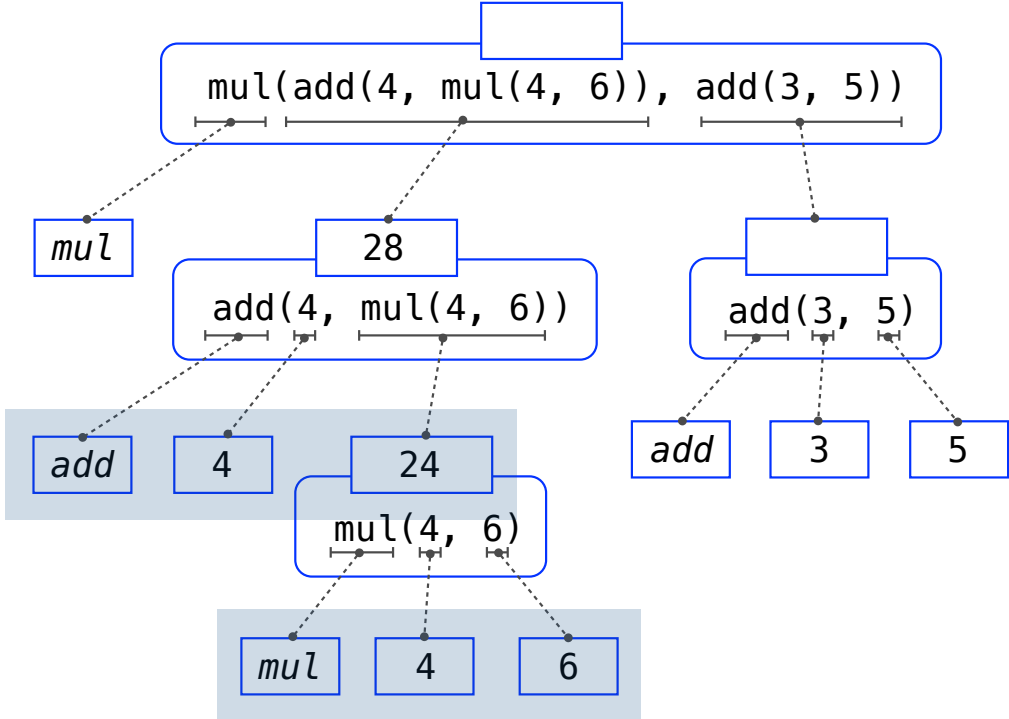
Evaluating Nested Expressions



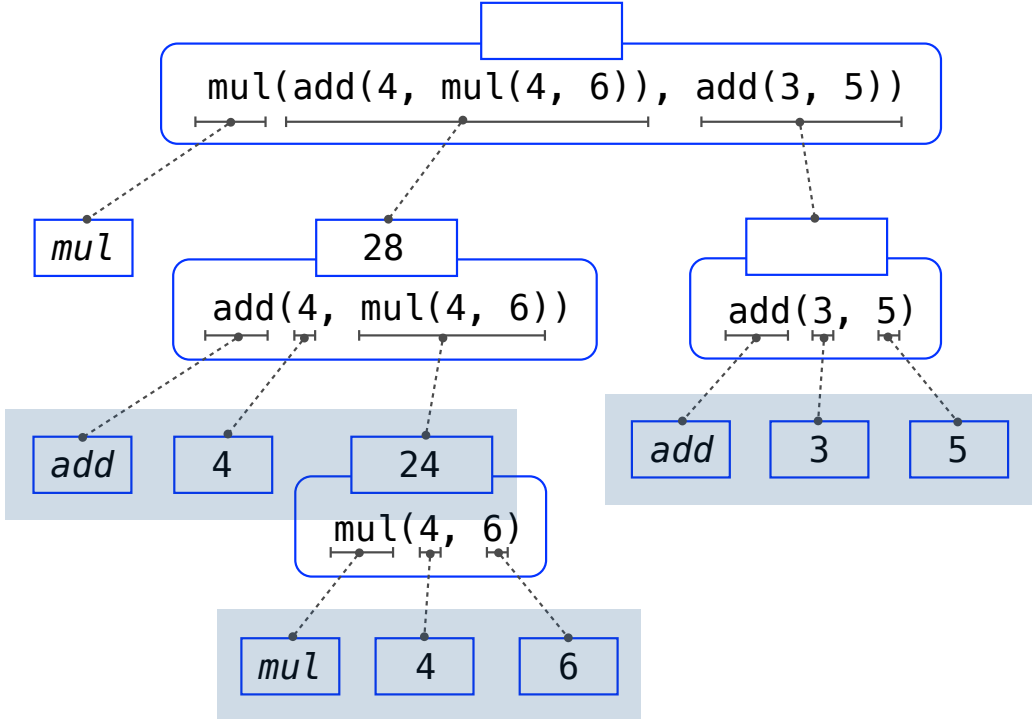
Evaluating Nested Expressions



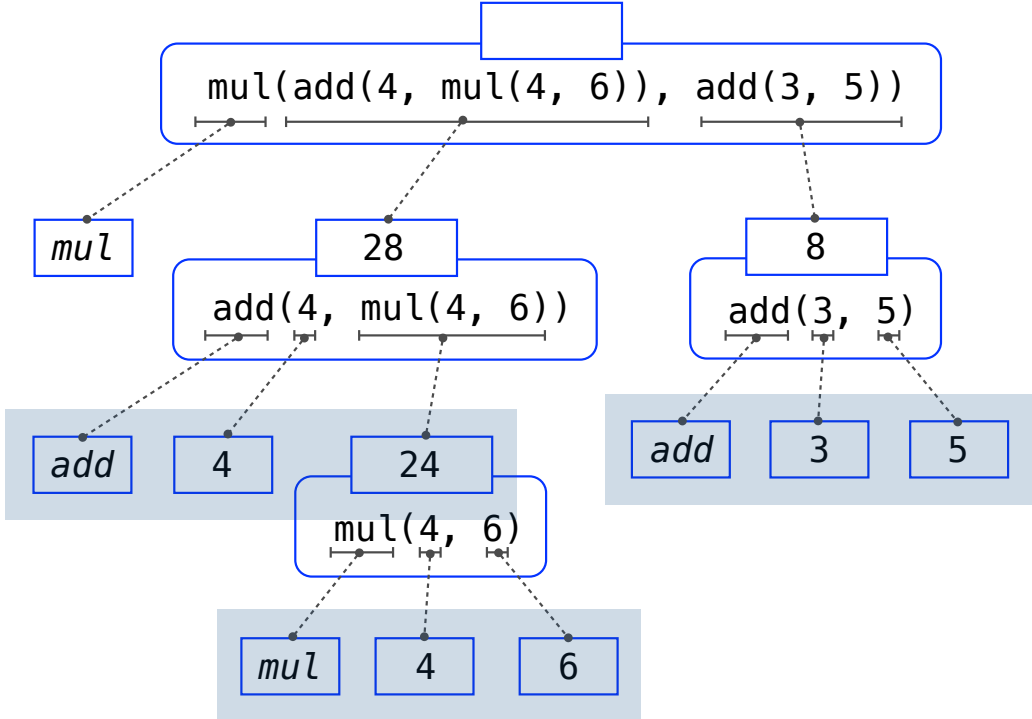
Evaluating Nested Expressions



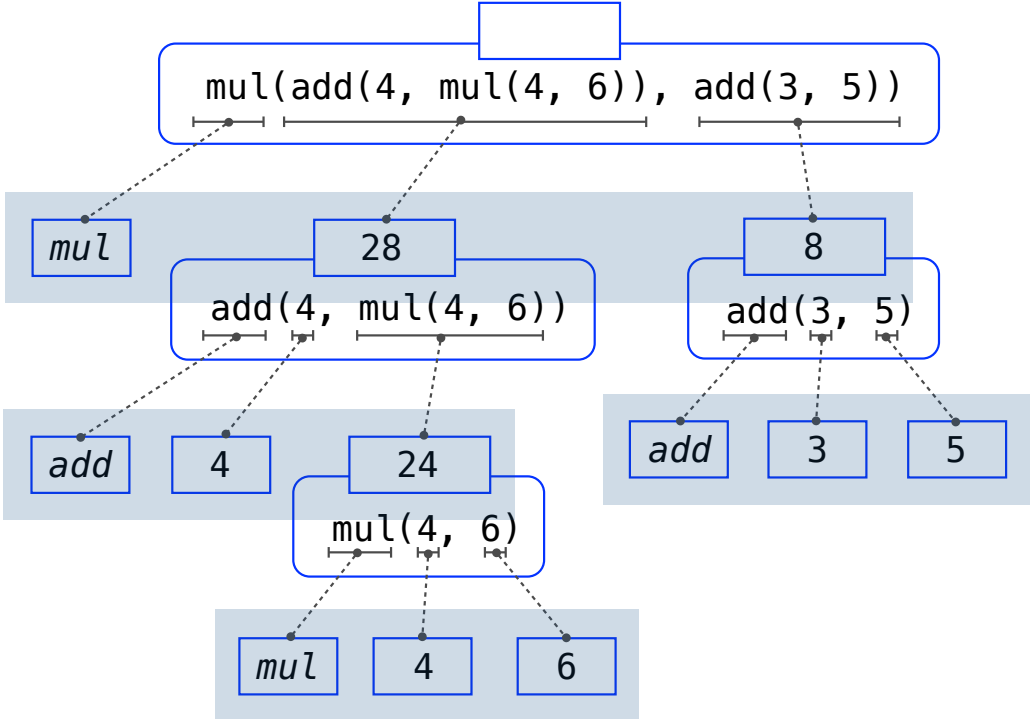
Evaluating Nested Expressions



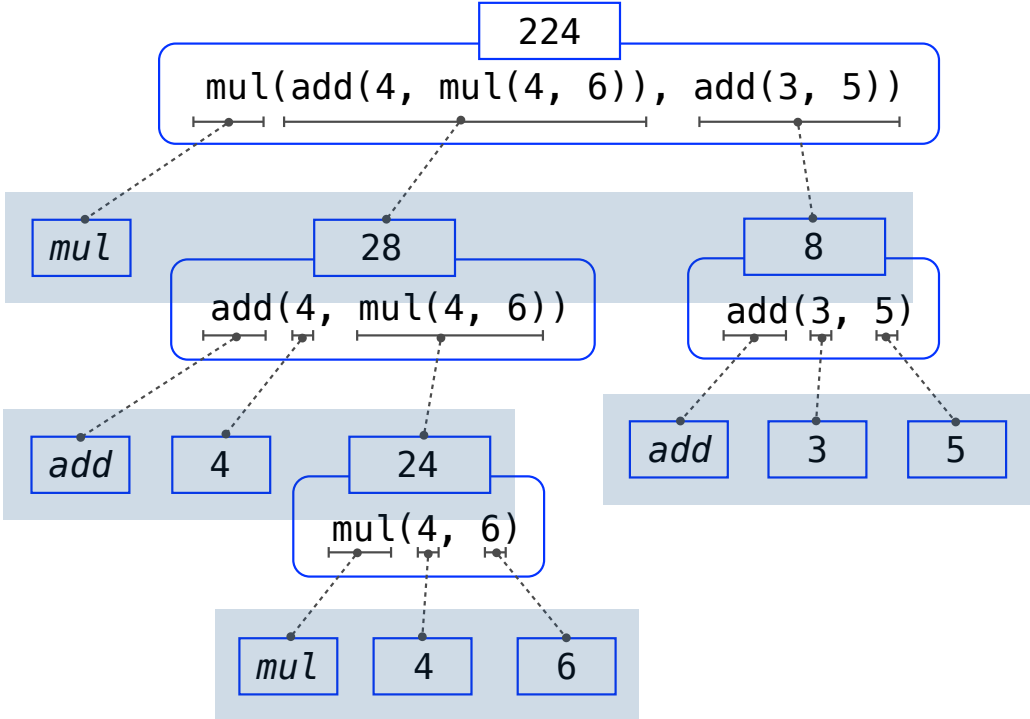
Evaluating Nested Expressions



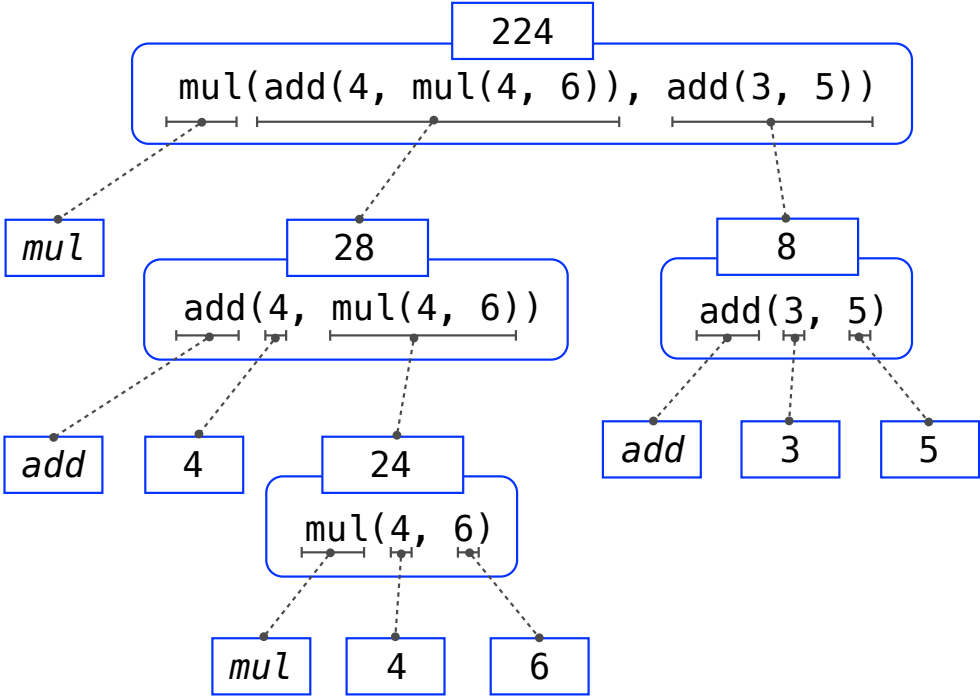
Evaluating Nested Expressions



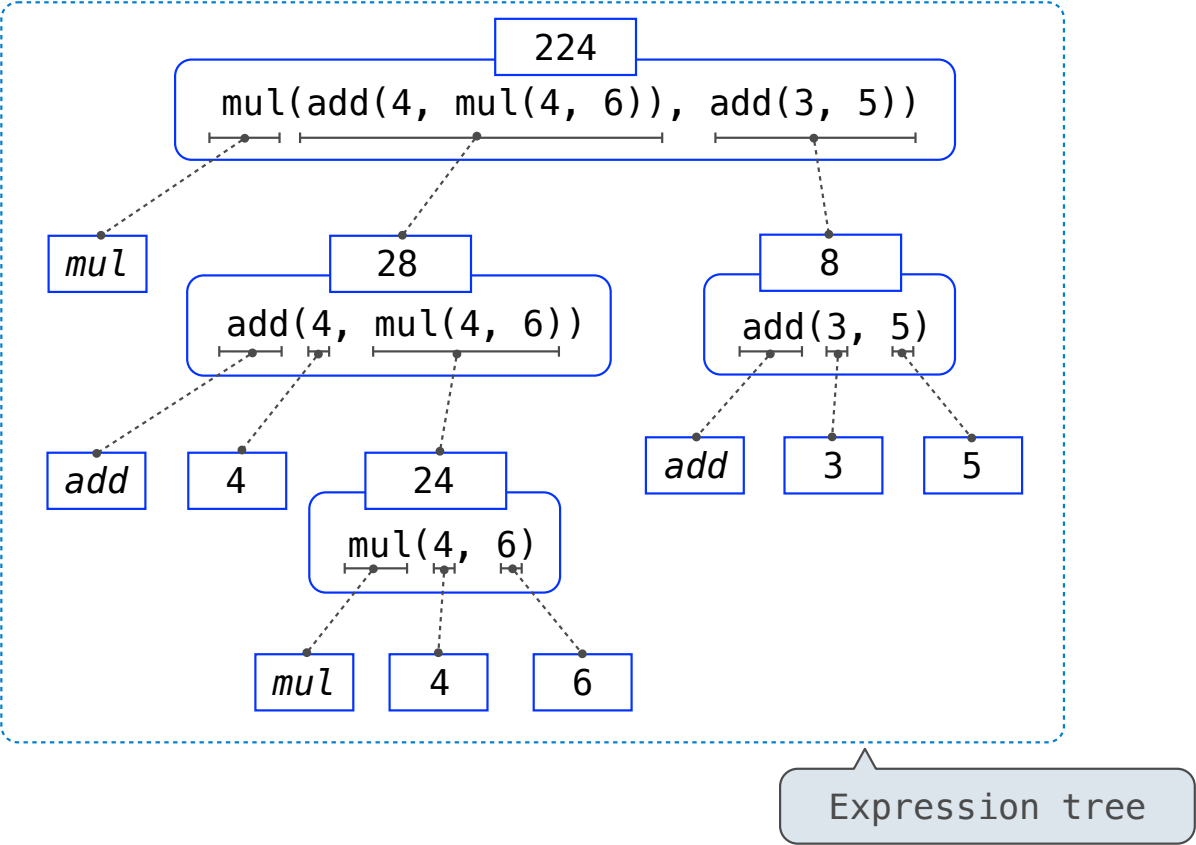
Evaluating Nested Expressions



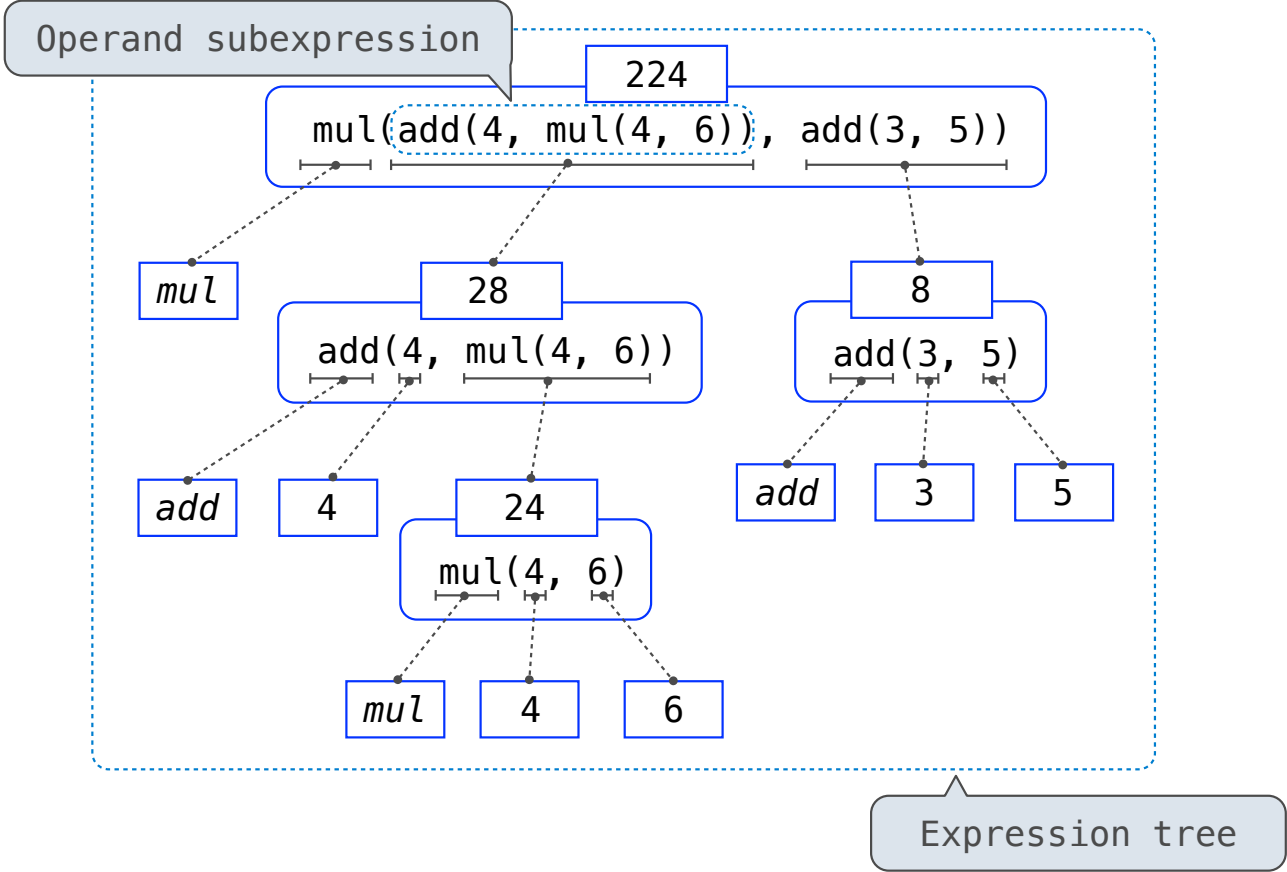
Evaluating Nested Expressions



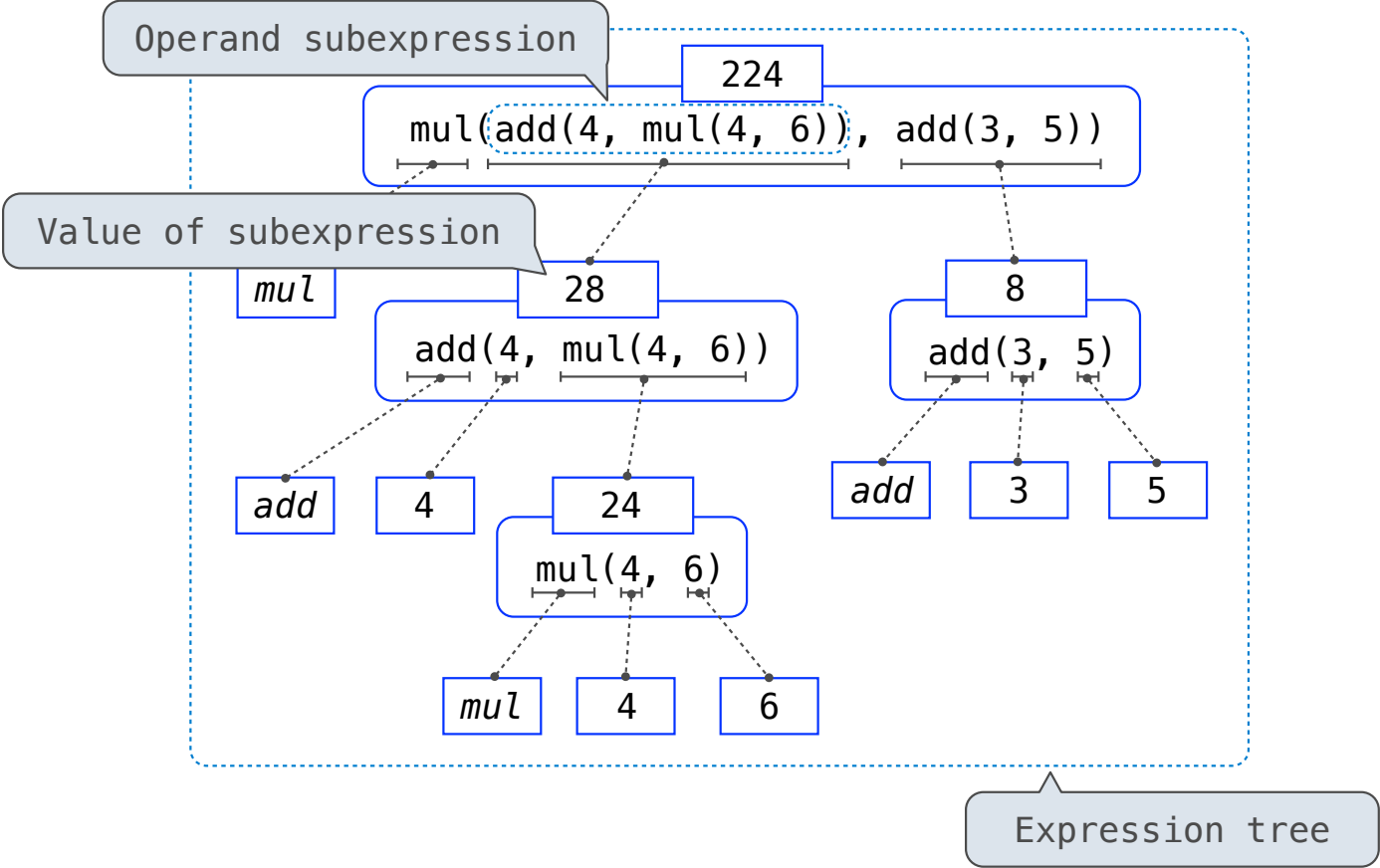
Evaluating Nested Expressions



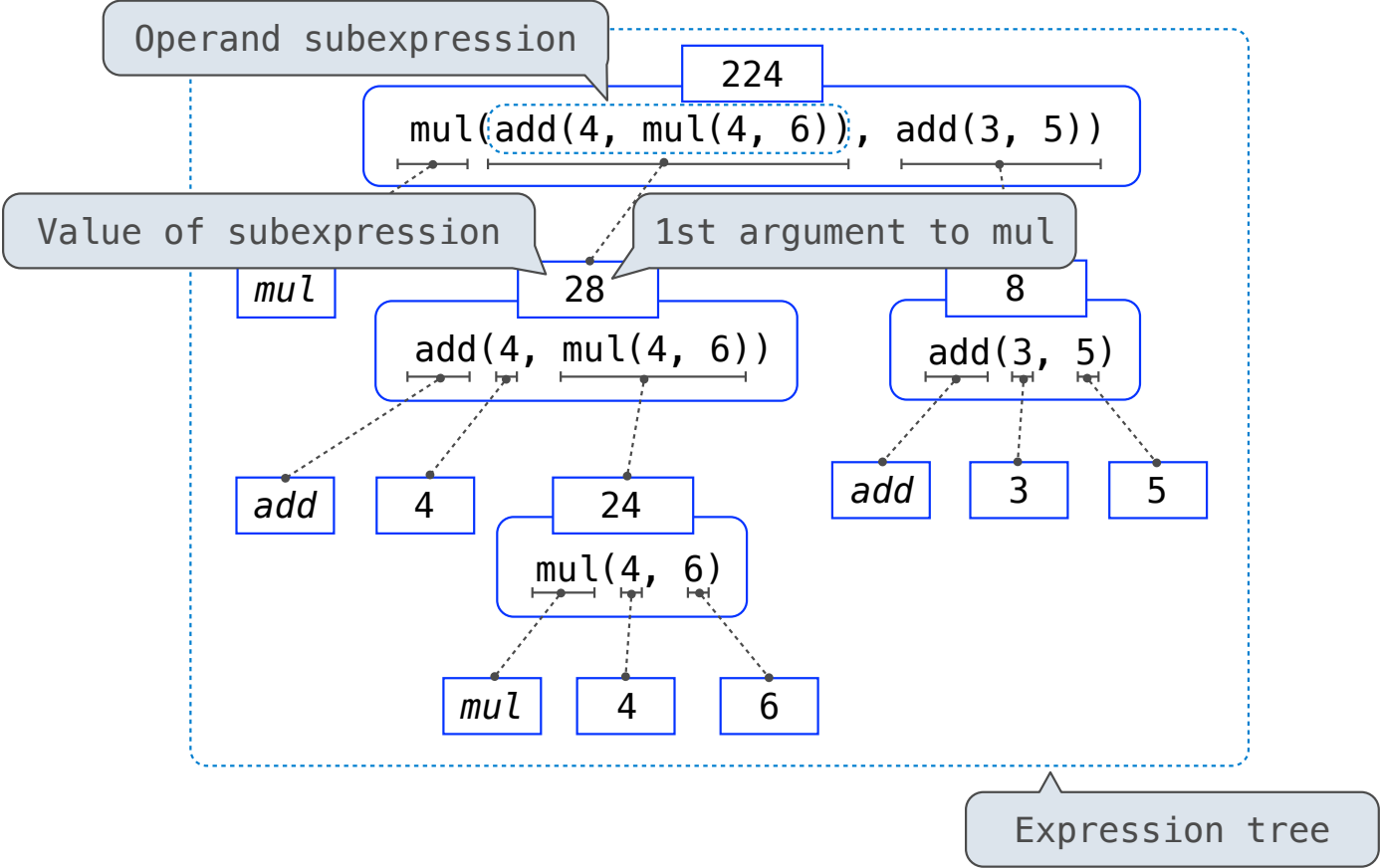
Evaluating Nested Expressions



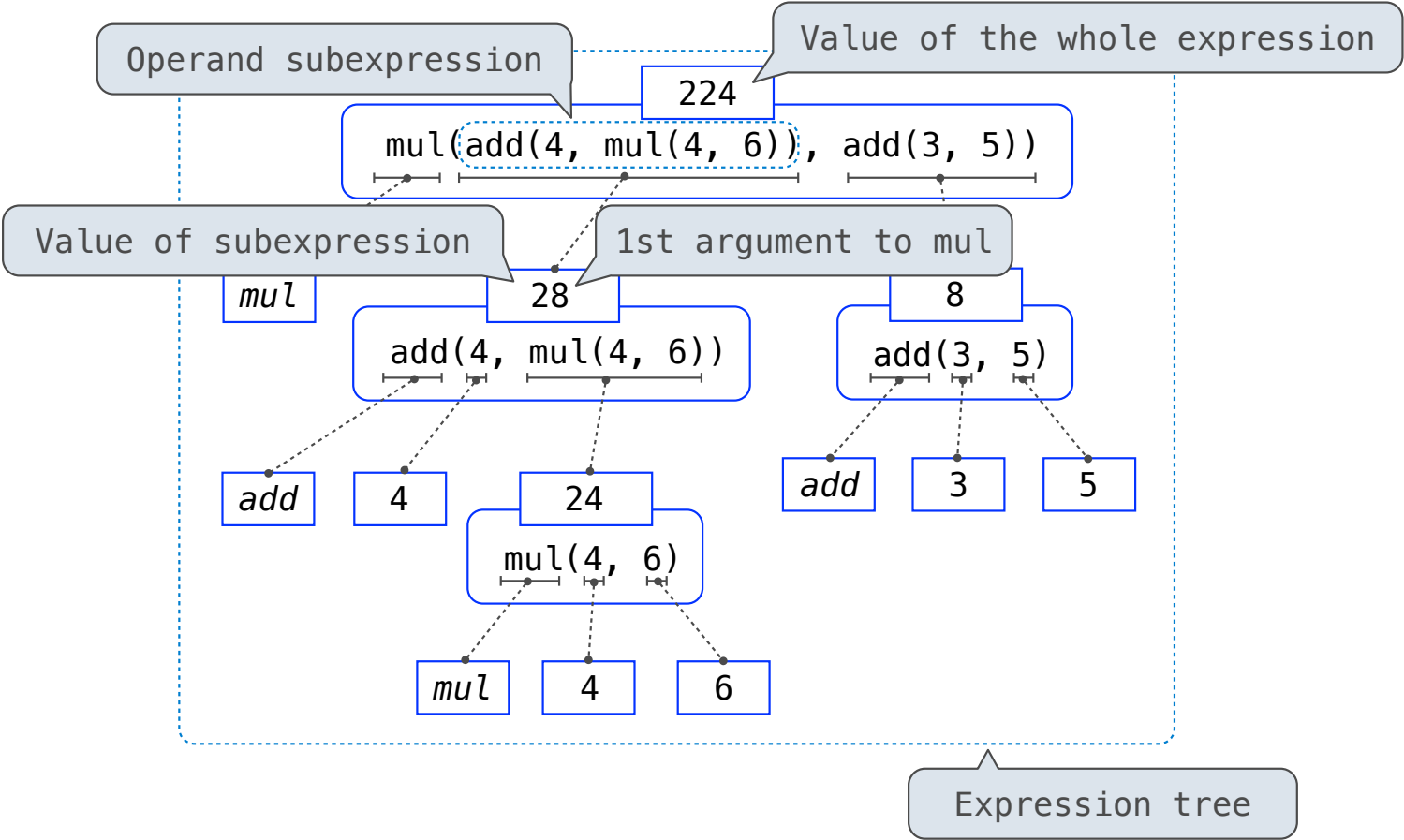
Evaluating Nested Expressions



Evaluating Nested Expressions



Evaluating Nested Expressions



Print and None

(Demo)

