

61A Lecture 3

Announcements

Multiple Environments

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

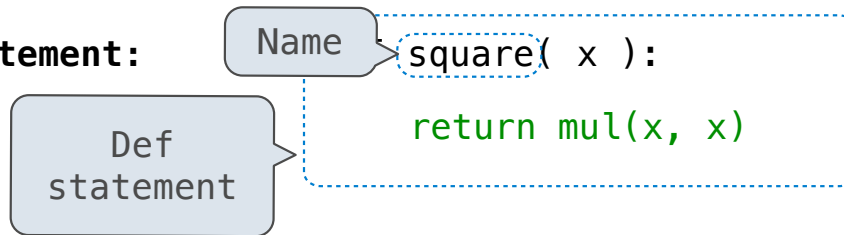
Call expression:

Calling/Applying:

Life Cycle of a User-Defined Function

What happens?

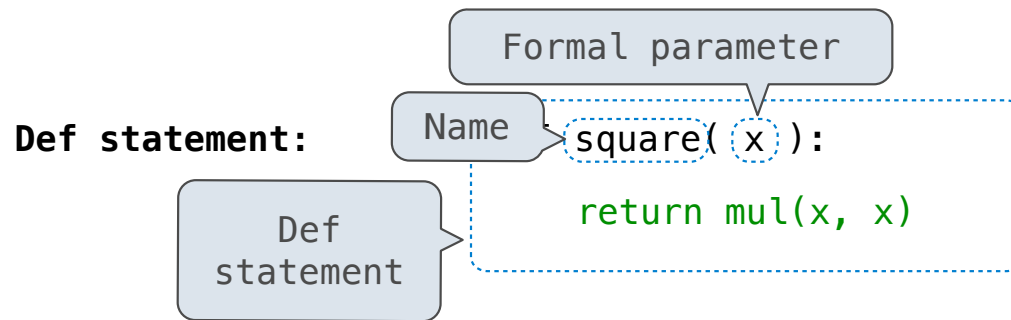
Def statement:



Call expression:

Calling/Applying:

Life Cycle of a User-Defined Function

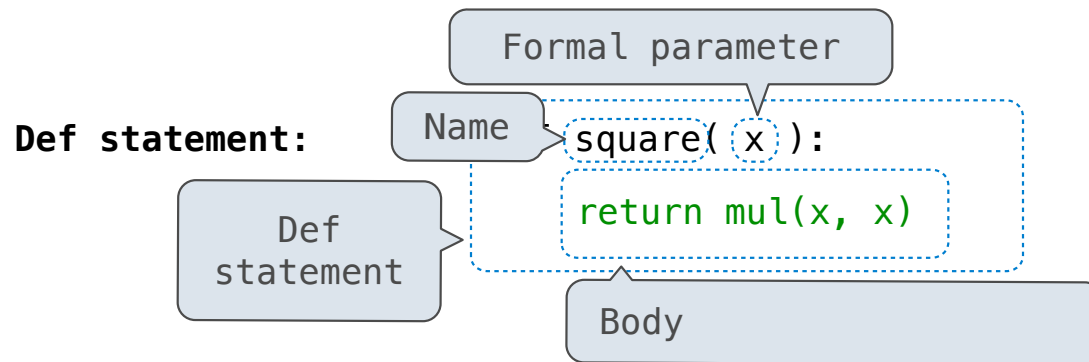


What happens?

Call expression:

Calling/Applying:

Life Cycle of a User-Defined Function

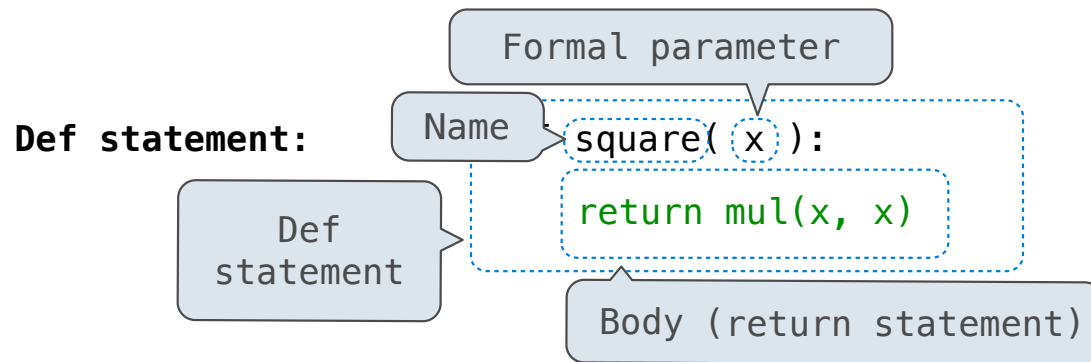


What happens?

Call expression:

Calling/Applying:

Life Cycle of a User-Defined Function

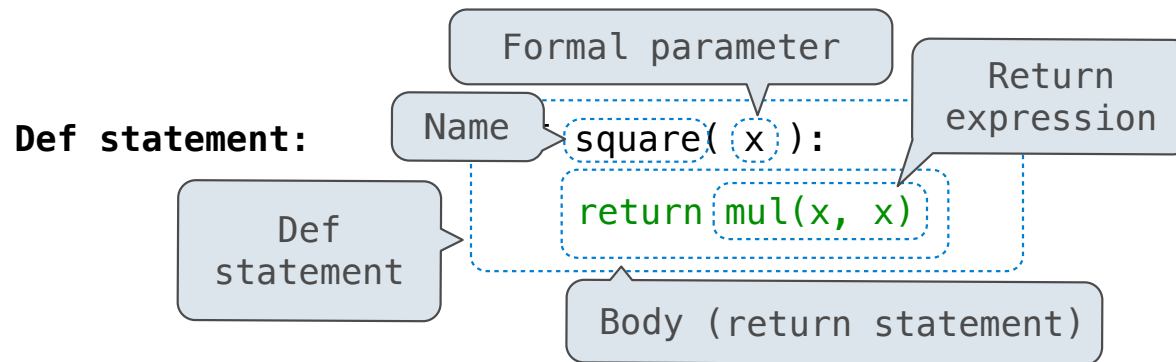


What happens?

Call expression:

Calling/Applying:

Life Cycle of a User-Defined Function

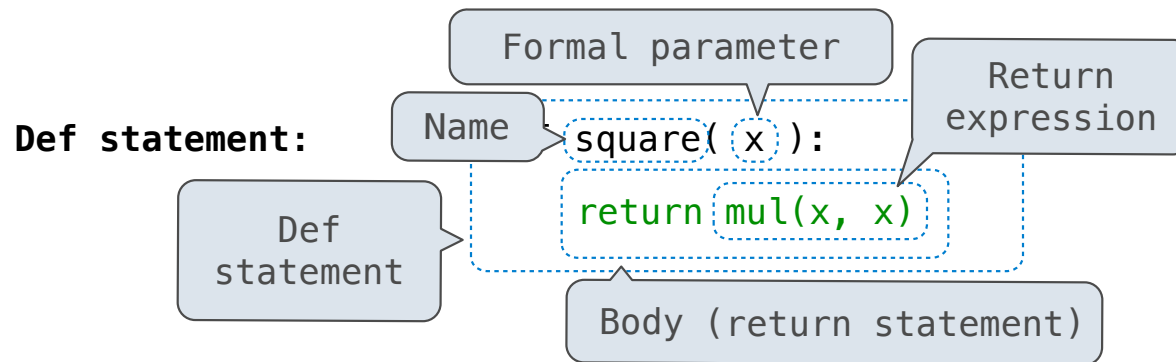


What happens?

Call expression:

Calling/Applying:

Life Cycle of a User-Defined Function



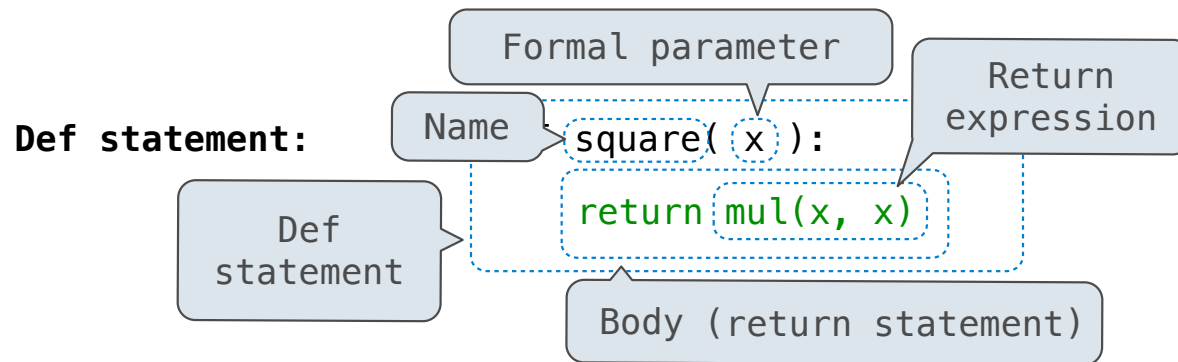
What happens?

A new function is created!

Call expression:

Calling/Applying:

Life Cycle of a User-Defined Function



What happens?

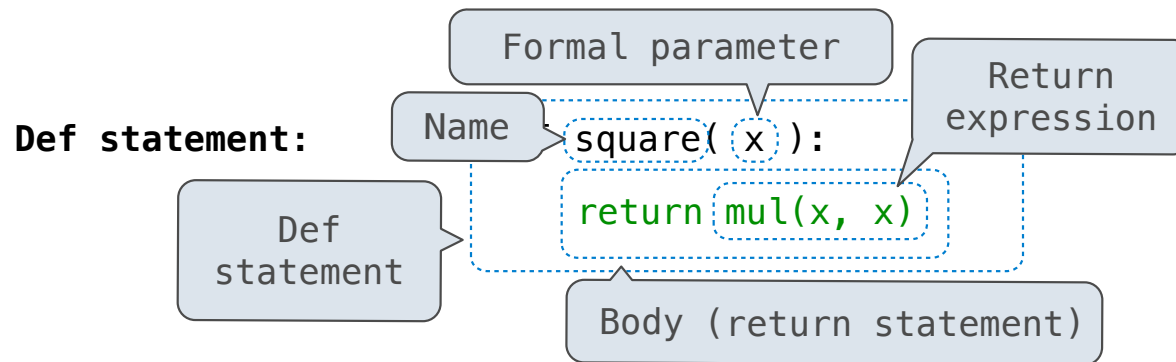
A new function is created!

Name bound to that function
in the current frame

Call expression:

Calling/Applying:

Life Cycle of a User-Defined Function



What happens?

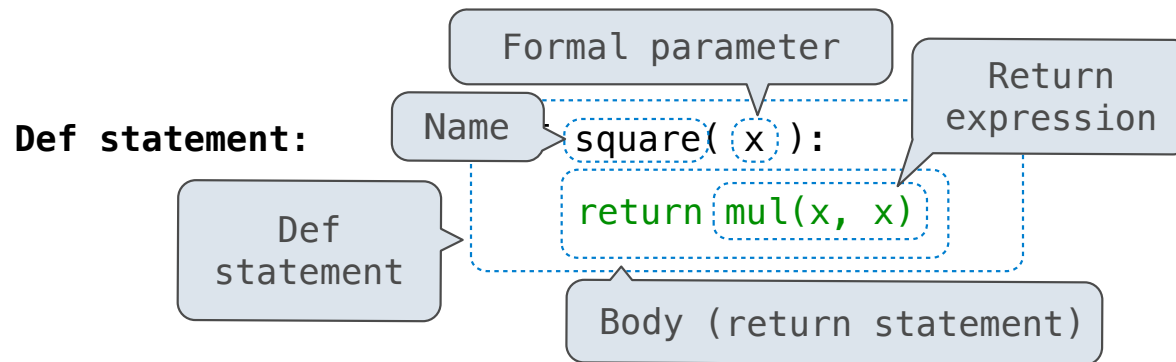
A new function is created!

Name bound to that function
in the current frame

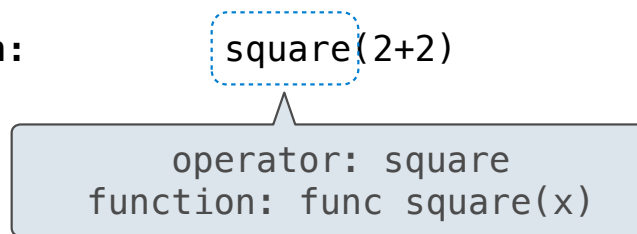
Call expression: `square(2+2)`

Calling/Applying:

Life Cycle of a User-Defined Function



Call expression:



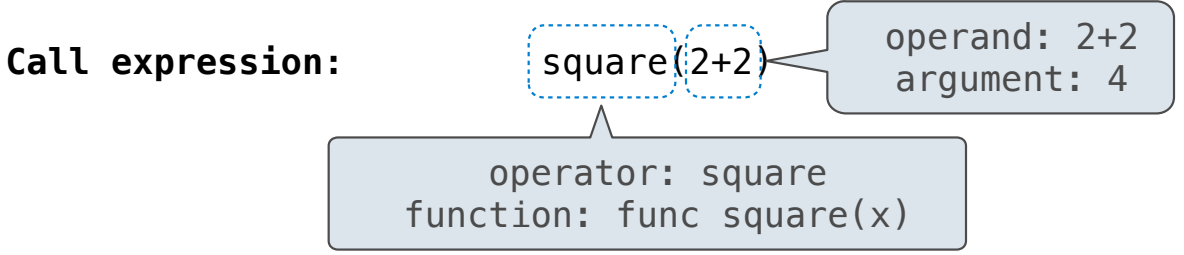
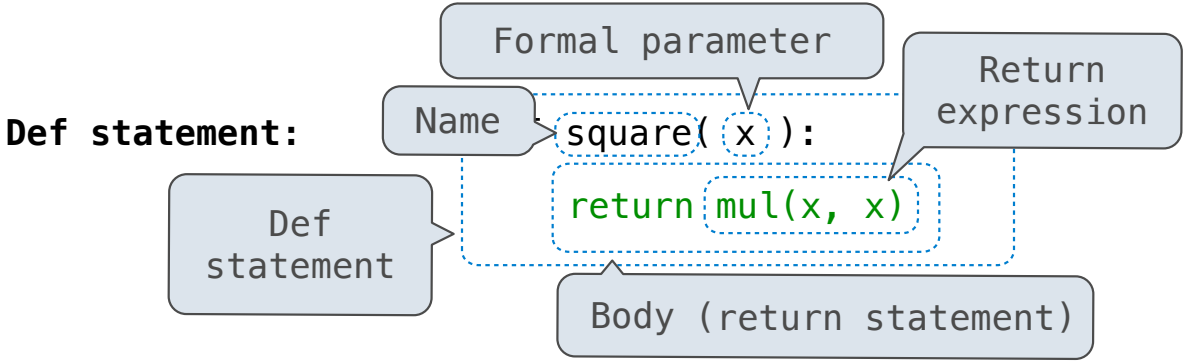
Calling/Applying:

What happens?

A new function is created!

