

61A Lecture 2

Friday, January 23, 2015

61A Lecture 2

Thursday, January 22, 2015

Announcements

Announcements

- Starting next week, submitting labs & attending section will provide a midterm safety net

Announcements

- Starting next week, submitting labs & attending section will provide a midterm safety net
- Homework 1 is due next Wednesday 1/28

Announcements

- Starting next week, submitting labs & attending section will provide a midterm safety net
- Homework 1 is due next Wednesday 1/28
 - All homework is graded on effort; you must make progress on each problem to earn 2/2

Announcements

- Starting next week, submitting labs & attending section will provide a midterm safety net
- Homework 1 is due next Wednesday 1/28
 - All homework is graded on effort; you must make progress on each problem to earn 2/2
 - Homework Party on Tuesday 1/27 5–6:30pm in 2050 VLSB

Announcements

- Starting next week, submitting labs & attending section will provide a midterm safety net
- Homework 1 is due next Wednesday 1/28
 - All homework is graded on effort; you must make progress on each problem to earn 2/2
 - Homework Party on Tuesday 1/27 5–6:30pm in 2050 VLSB
- Quiz 1 released next Wednesday 1/28 is due next Thursday 1/29 (graded on correctness)

Announcements

- Starting next week, submitting labs & attending section will provide a midterm safety net
- Homework 1 is due next Wednesday 1/28
 - All homework is graded on effort; you must make progress on each problem to earn 2/2
 - Homework Party on Tuesday 1/27 5–6:30pm in 2050 VLSB
- Quiz 1 released next Wednesday 1/28 is due next Thursday 1/29 (graded on correctness)
- Ask questions about lab and homework assignments in office hours! (cs61a.org/weekly.html)

Announcements

- Starting next week, submitting labs & attending section will provide a midterm safety net
- Homework 1 is due next Wednesday 1/28
 - All homework is graded on effort; you must make progress on each problem to earn 2/2
 - Homework Party on Tuesday 1/27 5–6:30pm in 2050 VLSB
- Quiz 1 released next Wednesday 1/28 is due next Thursday 1/29 (graded on correctness)
- Ask questions about lab and homework assignments in office hours! (cs61a.org/weekly.html)
 - 2 locations in Bechtel Engineering Center (Map: <http://goo.gl/dAcHXf>)

Announcements

- Starting next week, submitting labs & attending section will provide a midterm safety net
- Homework 1 is due next Wednesday 1/28
 - All homework is graded on effort; you must make progress on each problem to earn 2/2
 - Homework Party on Tuesday 1/27 5–6:30pm in 2050 VLSB
- Quiz 1 released next Wednesday 1/28 is due next Thursday 1/29 (graded on correctness)
- Ask questions about lab and homework assignments in office hours! (cs61a.org/weekly.html)
 - 2 locations in Bechtel Engineering Center (Map: <http://goo.gl/dAcHXf>)
 - 11–2 & 3–5 on Monday, 11–6 on Tuesday & Thursday, 11–2 & 3–4 on Wednesday, 11–1 on Friday

Announcements

- Starting next week, submitting labs & attending section will provide a midterm safety net
- Homework 1 is due next Wednesday 1/28
 - All homework is graded on effort; you must make progress on each problem to earn 2/2
 - Homework Party on Tuesday 1/27 5–6:30pm in 2050 VLSB
- Quiz 1 released next Wednesday 1/28 is due next Thursday 1/29 (graded on correctness)
- Ask questions about lab and homework assignments in office hours! (cs61a.org/weekly.html)
 - 2 locations in Bechtel Engineering Center (Map: <http://goo.gl/dAcHXf>)
 - 11–2 & 3–5 on Monday, 11–6 on Tuesday & Thursday, 11–2 & 3–4 on Wednesday, 11–1 on Friday
- You **need** to register a class account (Lab 0); that's how we track assignments

Announcements

- Starting next week, submitting labs & attending section will provide a midterm safety net
- Homework 1 is due next Wednesday 1/28
 - All homework is graded on effort; you must make progress on each problem to earn 2/2
 - Homework Party on Tuesday 1/27 5–6:30pm in 2050 VLSB
- Quiz 1 released next Wednesday 1/28 is due next Thursday 1/29 (graded on correctness)
- Ask questions about lab and homework assignments in office hours! (cs61a.org/weekly.html)
 - 2 locations in Bechtel Engineering Center (Map: <http://goo.gl/dAcHXf>)
 - 11–2 & 3–5 on Monday, 11–6 on Tuesday & Thursday, 11–2 & 3–4 on Wednesday, 11–1 on Friday
- You **need** to register a class account (Lab 0); that's how we track assignments
 - Please register even if you're on the waitlist or applying for concurrent enrollment

Names, Assignment, and User-Defined Functions

(Demo)

Types of Expressions

Types of Expressions

Primitive expressions:

Types of Expressions

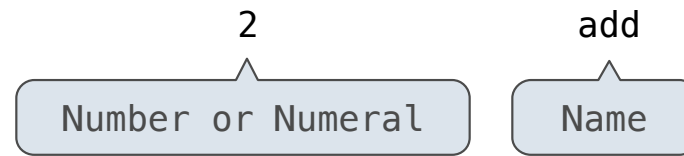
Primitive expressions:

2

Number or Numeral

Types of Expressions

Primitive expressions:



Types of Expressions

Primitive expressions:



Types of Expressions

Primitive expressions:



Call expressions:

Types of Expressions

Primitive expressions:



Call expressions:

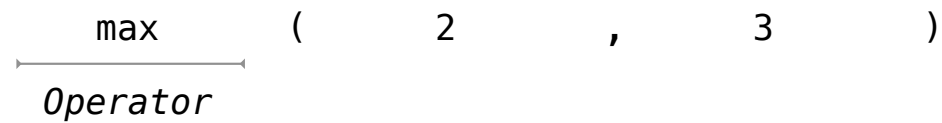
`max (2 , 3)`

Types of Expressions

Primitive expressions:



Call expressions:

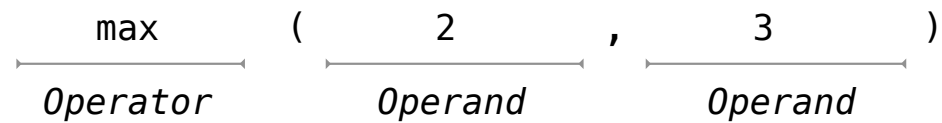


Types of Expressions

Primitive expressions:



Call expressions:

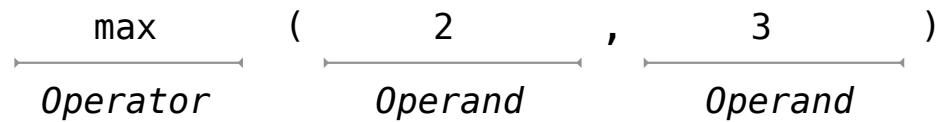


Types of Expressions

Primitive expressions:



Call expressions:



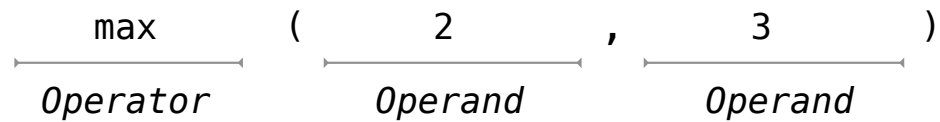
`max(min(pow(3, 5), -4), min(1, -2))`

Types of Expressions

Primitive expressions:



Call expressions:

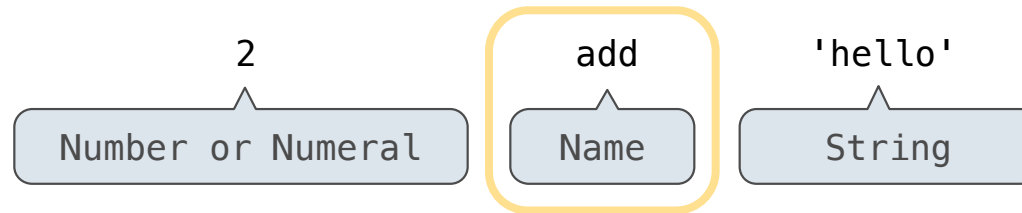


An operand can also
be a call expression

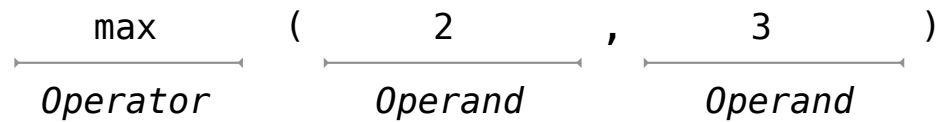
max(min(pow(3, 5), -4), min(1, -2))

Types of Expressions

Primitive expressions:



Call expressions:



An operand can also be a call expression

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

final expression

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?

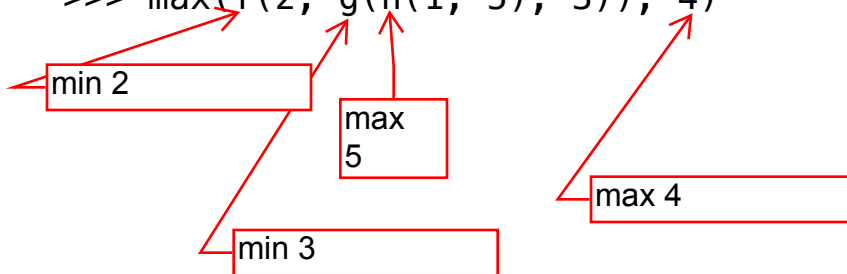
```
>>> f = min
```

```
>>> 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
```

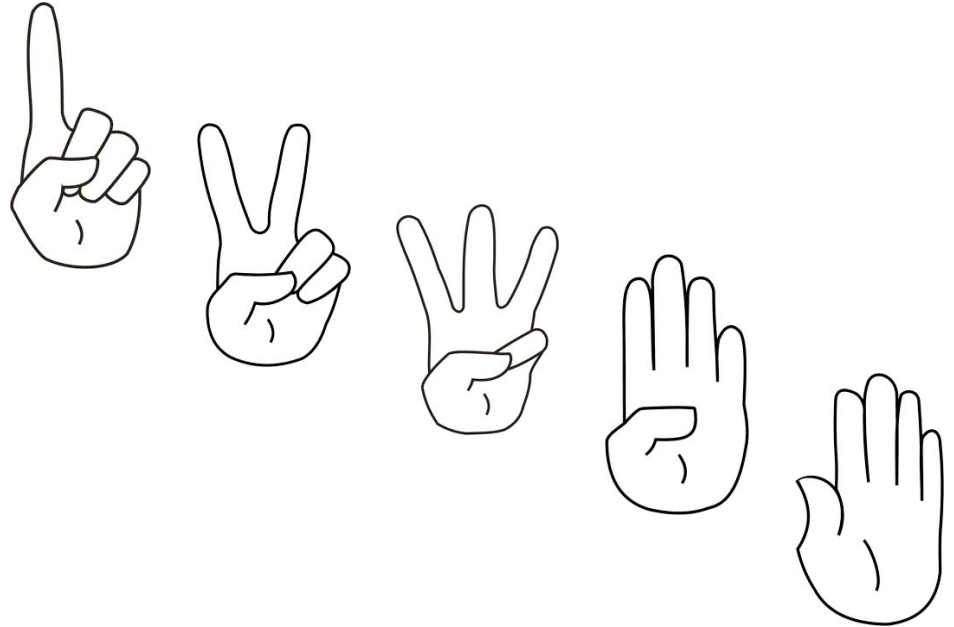
```
>>> f = max
```

```
>>> 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
```

Code (left):

Global frame	
pi	3.1416

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):

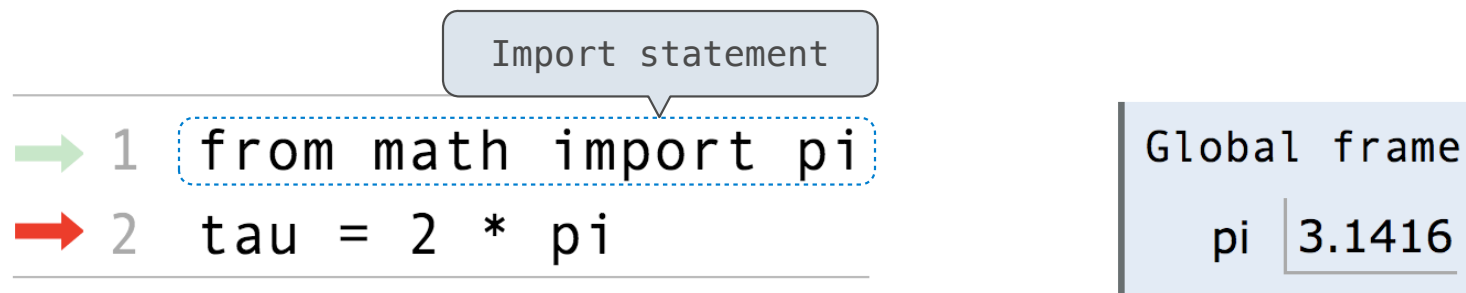
Statements and expressions

Frames (right):

Interactive Diagram

Environment Diagrams

Environment diagrams visualize the interpreter's process.