None Indicates that Nothing is Returned

None Indicates that Nothing is Returned

The special value **None** represents nothing in Python

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`

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

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  
... 
```

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
```



```
...     
```

The diagram shows a callout box with the text "No return" pointing to the expression "x * x" in the code above. The callout box is a light gray rounded rectangle with a black border and a black arrow pointing to the expression.


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)
```



The diagram shows a callout box with the text "No return" pointing to the end of the function body in the code snippet above. The callout box is a light gray rounded rectangle with a black border and a black arrow pointing to the right. The text "No return" is in a bold, black, sans-serif font.

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)
```

No return

None value is not displayed

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)
```

The diagram illustrates the behavior of a function that does not return a value. It shows a Python prompt where a function `does_not_return_square` is defined with a single line of code `x * x`. A callout box labeled "No return" points to the function definition. Below the definition, the function is called with the argument `4`. A second callout box labeled "None value is not displayed" points to the function call. Finally, the result of the function call is assigned to the variable `sixteen`.

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
```

No return

```
...     >>> does_not_return_square(4)
```

None value is not displayed

```
>>> sixteen = does_not_return_square(4)
```

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

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
```

No return

```
... 
```

None value is not displayed

```
>>> does_not_return_square(4)
```

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

```
>>> 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'
```

Pure Functions & Non-Pure Functions

Pure Functions

just return values

Non-Pure Functions

have side effects

Pure Functions & Non-Pure Functions

Pure Functions
just return values

```
abs
```



Non-Pure Functions
have side effects

Pure Functions & Non-Pure Functions

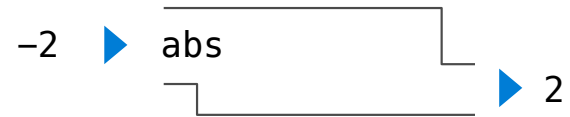
Pure Functions
just return values



Non-Pure Functions
have side effects

Pure Functions & Non-Pure Functions

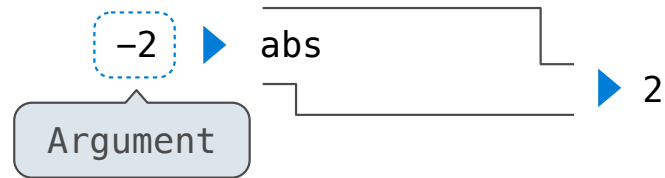
Pure Functions
just return values



Non-Pure Functions
have side effects

Pure Functions & Non-Pure Functions

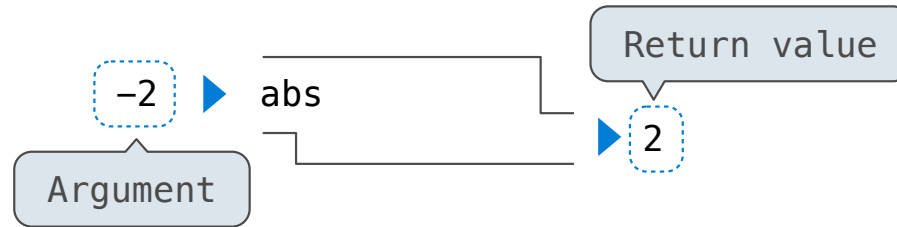
Pure Functions
just return values



Non-Pure Functions
have side effects

Pure Functions & Non-Pure Functions

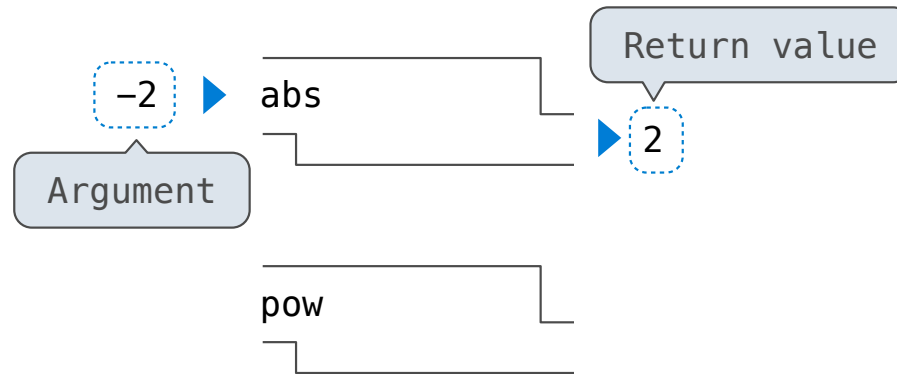
Pure Functions
just return values



Non-Pure Functions
have side effects

Pure Functions & Non-Pure Functions

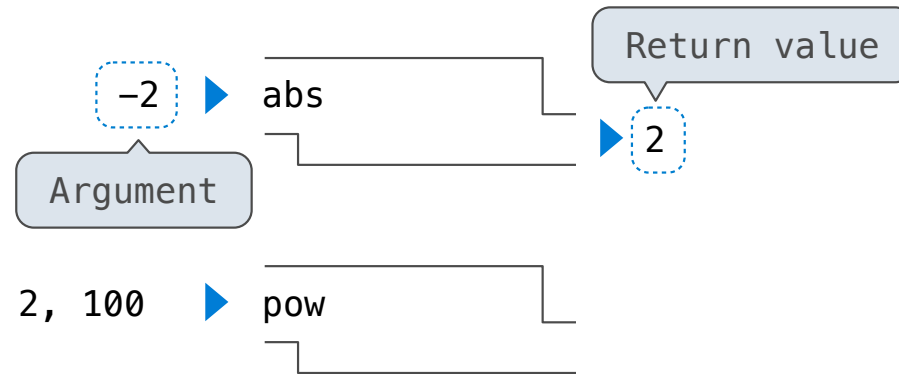
Pure Functions
just return values



Non-Pure Functions
have side effects

Pure Functions & Non-Pure Functions

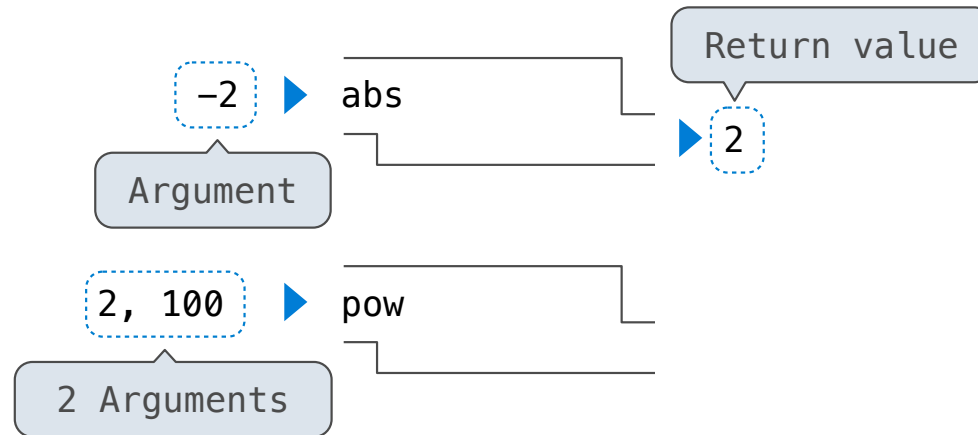
Pure Functions
just return values



Non-Pure Functions
have side effects

Pure Functions & Non-Pure Functions

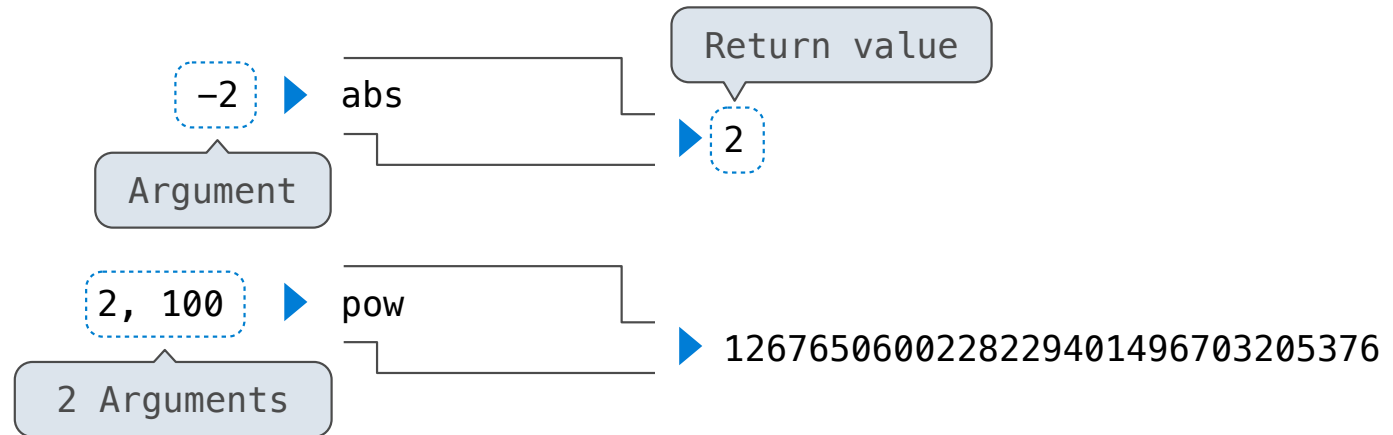
Pure Functions
just return values



Non-Pure Functions
have side effects

Pure Functions & Non-Pure Functions

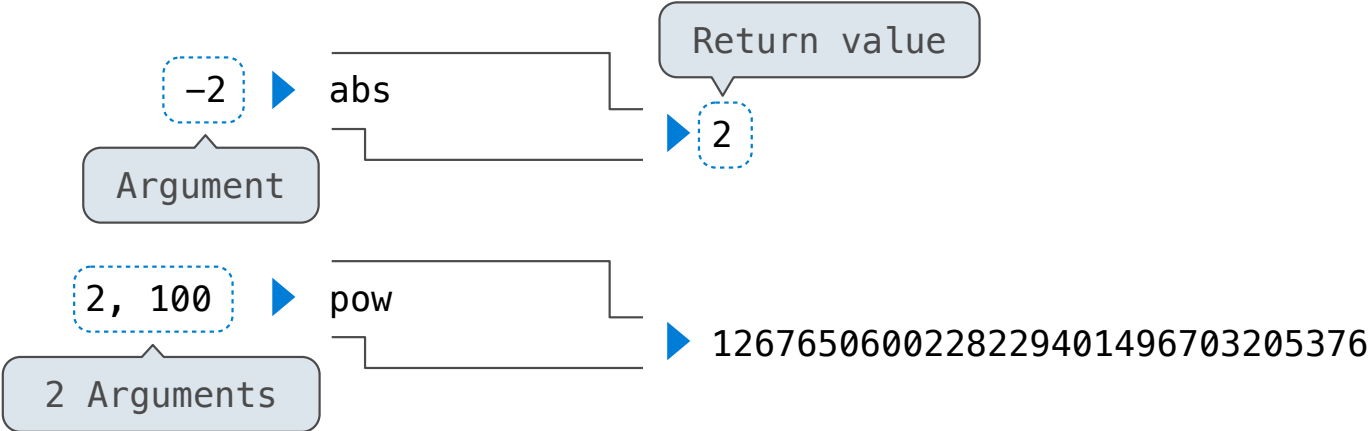
Pure Functions
just return values



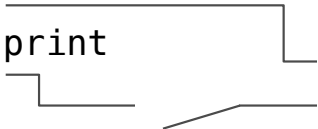
Non-Pure Functions
have side effects

Pure Functions & Non-Pure Functions

Pure Functions
just return values

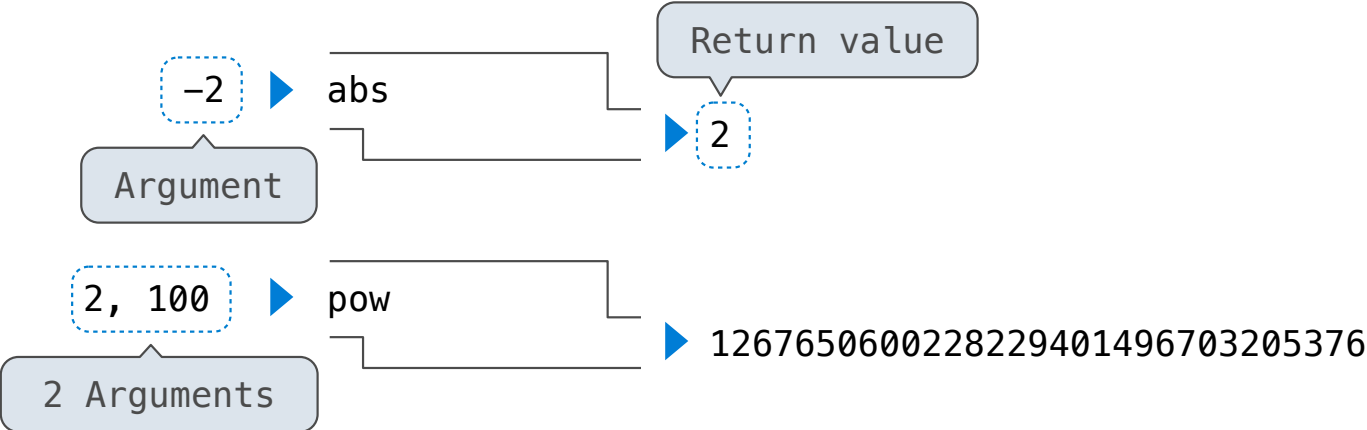


Non-Pure Functions
have side effects



Pure Functions & Non-Pure Functions

Pure Functions
just return values

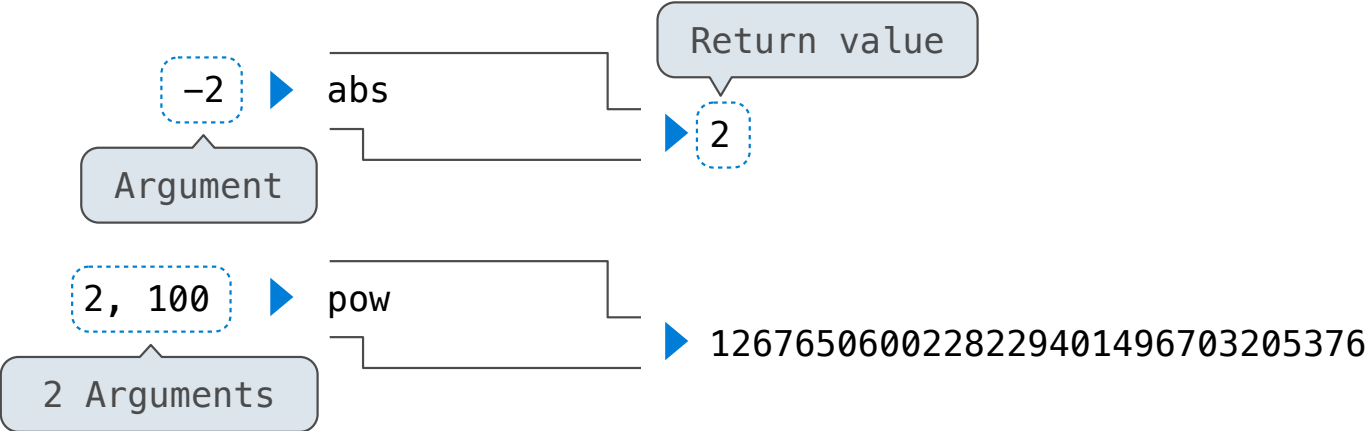


Non-Pure Functions
have side effects

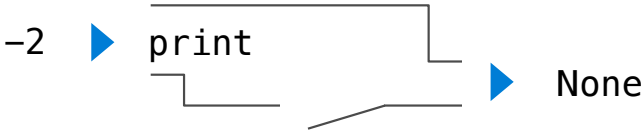


Pure Functions & Non-Pure Functions

Pure Functions
just return values

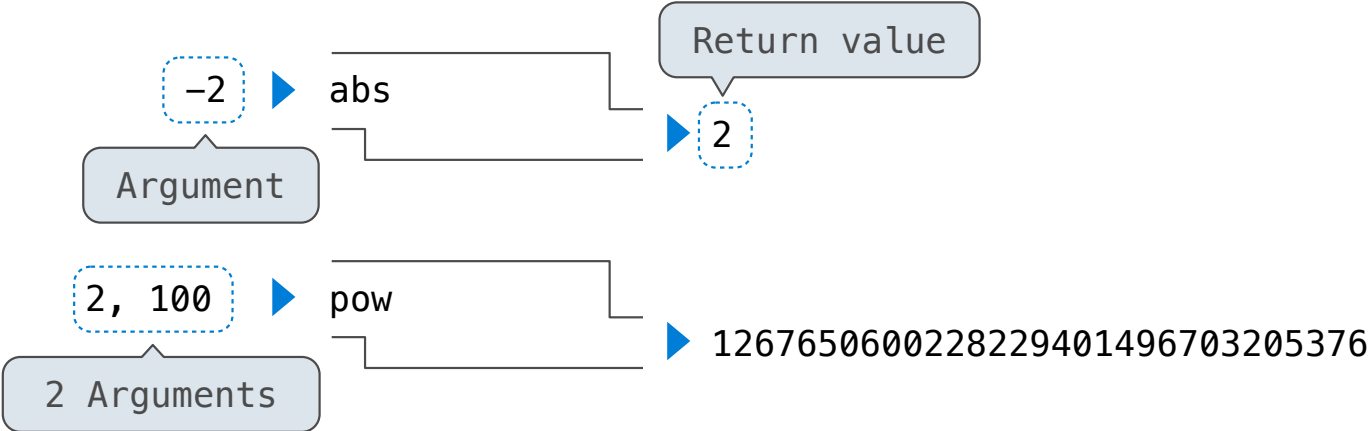


Non-Pure Functions
have side effects

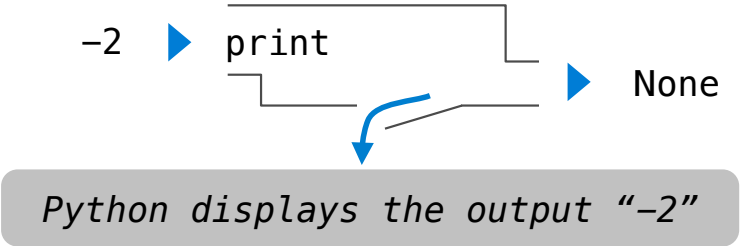


Pure Functions & Non-Pure Functions

Pure Functions
just return values

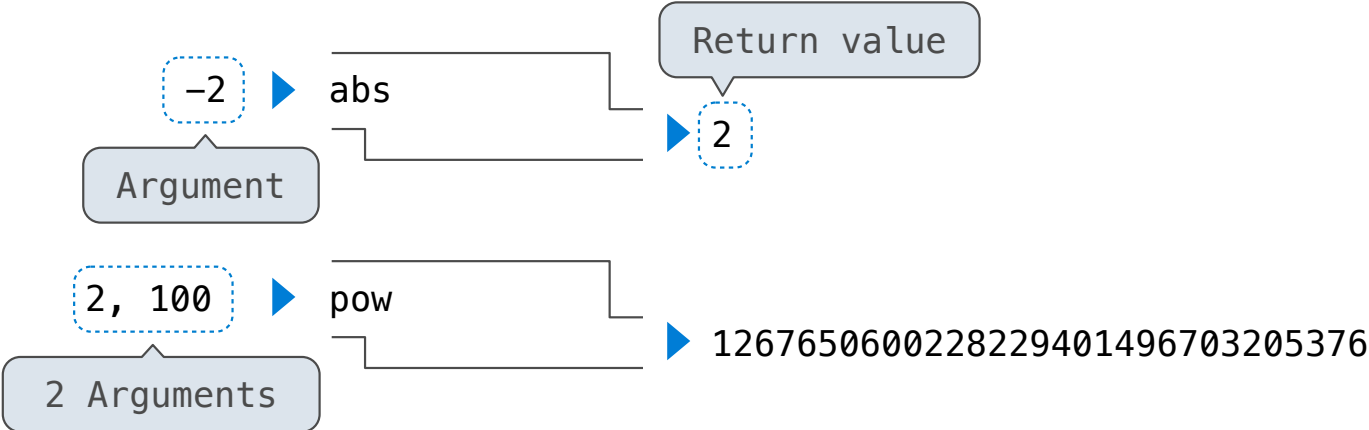


Non-Pure Functions
have side effects

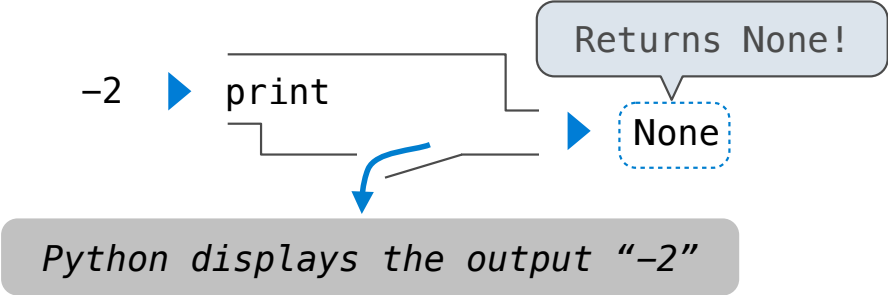


Pure Functions & Non-Pure Functions

Pure Functions
just return values

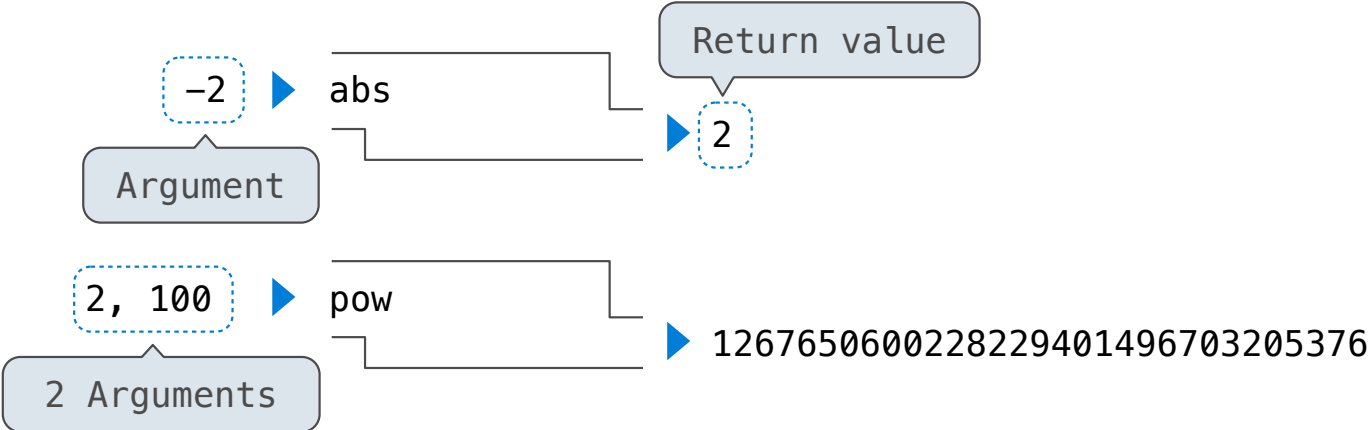


Non-Pure Functions
have side effects

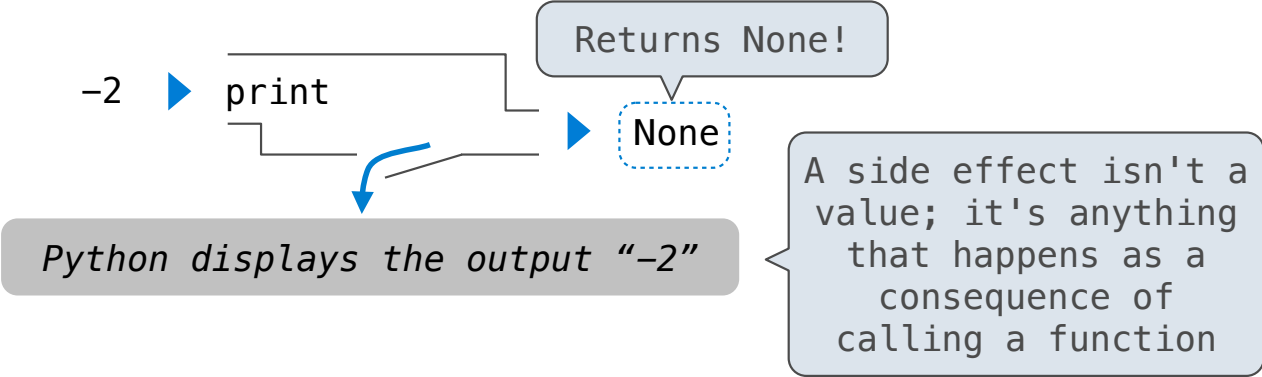


Pure Functions & Non-Pure Functions

Pure Functions
just return values

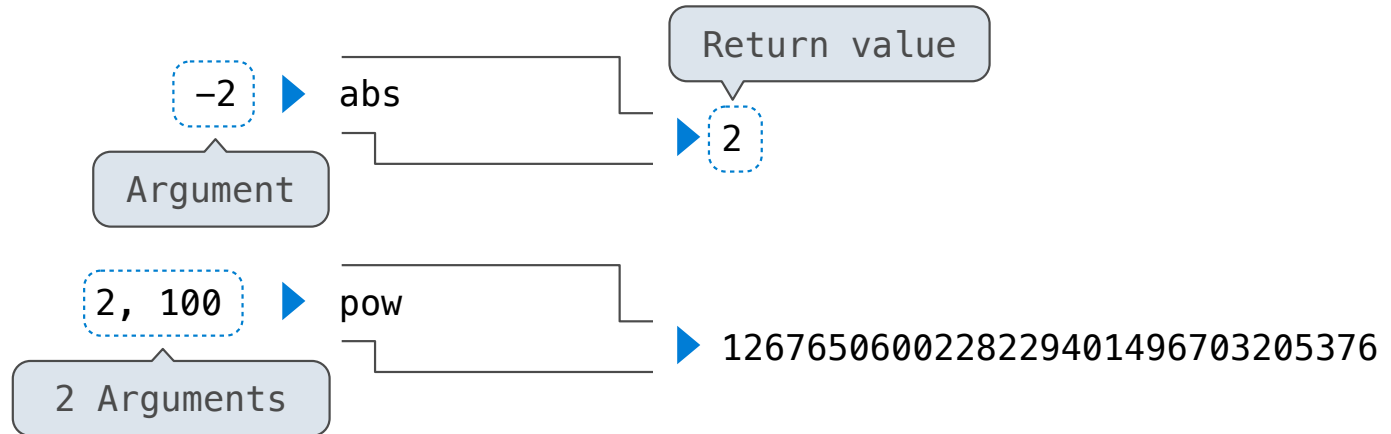


Non-Pure Functions
have side effects

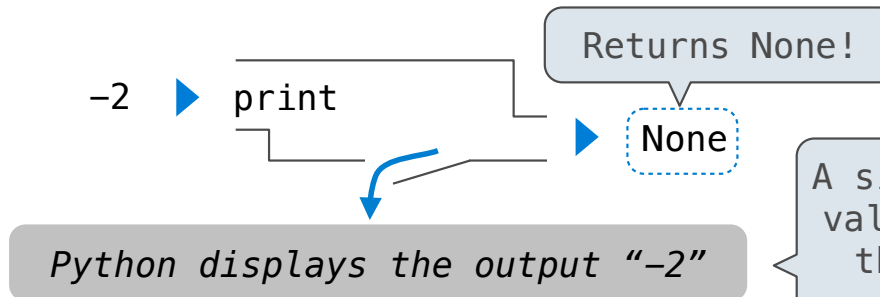


Pure Functions & Non-Pure Functions

Pure Functions
just return values



Non-Pure Functions
have side effects



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

A non-pure function doesn't have to return `None` (but `print` always does).

Nested Expressions with Print

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

Nested Expressions with Print

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