Name bound to that function
in the current frame

Life Cycle of a User-Defined Function

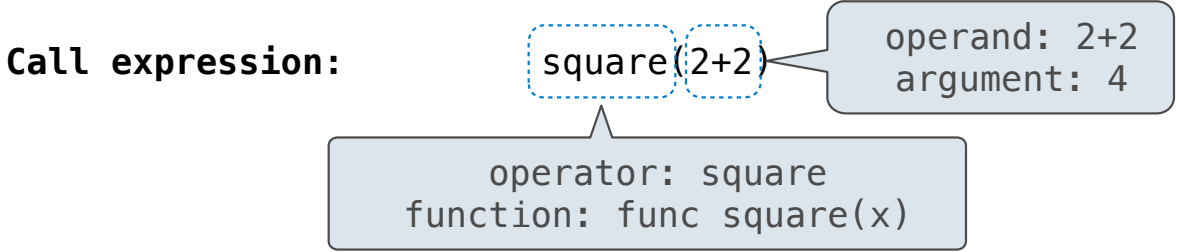
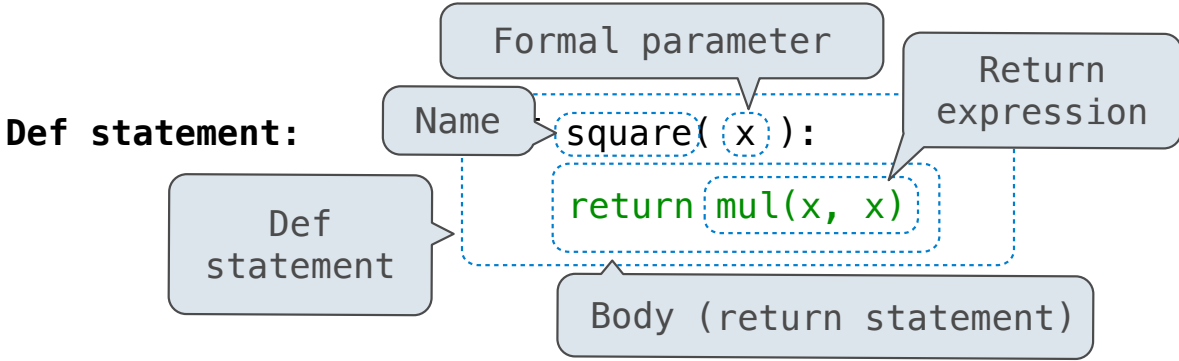


Calling/Applying:

What happens?

A new function is created!
Name bound to that function in the current frame

Life Cycle of a User-Defined Function

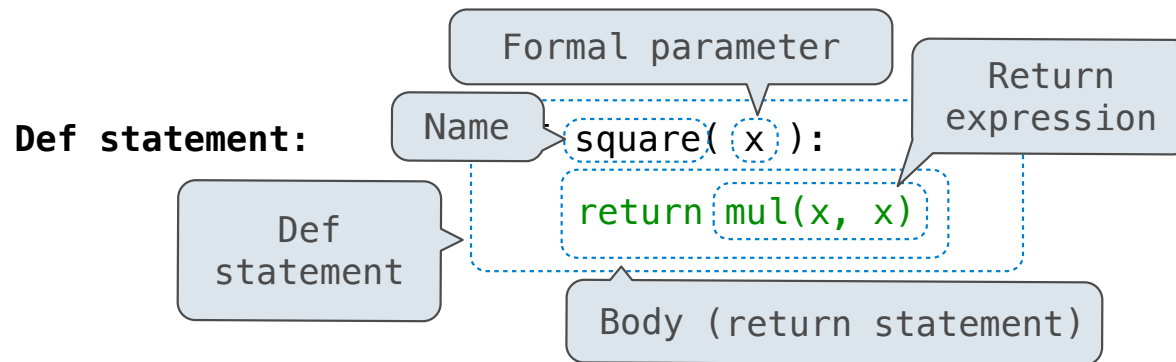


Calling/Applying:

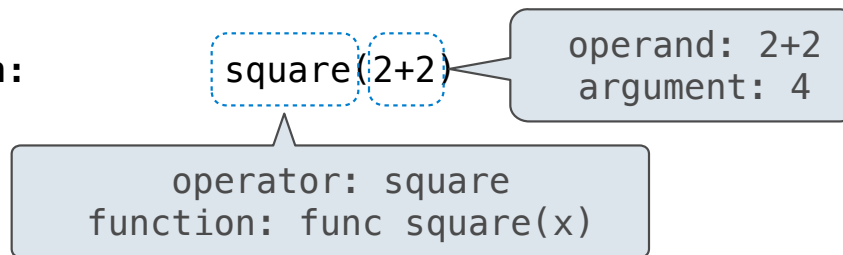
What happens?

- A new function is created!
- Name bound to that function in the current frame
- Operator & operands evaluated

Life Cycle of a User-Defined Function



Call expression:



Calling/Applying:

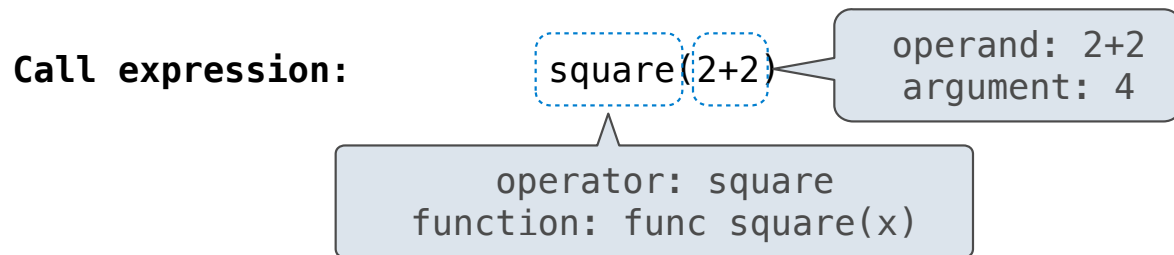
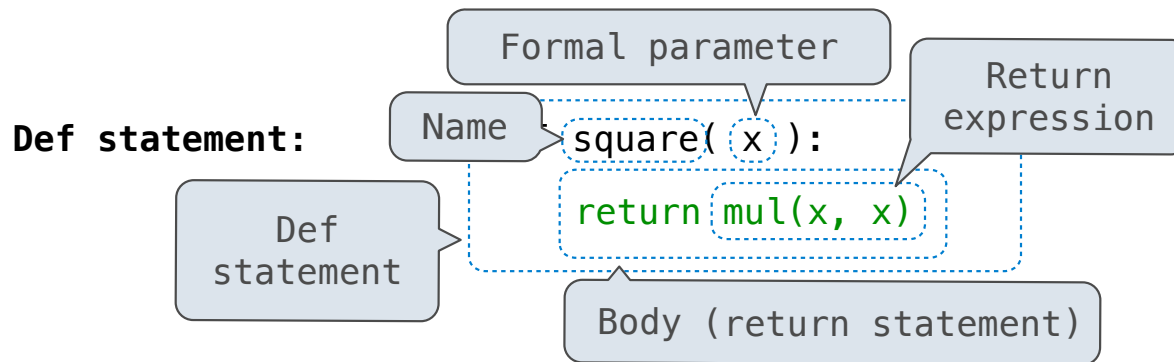
What happens?

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



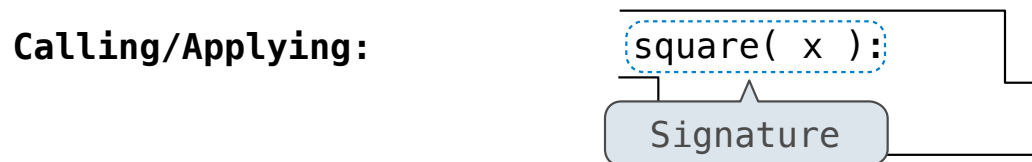
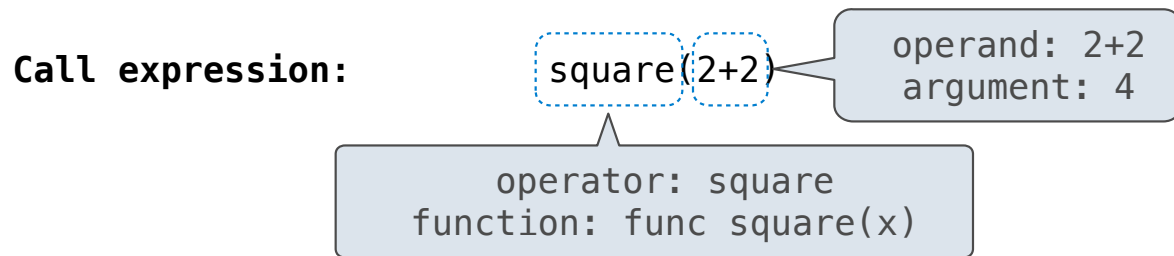
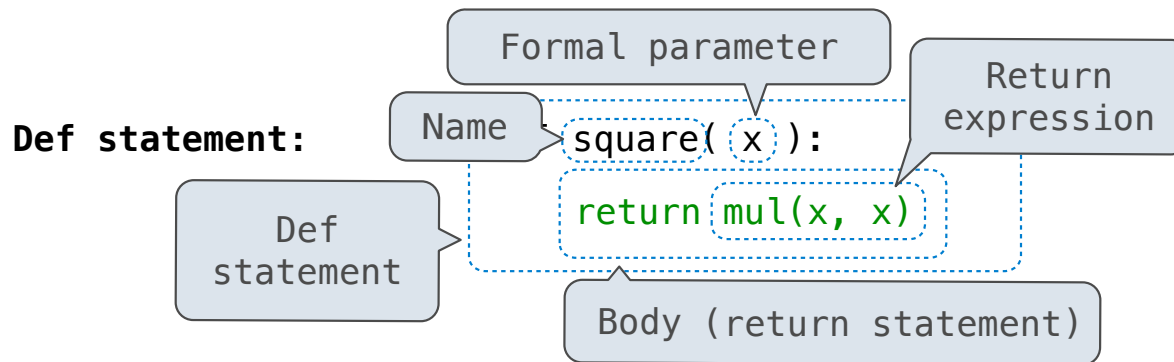
What happens?

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



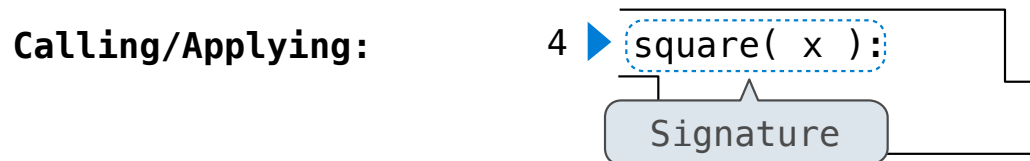
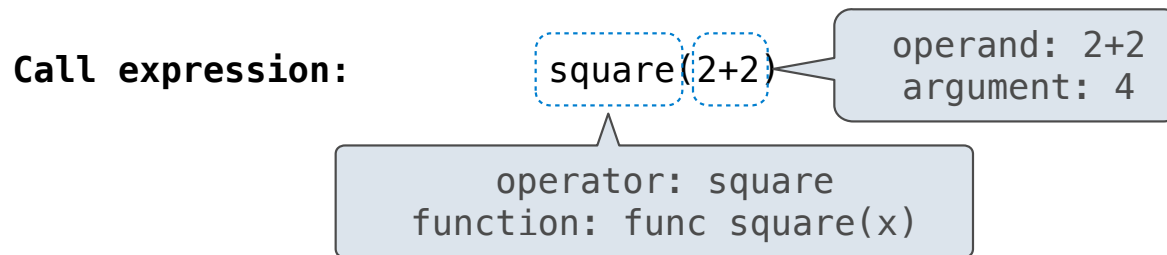
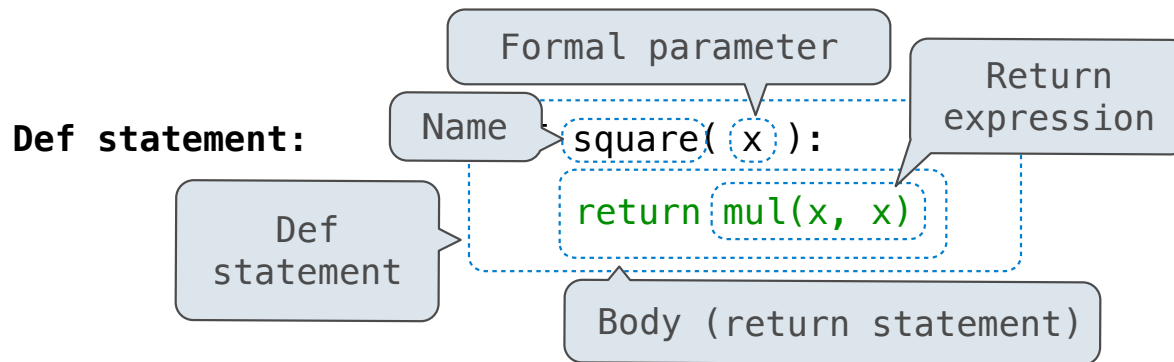
What happens?

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



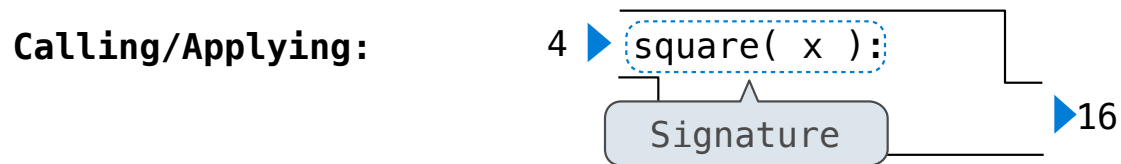
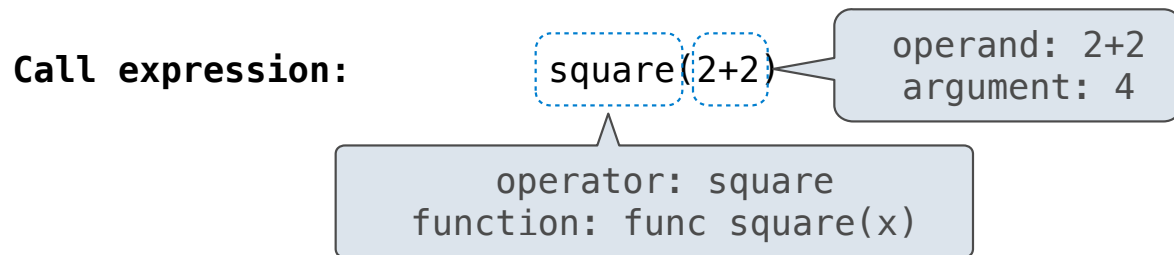
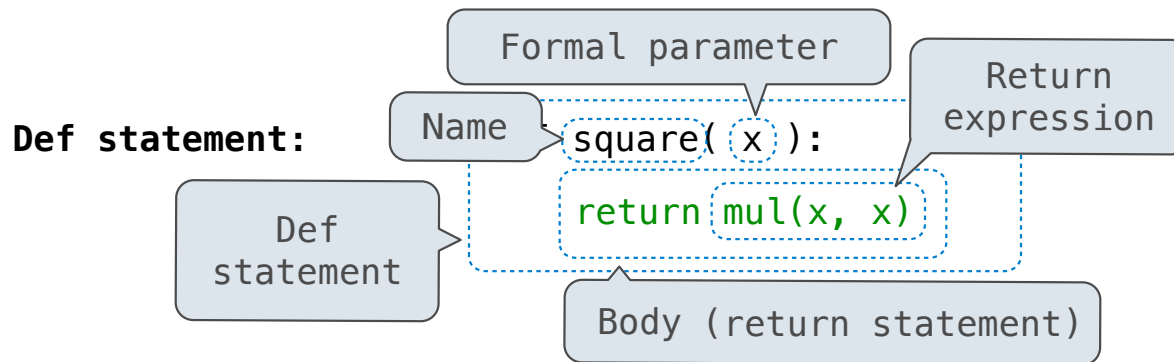
What happens?

