61A Lecture 3

Announcements

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
...
```

No **return**

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 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)
```

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)
```

No **return**

None value is not displayed

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

No **return**

None value is not displayed

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

Pure Functions & Non-Pure Functions

Pure Functions
just return values

Non-Pure Functions
have side effects

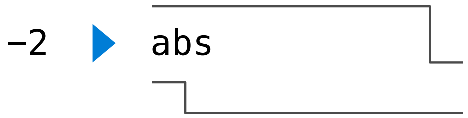
Pure Functions & Non-Pure Functions

Pure Functions
just return values

Non-Pure Functions
have side effects

Pure Functions & Non-Pure Functions

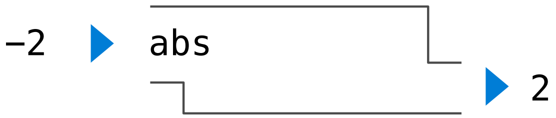
Pure Functions
just return values



Non-Pure Functions
have side effects

Pure Functions & Non-Pure Functions

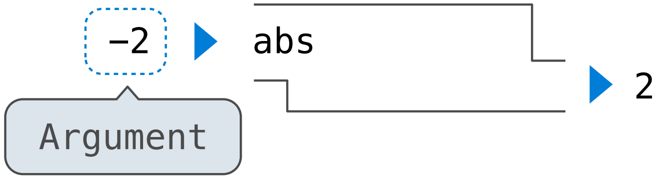
Pure Functions
just return values



Non-Pure Functions
have side effects

Pure Functions & Non-Pure Functions

Pure Functions
just return values



-2

abs

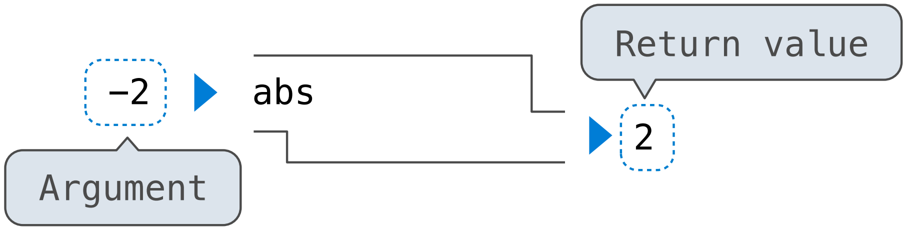
2

Argument

Non-Pure Functions
have side effects

Pure Functions & Non-Pure Functions

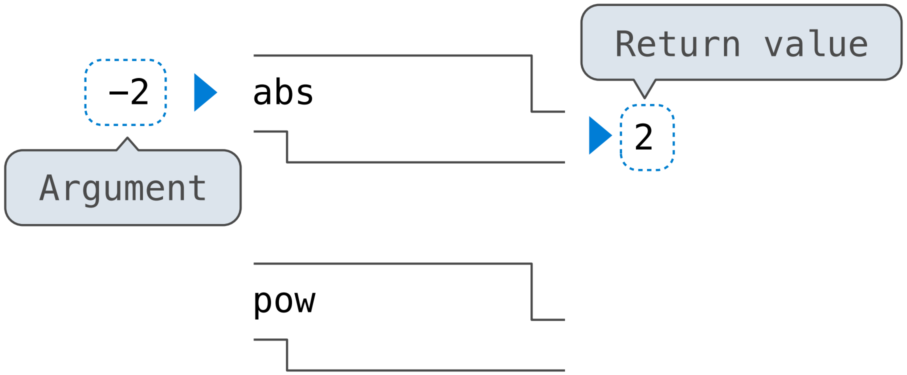
Pure Functions
just return values



Non-Pure Functions
have side effects

Pure Functions & Non-Pure Functions

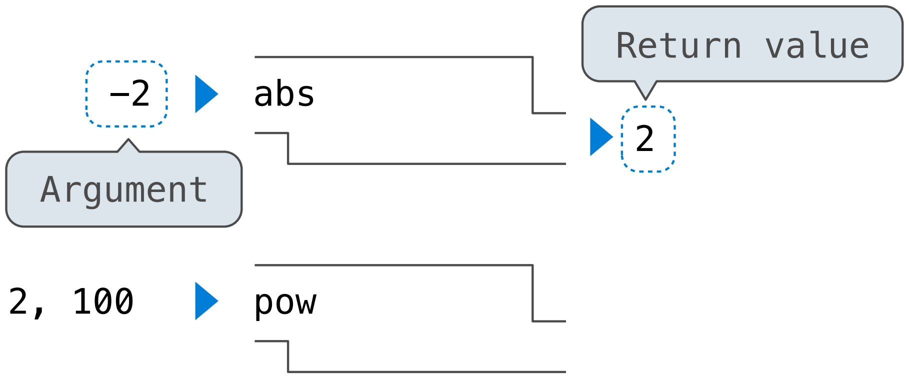
Pure Functions
just return values



Non-Pure Functions
have side effects

Pure Functions & Non-Pure Functions

Pure Functions
just return values



Return value

-2

abs

2

Argument

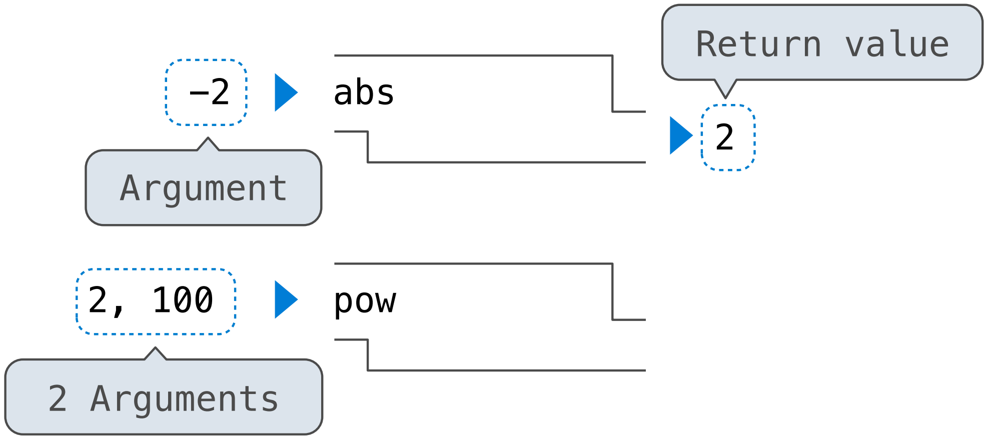
2, 100

pow

Non-Pure Functions
have side effects

Pure Functions & Non-Pure Functions

Pure Functions
just return values



Return value

-2

abs

2

Argument

2, 100

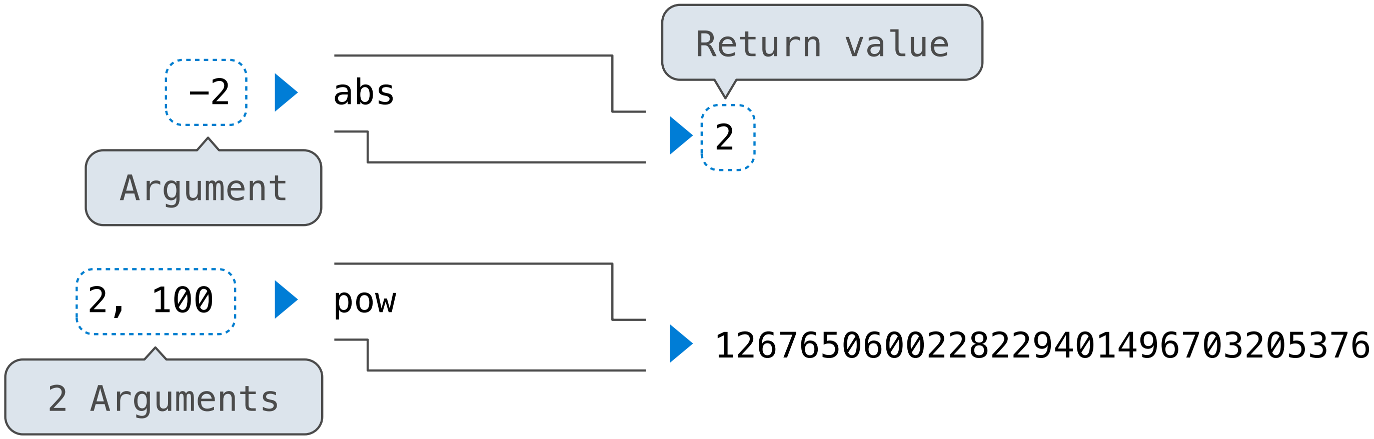
pow

2 Arguments

Non-Pure Functions
have side effects

Pure Functions & Non-Pure Functions

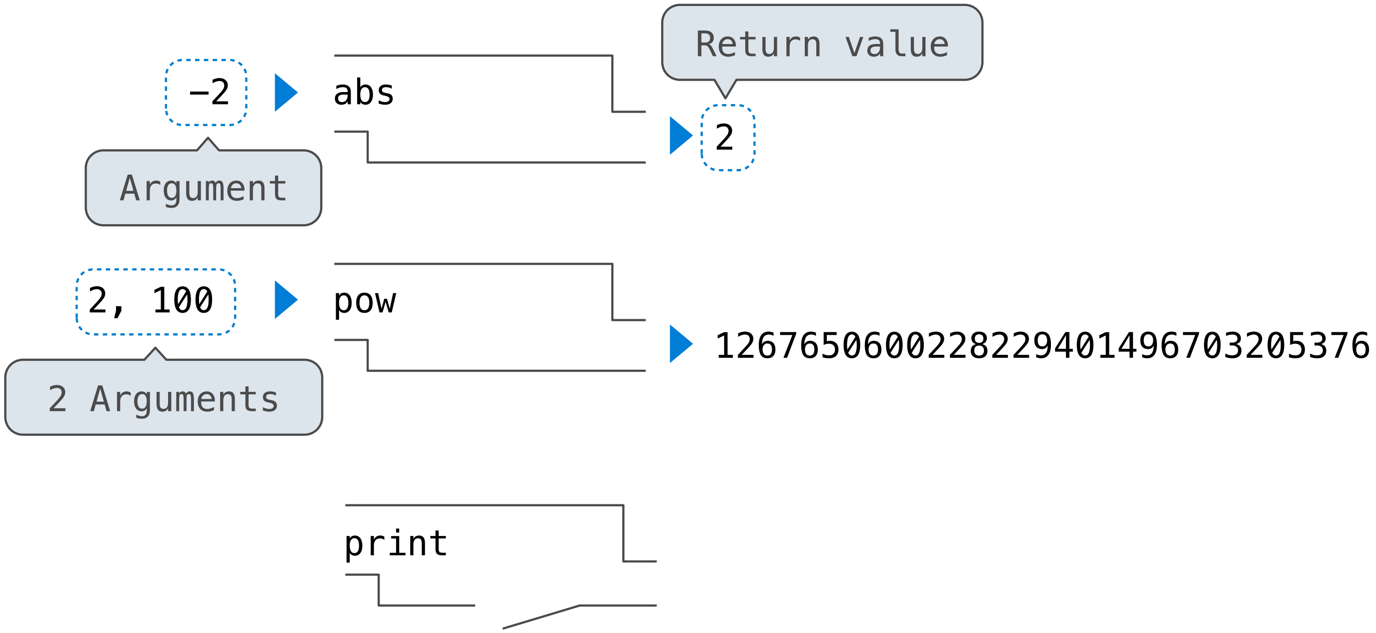
Pure Functions
just return values



Non-Pure Functions
have side effects

Pure Functions & Non-Pure Functions

Pure Functions
just return values



-2

abs

2

Argument

Return value

2, 100

pow

1267650600228229401496703205376

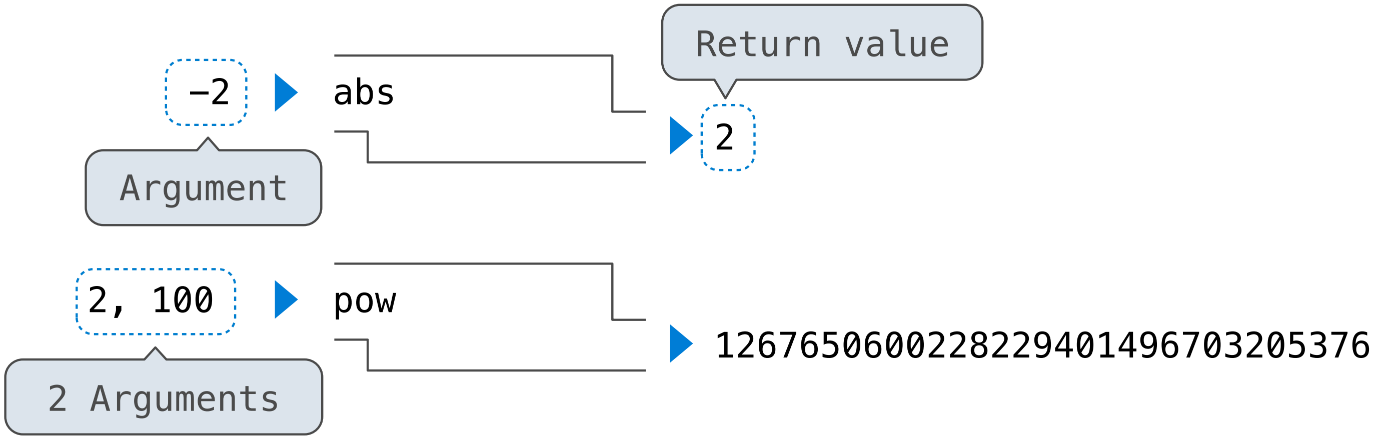
2 Arguments

Non-Pure Functions
have side effects

print

Pure Functions & Non-Pure Functions

Pure Functions
just return values



-2

abs

Return value

2

Argument

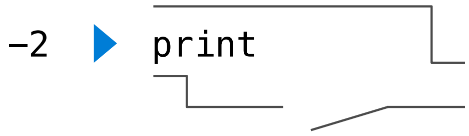
2, 100

pow

1267650600228229401496703205376

2 Arguments

Non-Pure Functions
have side effects

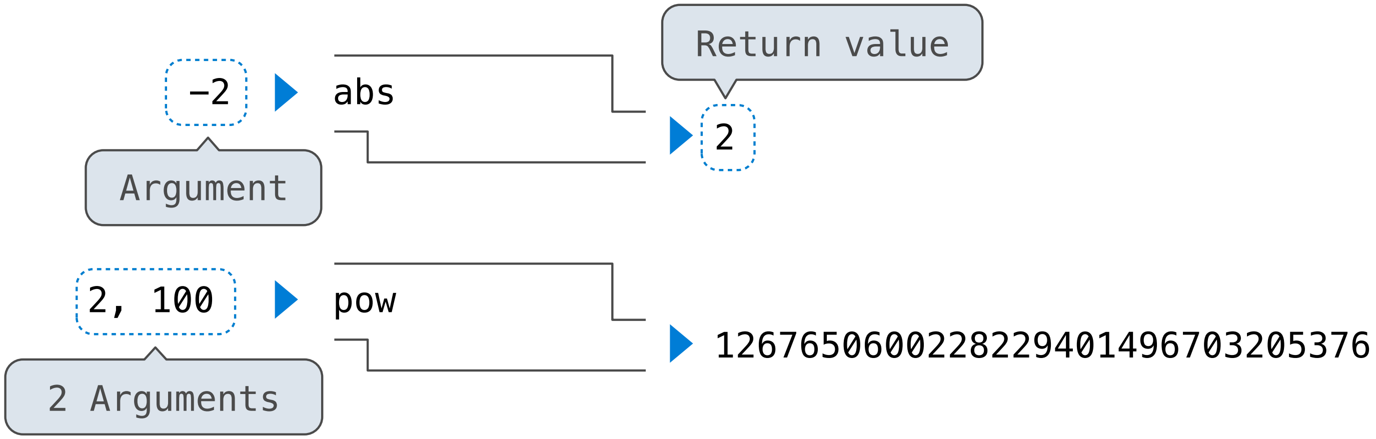


-2

print

Pure Functions & Non-Pure Functions

Pure Functions
just return values



-2

Argument

abs

Return value

2

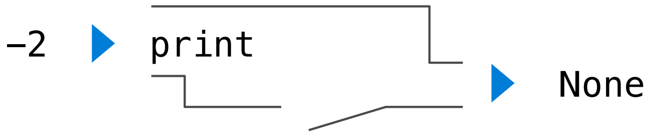
2, 100

2 Arguments

pow

1267650600228229401496703205376

Non-Pure Functions
have side effects



-2

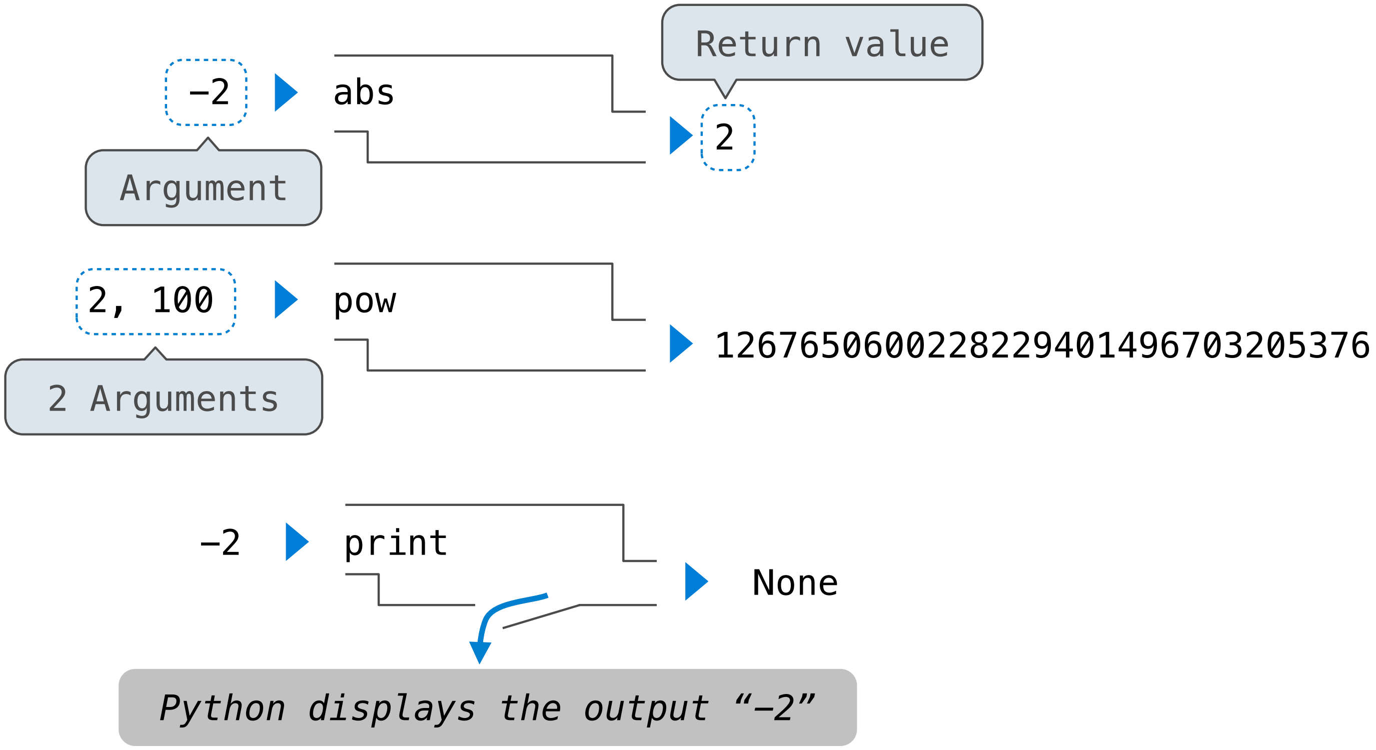
print

None

Pure Functions & Non-Pure Functions

Pure Functions
just return values

Non-Pure Functions
have side effects



Return value

-2

abs

2

Argument

2, 100

pow

1267650600228229401496703205376

2 Arguments

-2

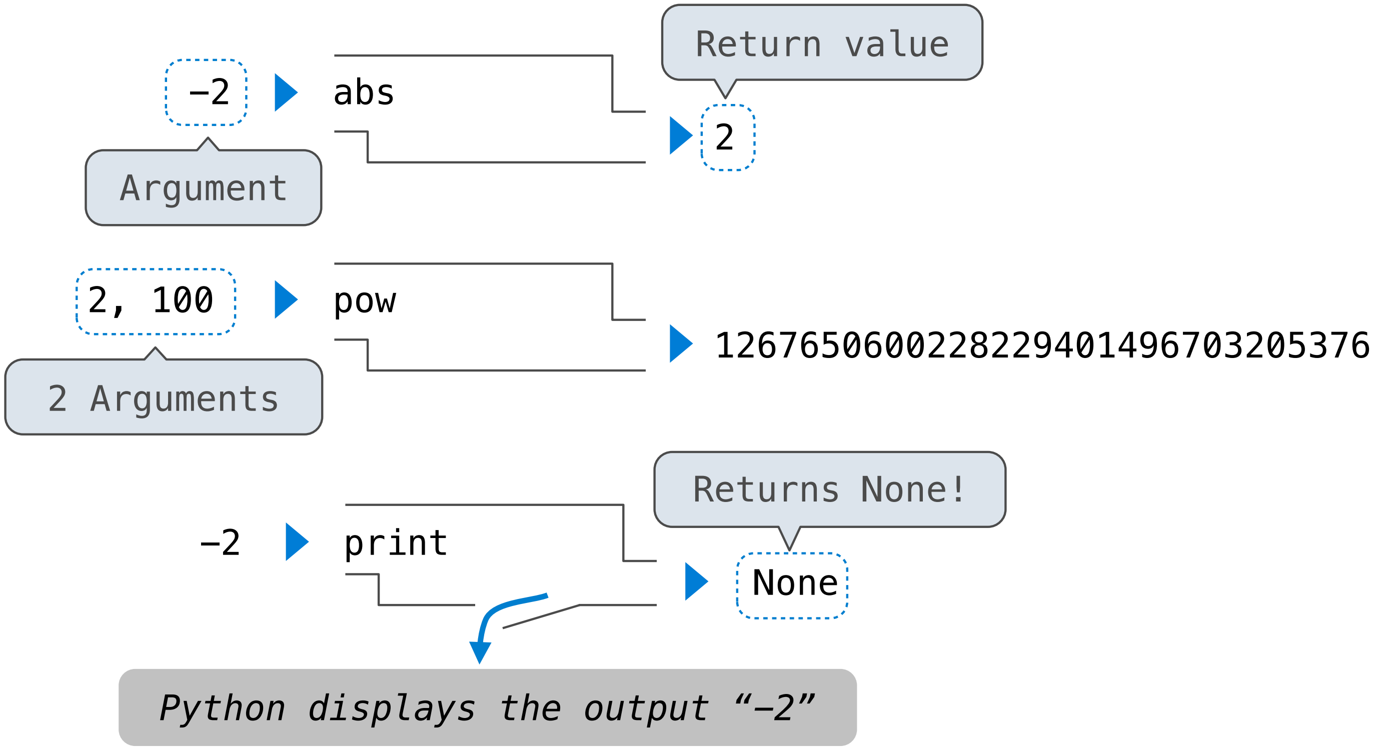
print

None

Python displays the output "-2"

Pure Functions & Non-Pure Functions

Pure Functions
just return values



Return value

−2 abs 2

Argument

2, 100 pow 1267650600228229401496703205376

2 Arguments

Non-Pure Functions
have side effects

Returns None!

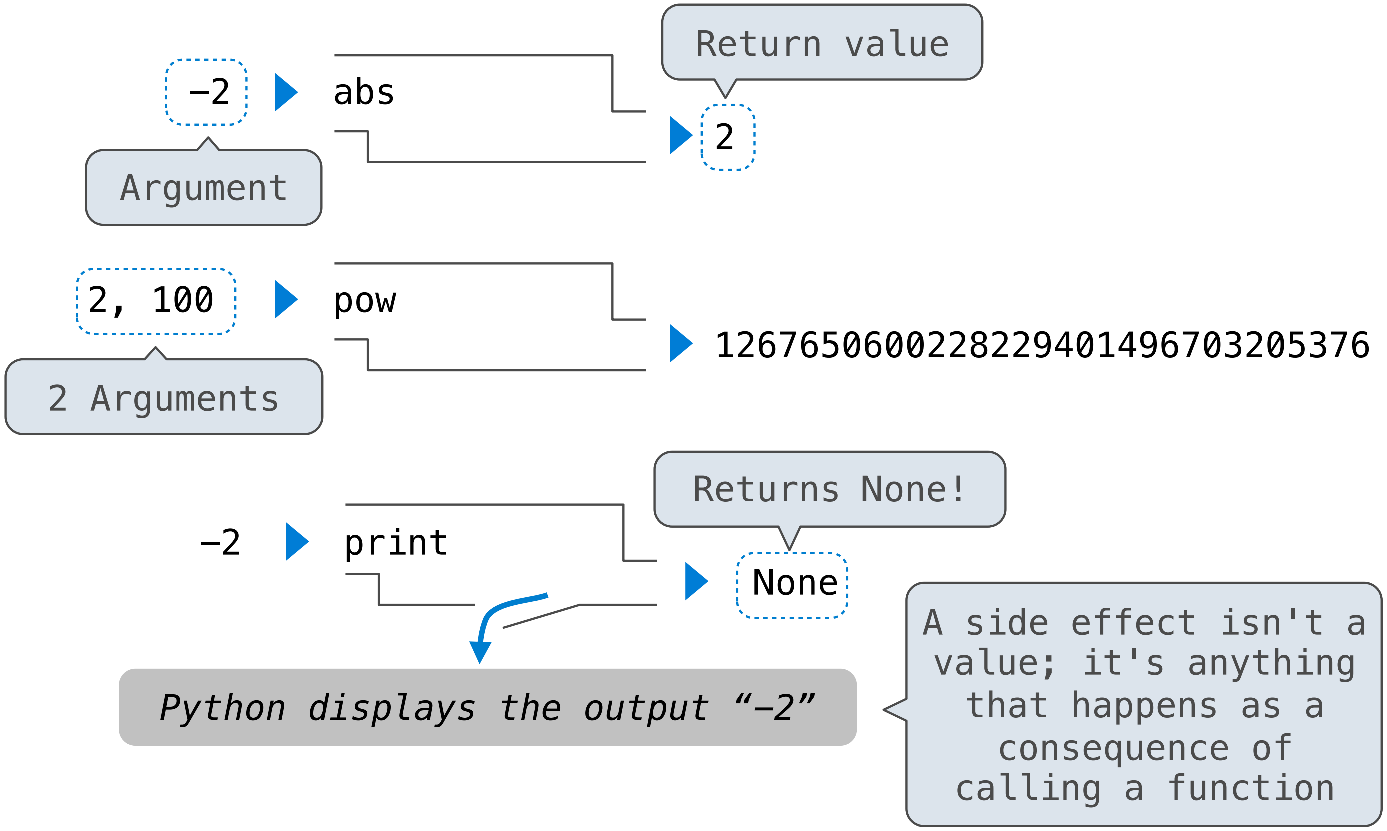
−2 print None

Python displays the output "–2"

Pure Functions & Non-Pure Functions

Pure Functions
just return values

Non-Pure Functions
have side effects



-2

abs

2

Return value

Argument

2, 100

pow

1267650600228229401496703205376

2 Arguments

-2

print

None

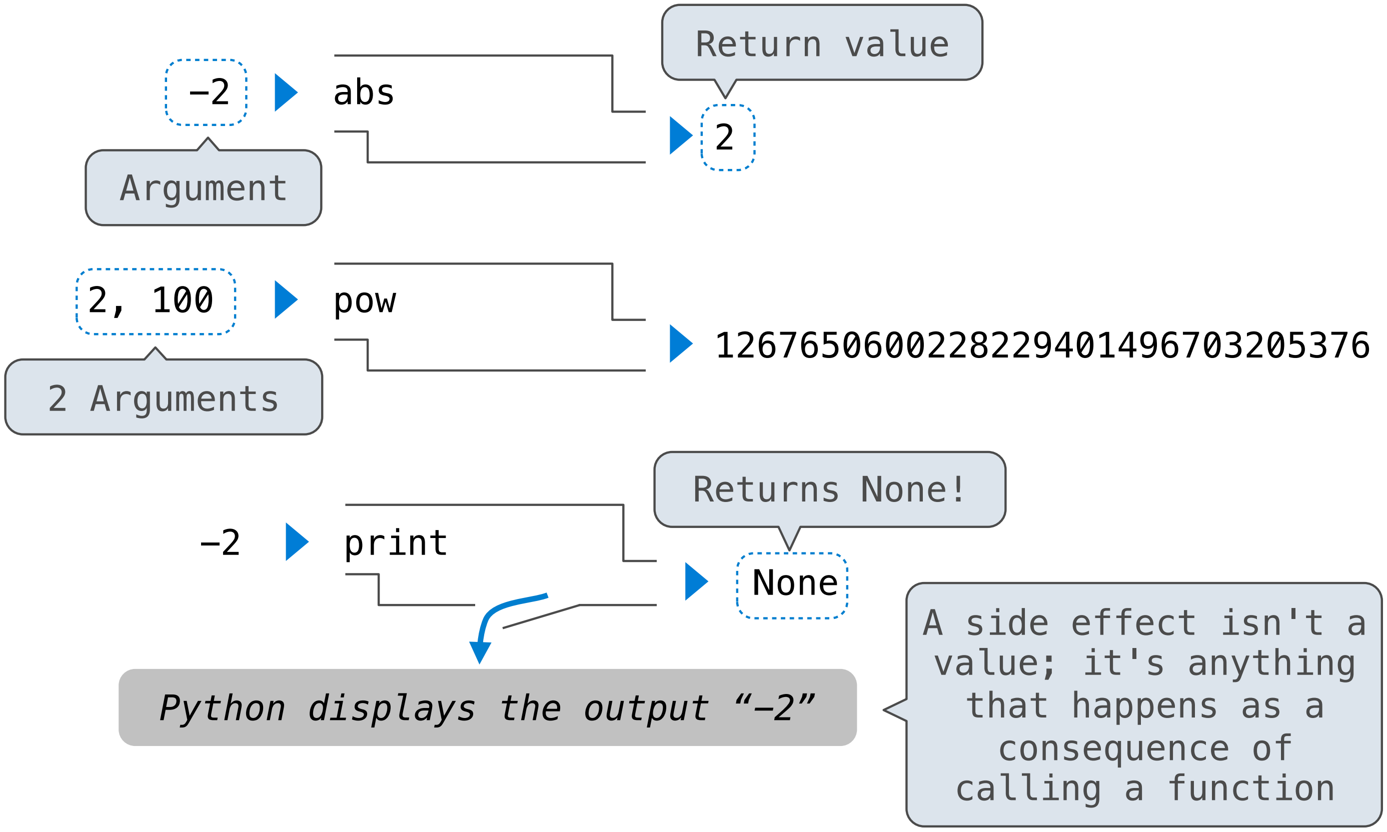
Returns None!

Python displays the output “-2”

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

Pure Functions & Non-Pure Functions

Pure Functions
just return values



Return value

-2 abs 2

Argument

2, 100 pow 1267650600228229401496703205376

2 Arguments

Non-Pure Functions
have side effects

Returns None!

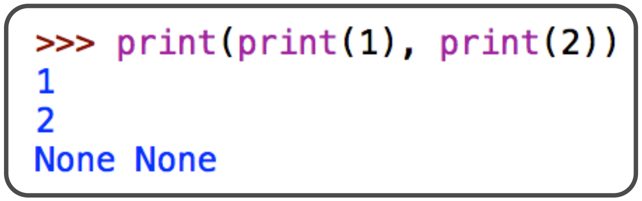
-2 print None

Python displays the output "-2"

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

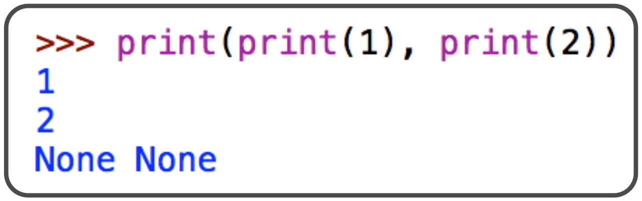
(Demo)

Nested Expressions with Print



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

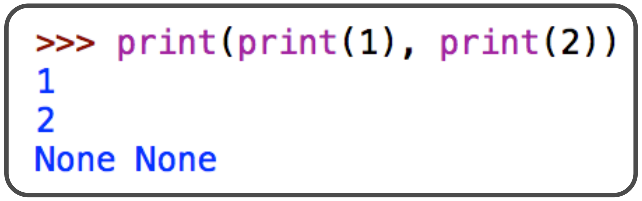
Nested Expressions with Print



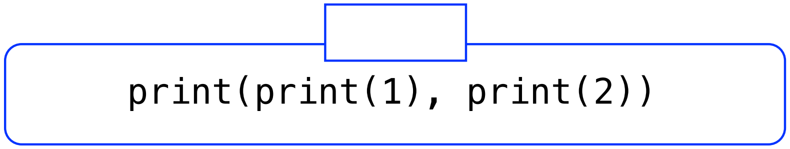
```
>>> print(print(1), print(2))
1
2
None None
```

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

Nested Expressions with Print

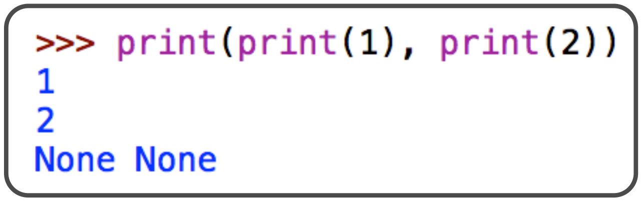