```
print(print(1), print(2))
```

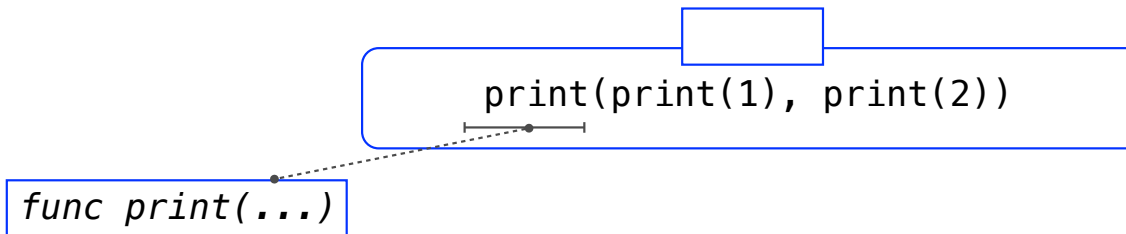
Nested Expressions with Print

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

```
print(print(1), print(2))
```

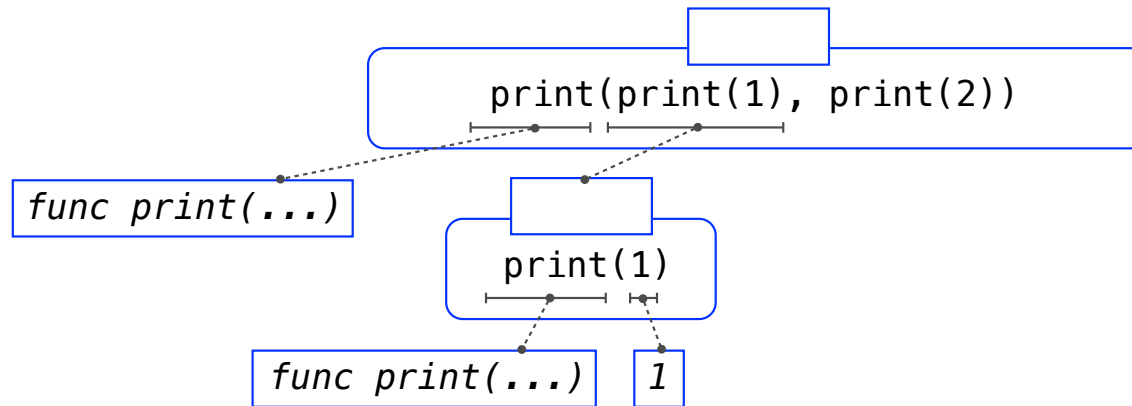
Nested Expressions with Print

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



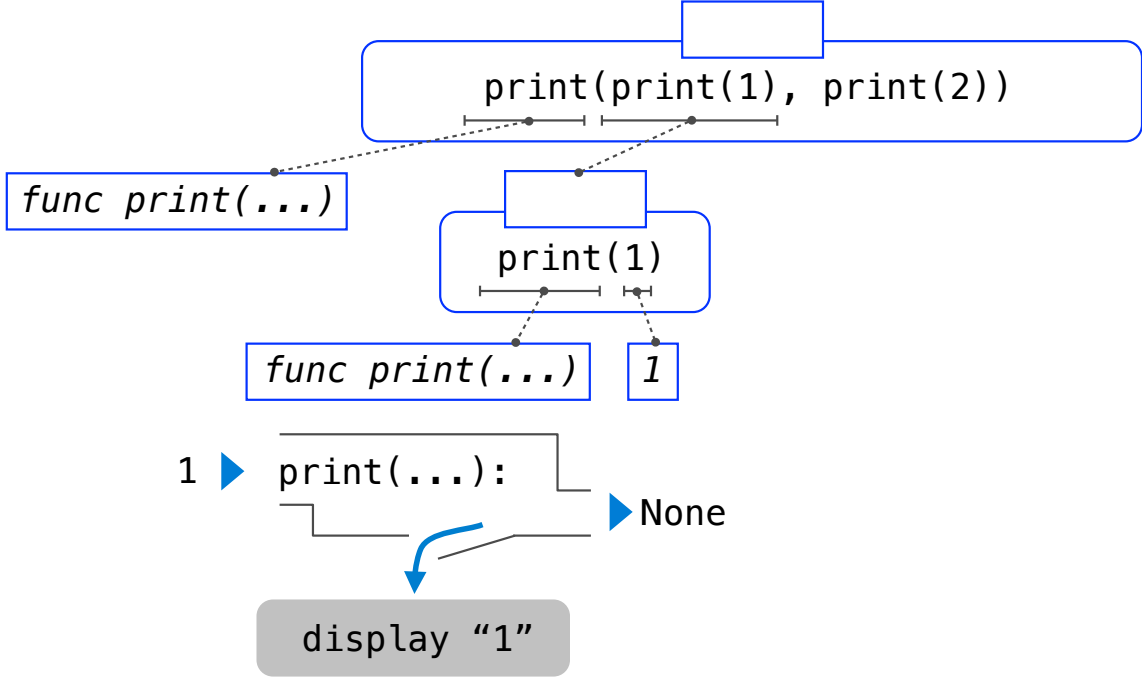
Nested Expressions with Print

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



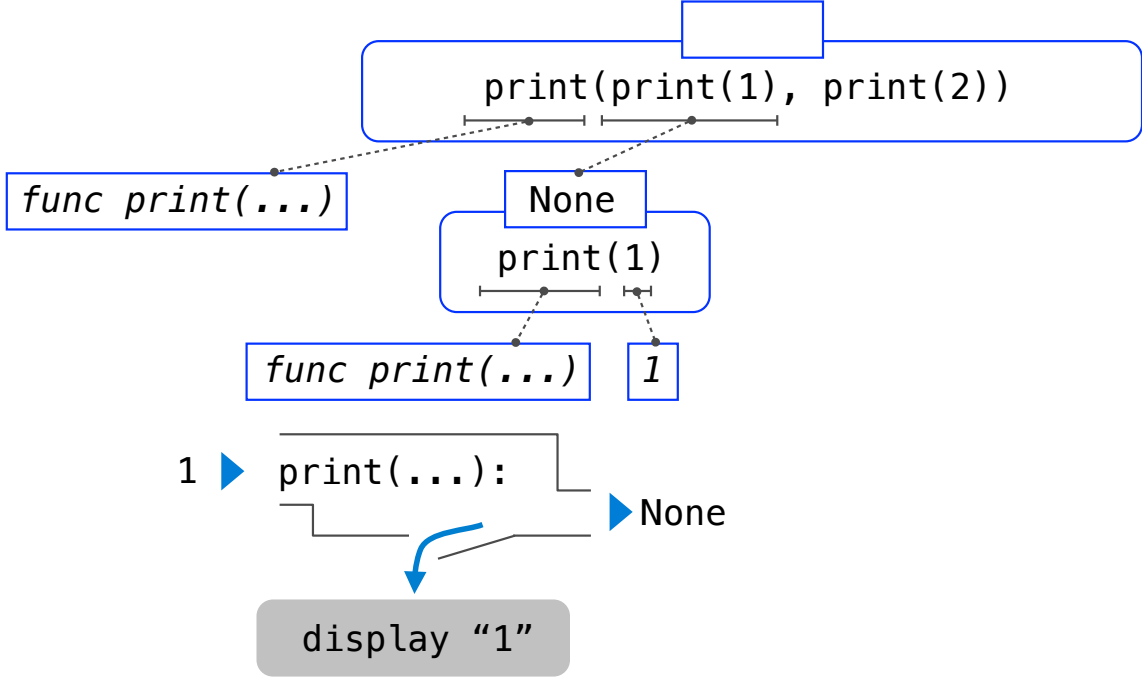
Nested Expressions with Print

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



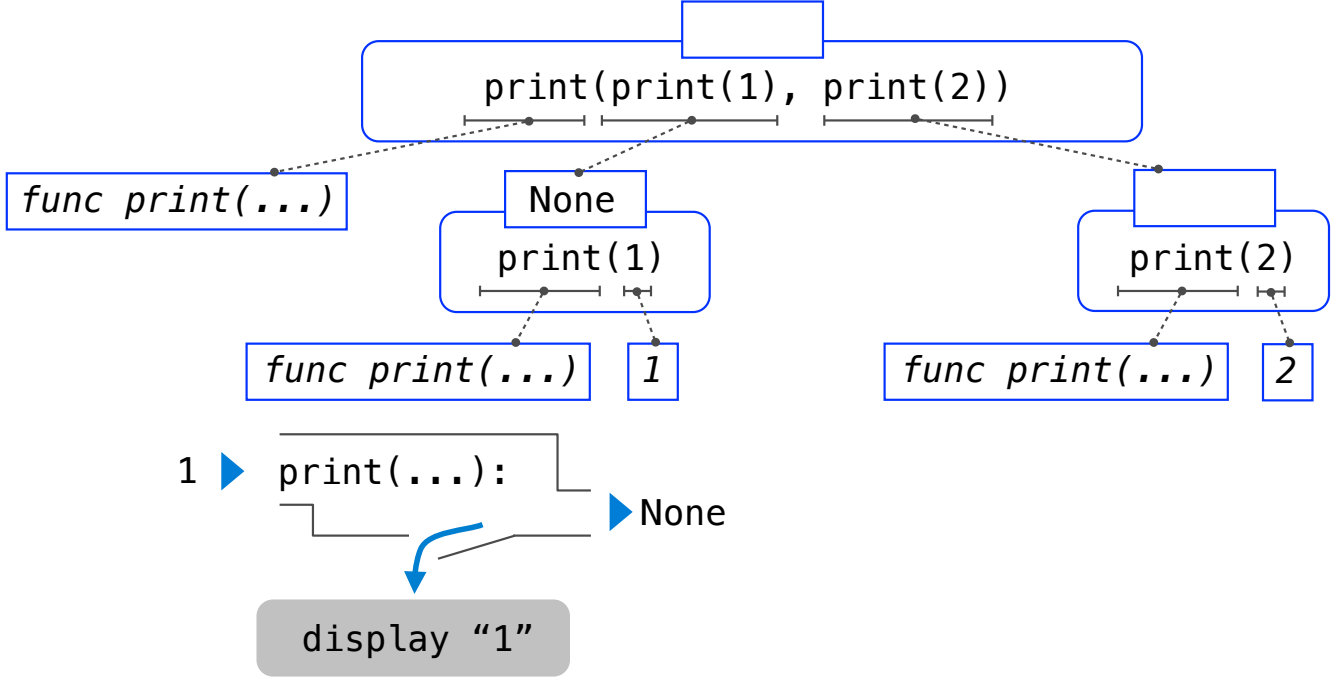
Nested Expressions with Print

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



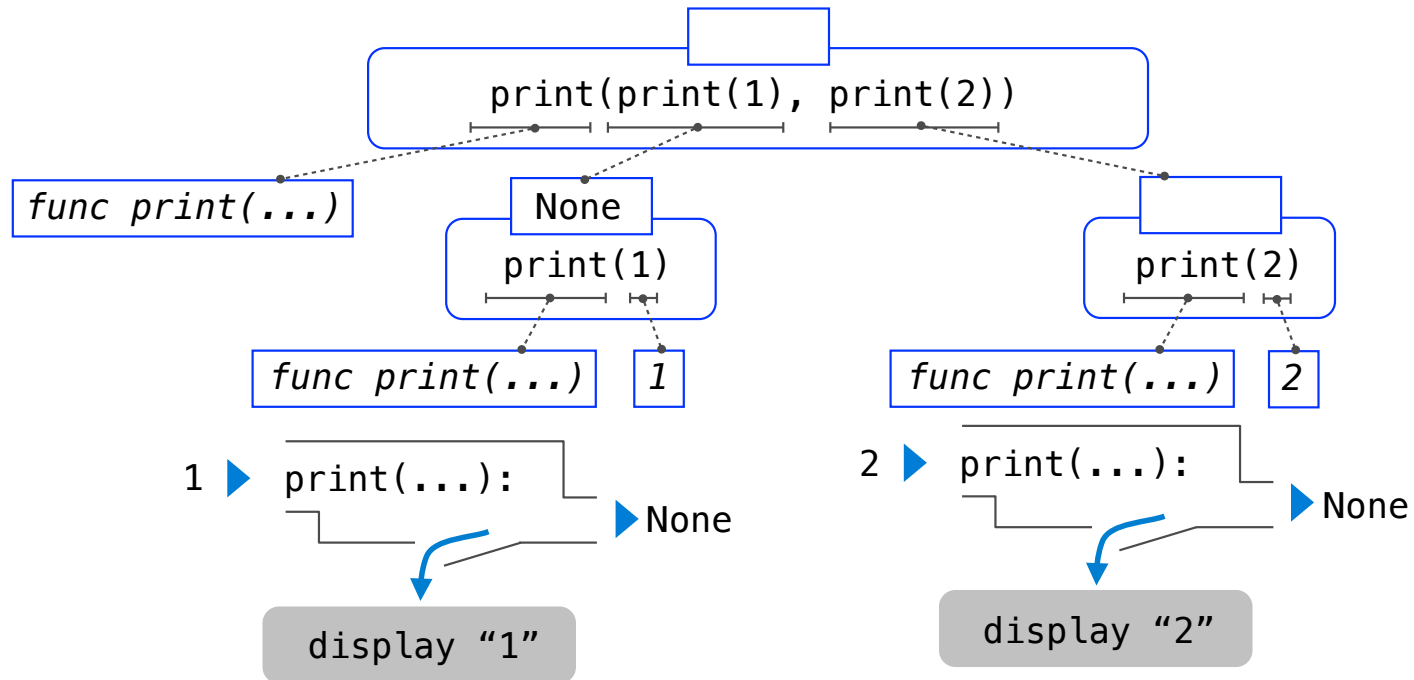
Nested Expressions with Print

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



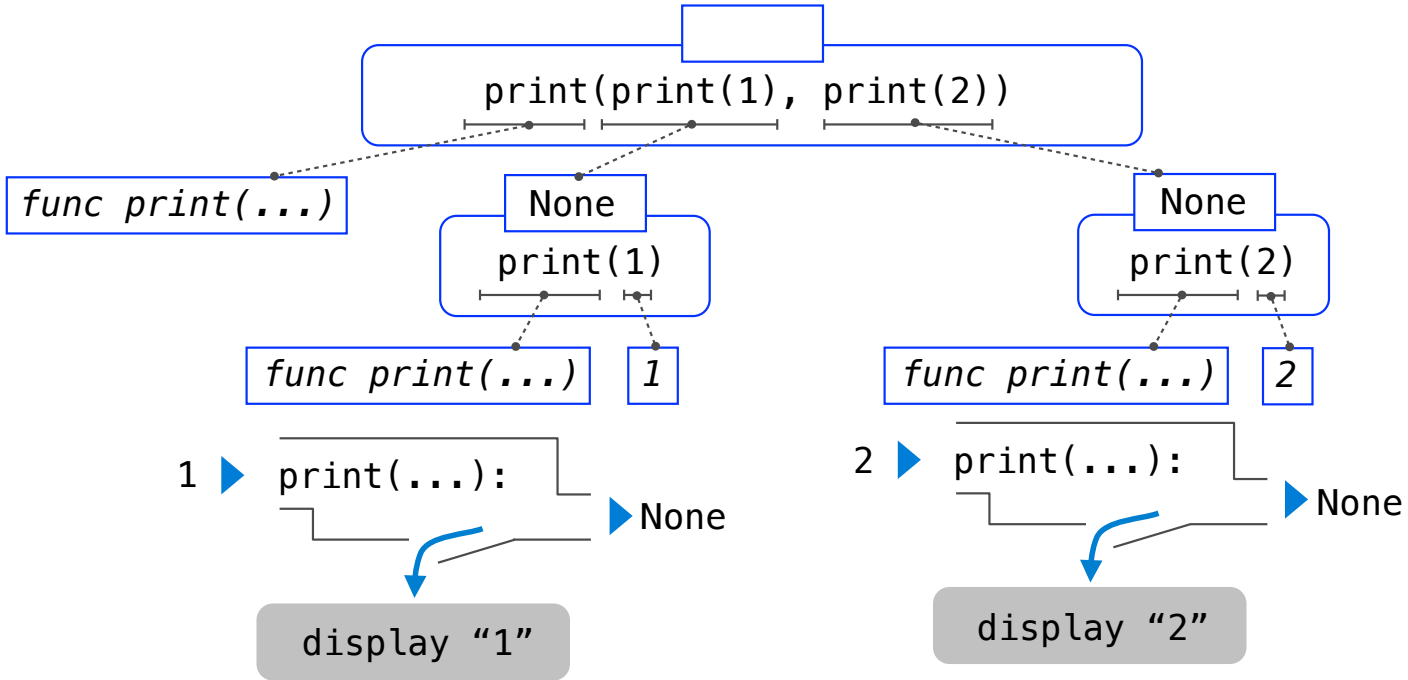
Nested Expressions with Print

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



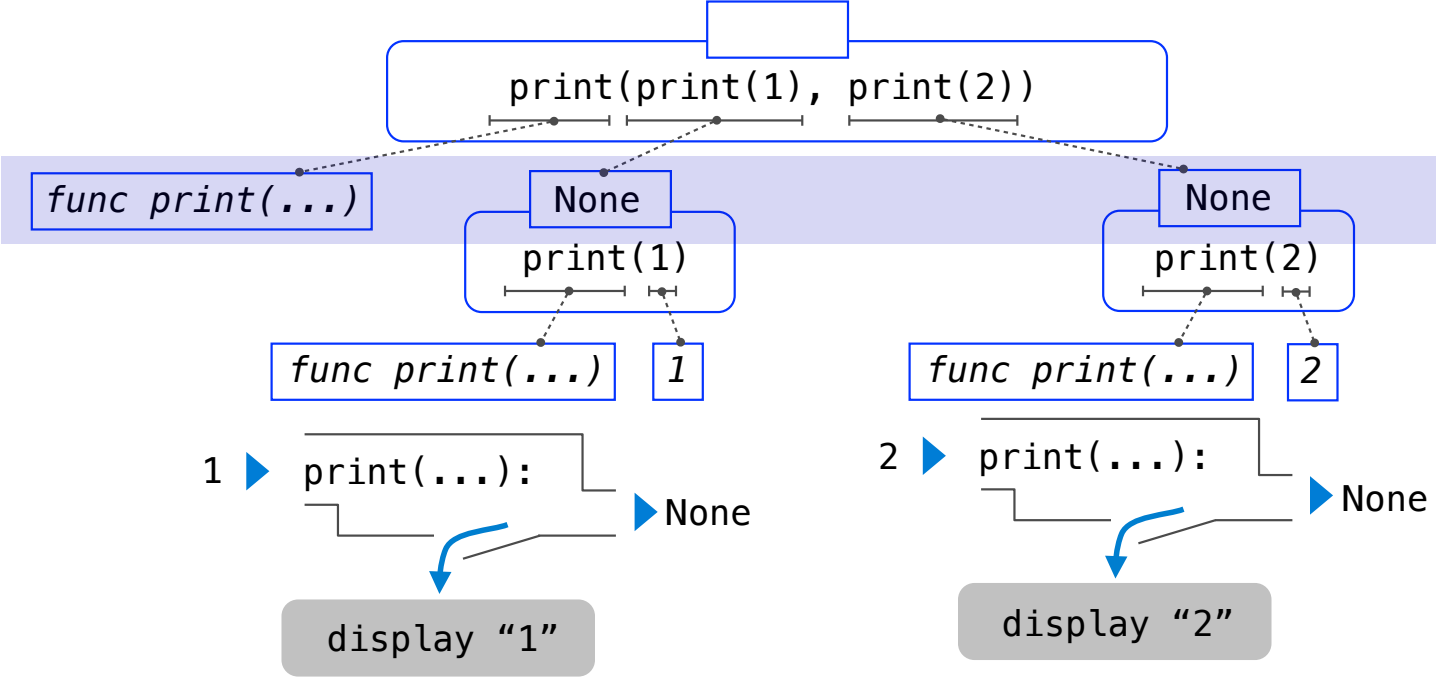
Nested Expressions with Print

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

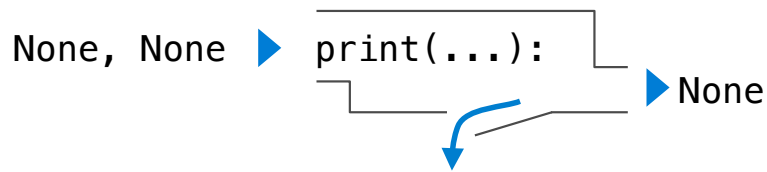


Nested Expressions with Print

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

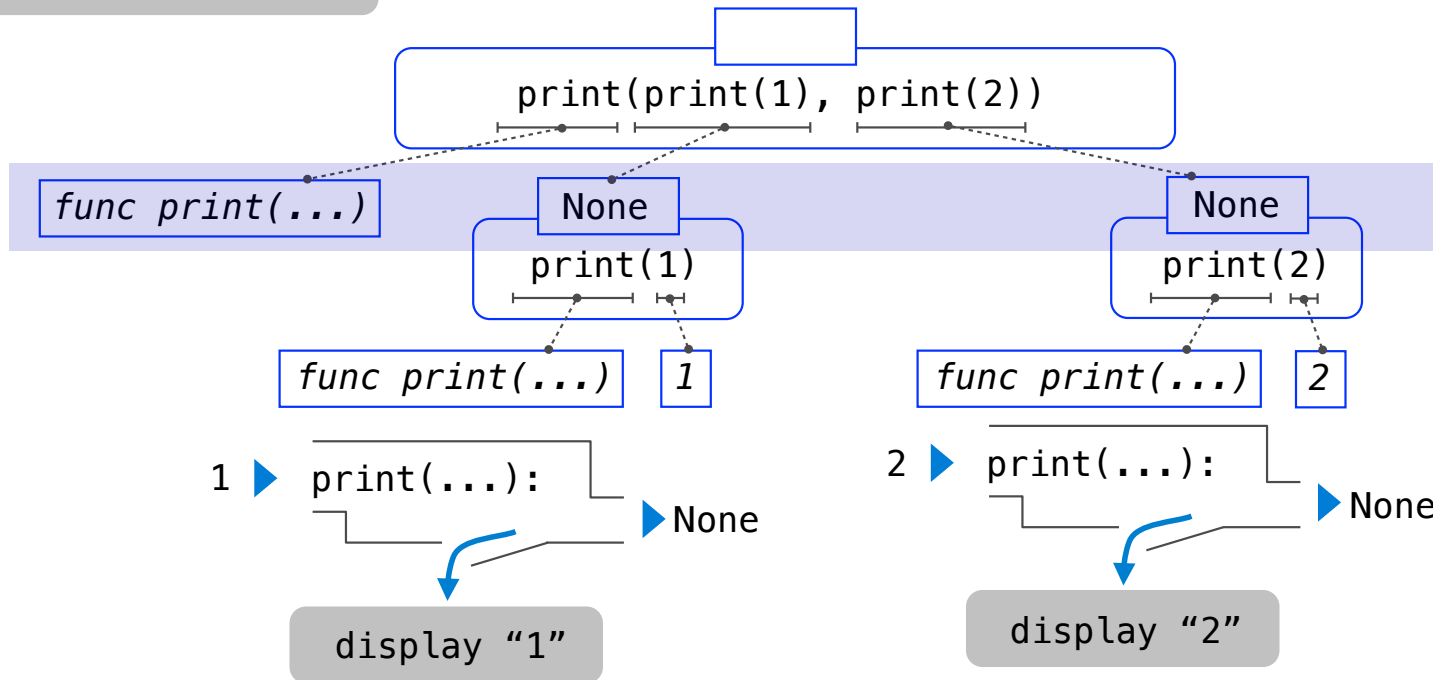


Nested Expressions with Print

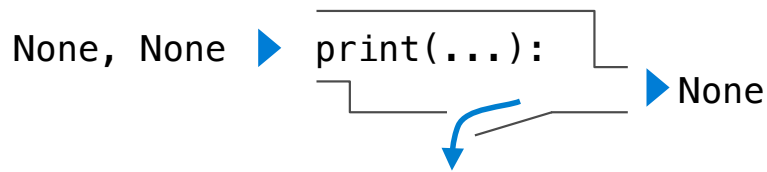


display "None None"

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

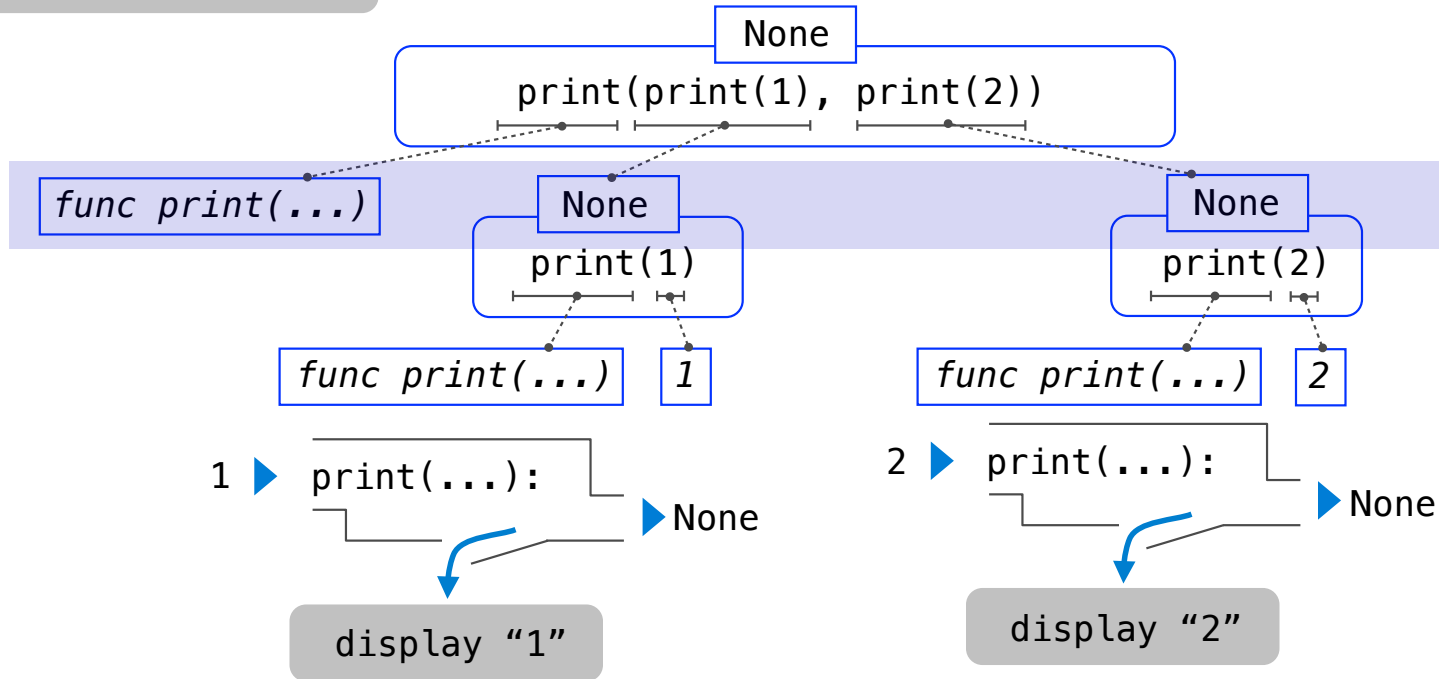


Nested Expressions with Print

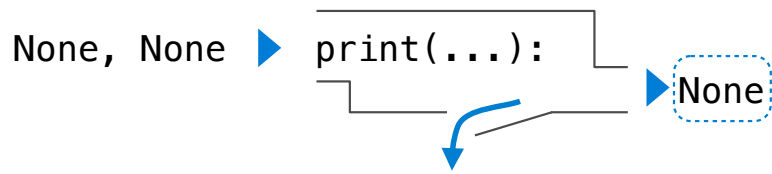


display "None None"

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

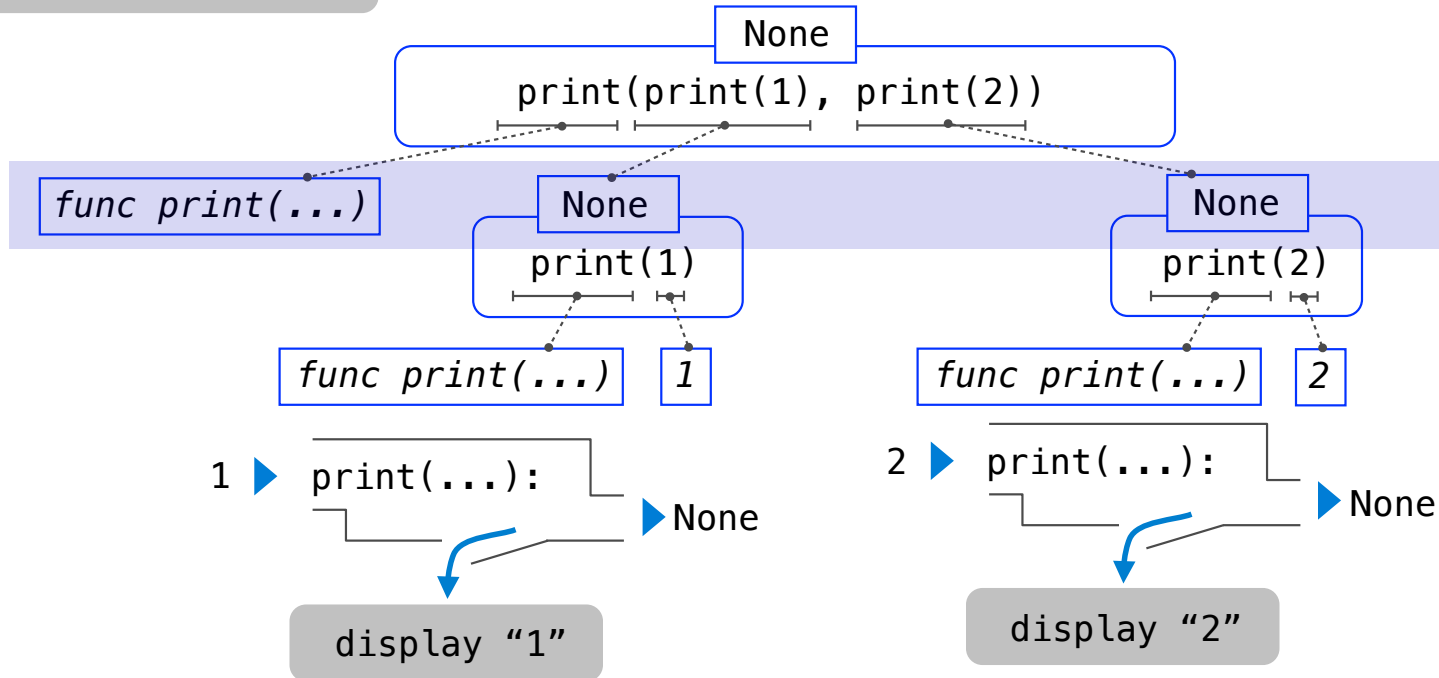


Nested Expressions with Print

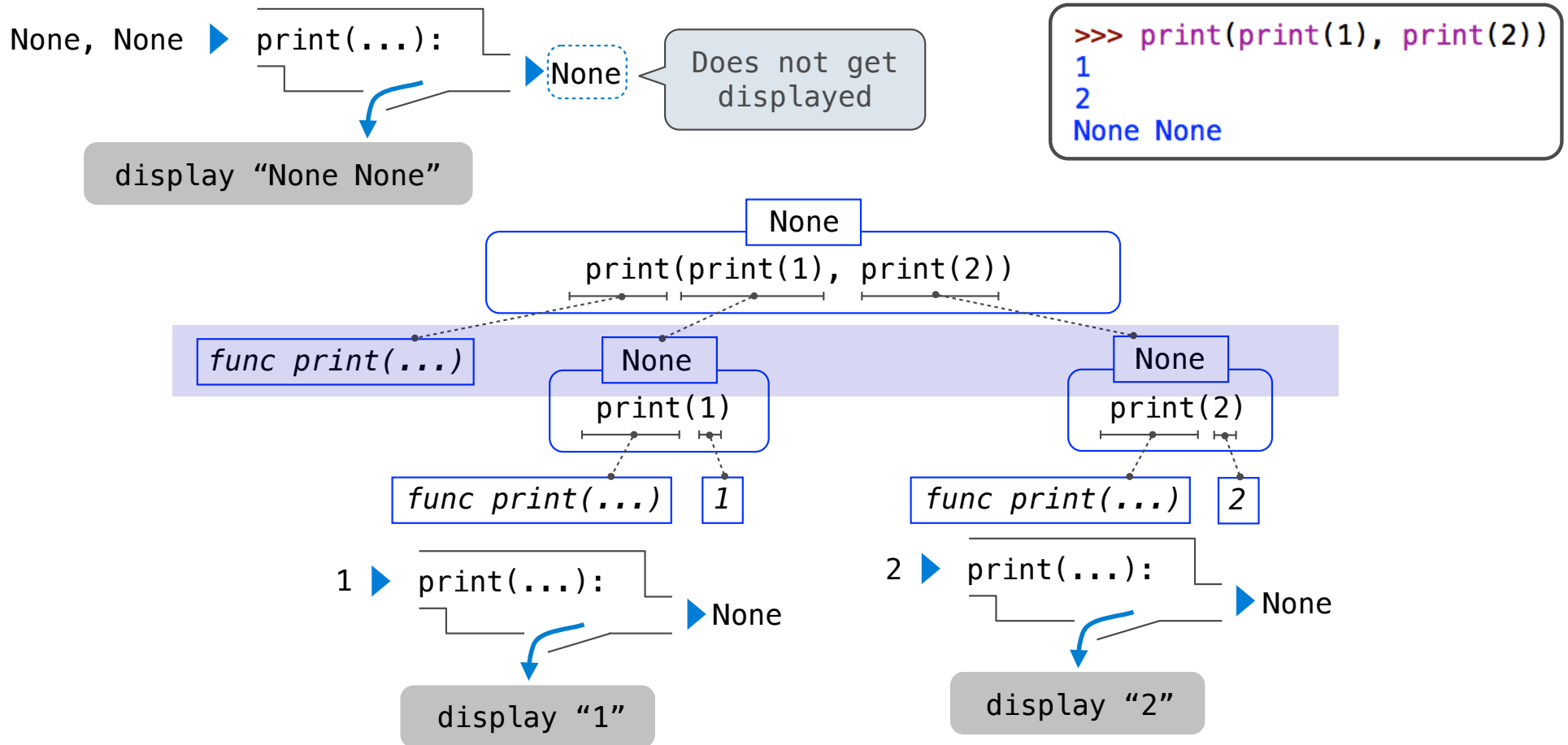


display "None None"

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



Nested Expressions with Print



Names, Assignment, and User-Defined Functions

(Demo)

Environment Diagrams

Environment Diagrams

Environment diagrams visualize the interpreter's process.