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



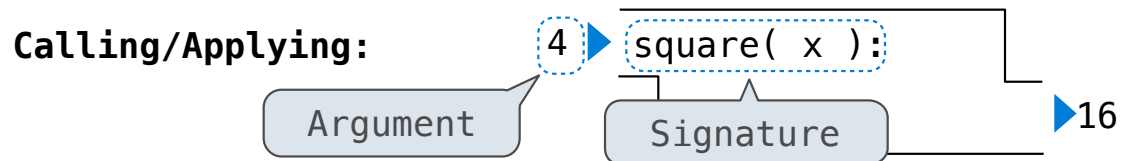
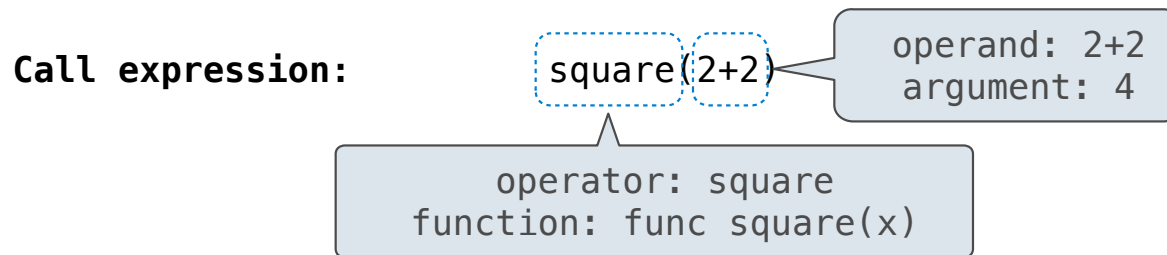
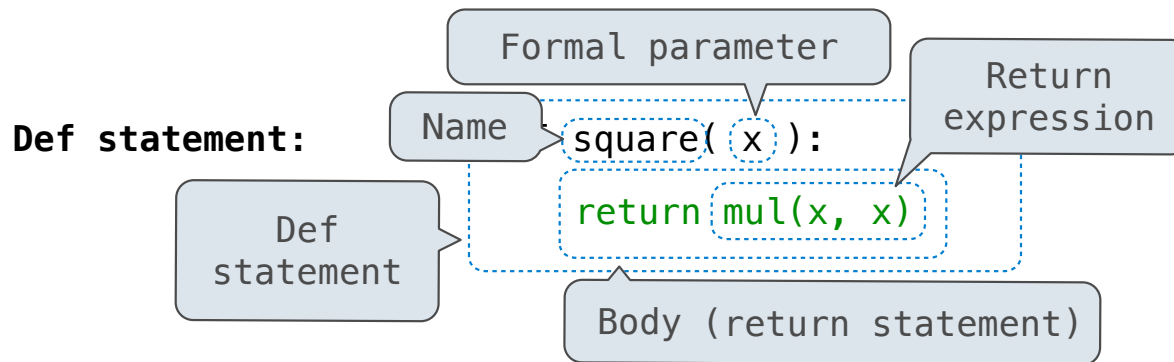
What happens?

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



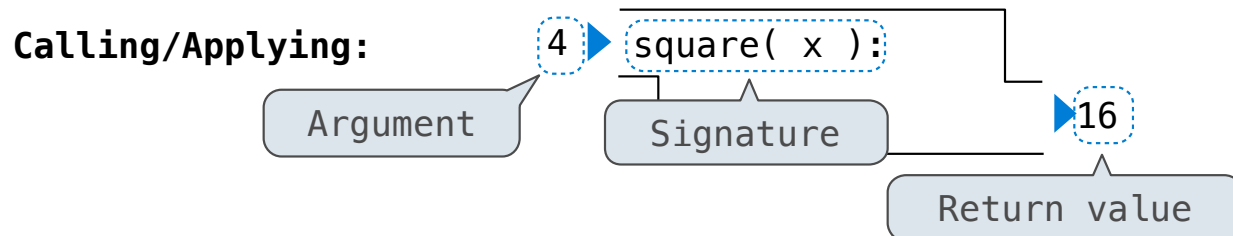
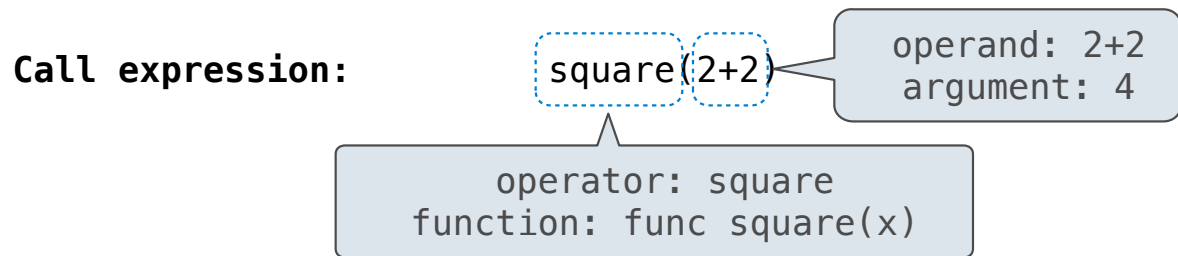
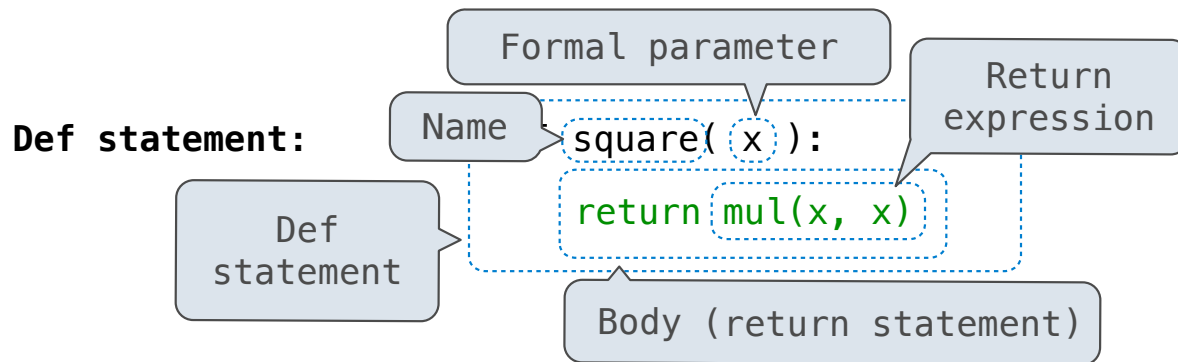
What happens?

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



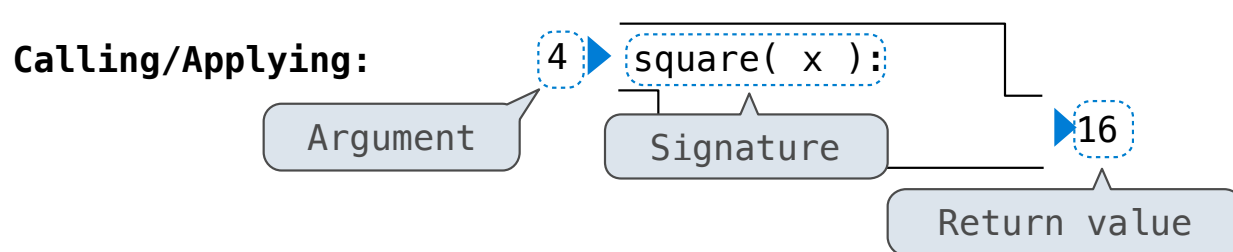
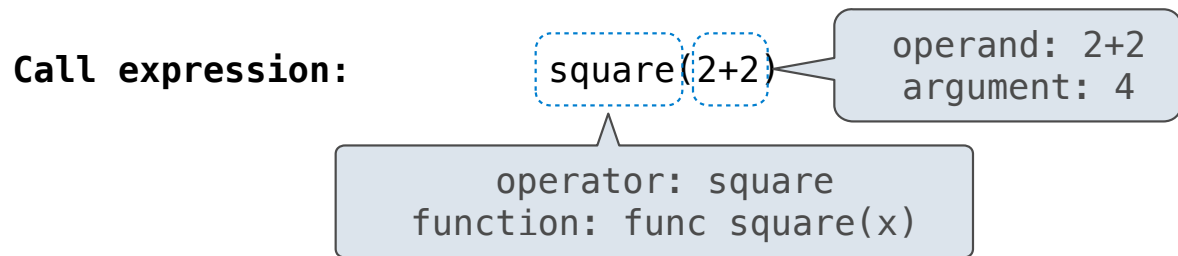
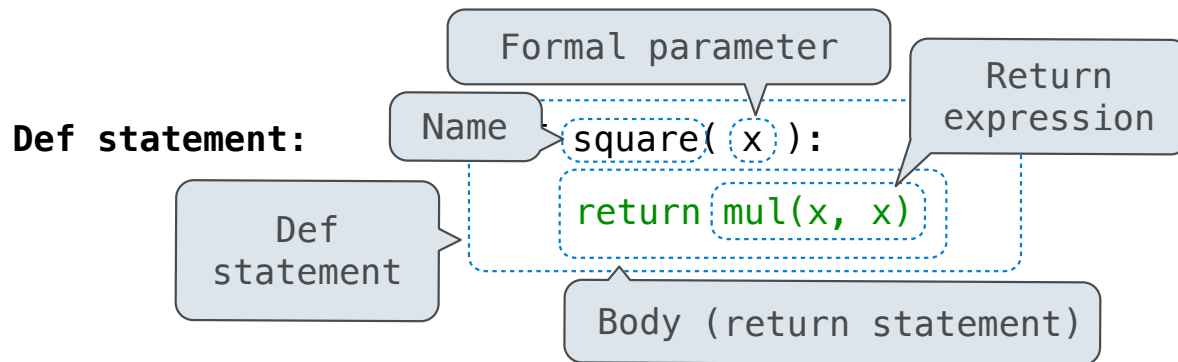
What happens?

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)

Life Cycle of a User-Defined Function



What happens?

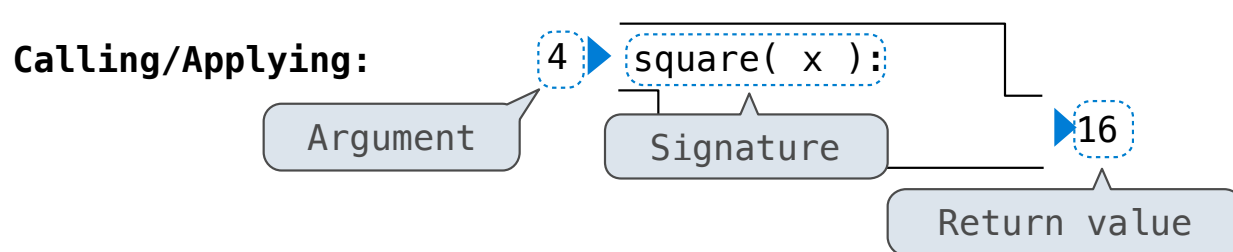
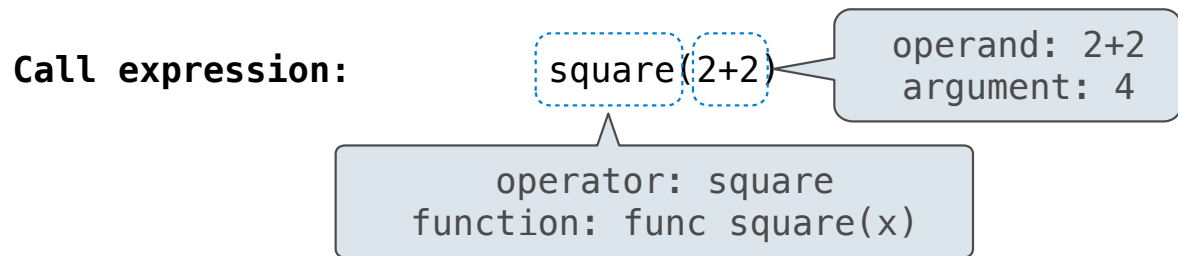
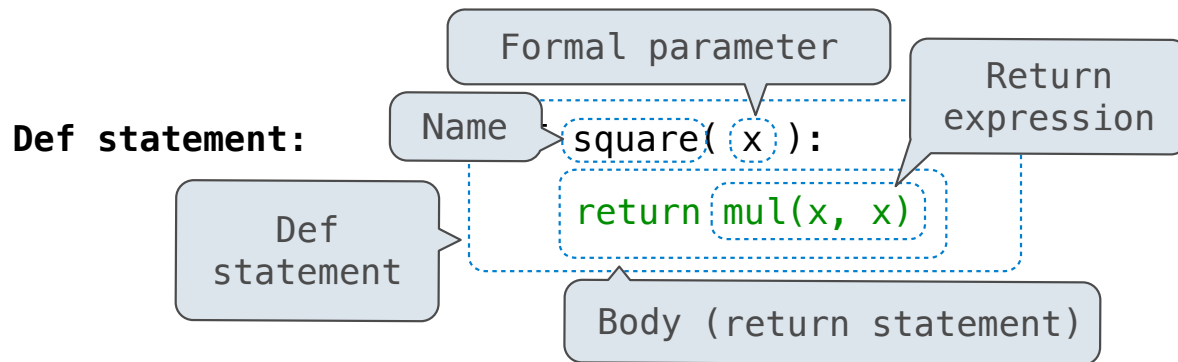
A new function is created!

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!