```
>>> print(print(1), print(2))
1
2
None None
```

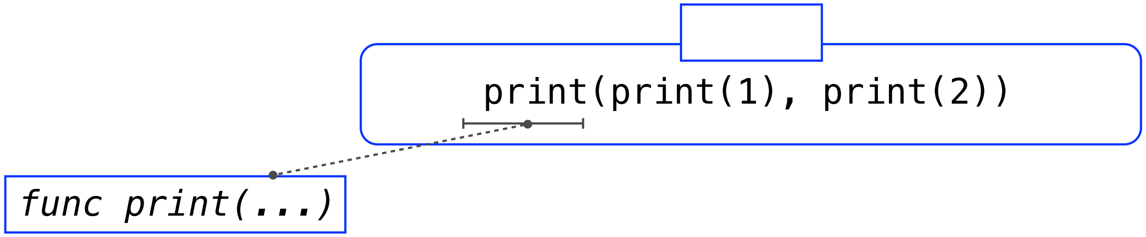


print(print(1), print(2))

Nested Expressions with Print



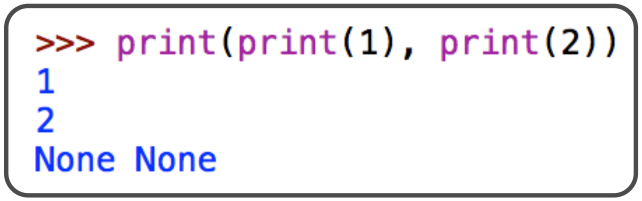
```
>>> print(print(1), print(2))
1
2
None None
```



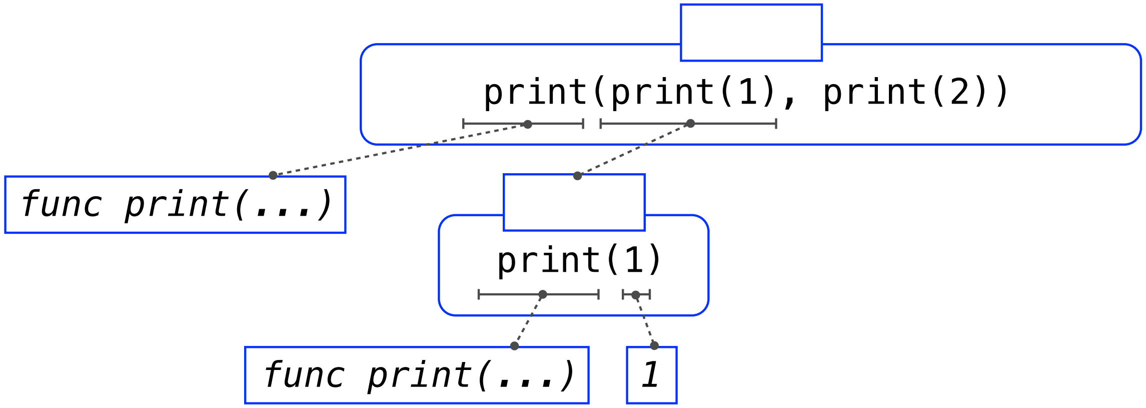
print(print(1), print(2))

func print(...)

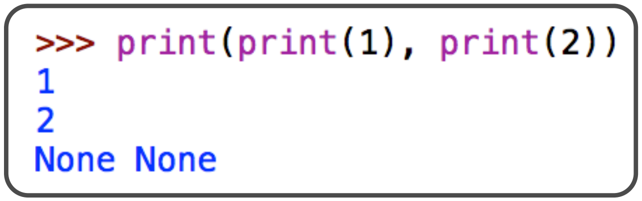
Nested Expressions with Print



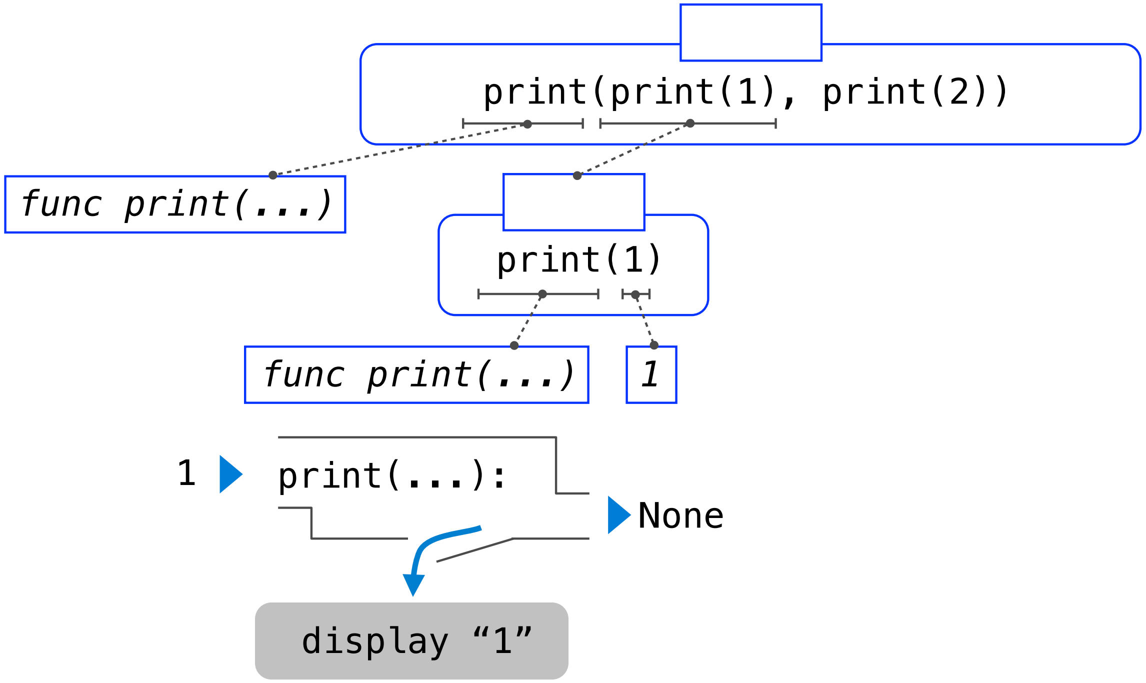
```
>>> print(print(1), print(2))
1
2
None None
```



Nested Expressions with Print

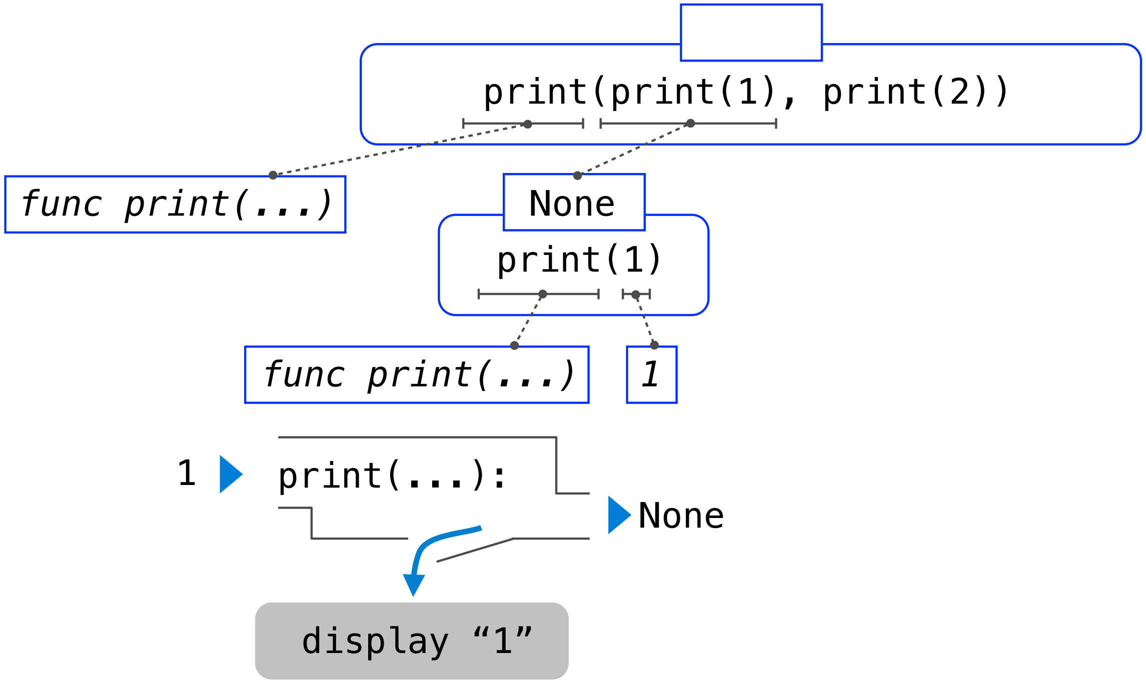


```
>>> print(print(1), print(2))
1
2
None None
```

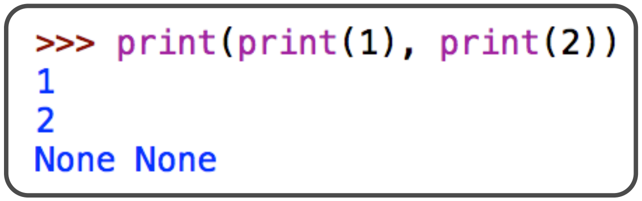


Nested Expressions with Print

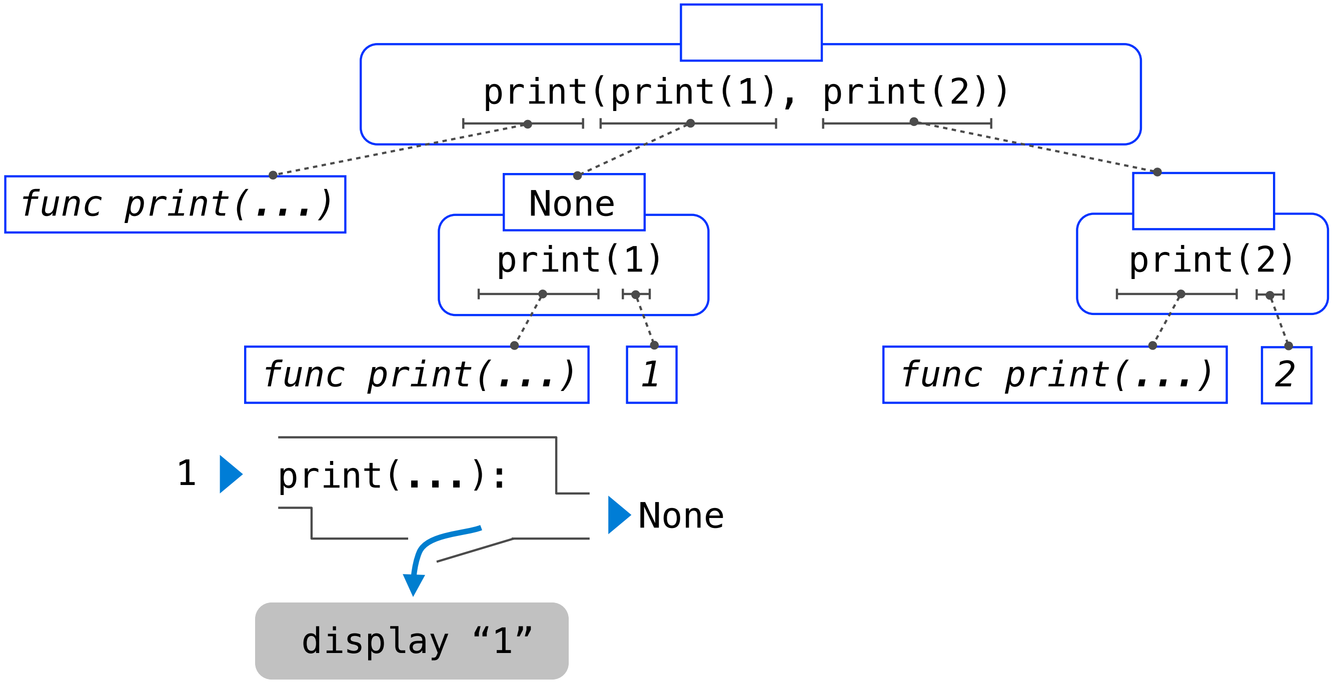
```
>>> print(print(1), print(2))
1
2
None None
```



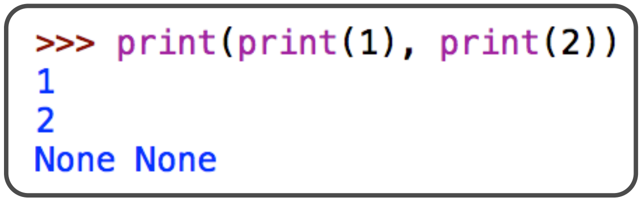
Nested Expressions with Print



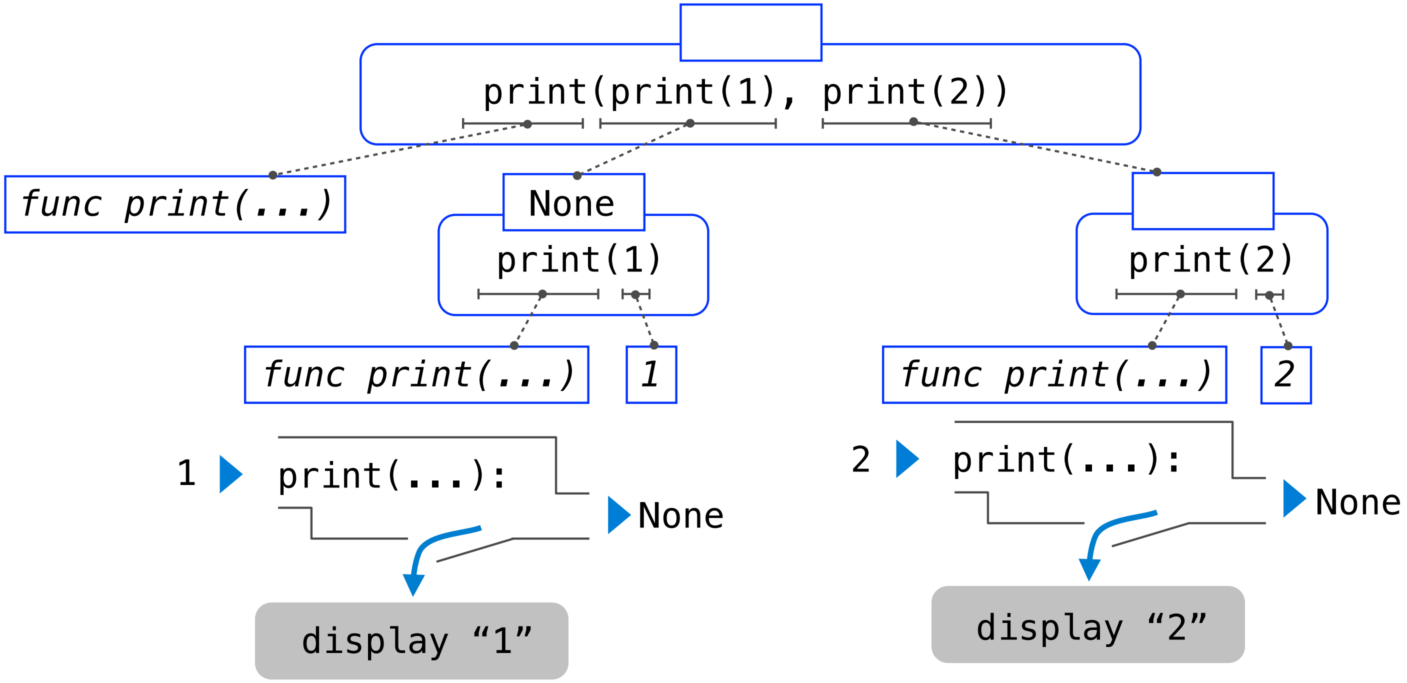
```
>>> print(print(1), print(2))
1
2
None None
```



Nested Expressions with Print

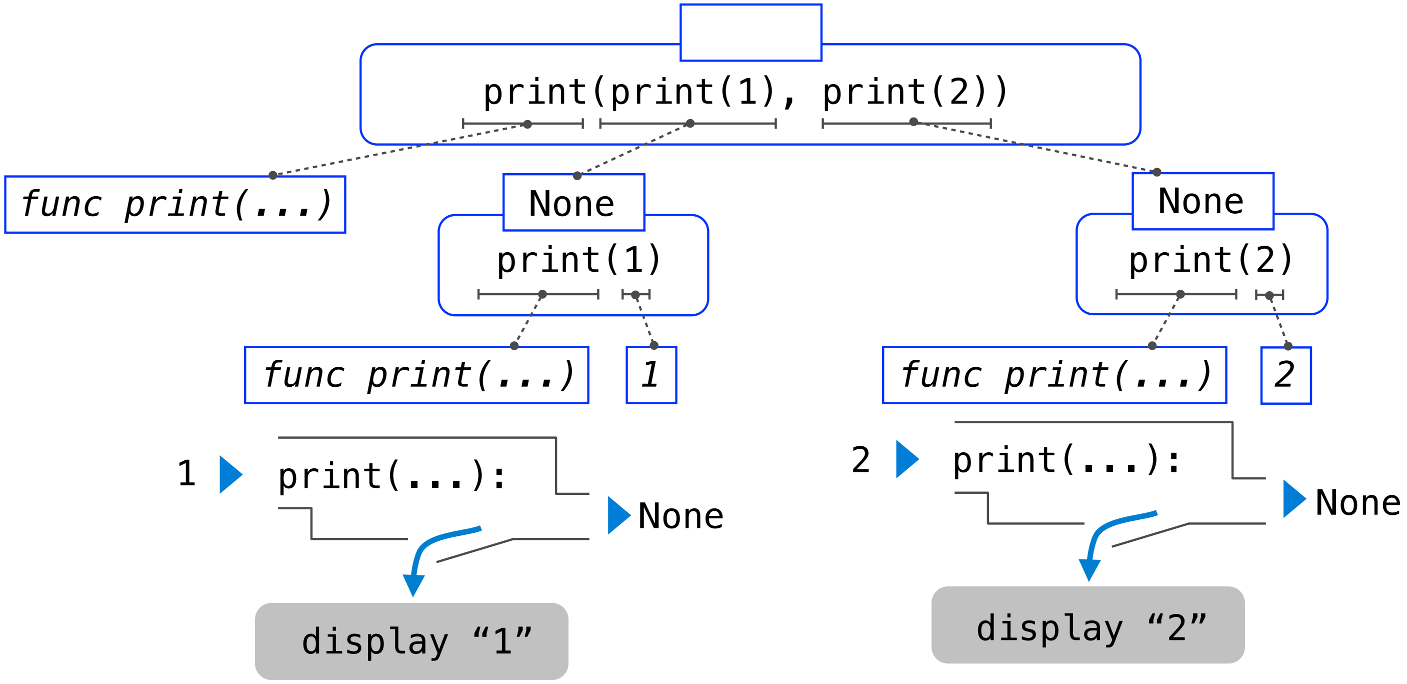