```
→ 1 from math import pi  
→ 2 tau = 2 * pi
```

Environment Diagrams

Environment diagrams visualize the interpreter's process.

```
→ 1 from math import pi  
→ 2 tau = 2 * pi
```

```
Global frame  
pi | 3.1416
```

Environment Diagrams

Environment diagrams visualize the interpreter's process.

```
→ 1 from math import pi  
→ 2 tau = 2 * pi
```

```
Global frame  
pi 3.1416
```

Code (left):

Frames (right):

Environment Diagrams

Environment diagrams visualize the interpreter's process.

```
→ 1 from math import pi  
→ 2 tau = 2 * pi
```

```
Global frame  
pi 3.1416
```

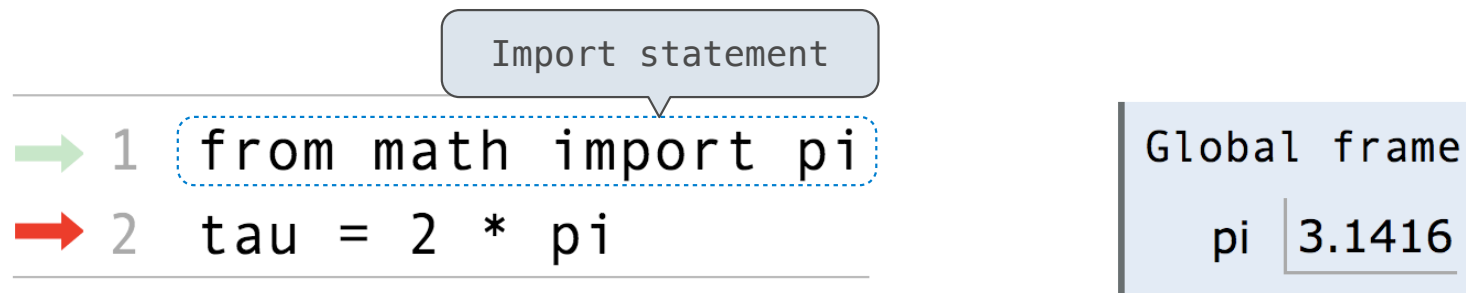
Code (left):

Statements and expressions

Frames (right):

Environment Diagrams

Environment diagrams visualize the interpreter's process.



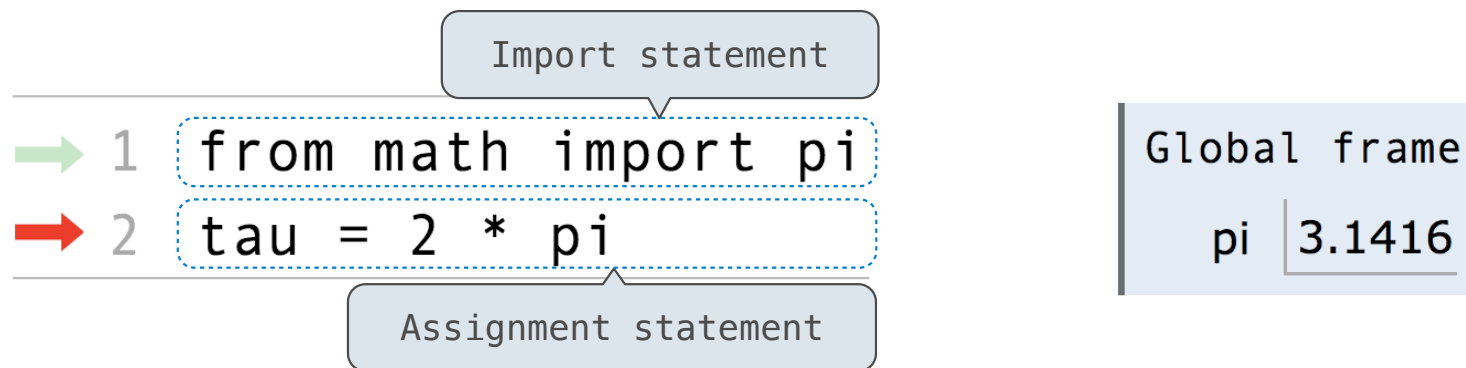
Code (left):

Statements and expressions

Frames (right):

Environment Diagrams

Environment diagrams visualize the interpreter's process.



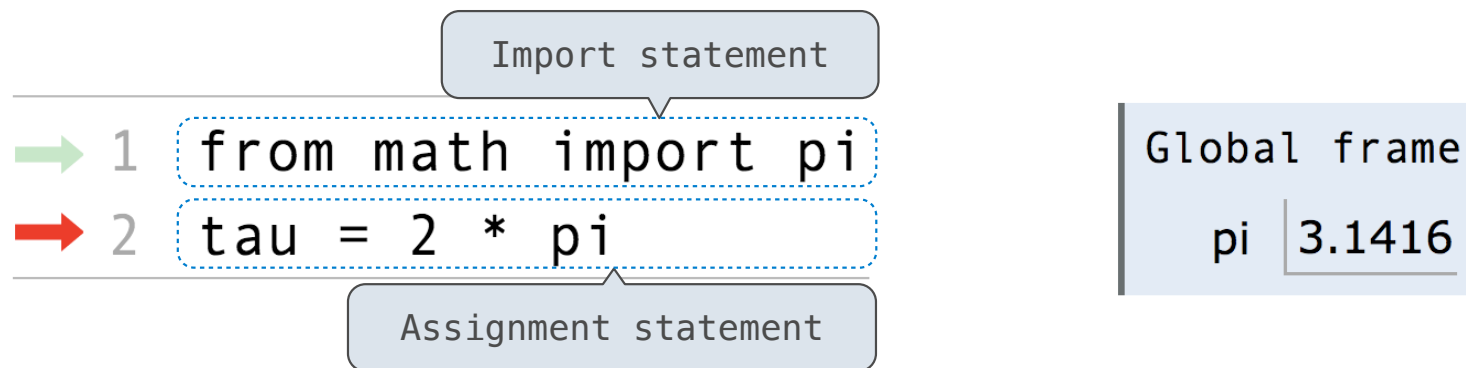
Code (left):

Statements and expressions

Frames (right):

Environment Diagrams

Environment diagrams visualize the interpreter's process.



Code (left):

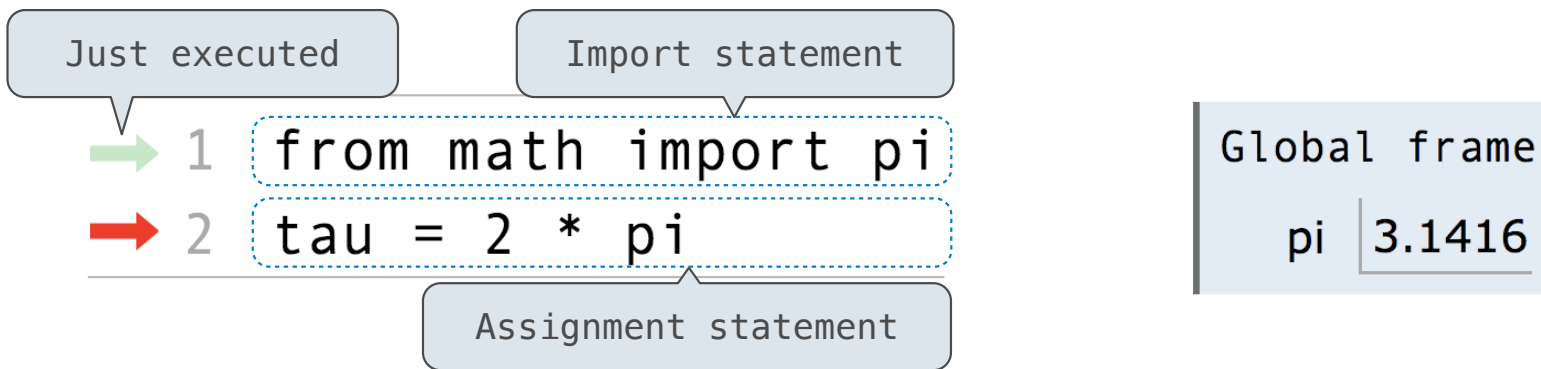
Statements and expressions

Arrows indicate evaluation order

Frames (right):

Environment Diagrams

Environment diagrams visualize the interpreter's process.



Code (left):

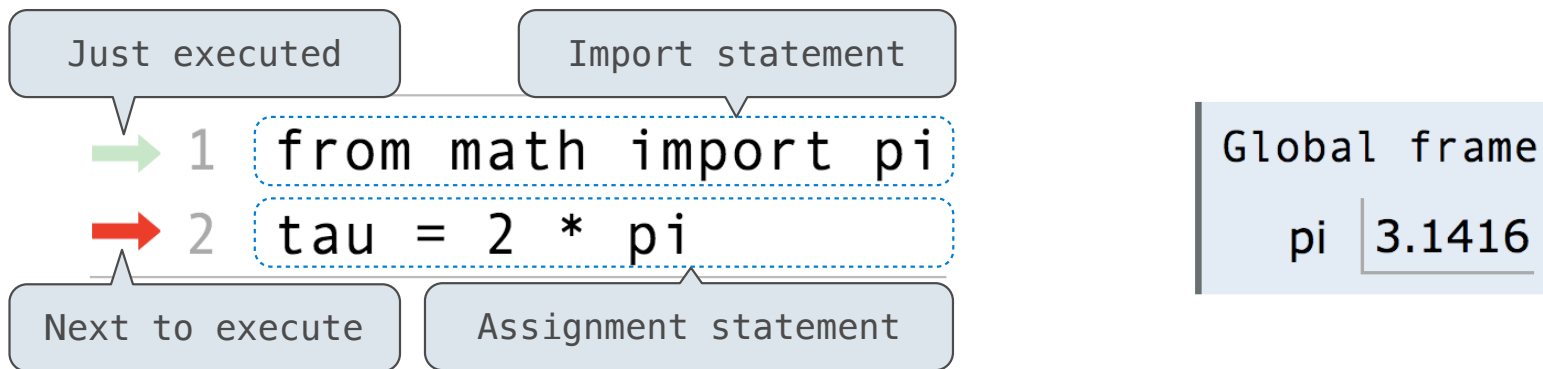
Statements and expressions

Arrows indicate evaluation order

Frames (right):

Environment Diagrams

Environment diagrams visualize the interpreter's process.



Code (left):

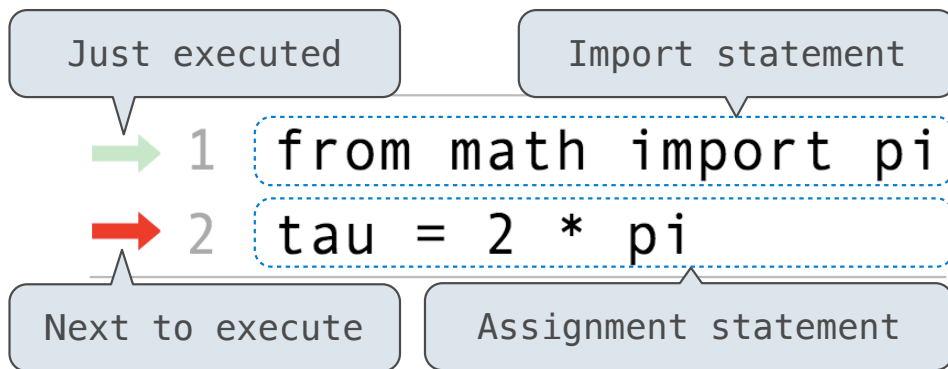
Statements and expressions

Arrows indicate evaluation order

Frames (right):

Environment Diagrams

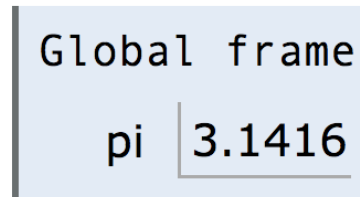
Environment diagrams visualize the interpreter's process.



Code (left):

Statements and expressions

Arrows indicate evaluation order

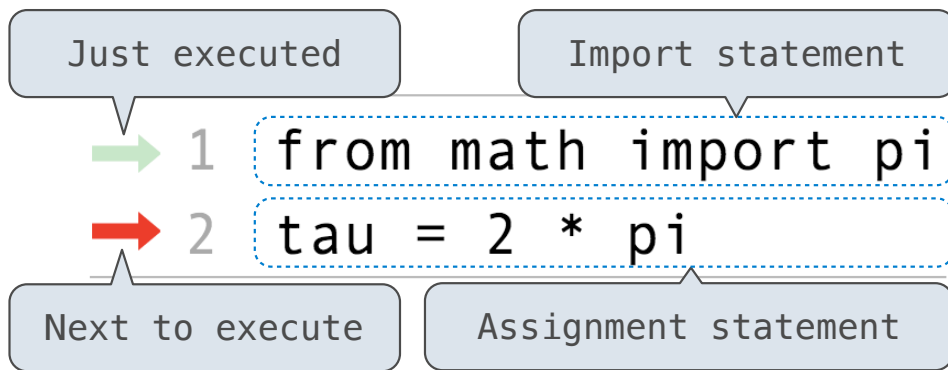


Frames (right):

Each name is bound to a value

Environment Diagrams

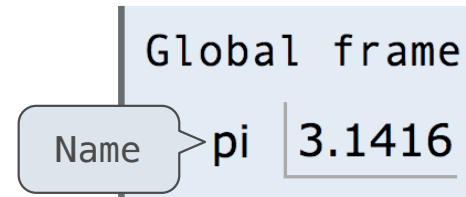
Environment diagrams visualize the interpreter's process.



Code (left):

Statements and expressions

Arrows indicate evaluation order

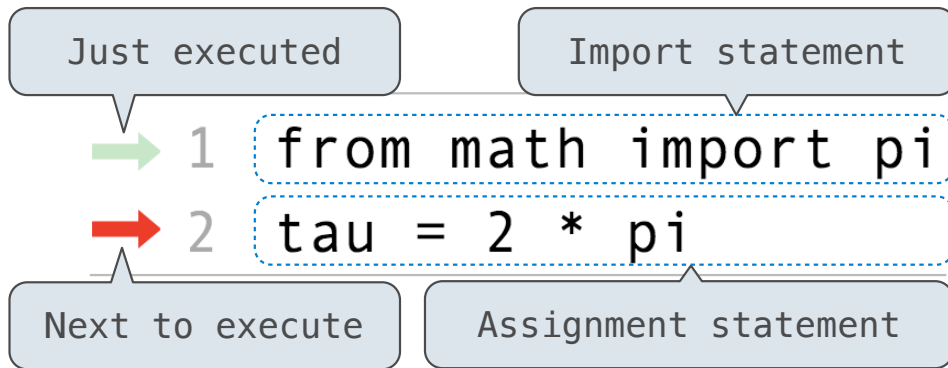


Frames (right):

Each name is bound to a value

Environment Diagrams

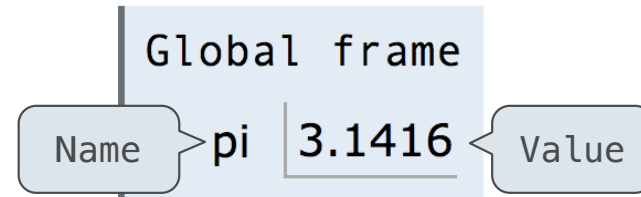
Environment diagrams visualize the interpreter's process.



Code (left):

Statements and expressions

Arrows indicate evaluation order

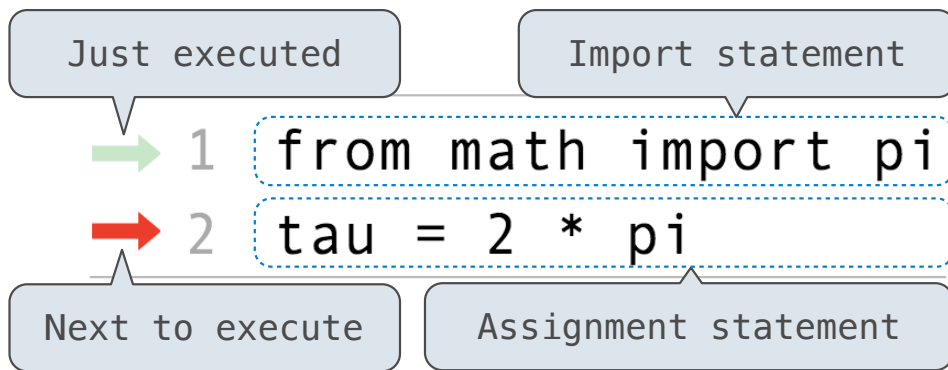


Frames (right):

Each name is bound to a value

Environment Diagrams

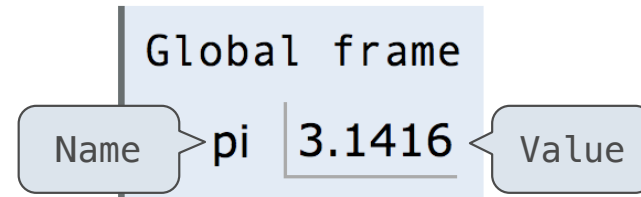
Environment diagrams visualize the interpreter's process.



Code (left):

Statements and expressions

Arrows indicate evaluation order



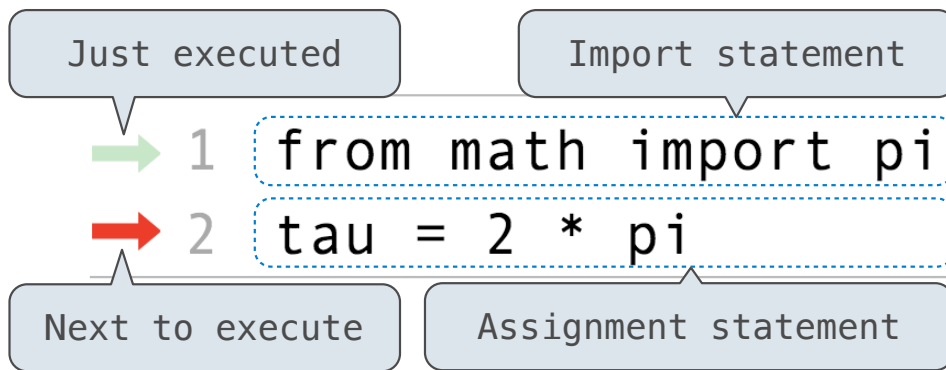
Frames (right):

Each name is bound to a value

Within a frame, a name cannot be repeated

Environment Diagrams

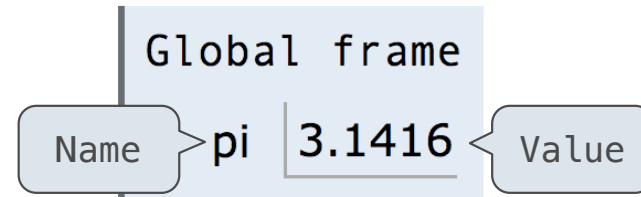
Environment diagrams visualize the interpreter's process.



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: tutor.cs61a.org)

Assignment Statements

Calling Functions

Calling Functions

(Demo: tutor.cs61a.org)

Calling User-Defined Functions

Calling User-Defined Functions

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

Calling User-Defined Functions

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

1. Add a local frame

Calling User-Defined Functions

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

1. Add a local frame
2. Bind the function's formal parameters to its arguments in that frame

Calling User-Defined Functions

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

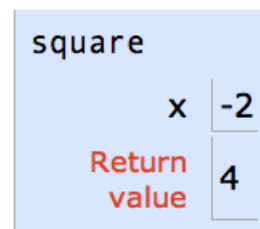
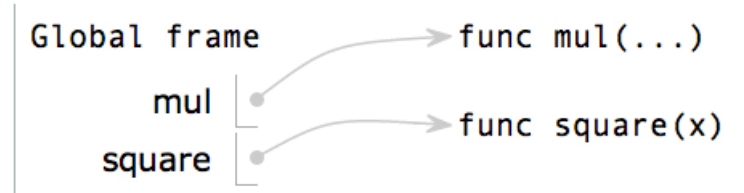
1. Add a local frame
2. Bind the function's formal parameters to its arguments in that frame
3. Execute the body of the function in that new environment

Calling User-Defined Functions

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

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

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

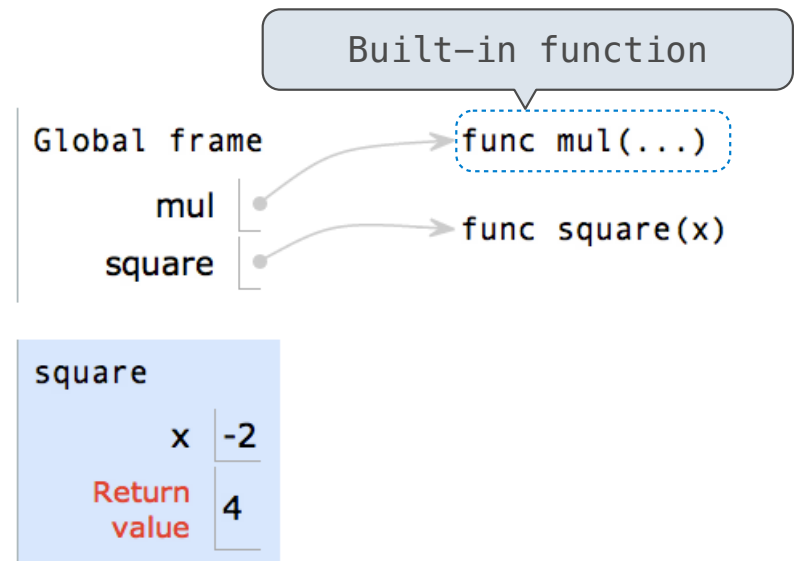


Calling User-Defined Functions

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

1. Add a local frame
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)
```