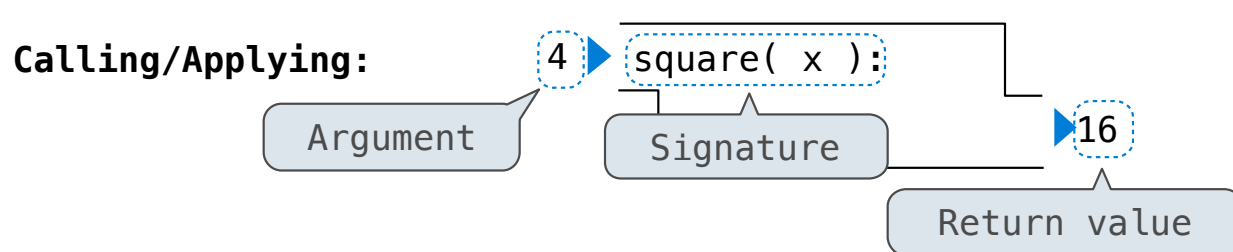
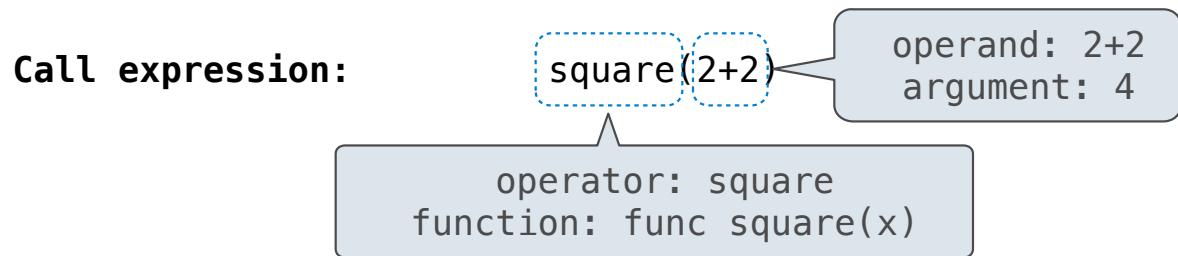
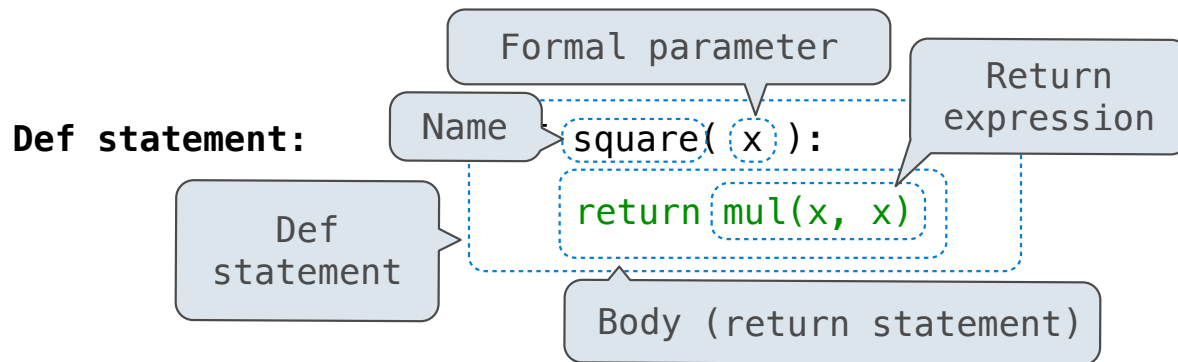
Name bound to that function
in the current frame

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

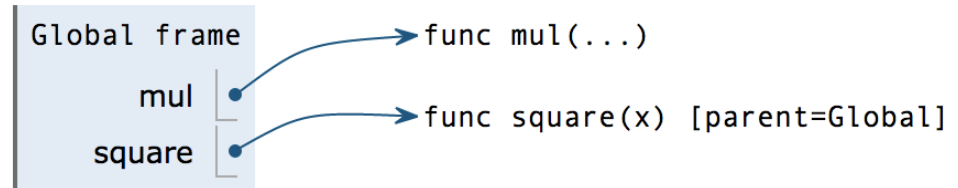
Body is executed in that new
environment

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!

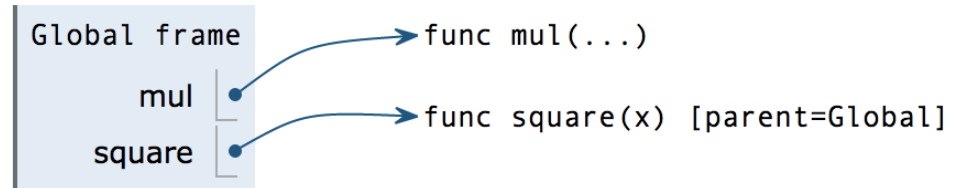
```
1 from operator import mul  
→ 2 def square(x):  
3     return mul(x, x)  
→ 4 square(square(3))
```



Interactive Diagram

Multiple Environments in One Diagram!

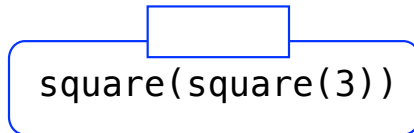
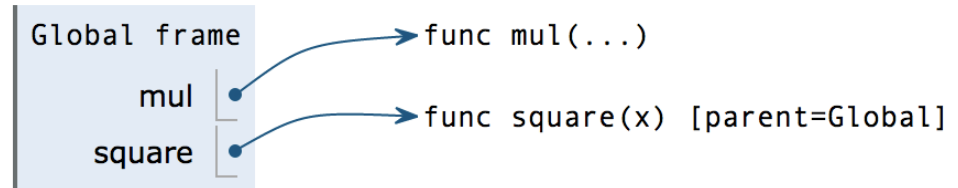
```
1 from operator import mul  
→ 2 def square(x):  
3     return mul(x, x)  
→ 4 square(square(3))
```



square(square(3))

Multiple Environments in One Diagram!

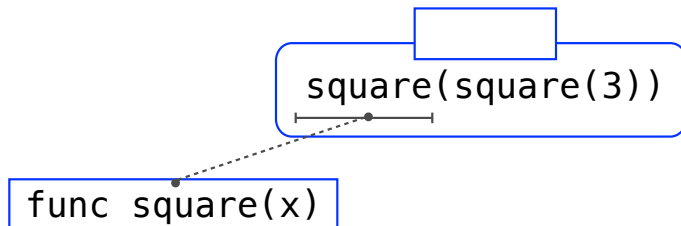
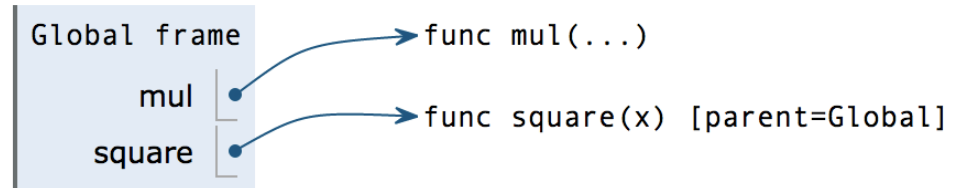
```
1 from operator import mul
→ 2 def square(x):
3     return mul(x, x)
→ 4 square(square(3))
```



Interactive Diagram

Multiple Environments in One Diagram!

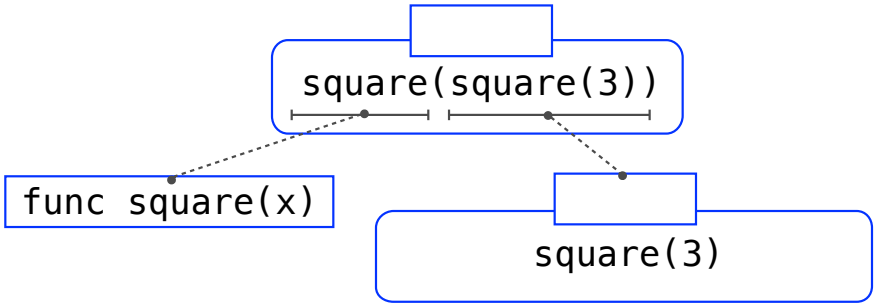
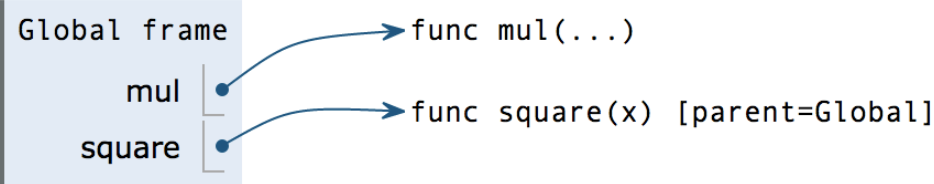
```
1 from operator import mul
2 def square(x):
3     return mul(x, x)
4 square(square(3))
```



Interactive Diagram

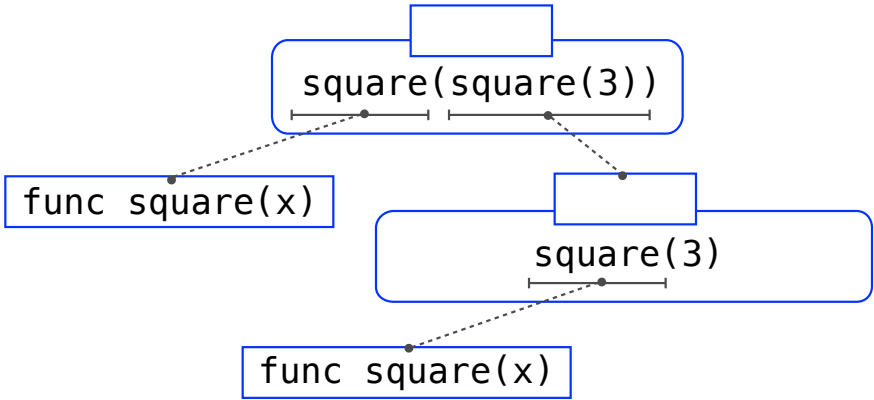
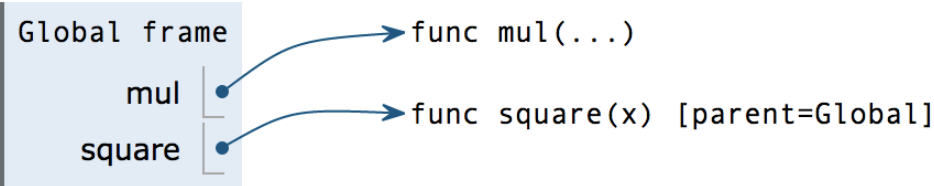
Multiple Environments in One Diagram!

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



Multiple Environments in One Diagram!

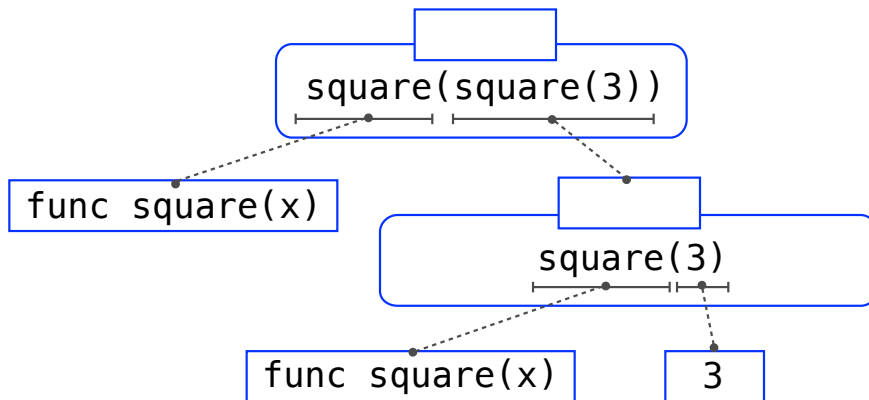
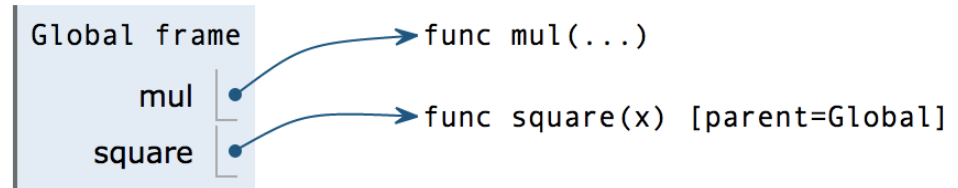
```
1 from operator import mul
2 def square(x):
3     return mul(x, x)
4 square(square(3))
```



Interactive Diagram

Multiple Environments in One Diagram!

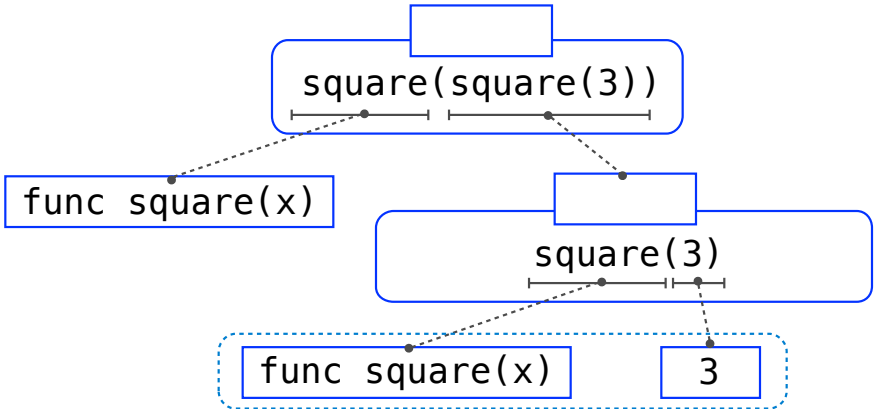
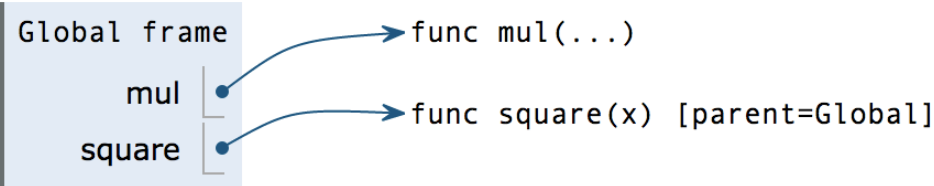
```
1 from operator import mul
2 def square(x):
3     return mul(x, x)
4 square(square(3))
```



Interactive Diagram

Multiple Environments in One Diagram!

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



Interactive Diagram

Multiple Environments in One Diagram!

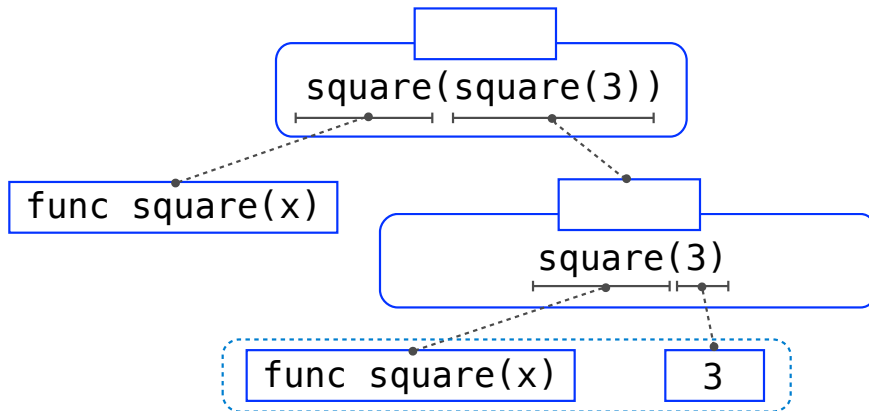
```
1 from operator import mul
2 def square(x):
3     return mul(x, x)
4 square(square(3))
```

Global frame

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

f1: square [parent=Global]

x | 3



Interactive Diagram

Multiple Environments in One Diagram!