```
>>> print(print(1), print(2))
1
2
None None
```

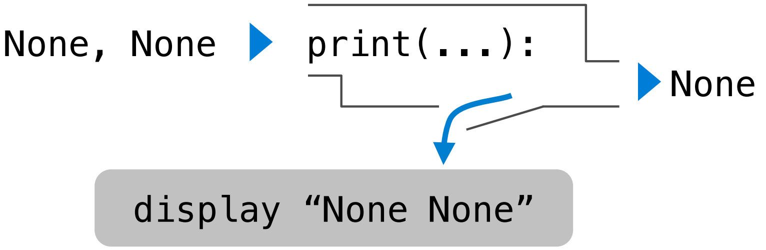


Nested Expressions with Print

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



Nested Expressions with Print

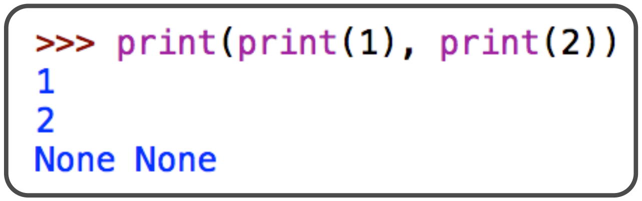


None, None

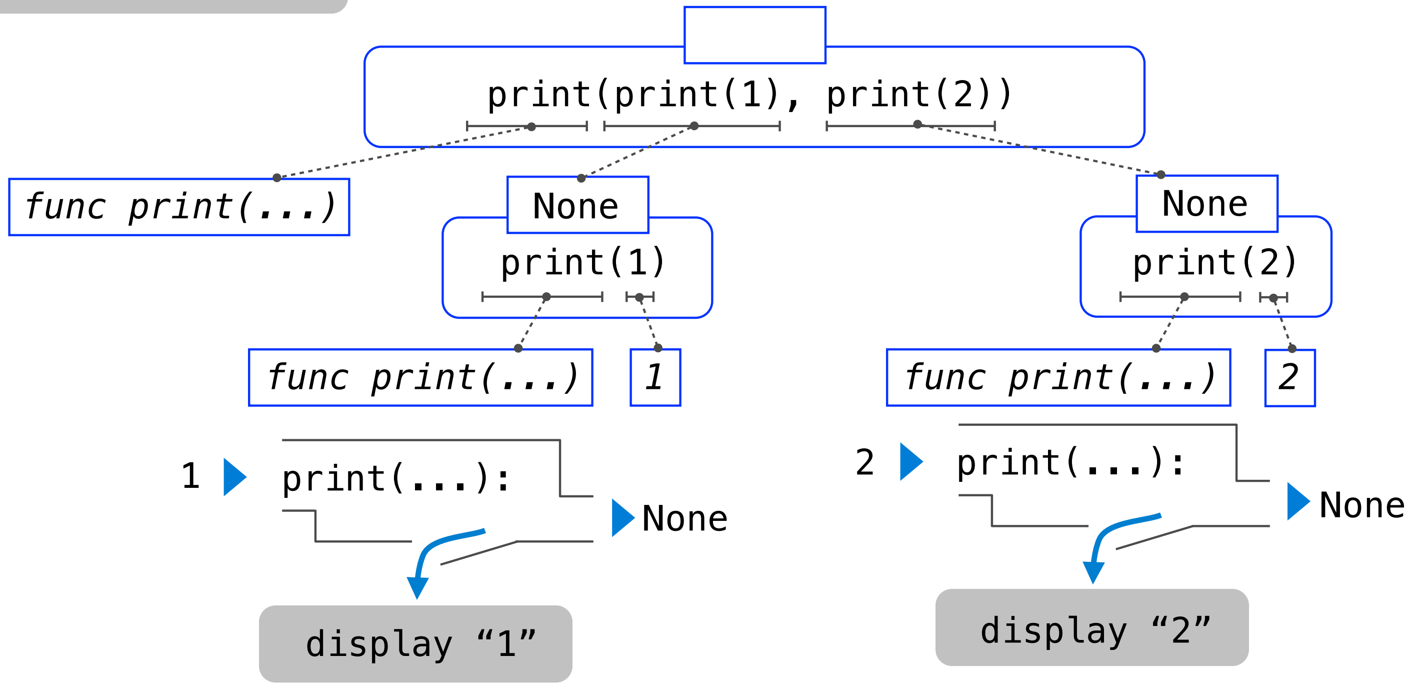
print(...):

None

display “None None”



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



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

func print(...)

None

print(1)

func print(...)

1

None

print(2)

func print(...)

2

1

print(...):

None

display “1”

2

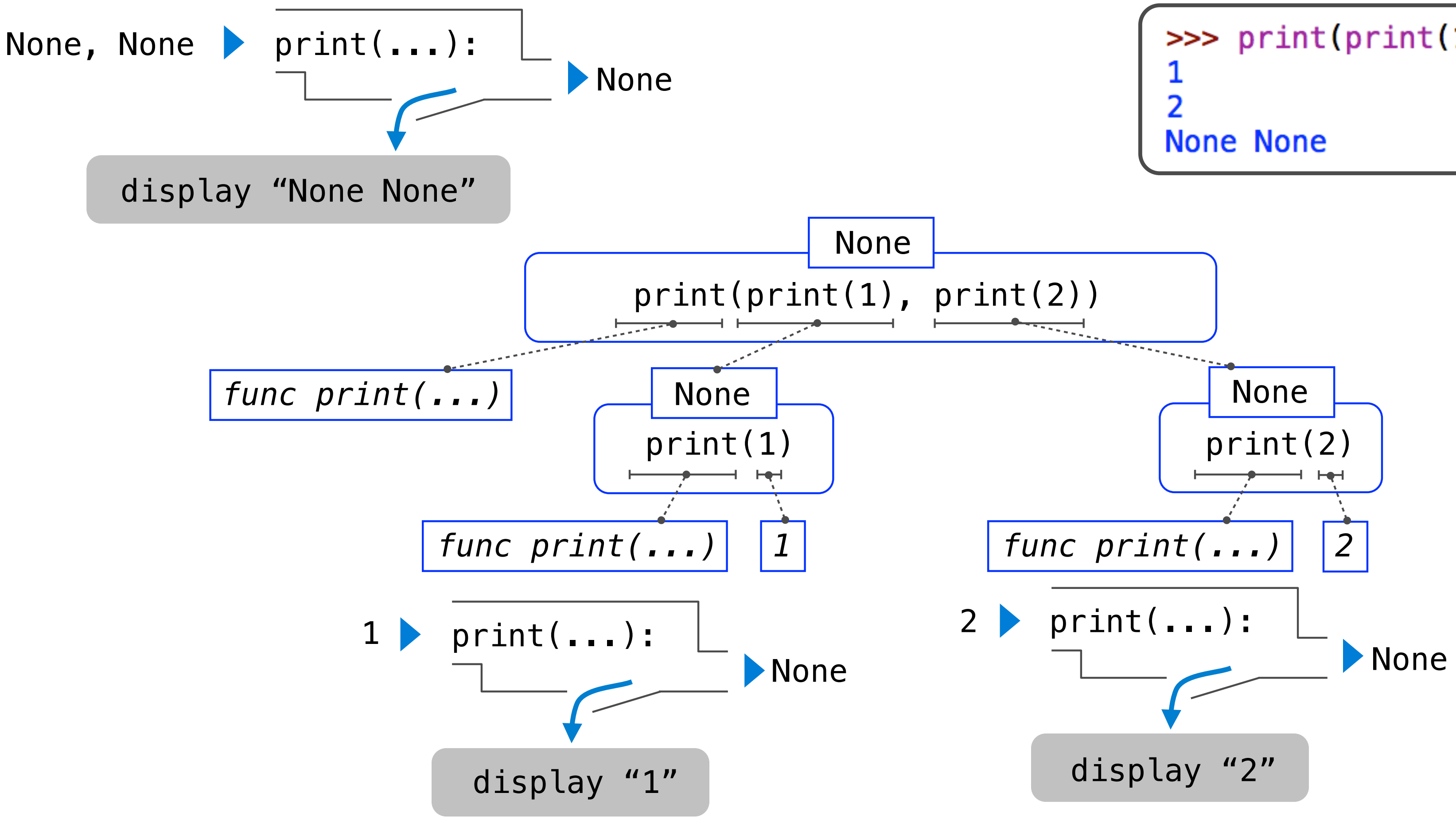
print(...):

None

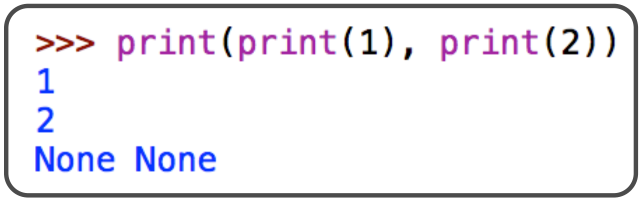
display “2”

Nested Expressions with Print

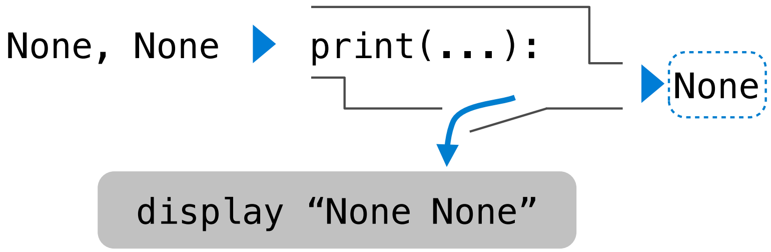
```
>>> print(print(1), print(2))
1
2
None None
```



Nested Expressions with Print



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

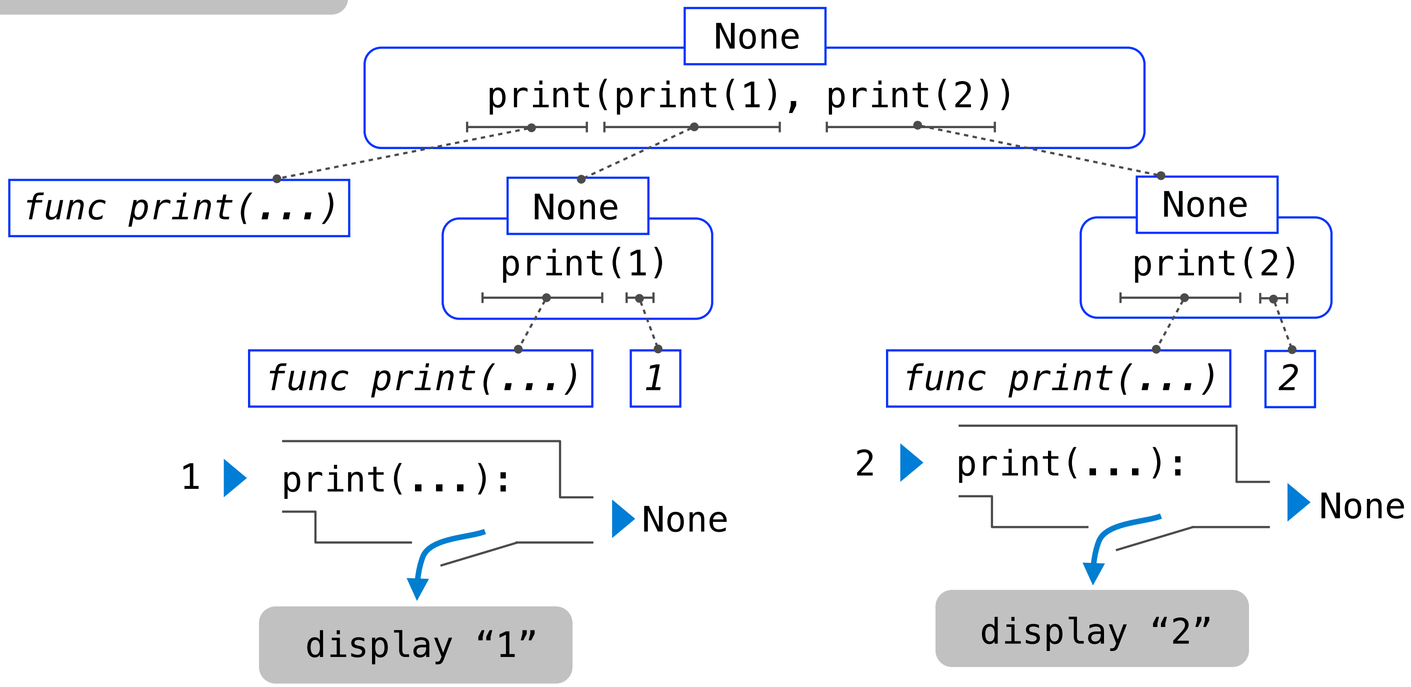


None, None

print(...):

None

display “None None”



None

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

func print(...)

None

print(1)

func print(...)

1

None

print(2)

func print(...)

2

1

print(...):

None

display “1”

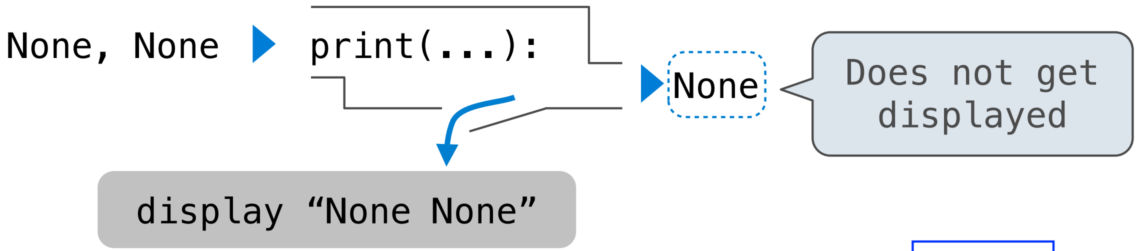
2

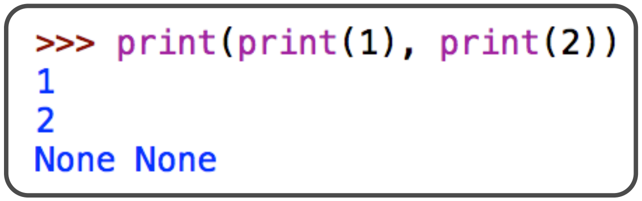
print(...):

None

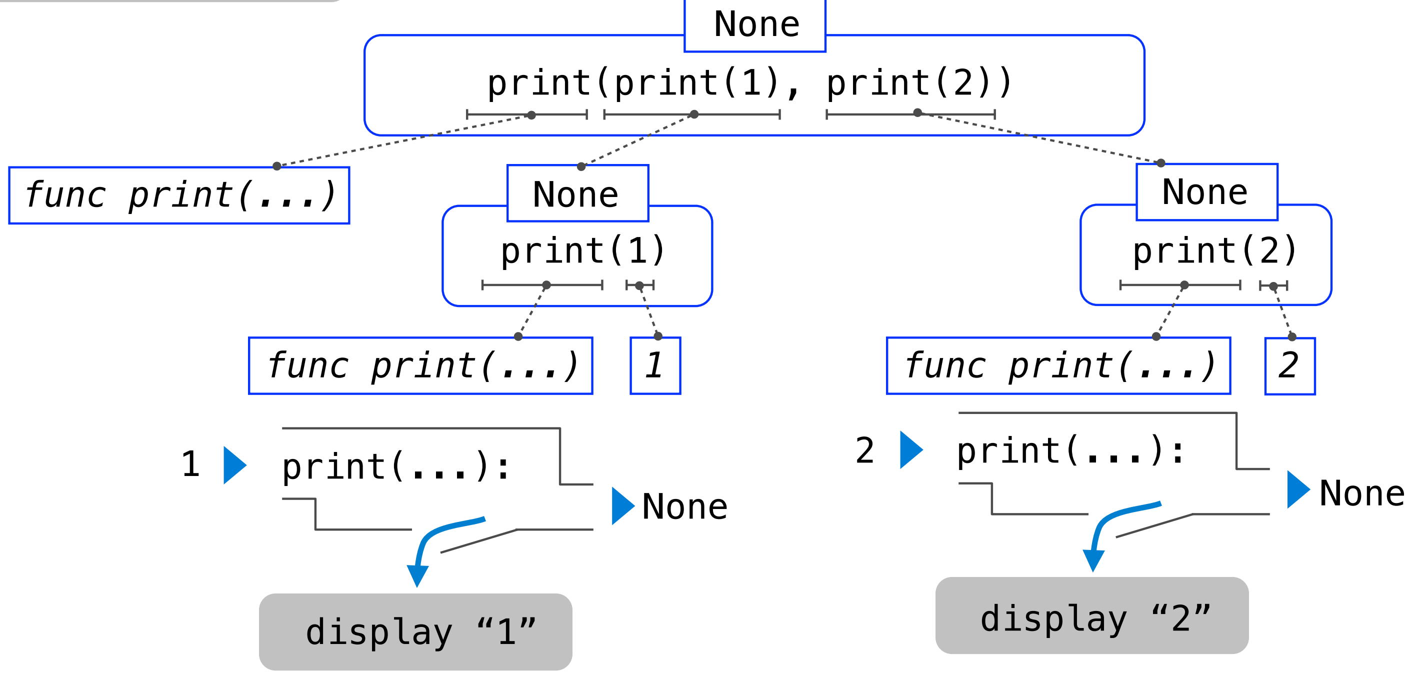
display “2”

Nested Expressions with Print





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



Multiple Environments

Life Cycle of a User-Defined Function

What happens?

Def statement:

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

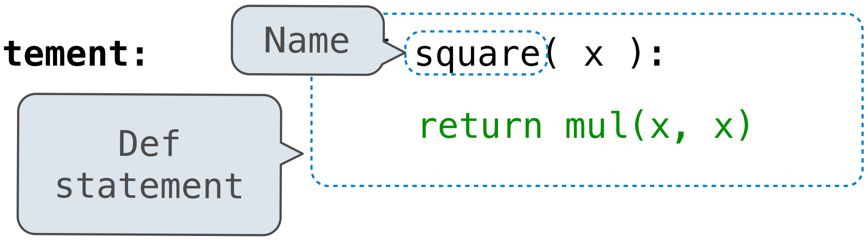
Call expression:

Calling/Applying:

Life Cycle of a User-Defined Function

What happens?

Def statement:



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

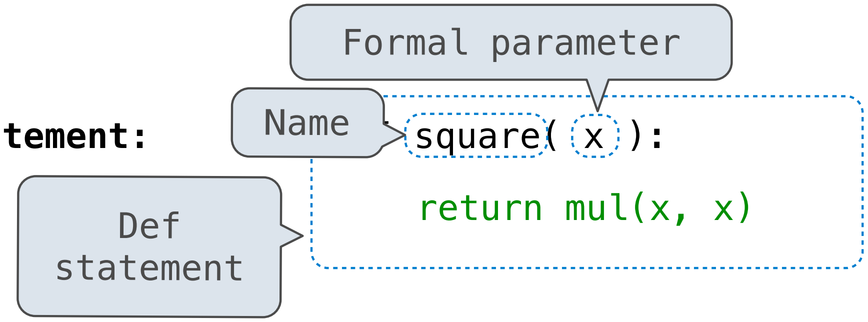
Call expression:

Calling/Applying:

Life Cycle of a User-Defined Function

What happens?

Def statement:



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

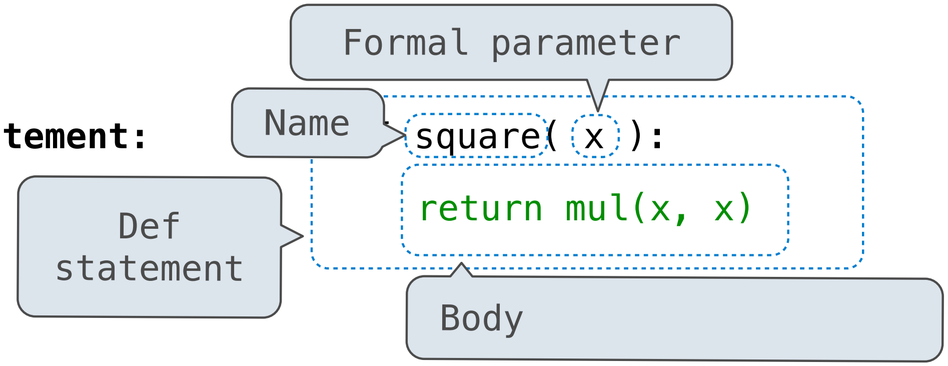
Call expression:

Calling/Applying:

Life Cycle of a User-Defined Function

What happens?

Def statement:



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

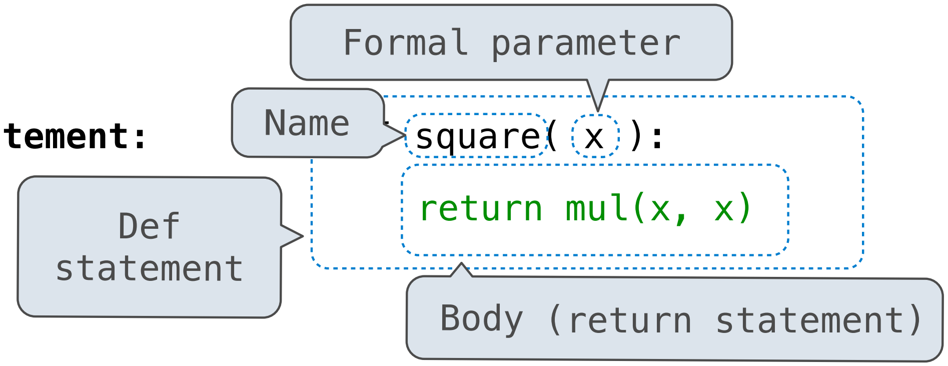
Call expression:

Calling/Applying:

Life Cycle of a User-Defined Function

What happens?

Def statement:



Formal parameter

Name

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

Def statement

Body (return statement)

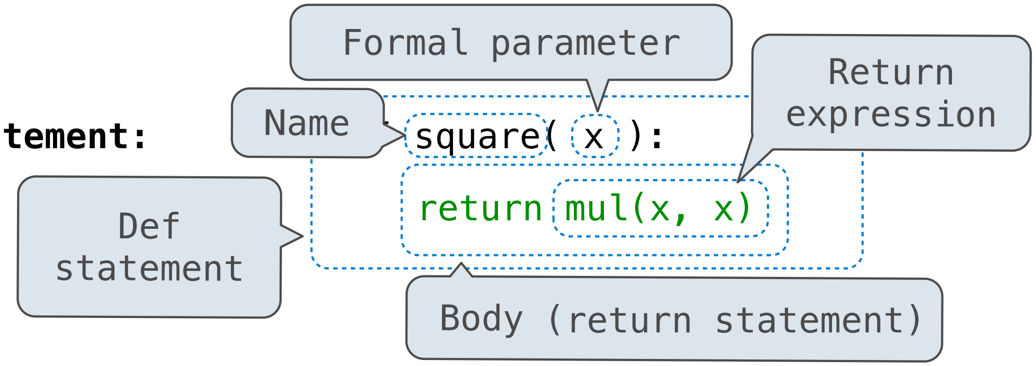
Call expression:

Calling/Applying:

Life Cycle of a User-Defined Function

What happens?

Def statement:



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

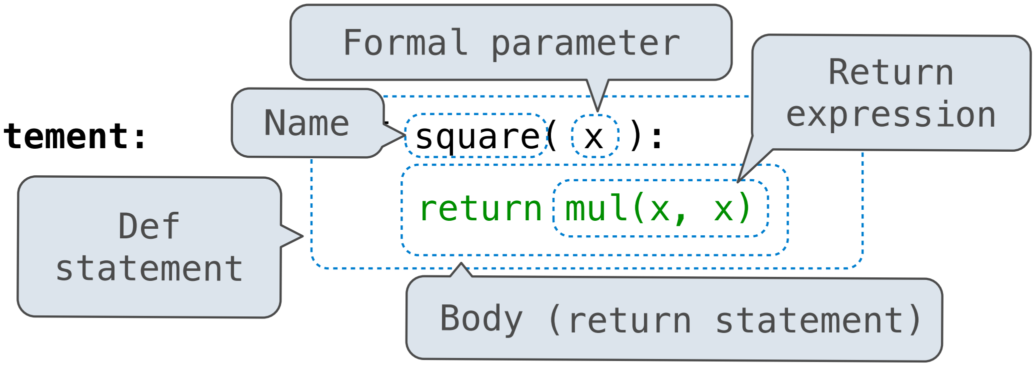
Call expression:

Calling/Applying:

Life Cycle of a User-Defined Function

What happens?

Def statement:



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

A new function is created!

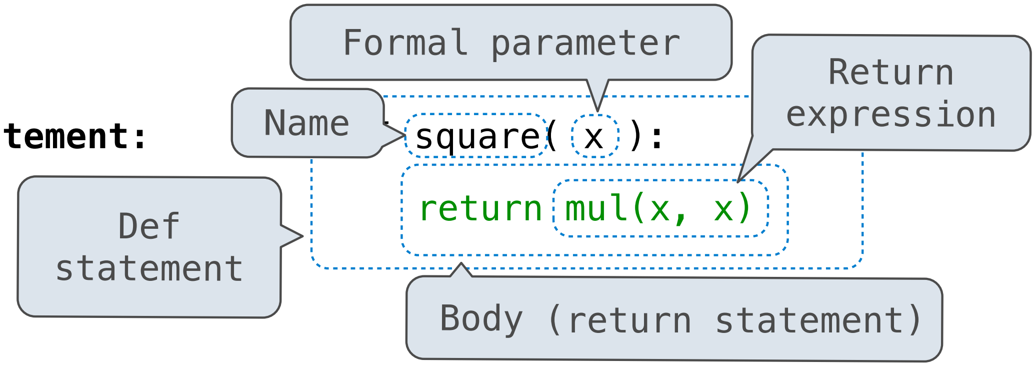
Call expression:

Calling/Applying:

Life Cycle of a User-Defined Function

What happens?

Def statement:



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

A new function is created!

Name bound to that function in the current frame

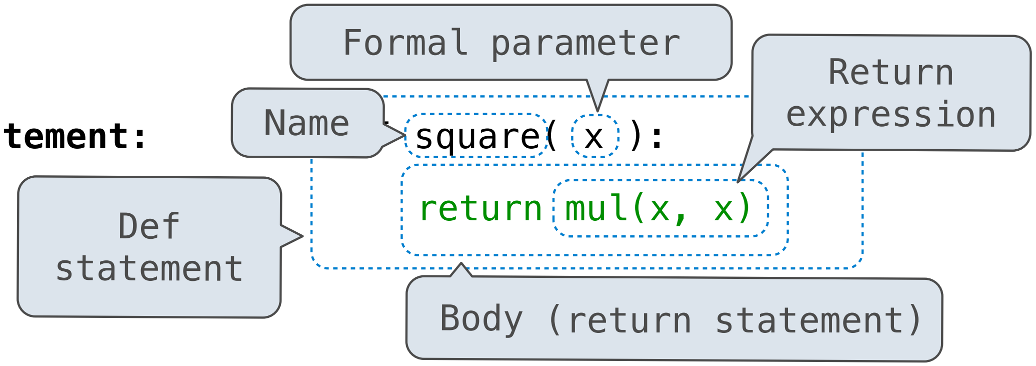
Call expression:

Calling/Applying:

Life Cycle of a User-Defined Function

What happens?

Def statement:



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

A new function is created!

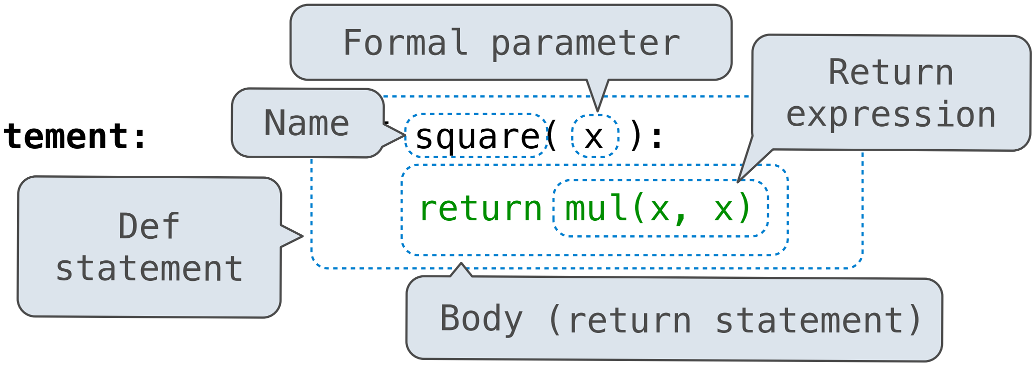
Name bound to that function in the current frame

Call expression: square(2+2)

Calling/Applying:

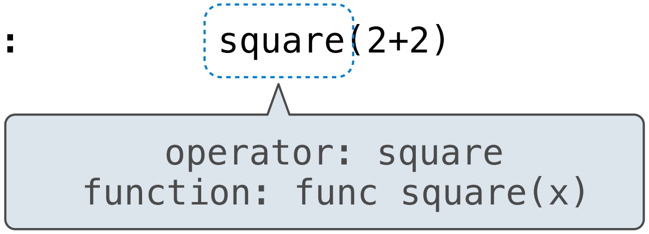
Life Cycle of a User-Defined Function

Def statement:



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

Call expression:



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

Calling/Applying:

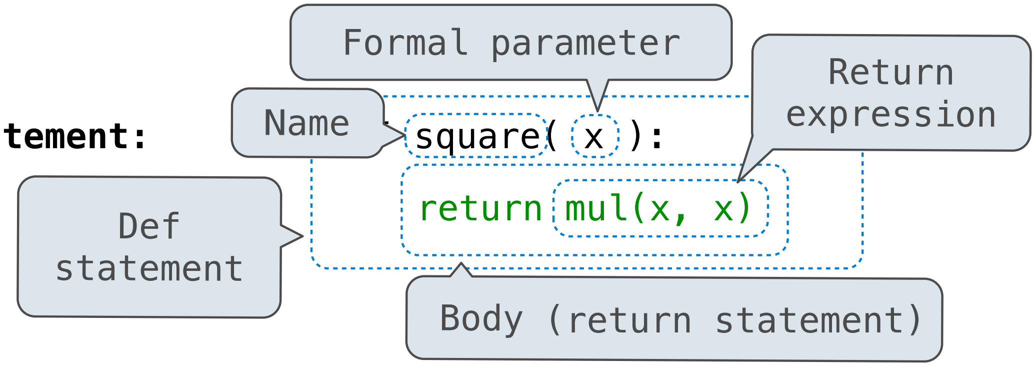
What happens?

A new function is created!

Name bound to that function
in the current frame

Life Cycle of a User-Defined Function

Def statement:



Formal parameter

Name

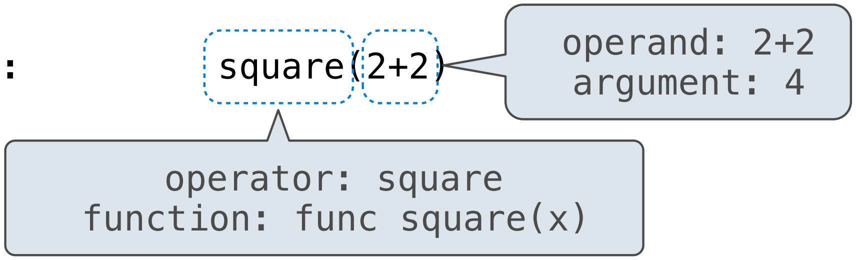
Return expression

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

Def statement

Body (return statement)

Call expression:



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

operand: 2+2
argument: 4

operator: square
function: func square(x)

Calling/Applying:

What happens?

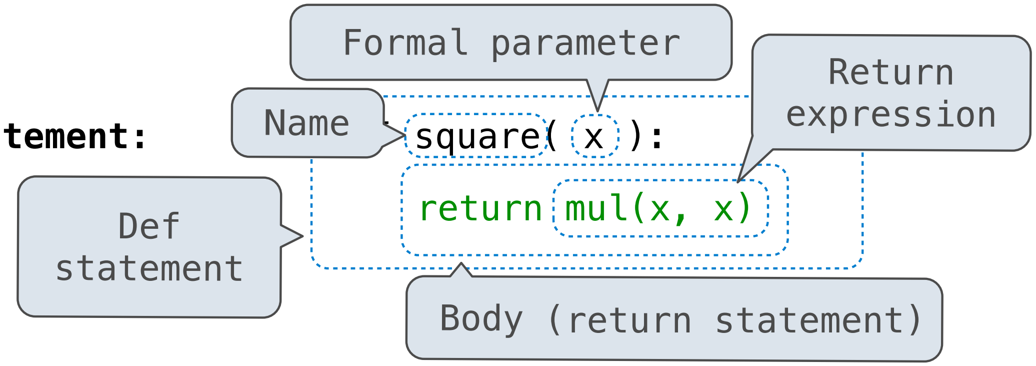
A new function is created!

Name bound to that function
in the current frame

Life Cycle of a User-Defined Function

What happens?

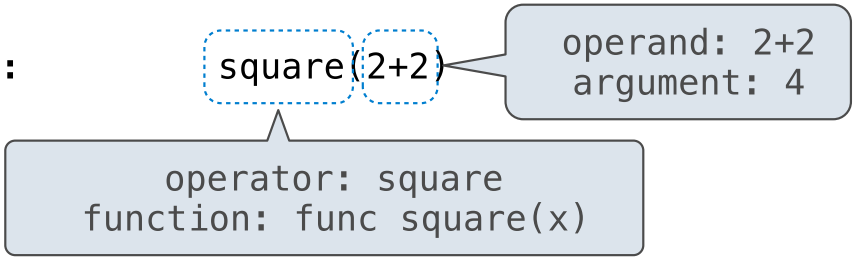
Def statement:


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

A new function is created!

Name bound to that function in the current frame

Call expression:


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

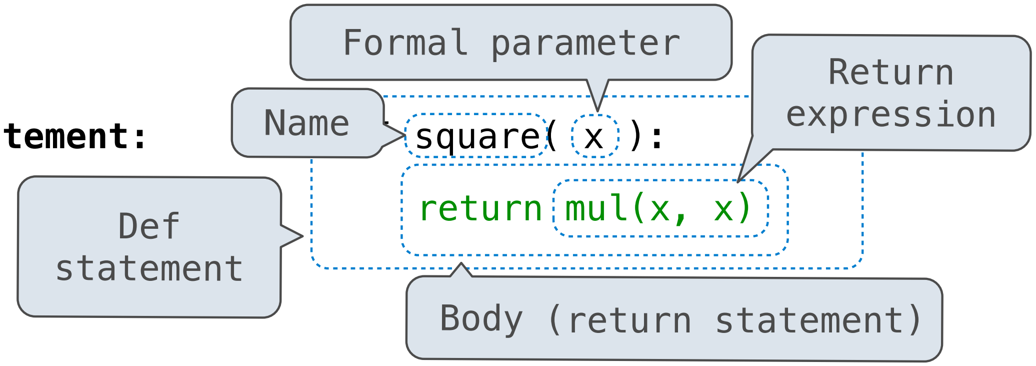
Operator & operands evaluated

Calling/Applying:

Life Cycle of a User-Defined Function

What happens?

Def statement:

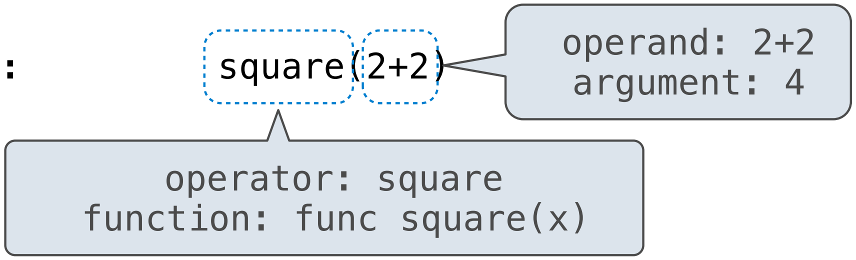