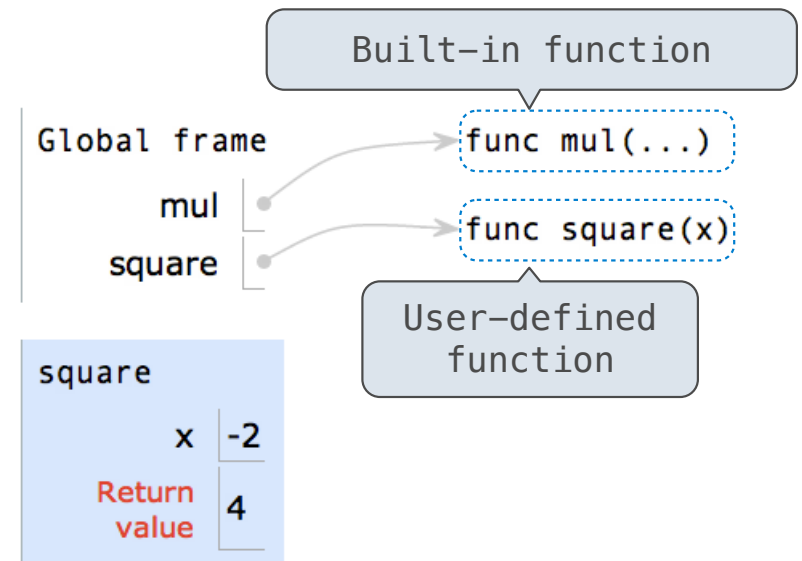


Calling User-Defined Functions

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

1. Add a local frame
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)
```

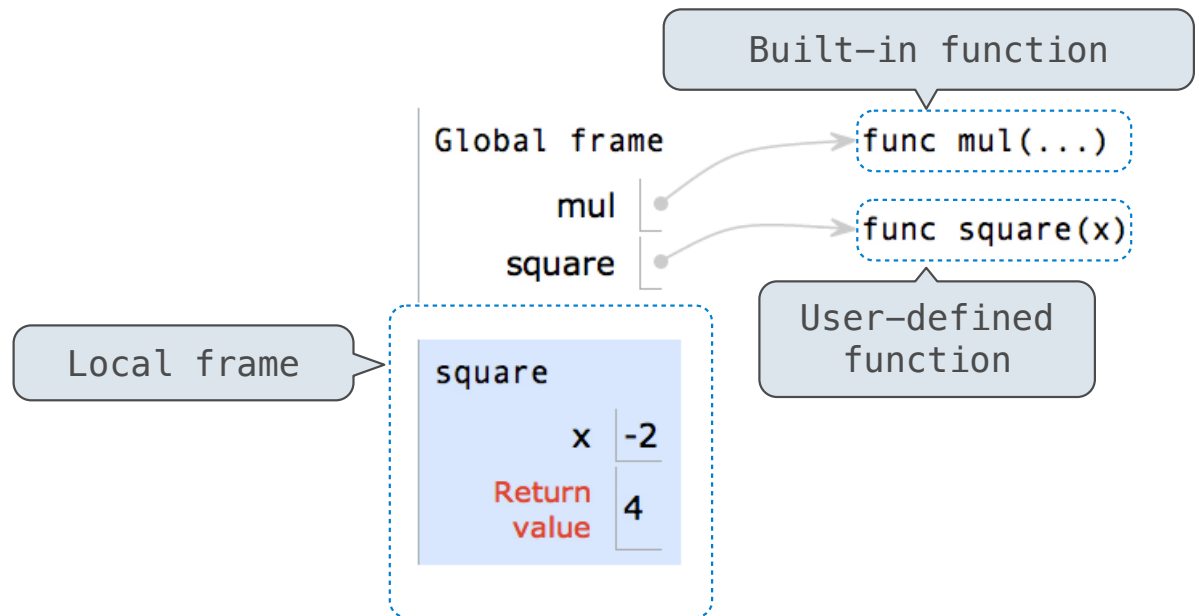


Calling User-Defined Functions

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

1. Add a local frame
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)
```

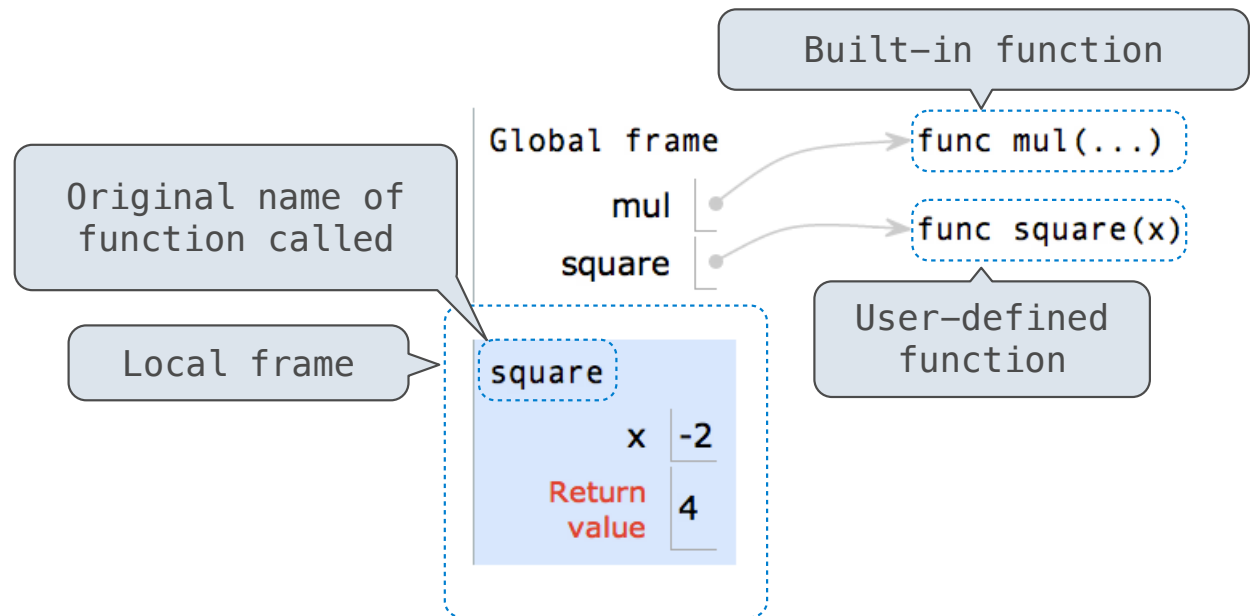


Calling User-Defined Functions

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

1. Add a local frame
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)
```

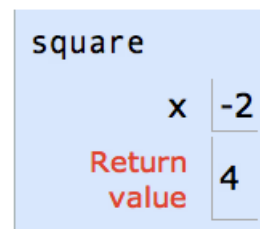
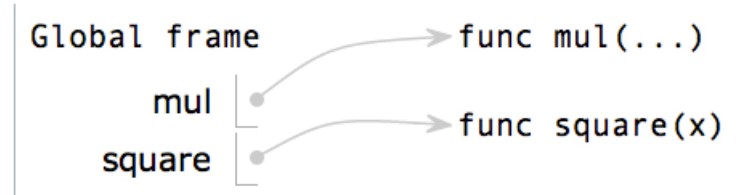


Calling User-Defined Functions

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

1. Add a local frame
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)
```



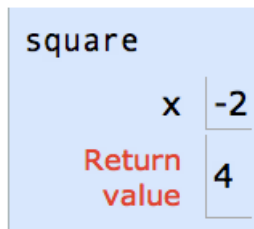
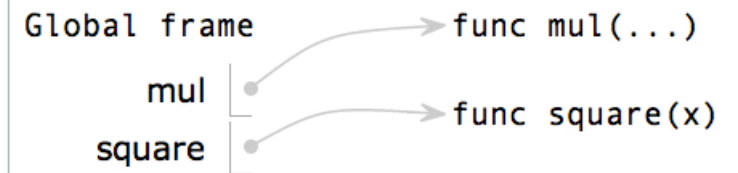
Calling User-Defined Functions