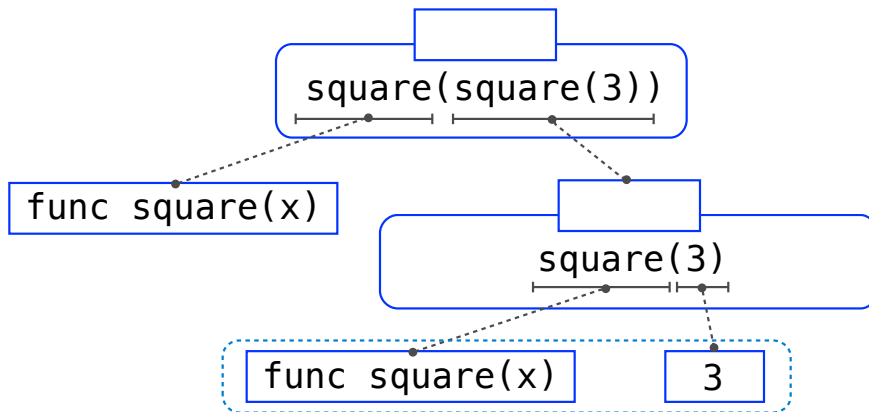
```
1 from operator import mul
2 def square(x):
3     return mul(x, x)
4 square(square(3))
```

Global frame

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

f1: square [parent=Global]

x	3
Return value	9

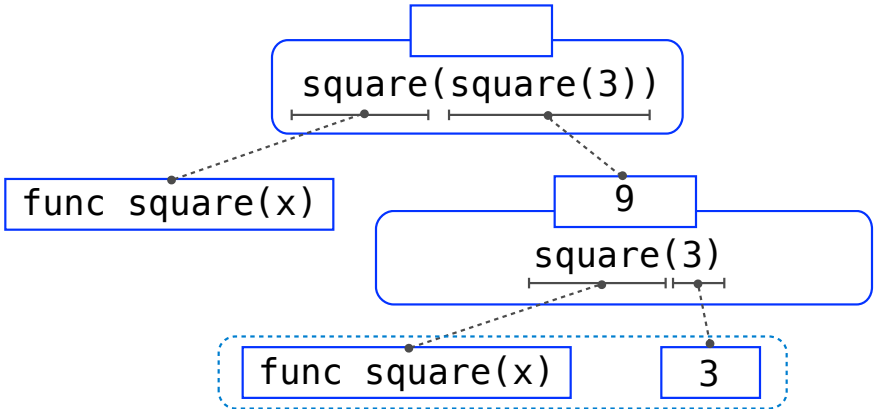
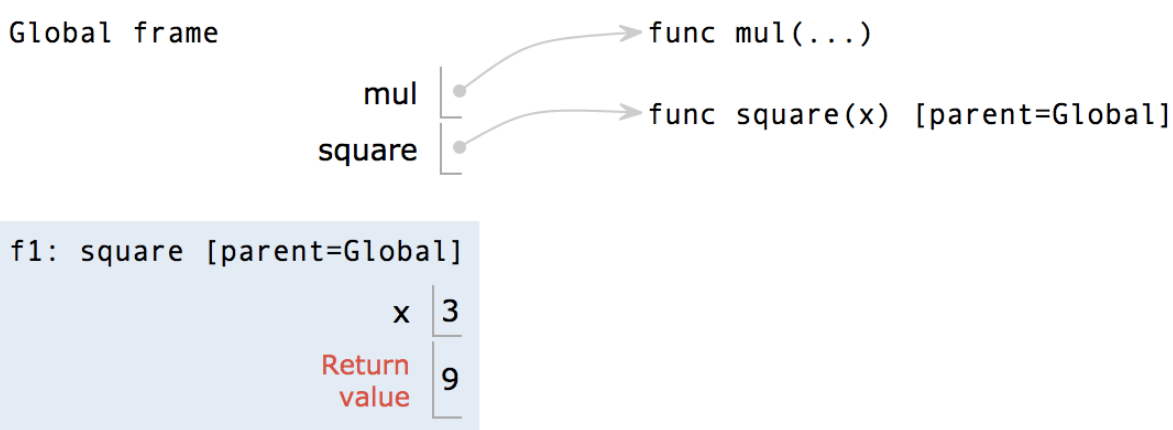


Interactive Diagram

Multiple Environments in One Diagram!

```

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

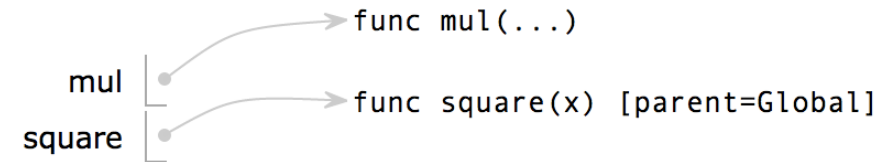


Interactive Diagram

Multiple Environments in One Diagram!

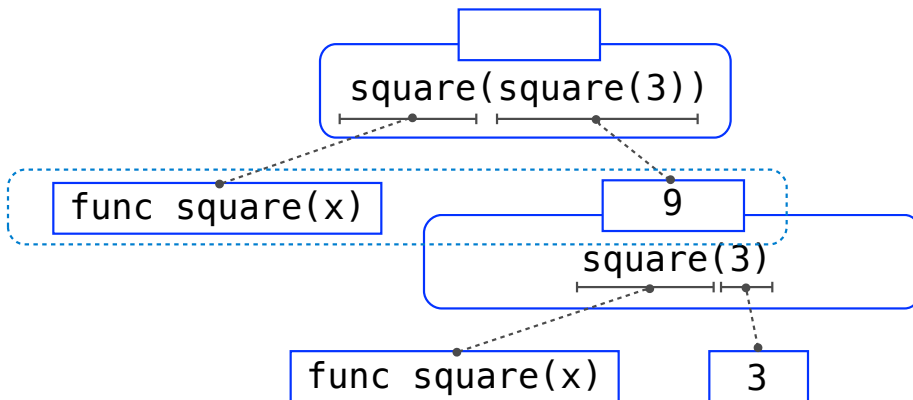
```
1 from operator import mul
2 def square(x):
3     return mul(x, x)
4 square(square(3))
```

Global frame



f1: square [parent=Global]

x | 3
Return value | 9

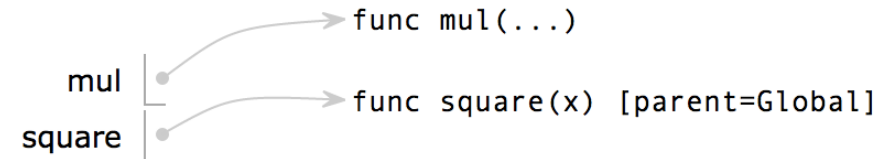


Interactive Diagram

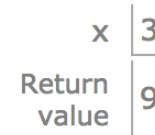
Multiple Environments in One Diagram!

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

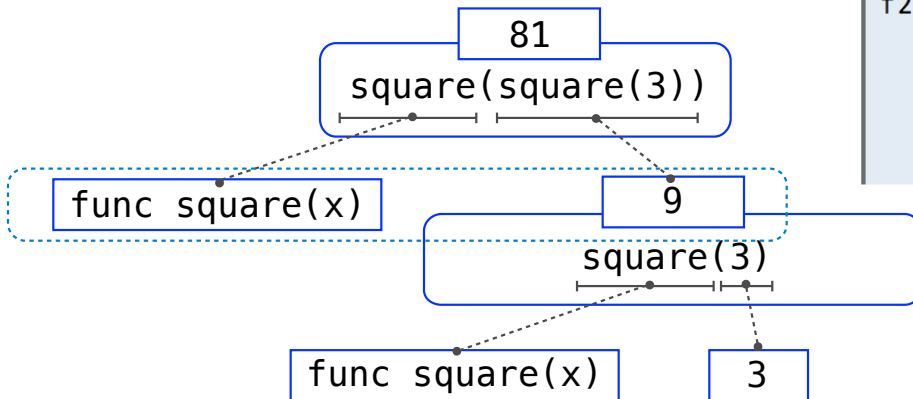
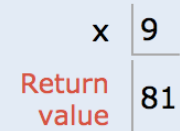
Global frame



f1: square [parent=Global]



f2: square [parent=Global]

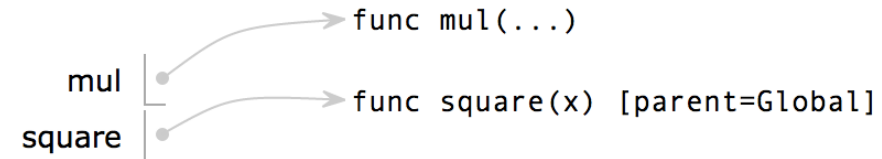


Interactive Diagram

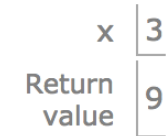
Multiple Environments in One Diagram!

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

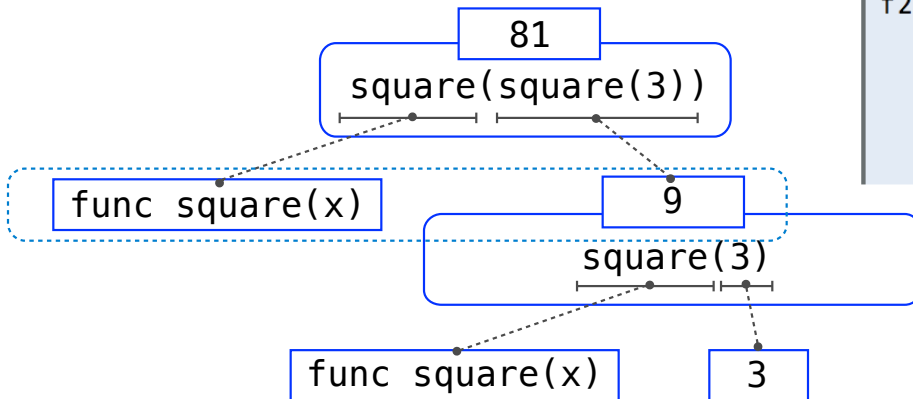
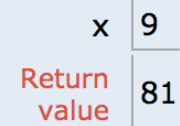
Global frame



f1: square [parent=Global]



f2: square [parent=Global]



An environment is a sequence of frames.

Interactive Diagram

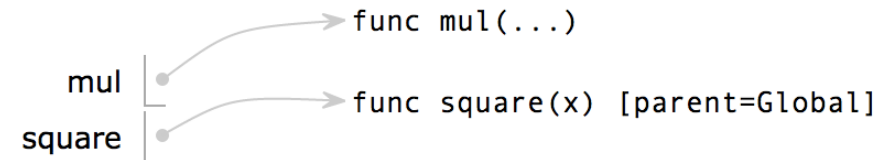
Multiple Environments in One Diagram!

```

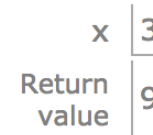
1 from operator import mul
2 def square(x):
3     return mul(x, x)
4 square(square(3))

```

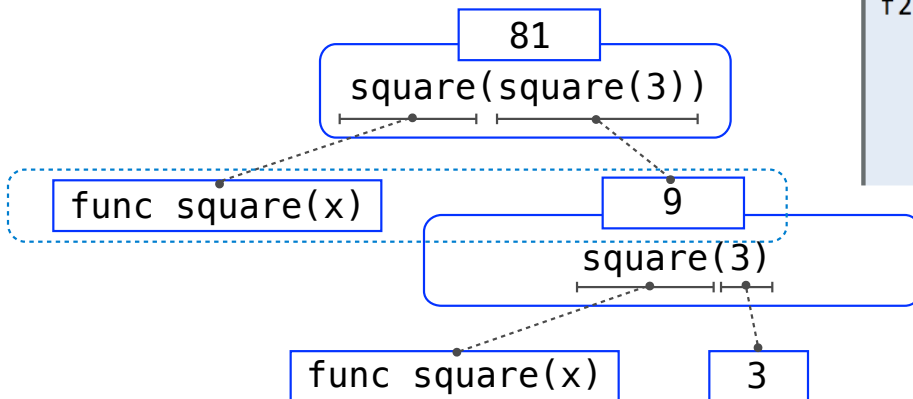
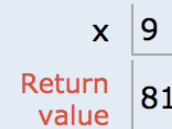
Global frame



f1: square [parent=Global]



f2: square [parent=Global]



An environment is a sequence of frames.

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

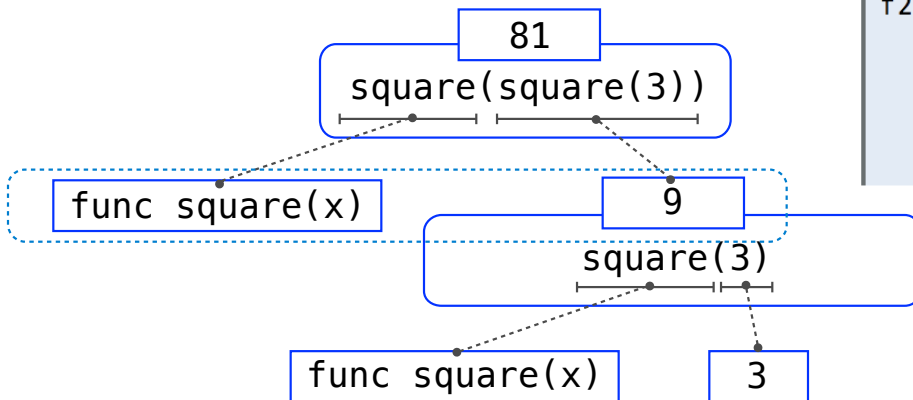
Interactive Diagram

Multiple Environments in One Diagram!

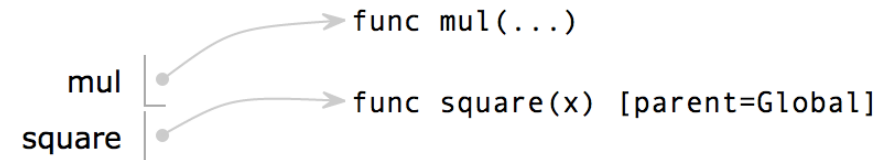
```

