

61A Lecture 2

Friday, August 28, 2015

61A Lecture 2

Names, Assignment, and User-Defined Functions

(Demo)

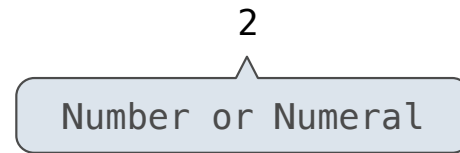
Types of Expressions

Types of Expressions

Primitive expressions:

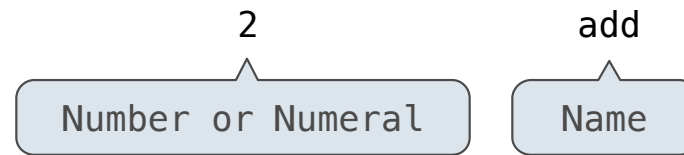
Types of Expressions

Primitive expressions:



Types of Expressions

Primitive expressions:



Types of Expressions

Primitive expressions:



Types of Expressions

Primitive expressions:



Call expressions:

Types of Expressions

Primitive expressions:



Call expressions:

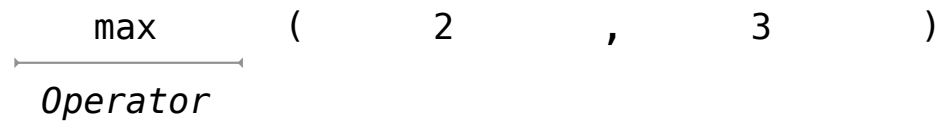
max (2 , 3)

Types of Expressions

Primitive expressions:



Call expressions:

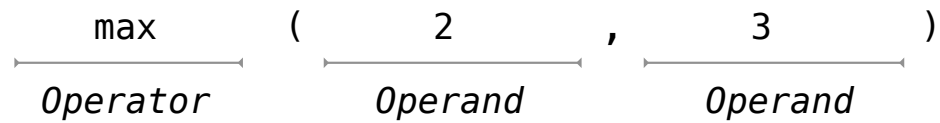


Types of Expressions

Primitive expressions:



Call expressions:

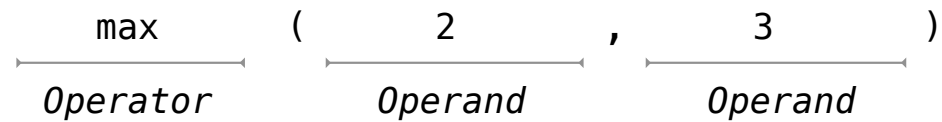


Types of Expressions

Primitive expressions:



Call expressions:



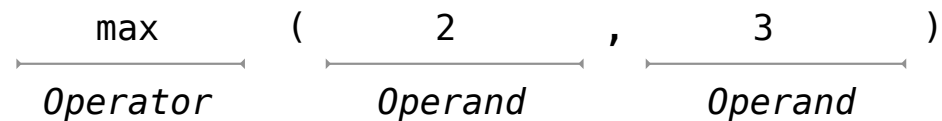
`max(min(pow(3, 5), -4), min(1, -2))`

Types of Expressions

Primitive expressions:



Call expressions:

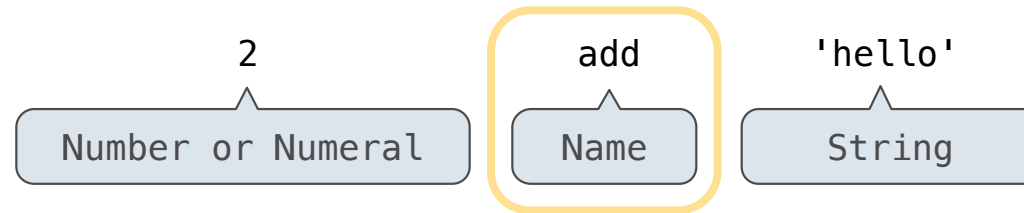


An operand can also be a call expression

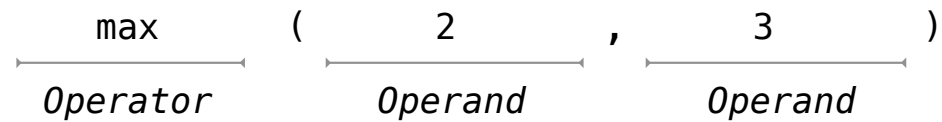
`max(min(pow(3, 5), -4), min(1, -2))`

Types of Expressions

Primitive expressions:



Call expressions:



An operand can also be a call expression

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

Discussion Question 1

Discussion Question 1

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

Discussion Question 1

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

```
>>> f = min
```

Discussion Question 1

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

```
>>> f = min
```

```
>>> f = max
```

Discussion Question 1

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

```
>>> f = min
```

```
>>> f = max
```

```
>>> g, h = min, max
```

Discussion Question 1

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

```
>>> f = min
```

```
>>> f = max
```

```
>>> g, h = min, max
```

```
>>> max = g
```

Discussion Question 1

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

```
>>> f = min
```

```
>>> f = max
```

```
>>> g, h = min, max
```

```
>>> max = g
```

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

Discussion Question 1

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

```
>>> f = min
```

```
>>> f = max
```

```
>>> g, h = min, max
```

```
>>> max = g
```

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

???

Discussion Question 1

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

```
>>> f = min
```

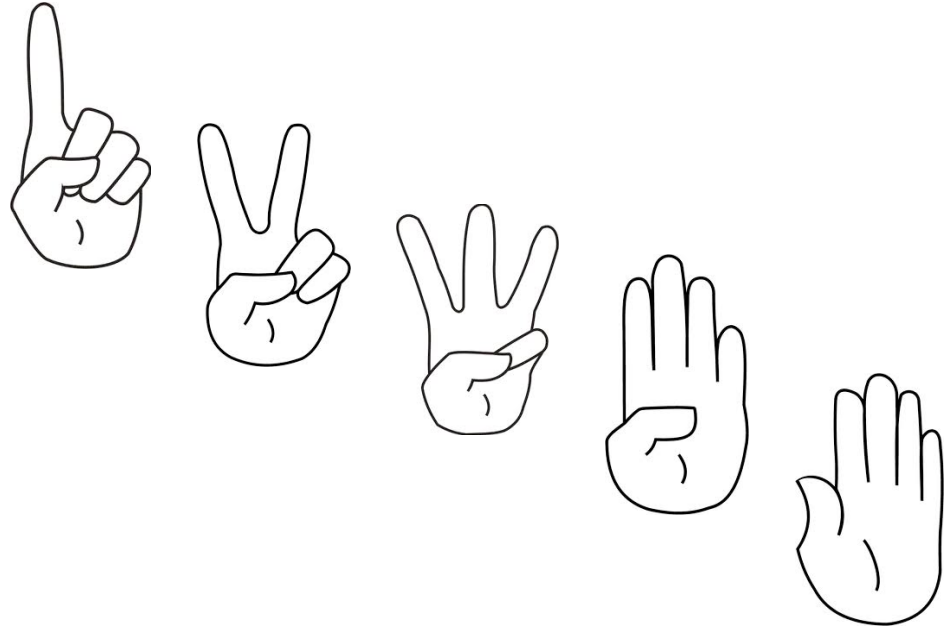
```
>>> f = max
```

```
>>> g, h = min, max
```

```
>>> max = g
```

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

???



Environment Diagrams

Environment Diagrams

Environment diagrams visualize the interpreter's process.

Interactive Diagram

Environment Diagrams

Environment diagrams visualize the interpreter's process.

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

[Interactive Diagram](#)

Environment Diagrams

Environment diagrams visualize the interpreter's process.

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

```
Global frame  
pi 3.1416
```

[Interactive Diagram](#)

Environment Diagrams

Environment diagrams visualize the interpreter's process.

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

```
Global frame  
pi 3.1416
```

Code (left):

Frames (right):

[Interactive Diagram](#)

Environment Diagrams

Environment diagrams visualize the interpreter's process.

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

```
Global frame  
pi 3.1416
```

Code (left):

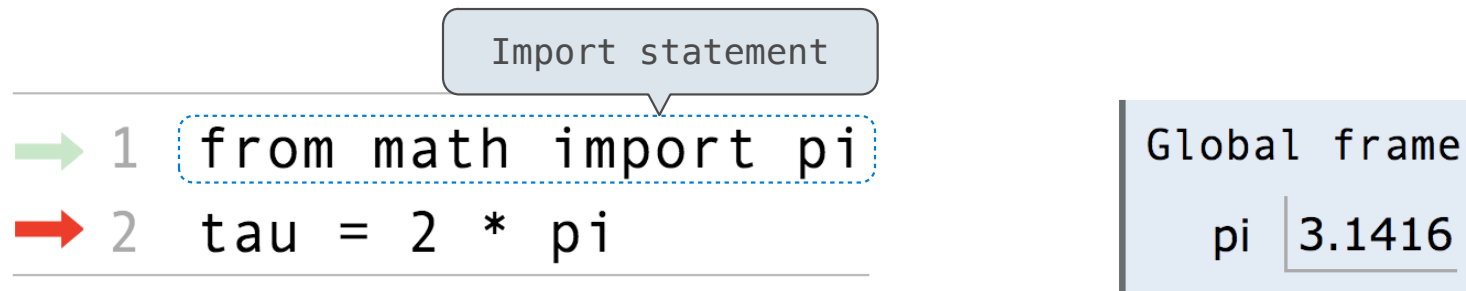
Statements and expressions

Frames (right):

[Interactive Diagram](#)

Environment Diagrams

Environment diagrams visualize the interpreter's process.



Code (left):

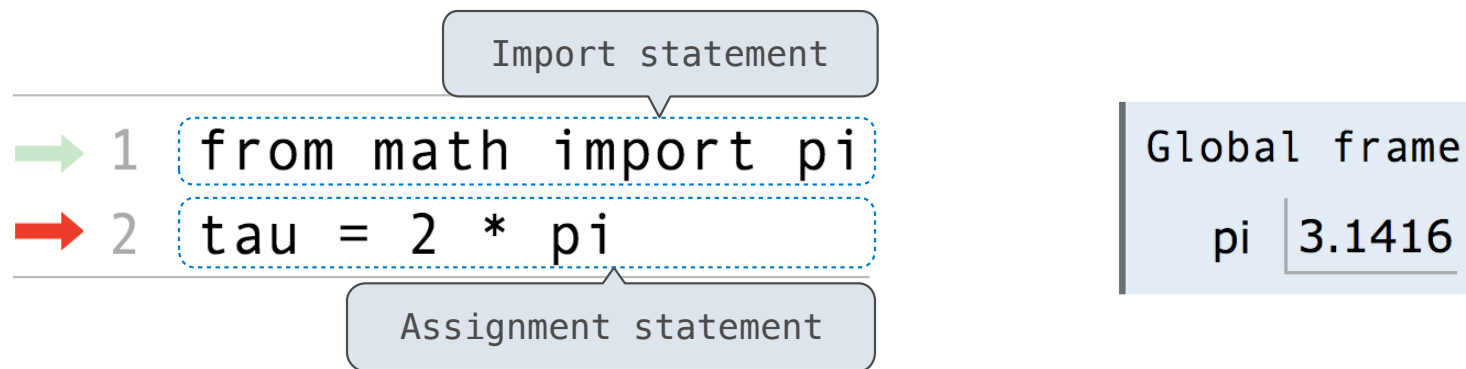
Statements and expressions

Frames (right):

[Interactive Diagram](#)

Environment Diagrams

Environment diagrams visualize the interpreter's process.



Code (left):

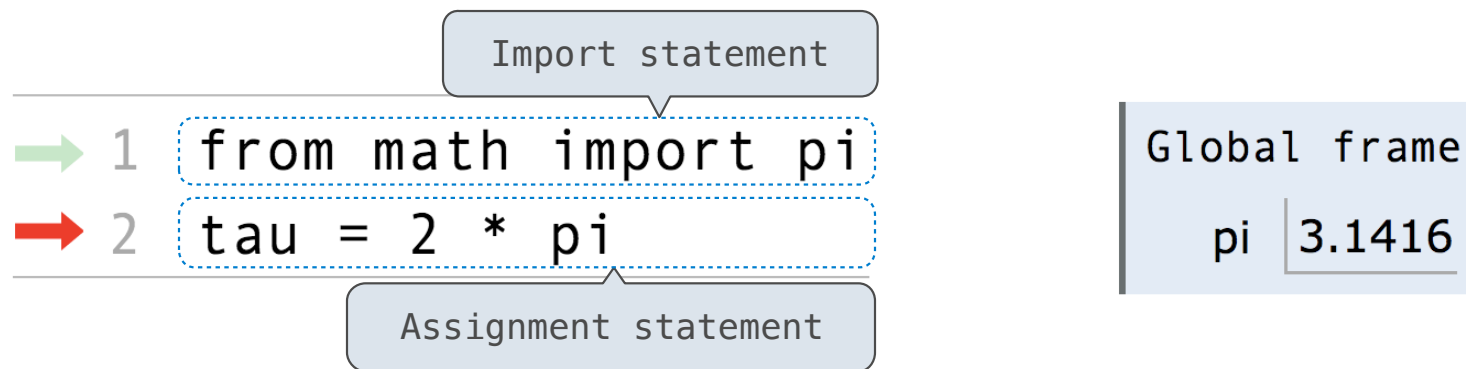
Statements and expressions

Frames (right):

[Interactive Diagram](#)

Environment Diagrams

Environment diagrams visualize the interpreter's process.



Code (left):

Statements and expressions

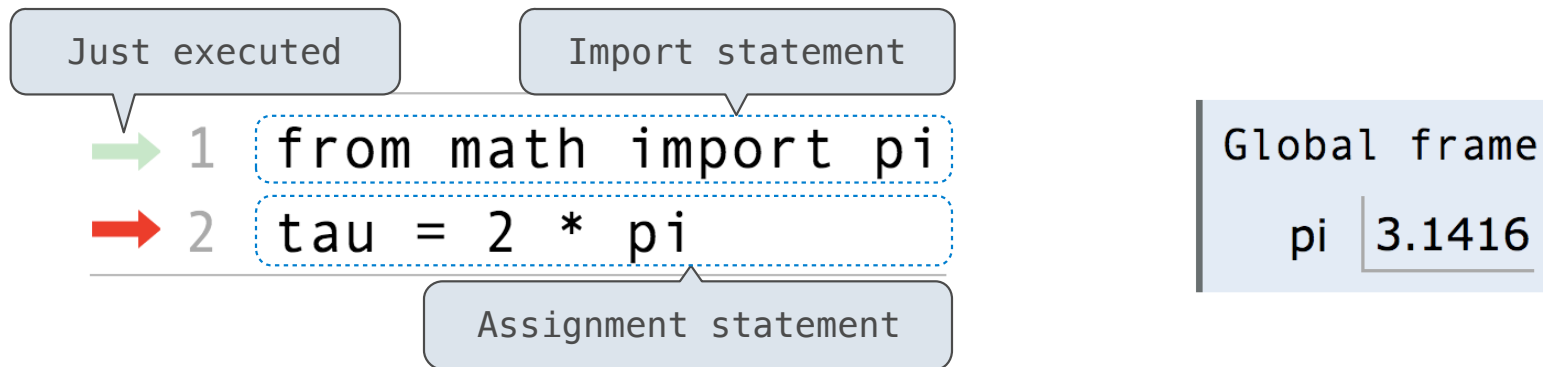
Arrows indicate evaluation order

Frames (right):

[Interactive Diagram](#)

Environment Diagrams

Environment diagrams visualize the interpreter's process.



Code (left):

Statements and expressions

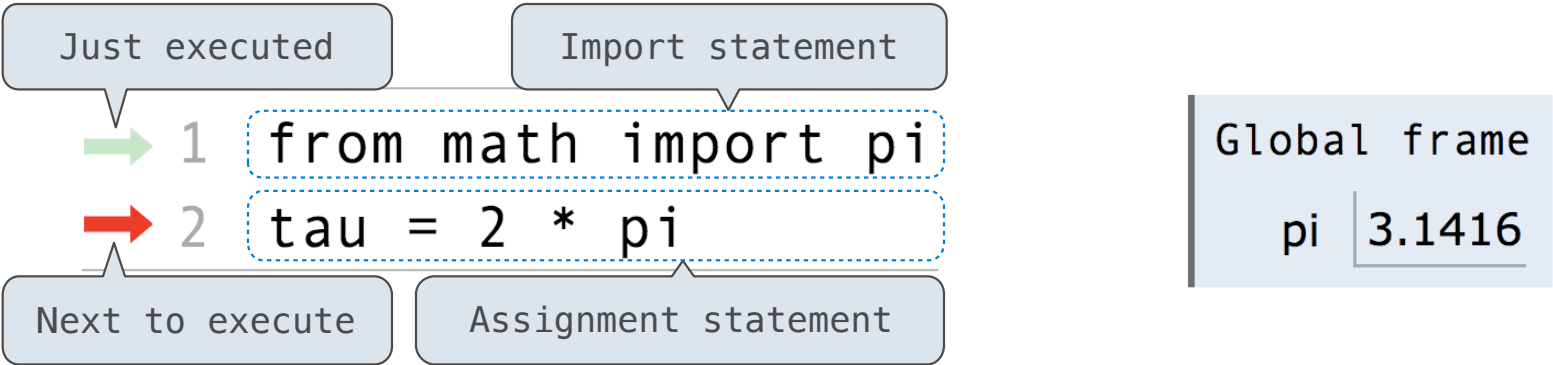
Arrows indicate evaluation order

Frames (right):

[Interactive Diagram](#)

Environment Diagrams

Environment diagrams visualize the interpreter's process.