Code (left):

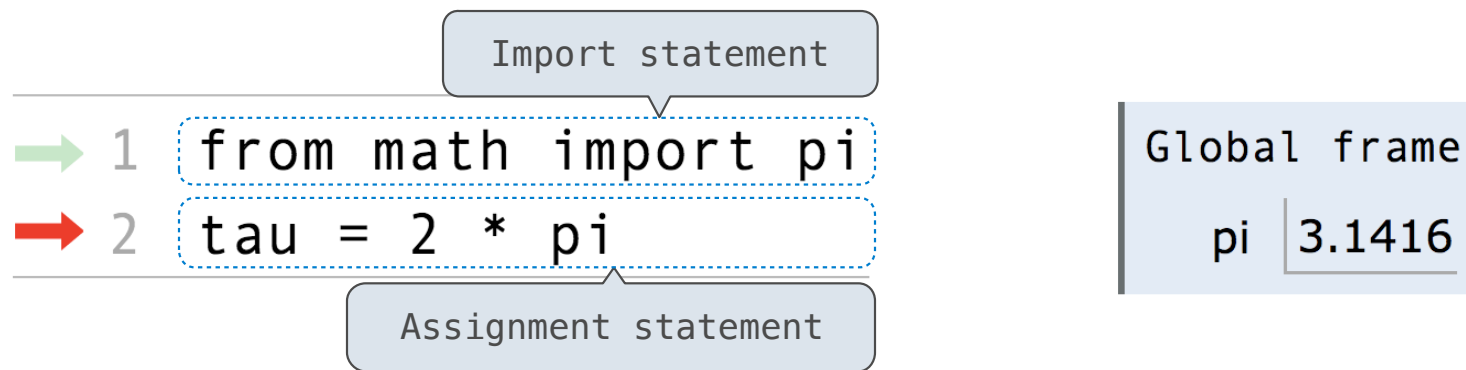
Statements and expressions

Frames (right):

Interactive Diagram

Environment Diagrams

Environment diagrams visualize the interpreter's process.



Code (left):

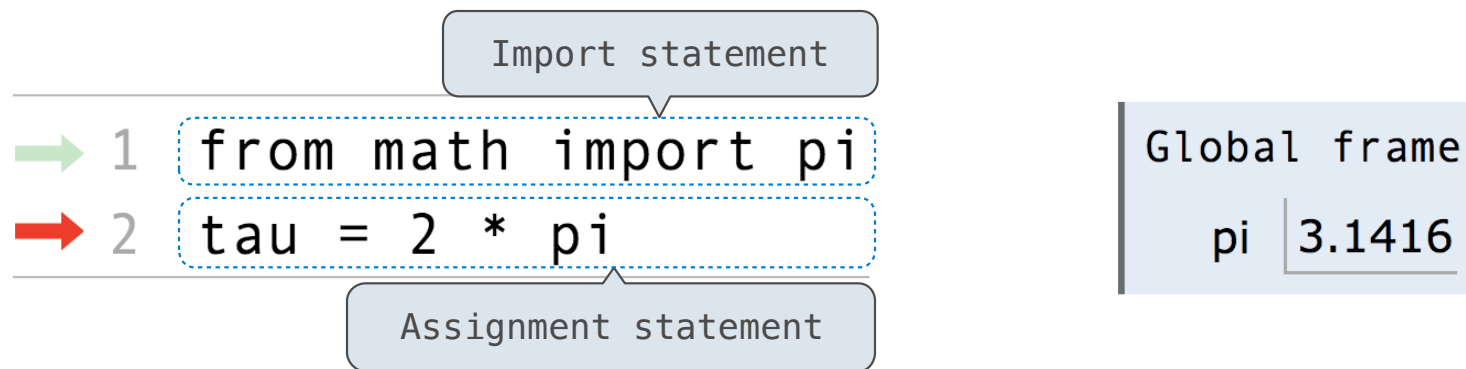
Statements and expressions

Frames (right):

Interactive Diagram

Environment Diagrams

Environment diagrams visualize the interpreter's process.



Code (left):

Statements and expressions

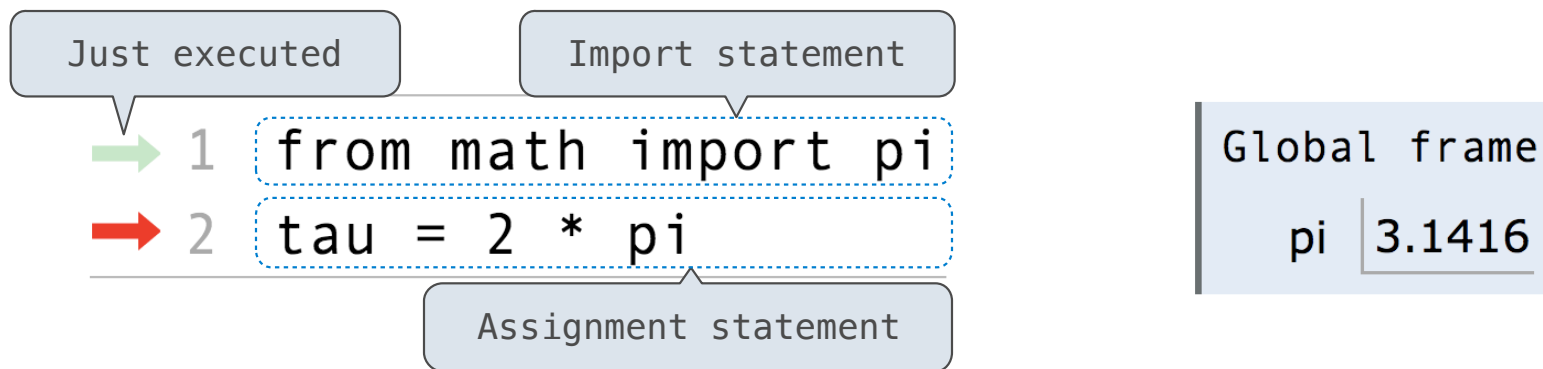
Arrows indicate evaluation order

Frames (right):

Interactive Diagram

Environment Diagrams

Environment diagrams visualize the interpreter's process.



Code (left):

Statements and expressions

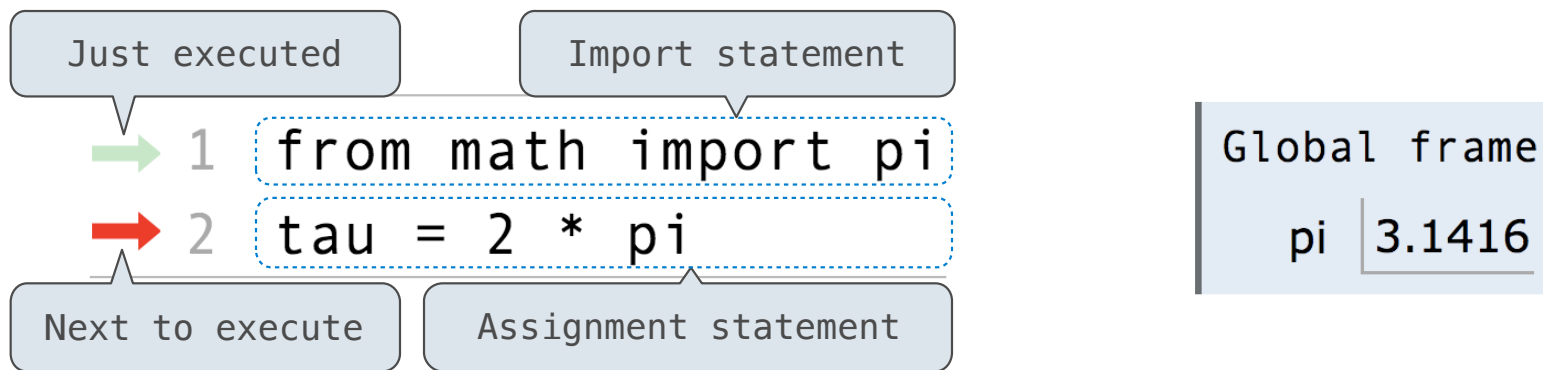
Arrows indicate evaluation order

Frames (right):

Interactive Diagram

Environment Diagrams

Environment diagrams visualize the interpreter's process.



Code (left):

Statements and expressions

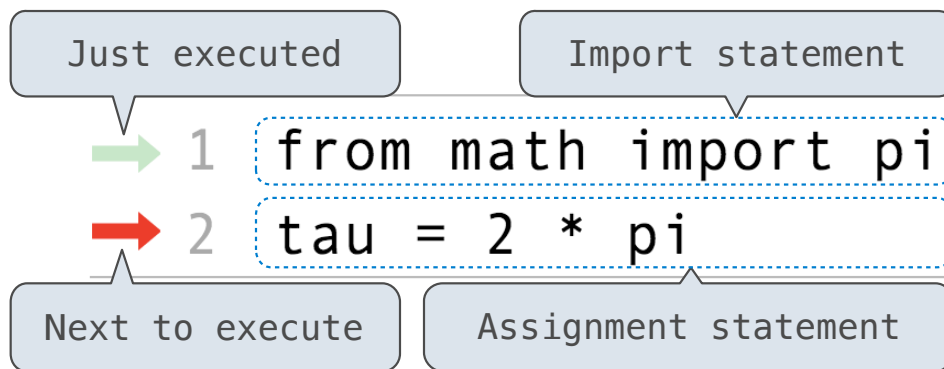
Arrows indicate evaluation order

Frames (right):

Interactive Diagram

Environment Diagrams

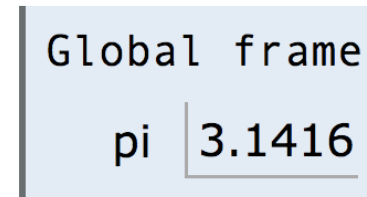
Environment diagrams visualize the interpreter's process.



Code (left):

Statements and expressions

Arrows indicate evaluation order



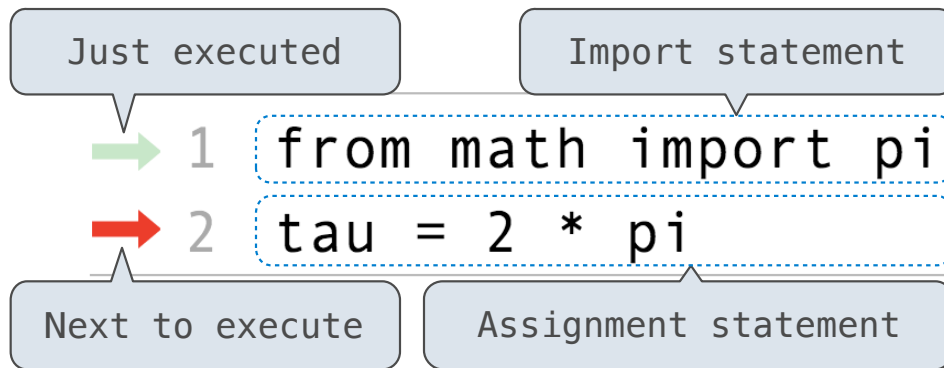
Frames (right):

Each name is bound to a value

Interactive Diagram

Environment Diagrams

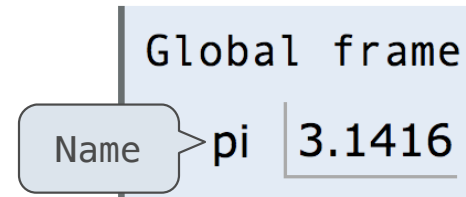
Environment diagrams visualize the interpreter's process.



Code (left):

Statements and expressions

Arrows indicate evaluation order



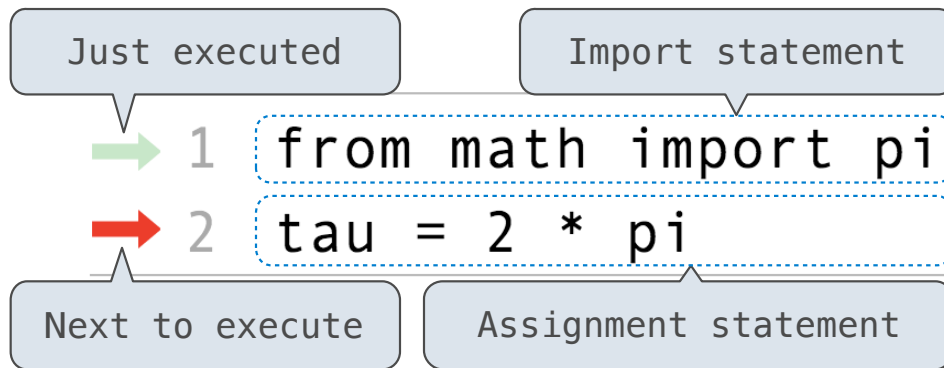
Frames (right):

Each name is bound to a value

Interactive Diagram

Environment Diagrams

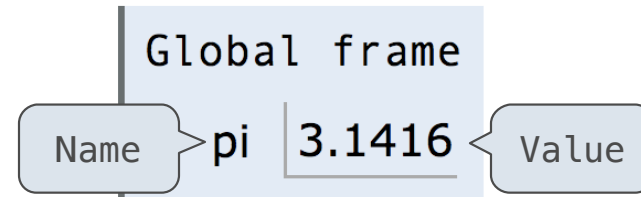
Environment diagrams visualize the interpreter's process.



Code (left):

Statements and expressions

Arrows indicate evaluation order



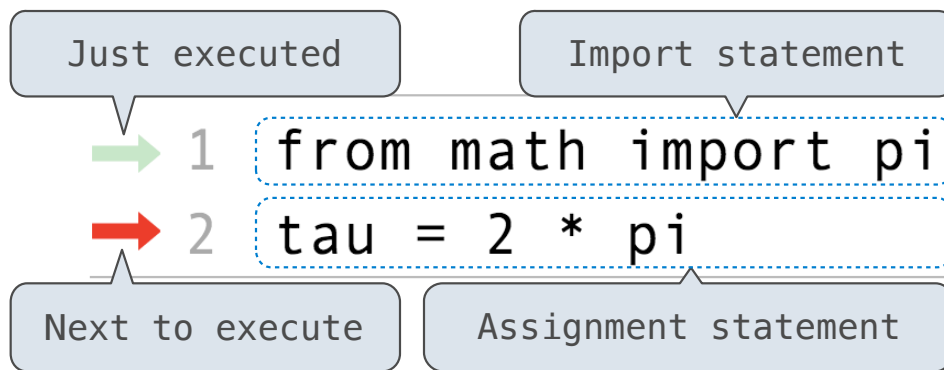
Frames (right):

Each name is bound to a value

Interactive Diagram

Environment Diagrams

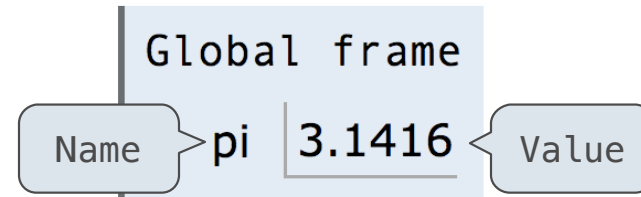
Environment diagrams visualize the interpreter's process.