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

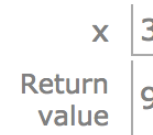
```



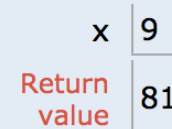
1 Global frame



f1: square [parent=Global]



f2: square [parent=Global]



An environment is a sequence of frames.

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

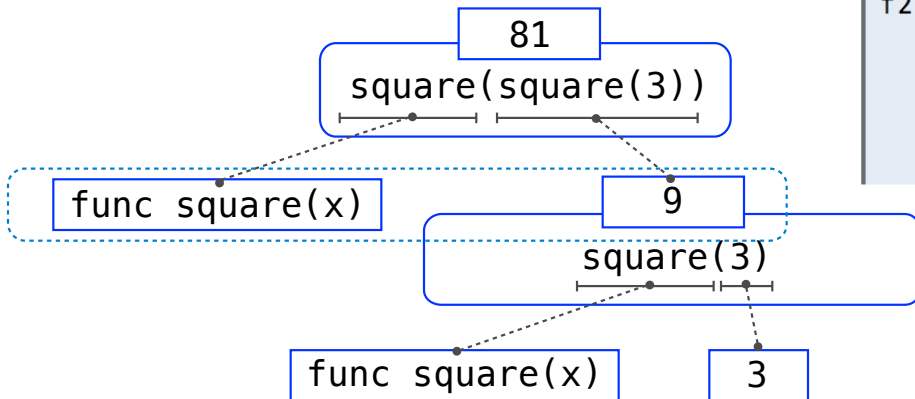
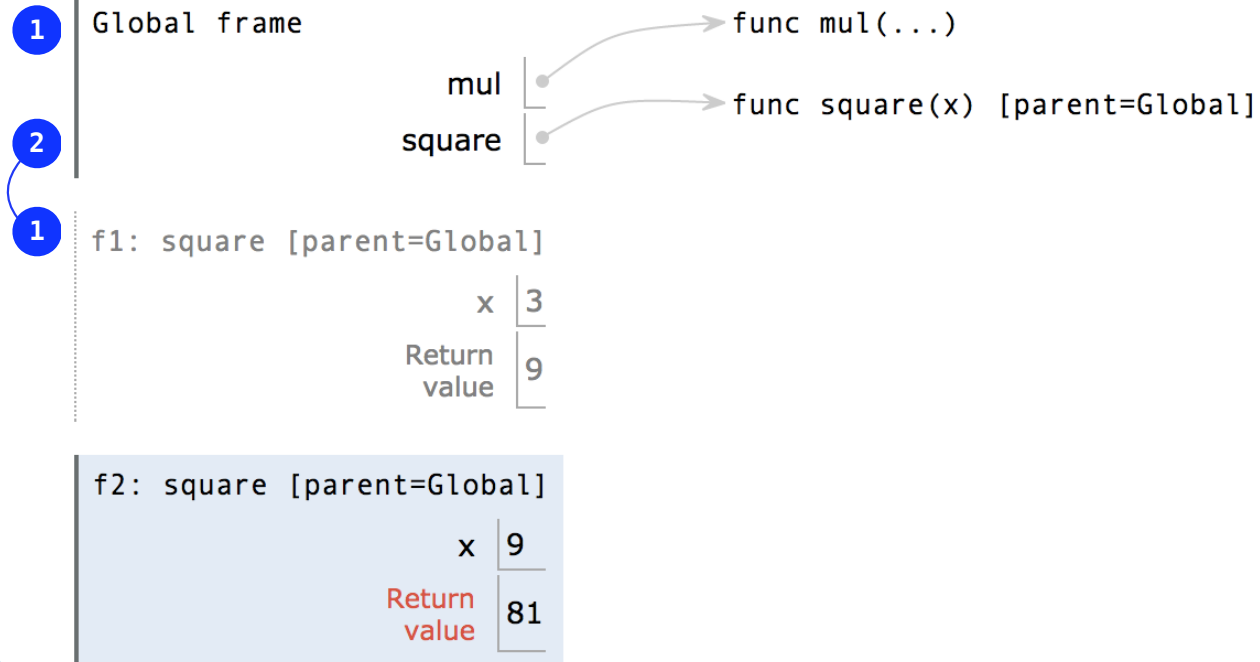
Interactive Diagram

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
- A local, then the global frame

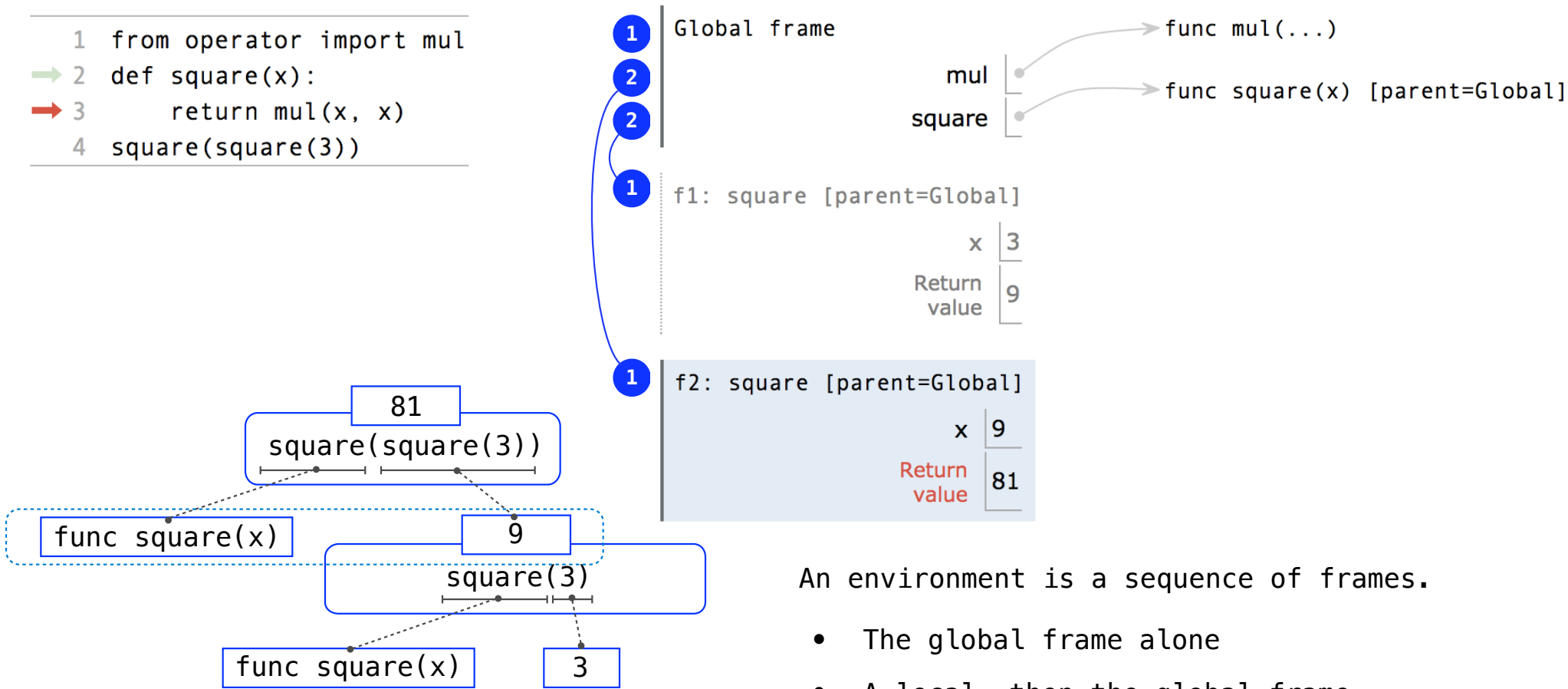
Interactive Diagram

Multiple Environments in One Diagram!