```
square( x ):
    return mul(x, x)
```

A new function is created!

Name bound to that function
in the current frame

Call expression:



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

Operator & operands evaluated

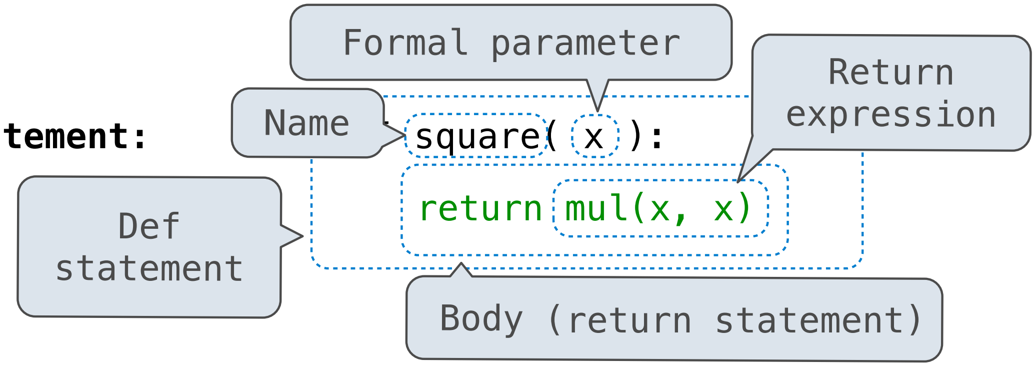
Function (value of operator)
called on arguments
(values of operands)

Calling/Applying:

Life Cycle of a User-Defined Function

What happens?

Def statement:



Formal parameter

Name

Return expression

square(x):

return mul(x, x)

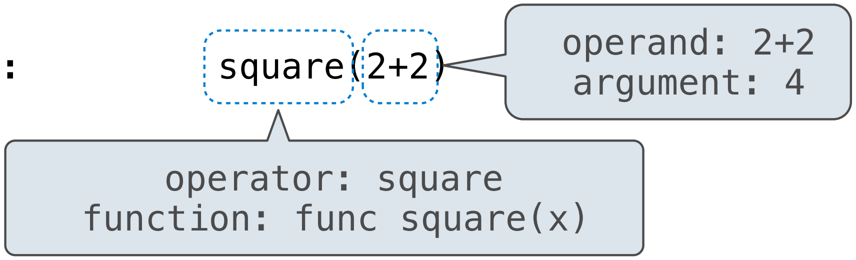
Def statement

Body (return statement)

A new function is created!

Name bound to that function in the current frame

Call expression:



square(2+2)

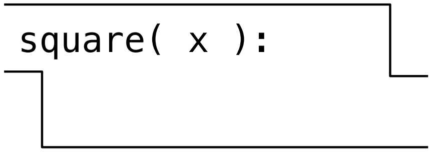
operand: 2+2
argument: 4

operator: square
function: func square(x)

Operator & operands evaluated

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

Calling/Applying:

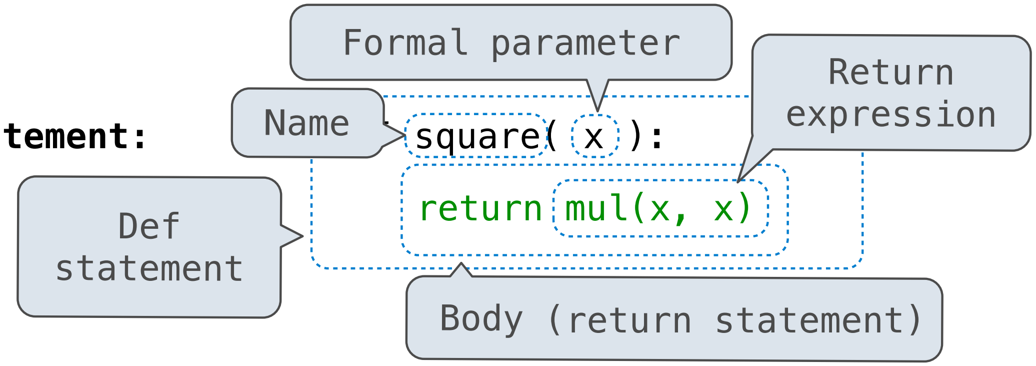


square(x):

Life Cycle of a User-Defined Function

What happens?

Def statement:



Formal parameter

Name

Return expression

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

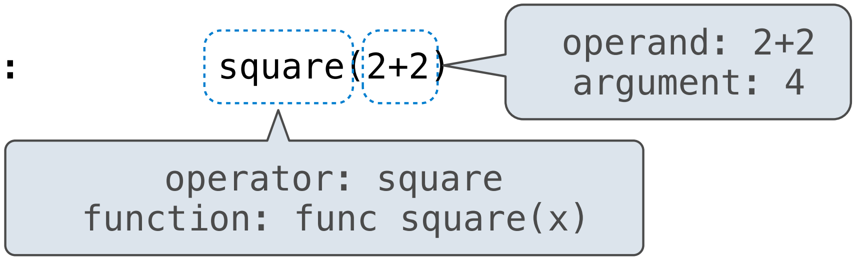
Def statement

Body (return statement)

A new function is created!

Name bound to that function
in the current frame

Call expression:



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

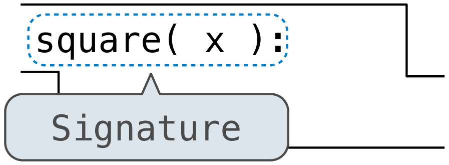
operand: 2+2
argument: 4

operator: square
function: func square(x)

Operator & operands evaluated

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

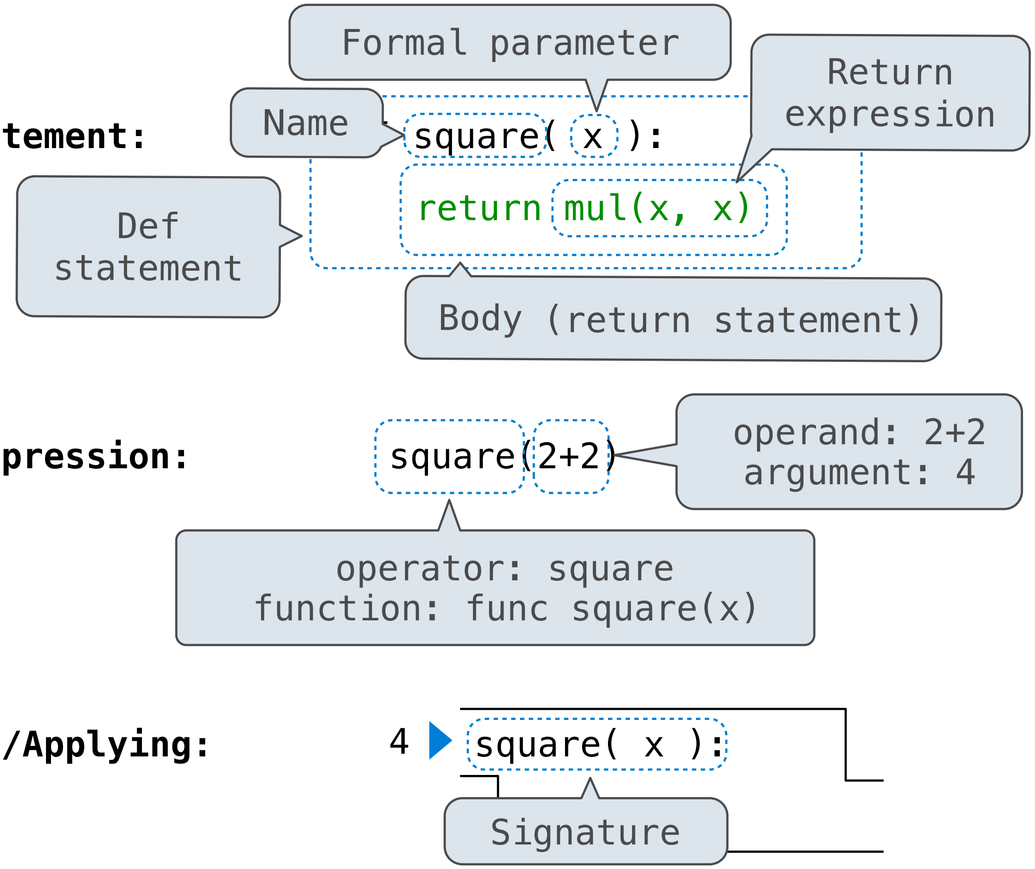
Calling/Applying:



```
square( x ):
```

Signature

Life Cycle of a User-Defined Function



Def statement:

Formal parameter

Name

Return expression

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

Def statement

Body (return statement)

Call expression:

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

operand: 2+2
argument: 4

operator: square
function: func square(x)

Calling/Applying:

4 ▶ square(x):

Signature

What happens?

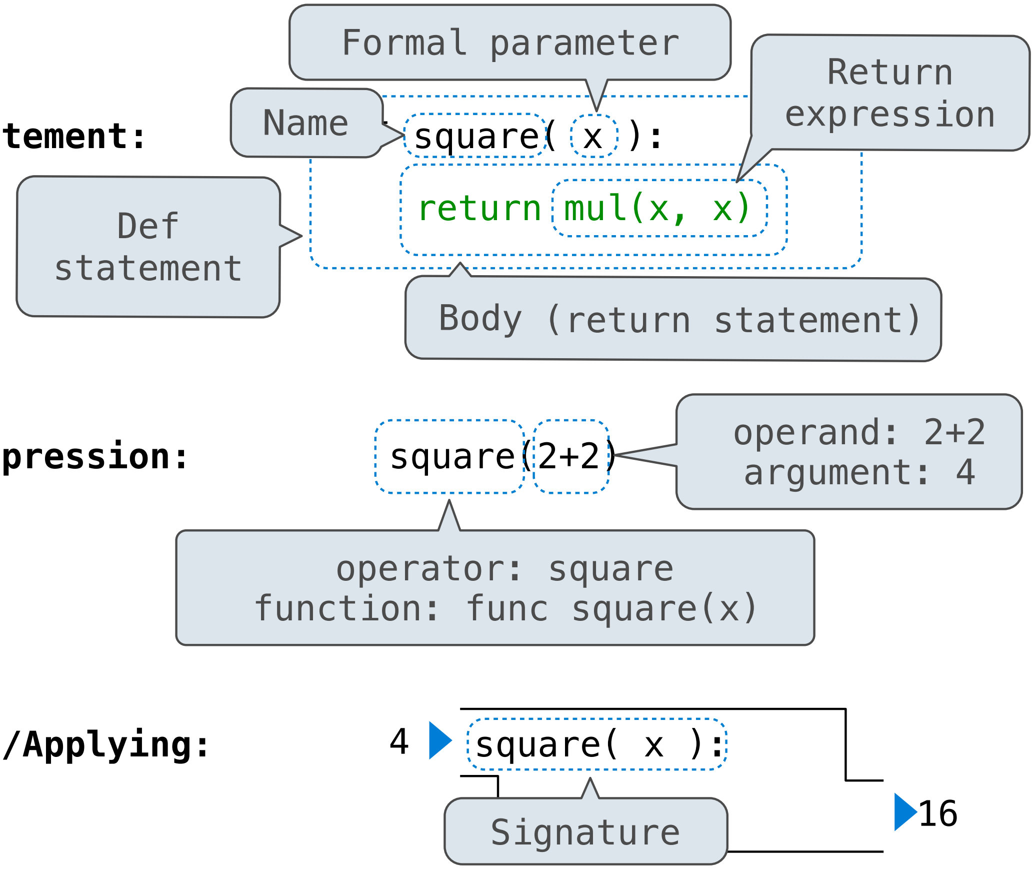
A new function is created!

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



Def statement:

Name

Formal parameter

Return expression

Def statement

Body (return statement)

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

Call expression:

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

operand: 2+2
argument: 4

operator: square
function: func square(x)

Calling/Applying:

4 ▶ square(x): ▶16

Signature

What happens?

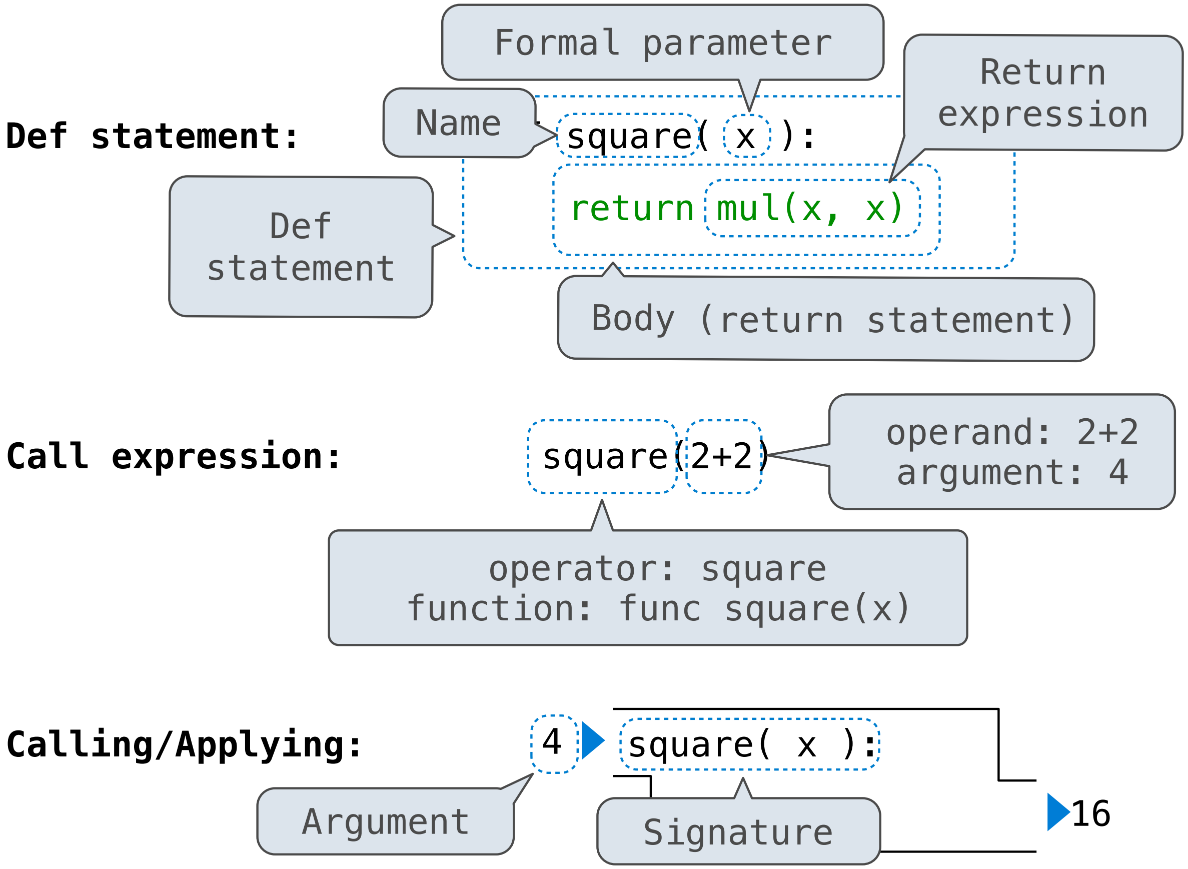
A new function is created!

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



Def statement:

Formal parameter

Name

Return expression

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

Def statement

Body (return statement)

Call expression:

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

operand: 2+2
argument: 4

operator: square
function: func square(x)

Calling/Applying:

4 ▶ square(x): ▶16

Argument

Signature

What happens?

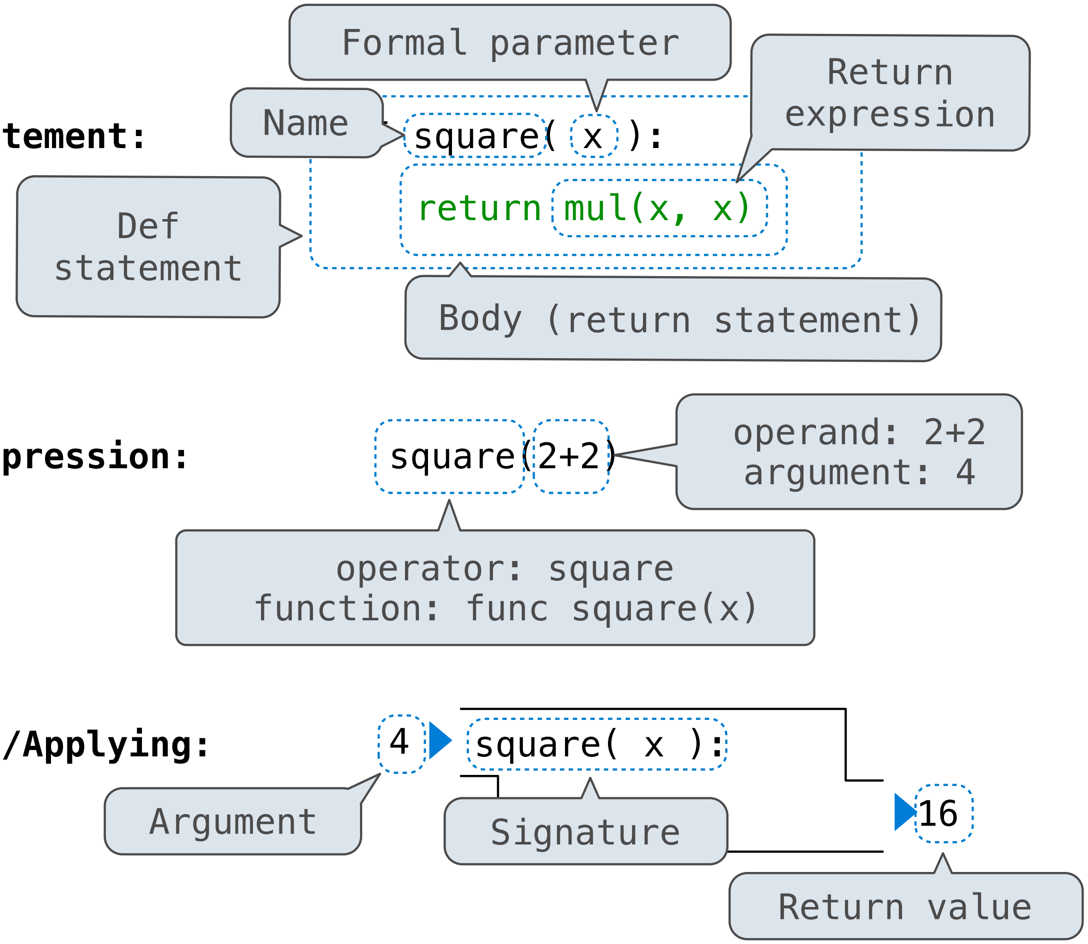
A new function is created!

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



Def statement:

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

Name

Formal parameter

Return expression

Def statement

Body (return statement)

Call expression:

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

operand: 2+2
argument: 4

operator: square
function: func square(x)

Calling/Applying:

```
4 ▶ square( x ): ▶ 16
```

Argument

Signature

Return value

What happens?

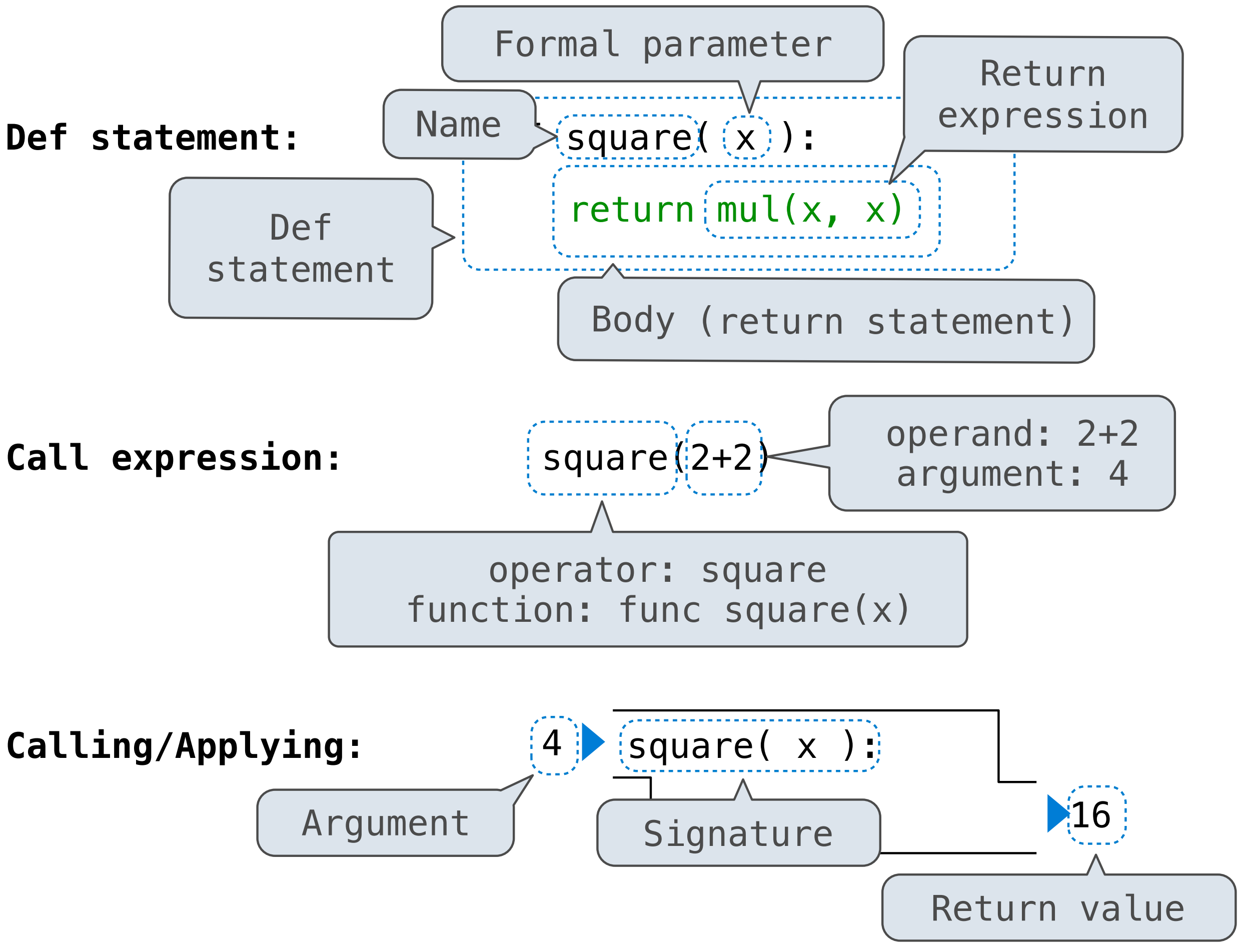
A new function is created!

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



Def statement:

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

Formal parameter

Name

Return expression

Def statement

Body (return statement)

Call expression:

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

operand: 2+2
argument: 4

operator: square
function: func square(x)

Calling/Applying:

```
4 ► square( x ): ► 16
```

Argument

Signature

Return value

What happens?

A new function is created!

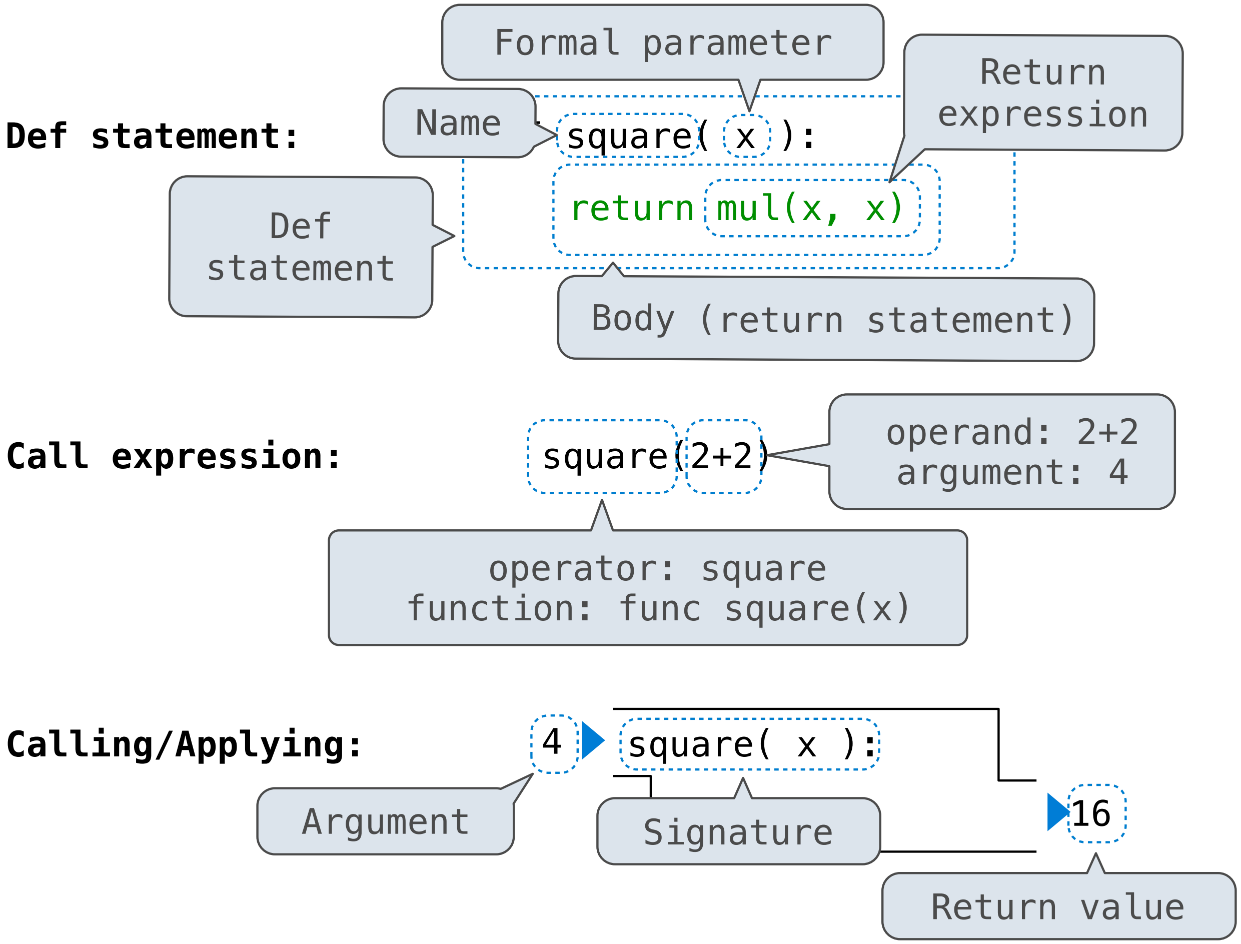
Name bound to that function
in the current frame

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



Def statement:

Formal parameter

Name

Return expression

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

Def statement

Body (return statement)

Call expression:

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

operand: 2+2
argument: 4

operator: square
function: func square(x)

Calling/Applying:

```
4 ▶ square( x ): ▶ 16
```

Argument

Signature

Return value

What happens?

A new function is created!

Name bound to that function in the current frame

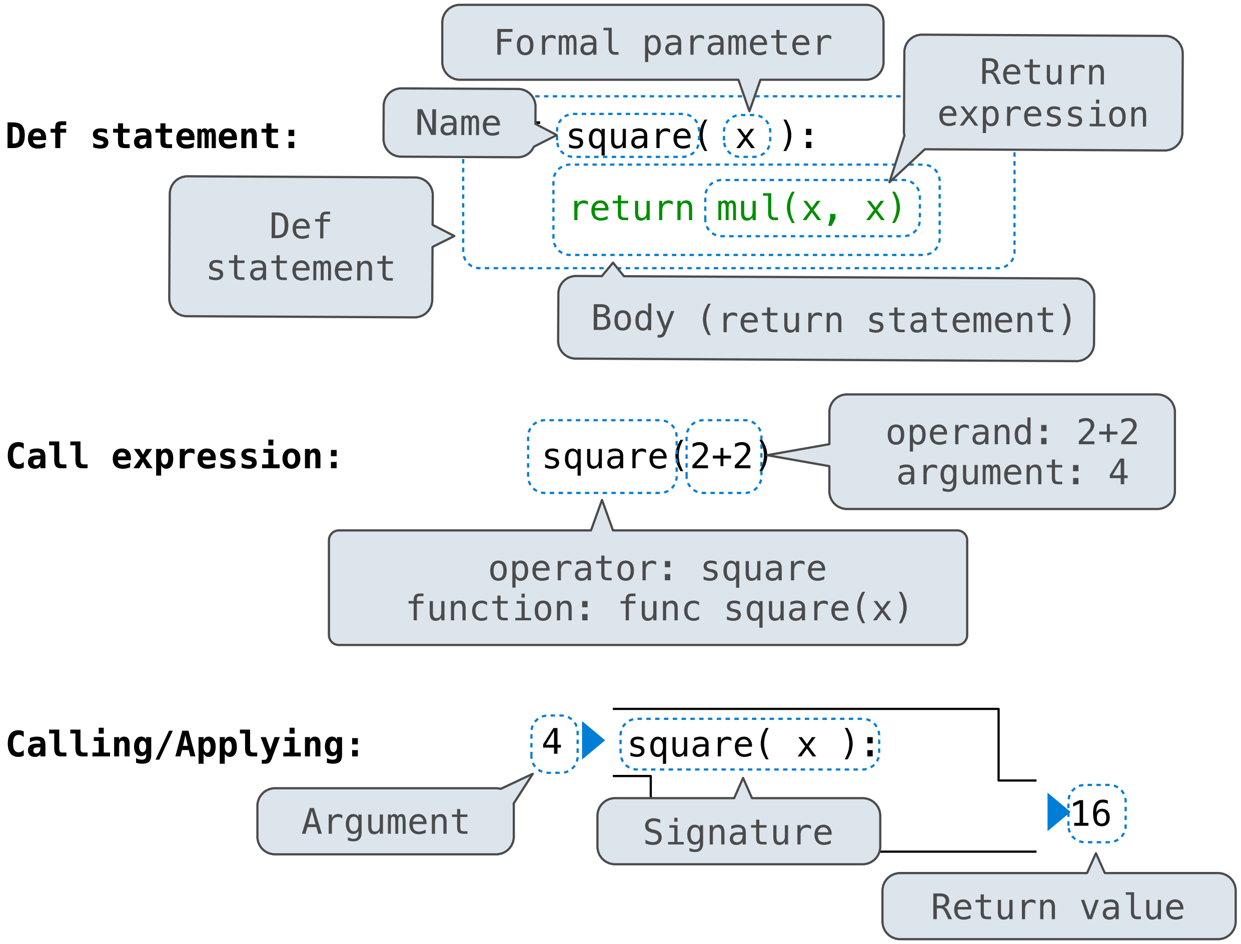
Operator & operands evaluated

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



Def statement:

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

Call expression:

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

Calling/Applying:

```
4 ▶ square( x ): ▶ 16
```

What happens?

A new function is created!

Name bound to that function
in the current frame

Operator & operands evaluated

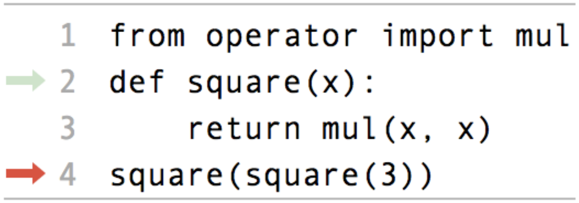
Function (value of operator)
called on arguments
(values of operands)

A new frame is created!

Parameters bound to arguments

Body is executed in that new environment

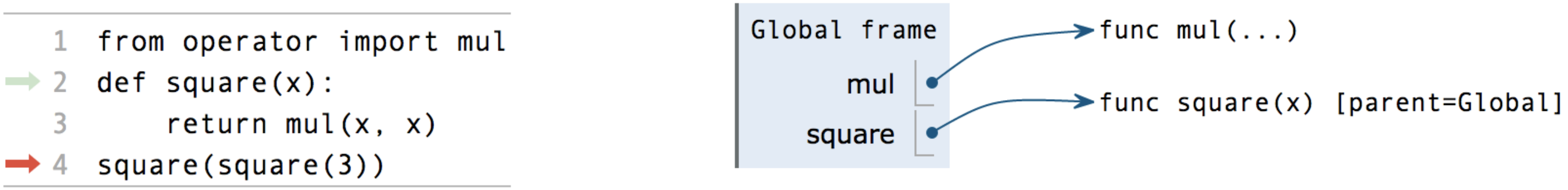
Multiple Environments in One Diagram!



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

Interactive Diagram

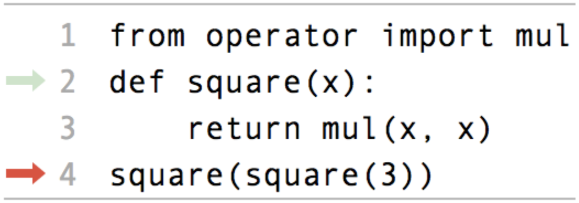
Multiple Environments in One Diagram!



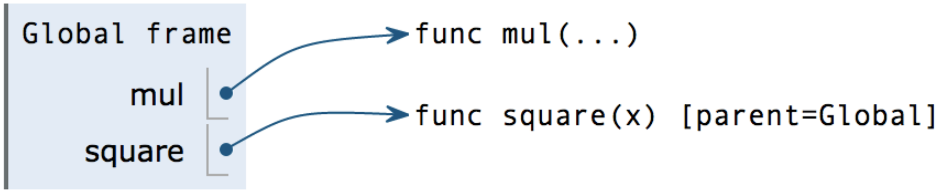
```
from operator import mul
def square(x):
    return mul(x, x)
square(square(3))
```

Interactive Diagram

Multiple Environments in One Diagram!



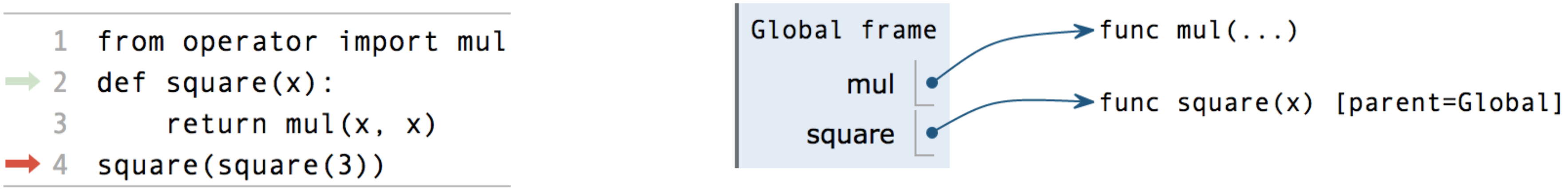
```
from operator import mul
def square(x):
    return mul(x, x)
square(square(3))
```



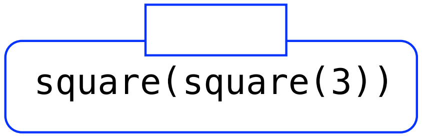
```
square(square(3))
```

Interactive Diagram

Multiple Environments in One Diagram!



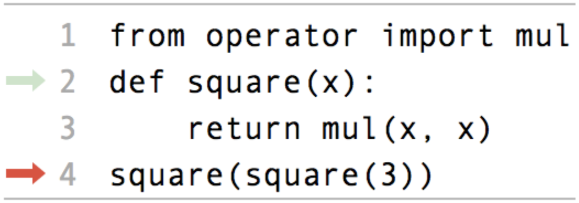
```
from operator import mul
def square(x):
    return mul(x, x)
square(square(3))
```



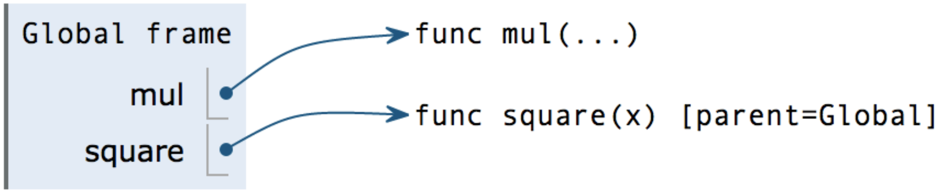
square(square(3))

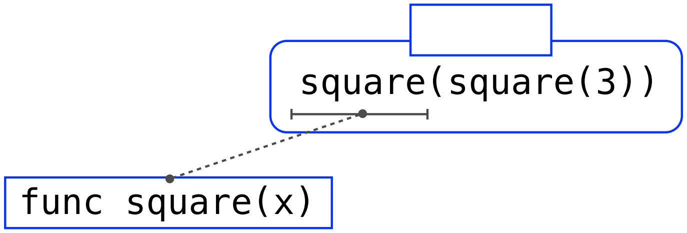
Interactive Diagram

Multiple Environments in One Diagram!



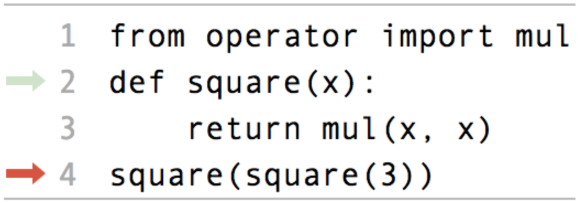
```
from operator import mul
def square(x):
    return mul(x, x)
square(square(3))
```



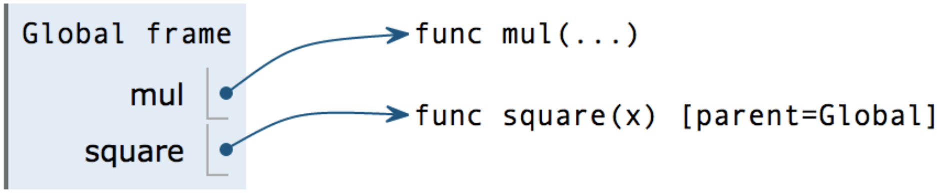


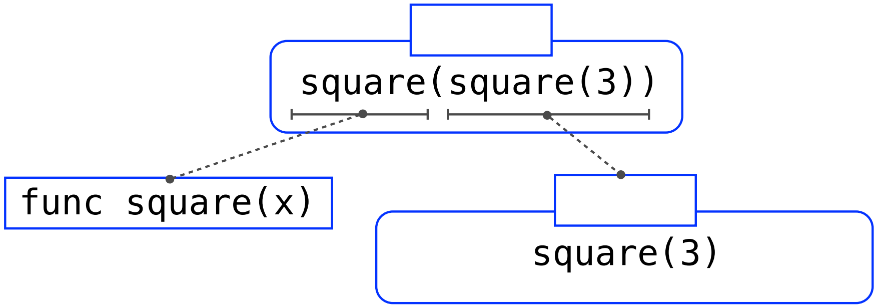
Interactive Diagram

Multiple Environments in One Diagram!



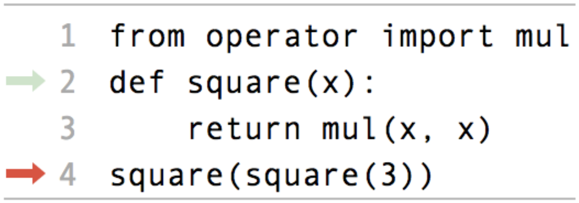
```python
from operator import mul
def square(x):
    return mul(x, x)
square(square(3))
```



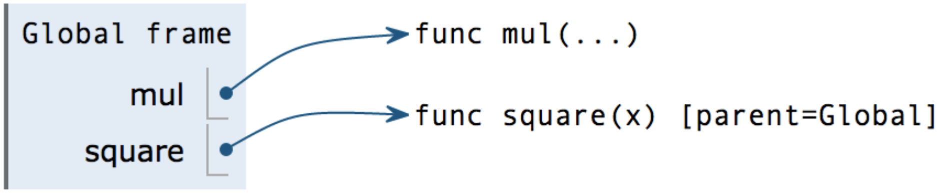


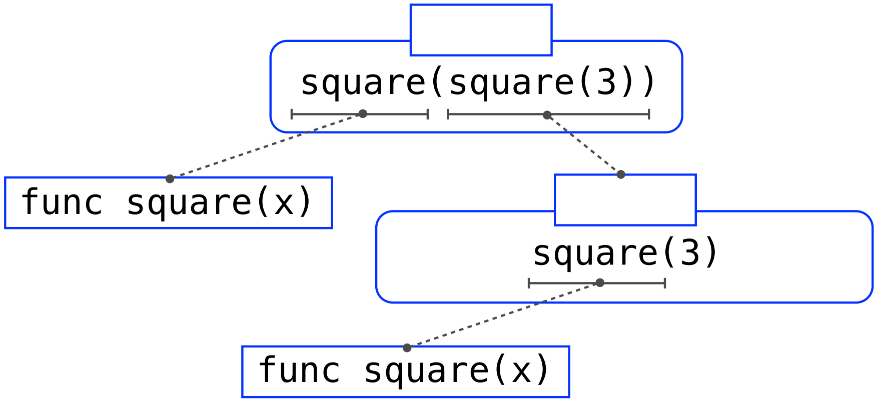
Interactive Diagram

Multiple Environments in One Diagram!



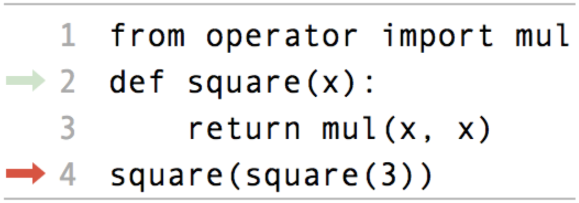
```
from operator import mul
def square(x):
    return mul(x, x)
square(square(3))
```



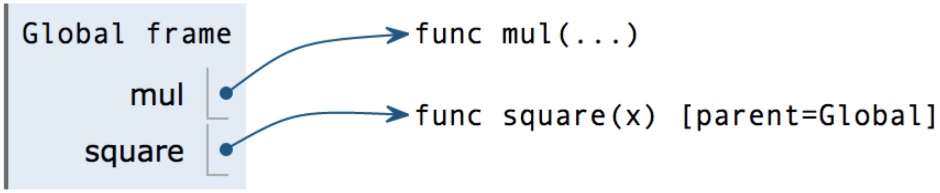


Interactive Diagram

Multiple Environments in One Diagram!



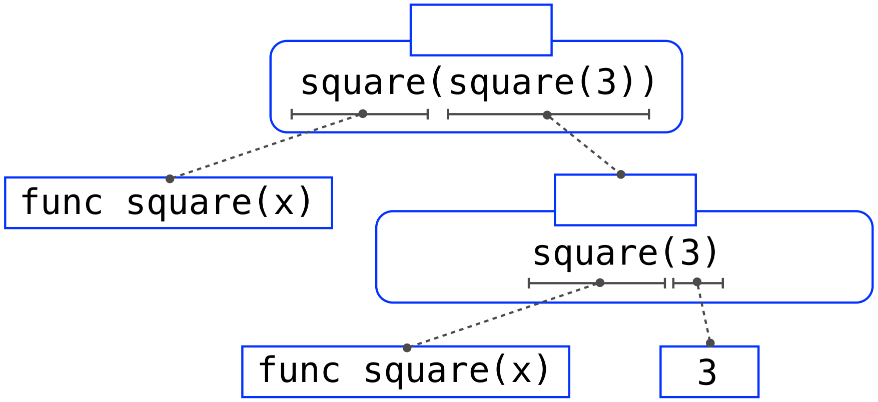
```
from operator import mul
def square(x):
    return mul(x, x)
square(square(3))
```



Global frame

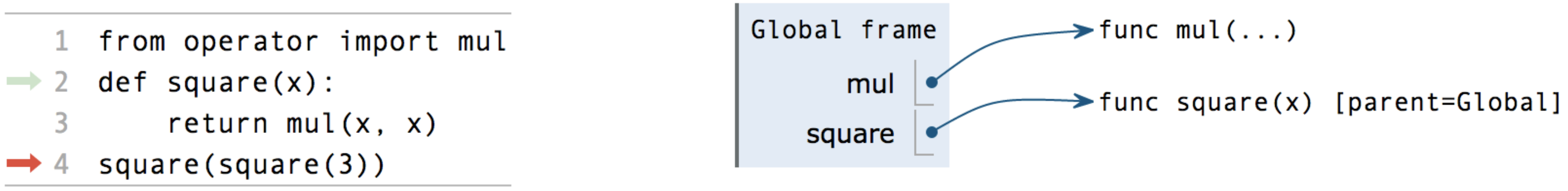
mul → func mul(...)

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

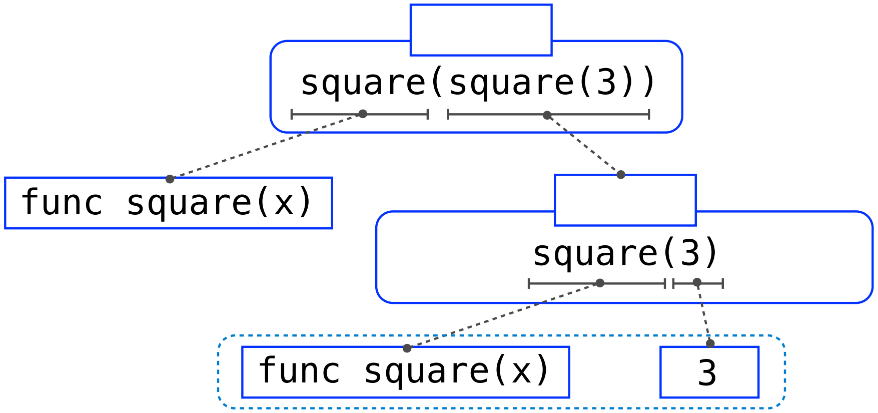


Interactive Diagram

Multiple Environments in One Diagram!

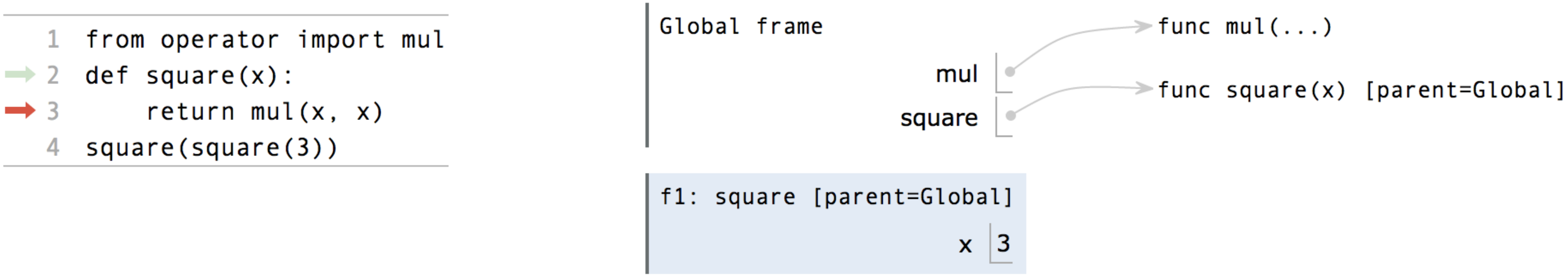


```
from operator import mul
def square(x):
    return mul(x, x)
square(square(3))
```



Interactive Diagram

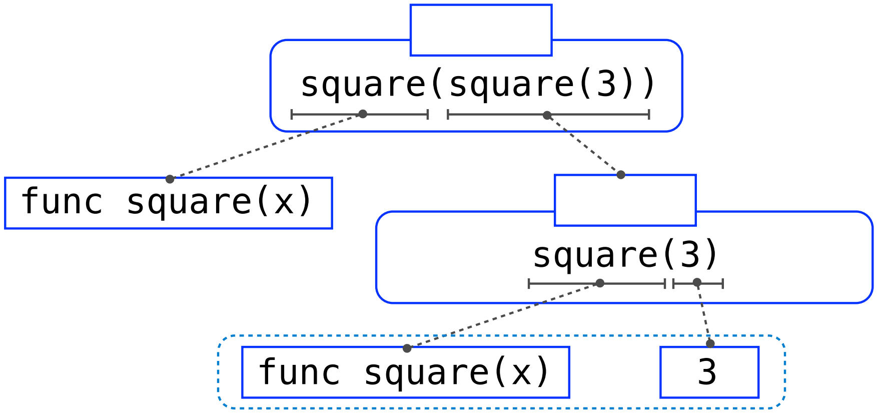
Multiple Environments in One Diagram!



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

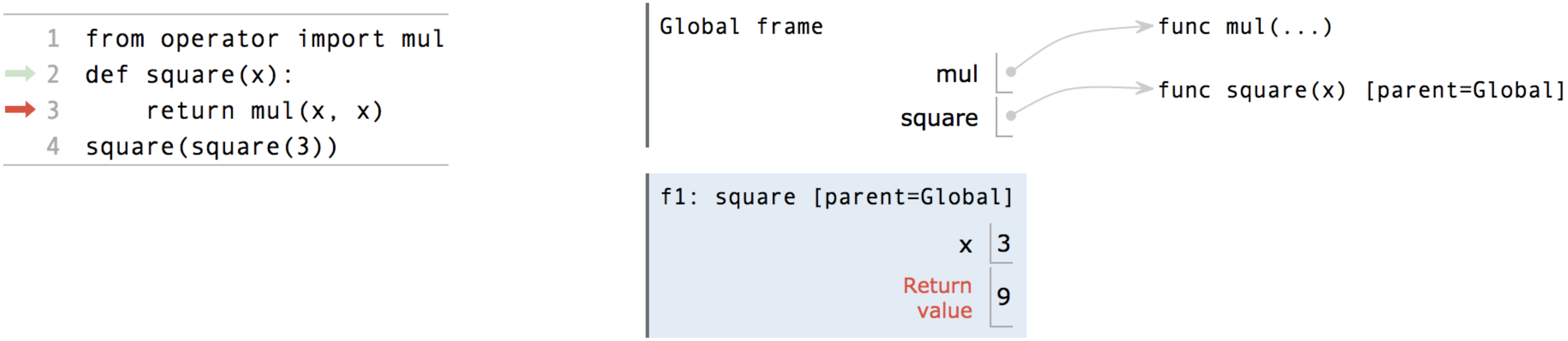
```
Global frame
                  mul  •──> func mul(...)
               square  •──> func square(x) [parent=Global]