Code (left):

Statements and expressions

Arrows indicate evaluation order



Frames (right):

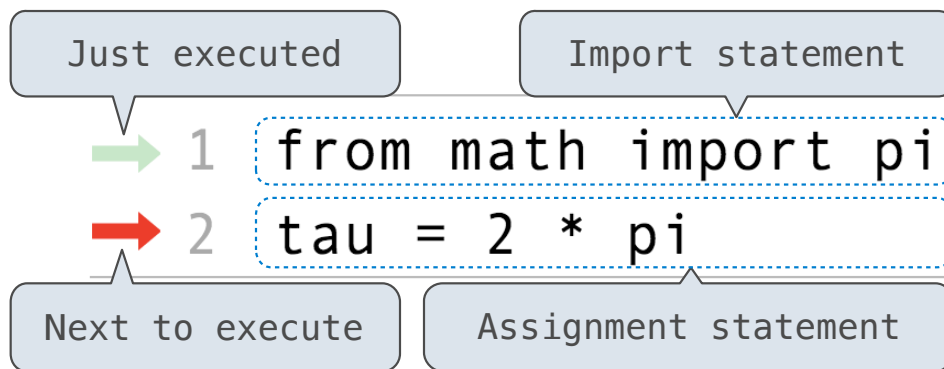
Each name is bound to a value

Within a frame, a name cannot be repeated

Interactive Diagram

Environment Diagrams

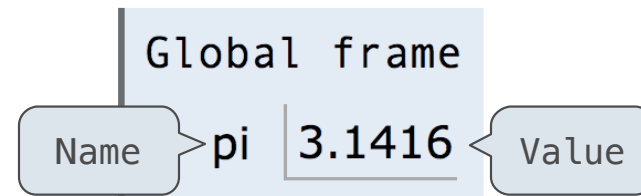
Environment diagrams visualize the interpreter's process.



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

Interactive Diagram

Assignment Statements

Just executed

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

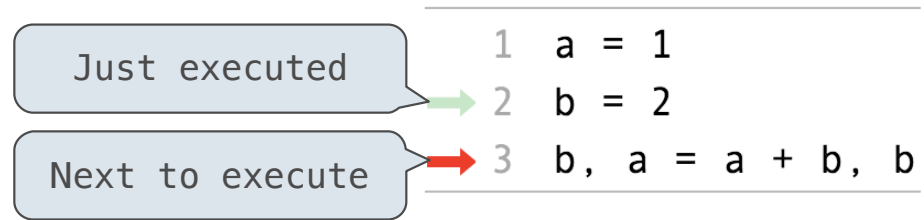
Global frame

a | 1

b | 2

Interactive Diagram

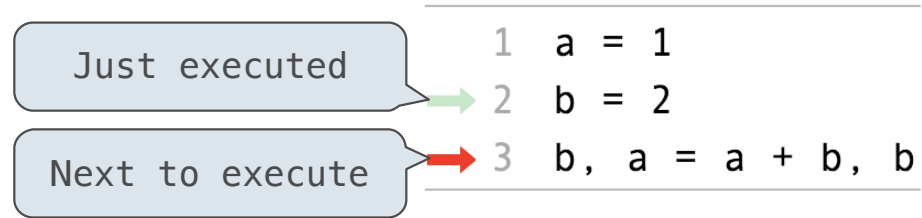
Assignment Statements



Global frame

a	1
b	2

Assignment Statements



Global frame

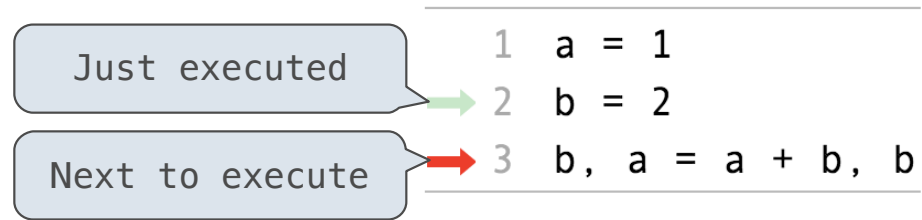
a | 1

b | 2

Execution rule for assignment statements:

Interactive Diagram

Assignment Statements



Global frame

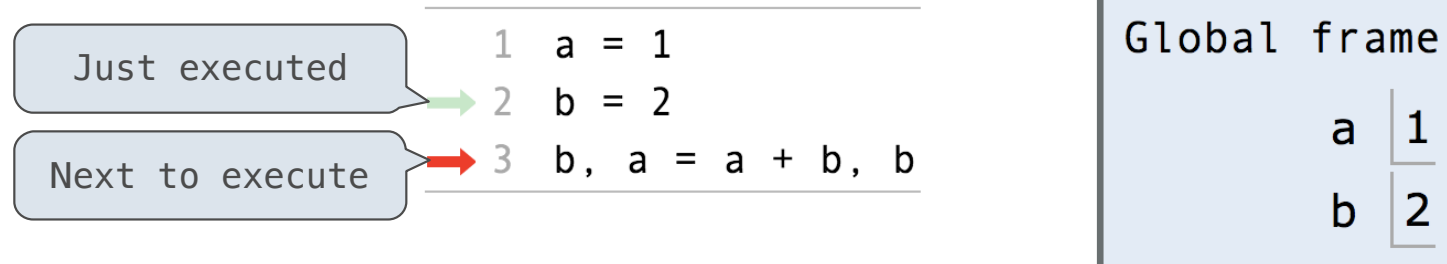
a	1
b	2

Execution rule for assignment statements:

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

Interactive Diagram

Assignment Statements

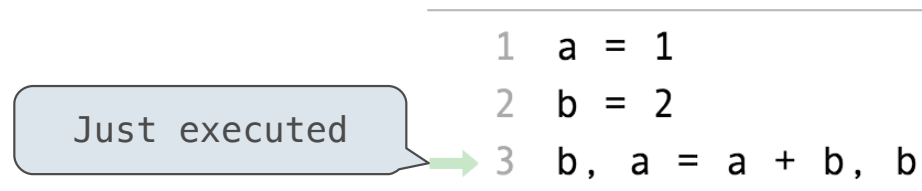
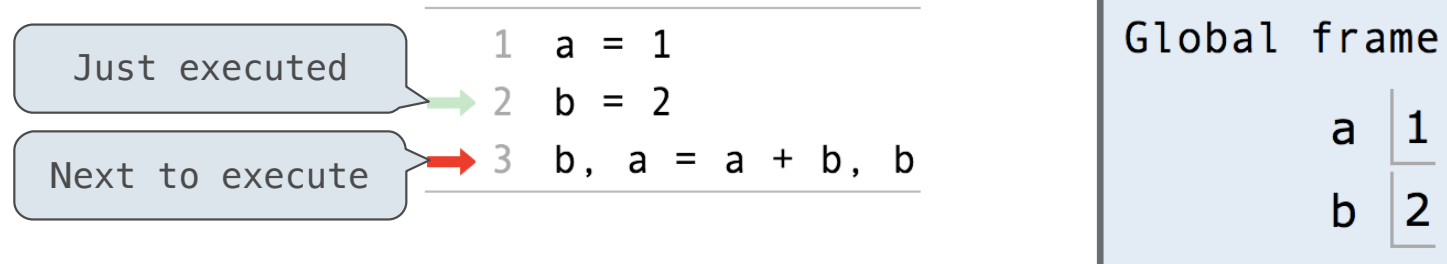


Execution rule for assignment statements:

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.

[Interactive Diagram](#)

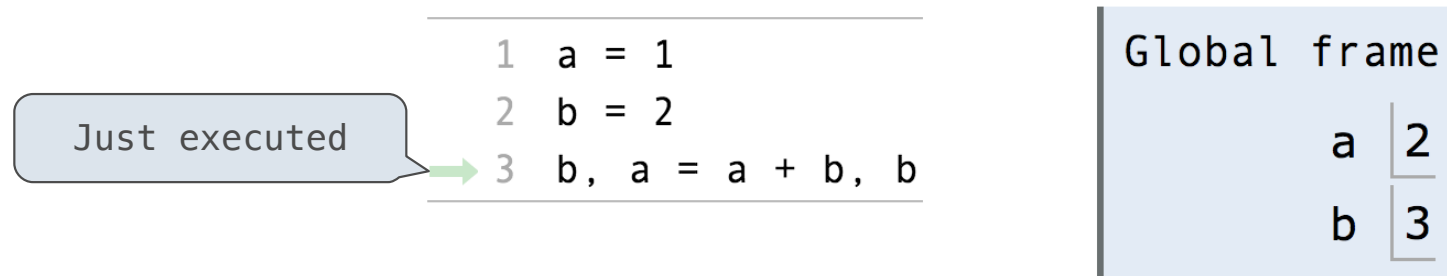
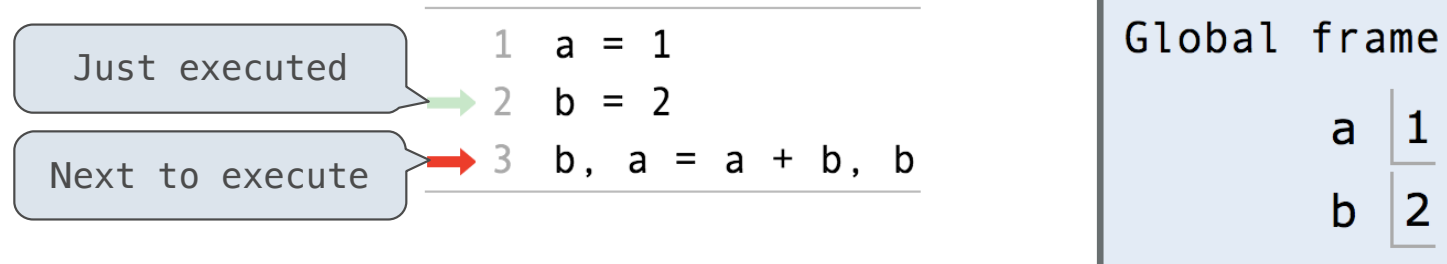
Assignment Statements



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)

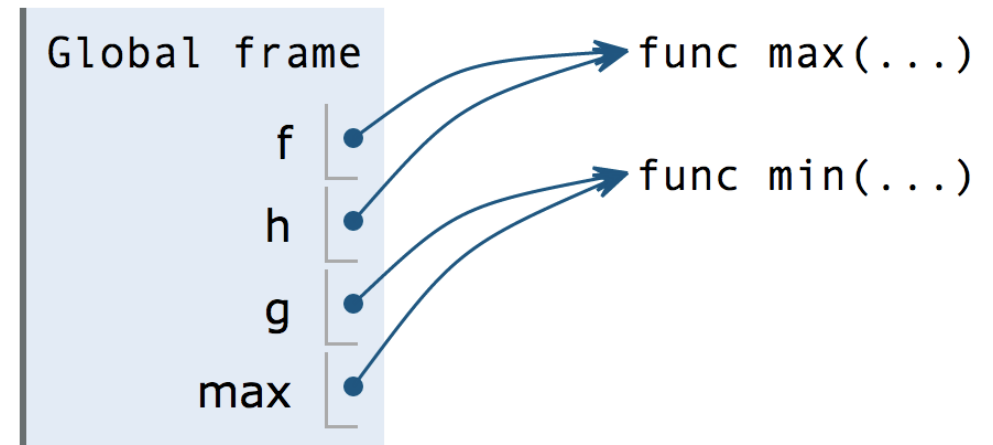
```
1 f = min
2 f = max
3 g, h = min, max
→ 4 max = g
→ 5 max(f(2, g(h(1, 5), 3)), 4)
```

[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)

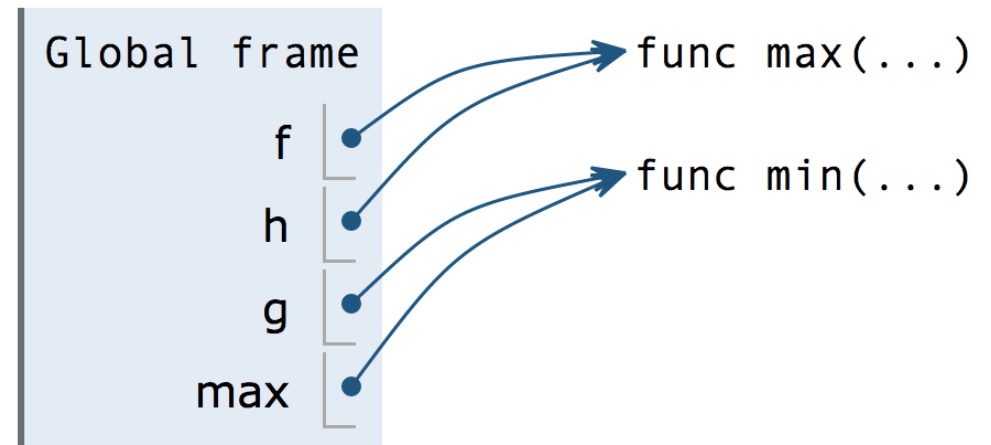


Interactive Diagram

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)