Code (left):

Statements and expressions

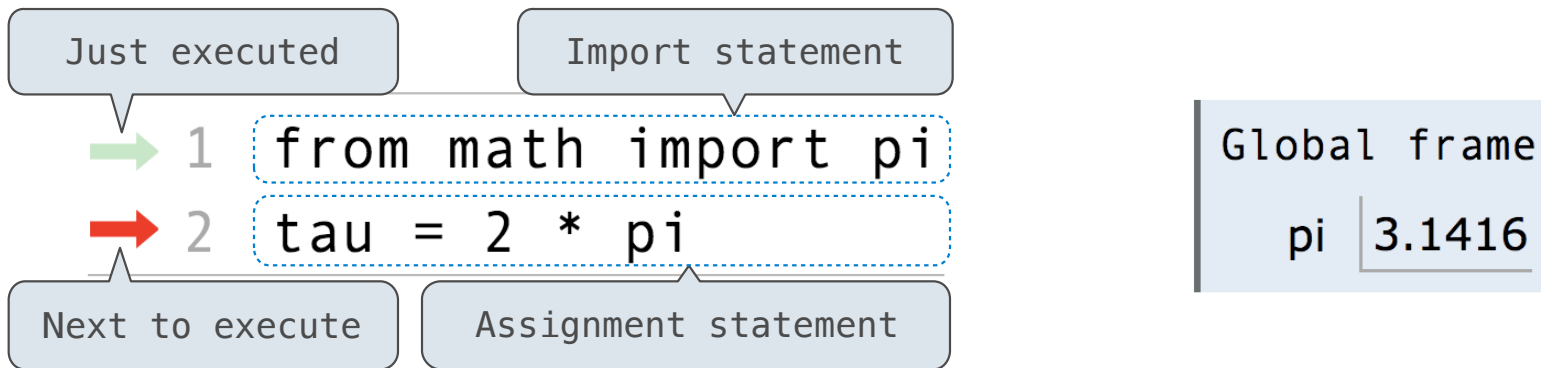
Arrows indicate evaluation order

Frames (right):

Interactive Diagram

Environment Diagrams

Environment diagrams visualize the interpreter's process.



Code (left):

Statements and expressions

Arrows indicate evaluation order

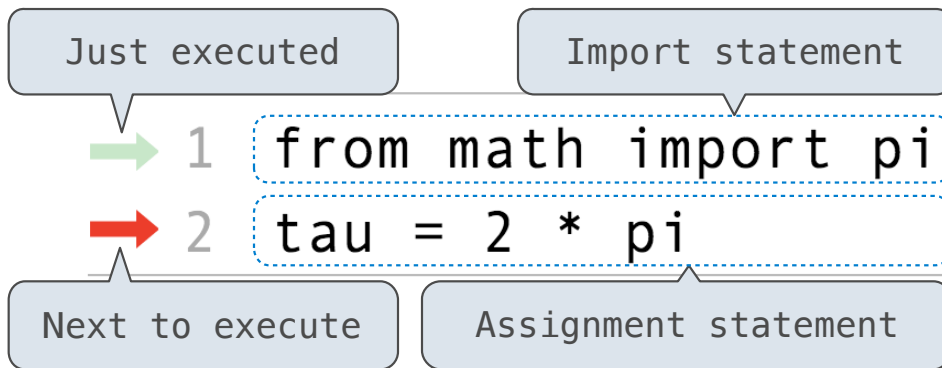
Frames (right):

Each name is bound to a value

[Interactive Diagram](#)

Environment Diagrams

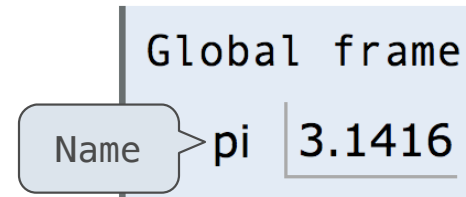
Environment diagrams visualize the interpreter's process.



Code (left):

Statements and expressions

Arrows indicate evaluation order



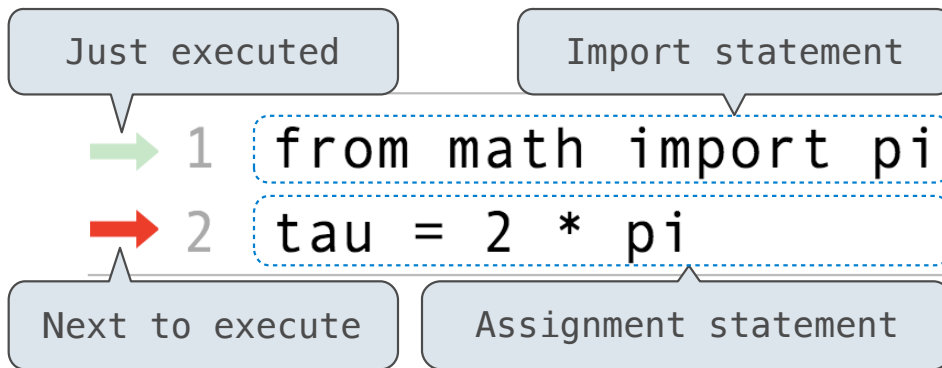
Frames (right):

Each name is bound to a value

[Interactive Diagram](#)

Environment Diagrams

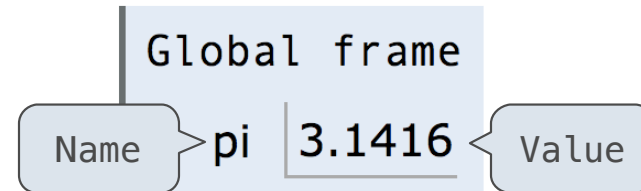
Environment diagrams visualize the interpreter's process.



Code (left):

Statements and expressions

Arrows indicate evaluation order



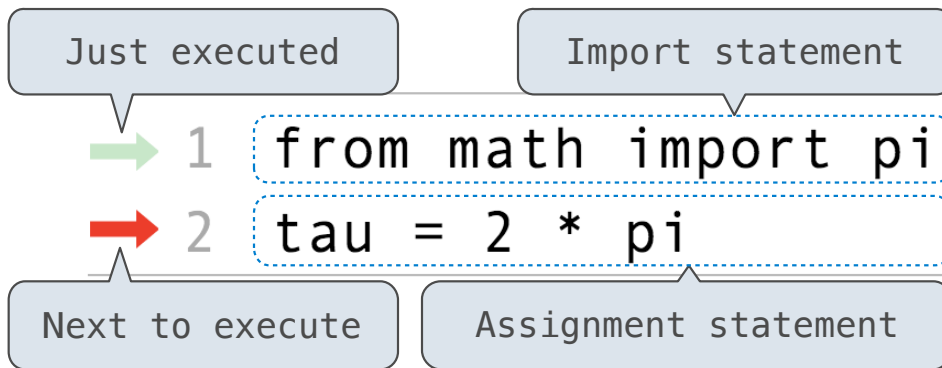
Frames (right):

Each name is bound to a value

[Interactive Diagram](#)

Environment Diagrams

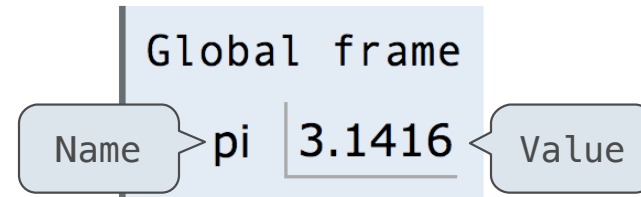
Environment diagrams visualize the interpreter's process.



Code (left):

Statements and expressions

Arrows indicate evaluation order



Frames (right):

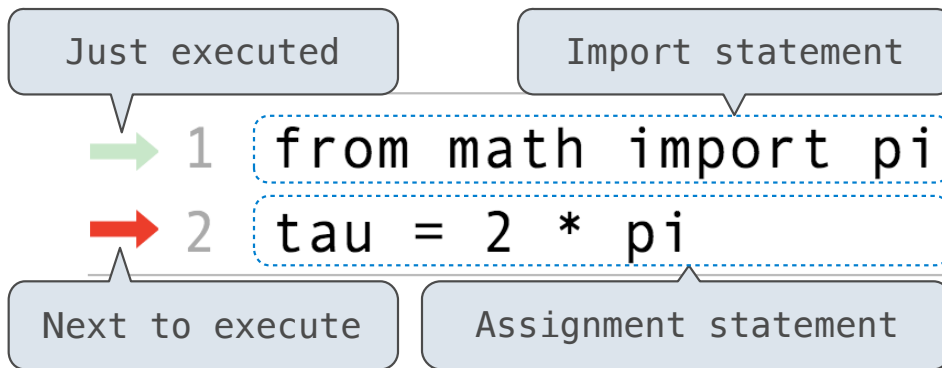
Each name is bound to a value

Within a frame, a name cannot be repeated

[Interactive Diagram](#)

Environment Diagrams

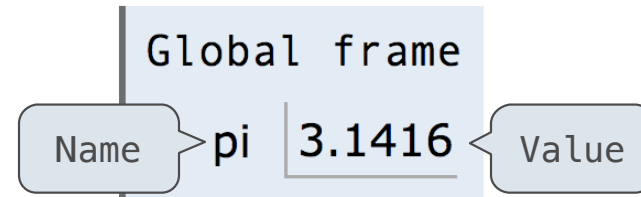
Environment diagrams visualize the interpreter's process.



Code (left):

Statements and expressions

Arrows indicate evaluation order



Frames (right):

Each name is bound to a value

Within a frame, a name cannot be repeated

(Demo)

[Interactive Diagram](#)

Assignment Statements

Interactive Diagram

Assignment Statements

```
1 a = 1  
→ 2 b = 2  
→ 3 b, a = a + b, b
```

Interactive Diagram

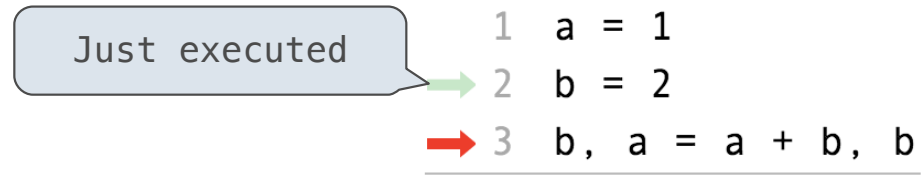
Assignment Statements

```
1 a = 1  
→ 2 b = 2  
→ 3 b, a = a + b, b
```

Global frame

a	1
b	2

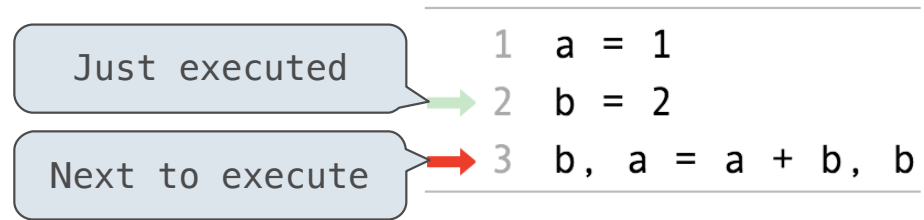
Assignment Statements



Global frame

a	1
b	2

Assignment Statements

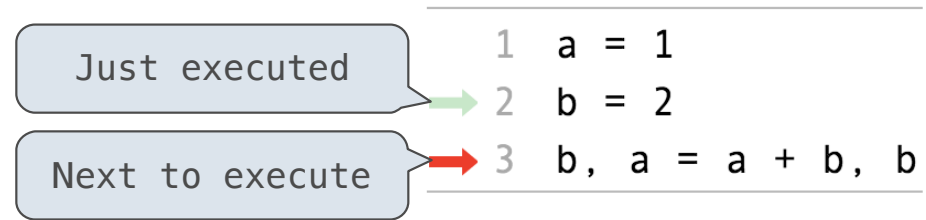


Global frame

a | 1

b | 2

Assignment Statements



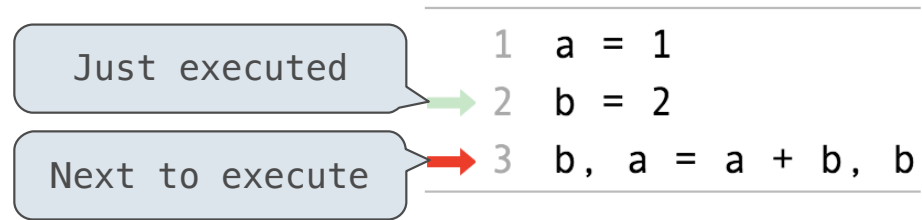
Global frame

a | 1

b | 2

Execution rule for assignment statements:

Assignment Statements



Global frame

a | 1

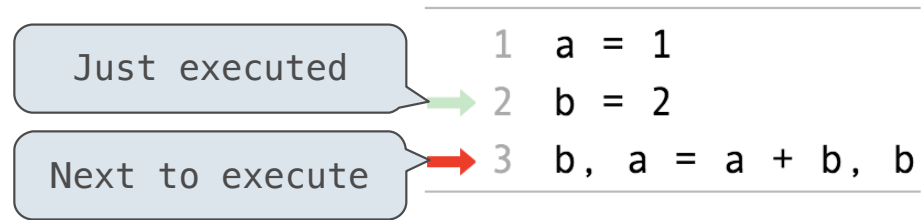
b | 2

Execution rule for assignment statements:

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

[Interactive Diagram](#)

Assignment Statements



Global frame

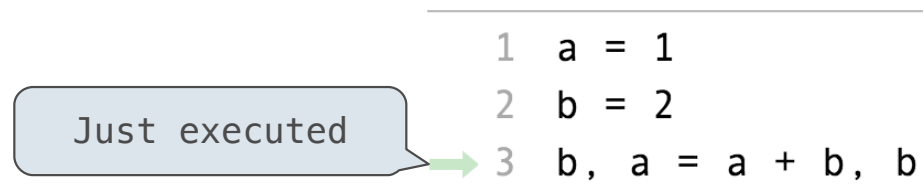
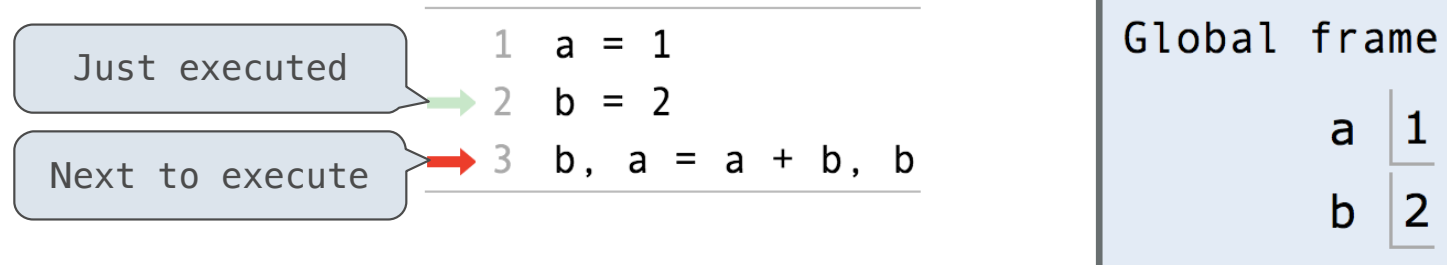
a	1
b	2

Execution rule for assignment statements:

1. Evaluate all expressions to the right of = from left to right.
2. Bind all names to the left of = to those resulting values in the current frame.

[Interactive Diagram](#)

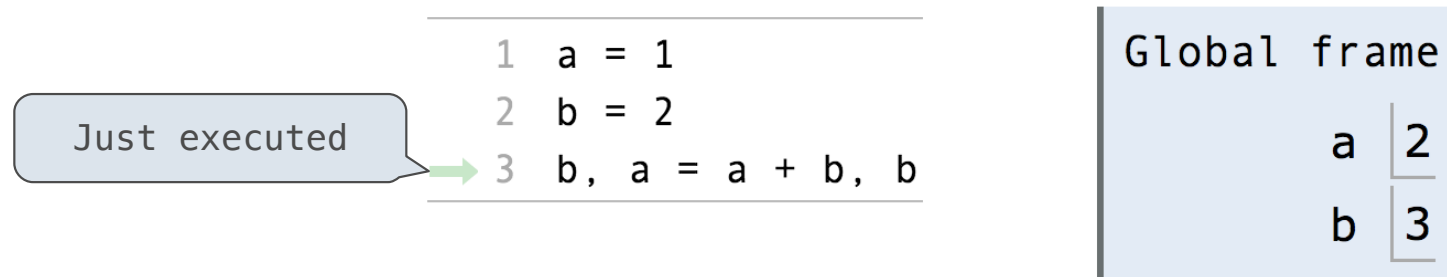
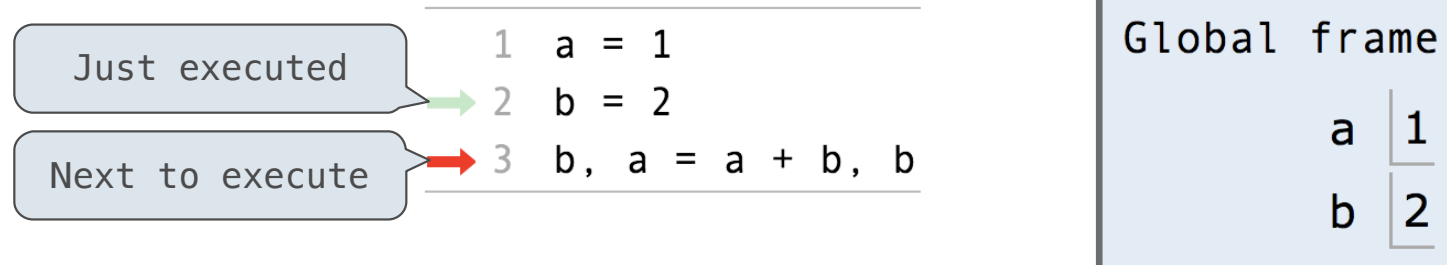
Assignment Statements



Execution rule for assignment statements:

1. Evaluate all expressions to the right of = from left to right.
2. Bind all names to the left of = to those resulting values in the current frame.

Assignment Statements



Execution rule for assignment statements:

1. Evaluate all expressions to the right of = from left to right.
2. Bind all names to the left of = to those resulting values in the current frame.