f1: square [parent=Global]
                    x  3
```



Interactive Diagram

Multiple Environments in One Diagram!



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

Global frame

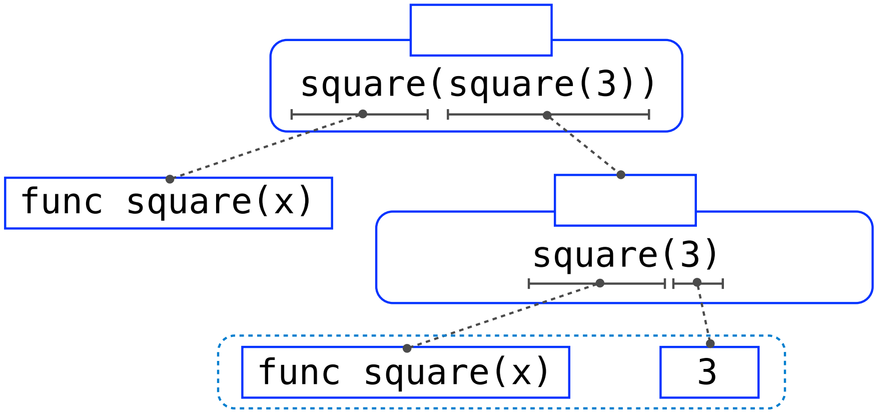
mul → func mul(...)

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

f1: square [parent=Global]

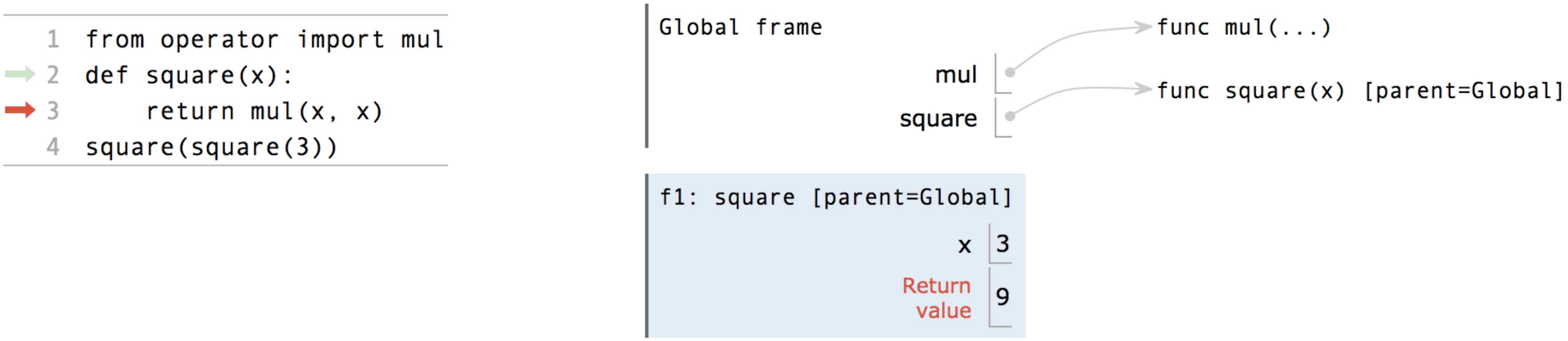
x 3

Return value 9

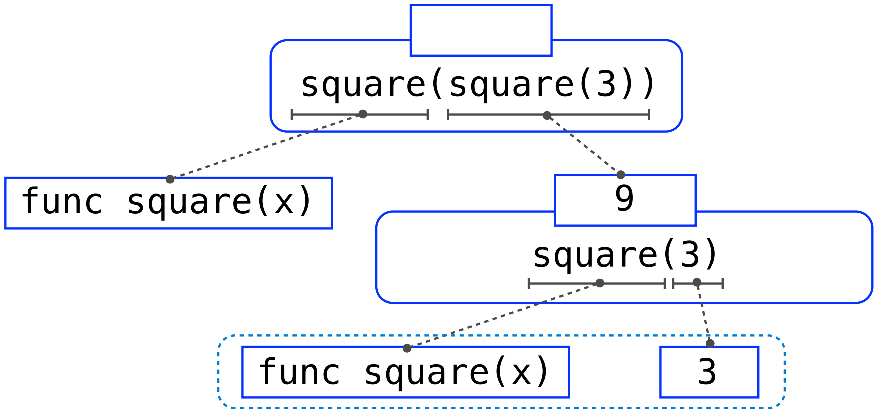


Interactive Diagram

Multiple Environments in One Diagram!



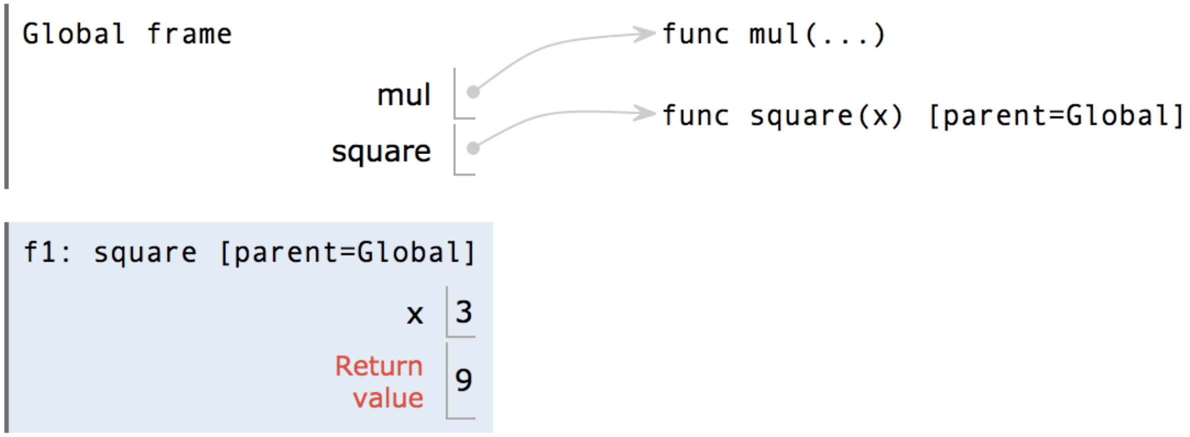
```
from operator import mul
def square(x):
    return mul(x, x)
square(square(3))
```

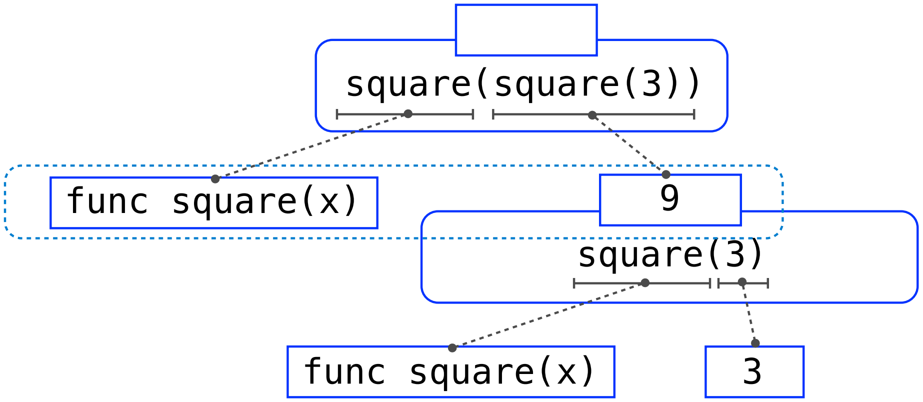


Interactive Diagram

Multiple Environments in One Diagram!

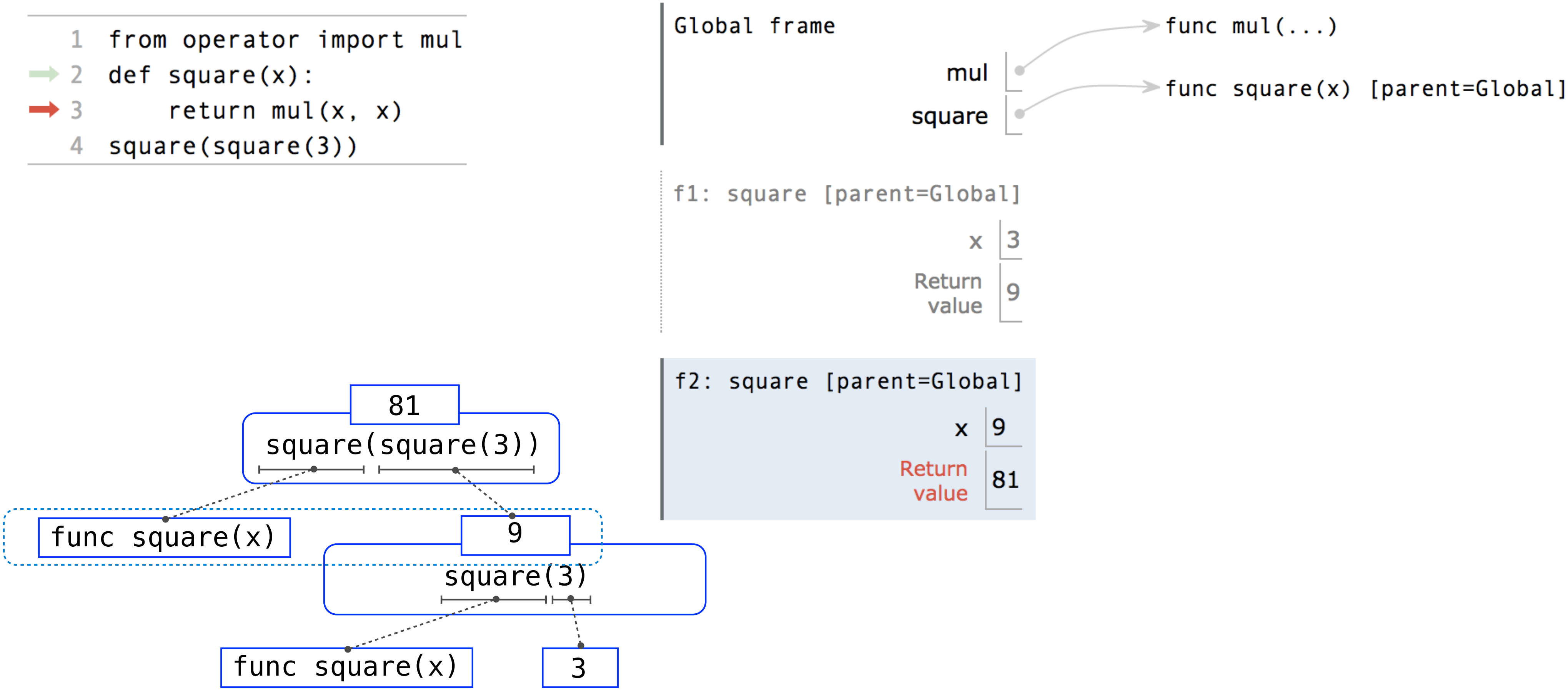
```
from operator import mul
def square(x):
    return mul(x, x)
square(square(3))
```





Interactive Diagram

Multiple Environments in One Diagram!



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

Global frame

mul → func mul(...)

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

f1: square [parent=Global]

x 3

Return value 9

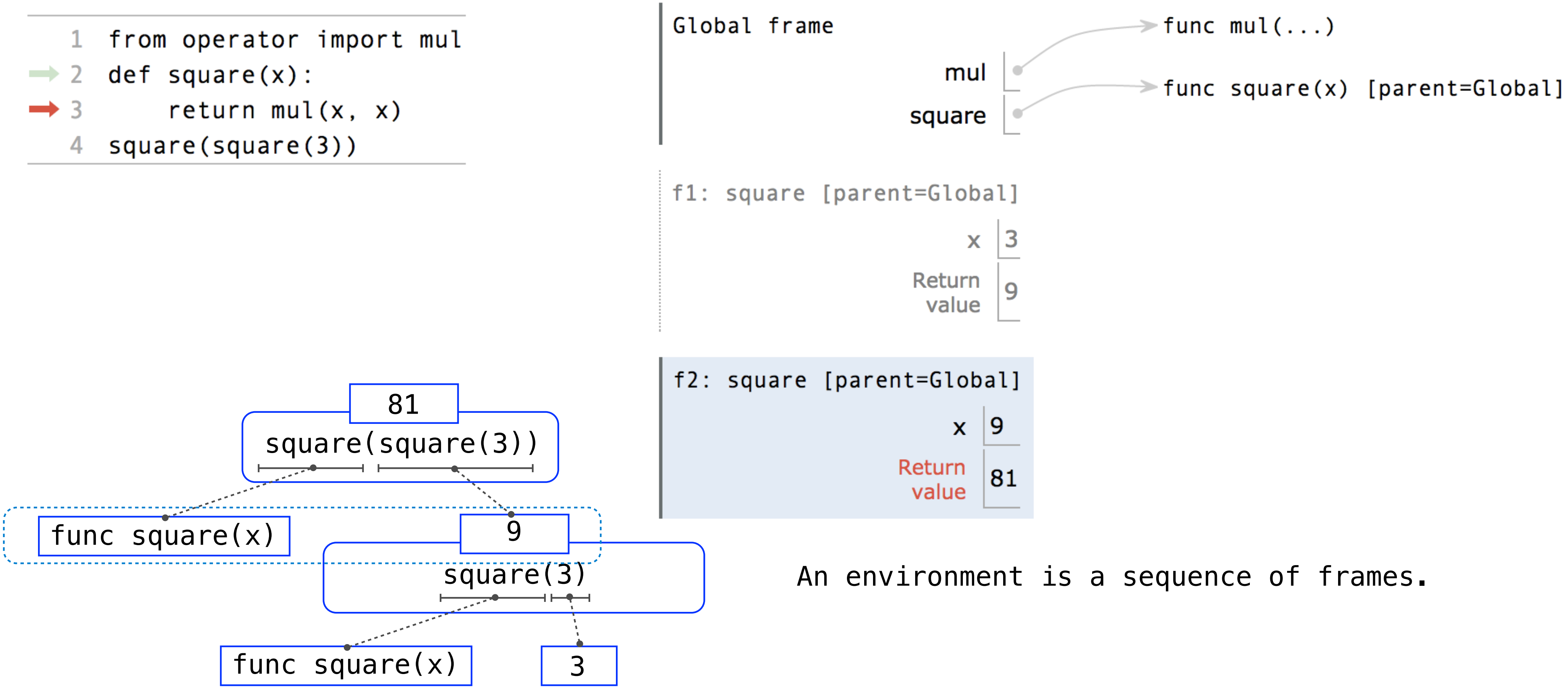
f2: square [parent=Global]

x 9

Return value 81

Interactive Diagram

Multiple Environments in One Diagram!



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

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

f1: square [parent=Global]
- x: 3
- Return value: 9

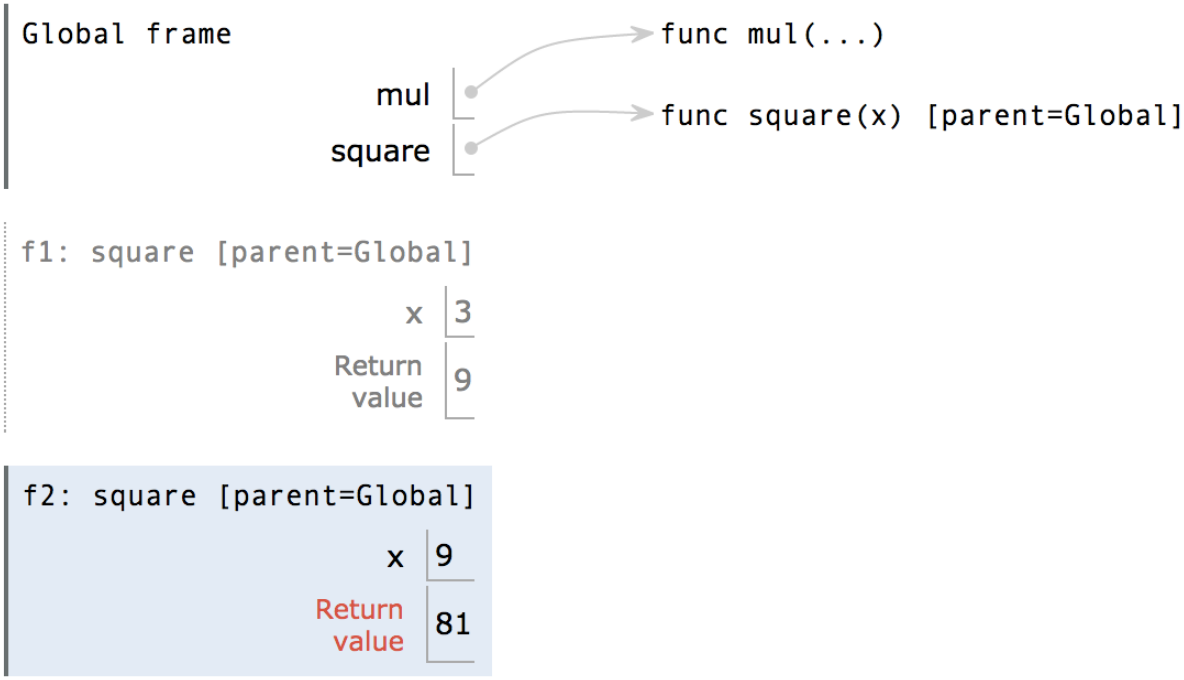
f2: square [parent=Global]
- x: 9
- Return value: 81

An environment is a sequence of frames.

Interactive Diagram

Multiple Environments in One Diagram!

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



Global frame

mul → func mul(...)

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

f1: square [parent=Global]

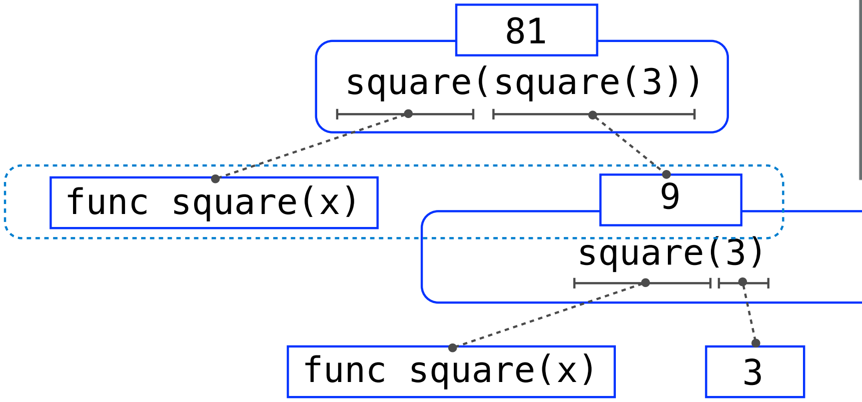
x 3

Return value 9

f2: square [parent=Global]

x 9

Return value 81

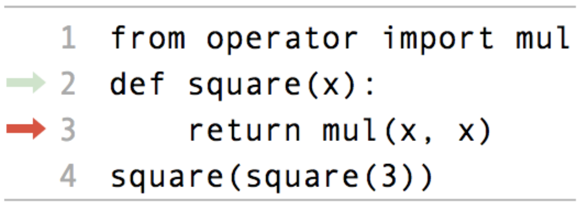


An environment is a sequence of frames.

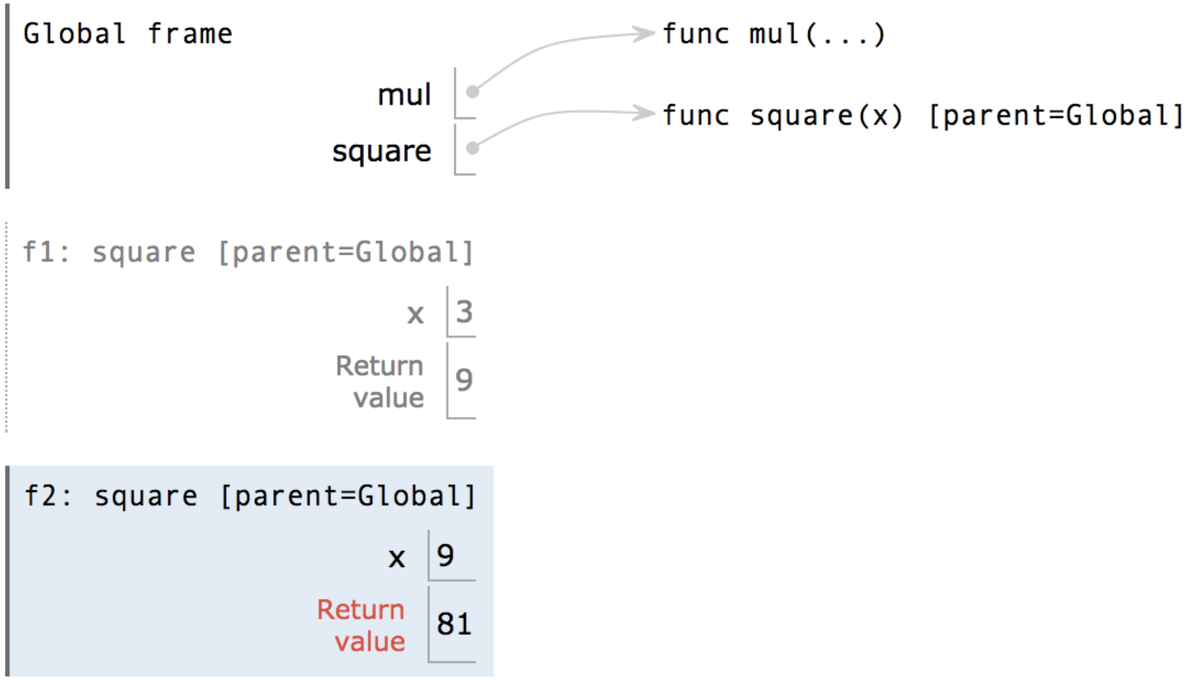
- The global frame alone
- A local, then the global frame

Interactive Diagram

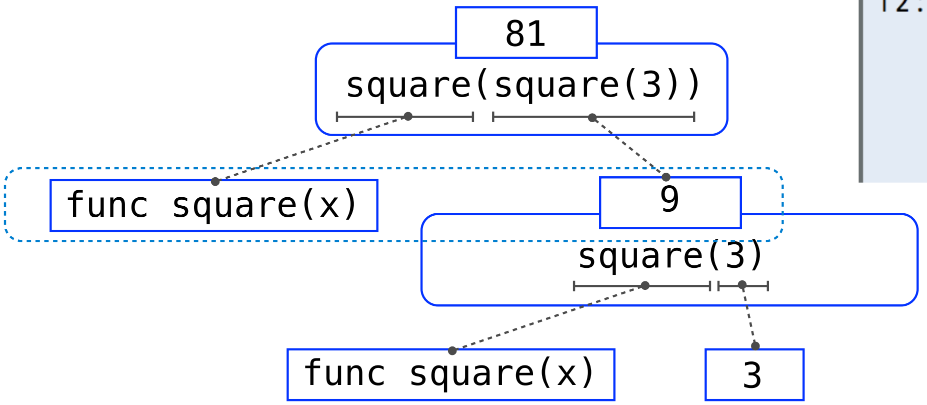
Multiple Environments in One Diagram!



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



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

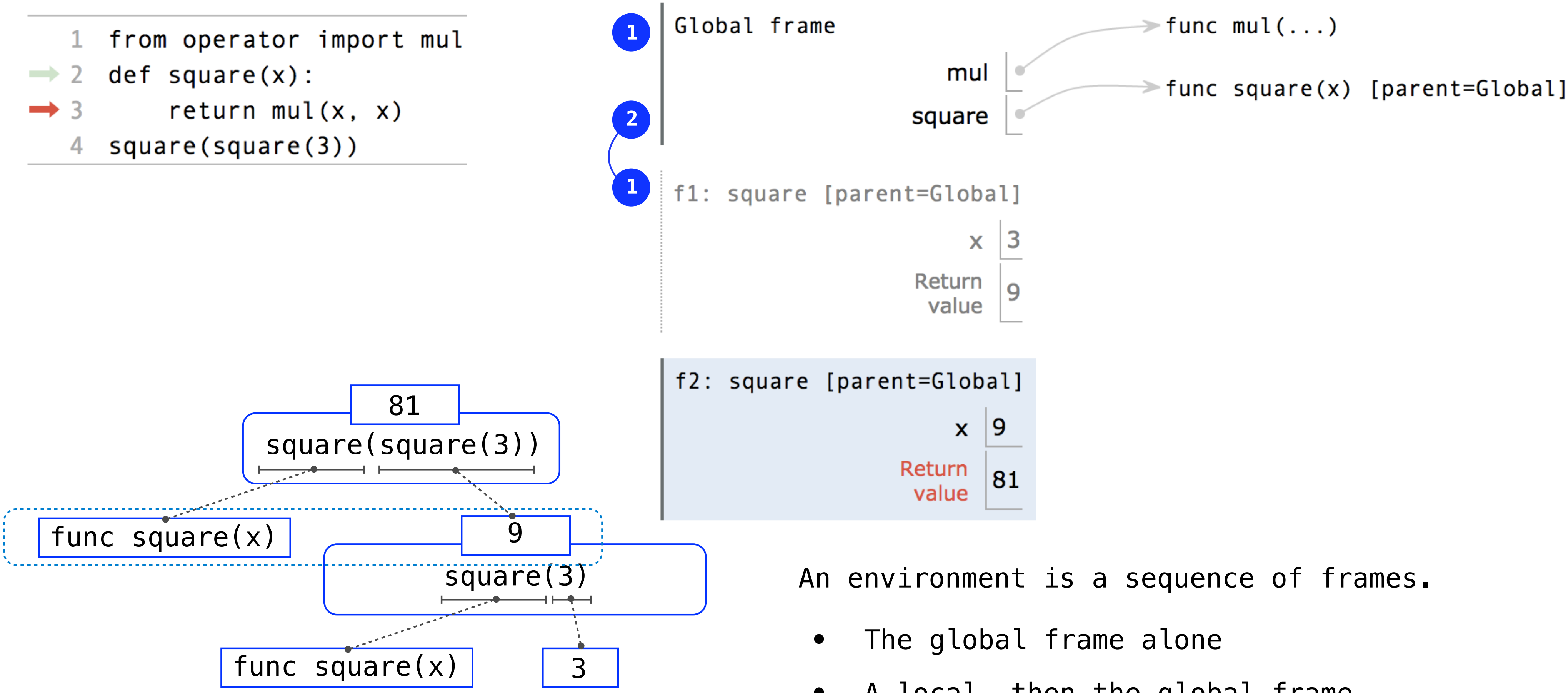


An environment is a sequence of frames.

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

Multiple Environments in One Diagram!

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

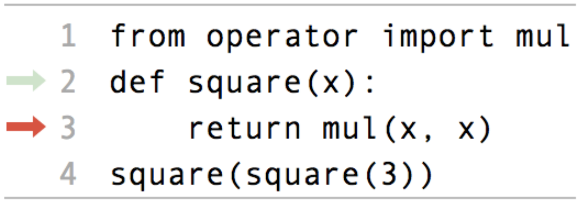


An environment is a sequence of frames.

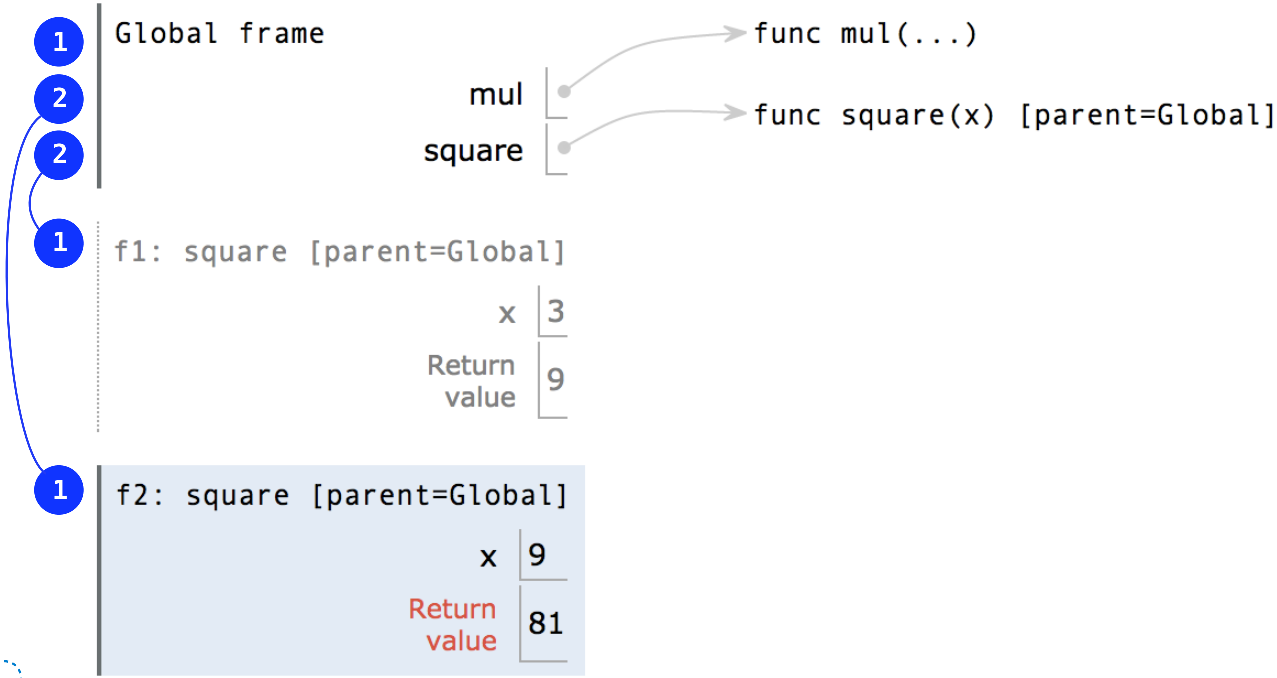
- The global frame alone
- A local, then the global frame

Interactive Diagram

Multiple Environments in One Diagram!



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



Global frame

mul → func mul(...)

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

f1: square [parent=Global]

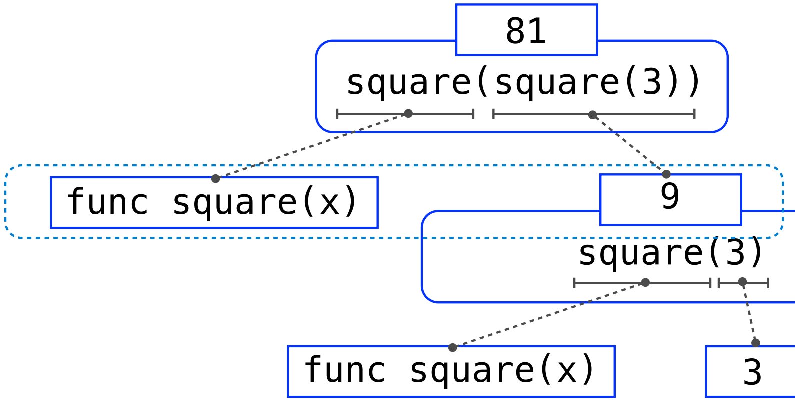
x 3

Return value 9

f2: square [parent=Global]

x 9

Return value 81

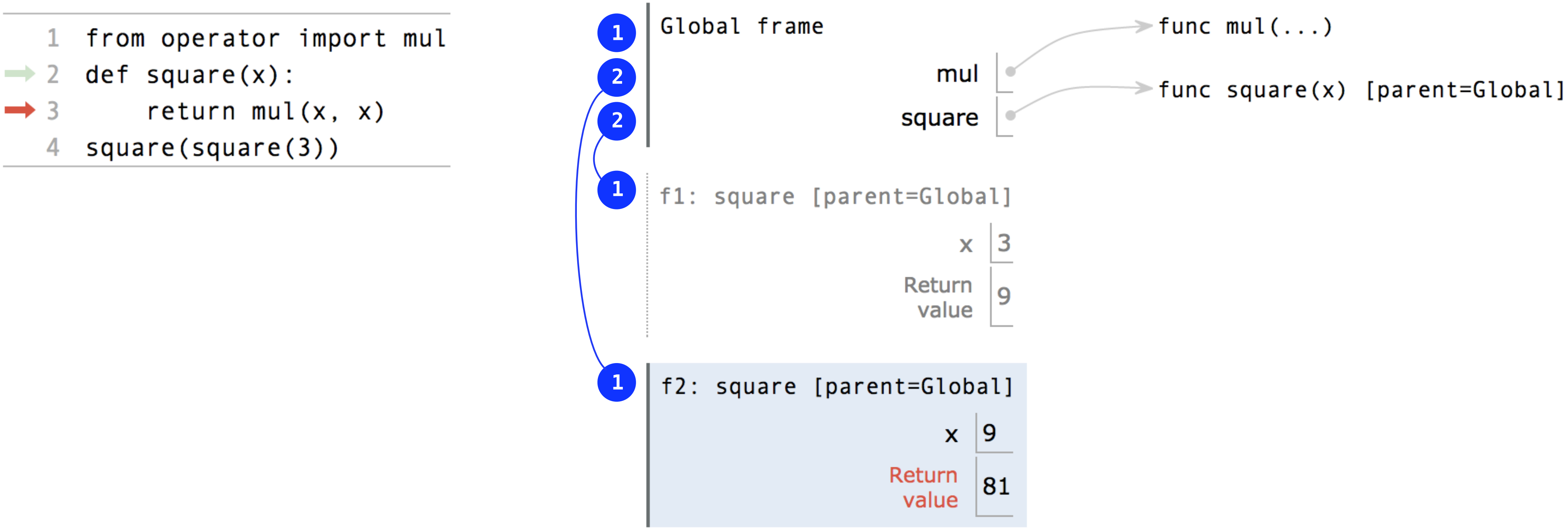


An environment is a sequence of frames.

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

Interactive Diagram

Names Have No Meaning Without Environments



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

Global frame

mul → func mul(...)

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

f1: square [parent=Global]

x 3

Return value 9

f2: square [parent=Global]

x 9

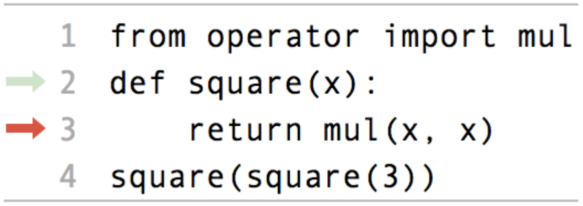
Return value 81

An environment is a sequence of frames.

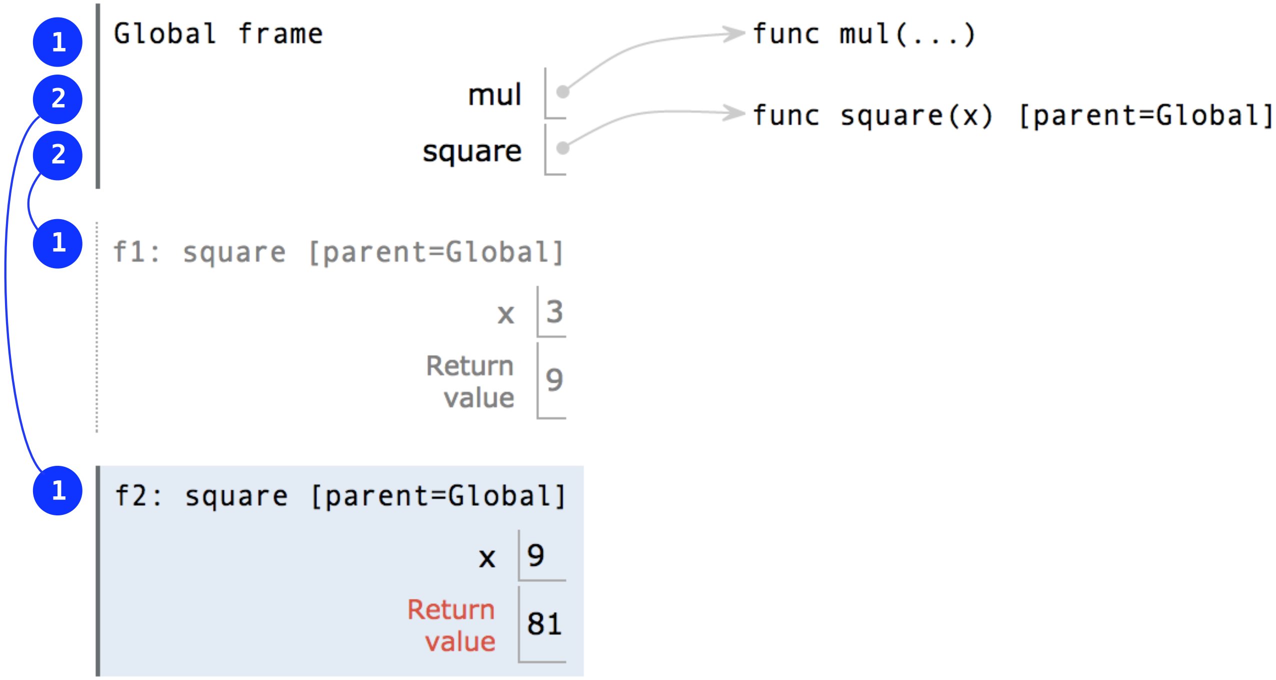
- The global frame alone
- A local, then the global frame

Interactive Diagram

Names Have No Meaning Without Environments



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



Global frame

mul → func mul(...)

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

f1: square [parent=Global]

x 3

Return value 9

f2: square [parent=Global]

x 9

Return value 81

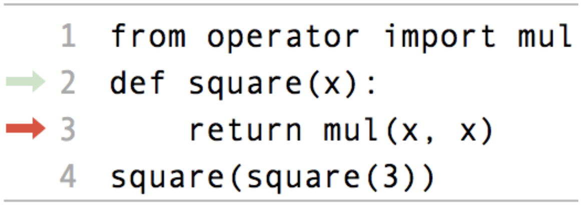
Every expression is evaluated in the context of an environment.

An environment is a sequence of frames.

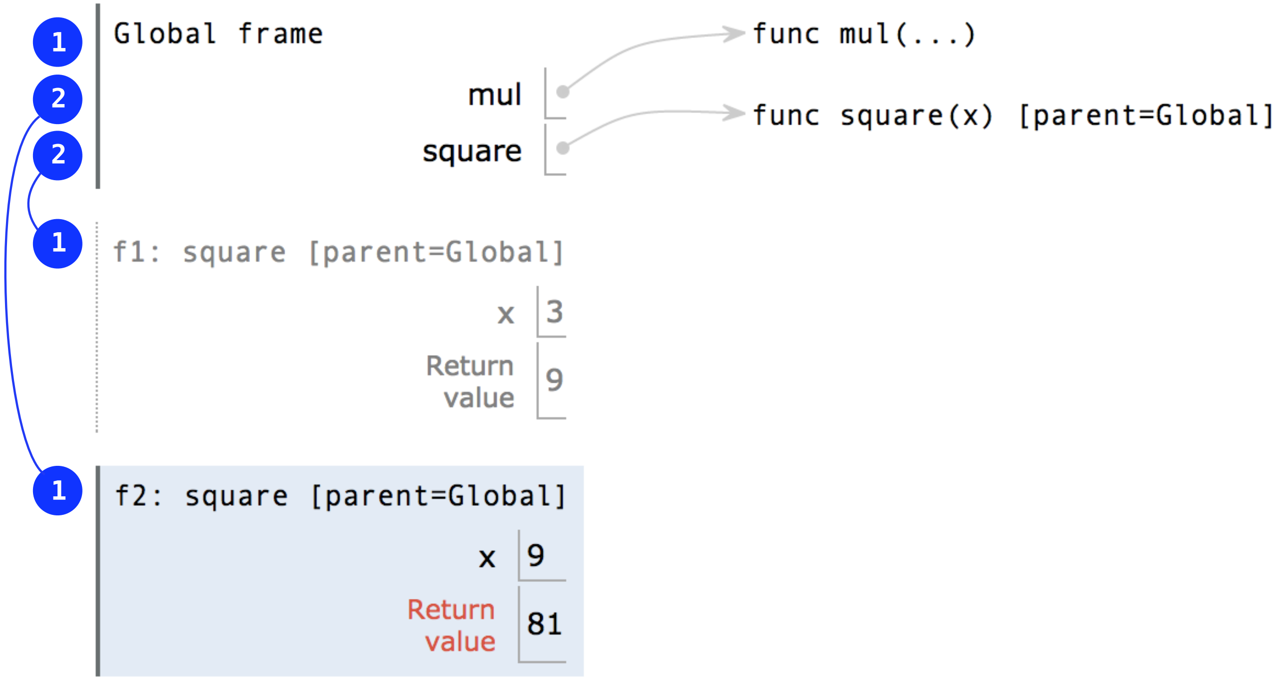
- The global frame alone
- A local, then the global frame

Interactive Diagram

Names Have No Meaning Without Environments



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



Global frame

mul → func mul(...)

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

f1: square [parent=Global]

x 3

Return value 9

f2: square [parent=Global]

x 9

Return value 81

Every expression is evaluated in the context of an environment.

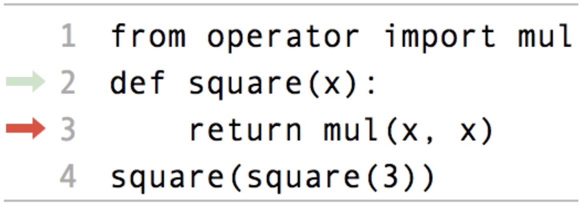
A name evaluates to the value bound to that name in the earliest frame of the current environment in which that name is found.

An environment is a sequence of frames.

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

Interactive Diagram

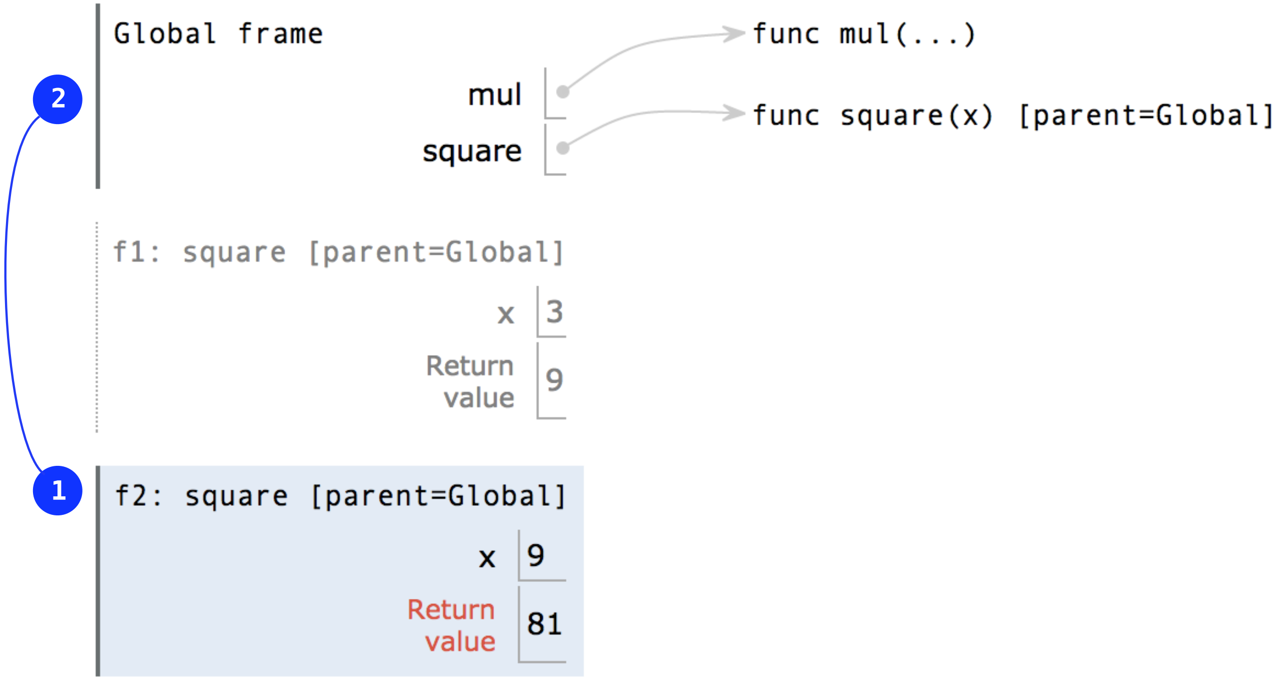
Names Have No Meaning Without Environments