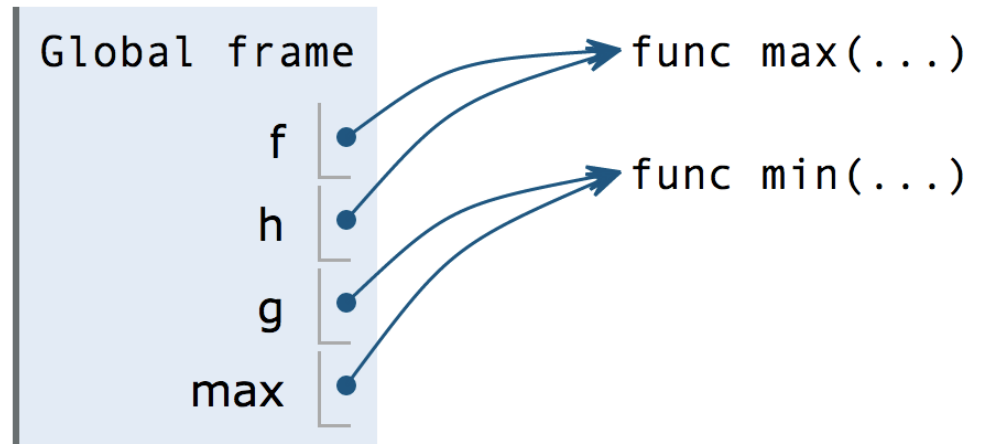
Interactive Diagram

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)
```

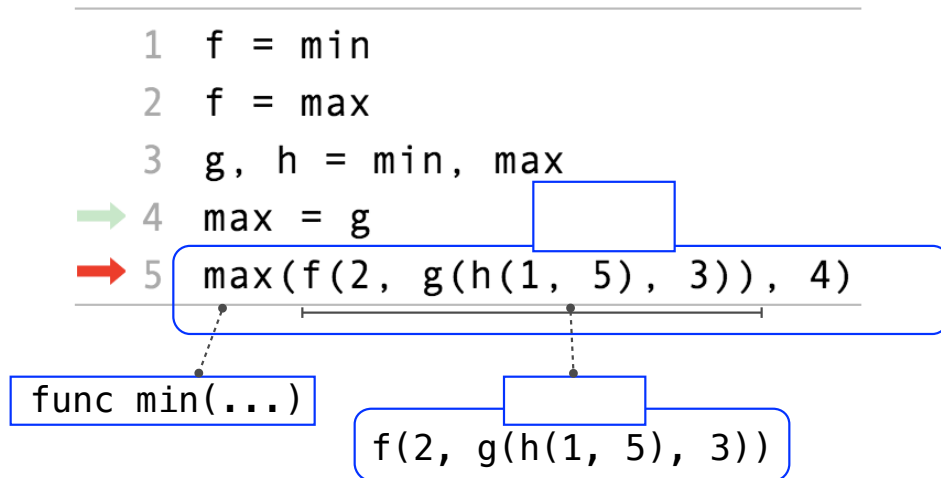
func min(...)

(Demo)

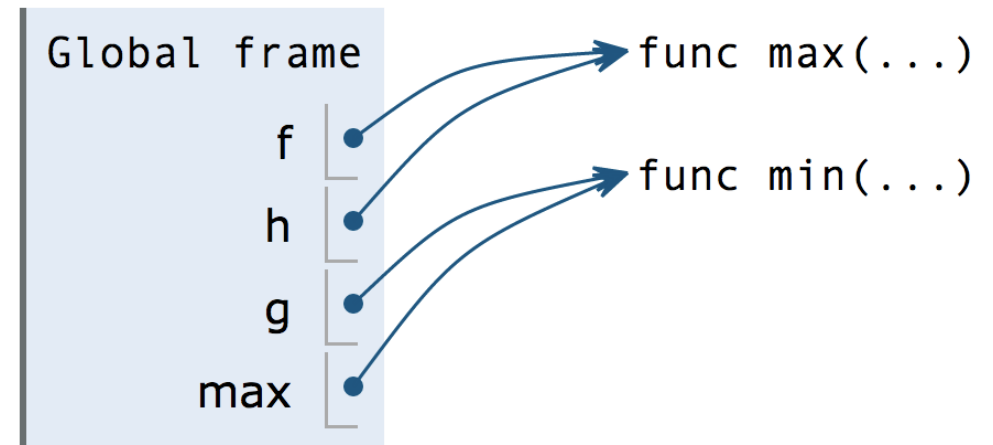


Interactive Diagram

Discussion Question 1 Solution

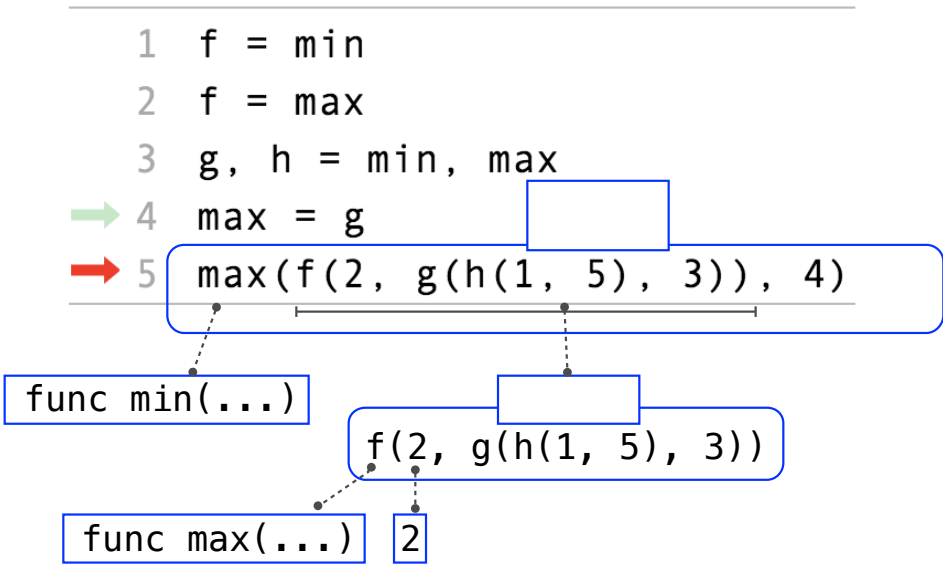


(Demo)

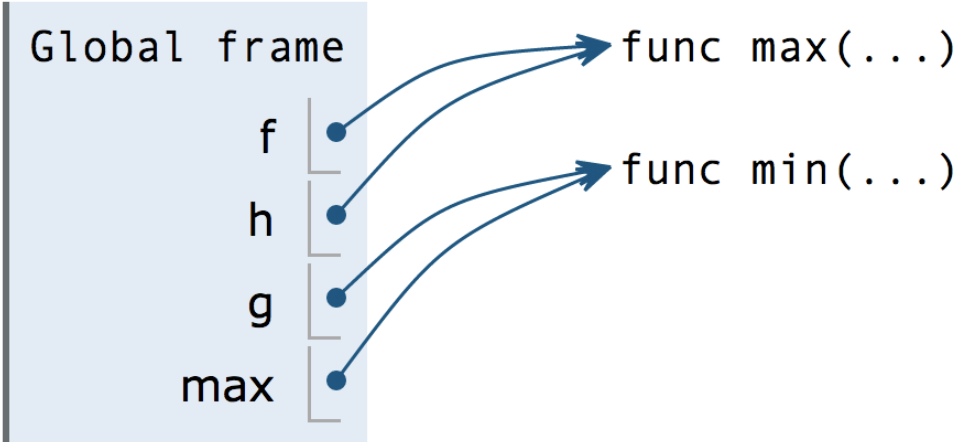


Interactive Diagram

Discussion Question 1 Solution

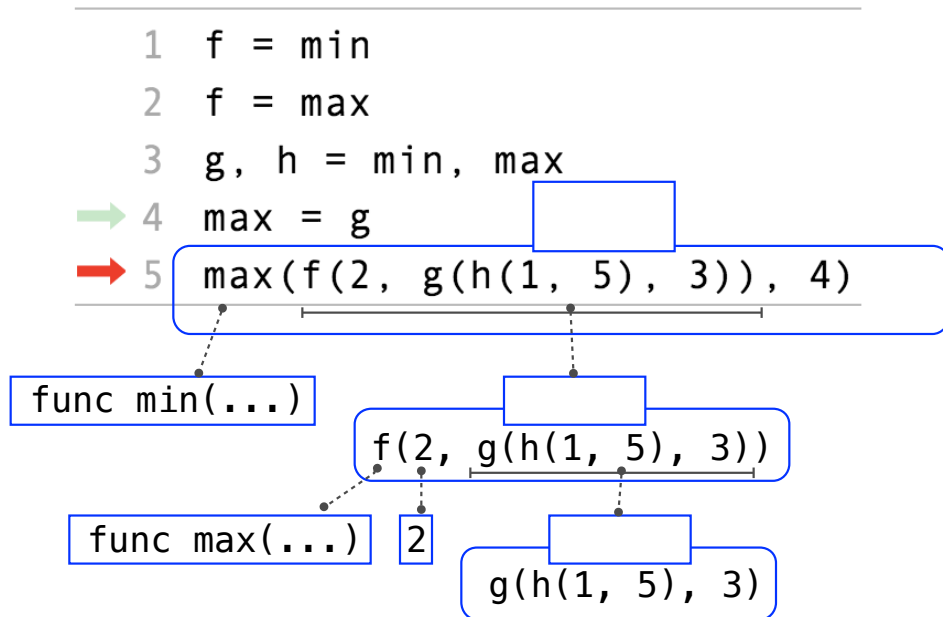


(Demo)

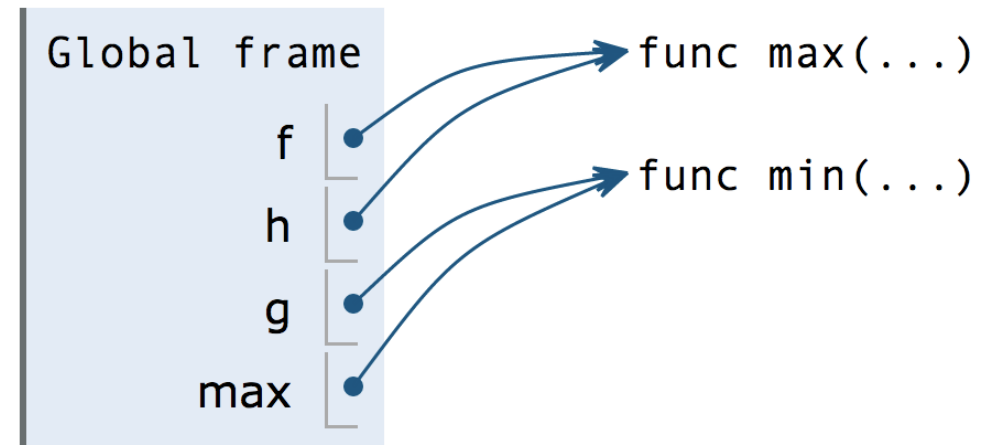


Interactive Diagram

Discussion Question 1 Solution

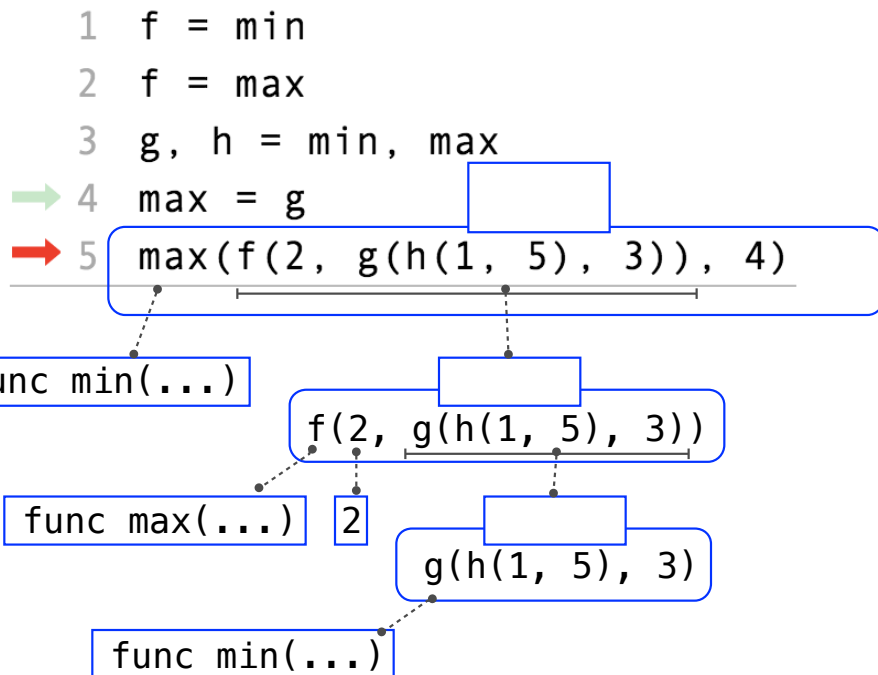


(Demo)

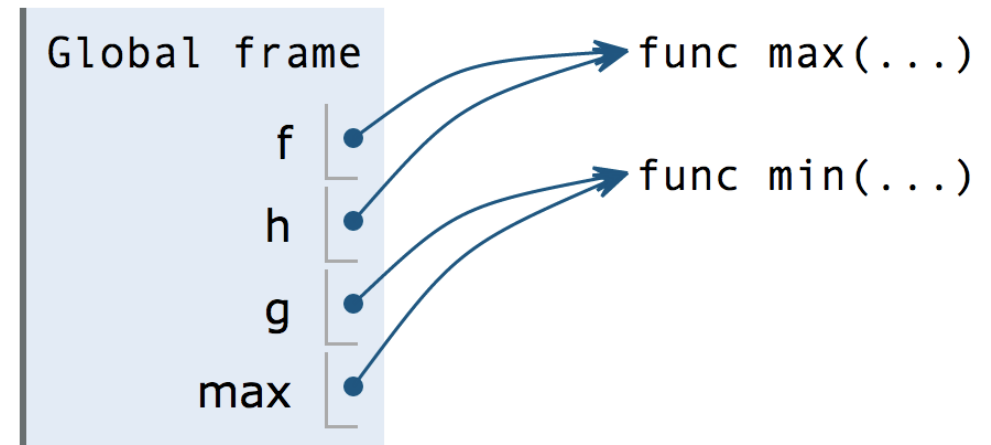


Interactive Diagram

Discussion Question 1 Solution



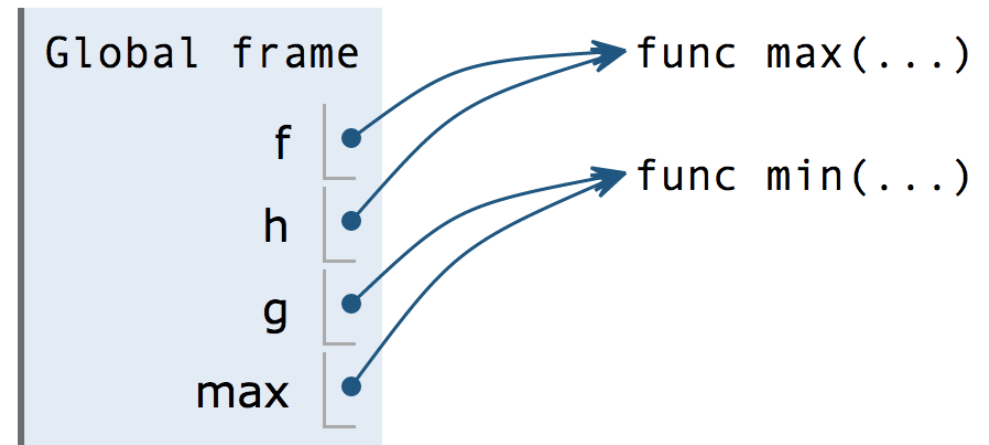