Discussion Question 1 Solution

(Demo)

Interactive Diagram

Discussion Question 1 Solution

(Demo)

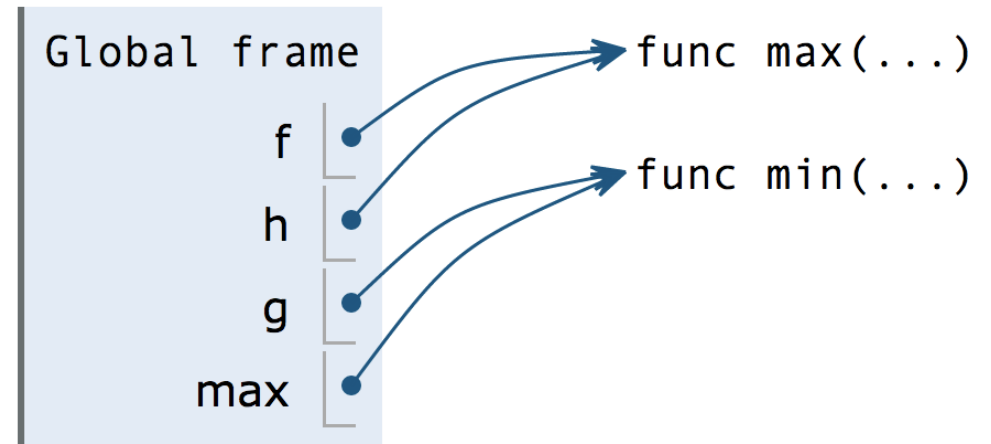
```
1 f = min
2 f = max
3 g, h = min, max
→ 4 max = g
→ 5 max(f(2, g(h(1, 5), 3)), 4)
```

[Interactive Diagram](#)

Discussion Question 1 Solution

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

(Demo)

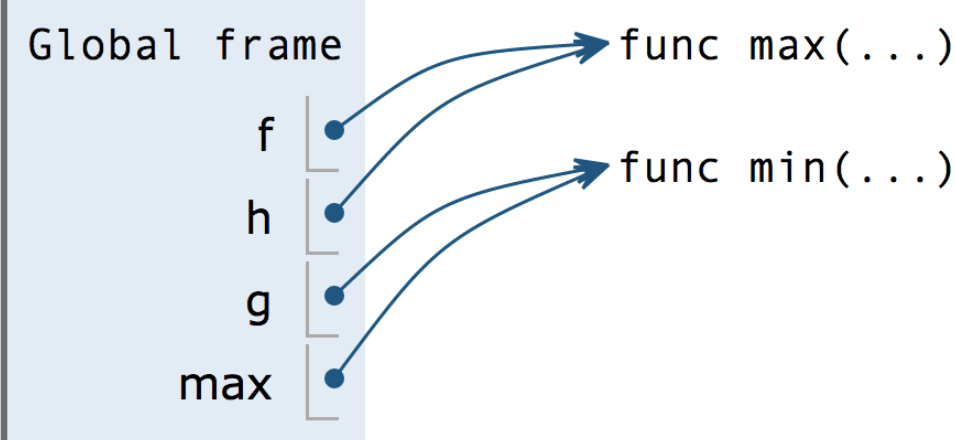


Interactive Diagram

Discussion Question 1 Solution

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

(Demo)

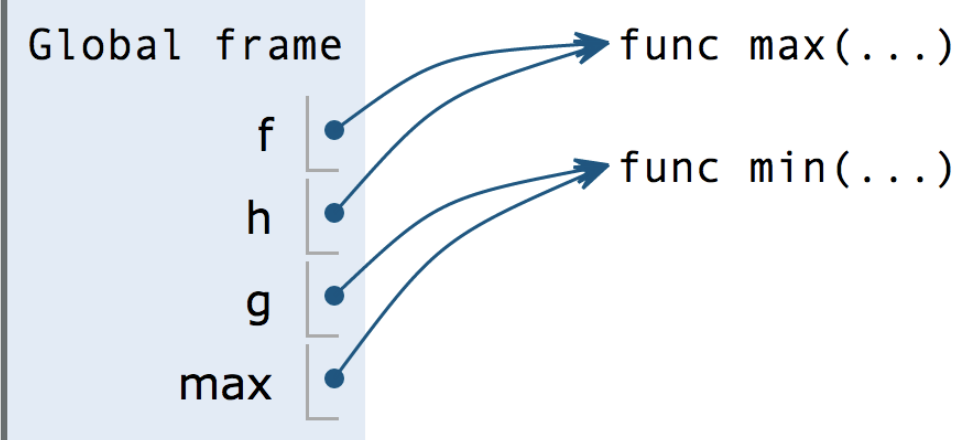


Discussion Question 1 Solution

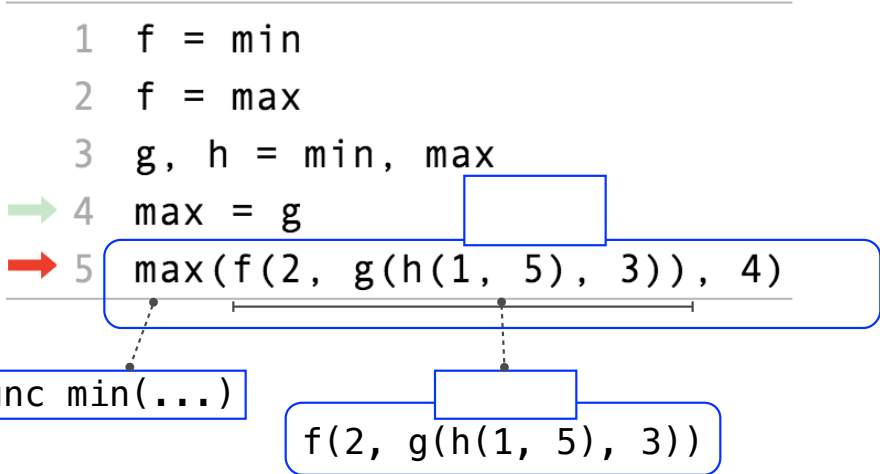
```
1 f = min
2 f = max
3 g, h = min, max
4 max = g
5 max(f(2, g(h(1, 5), 3)), 4)
```

func min(...)

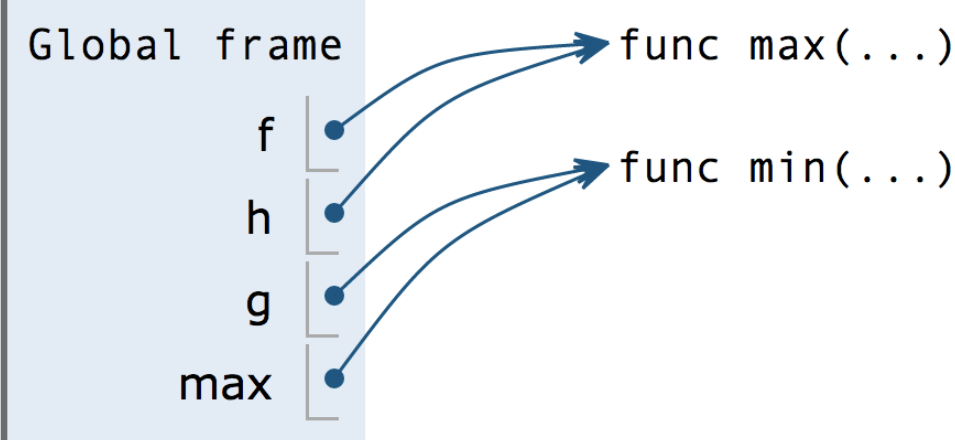
(Demo)



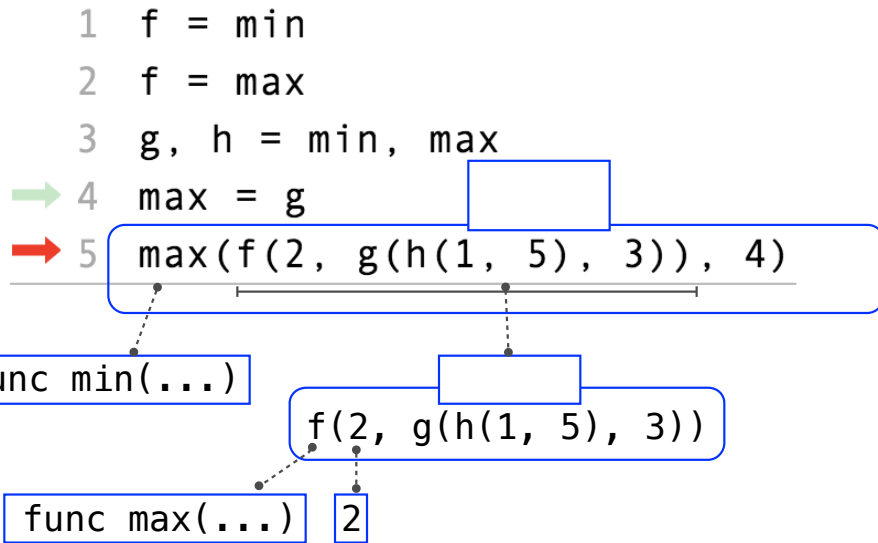
Discussion Question 1 Solution



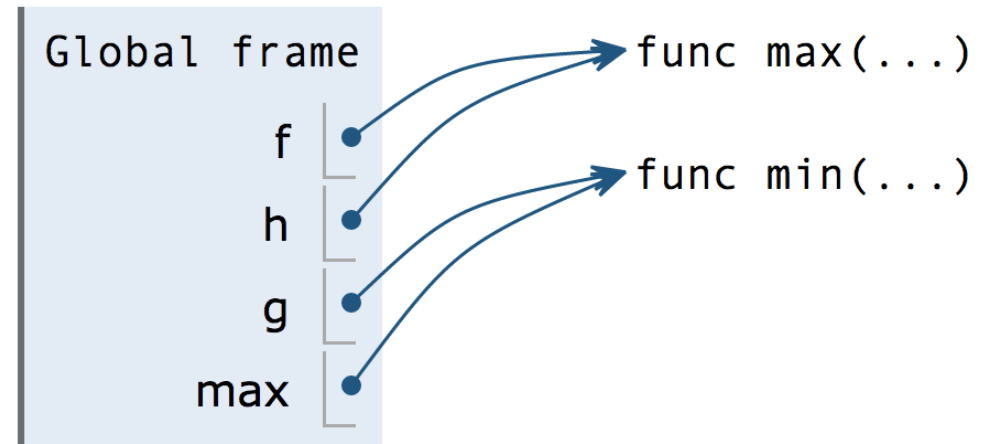
(Demo)



Discussion Question 1 Solution



(Demo)



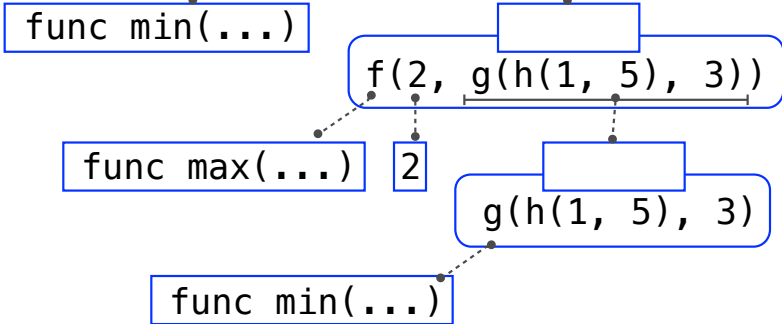
Interactive Diagram

Discussion Question 1 Solution

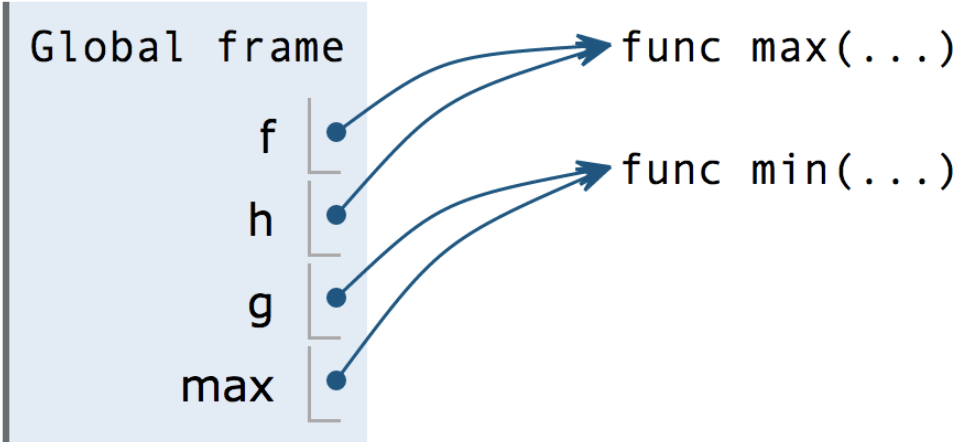
```

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

```

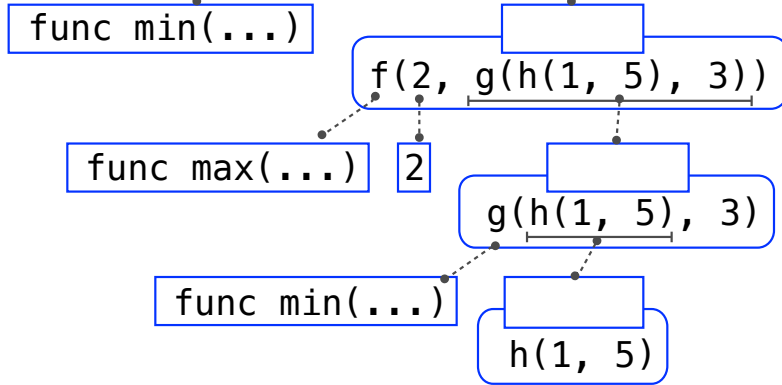


(Demo)

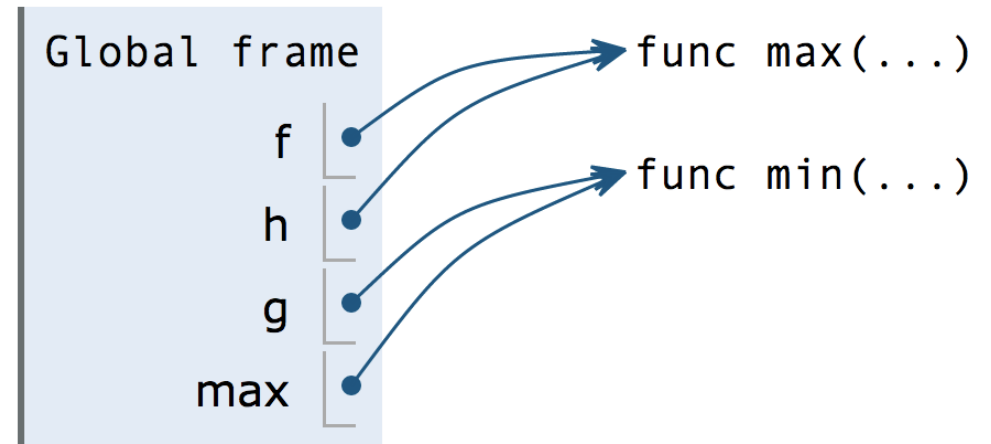


Discussion Question 1 Solution

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



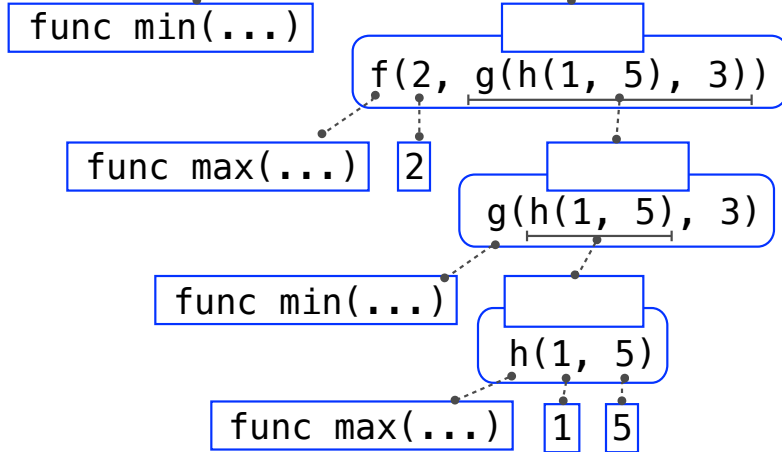
(Demo)



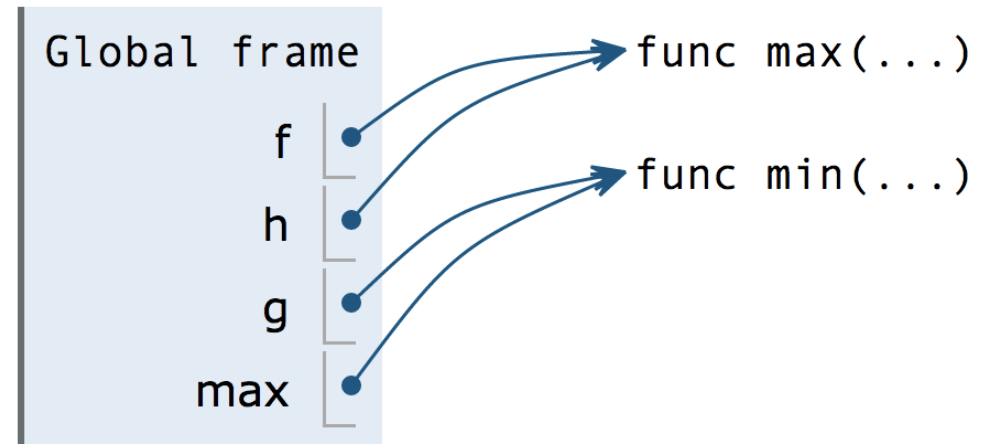
Interactive Diagram

Discussion Question 1 Solution

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



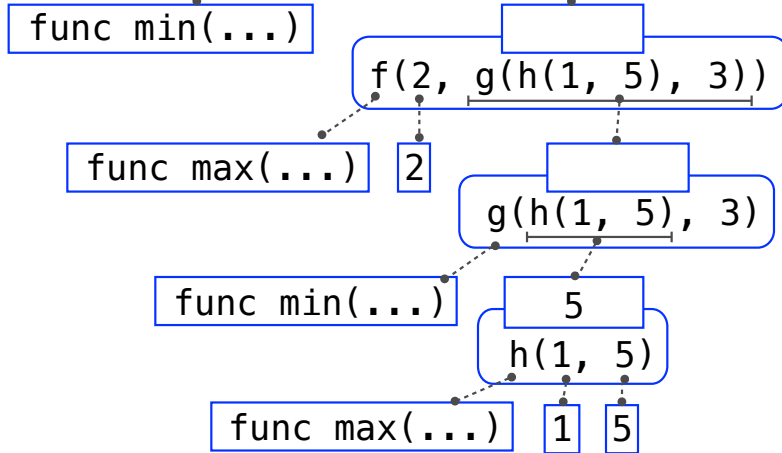
(Demo)