```
from operator import mul
def square(x):
    return mul(x, x)
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
square
func mul(...)
func square(x) [parent=Global]
f1: square [parent=Global]
x
3
Return value
9
f2: square [parent=Global]
x
9
Return value
81
1
2

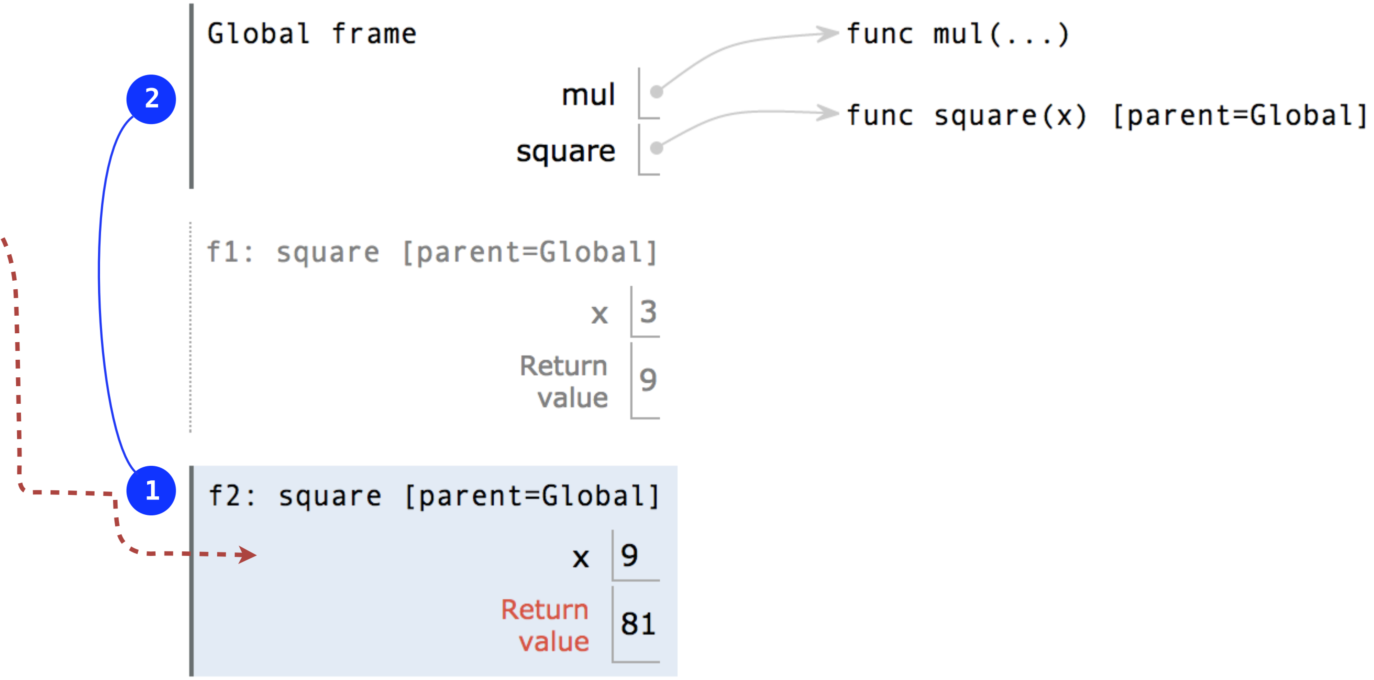
An environment is a sequence of frames.

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

Interactive Diagram

Names Have No Meaning Without Environments

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


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

Every expression is evaluated in the context of an environment.

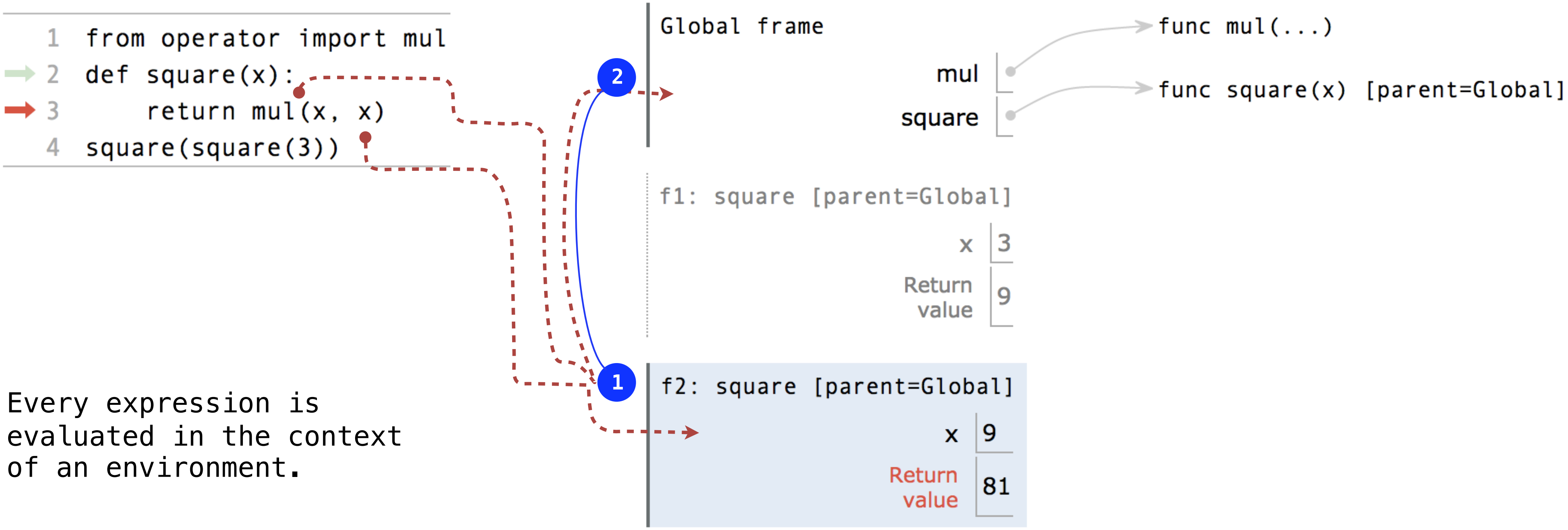
A name evaluates to the value bound to that name in the earliest frame of the current environment in which that name is found.

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



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

Global frame

mul → func mul(...)

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

f1: square [parent=Global]

x 3

Return value 9

f2: square [parent=Global]

x 9

Return value 81

Every expression is evaluated in the context of an environment.

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

An environment is a sequence of frames.

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

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

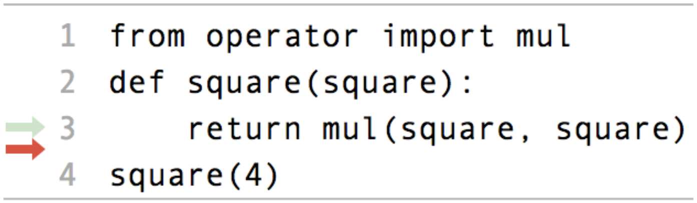
Every expression is evaluated in the context of an environment.

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



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

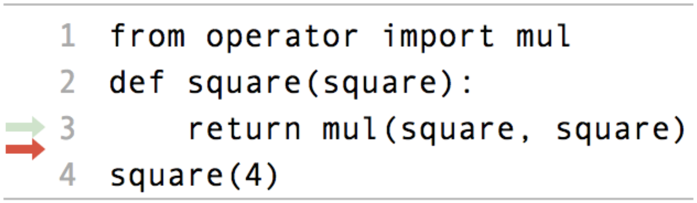
Every expression is evaluated in the context of an environment.

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

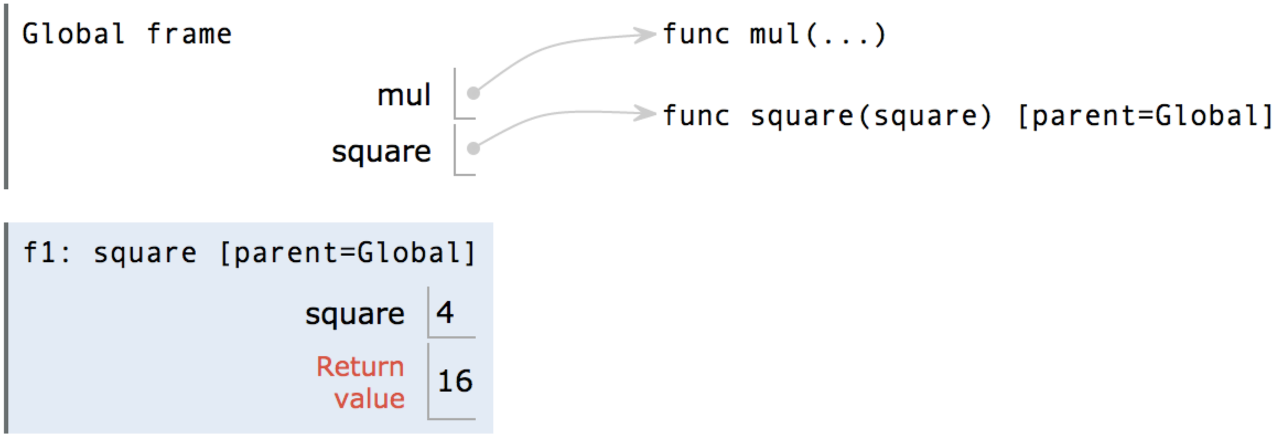
Interactive Diagram

Names Have Different Meanings in Different Environments

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



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


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

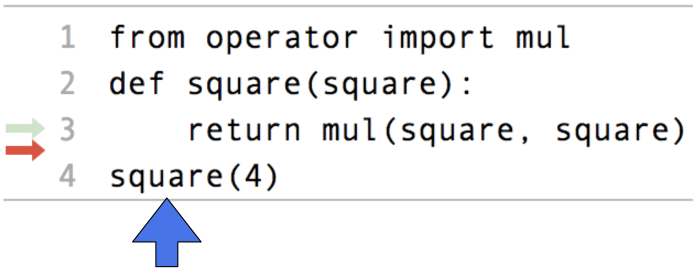
Every expression is evaluated in the context of an environment.

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

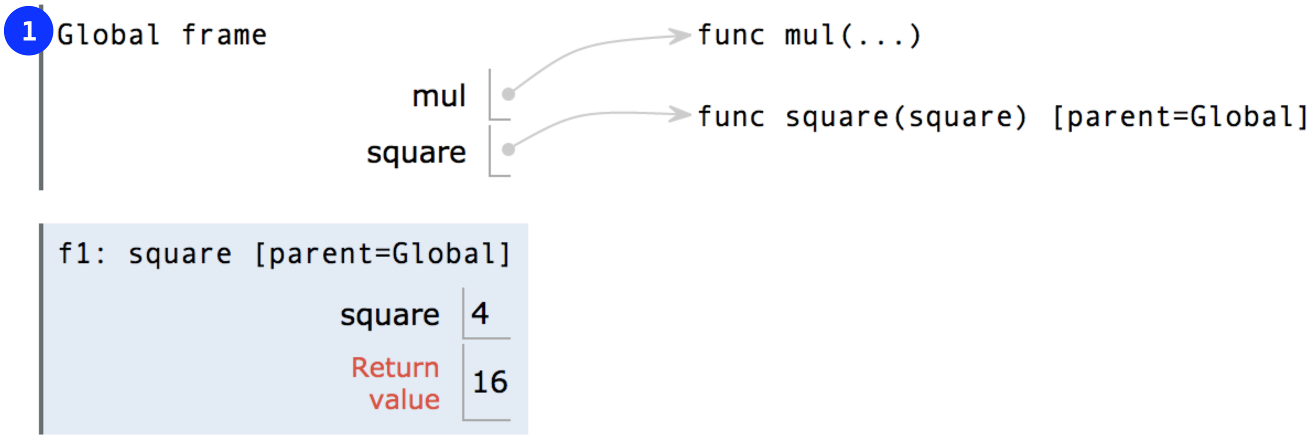
Interactive Diagram

Names Have Different Meanings in Different Environments

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



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


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

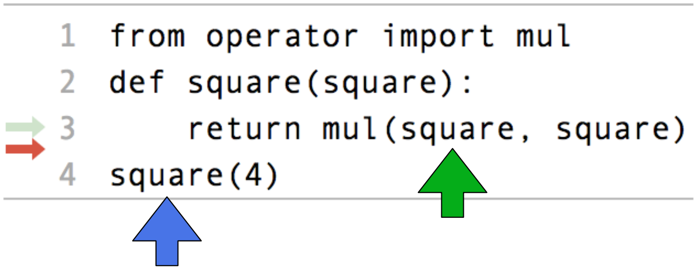
Every expression is evaluated in the context of an environment.

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

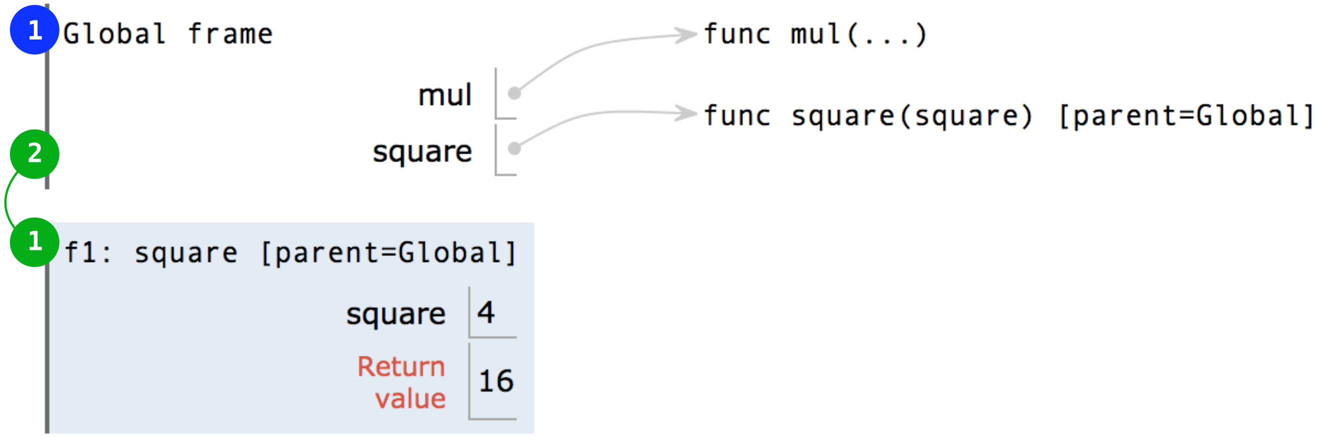
Interactive Diagram

Names Have Different Meanings in Different Environments

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



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



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

Every expression is
evaluated in the context
of an environment.