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

1. Add a local frame
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)
```

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



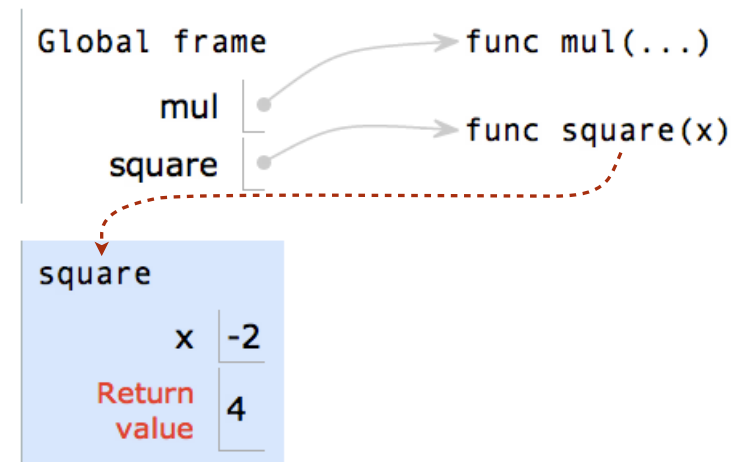
Calling User-Defined Functions

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

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

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

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



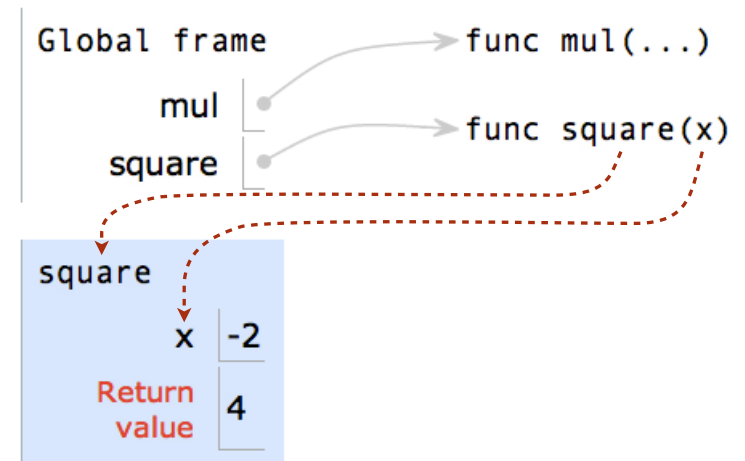
Calling User-Defined Functions

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

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

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

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



Life Cycle of a User-Defined Function

What happens?

Def statement:

Call expression:

Calling/Applying:

Life Cycle of a User-Defined Function

What happens?

Def statement: `>>> def square(x):`
 `return mul(x, x)`

Call expression:

Calling/Applying:


Life Cycle of a User-Defined Function

What happens?

Def statement:

>>>

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



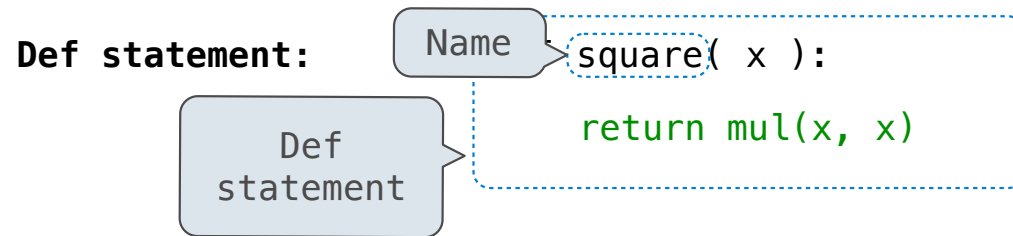
Def
statement

Call expression:

Calling/Applying:

Life Cycle of a User-Defined Function

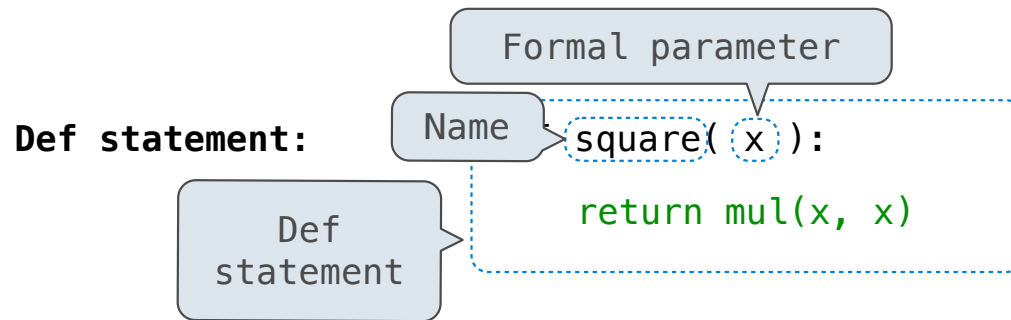
What happens?



Call expression:

Calling/Applying:

Life Cycle of a User-Defined Function

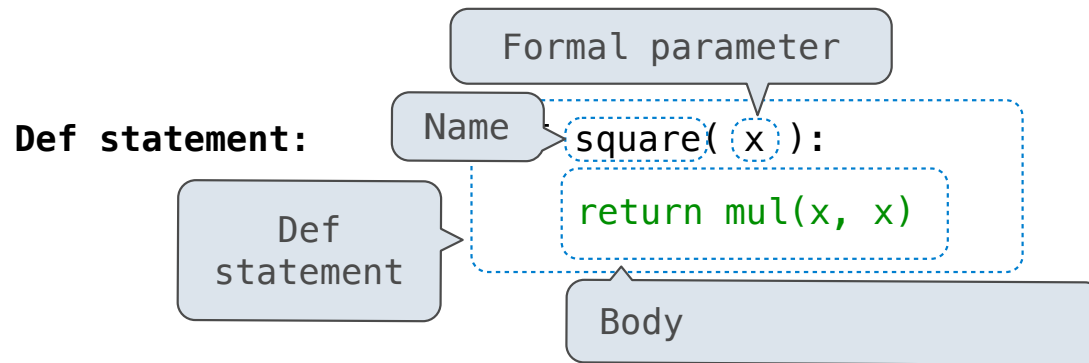


What happens?

Call expression:

Calling/Applying:

Life Cycle of a User-Defined Function

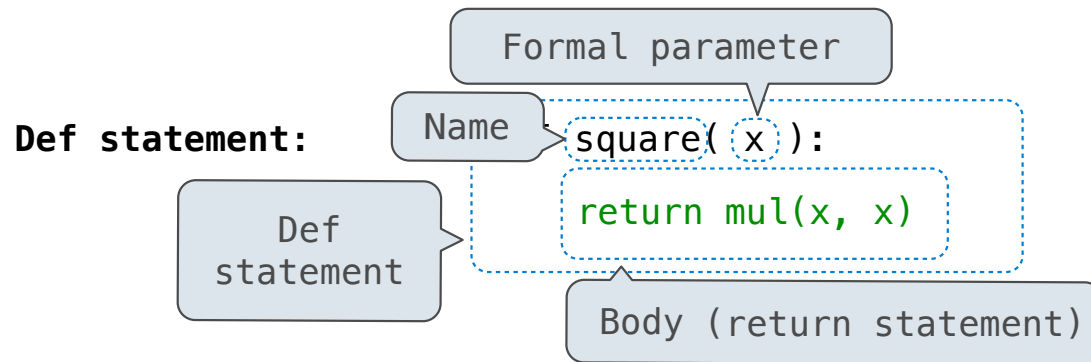


What happens?

Call expression:

Calling/Applying:

Life Cycle of a User-Defined Function

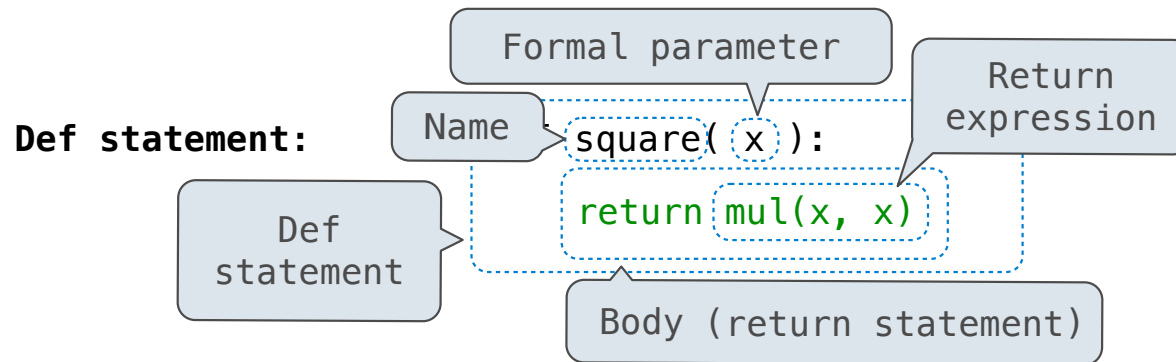


What happens?

Call expression:

Calling/Applying:

Life Cycle of a User-Defined Function

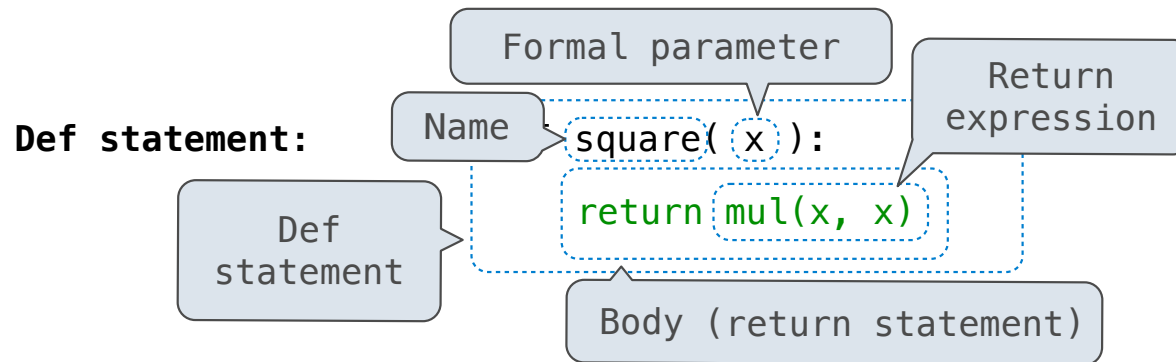


What happens?

Call expression:

Calling/Applying:

Life Cycle of a User-Defined Function



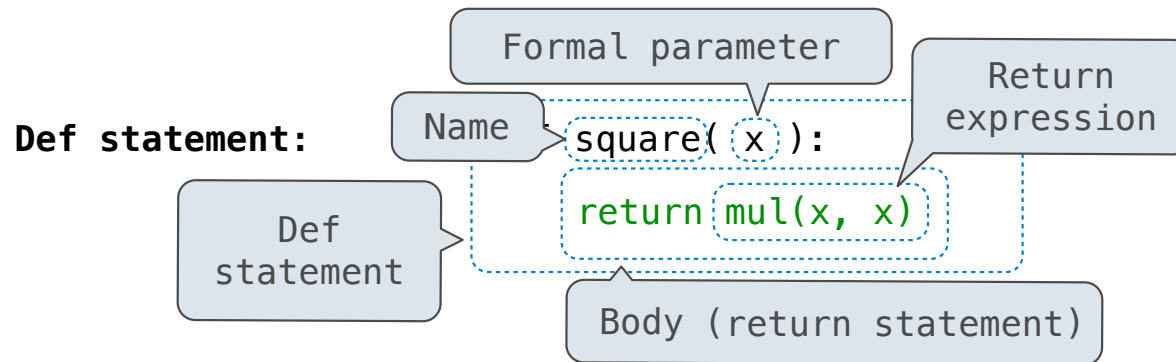
What happens?

A new function is created!

Call expression:

Calling/Applying:

Life Cycle of a User-Defined Function



What happens?

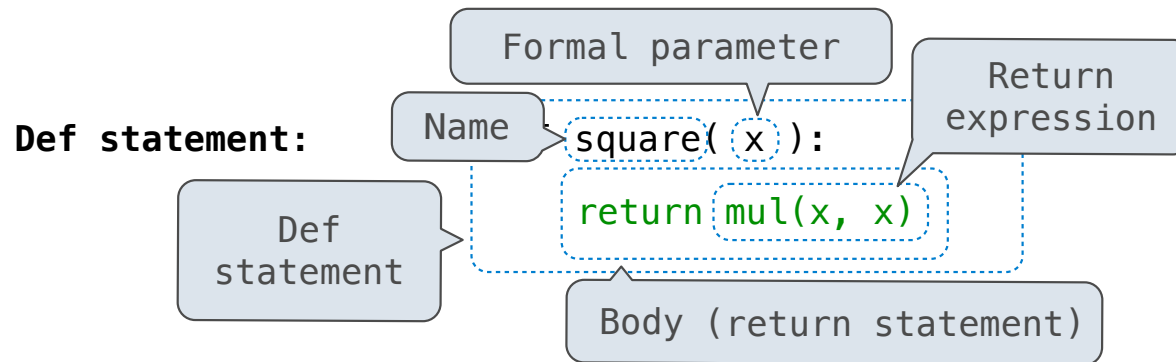
A new function is created!

Name bound to that function
in the current frame

Call expression:

Calling/Applying:

Life Cycle of a User-Defined Function



What happens?

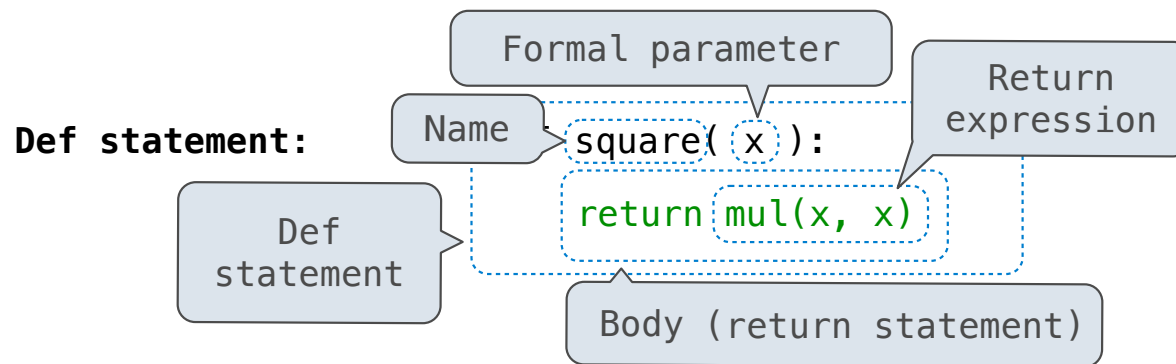
A new function is created!

Name bound to that function in the current frame

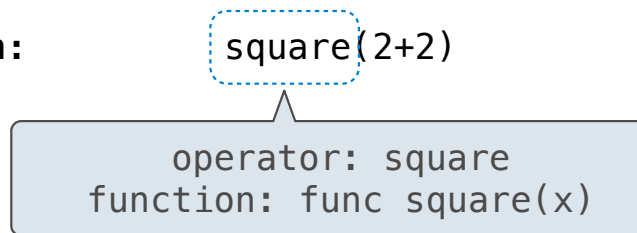
Call expression: `square(2+2)`

Calling/Applying:

Life Cycle of a User-Defined Function



Call expression:



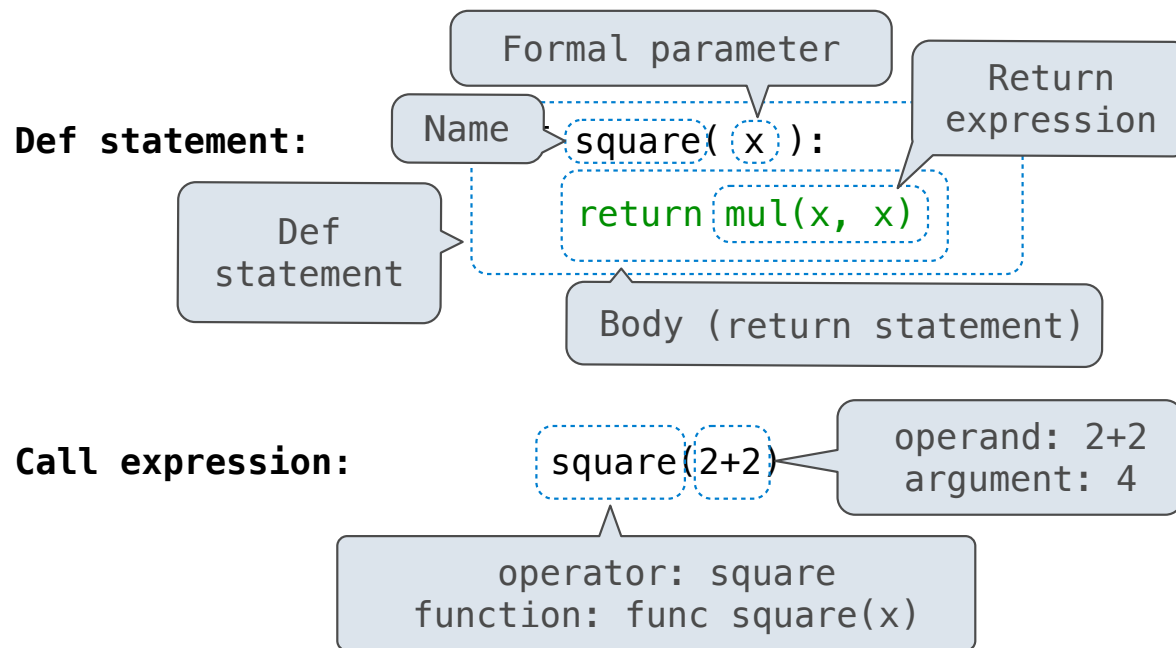
Calling/Applying:

What happens?

A new function is created!

Name bound to that function
in the current frame

Life Cycle of a User-Defined Function



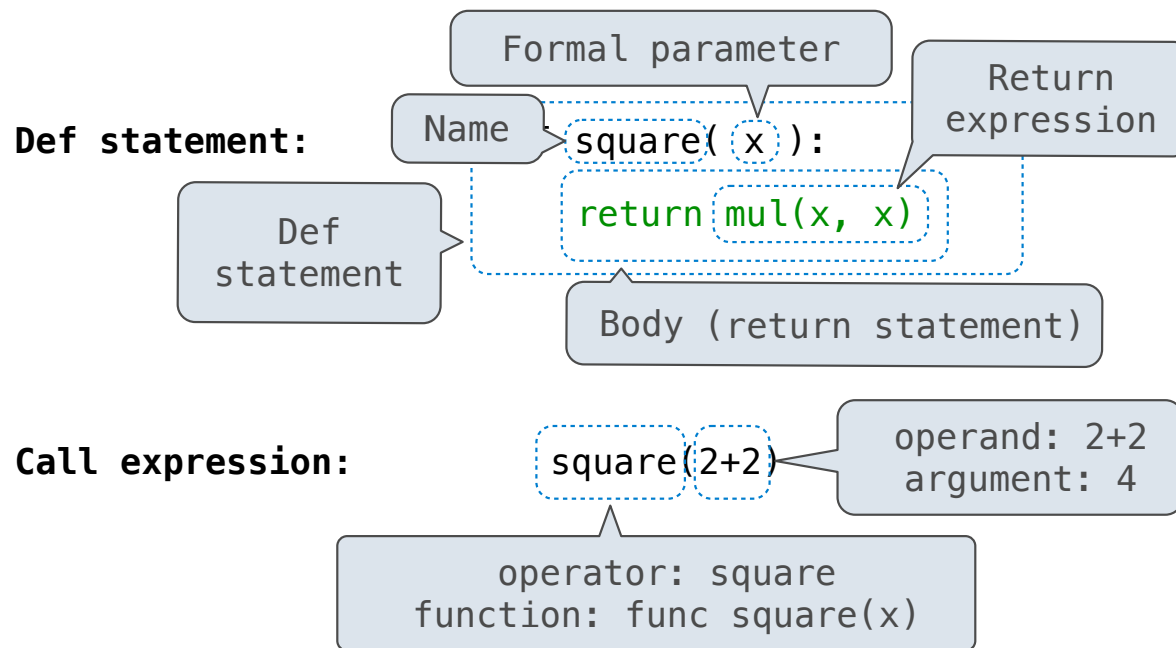
What happens?

A new function is created!

Name bound to that function
in the current frame

Calling/Applying:

Life Cycle of a User-Defined Function



What happens?

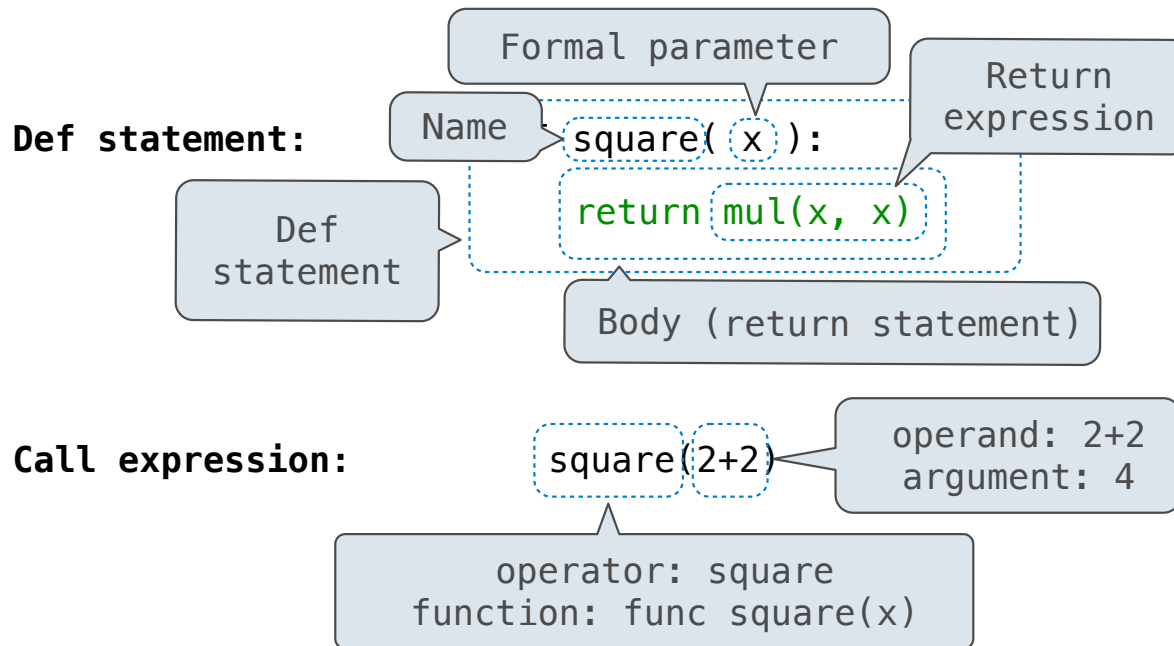
A new function is created!

Name bound to that function
in the current frame

Operator & operands evaluated

Calling/Applying:

Life Cycle of a User-Defined Function



What happens?

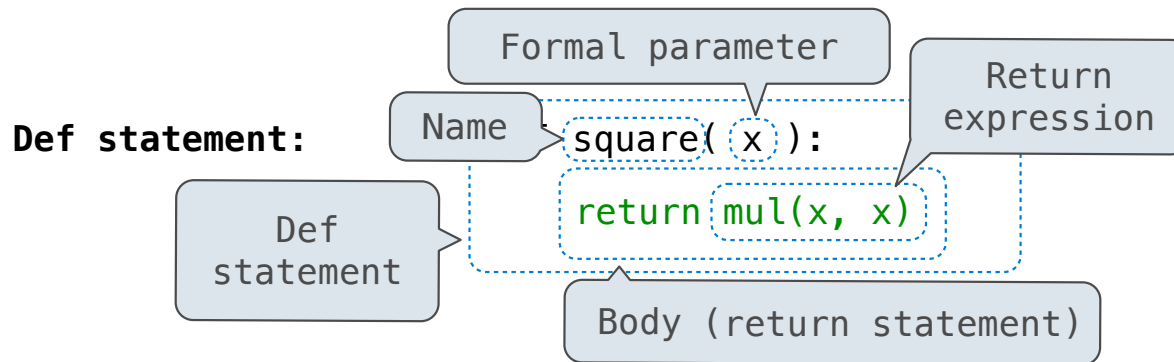
A new function is created!

Name bound to that function in the current frame

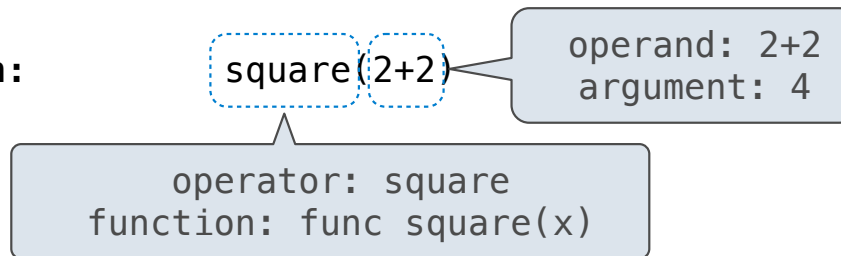
Operator & operands evaluated
Function (value of operator) called on arguments (values of operands)

Calling/Applying:

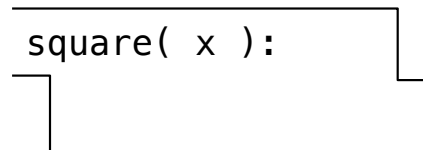
Life Cycle of a User-Defined Function



Call expression:



Calling/Applying:



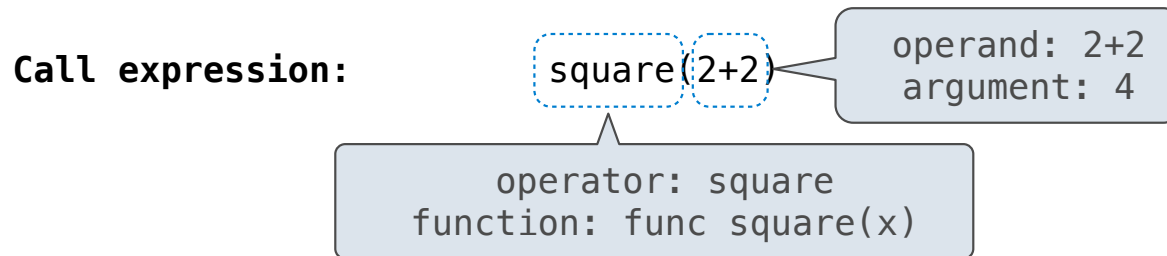
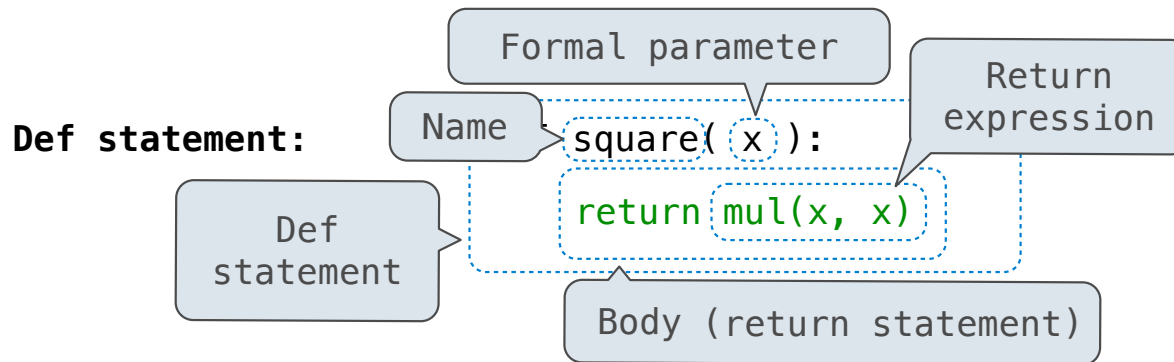
What happens?

A new function is created!

Name bound to that function in the current frame

Operator & operands evaluated
Function (value of operator) called on arguments (values of operands)

Life Cycle of a User-Defined Function



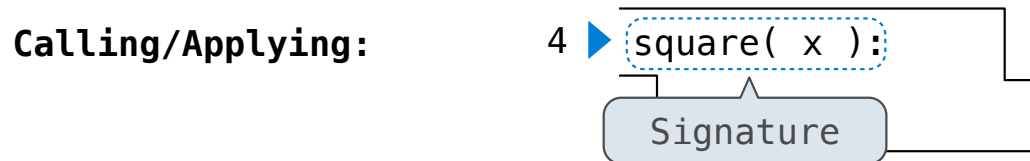
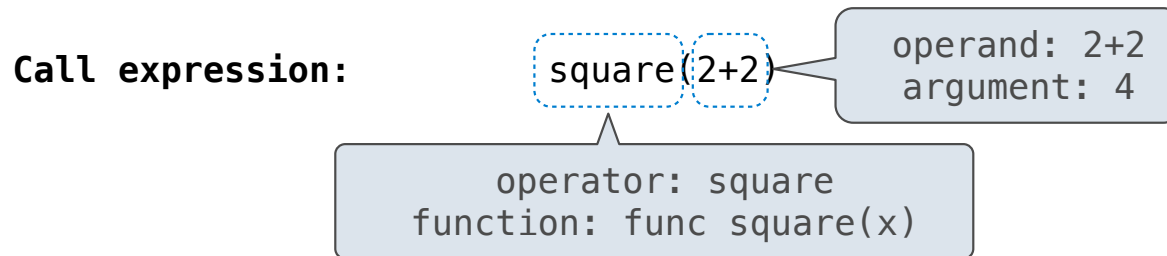
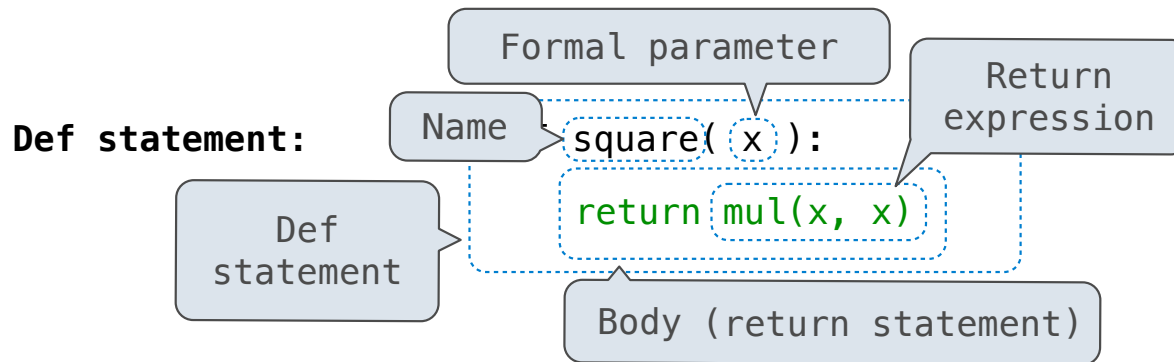
What happens?

A new function is created!

Name bound to that function in the current frame

Operator & operands evaluated
Function (value of operator) called on arguments (values of operands)

Life Cycle of a User-Defined Function



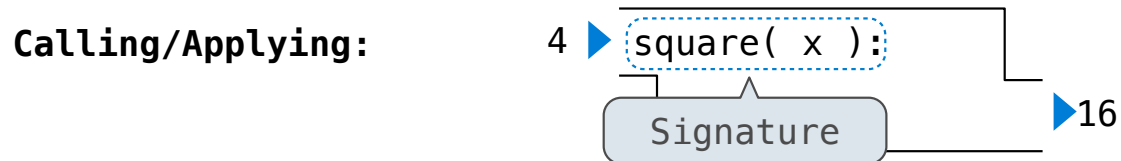
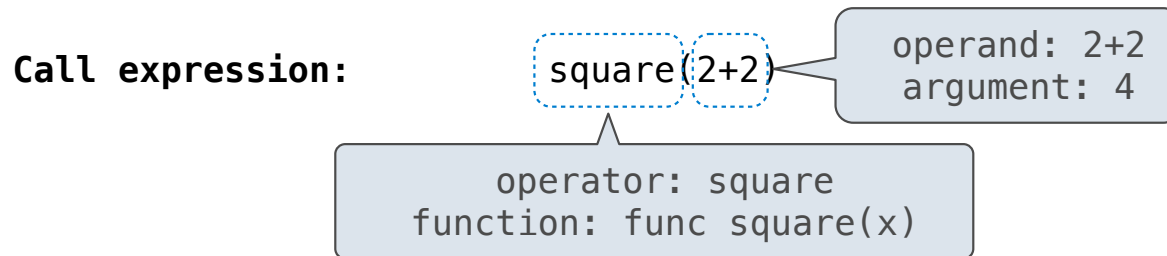
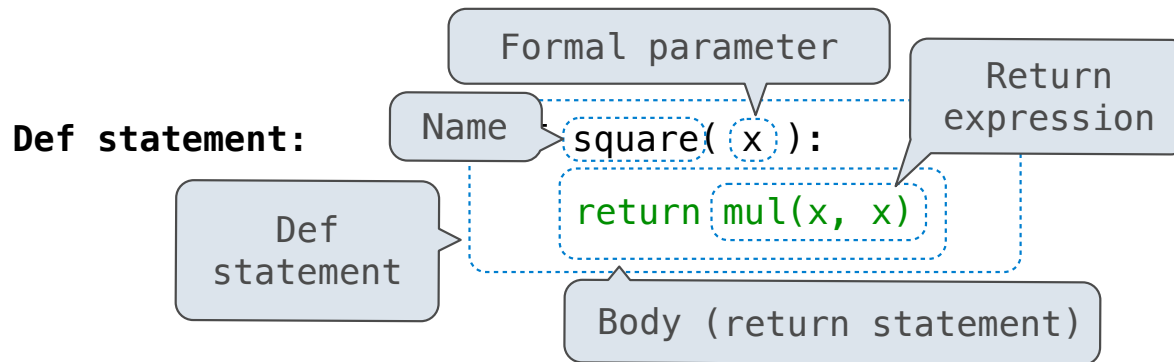
What happens?

A new function is created!

Name bound to that function in the current frame

Operator & operands evaluated
Function (value of operator) called on arguments (values of operands)

Life Cycle of a User-Defined Function



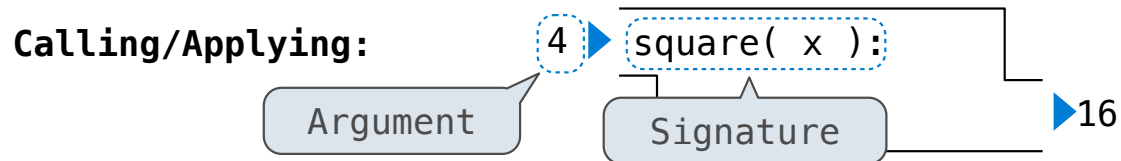
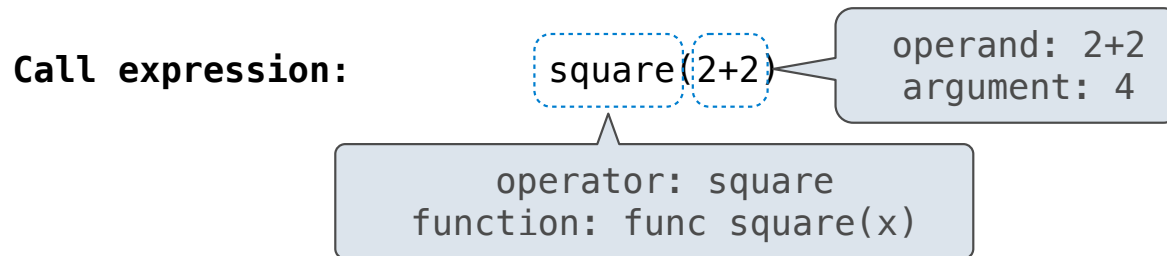
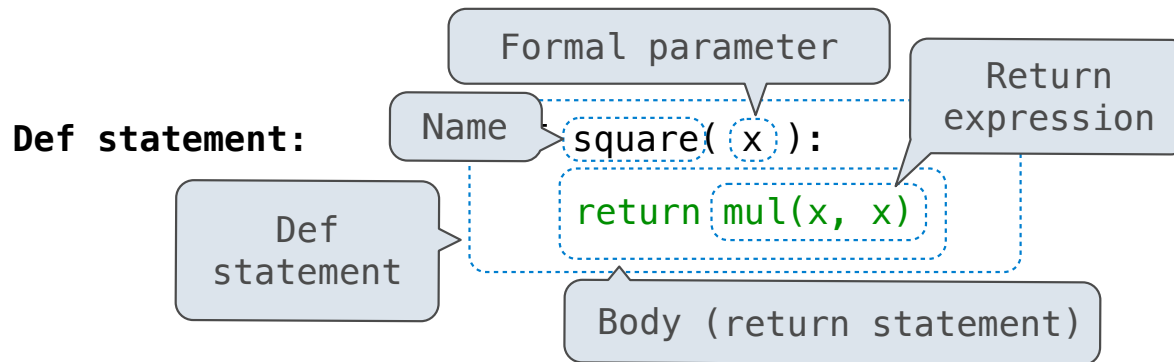
What happens?

A new function is created!

Name bound to that function in the current frame

Operator & operands evaluated
Function (value of operator) called on arguments (values of operands)

Life Cycle of a User-Defined Function



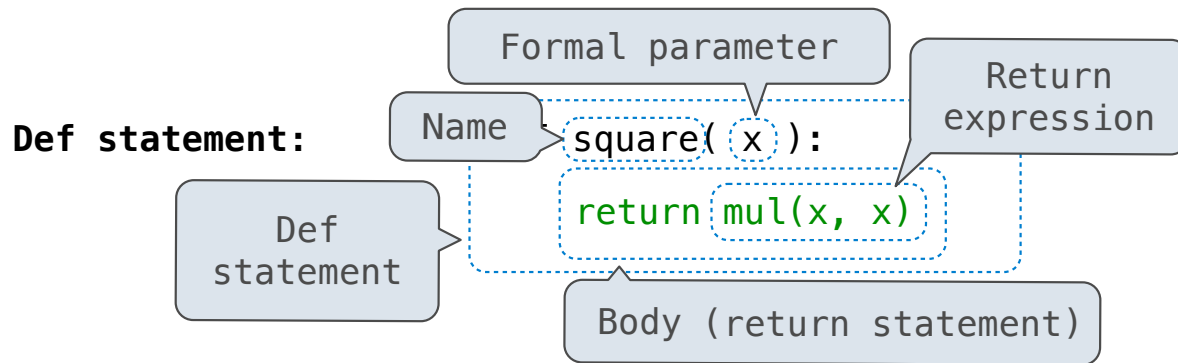
What happens?

A new function is created!

Name bound to that function in the current frame

Operator & operands evaluated
Function (value of operator) called on arguments (values of operands)

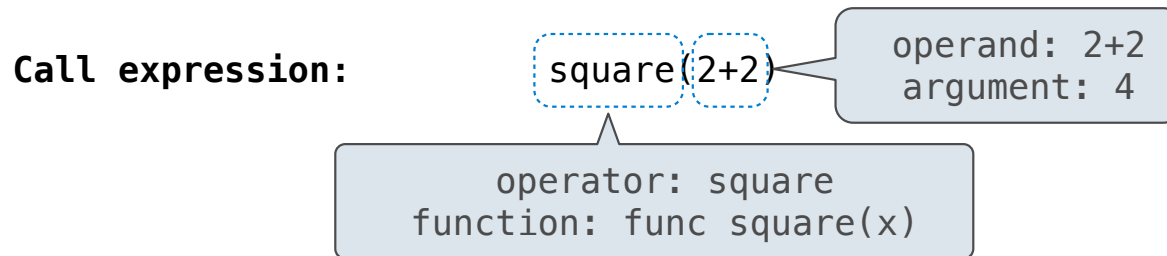
Life Cycle of a User-Defined Function



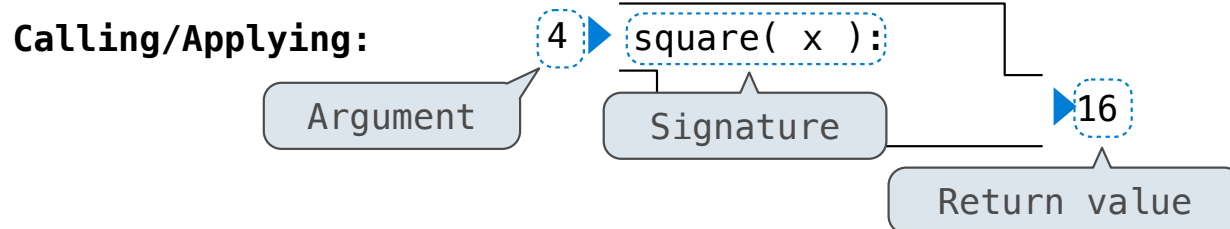
What happens?

A new function is created!

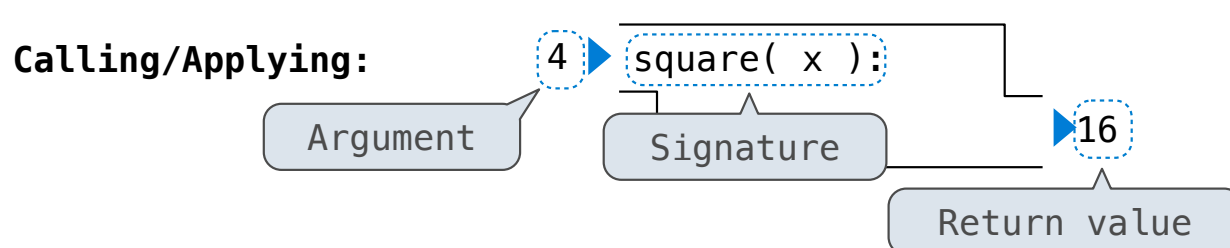
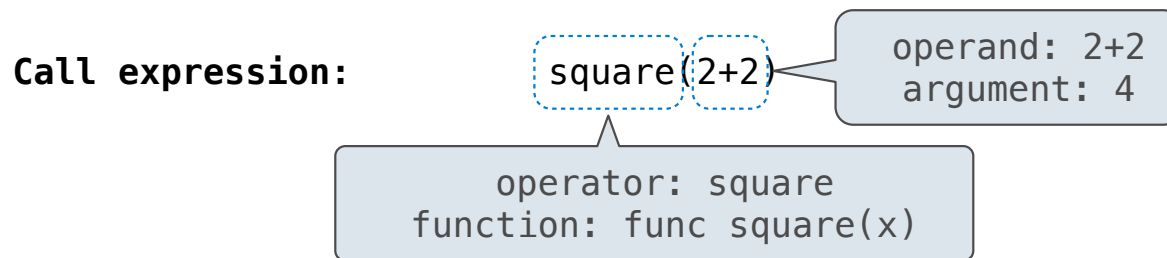
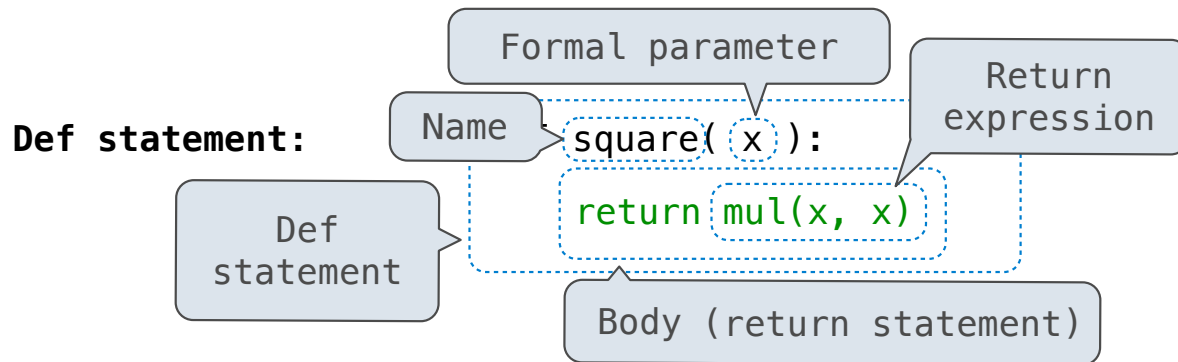
Name bound to that function in the current frame



Operator & operands evaluated
Function (value of operator)
called on arguments
(values of operands)



Life Cycle of a User-Defined Function



What happens?

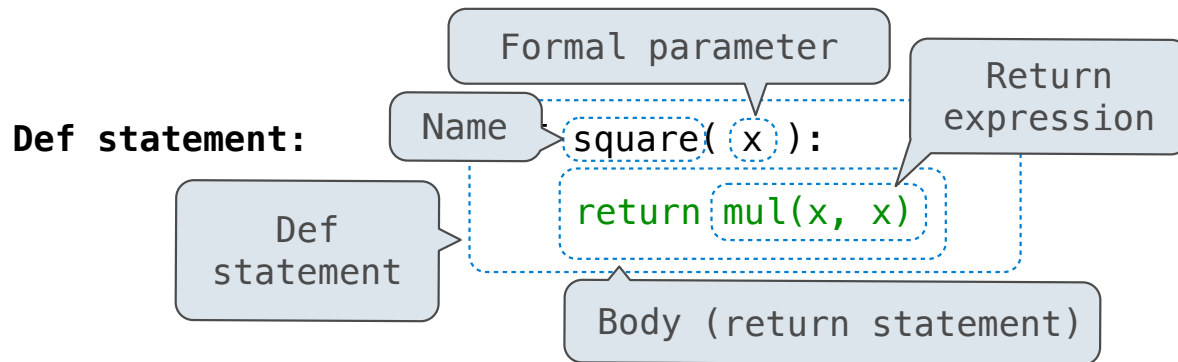
A new function is created!

Name bound to that function
in the current frame

Operator & operands evaluated
Function (value of operator)
called on arguments
(values of operands)

A new frame is created!

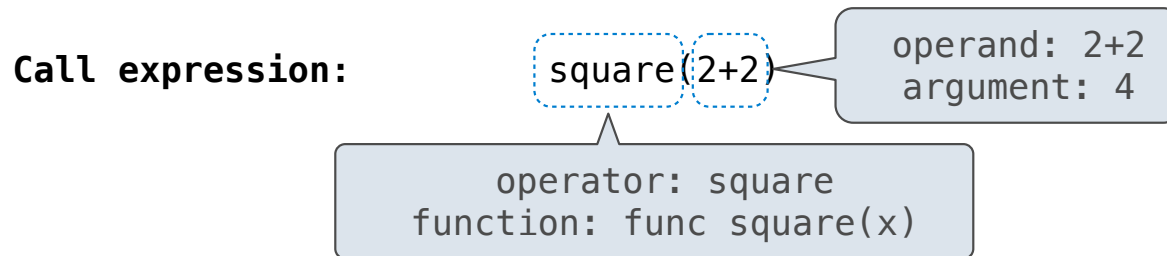
Life Cycle of a User-Defined Function



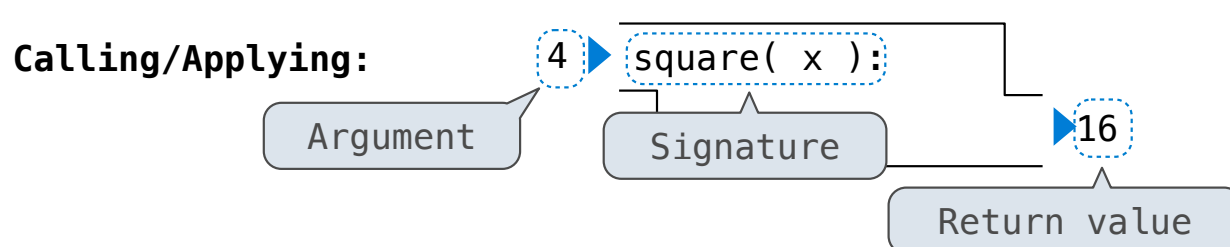
What happens?

A new function is created!

Name bound to that function in the current frame



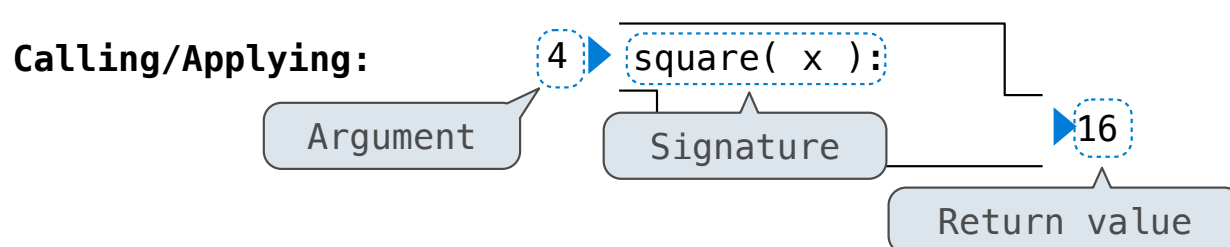
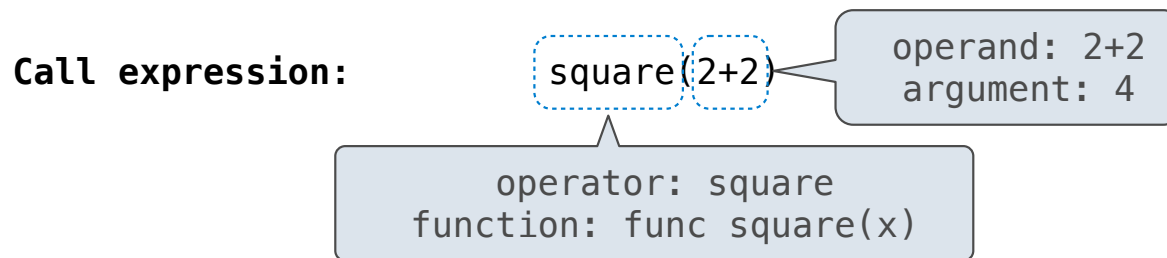
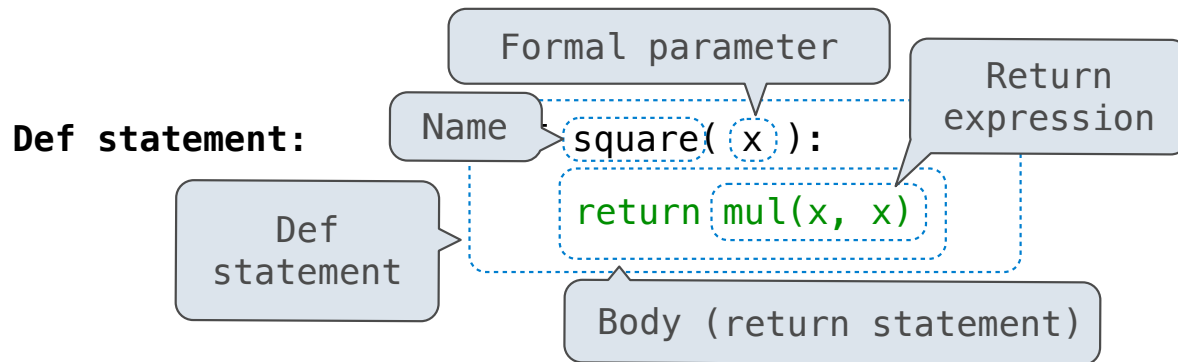
Operator & operands evaluated
Function (value of operator) called on arguments (values of operands)



A new frame is created!

Parameters bound to arguments

Life Cycle of a User-Defined Function



What happens?

A new function is created!

Name bound to that function
in the current frame

Operator & operands evaluated
Function (value of operator)
called on arguments
(values of operands)

A new frame is created!

Parameters bound to arguments

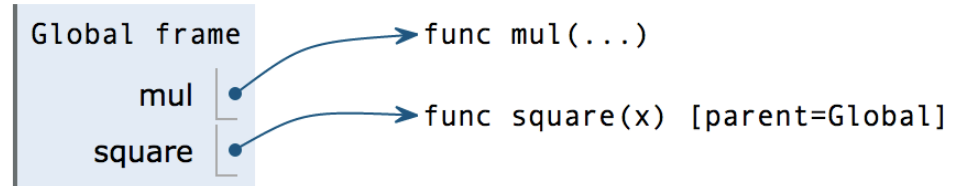
Body is executed

Multiple Environments in One Diagram!

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

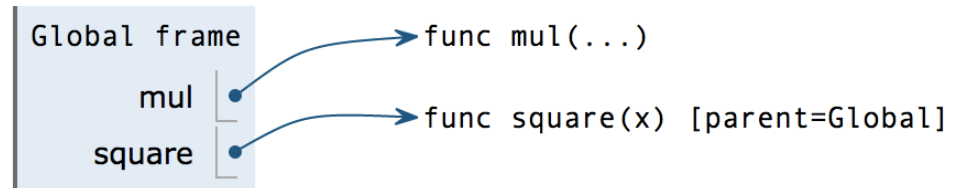
Multiple Environments in One Diagram!

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



Multiple Environments in One Diagram!

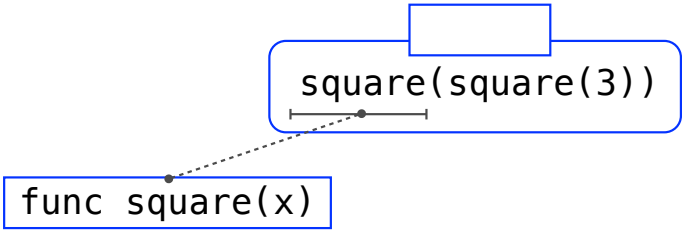
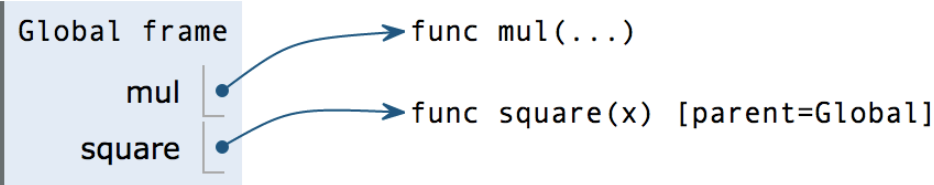
```
1 from operator import mul
→ 2 def square(x):
3     return mul(x, x)
→ 4 square(square(3))
```



square(square(3))

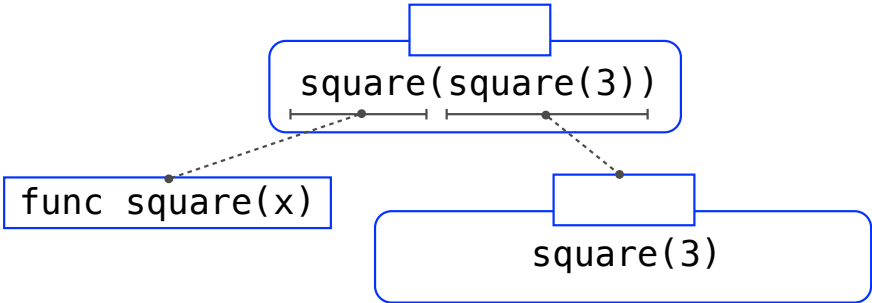
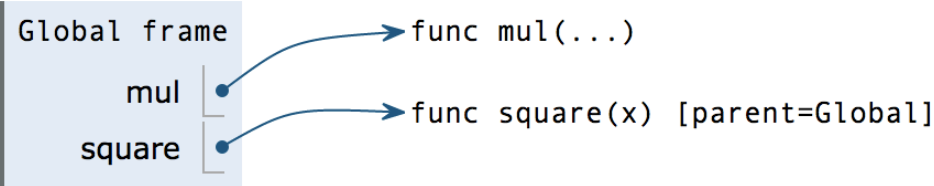
Multiple Environments in One Diagram!

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



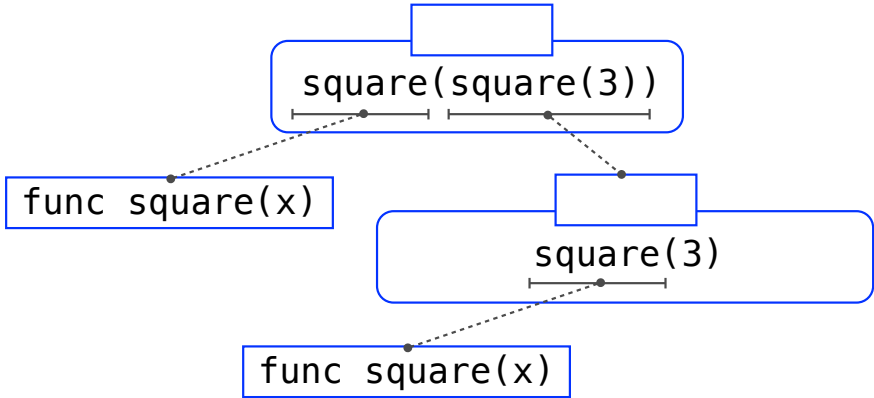
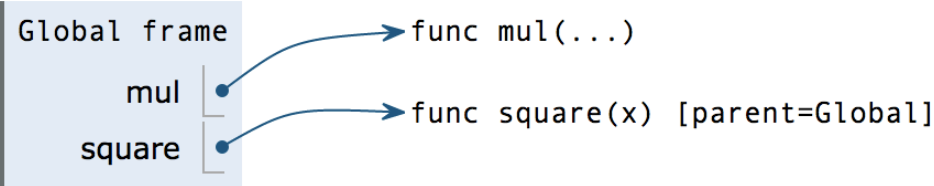
Multiple Environments in One Diagram!

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



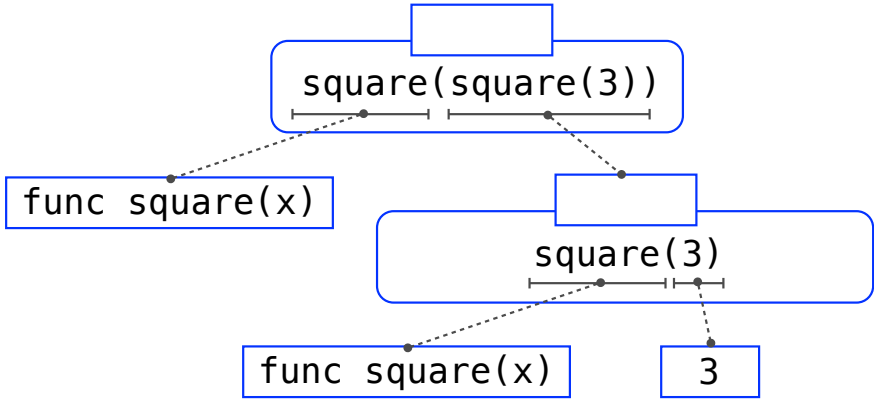
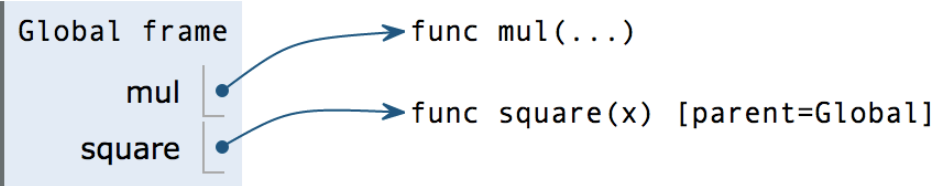
Multiple Environments in One Diagram!

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



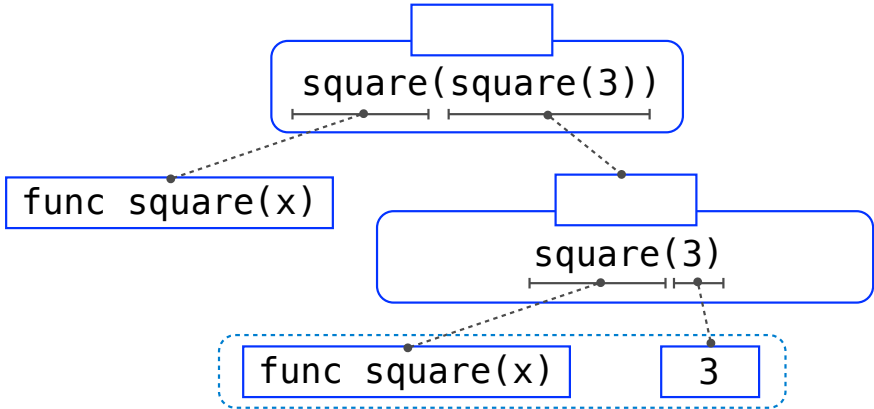
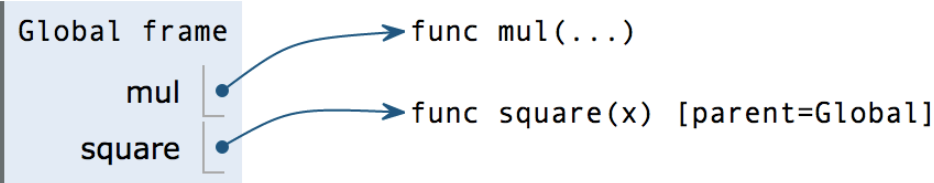
Multiple Environments in One Diagram!

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



Multiple Environments in One Diagram!

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



Multiple Environments in One Diagram!

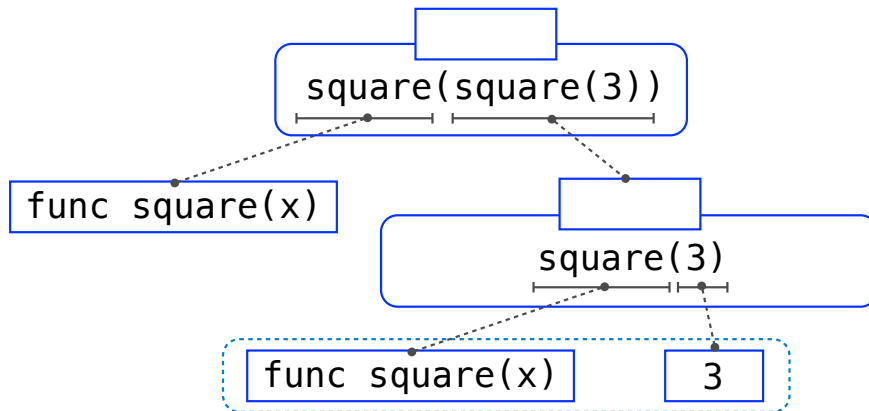
```
1 from operator import mul
2 def square(x):
3     return mul(x, x)
4 square(square(3))
```

Global frame

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

f1: square [parent=Global]

x 3

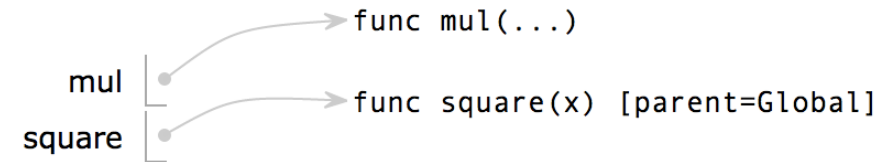


Multiple Environments in One Diagram!