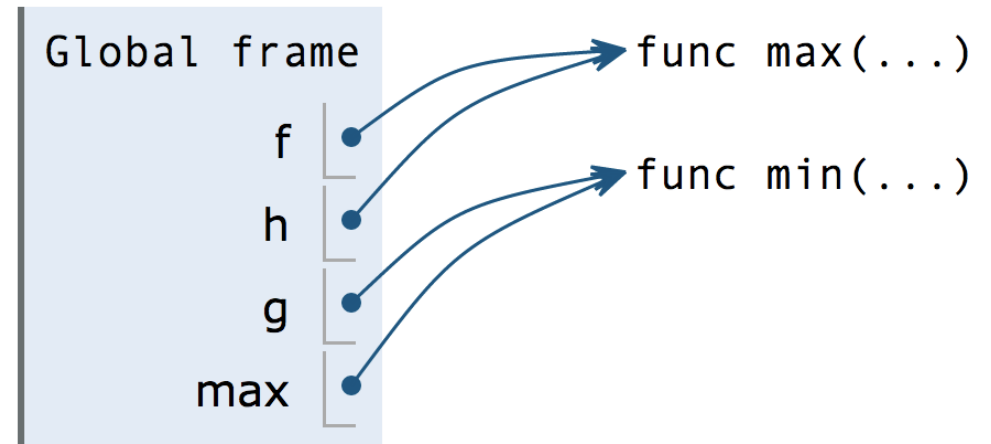
Interactive Diagram

Discussion Question 1 Solution

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



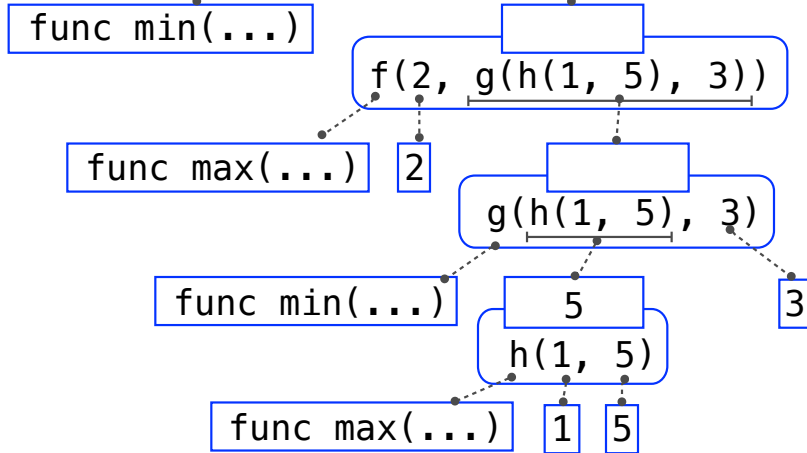
(Demo)



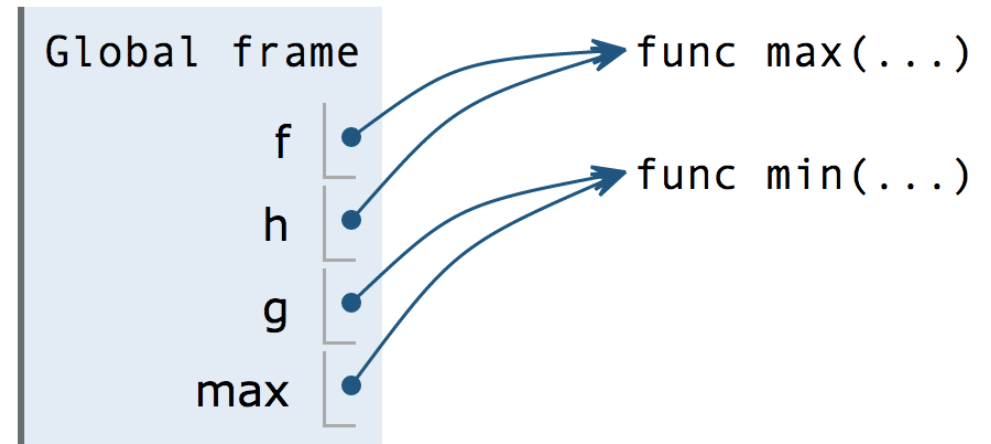
Interactive Diagram

Discussion Question 1 Solution

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



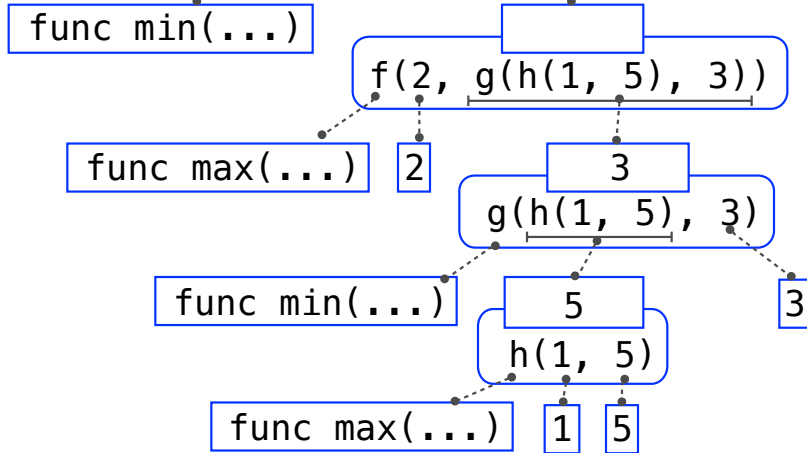
(Demo)



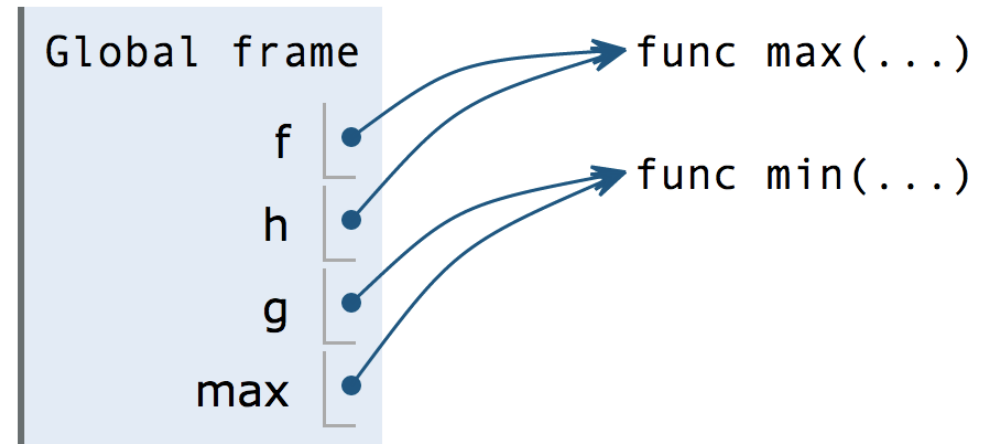
Interactive Diagram

Discussion Question 1 Solution

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



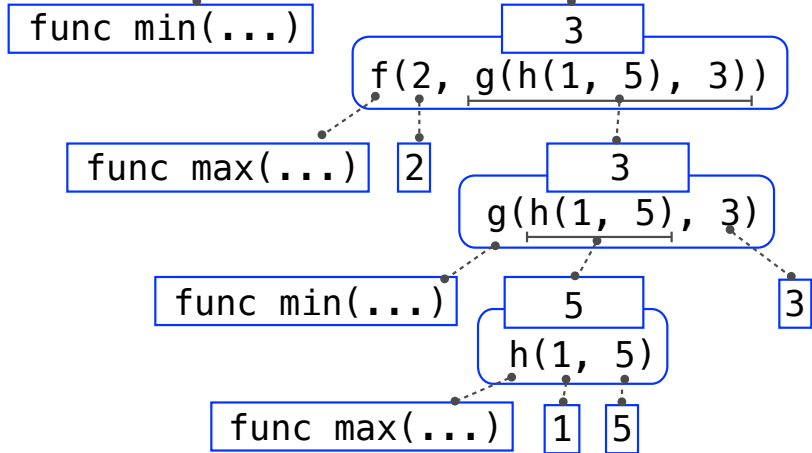
(Demo)



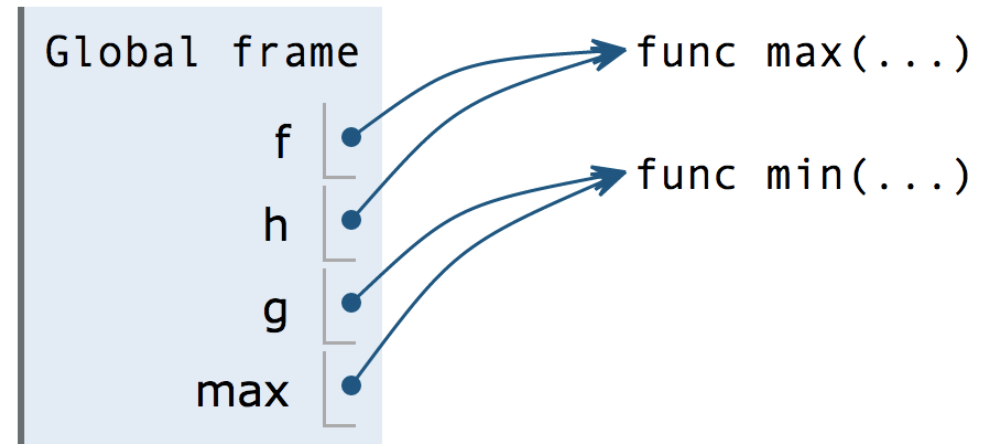
Interactive Diagram

Discussion Question 1 Solution

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



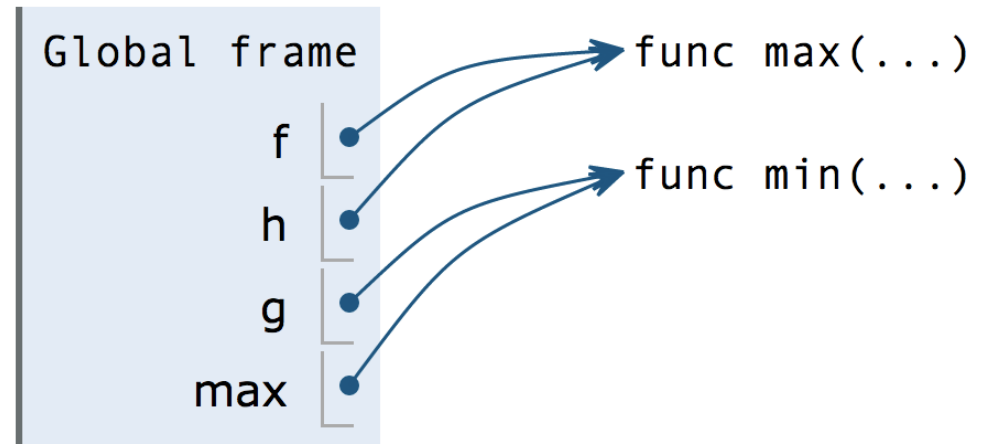
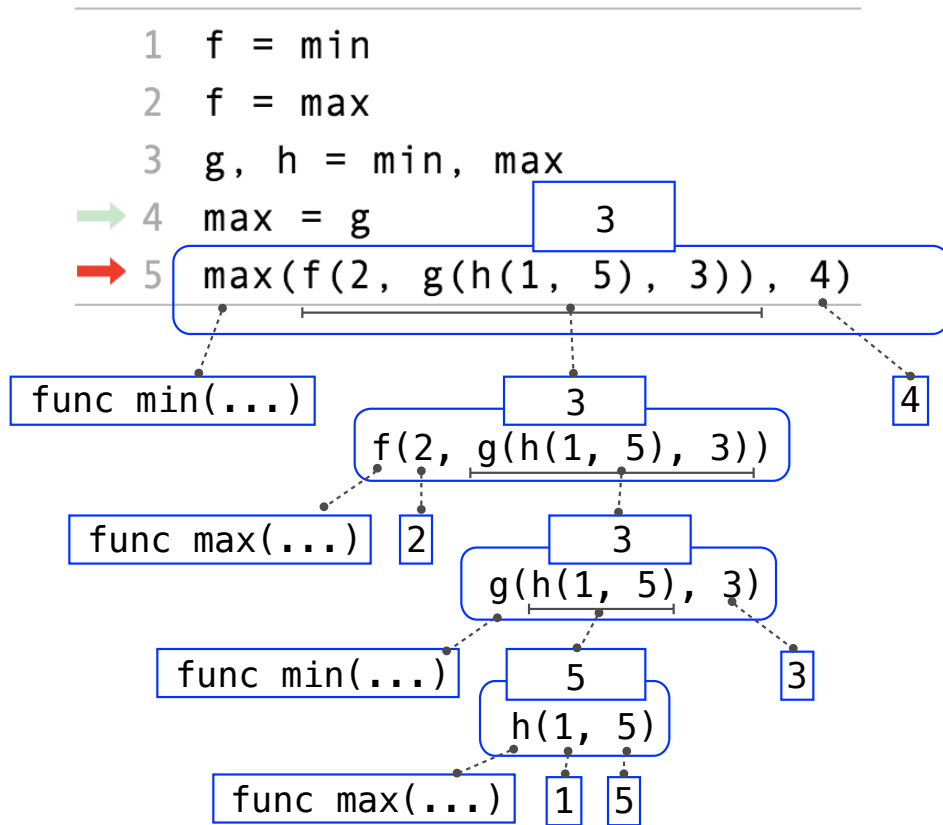
(Demo)



Interactive Diagram

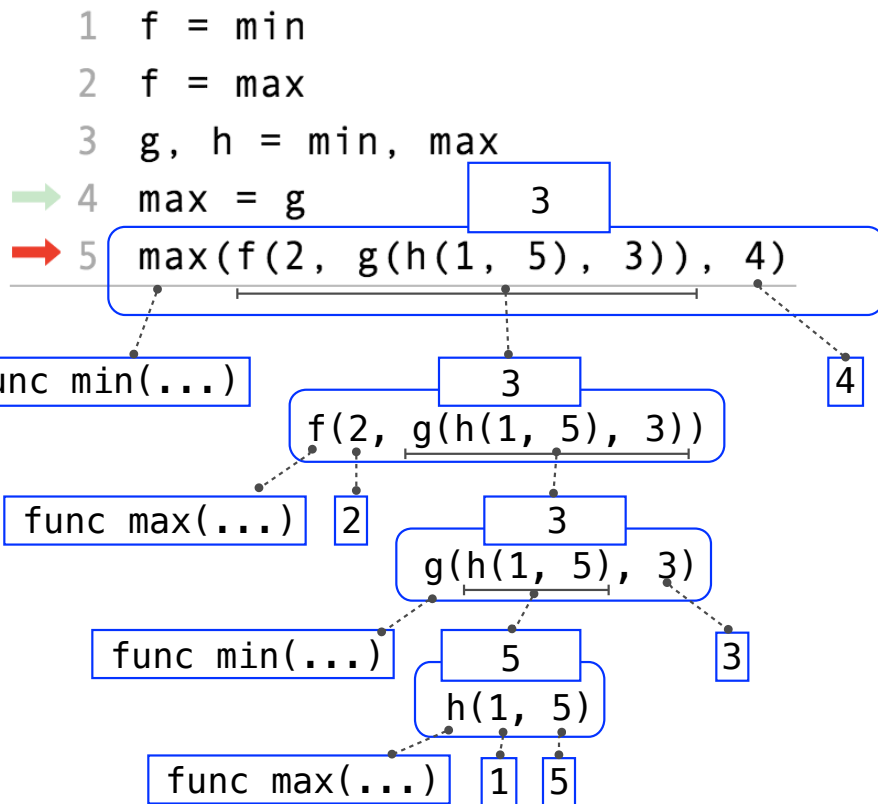
Discussion Question 1 Solution

(Demo)

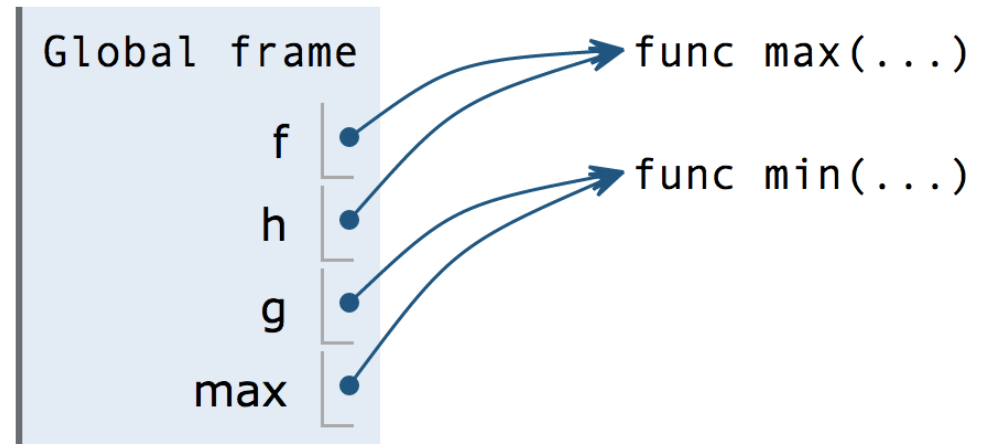


Interactive Diagram

Discussion Question 1 Solution



(Demo)



3

Interactive Diagram

Defining Functions

Defining Functions

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

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

Defining Functions

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

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

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

Defining Functions

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

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

Function *signature* indicates how many arguments a function takes

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


Defining Functions

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

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

Function *signature* indicates how many arguments a function takes

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

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

Defining Functions

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

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

Function *signature* indicates how many arguments a function takes

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

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

Execution procedure for def statements:

Defining Functions

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

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

Function *signature* indicates how many arguments a function takes

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

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

Execution procedure for def statements:

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

Defining Functions

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

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

Function *signature* indicates how many arguments a function takes

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

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

Execution procedure for def statements:

1. Create a function with signature `<name>(<formal parameters>)`
2. Set the body of that function to be everything indented after the first line

Defining Functions

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

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

Function *signature* indicates how many arguments a function takes

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

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

Execution procedure for def statements:

1. Create a function with signature `<name>(<formal parameters>)`
2. Set the body of that function to be everything indented after the first line
3. Bind `<name>` to that function in the current frame