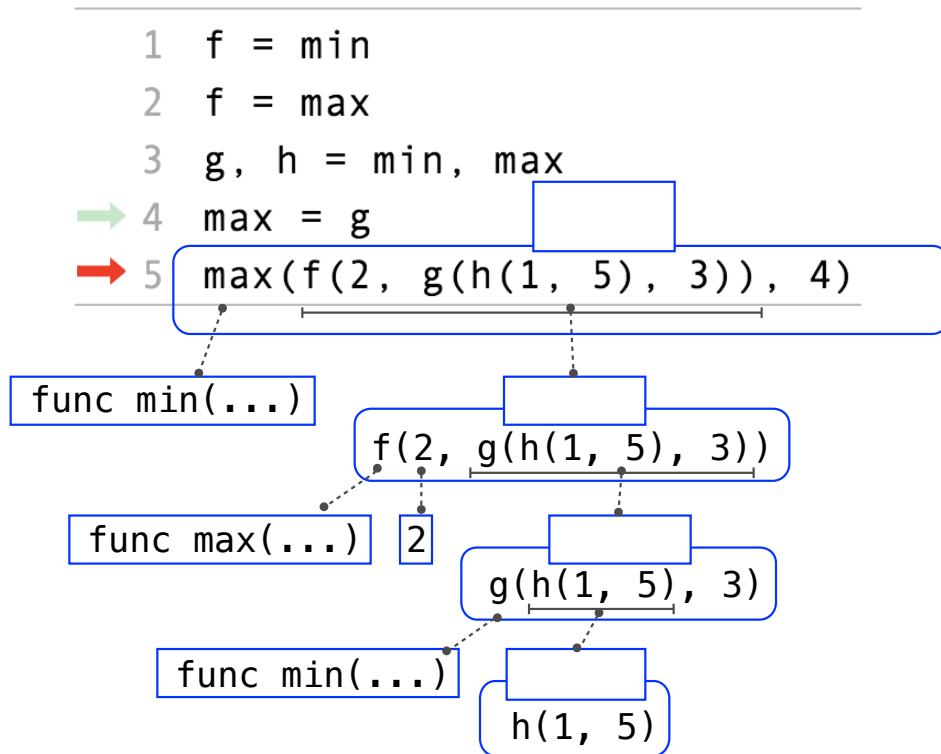
(Demo)



Interactive Diagram

Discussion Question 1 Solution

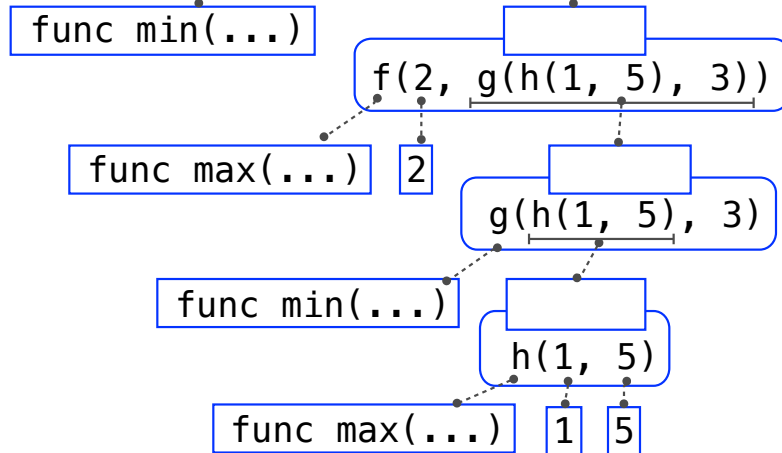
(Demo)



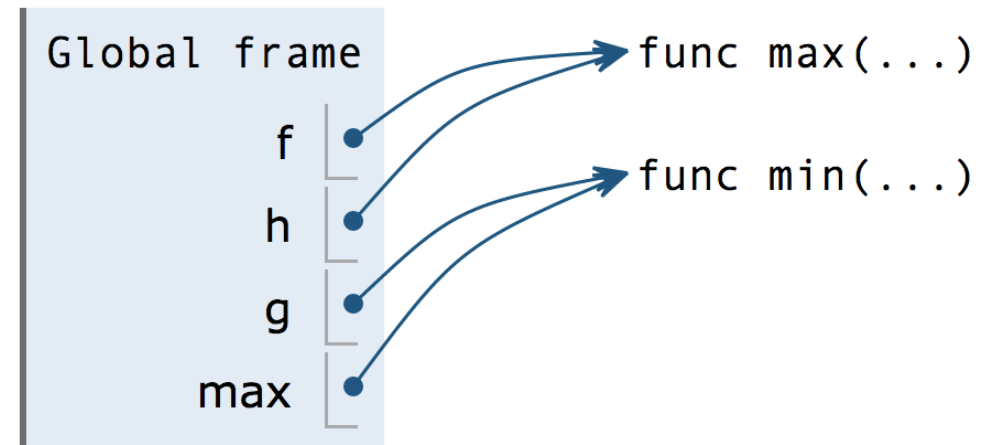
Interactive Diagram

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)



Interactive Diagram

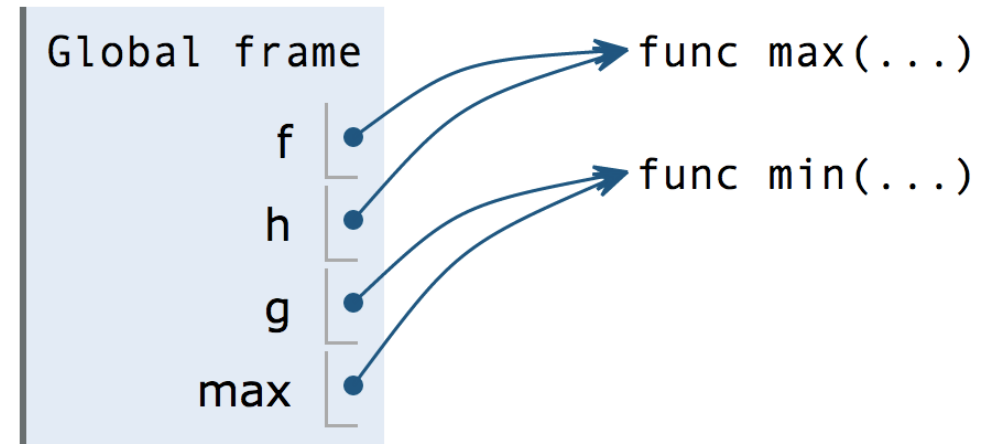
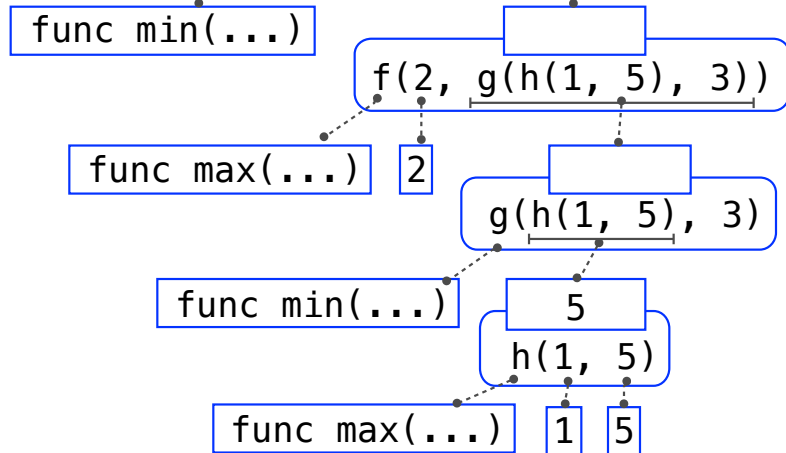
Discussion Question 1 Solution

(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

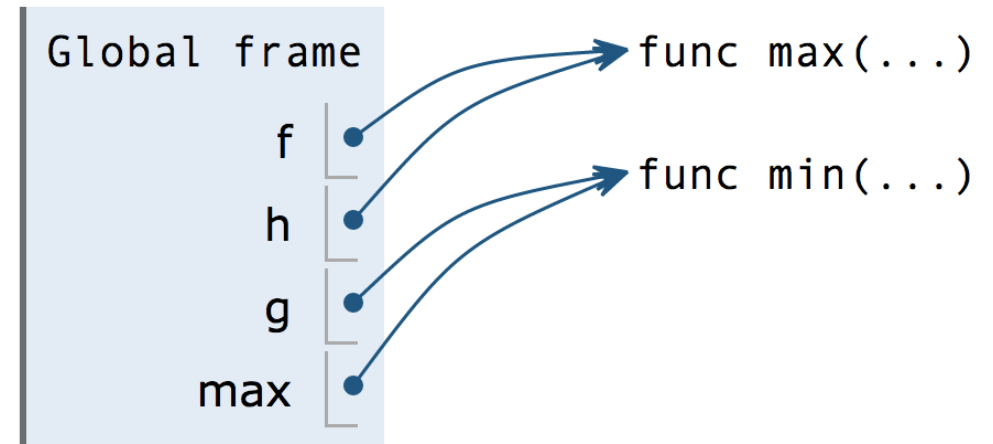
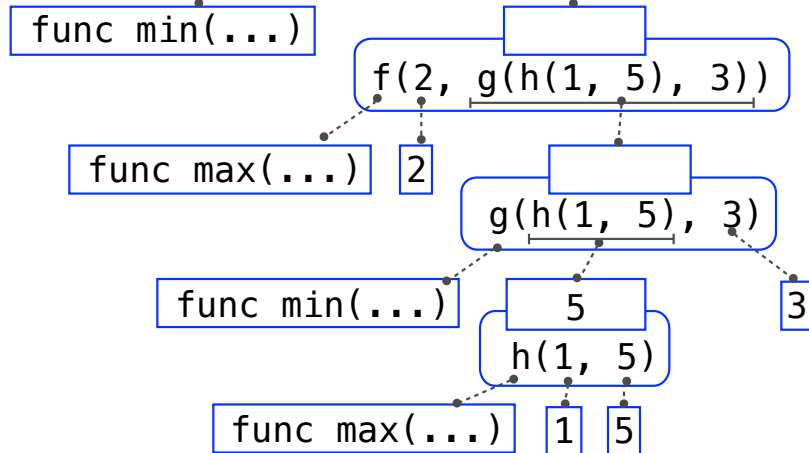
Discussion Question 1 Solution

(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)

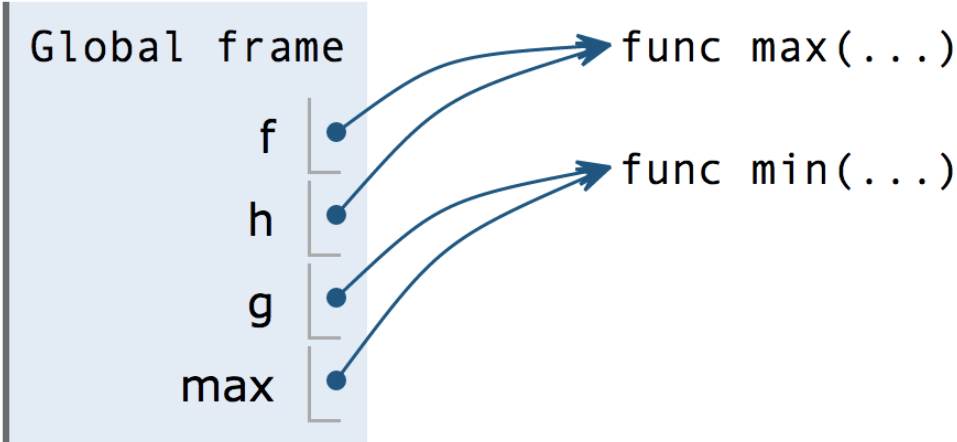
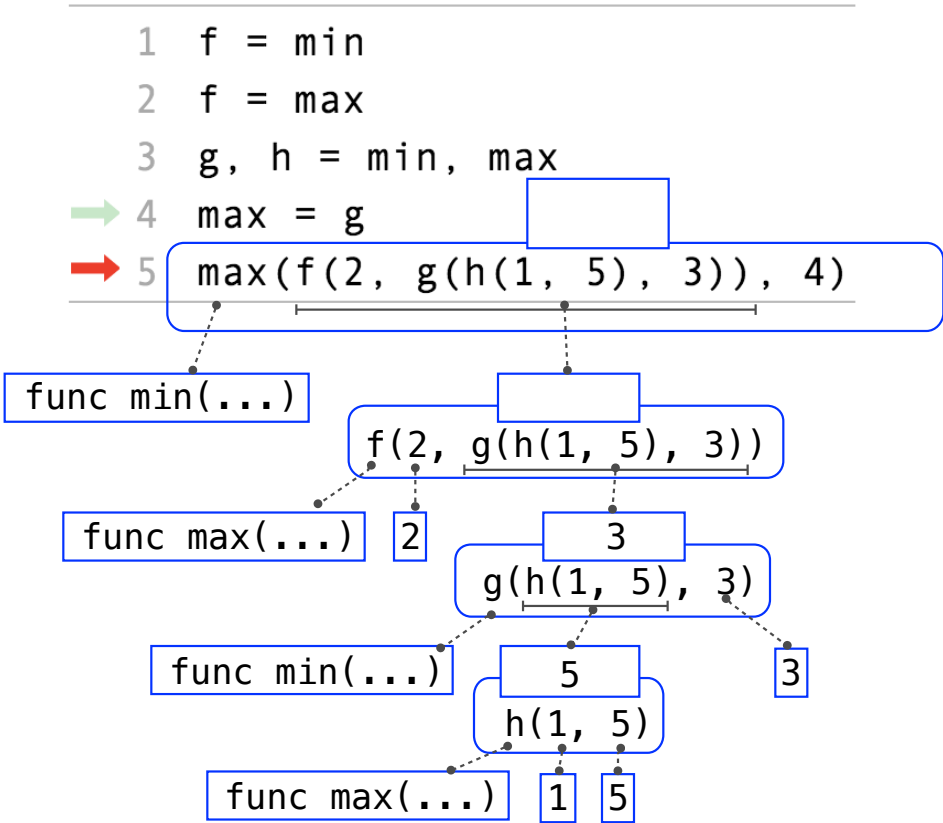
```