```

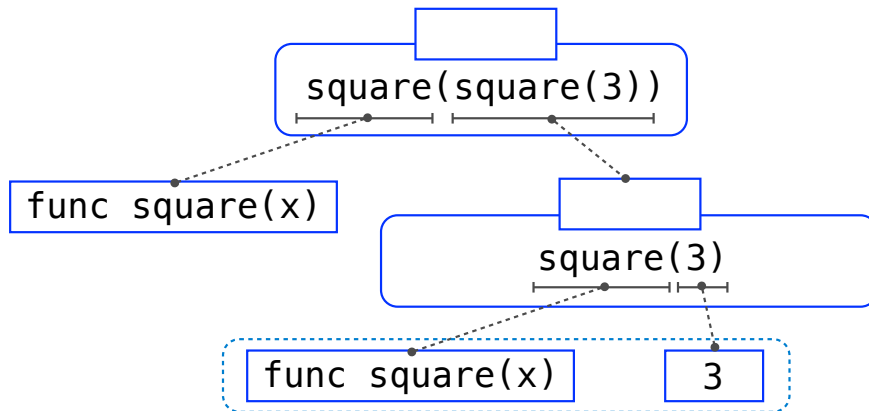
1 from operator import mul
2 def square(x):
3     return mul(x, x)
4 square(square(3))
    
```

Global frame



f1: square [parent=Global]

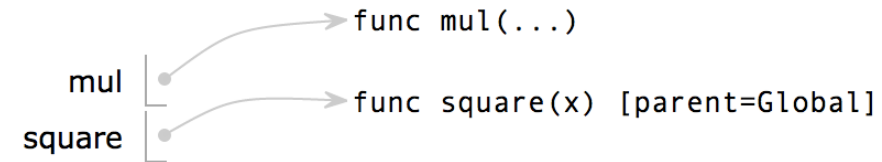
x	3
Return value	9



Multiple Environments in One Diagram!

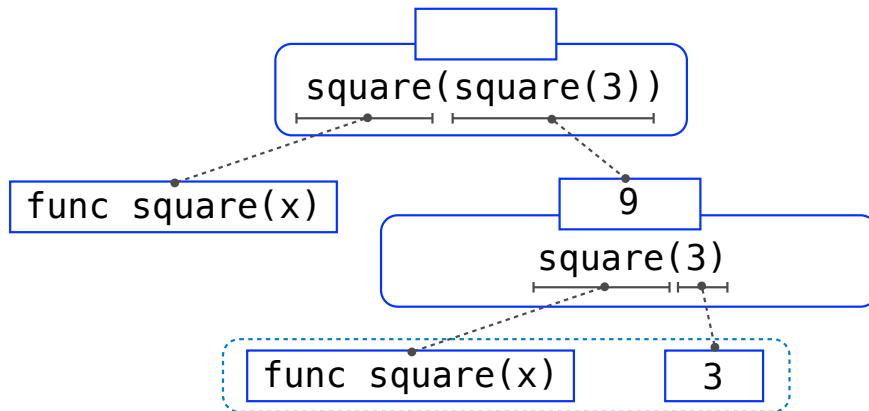
```
1 from operator import mul
2 def square(x):
3     return mul(x, x)
4 square(square(3))
```

Global frame



f1: square [parent=Global]

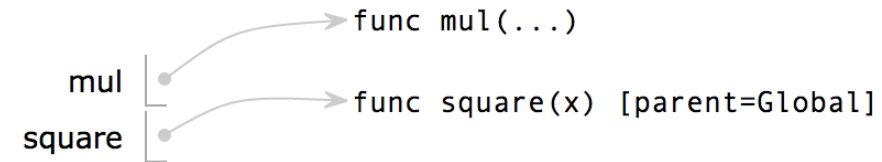
```
x | 3
Return value | 9
```



Multiple Environments in One Diagram!

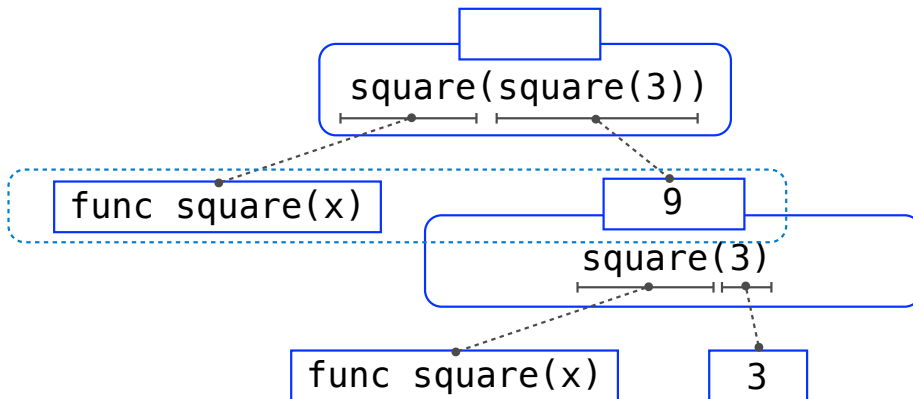
```
1 from operator import mul
2 def square(x):
3     return mul(x, x)
4 square(square(3))
```

Global frame



f1: square [parent=Global]

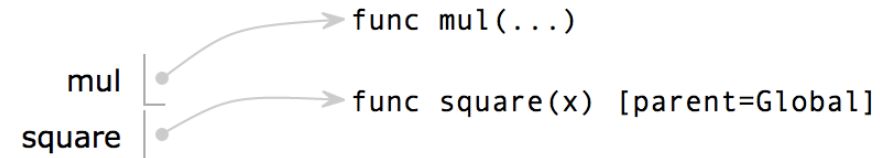
```
x | 3
Return value | 9
```



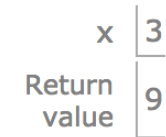
Multiple Environments in One Diagram!

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

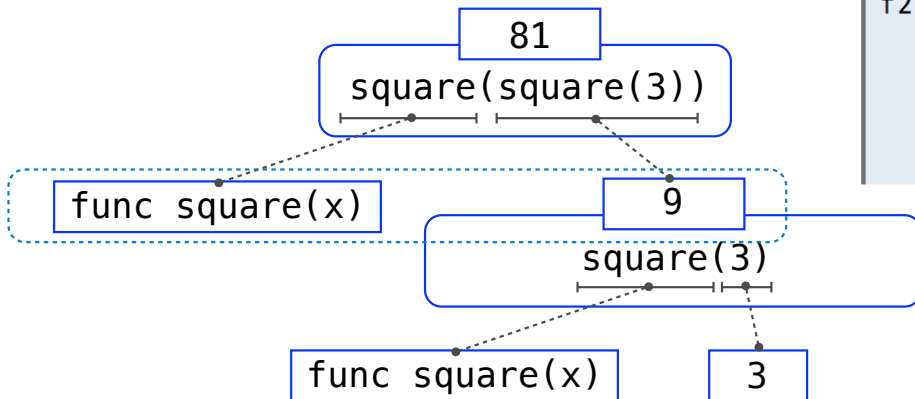
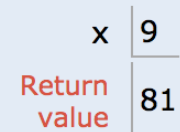
Global frame



f1: square [parent=Global]



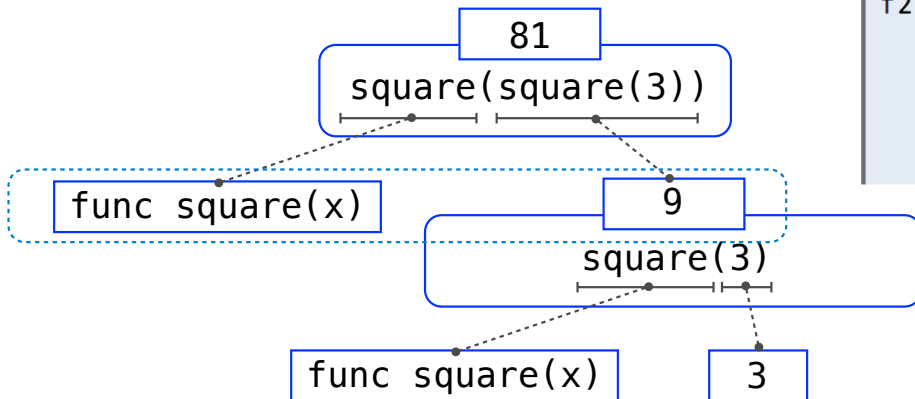
f2: square [parent=Global]



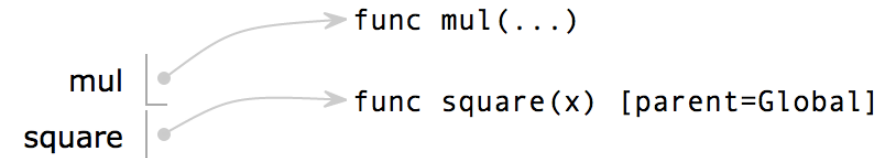
Multiple Environments in One Diagram!