Calling User-Defined Functions

Interactive Diagram

Calling User-Defined Functions

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

[Interactive Diagram](#)

Calling User-Defined Functions

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

1. Add a local frame, forming a new environment

[Interactive Diagram](#)

Calling User-Defined Functions

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

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

Calling User-Defined Functions

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

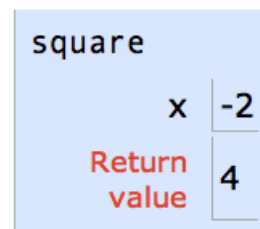
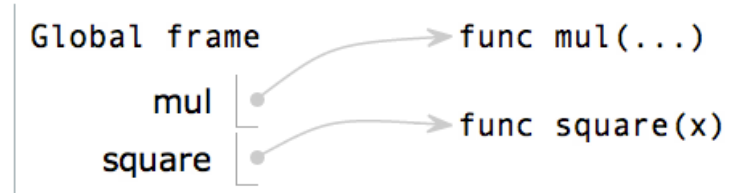
1. Add a local frame, forming a new environment
2. Bind the function's formal parameters to its arguments in that frame
3. Execute the body of the function in that new environment

Calling User-Defined Functions

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

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

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



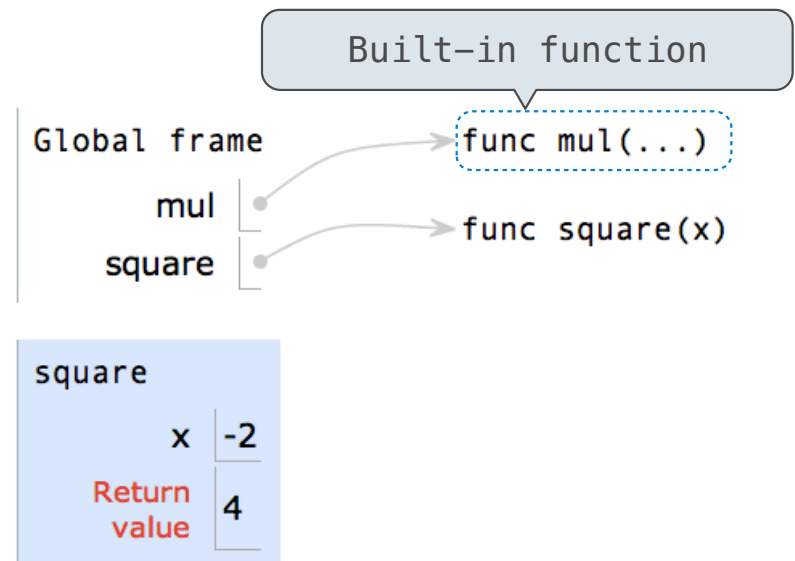
Interactive Diagram

Calling User-Defined Functions

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

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

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



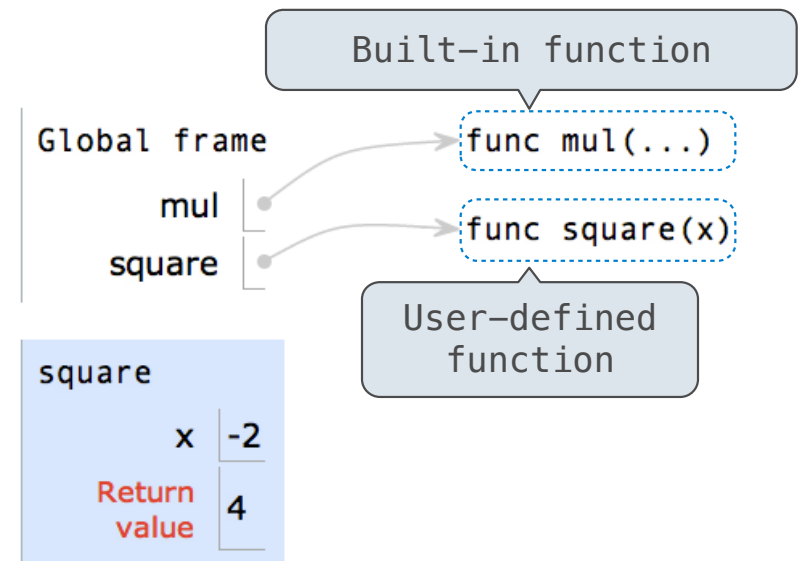
Interactive Diagram

Calling User-Defined Functions

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

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

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



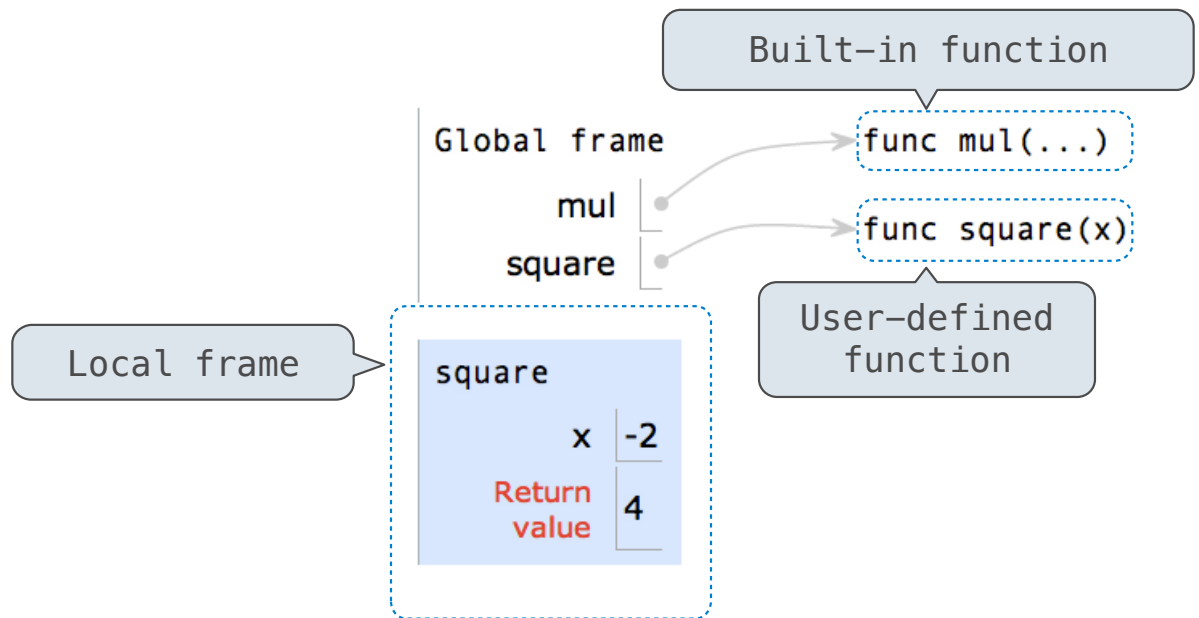
Interactive Diagram

Calling User-Defined Functions

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

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

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



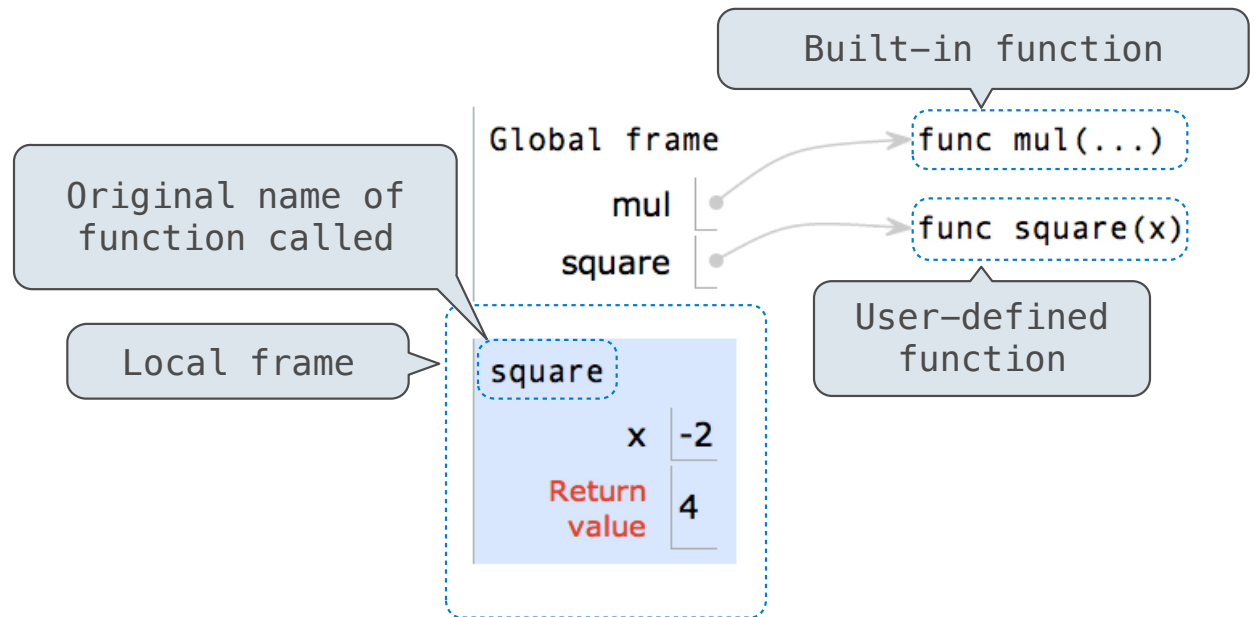
Interactive Diagram

Calling User-Defined Functions

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

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

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



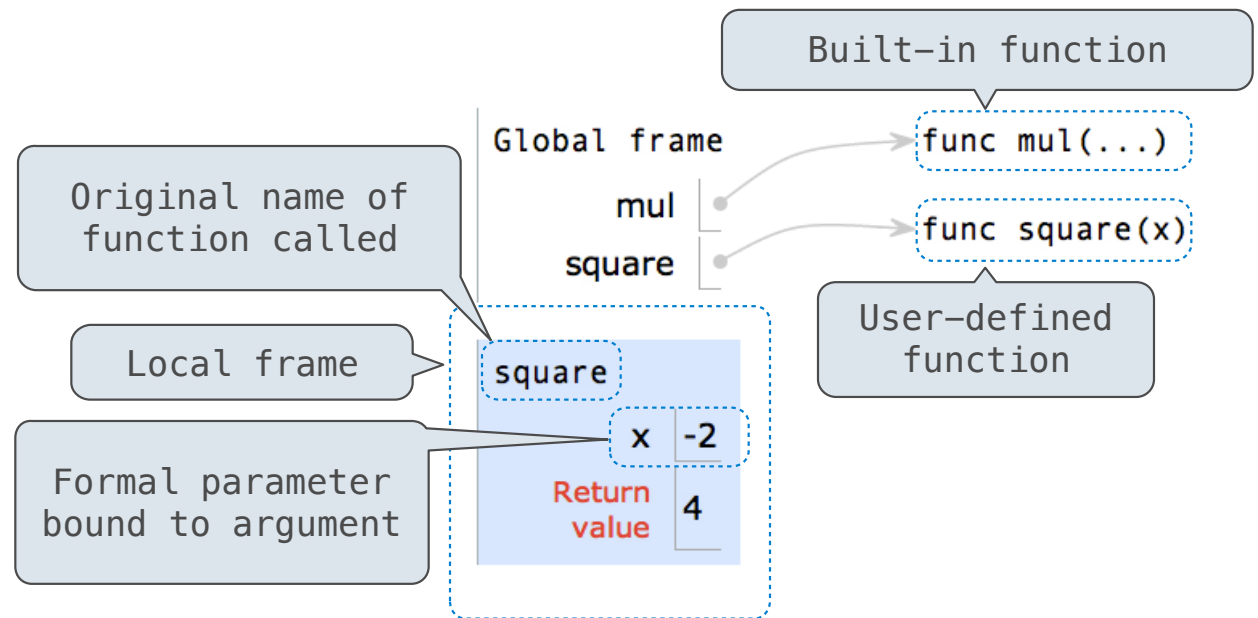
Interactive Diagram

Calling User-Defined Functions

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

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

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



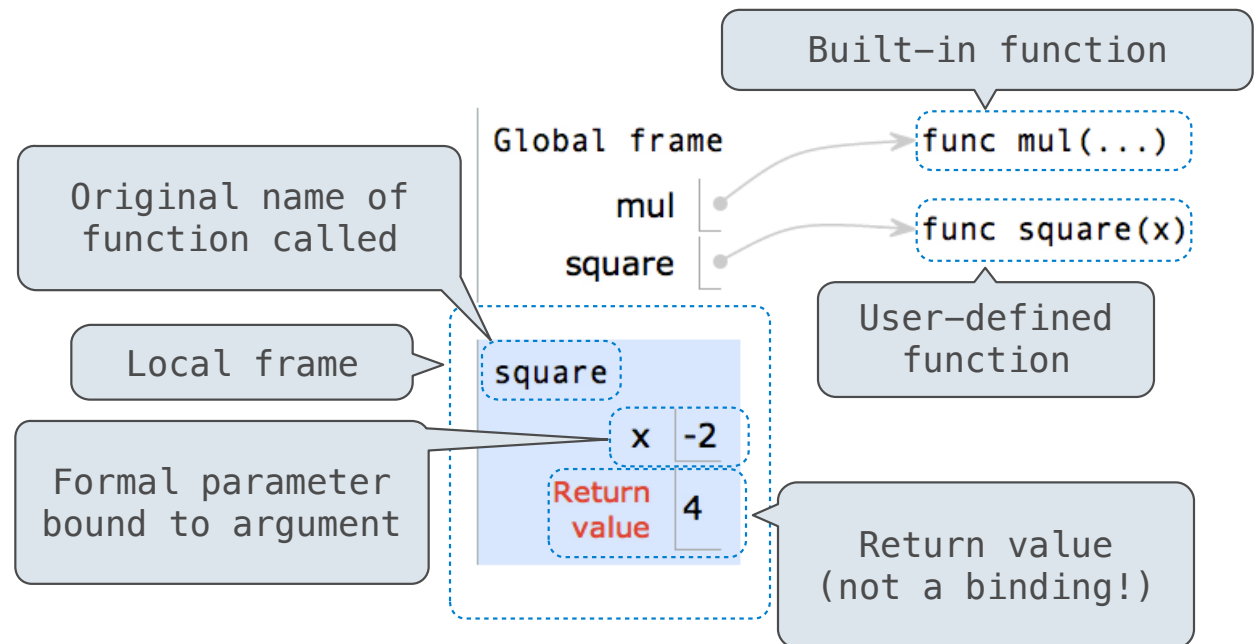
Interactive Diagram

Calling User-Defined Functions

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

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

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



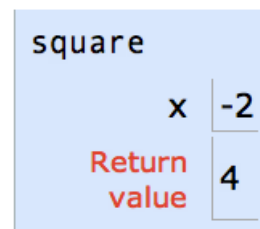
Interactive Diagram

Calling User-Defined Functions

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

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

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



Interactive Diagram

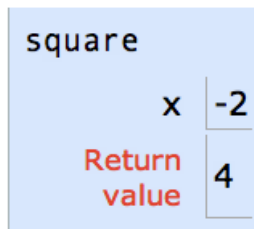
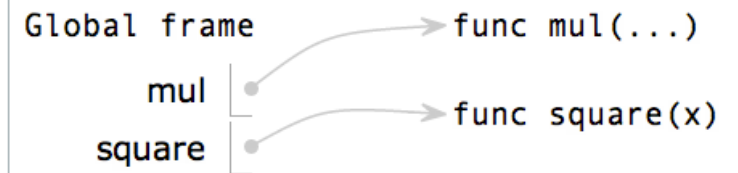
Calling User-Defined Functions

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

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

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

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



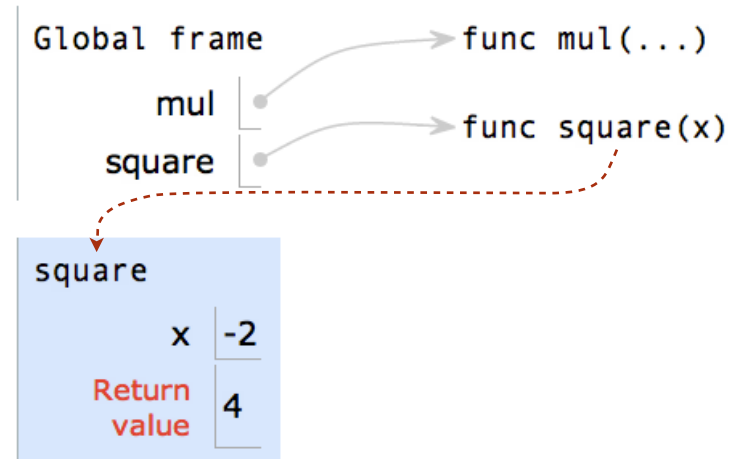
Calling User-Defined Functions

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

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

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

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



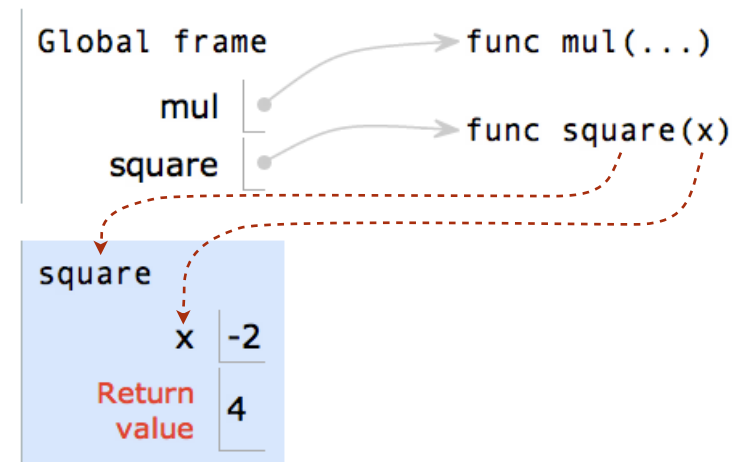
Calling User-Defined Functions

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

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

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

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



Looking Up Names In Environments

Looking Up Names In Environments

Every expression is evaluated in the context of an environment.

Looking Up Names In Environments

Every expression is evaluated in the context of an environment.

So far, the current environment is either:

Looking Up Names In Environments

Every expression is evaluated in the context of an environment.