Interactive Diagram

Discussion Question 1 Solution

(Demo)



Interactive Diagram

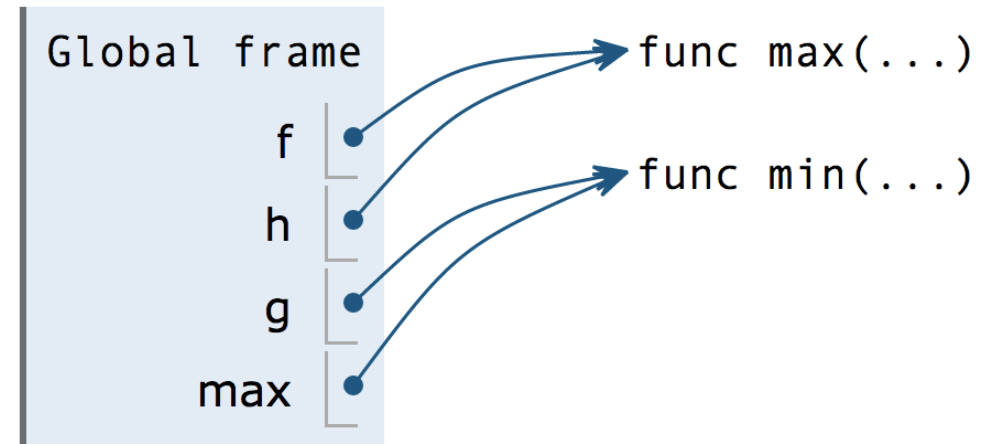
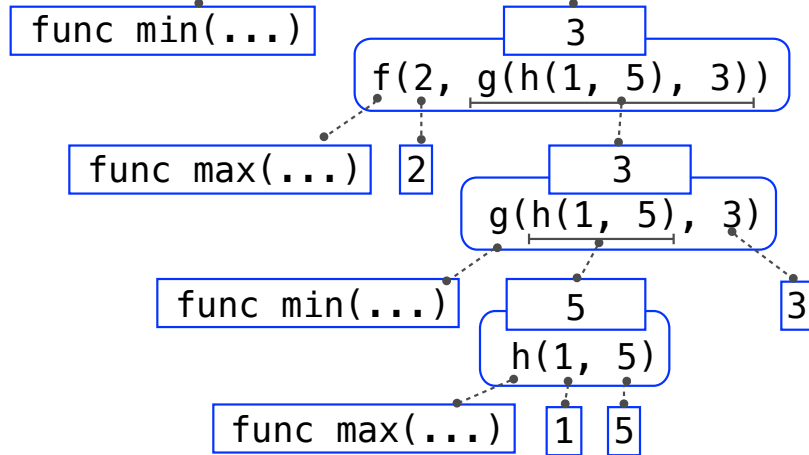
Discussion Question 1 Solution

(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)

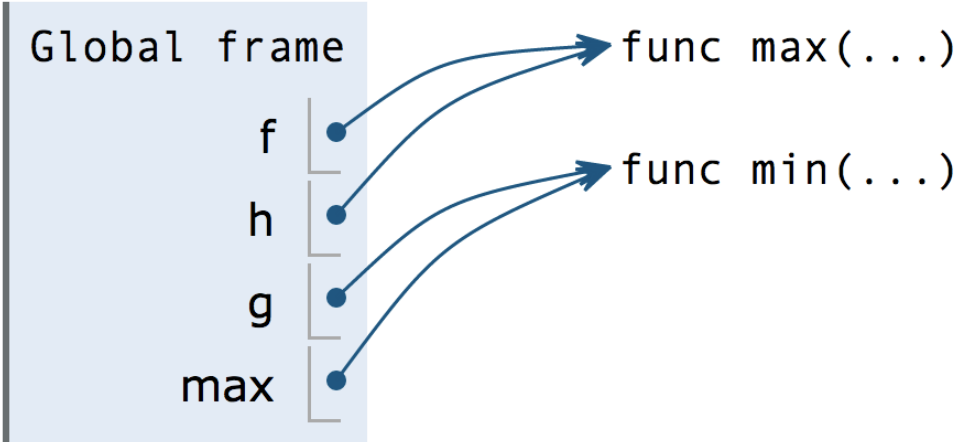
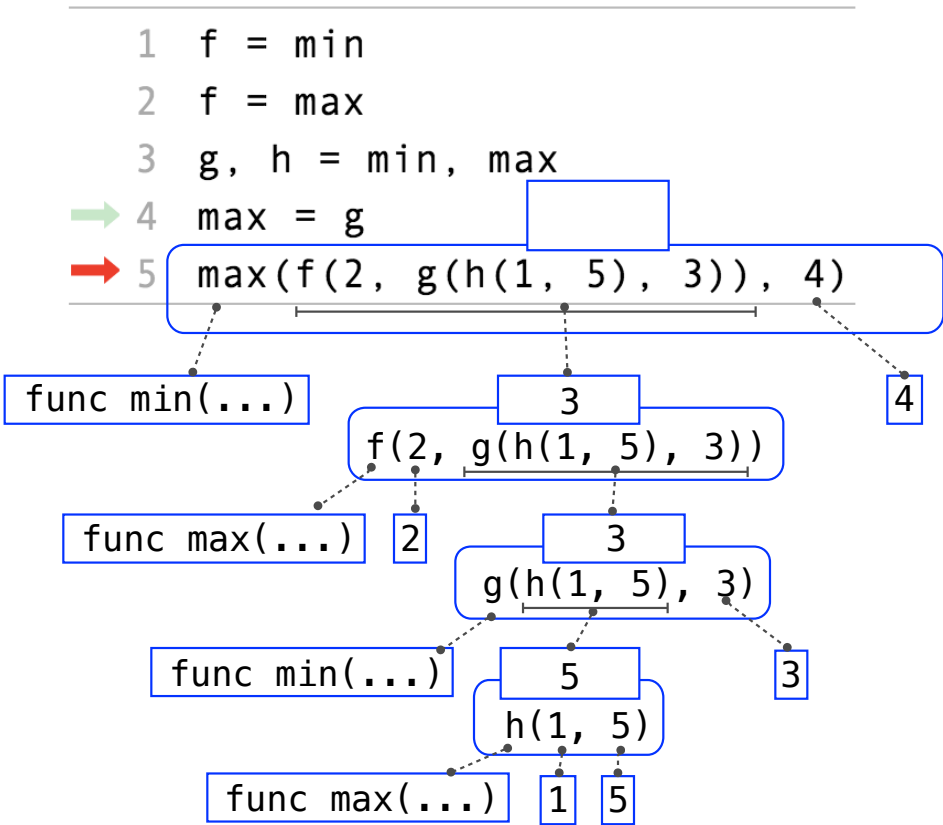
```



Interactive Diagram

Discussion Question 1 Solution

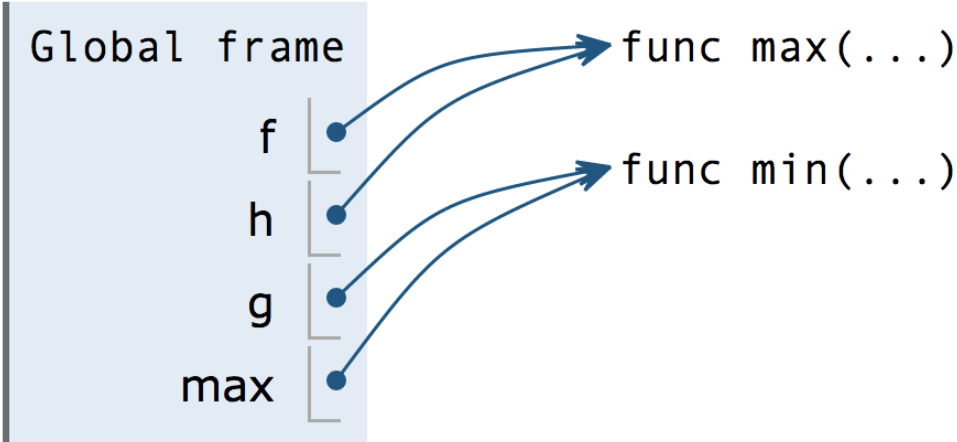
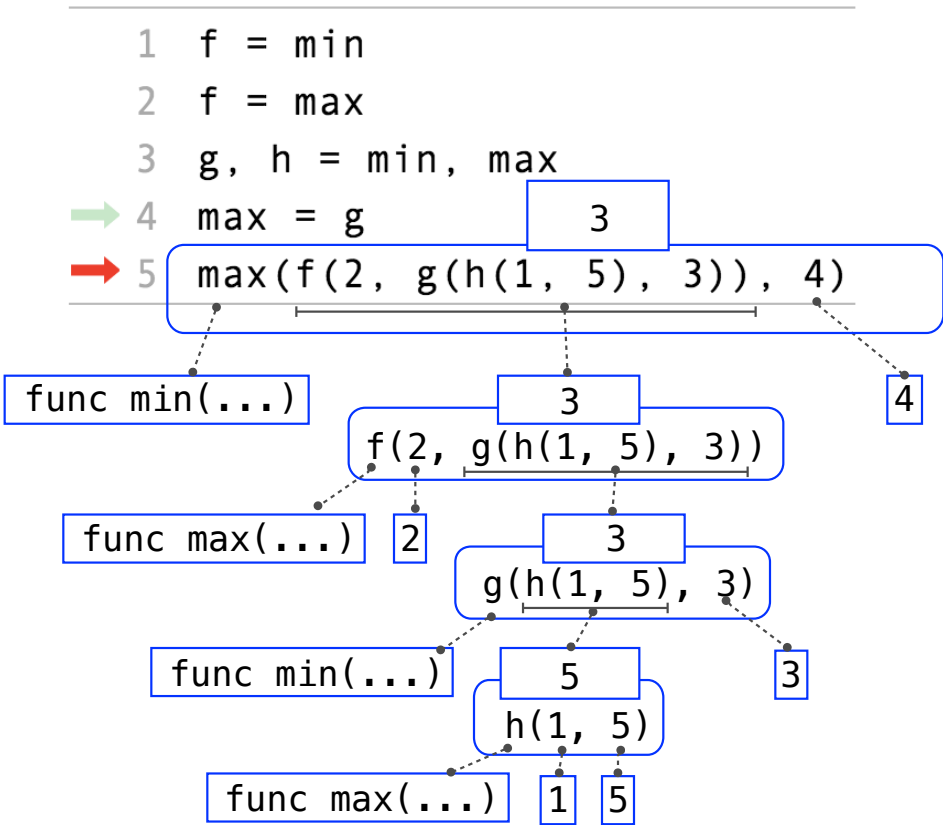
(Demo)



Interactive Diagram

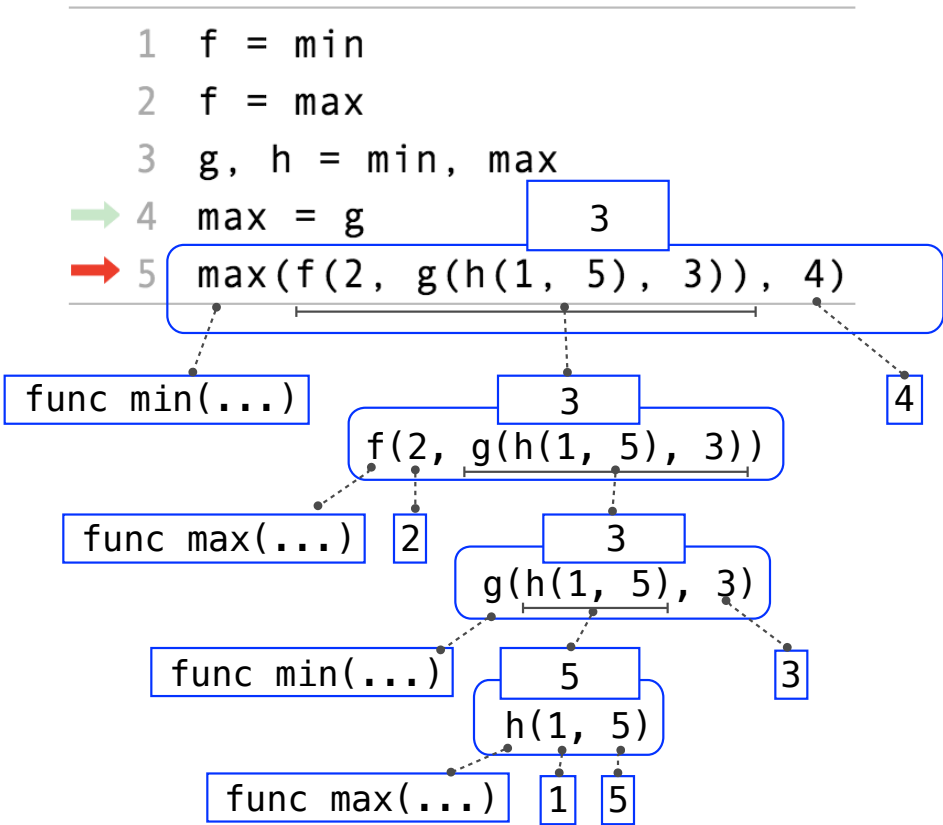
Discussion Question 1 Solution

(Demo)

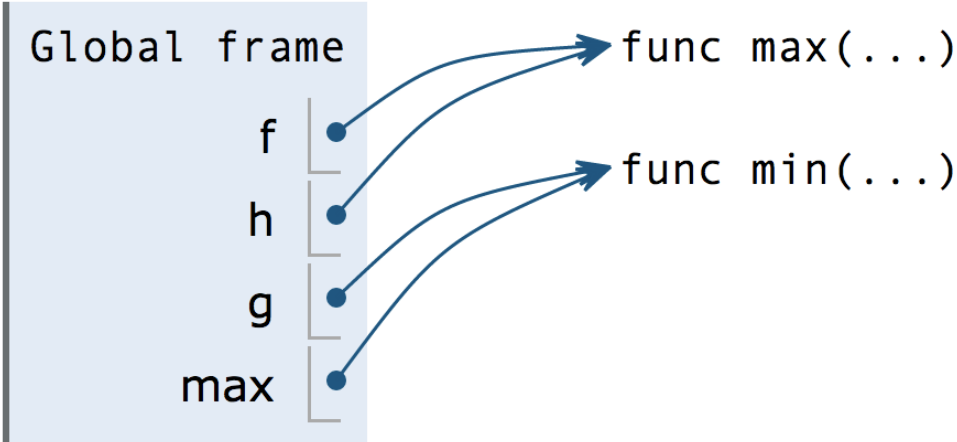


Interactive Diagram

Discussion Question 1 Solution



(Demo)



3

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

[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

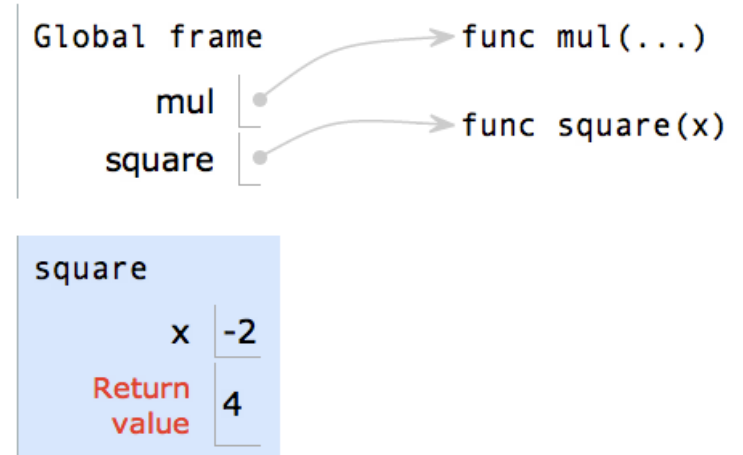
[Interactive Diagram](#)

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)
```