```

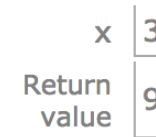
1 from operator import mul
2 def square(x):
3     return mul(x, x)
4 square(square(3))
    
```



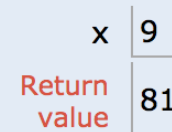
Global frame



f1: square [parent=Global]



f2: square [parent=Global]

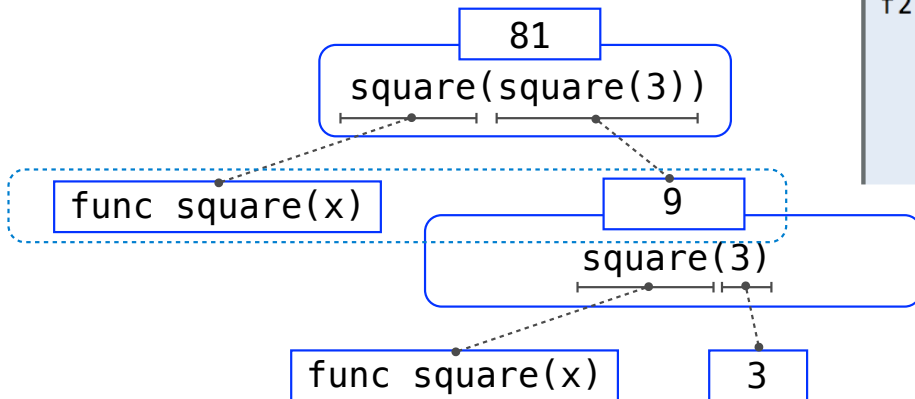


An **environment** is a sequence of frames.

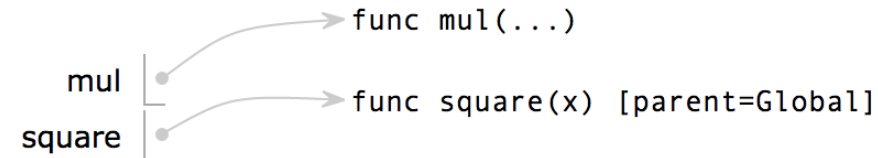
Multiple Environments in One Diagram!

```

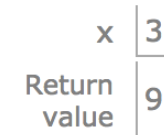
1 from operator import mul
2 def square(x):
3     return mul(x, x)
4 square(square(3))
    
```



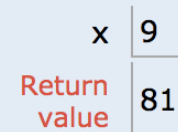
Global frame



f1: square [parent=Global]



f2: square [parent=Global]



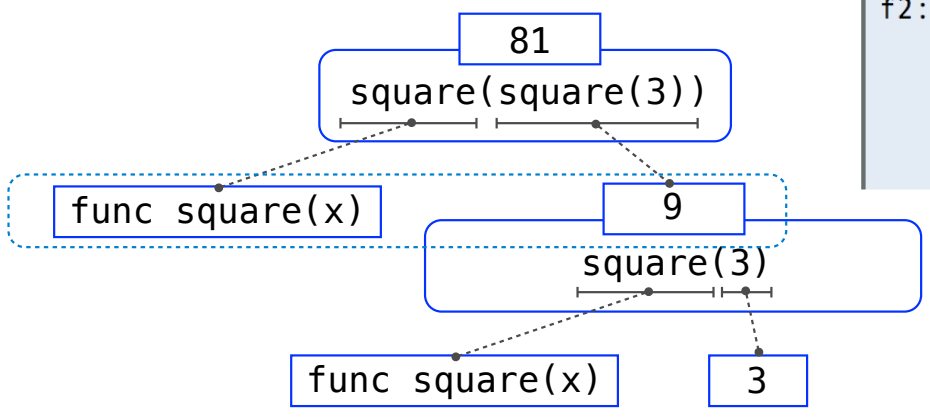
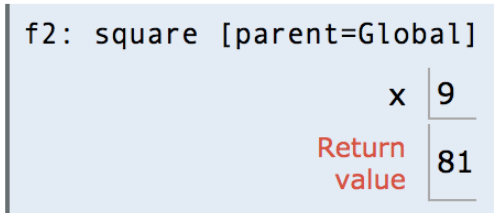
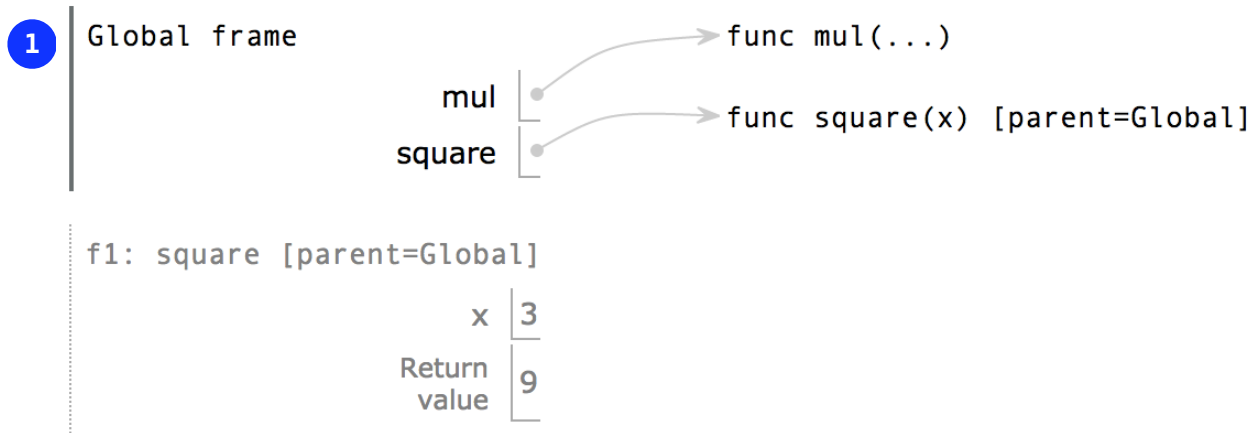
An **environment** is a sequence of frames.

- The global frame alone **OR**
- A local frame, then the global frame

Multiple Environments in One Diagram!

```

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



- An **environment** is a sequence of frames.
- The global frame alone **OR**
 - A local frame, then the global frame

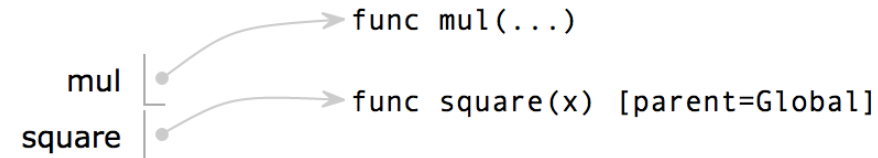
Multiple Environments in One Diagram!

```

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

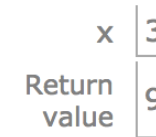
```

1 Global frame

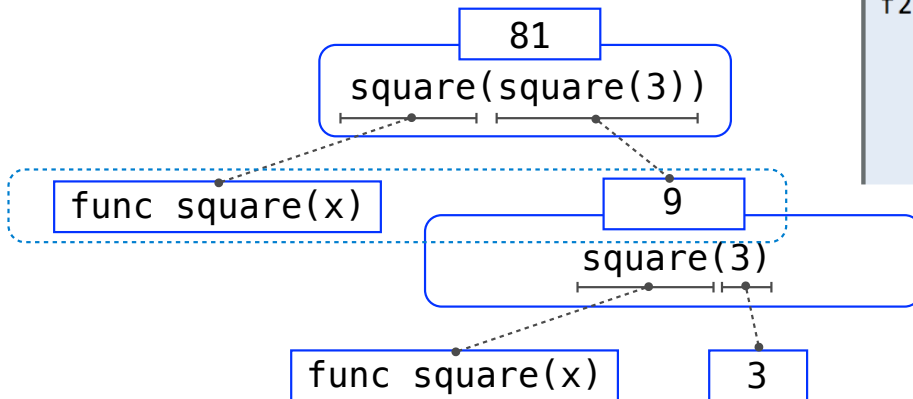
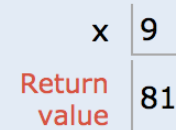


2

1 f1: square [parent=Global]



f2: square [parent=Global]



An **environment** is a sequence of frames.

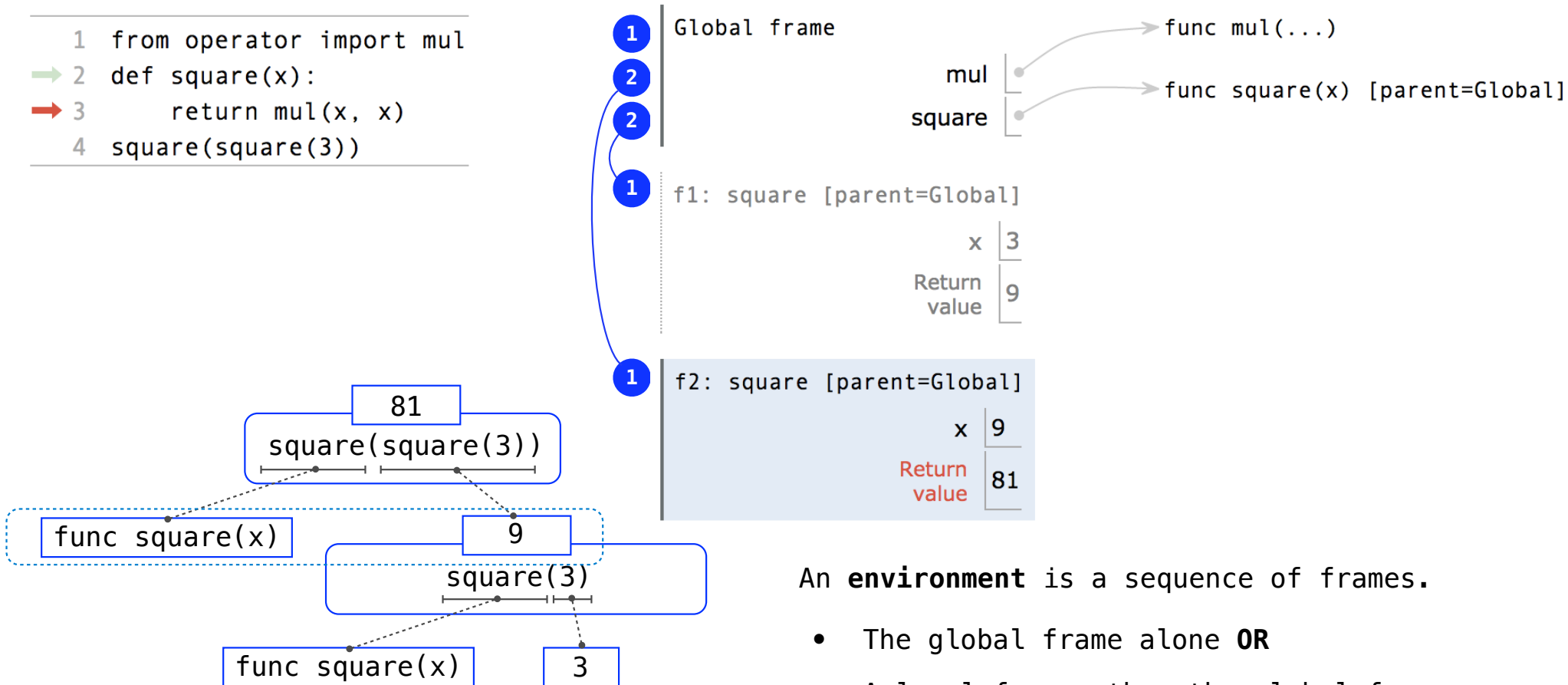
- The global frame alone **OR**
- A local frame, then the global frame

Multiple Environments in One Diagram!

```

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

```

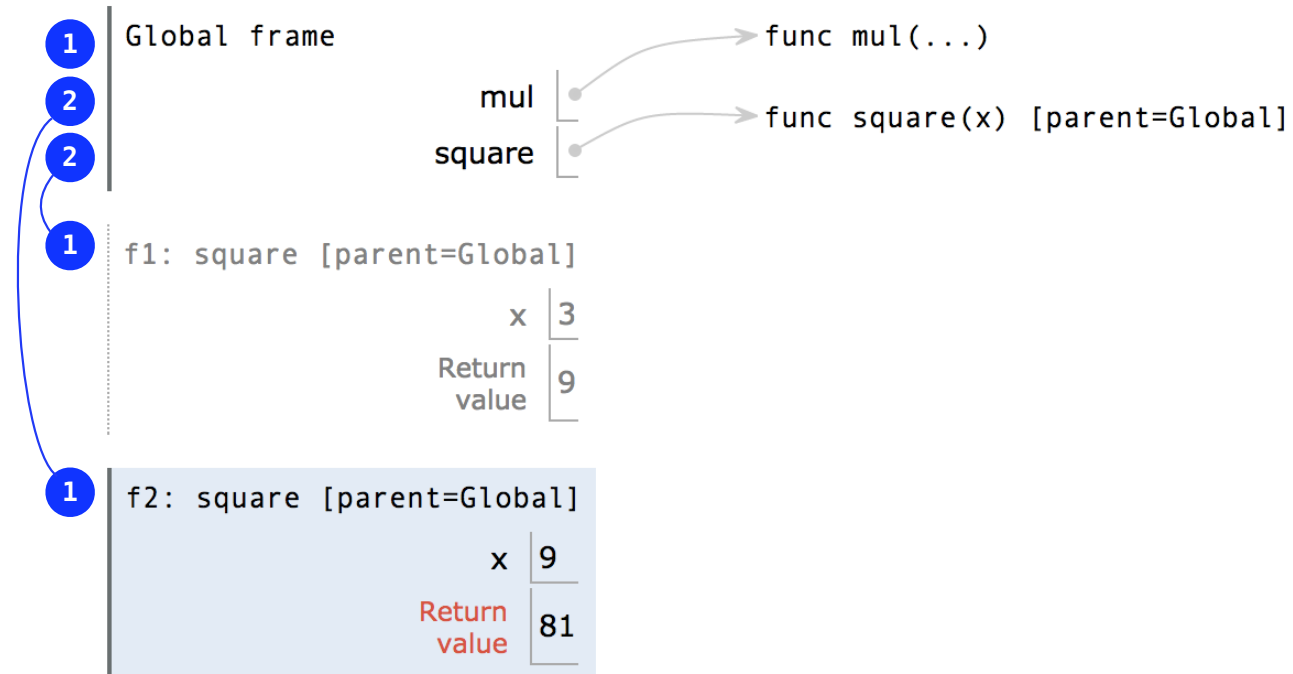


An **environment** is a sequence of frames.

- The global frame alone **OR**
- A local frame, then the global frame

Names Have No Meaning Without Environments

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



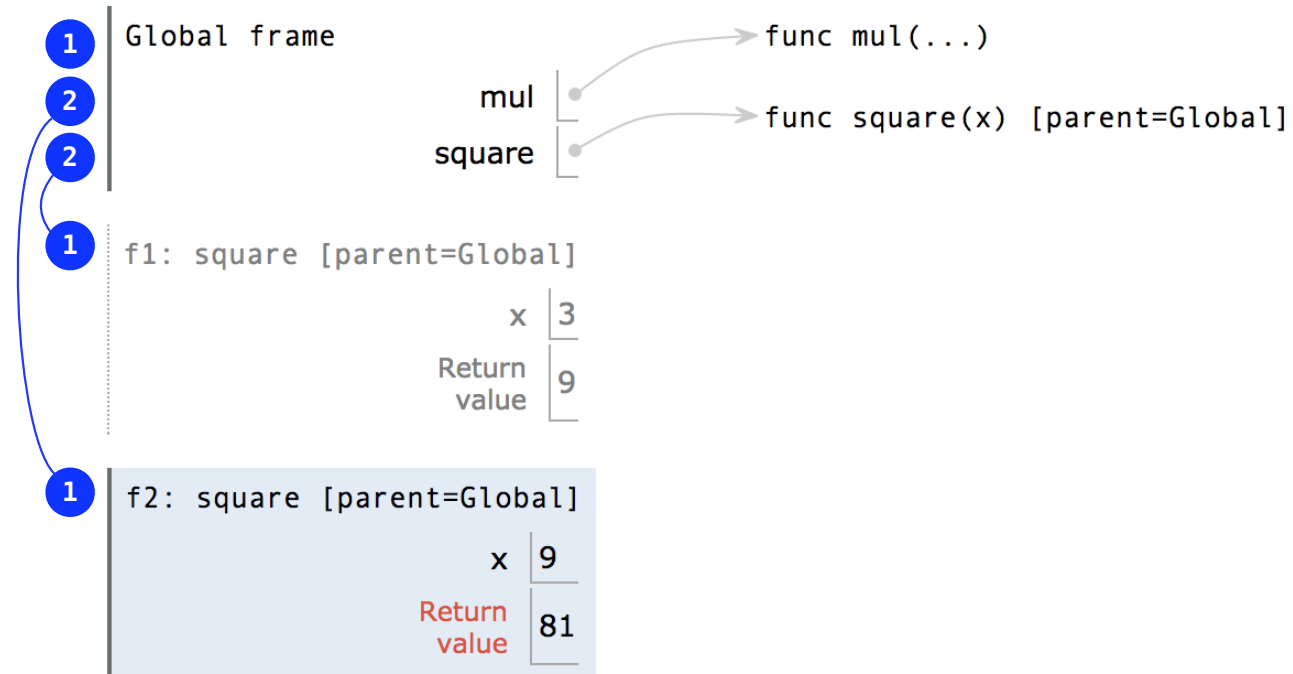
An environment is a sequence of frames.

- The global frame alone **OR**
- A local frame, then the global frame

Names Have No Meaning Without Environments

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

Every **expression is evaluated in the context of an environment.**



An environment is a sequence of frames.

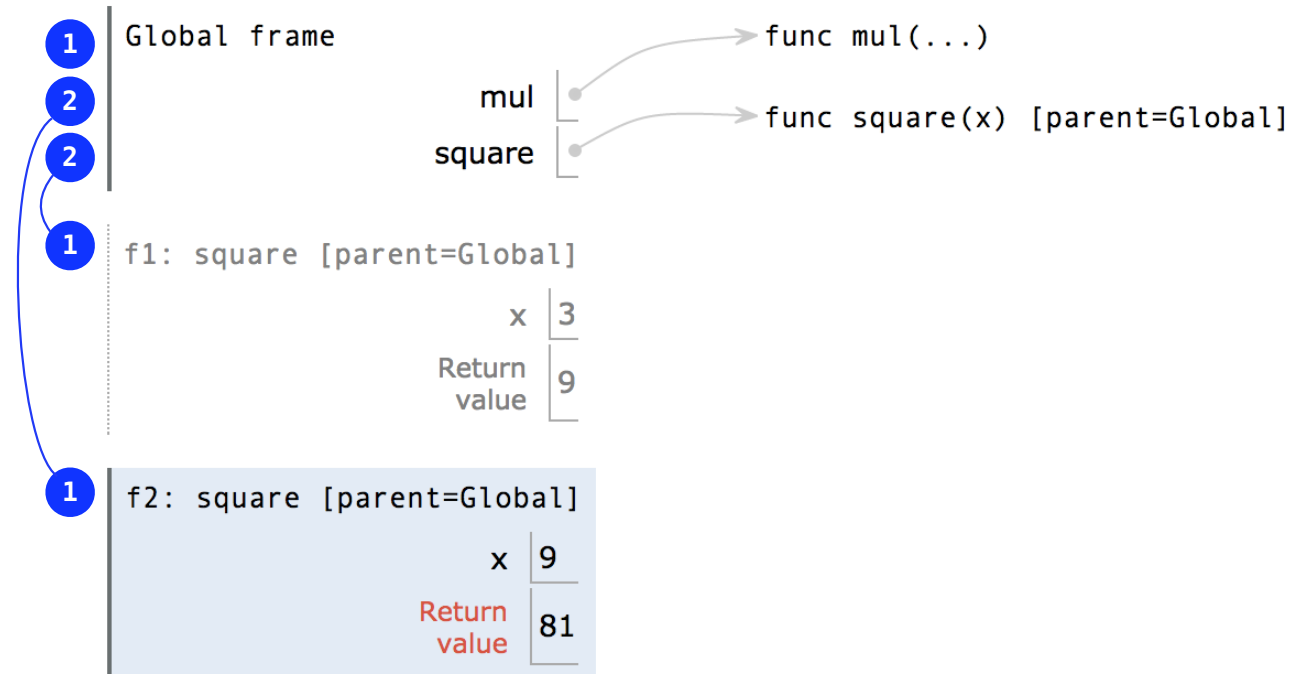
- The global frame alone **OR**
- A local frame, then the global frame

Names Have No Meaning Without Environments

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

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.

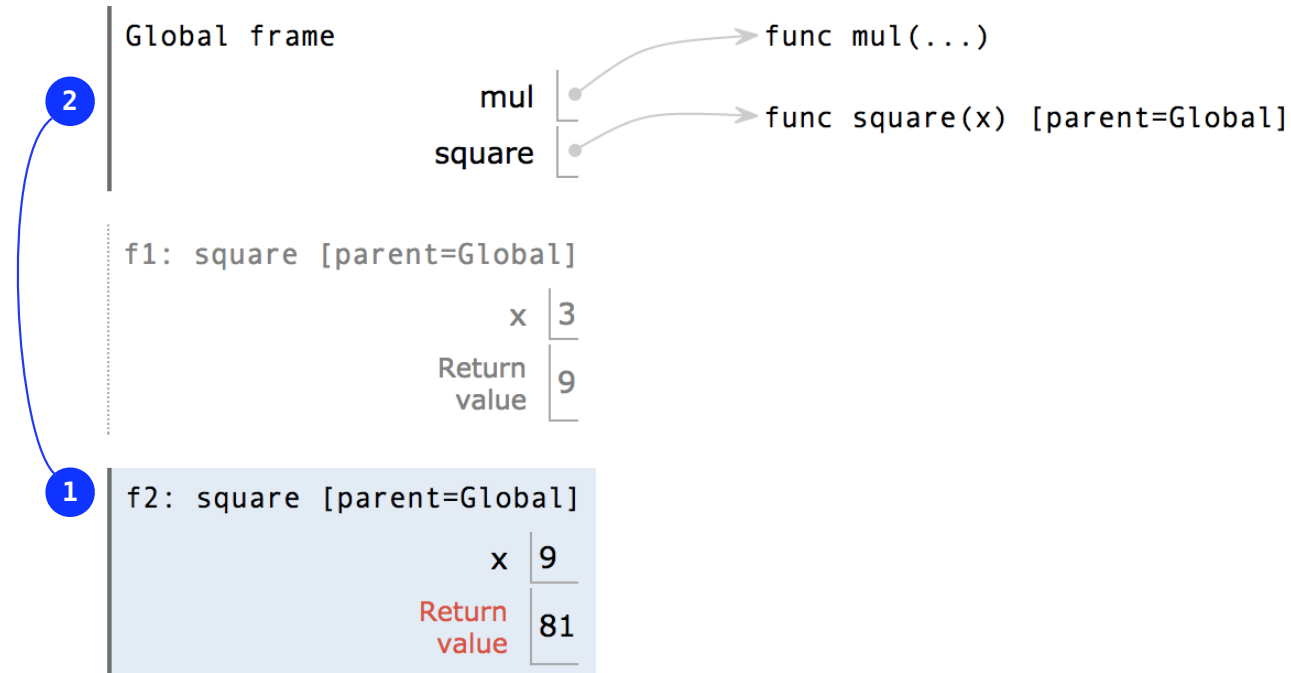
- The global frame alone **OR**
- A local frame, then the global frame

Names Have No Meaning Without Environments

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

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 **OR**
- A local frame, then the global frame

Names Have No Meaning Without Environments

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

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.

Global frame

mul	→	func mul(...)
square	→	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.

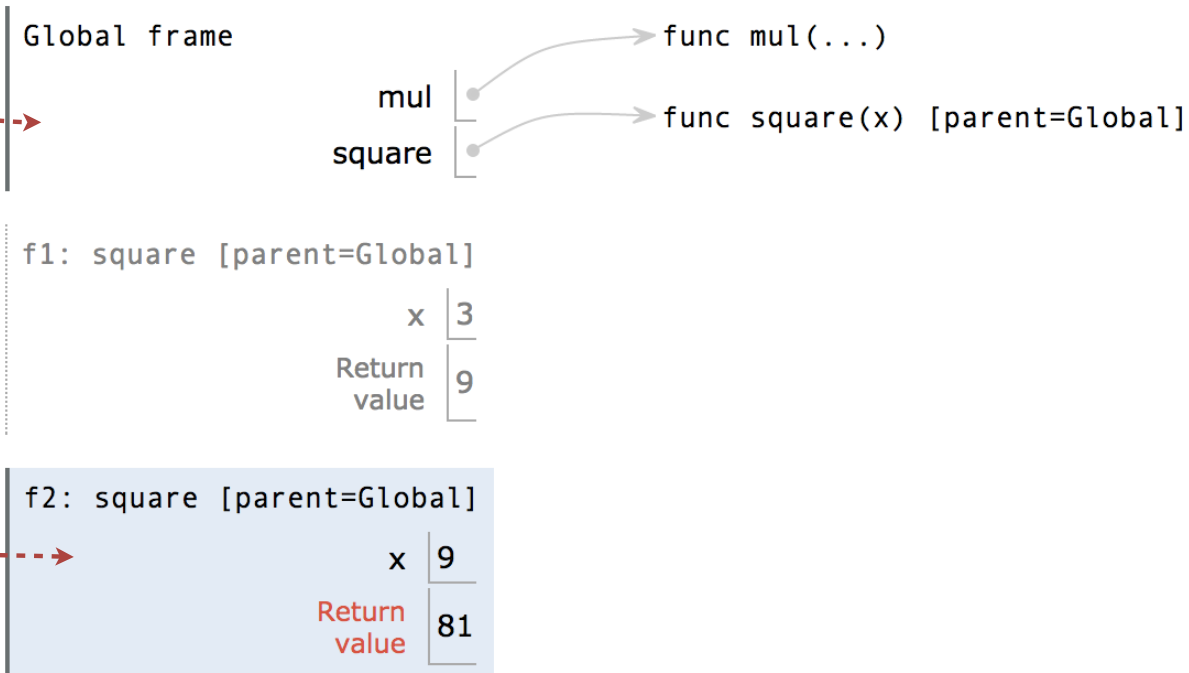
- The global frame alone **OR**
- A local frame, then the global frame

Names Have No Meaning Without Environments

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

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.

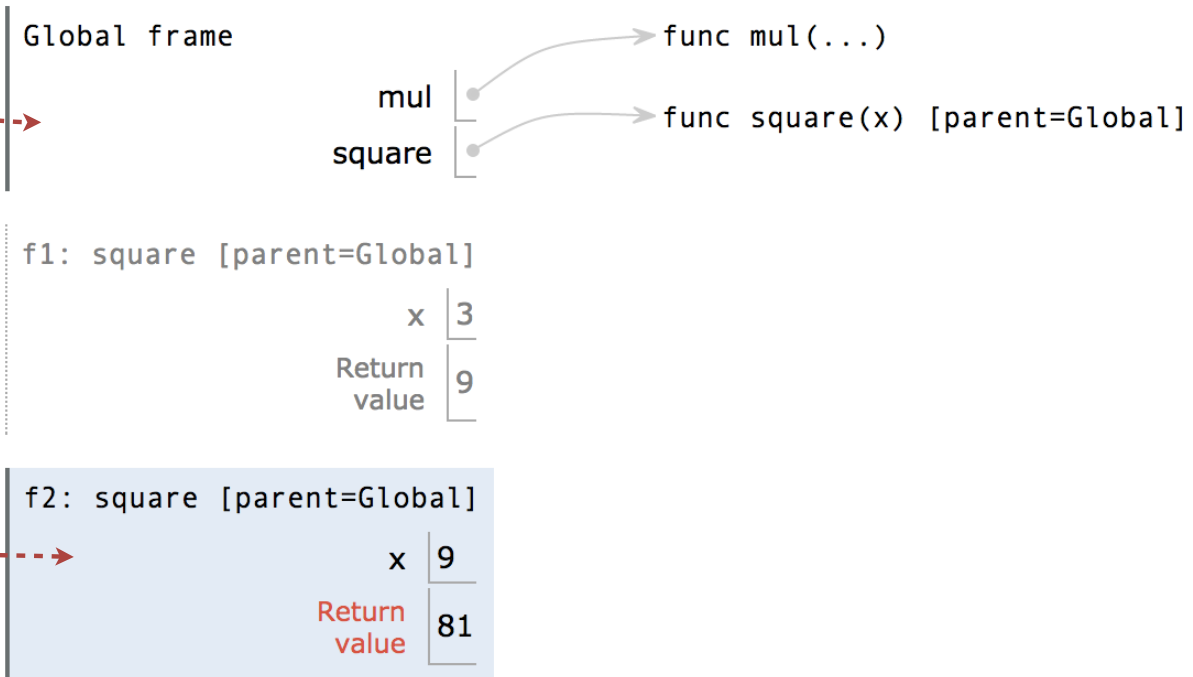


An environment is a sequence of frames.

- The global frame alone **OR**
- A local frame, then the global frame

Names Have No Meaning Without Environments

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



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 **OR**
- A local frame, then the global frame

(Demo)