So far, the current environment is either:

- The global frame alone, or

Looking Up Names In Environments

Every expression is evaluated in the context of an environment.

So far, the current environment is either:

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

Looking Up Names In Environments

Every expression is evaluated in the context of an environment.

So far, the current environment is either:

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

Most important two things I'll say all day:

Looking Up Names In Environments

Every expression is evaluated in the context of an environment.

So far, the current environment is either:

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

Most important two things I'll say all day:

An environment is a sequence of frames.

Looking Up Names In Environments

Every expression is evaluated in the context of an environment.

So far, the current environment is either:

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

Most important two things I'll say all day:

An environment is a sequence of frames.

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

Looking Up Names In Environments

Every expression is evaluated in the context of an environment.

So far, the current environment is either:

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

Most important two things I'll say all day:

An environment is a sequence of frames.

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

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

Looking Up Names In Environments

Every expression is evaluated in the context of an environment.

So far, the current environment is either:

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

Most important two things I'll say all day:

An environment is a sequence of frames.

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

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

- Look for that name in the local frame.

Looking Up Names In Environments

Every expression is evaluated in the context of an environment.

So far, the current environment is either:

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

Most important two things I'll say all day:

An environment is a sequence of frames.

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

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

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

Looking Up Names In Environments

Every expression is evaluated in the context of an environment.

So far, the current environment is either:

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

Most important two things I'll say all day:

An environment is a sequence of frames.

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

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

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

(Demo)

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_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_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_square(x):  
...     x * x  
...  
>>> does_not_square(4)
```



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

None Indicates that Nothing is Returned

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

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

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

```
>>> def does_not_square(x):  
...     x * x  
...  
>>> does_not_square(4)
```

The diagram illustrates the execution of the `does_not_square` function. A callout box labeled "No return" points to the absence of a `return` statement in the function definition. Another callout box labeled "None value is not displayed" points to the function call `does_not_square(4)`, indicating that the interpreter does not show the `None` result.

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_square(x):  
...     x * x  
...  
>>> does_not_square(4)  
>>> sixteen = does_not_square(4)
```

The diagram illustrates the execution of the `does_not_square` function. A callout box labeled "No return" points to the `x * x` line, indicating that the function does not return a value. Another callout box labeled "None value is not displayed" points to the `does_not_square(4)` line, indicating that the interpreter does not display the `None` result.

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_square(x):
```

```
...     x * x
```

No return

```
...     >>> does_not_square(4)
```

None value is not displayed

```
>>> sixteen = does_not_square(4)
```

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

None Indicates that Nothing is Returned

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

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

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

```
>>> def does_not_square(x):
```

```
...     x * x
...     
```

No return

```
>>> does_not_square(4)
```

None value is not displayed

```
>>> sixteen = does_not_square(4)
```

```
>>> sixteen + 4
```

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

```
Traceback (most recent call last):
```

```
File "<stdin>", line 1, in <module>
```

```
TypeError: unsupported operand type(s) for +: 'NoneType' and 'int'
```

Pure Functions & Non-Pure Functions

Pure Functions

just return values

Non-Pure Functions

have side effects

Pure Functions & Non-Pure Functions

Pure Functions
just return values

```
abs
```



Non-Pure Functions
have side effects

Pure Functions & Non-Pure Functions

Pure Functions
just return values



Non-Pure Functions
have side effects

Pure Functions & Non-Pure Functions

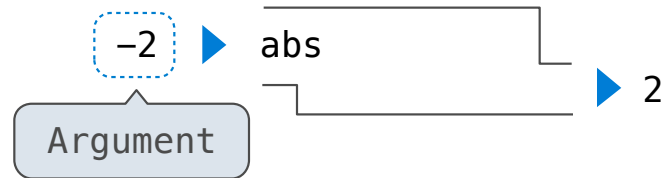
Pure Functions
just return values



Non-Pure Functions
have side effects

Pure Functions & Non-Pure Functions

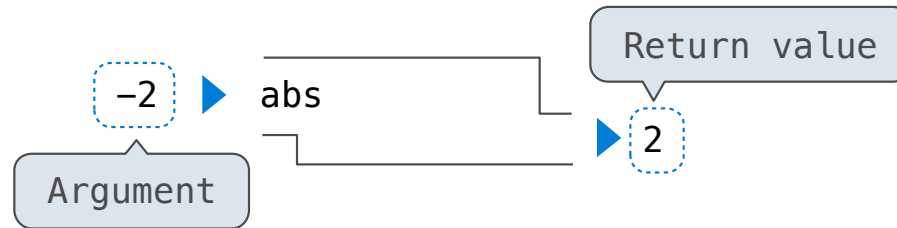
Pure Functions
just return values



Non-Pure Functions
have side effects

Pure Functions & Non-Pure Functions

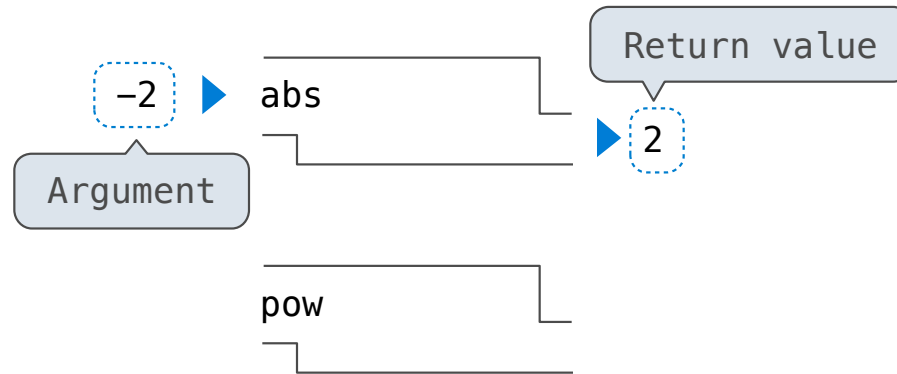
Pure Functions
just return values



Non-Pure Functions
have side effects

Pure Functions & Non-Pure Functions

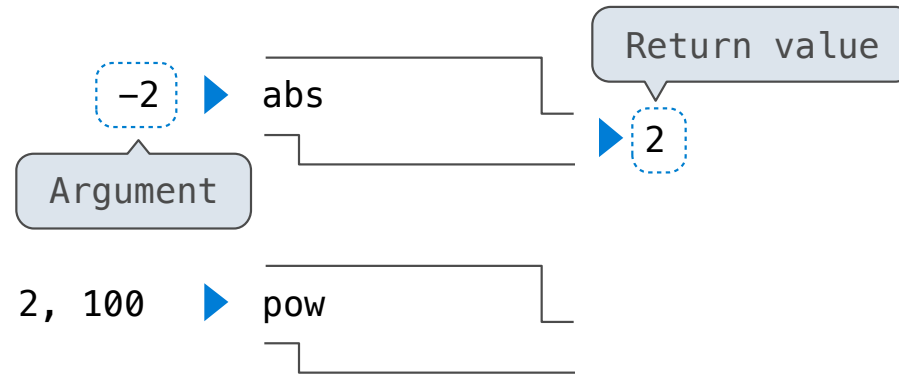
Pure Functions
just return values



Non-Pure Functions
have side effects

Pure Functions & Non-Pure Functions

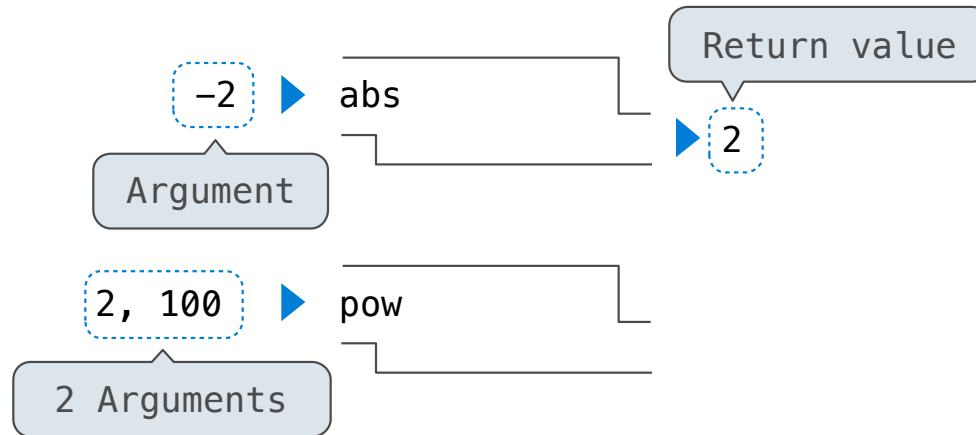
Pure Functions
just return values



Non-Pure Functions
have side effects

Pure Functions & Non-Pure Functions

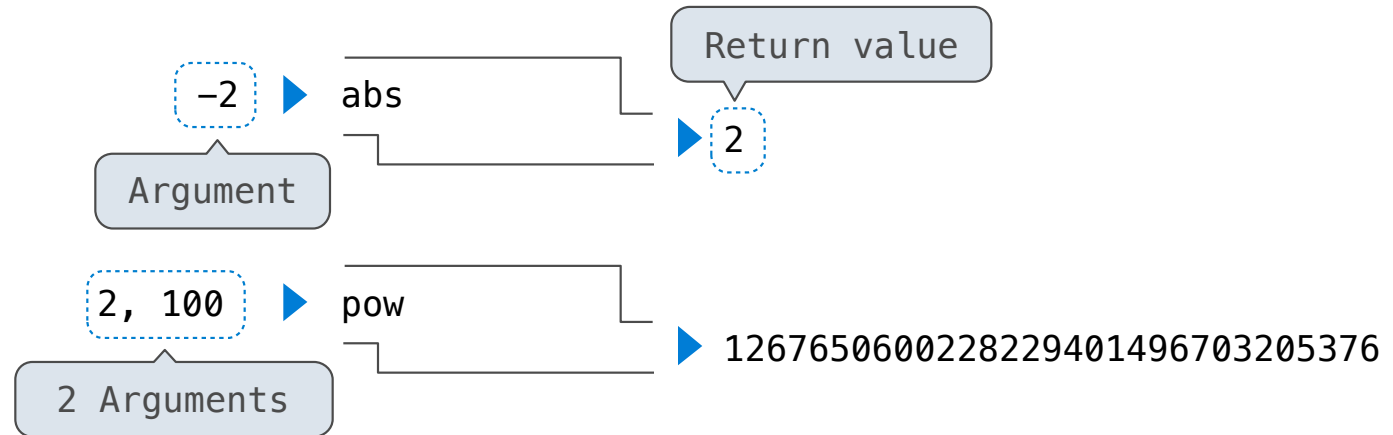
Pure Functions
just return values



Non-Pure Functions
have side effects

Pure Functions & Non-Pure Functions

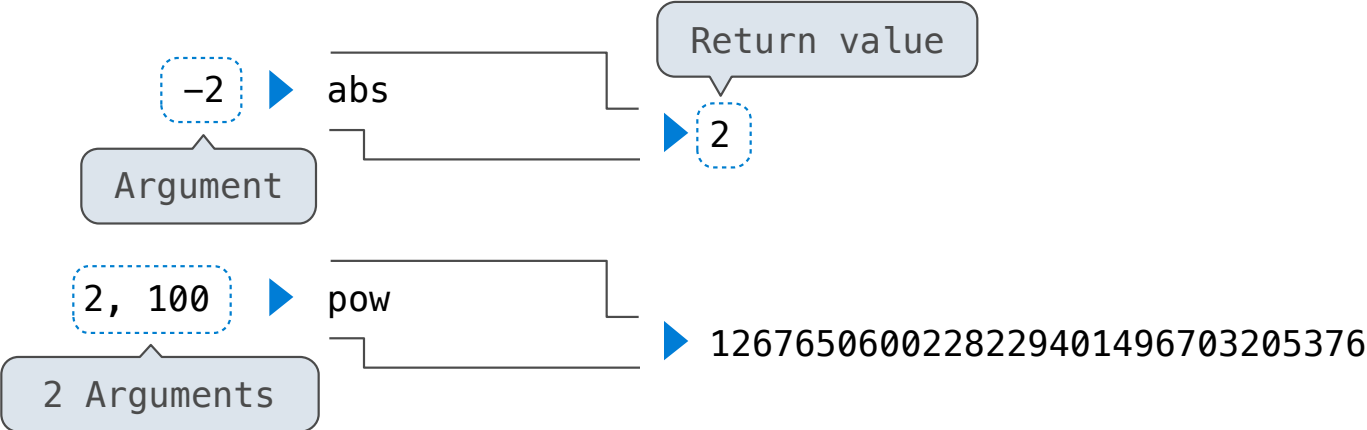
Pure Functions
just return values



Non-Pure Functions
have side effects

Pure Functions & Non-Pure Functions

Pure Functions
just return values

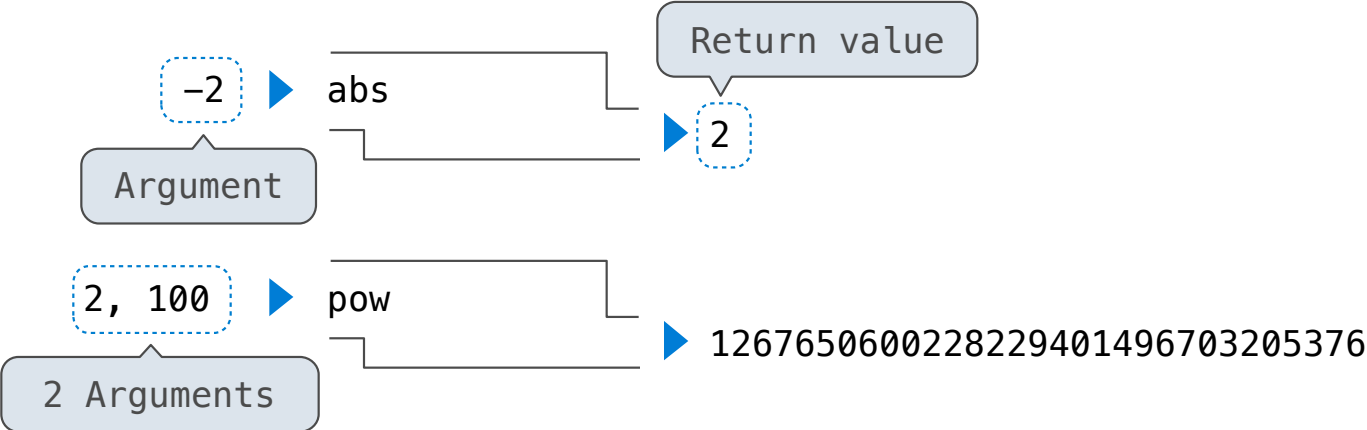


Non-Pure Functions
have side effects



Pure Functions & Non-Pure Functions

Pure Functions
just return values

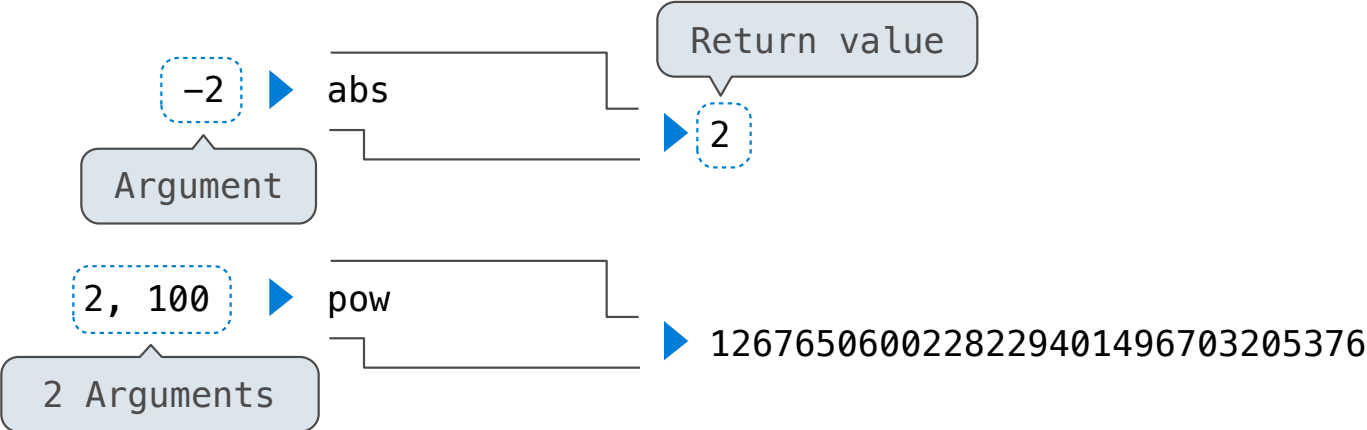


Non-Pure Functions
have side effects

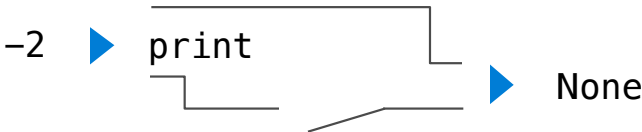


Pure Functions & Non-Pure Functions

Pure Functions
just return values

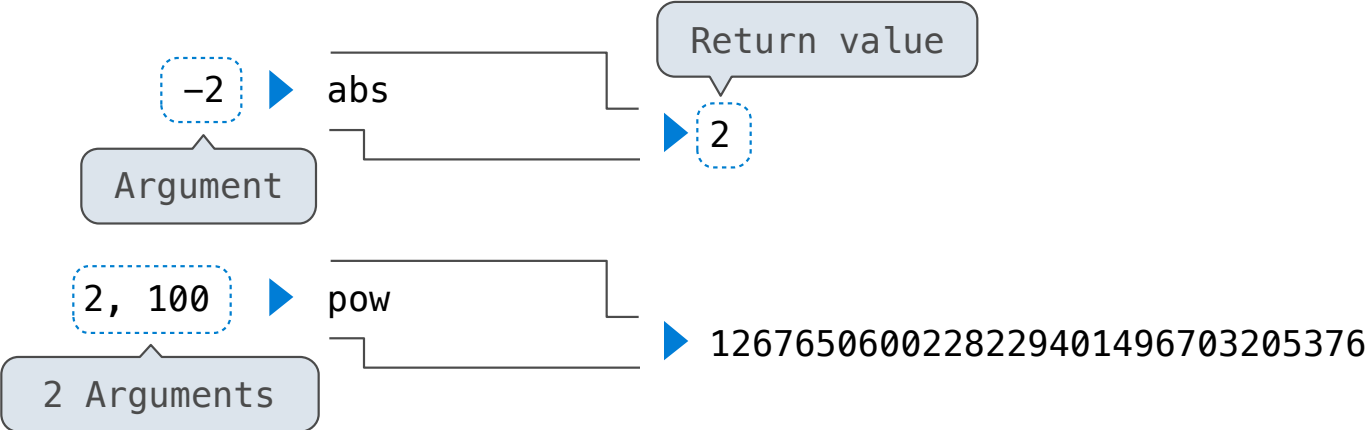


Non-Pure Functions
have side effects

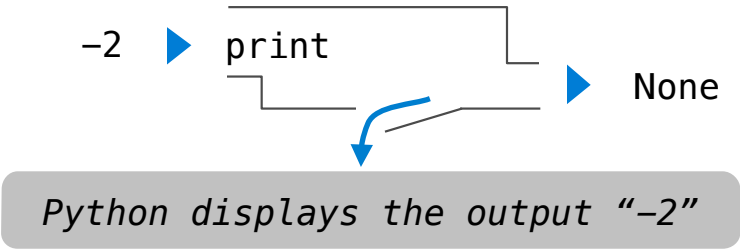


Pure Functions & Non-Pure Functions

Pure Functions
just return values

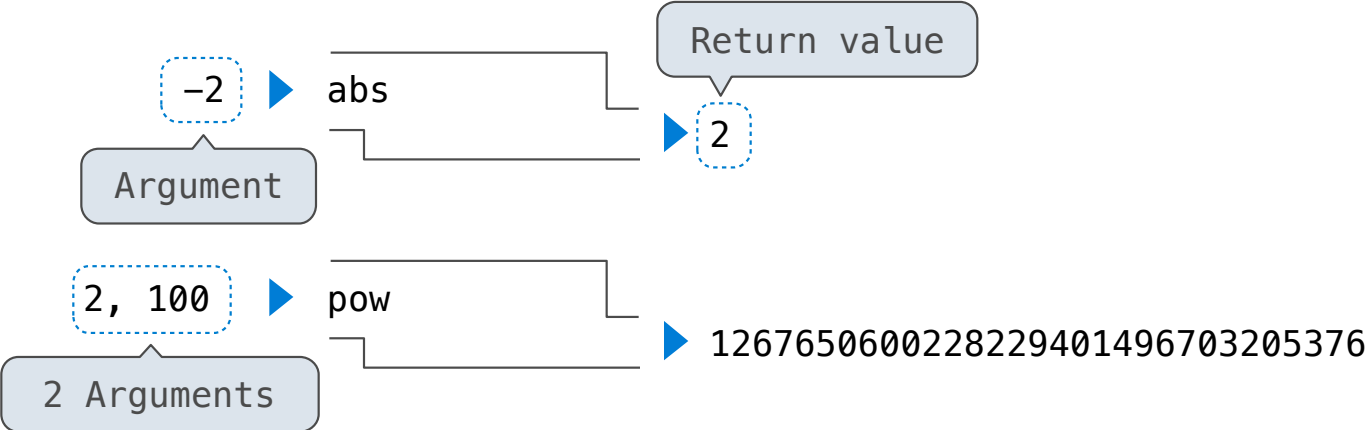


Non-Pure Functions
have side effects

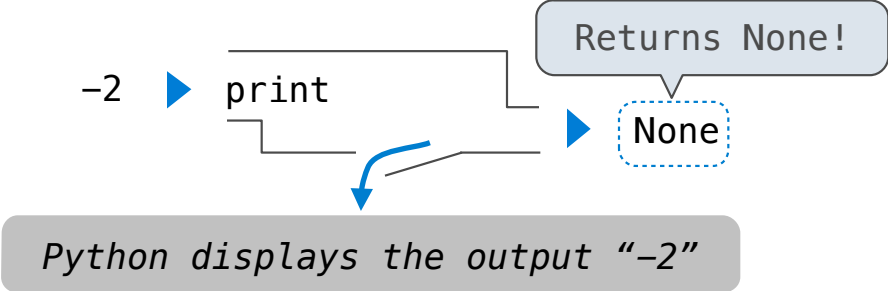


Pure Functions & Non-Pure Functions

Pure Functions
just return values

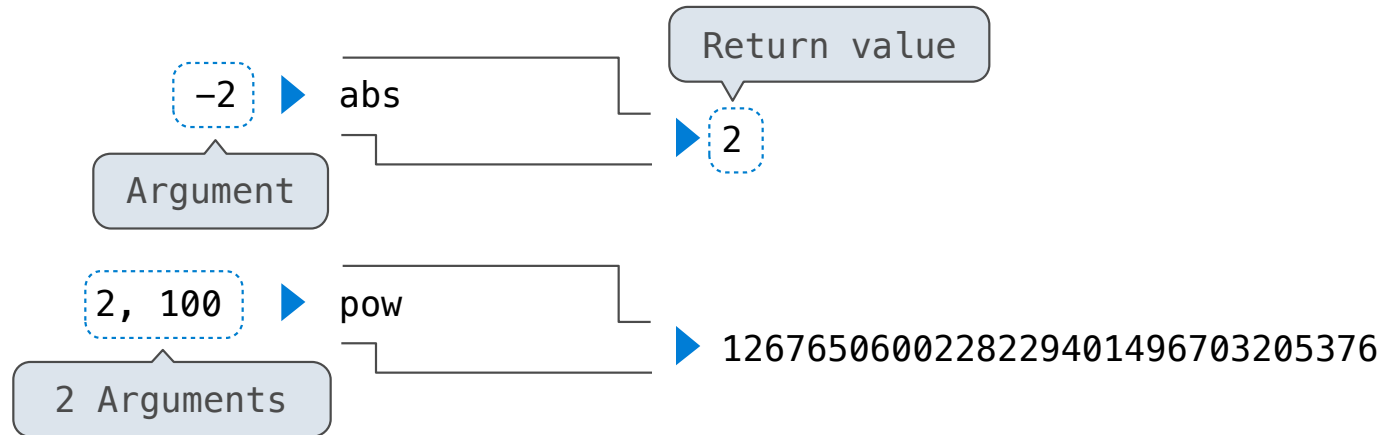


Non-Pure Functions
have side effects

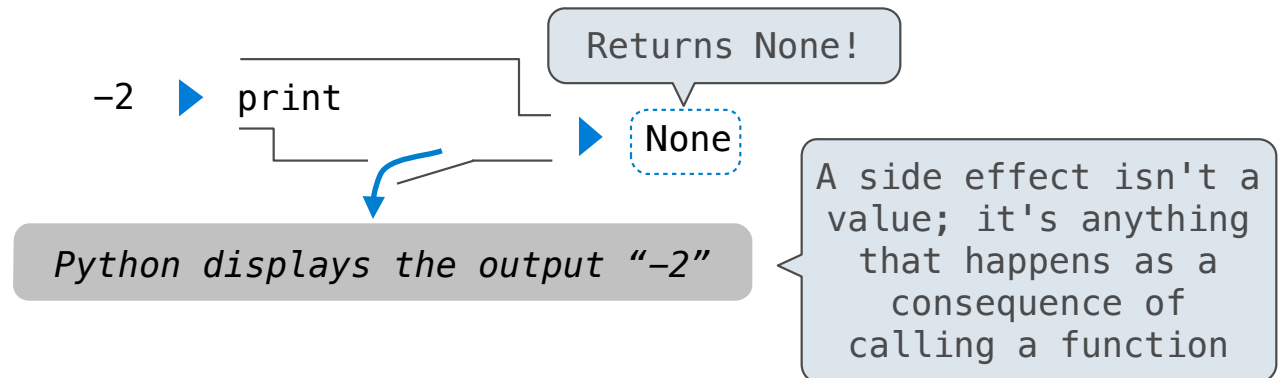


Pure Functions & Non-Pure Functions

Pure Functions
just return values



Non-Pure Functions
have side effects



Nested Expressions with Print

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

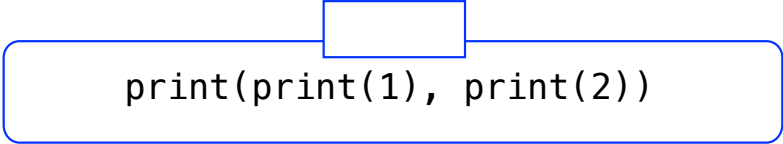
Nested Expressions with Print