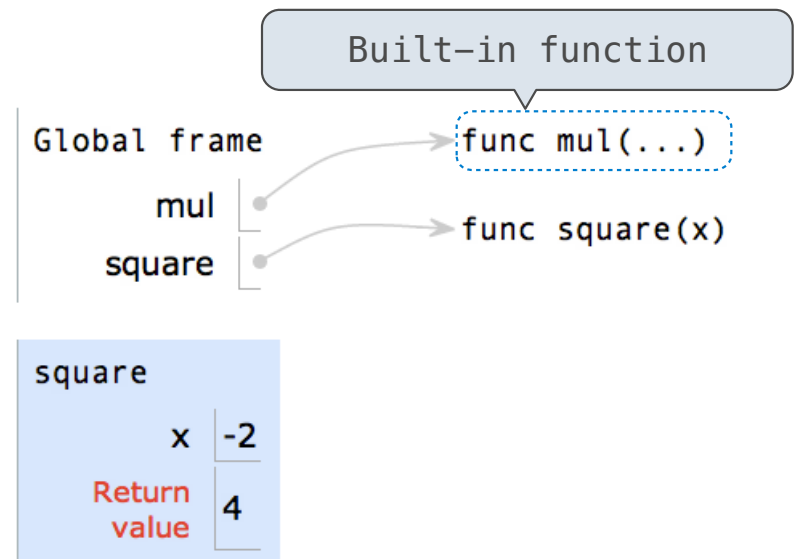
Interactive Diagram

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)
```



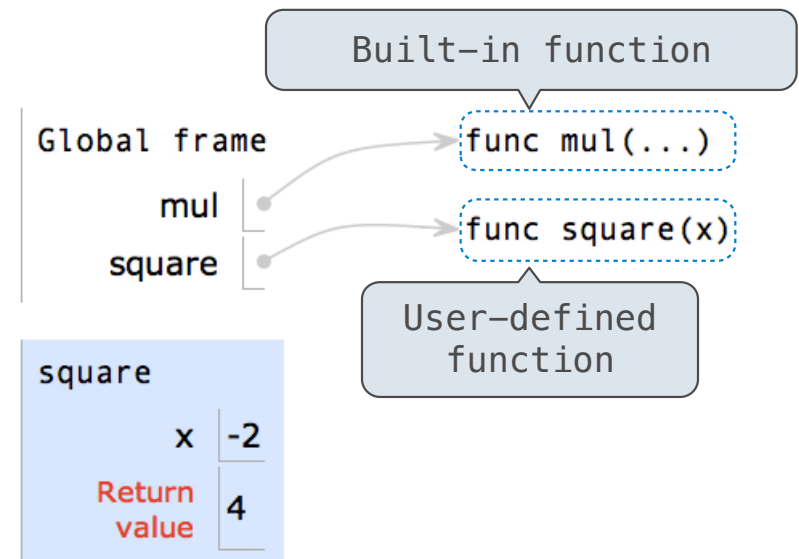
Interactive Diagram

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)
```

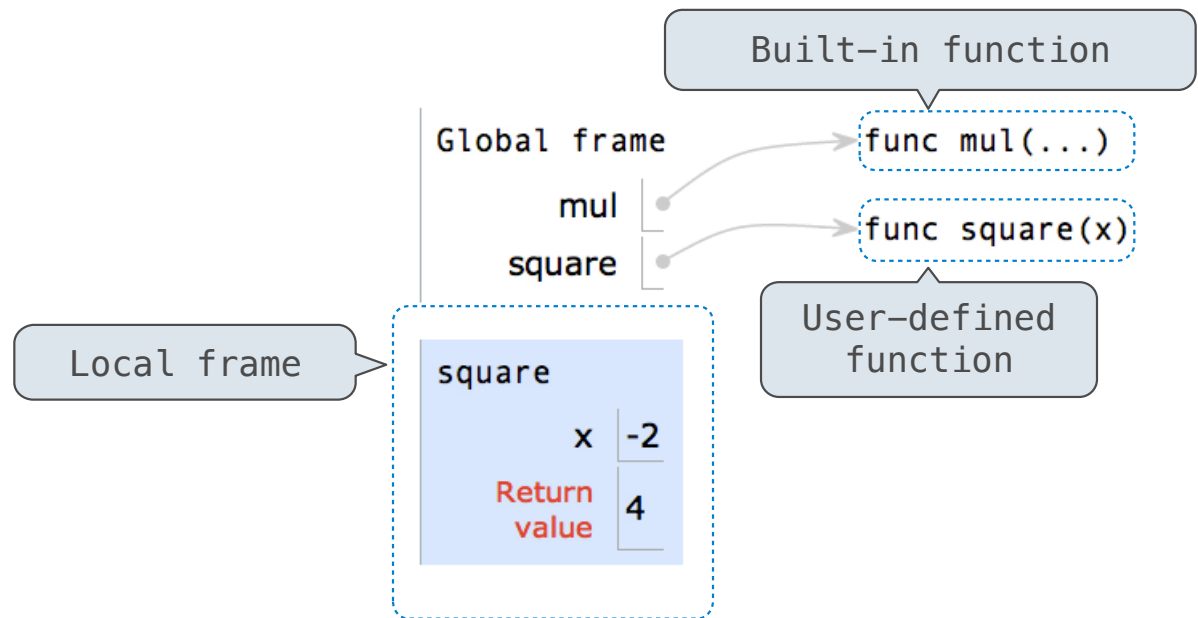


Calling User-Defined Functions

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)
```



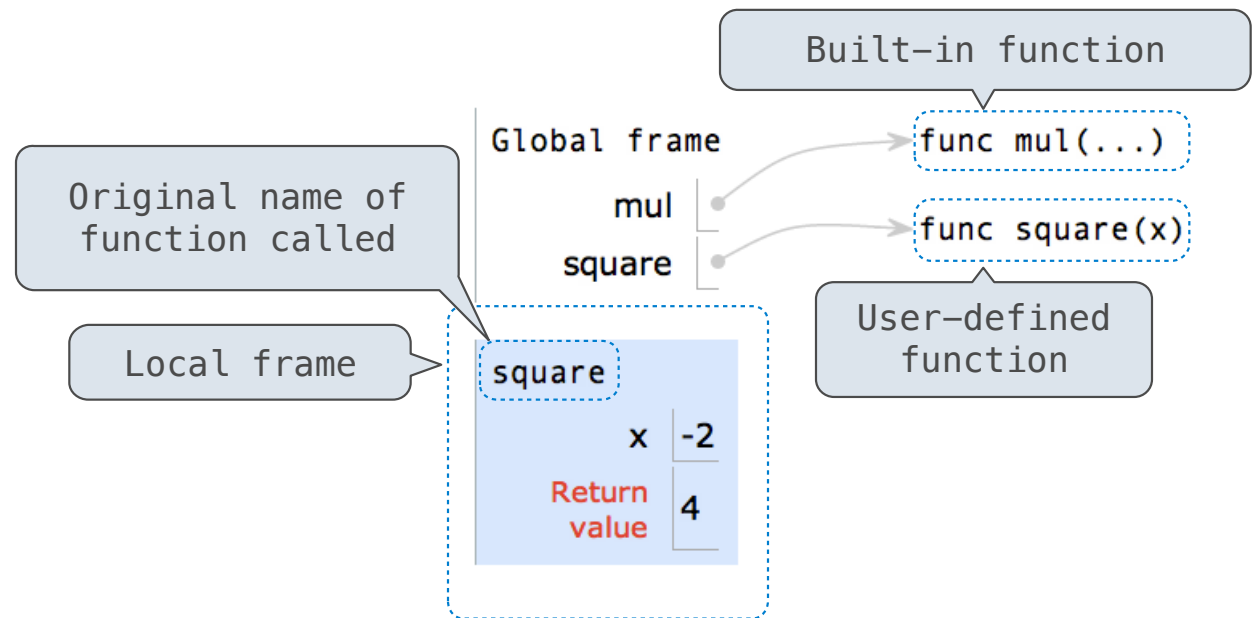
Interactive Diagram

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)
```



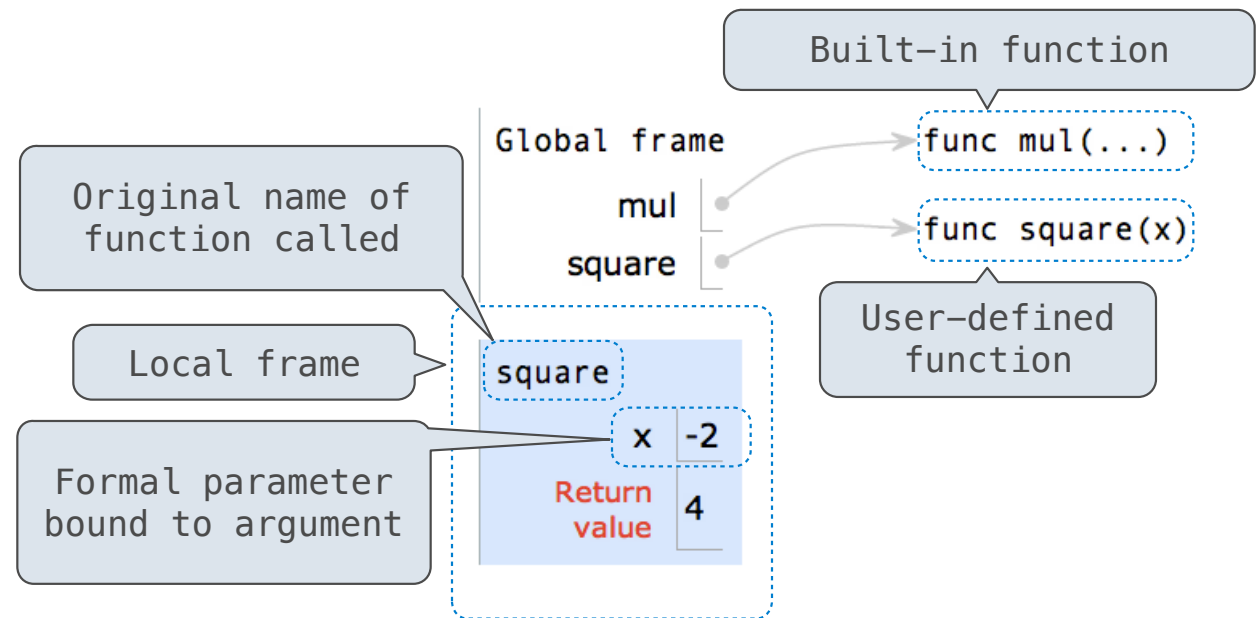
Interactive Diagram

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)
```



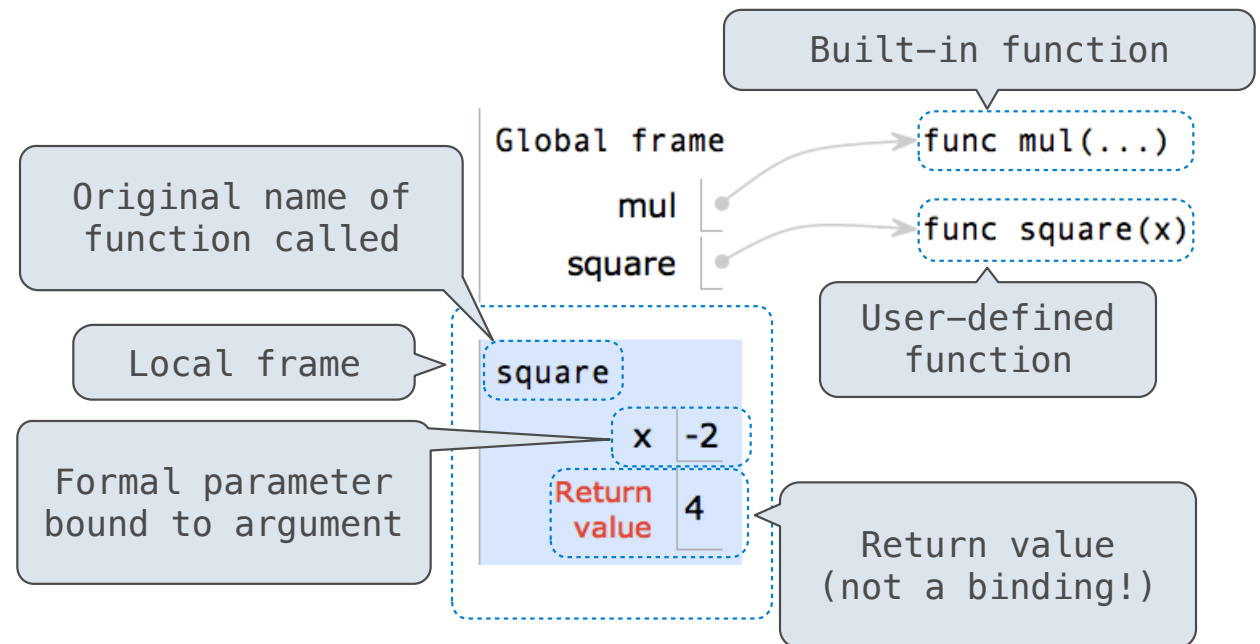
Interactive Diagram

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)
```