```

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

```



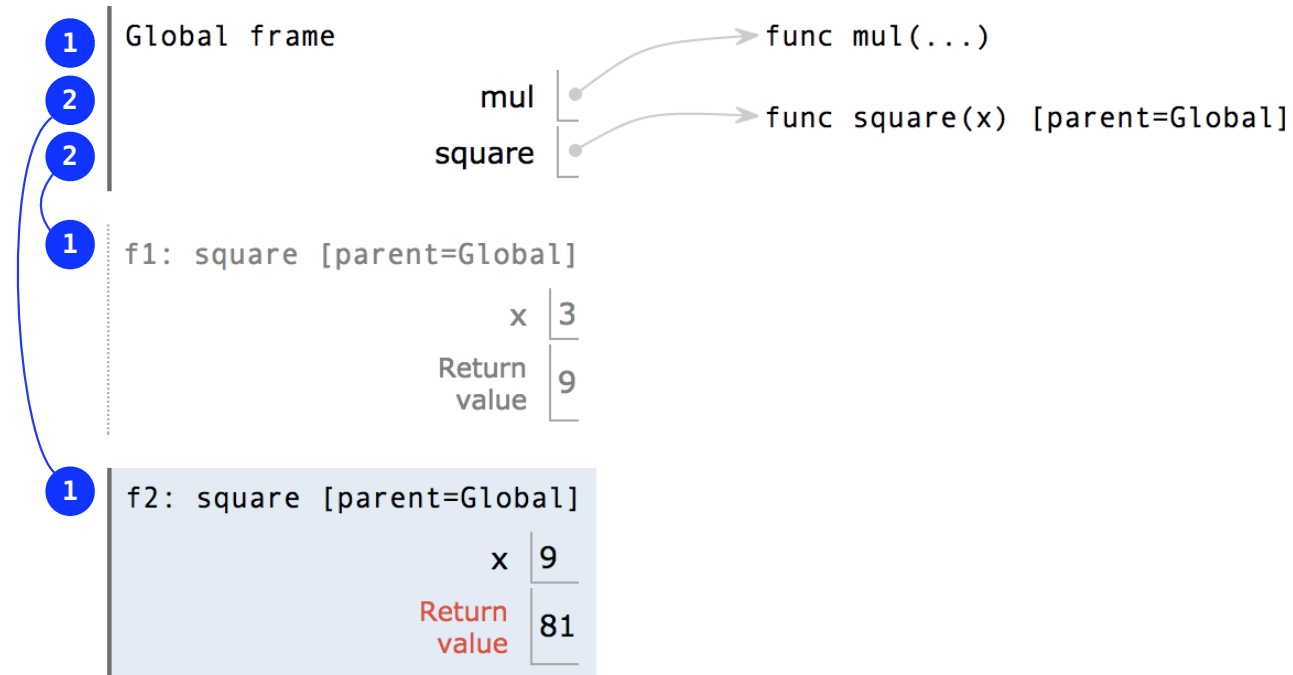
An environment is a sequence of frames.

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

Interactive Diagram

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.

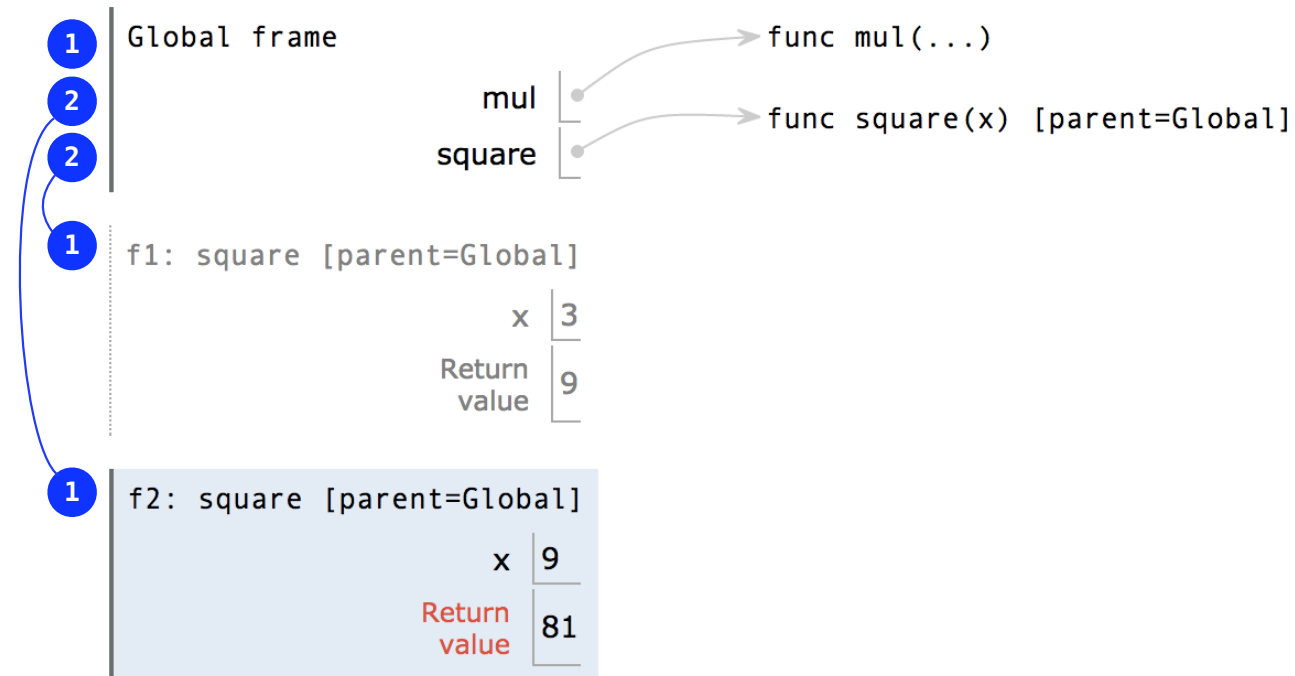
- The global frame alone
- A local, then the global frame

Interactive Diagram

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
- A local, then the global frame

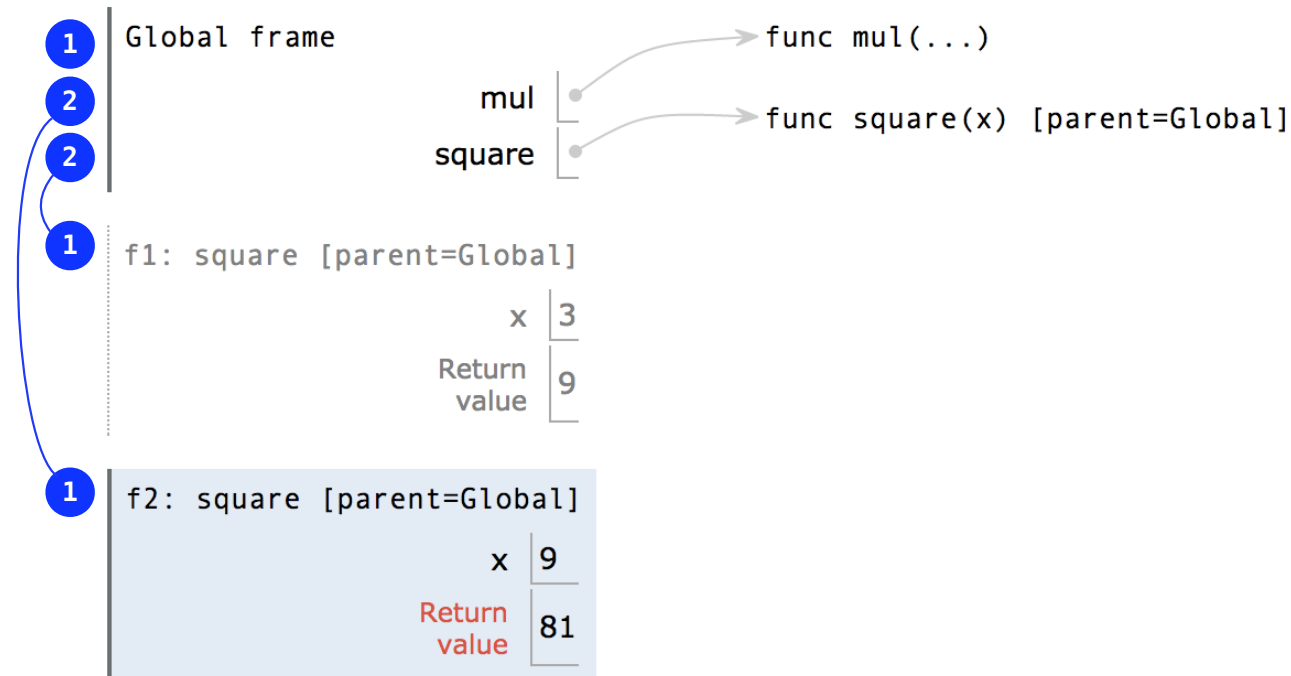
Interactive Diagram

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
- A local, then the global frame

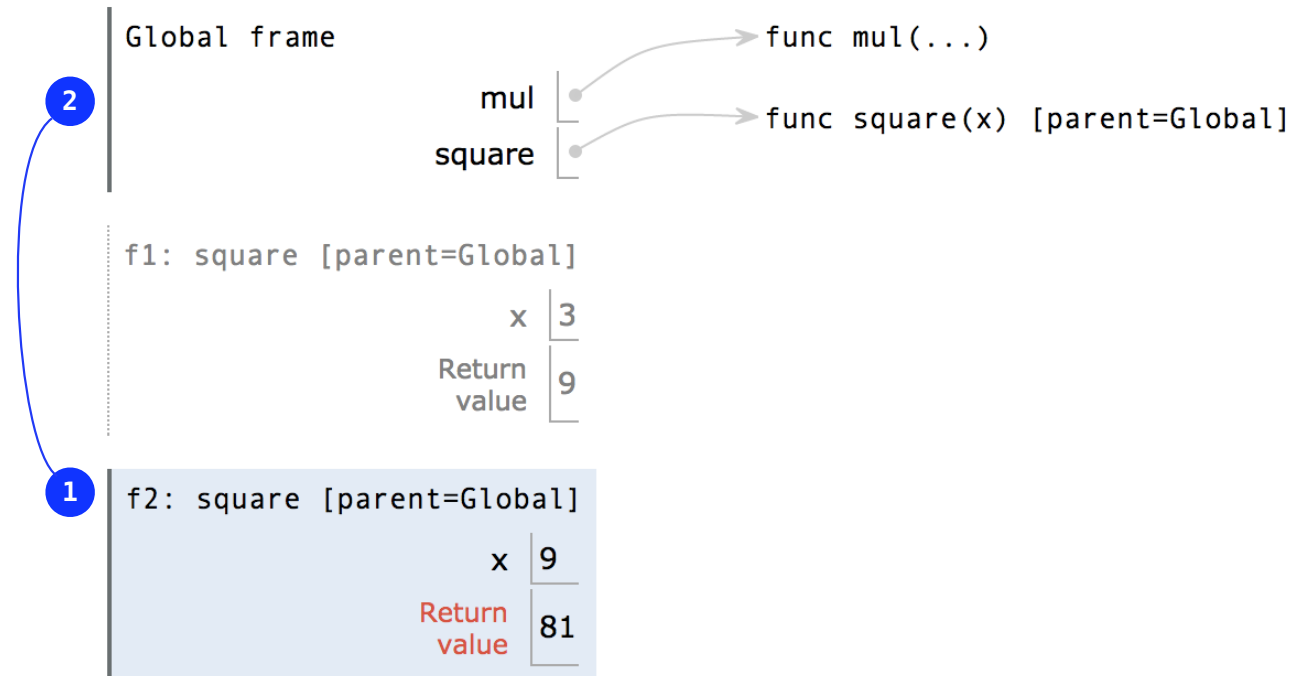
Interactive Diagram

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
- A local, then the global frame

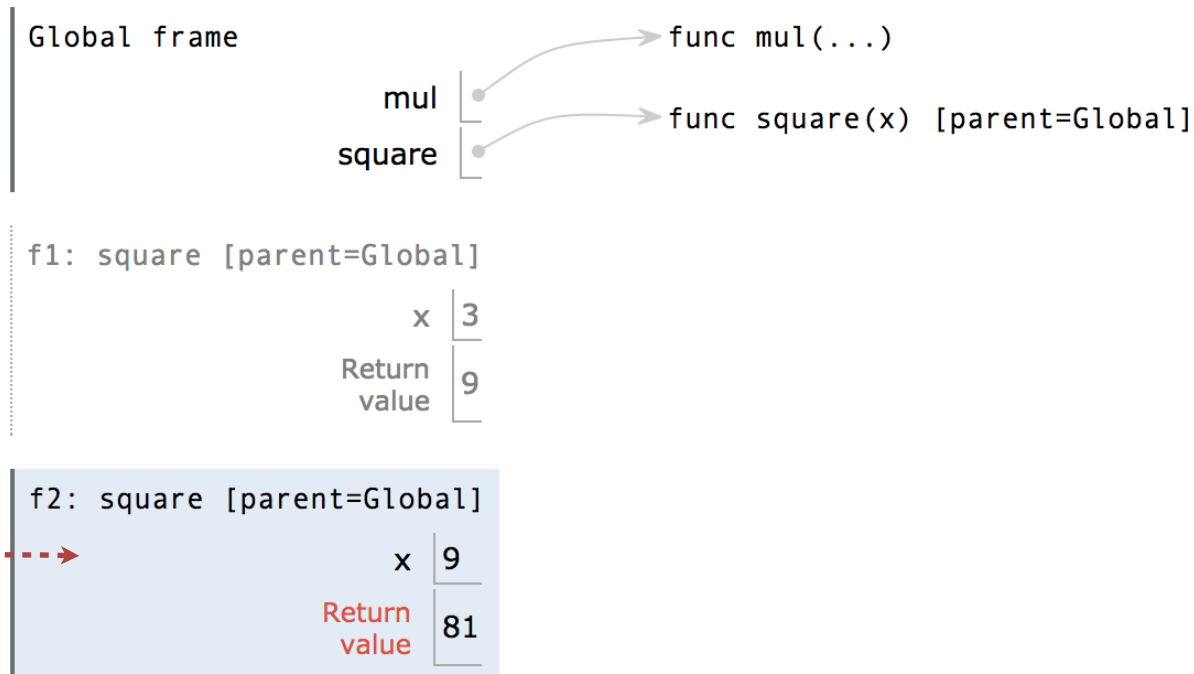
Interactive Diagram

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.

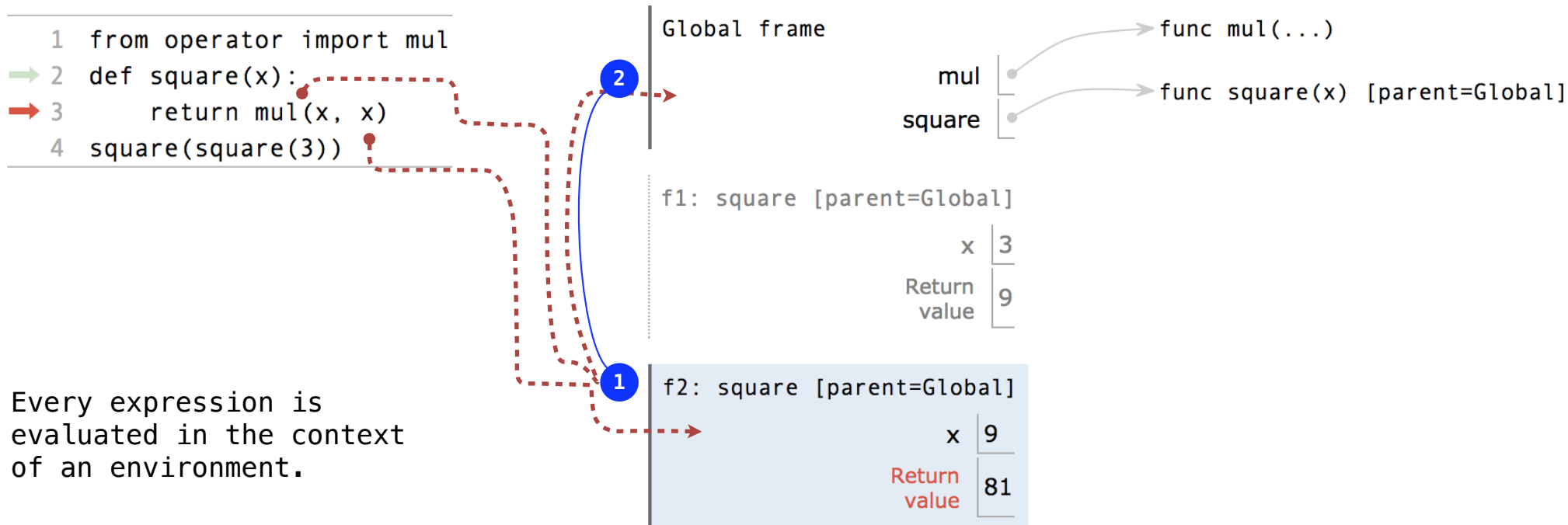


An environment is a sequence of frames.

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

Interactive Diagram

Names Have No Meaning Without Environments



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

Interactive Diagram

Names Have Different Meanings in Different Environments

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.

[Interactive Diagram](#)

Names Have Different Meanings in Different Environments

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

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.

[Interactive Diagram](#)

Names Have Different Meanings in Different Environments

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

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

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.

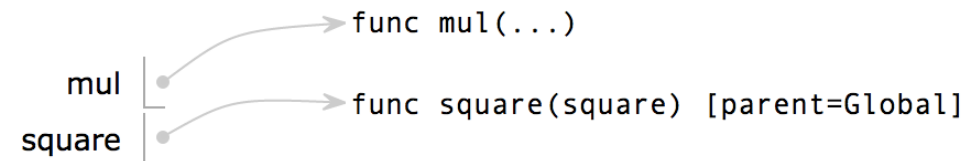
[Interactive Diagram](#)

Names Have Different Meanings in Different Environments

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

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

Global frame



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.

[Interactive Diagram](#)

Names Have Different Meanings in Different Environments

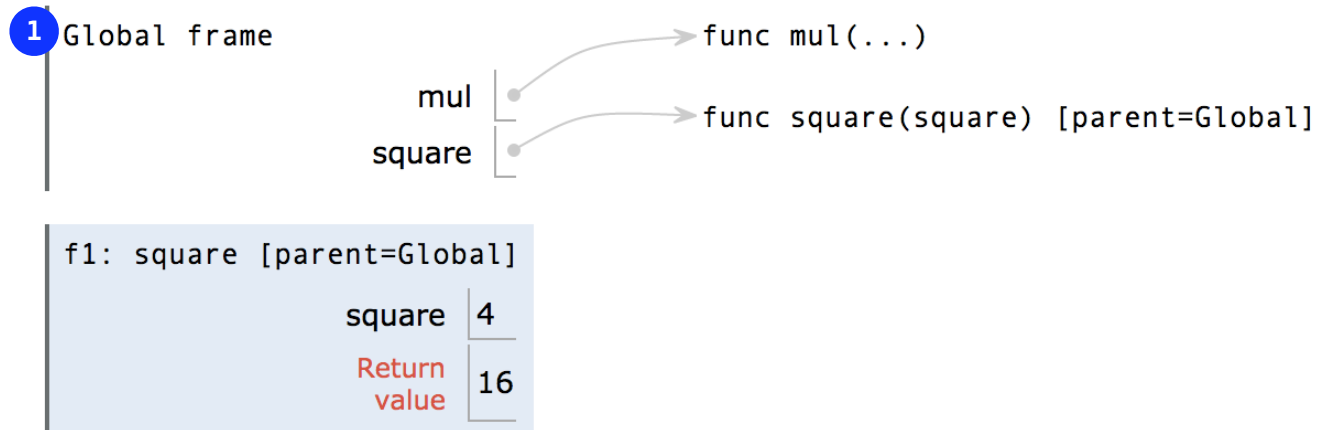
A call expression and the body of the function being called are evaluated in different environments

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



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.

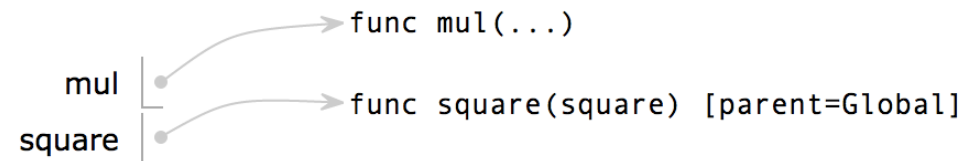
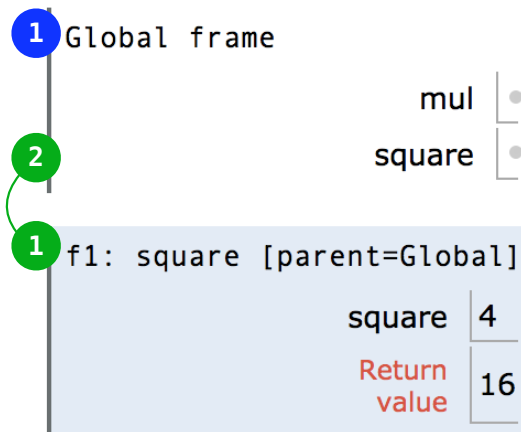
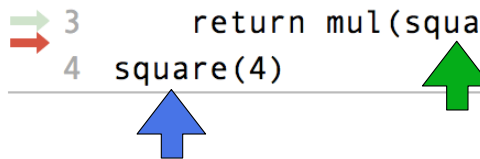


Interactive Diagram

Names Have Different Meanings in Different Environments

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

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



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.

Interactive Diagram

Miscellaneous Python Features

Division
Multiple Return Values
Source Files
Doctests
Default Arguments

(Demo)

Conditional Statements

Statements

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

Statements

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

Compound statements:

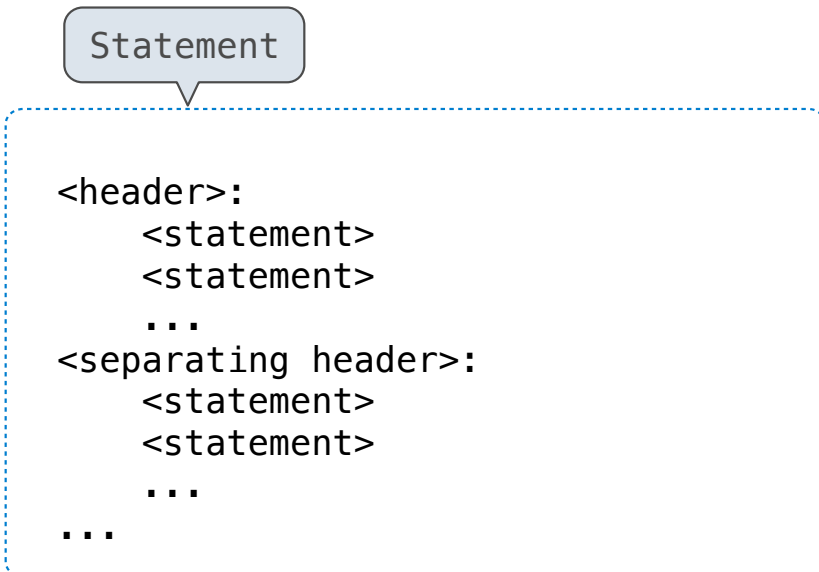
```
<header>:  
    <statement>  
    <statement>  
    ...  
<separating header>:  
    <statement>  
    <statement>  
    ...  
...
```

Statements

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

Compound statements:

Statement

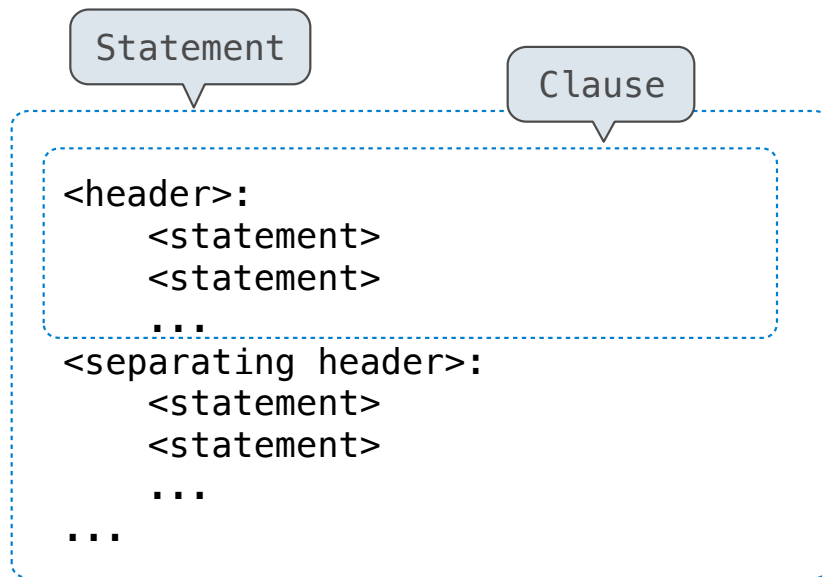