```
>>> 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
```

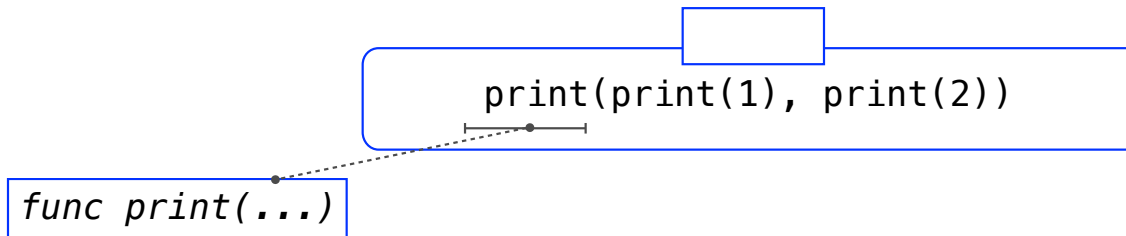


The diagram illustrates the execution of the nested print function call. It features a large rounded rectangular box containing the code `print(print(1), print(2))`. A smaller, empty rectangular box is positioned above the first `print(1)` argument. A vertical line connects the top of this smaller box to the `print(1)` argument, indicating the flow of data from the inner function call to the outer one.

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

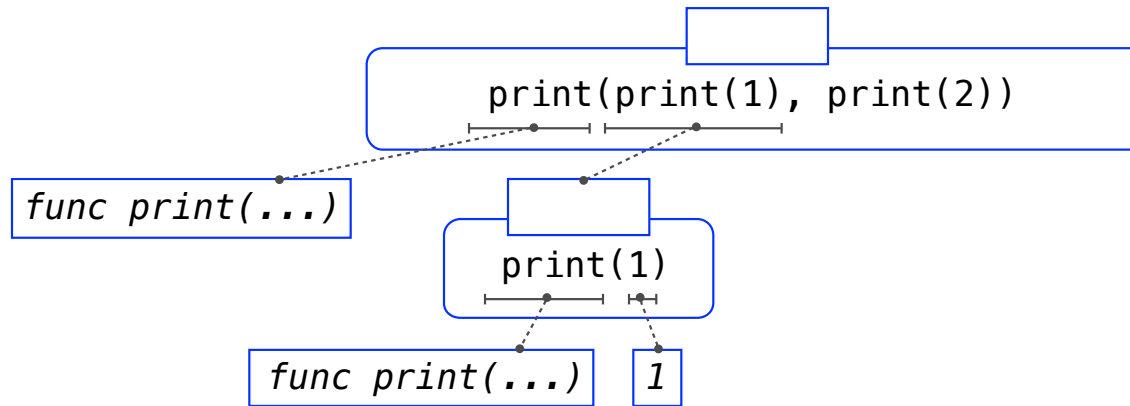

Nested Expressions with Print

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



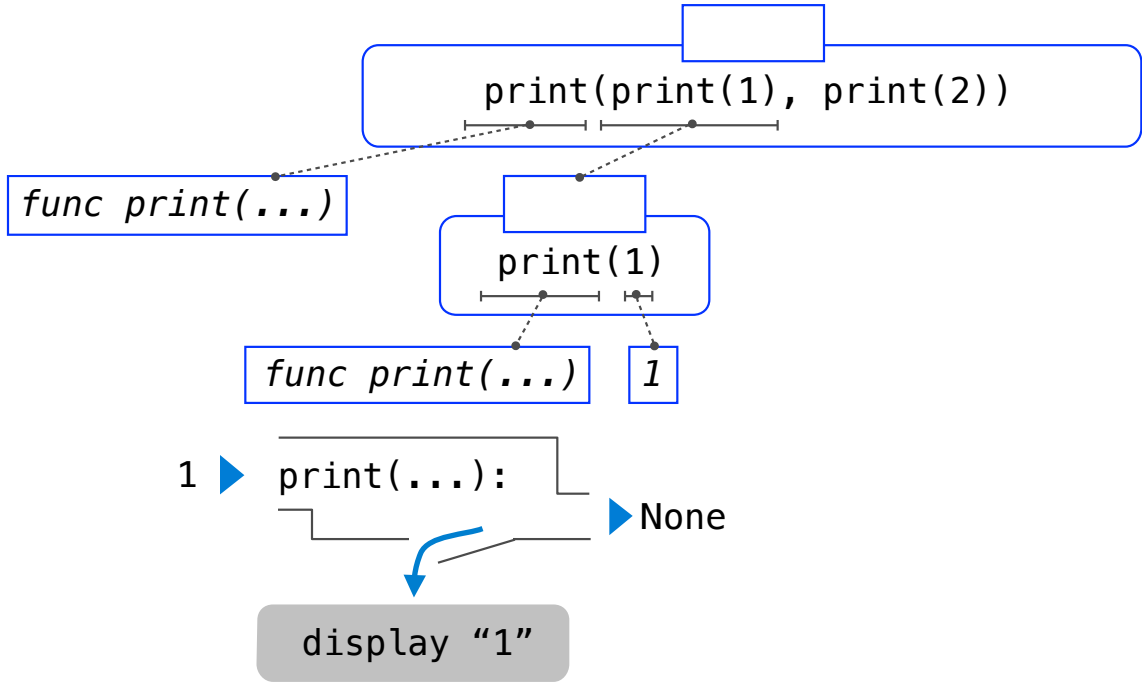
Nested Expressions with Print

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



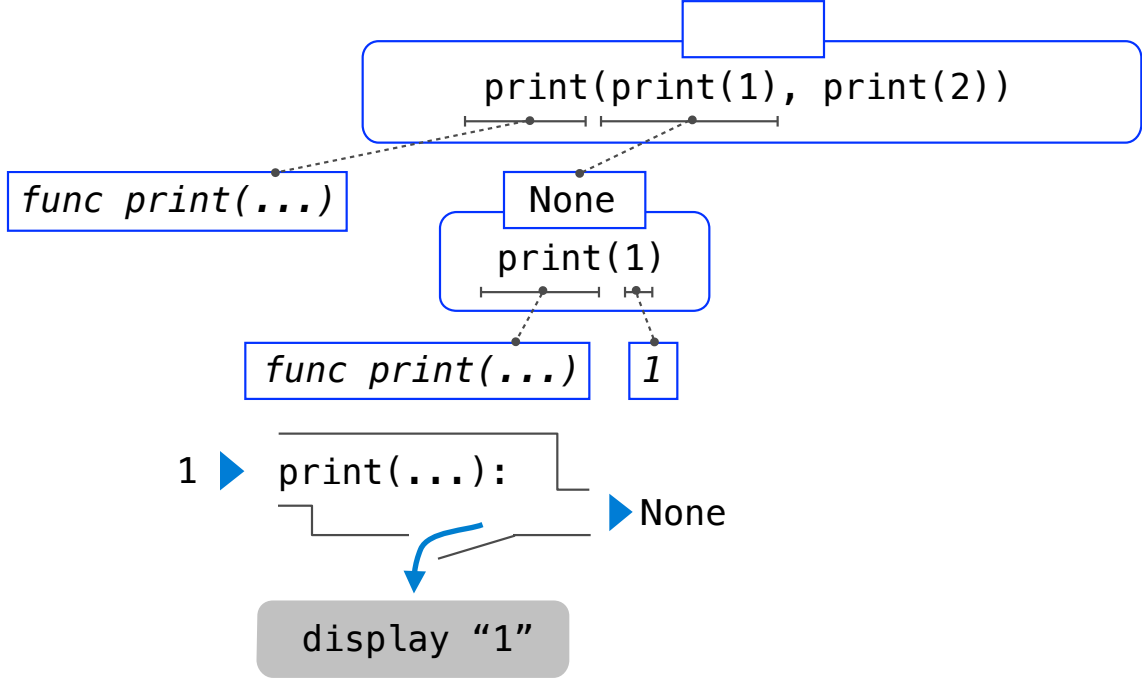
Nested Expressions with Print

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



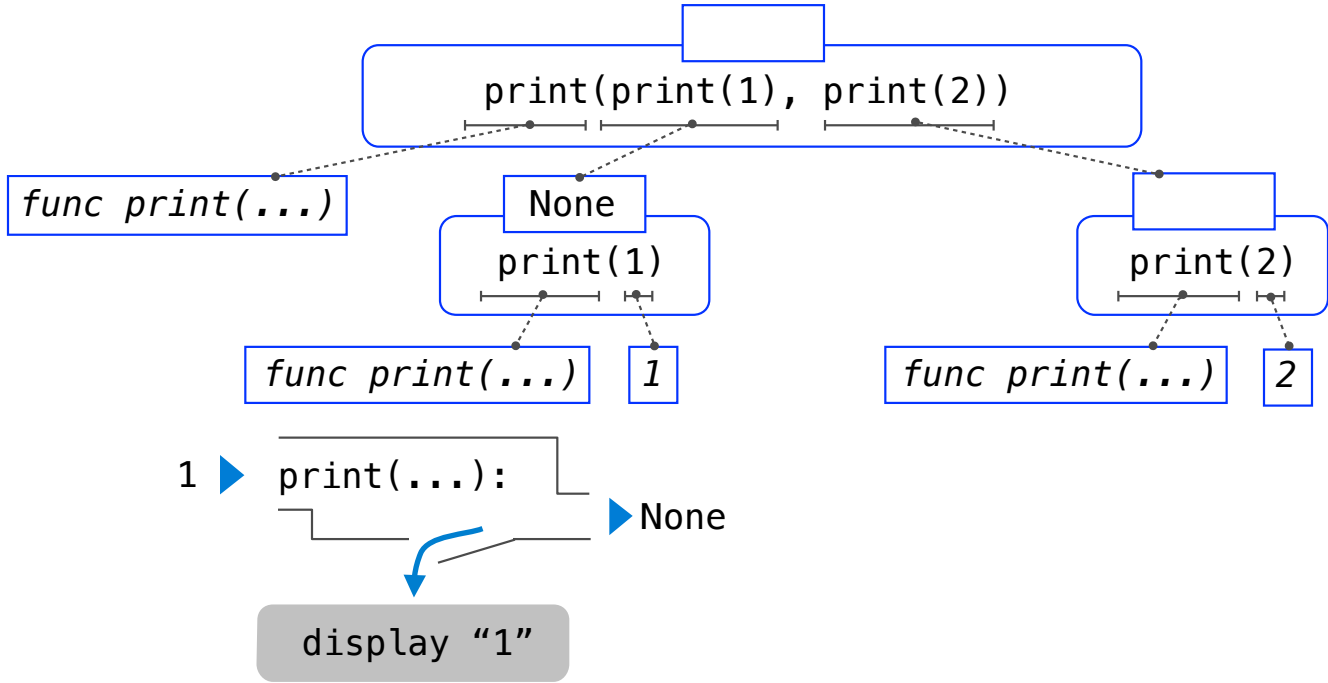