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

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

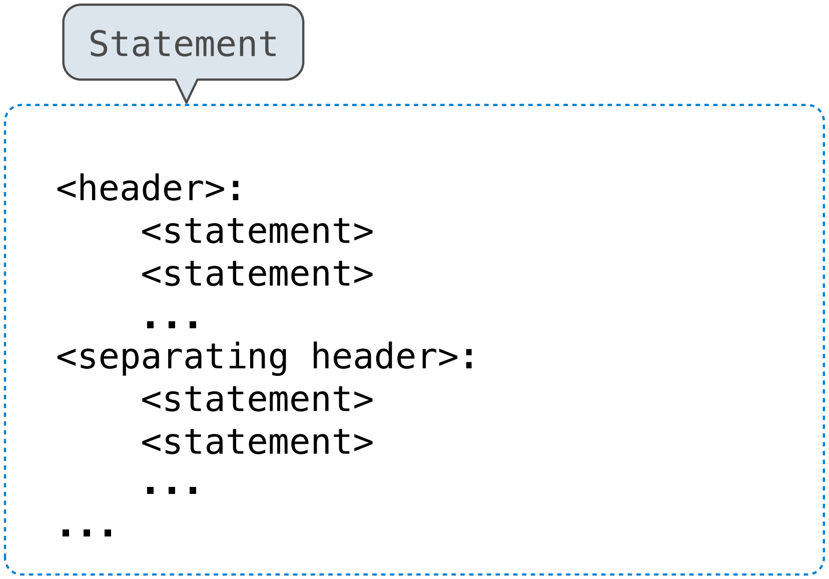
Compound statements:

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

Statements

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

Compound statements:

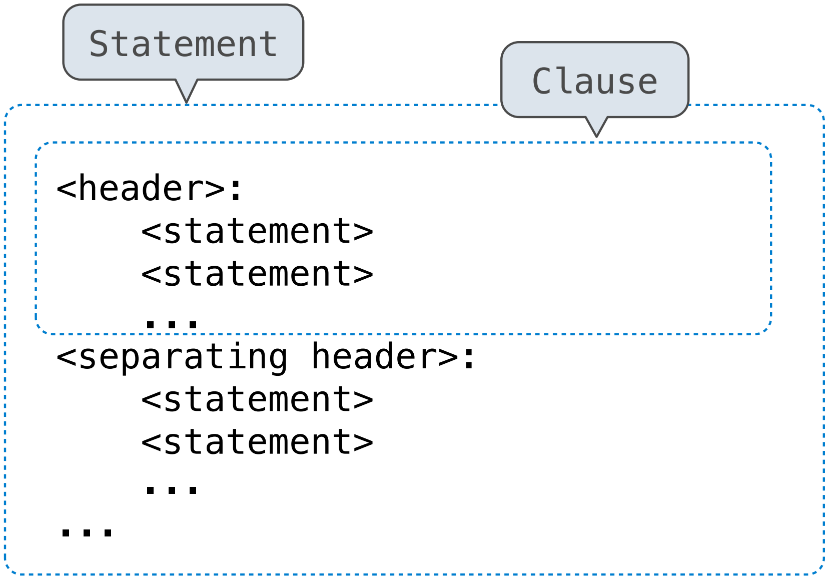


```
<header>:
    <statement>
    <statement>
    ...
<separating header>:
    <statement>
    <statement>
    ...
...
```

Statements

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

Compound statements:

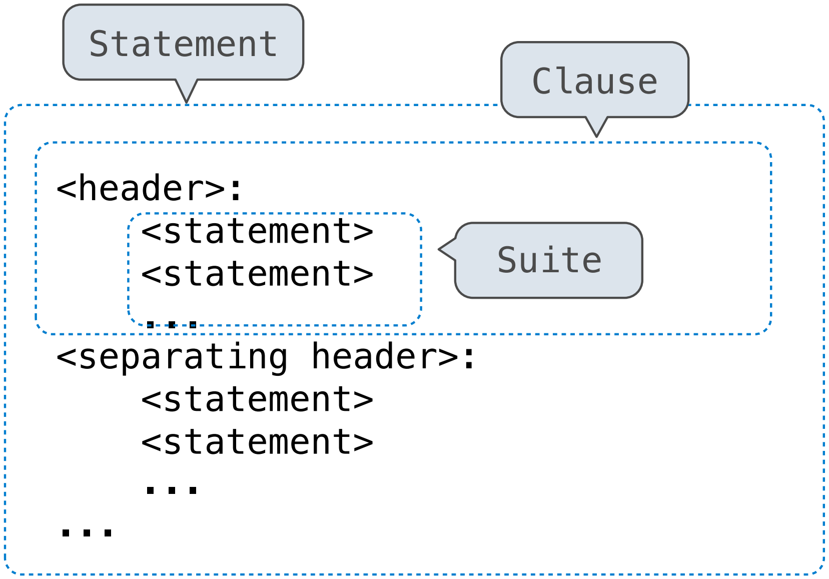


```
<header>:
    <statement>
    <statement>
    ...
<separating header>:
    <statement>
    <statement>
    ...
...
```

Statements

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

Compound statements:

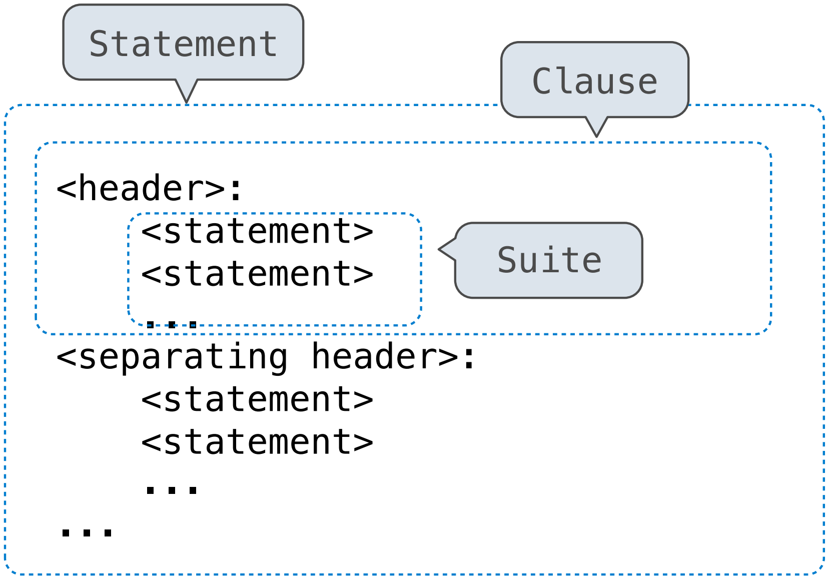


```
<header>:
    <statement>
    <statement>
    ...
<separating header>:
    <statement>
    <statement>
    ...
...
```

Statements

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

Compound statements:



Statement

Clause

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

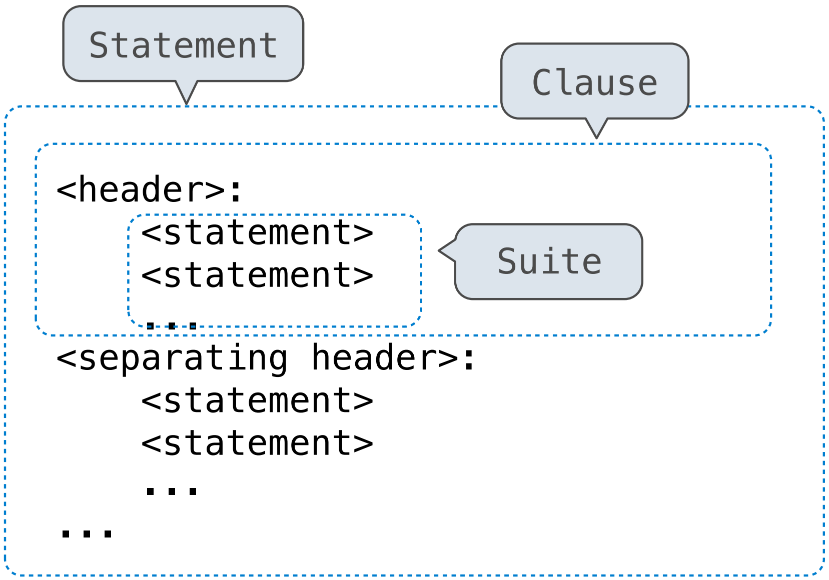
Suite

The first header determines a statement's type

Statements

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

Compound statements:



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

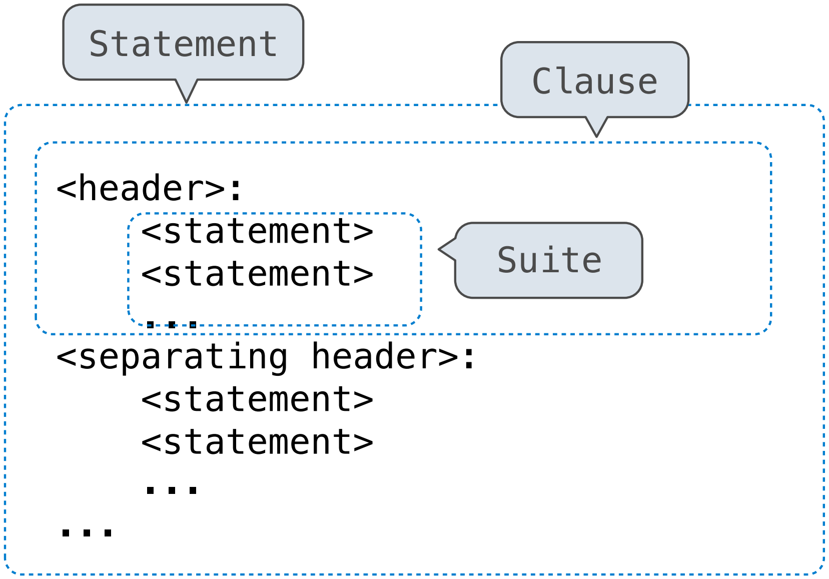
The first header determines a statement’s type

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

Statements

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

Compound statements:



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

The first header determines a statement's type

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

def statements are compound statements

Compound Statements

Compound statements:

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

Suite

Compound Statements

Compound statements:

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

Suite

A suite is a sequence of statements

Compound Statements

Compound statements:

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

Suite

A suite is a sequence of statements

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

Compound Statements

Compound statements:

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

Suite

A suite is a sequence of statements

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

Execution Rule for a sequence of statements:

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

Conditional Statements

```
(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)

```
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

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:

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.

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)

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.

Syntax Tips:

1. Always starts with "if" clause.

2. Zero or more "elif" clauses.

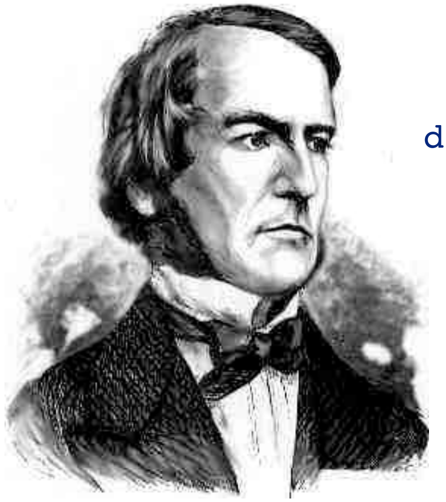
3. Zero or one "else" clause,
   always at the end.

Boolean Contexts

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

George Boole

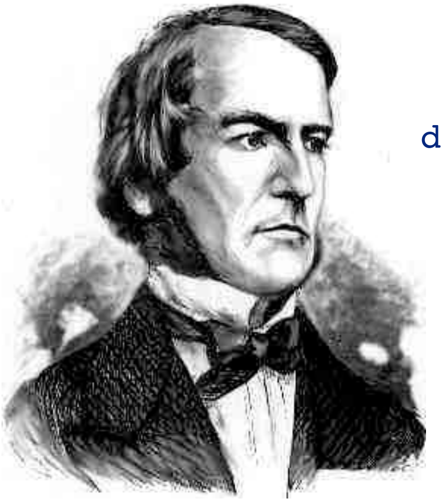
Boolean Contexts



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

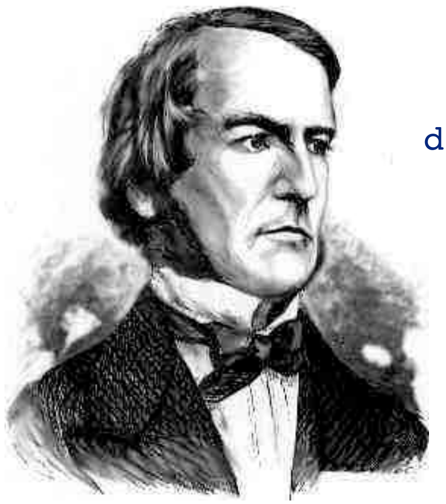


George Boole

```
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



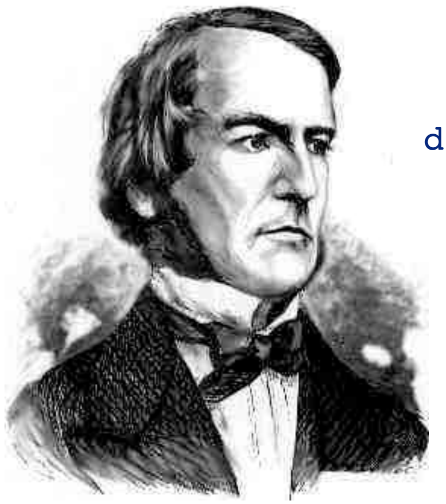
George Boole

```
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



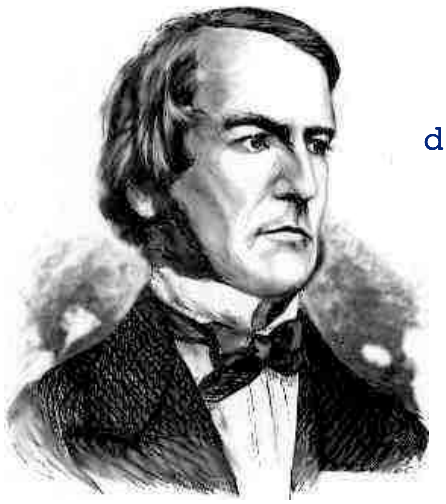
George Boole

```
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

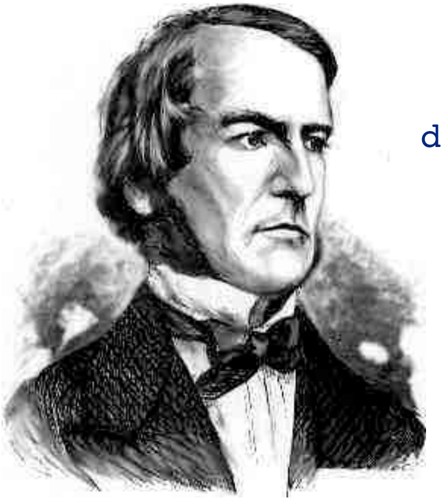
```
def absolute_value(x):
    """Return the absolute value of x."""
    if x < 0:
        return -x
    elif x == 0:
        return 0
    else:
        return x
```

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!

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

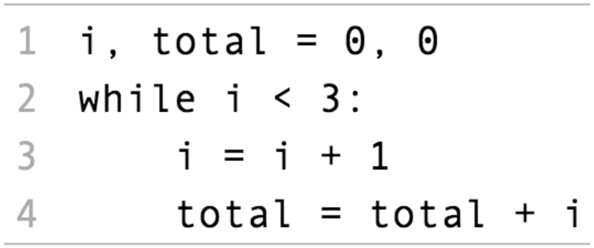
Iteration

While Statements

(Demo)

While Statements

(Demo)



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

While Statements

(Demo)

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

George Boole

(Demo)

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

George Boole

(Demo)

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

George Boole

(Demo)

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

Execution Rule for While Statements:

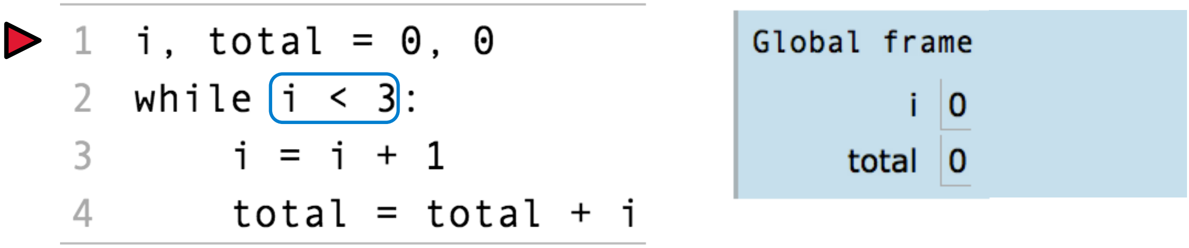
1. Evaluate the header's expression.

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

While Statements

George Boole

(Demo)



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

Global frame

i	0
total	0

Execution Rule for While Statements:

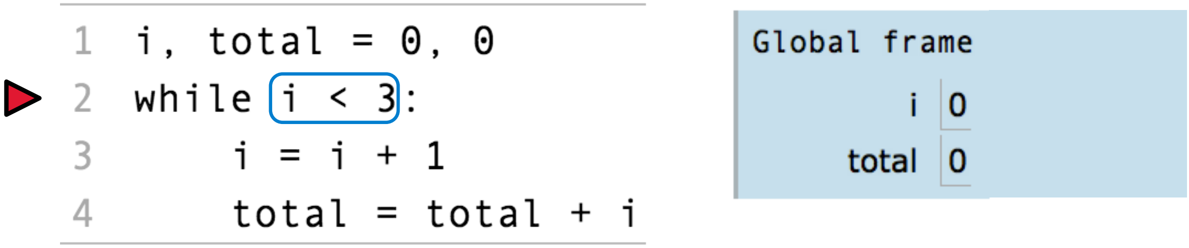
1. Evaluate the header’s expression.

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

While Statements

George Boole

(Demo)



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

Global frame

i 0

total 0

Execution Rule for While Statements:

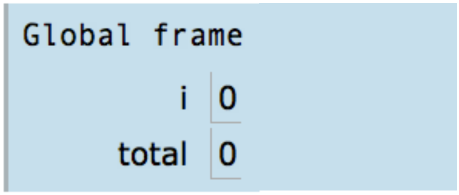
1. Evaluate the header's expression.
2. If it is a true value, execute the (whole) suite, then return to step 1.

While Statements

George Boole

(Demo)

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



Global frame

i	0
total	0

Execution Rule for While Statements:

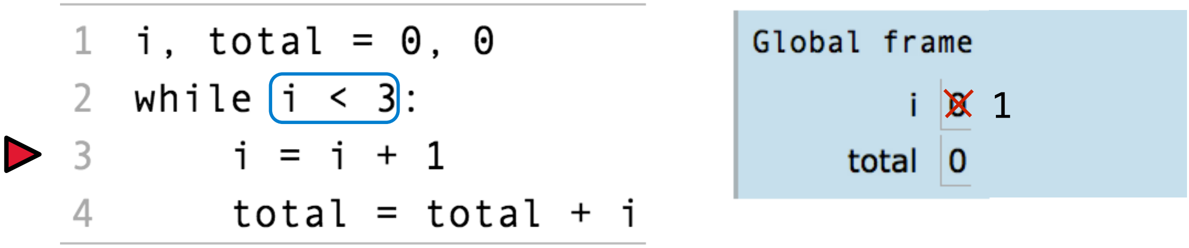
1. Evaluate the header’s expression.

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

While Statements

(Demo)

George Boole



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

Execution Rule for While Statements:

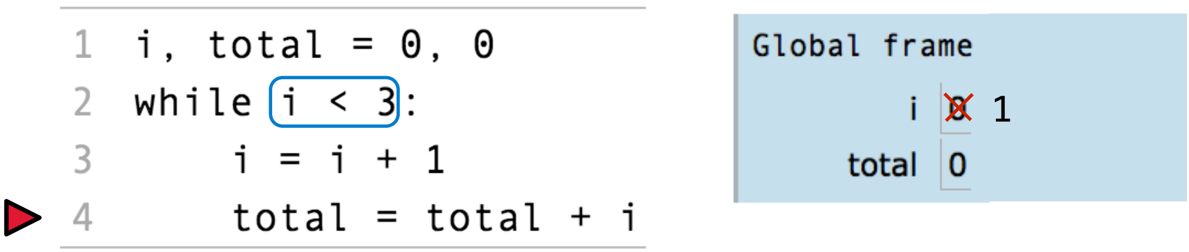
1. Evaluate the header’s expression.

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

While Statements

(Demo)

George Boole



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

Execution Rule for While Statements:

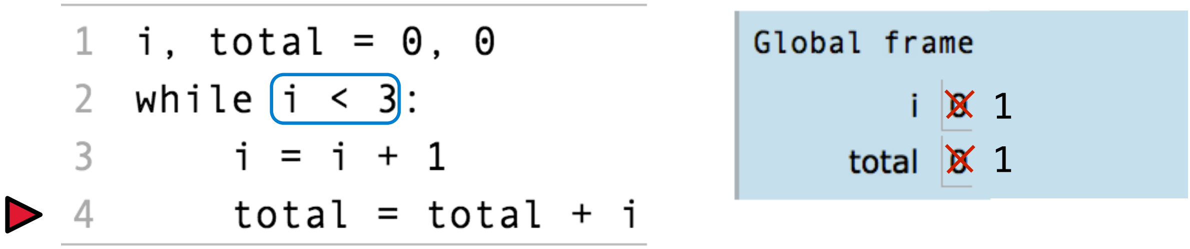
1. Evaluate the header’s expression.

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

While Statements

George Boole

(Demo)



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

Execution Rule for While Statements:

1. Evaluate the header's expression.

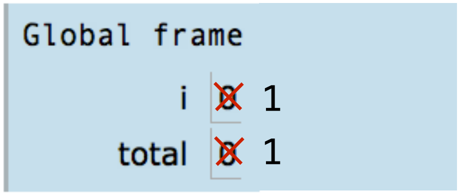
2. If it is a true value,
   execute the (whole) suite,
   then return to step 1.

While Statements

George Boole

(Demo)

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



Execution Rule for While Statements:

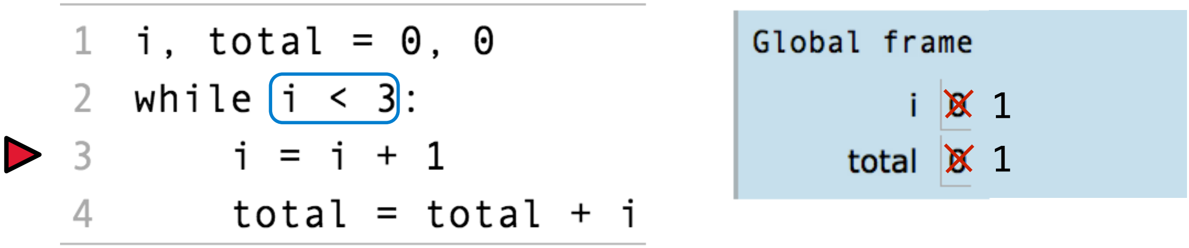
1. Evaluate the header’s expression.

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

While Statements

George Boole

(Demo)



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

Execution Rule for While Statements:

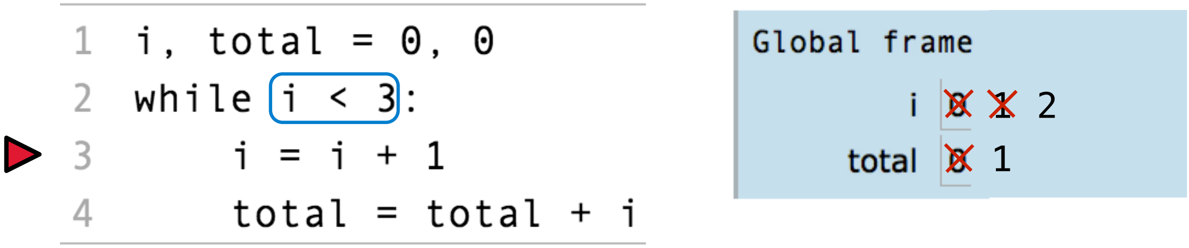
1. Evaluate the header’s expression.

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

While Statements

George Boole

(Demo)



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

Global frame

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

total ~~0~~ 1

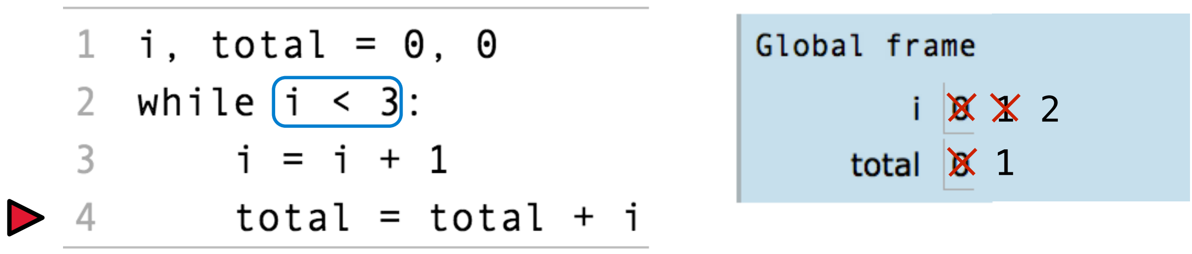
Execution Rule for While Statements:

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

While Statements

George Boole

(Demo)



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

Global frame

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

total ~~0~~ 1

Execution Rule for While Statements:

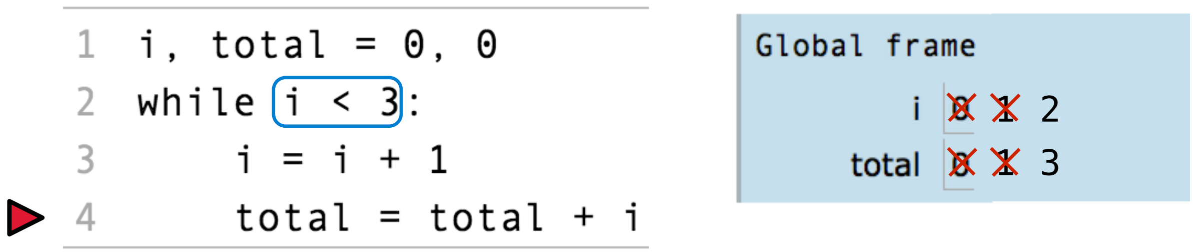
1. Evaluate the header’s expression.

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

While Statements

George Boole

(Demo)



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

Global frame

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

Execution Rule for While Statements:

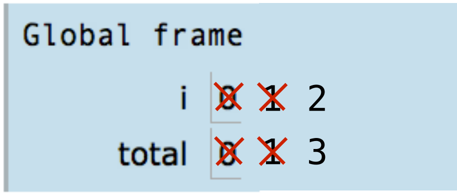
1. Evaluate the header's expression.
2. If it is a true value, execute the (whole) suite, then return to step 1.

While Statements

George Boole

(Demo)

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



Global frame

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

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

Execution Rule for While Statements:

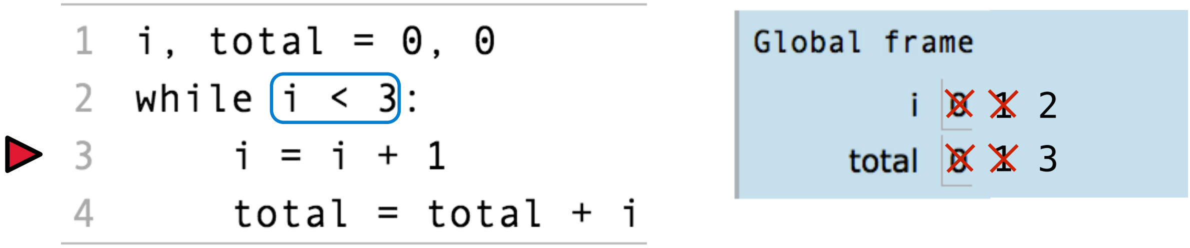
1. Evaluate the header’s expression.

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

While Statements

George Boole

(Demo)



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

Execution Rule for While Statements:

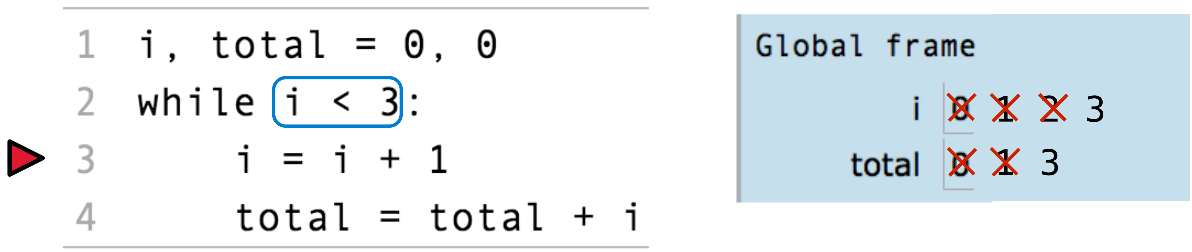
1. Evaluate the header’s expression.

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

While Statements

George Boole

(Demo)



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

Global frame

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

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

Execution Rule for While Statements:

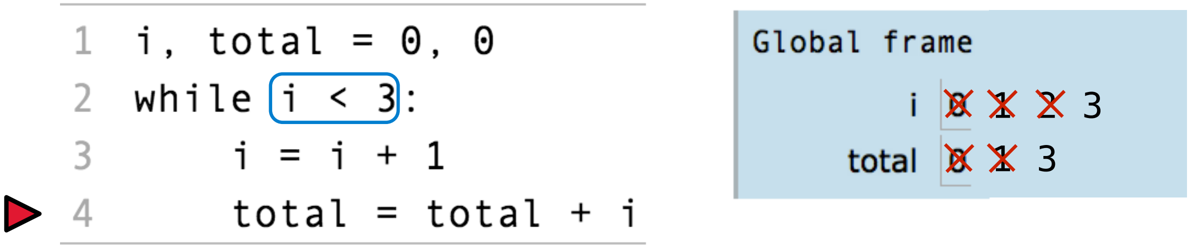
1. Evaluate the header’s expression.

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

While Statements

George Boole

(Demo)



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

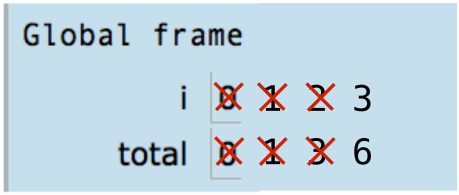
2. If it is a true value,
   execute the (whole) suite,
   then return to step 1.

While Statements

George Boole

(Demo)

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



Global frame

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

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

Execution Rule for While Statements:

1. Evaluate the header’s expression.

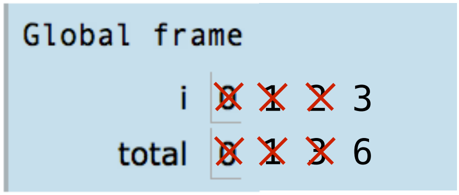
2. If it is a true value,
   execute the (whole) suite,
   then return to step 1.

While Statements

George Boole

(Demo)

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



Global frame

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

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

Execution Rule for While Statements:

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