```
<header>:  
  <statement>  
  <statement>  
  ...  
<separating header>:  
  <statement>  
  <statement>  
  ...  
...
```

Statements

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

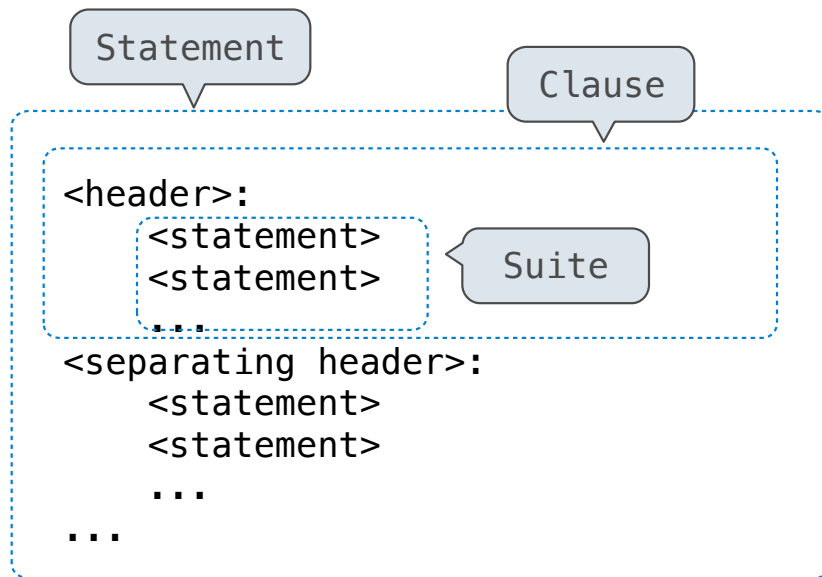
Compound statements:



Statements

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

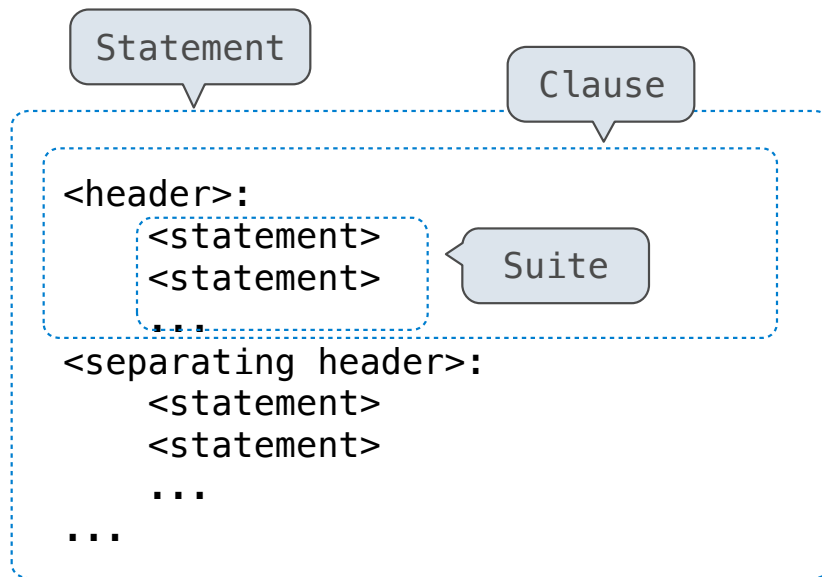
Compound statements:



Statements

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

Compound statements:

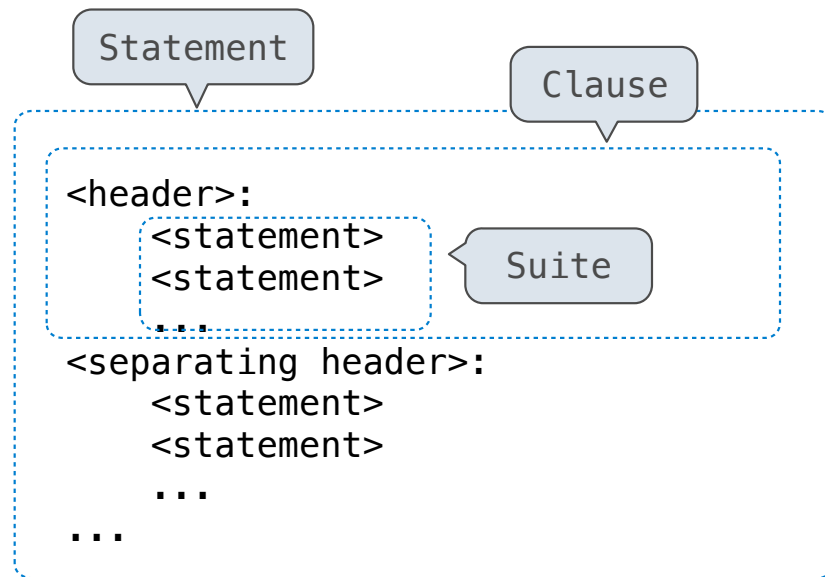


The first header determines a statement's type

Statements

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

Compound statements:



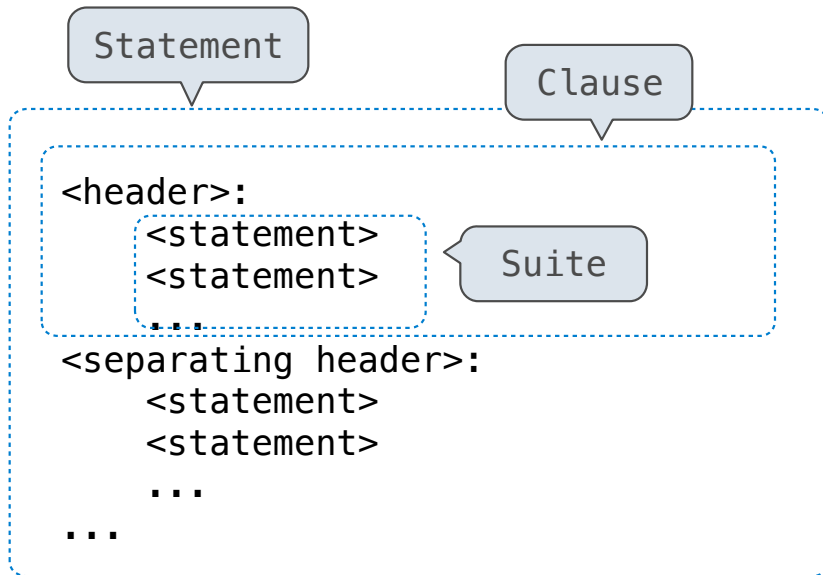
The first header determines a statement's type

The header of a clause "controls" the suite that follows

Statements

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

Compound statements:



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>  
  ...  
...
```




The diagram illustrates the structure of compound statements. A dashed blue box encloses the first block of code (under the header), and a light blue callout box labeled "Suite" points to it. The code consists of a header, a block of statements, a separating header, another block of statements, and an ellipsis.

Compound Statements

Compound statements:

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




The diagram illustrates the structure of compound statements. A dashed blue box highlights a sequence of statements (two <statement> tags followed by an ellipsis) within the <header> section. A light blue callout box labeled 'Suite' points to this sequence, indicating that a suite is a sequence of statements.

A suite is a sequence of statements

Compound Statements

Compound statements:

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



The diagram illustrates a compound statement structure. A dashed blue box encloses the first three lines of the code: the header label, two statements, and an ellipsis. A callout box labeled "Suite" points to this enclosed section, indicating that the sequence of statements within a header is referred to as a suite.


A suite is a sequence of statements

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

Compound Statements

Compound statements:

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



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

(Demo)

Conditional Statements

(Demo)

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

Conditional Statements

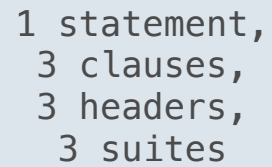
(Demo)

1 statement,
3 clauses,
3 headers,
3 suites

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

Conditional Statements

(Demo)



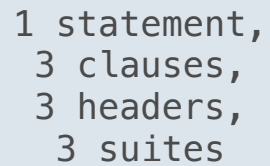
1 statement,
3 clauses,
3 headers,
3 suites

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

Execution Rule for Conditional Statements:

Conditional Statements

(Demo)



1 statement,
3 clauses,
3 headers,
3 suites

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

Execution Rule for Conditional Statements:

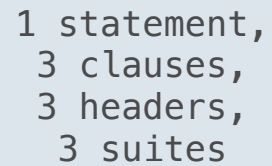
Each clause is considered in order.

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

Conditional Statements

(Demo)

```
def absolute_value(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.

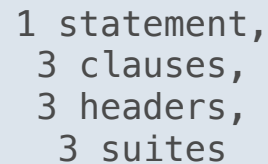
1. Evaluate the header's expression.
2. If it is a true value, execute the suite & skip the remaining clauses.

Syntax Tips:

Conditional Statements

(Demo)

```
def absolute_value(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.

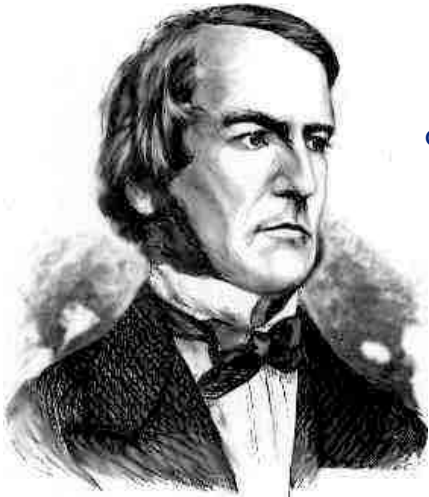
Boolean Contexts



George Boole

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

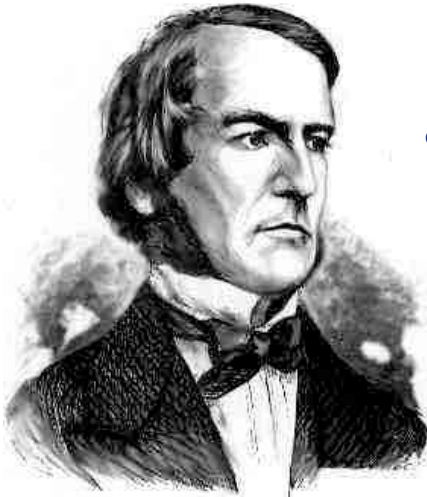

Boolean Contexts



George Boole

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

Boolean Contexts

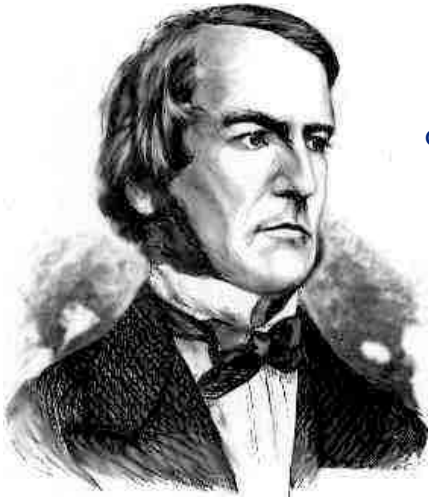


George Boole

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

Two boolean contexts

Boolean Contexts



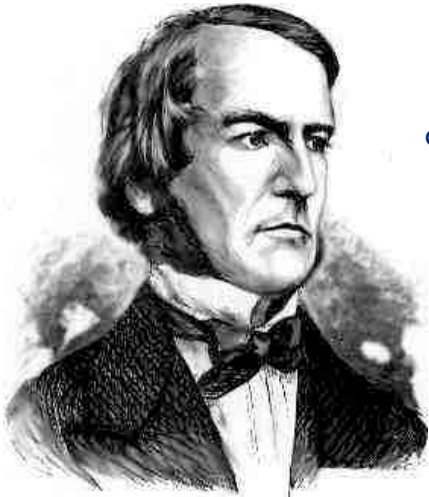
George Boole

```
def absolute_value(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

Boolean Contexts



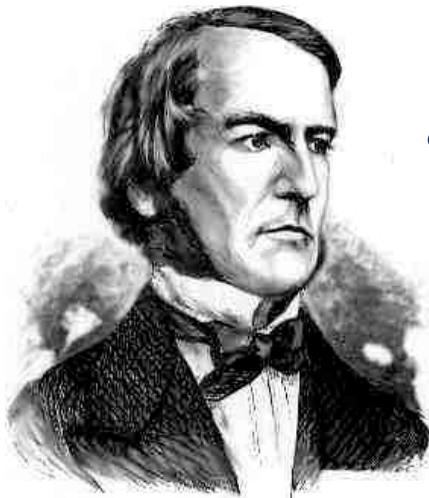
George Boole

```
def absolute_value(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)*

Boolean Contexts



George Boole

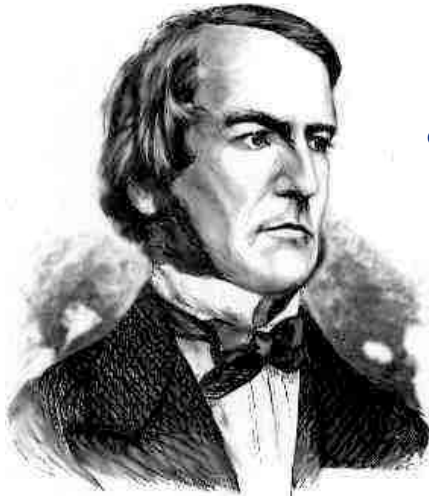
```
def absolute_value(x):  
    """Return the absolute value of x."""  
    if x < 0:  
        return -x  
    elif x == 0:  
        return 0  
    else:  
        return x
```

Two boolean contexts

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

True values in Python: Anything else (True)

Boolean Contexts



George Boole

```
def absolute_value(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!

Iteration

While Statements

(Demo)

While Statements

(Demo)

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

While Statements

(Demo)

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

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.

While Statements

(Demo)



George Boole

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

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.

While Statements

(Demo)



George Boole

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

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.

While Statements

(Demo)



George Boole

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

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.

While Statements

(Demo)



George Boole

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

Global frame

i	0
total	0

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.

While Statements

(Demo)



George Boole

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

Global frame	
i	0
total	0

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.

While Statements

(Demo)



George Boole

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

Global frame

i	0
total	0

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.

While Statements

(Demo)



George Boole

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

Global frame

i	0	1
total		0

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.

While Statements

(Demo)



George Boole

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

Global frame

i	0	1
total		0

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.

While Statements

(Demo)



George Boole

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

Global frame		
i	0	1
total	0	1

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.

While Statements

(Demo)



George Boole

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

Global frame

i	0	1
total	0	1

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.

While Statements

(Demo)



George Boole

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

```
Global frame
  i 0 1
  total 0 1
```

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.

While Statements

(Demo)



George Boole

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

Global frame

i	0	2
total	0	1

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.

While Statements

(Demo)



George Boole

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

```
Global frame
i 0 2
total 0 1
```

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.

While Statements

(Demo)



George Boole

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

```
Global frame
i 0 1 2
total 0 1 3
```

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.

While Statements

(Demo)



George Boole

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

```
Global frame
i 0 1 2
total 0 1 3
```

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.

While Statements

(Demo)



George Boole

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

```
Global frame
i 0 1 2
total 0 1 3
```

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.

While Statements

(Demo)



George Boole

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

Global frame

i	0	1	2	3
total	0	1	3	

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.

While Statements

(Demo)



George Boole

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

Global frame			
i	0	1	2 3
total	0	1	3

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.

While Statements

(Demo)



George Boole

```
1 i, total = 0, 0
2 while i < 3:
3     i = i + 1
▶ 4     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.

While Statements

(Demo)



George Boole

```
1 i, total = 0, 0
▶ 2 while i < 3:
3     i = i + 1
4     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.