Nested Expressions with Print

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



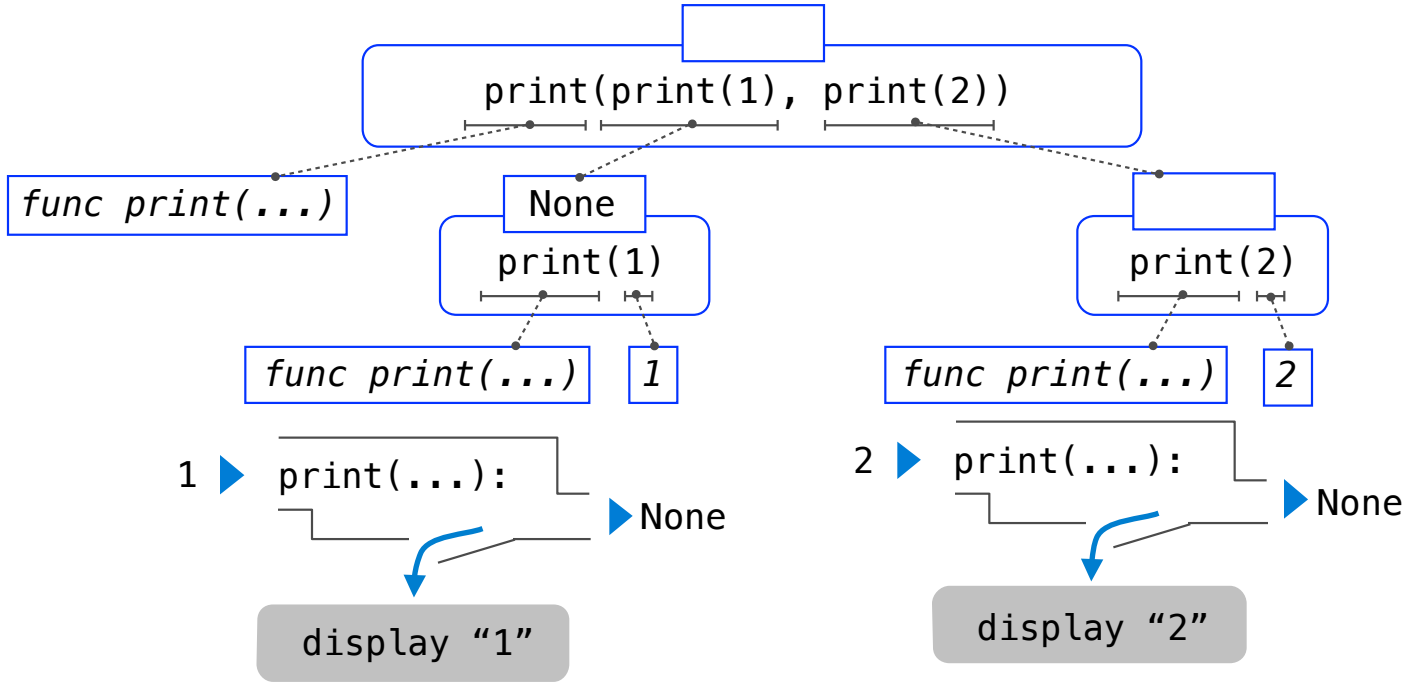
Nested Expressions with Print

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



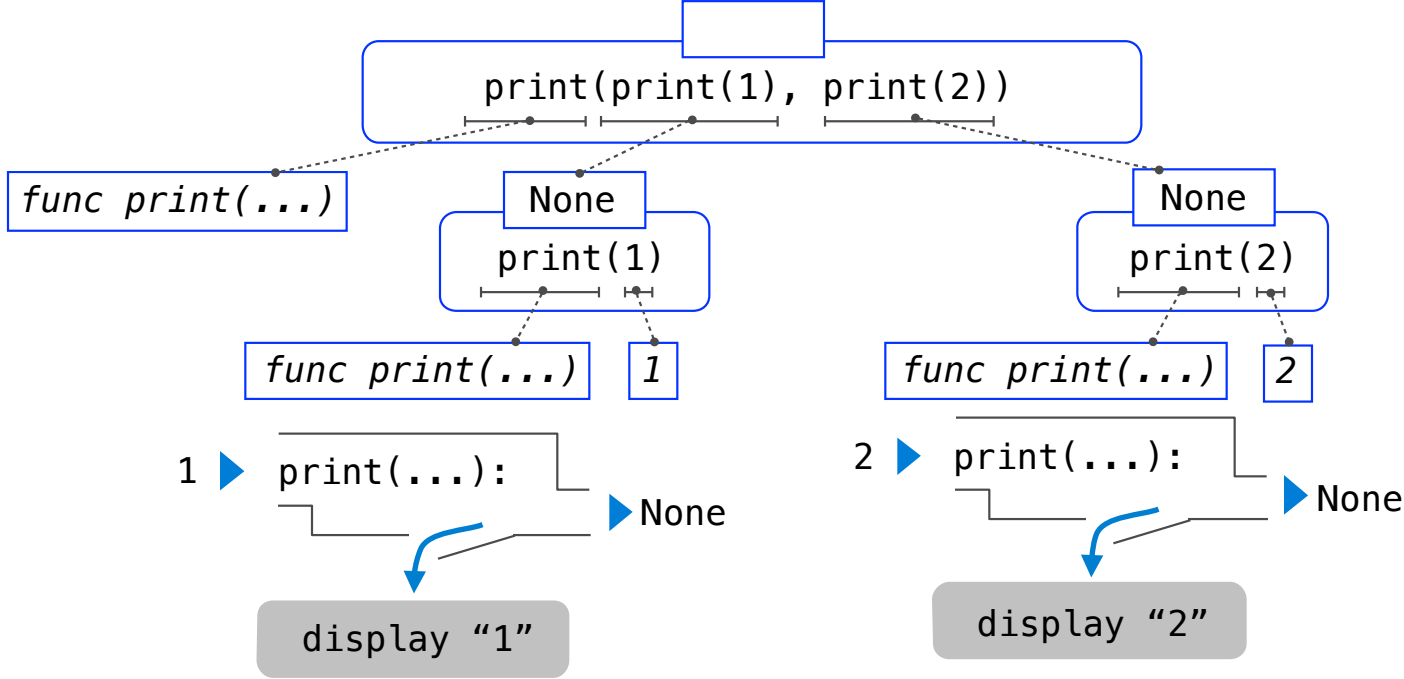
Nested Expressions with Print

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

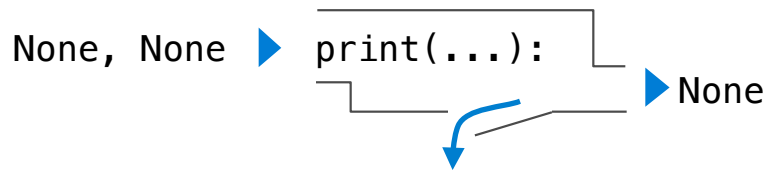


Nested Expressions with Print

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

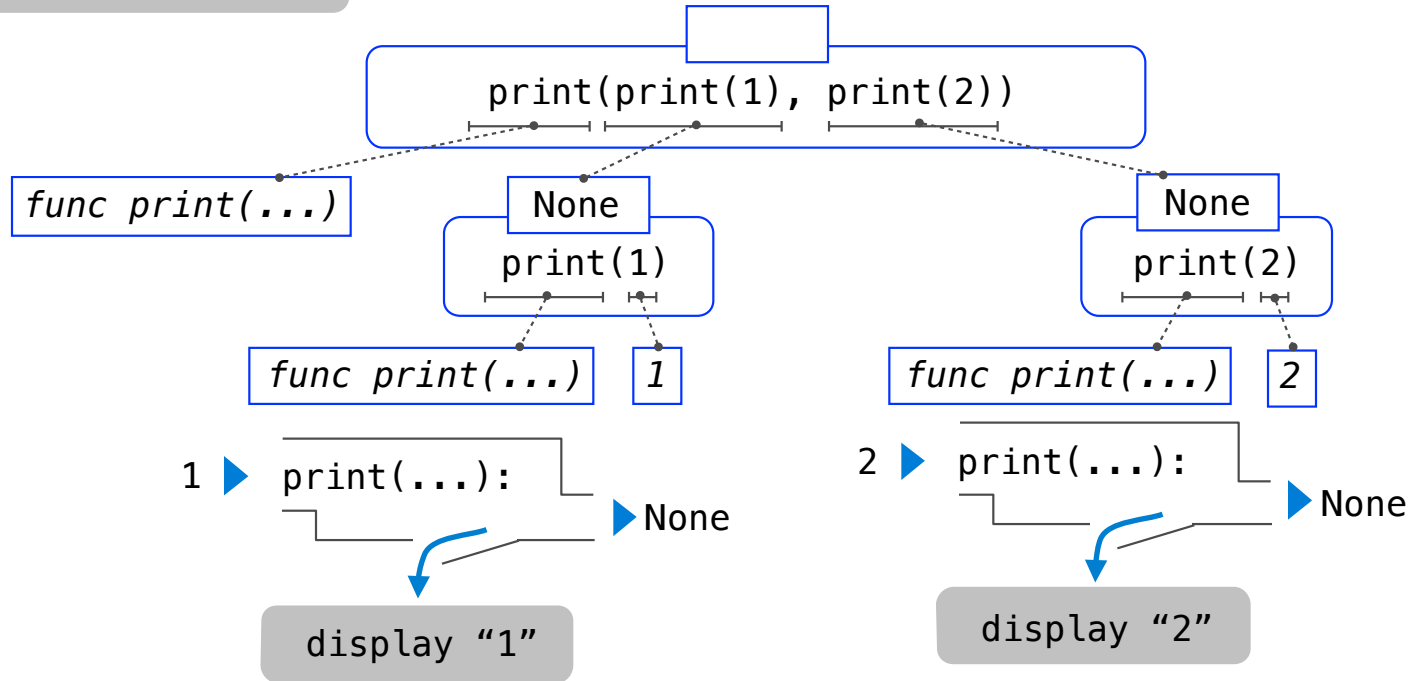


Nested Expressions with Print

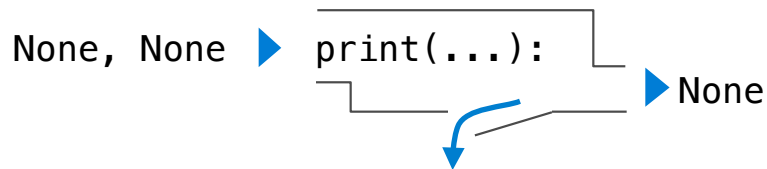


display "None None"

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

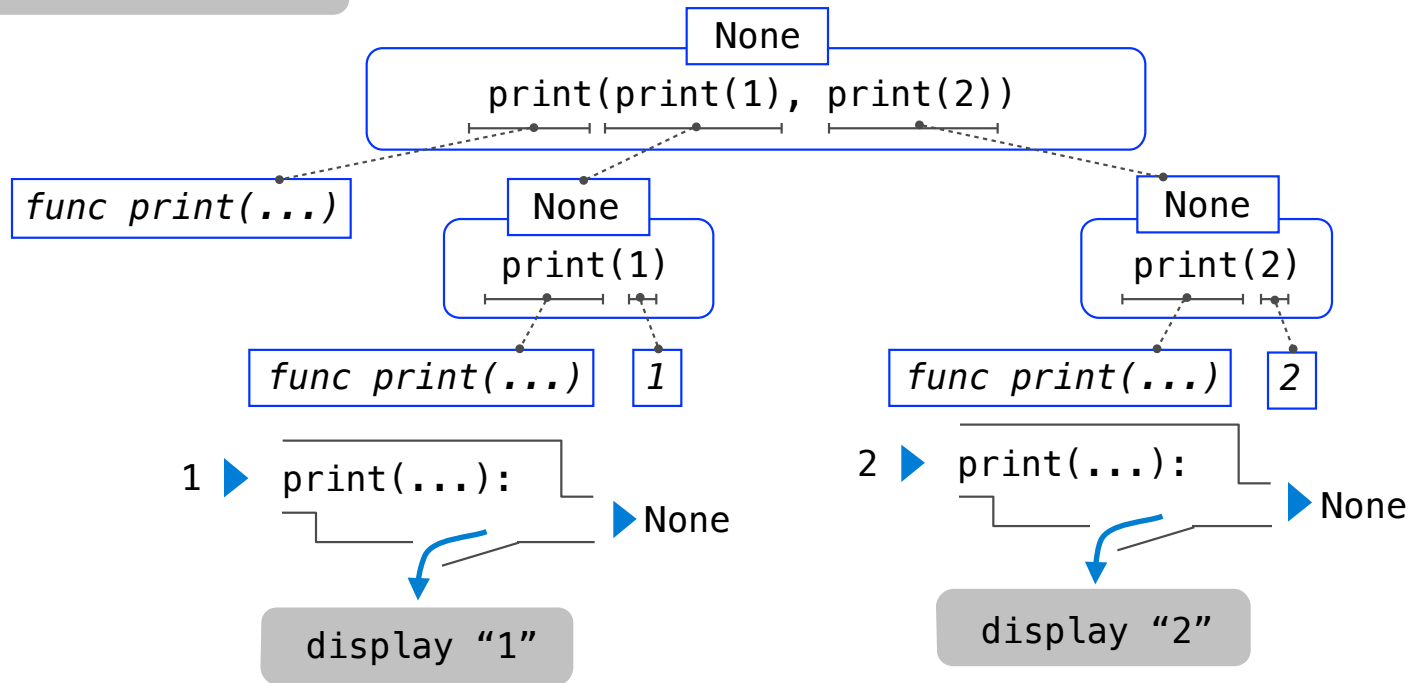


Nested Expressions with Print

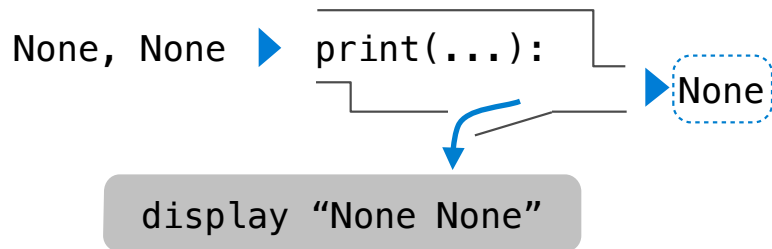


display "None None"

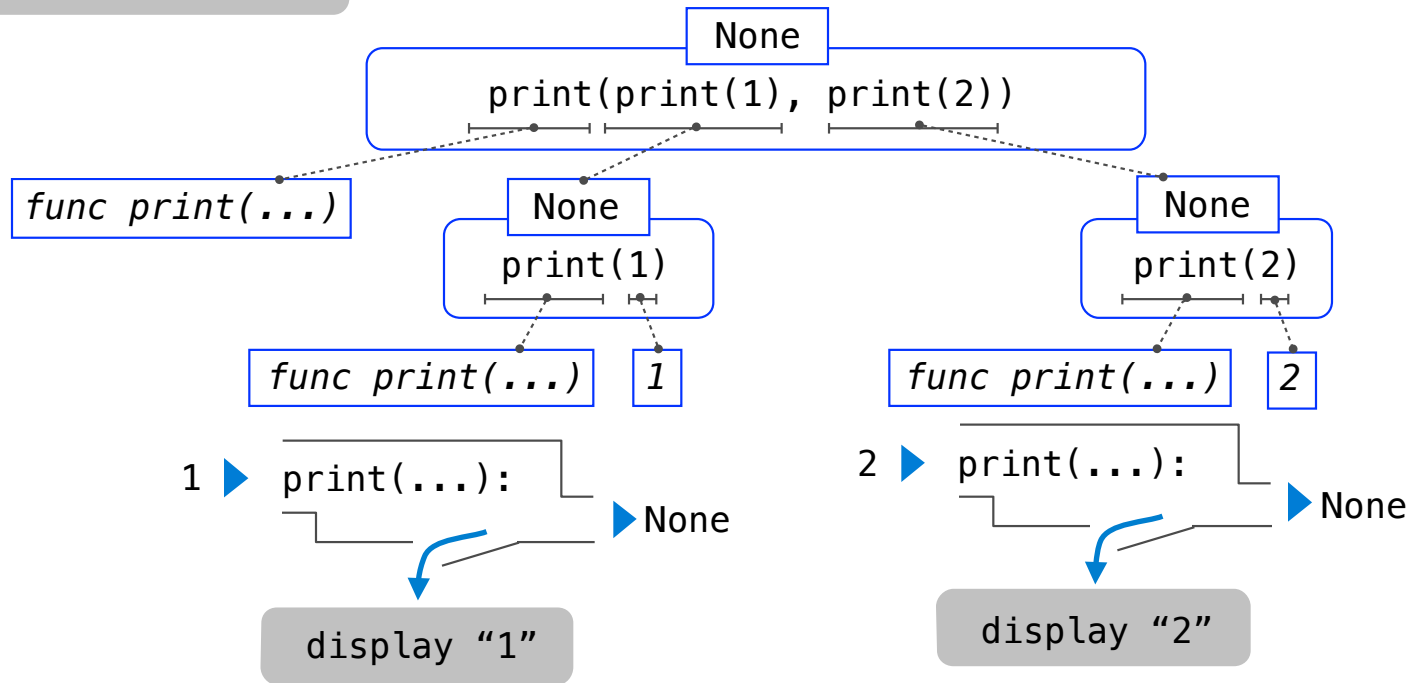
```
>>> print(print(1), print(2))
1
2
None None
```



Nested Expressions with Print



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



Nested Expressions with Print