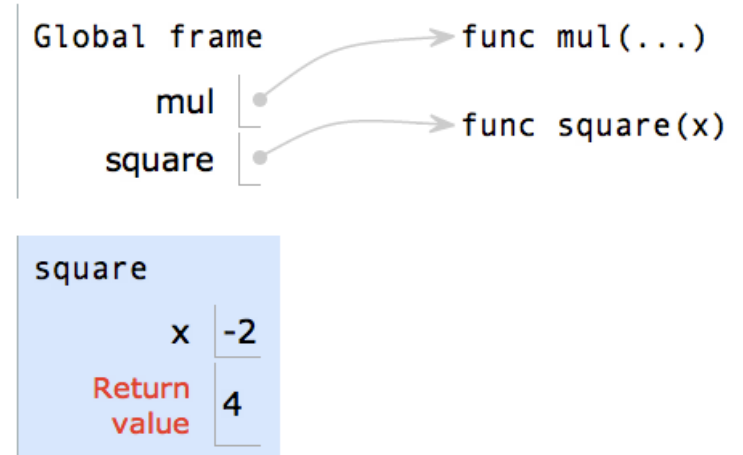
Interactive Diagram

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

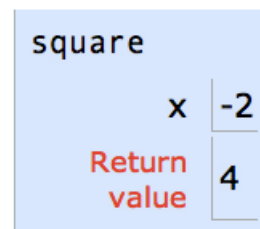
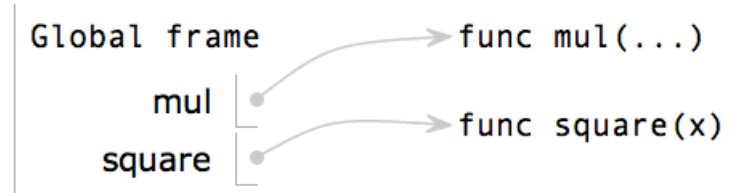
Calling User-Defined Functions

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



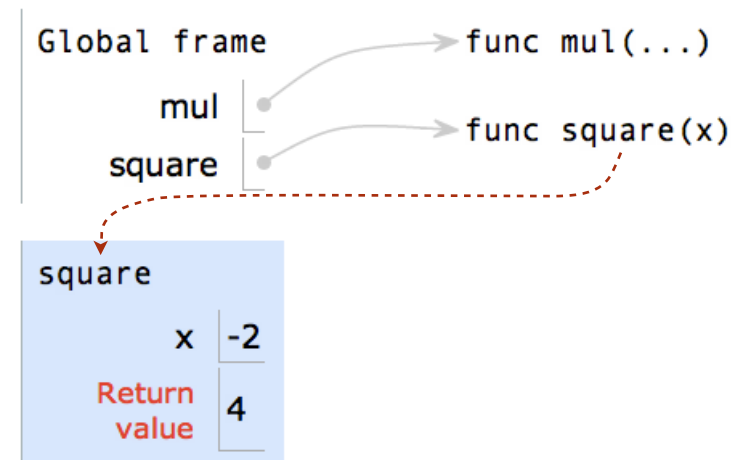
Calling User-Defined Functions

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



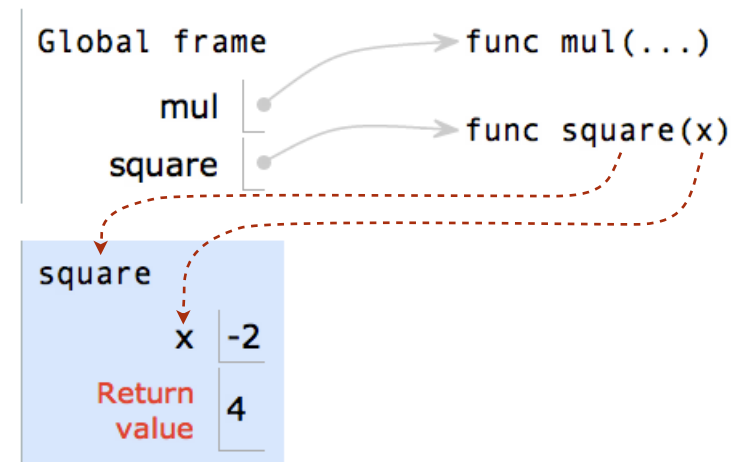
Calling User-Defined Functions

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 illustrates the function call `does_not_square(4)`. A dashed arrow points from the function name to the expression `x * x` inside the function body. A callout box labeled "No return" points to the end of the function body, indicating that the function does not have an explicit return statement and therefore returns `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
```

```
...     
```

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

The diagram illustrates the behavior of the `does_not_square` function. A callout bubble labeled "No return" points to the function definition, indicating that it does not have an explicit `return` statement. Another callout bubble labeled "None value is not displayed" points to the function call `does_not_square(4)`, indicating that the `None` value returned by the function is not displayed by the interpreter.

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

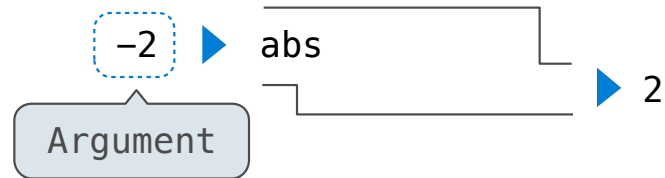
Pure Functions
just return values



Non-Pure Functions
have side effects

Pure Functions & Non-Pure Functions

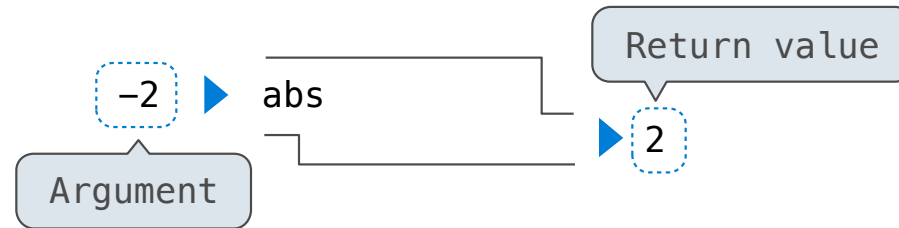
Pure Functions
just return values



Non-Pure Functions
have side effects

Pure Functions & Non-Pure Functions

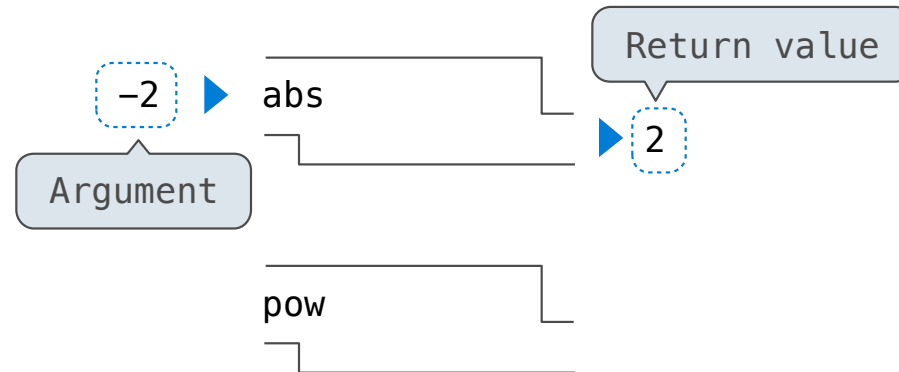
Pure Functions
just return values



Non-Pure Functions
have side effects

Pure Functions & Non-Pure Functions

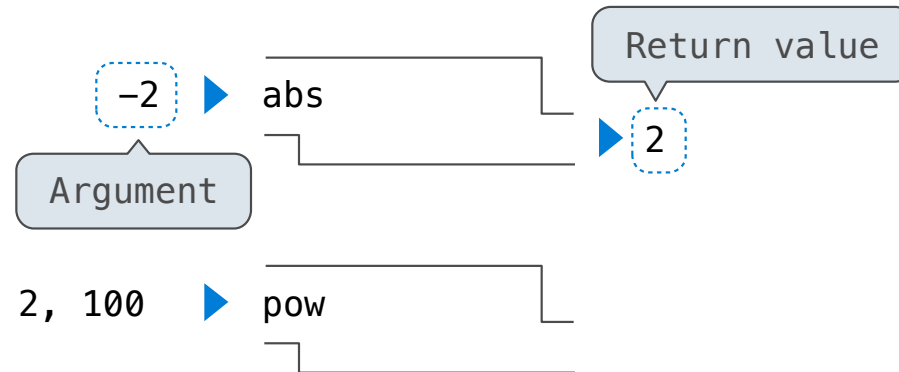
Pure Functions
just return values



Non-Pure Functions
have side effects

Pure Functions & Non-Pure Functions

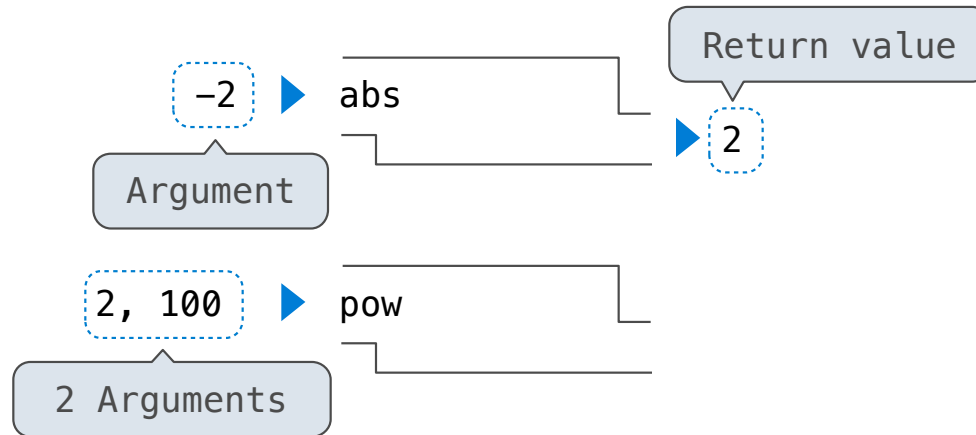
Pure Functions
just return values



Non-Pure Functions
have side effects

Pure Functions & Non-Pure Functions

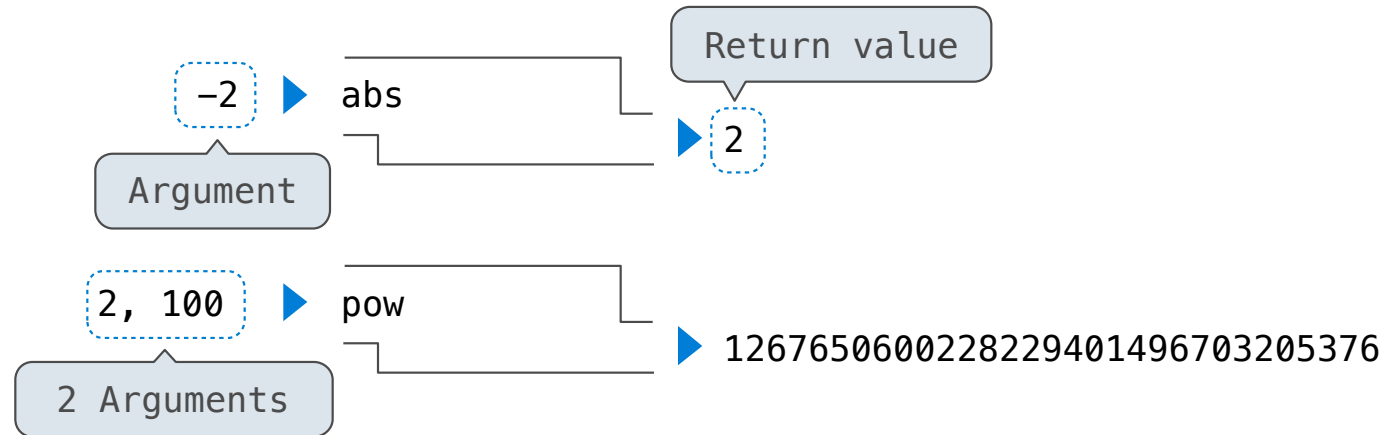
Pure Functions
just return values



Non-Pure Functions
have side effects

Pure Functions & Non-Pure Functions

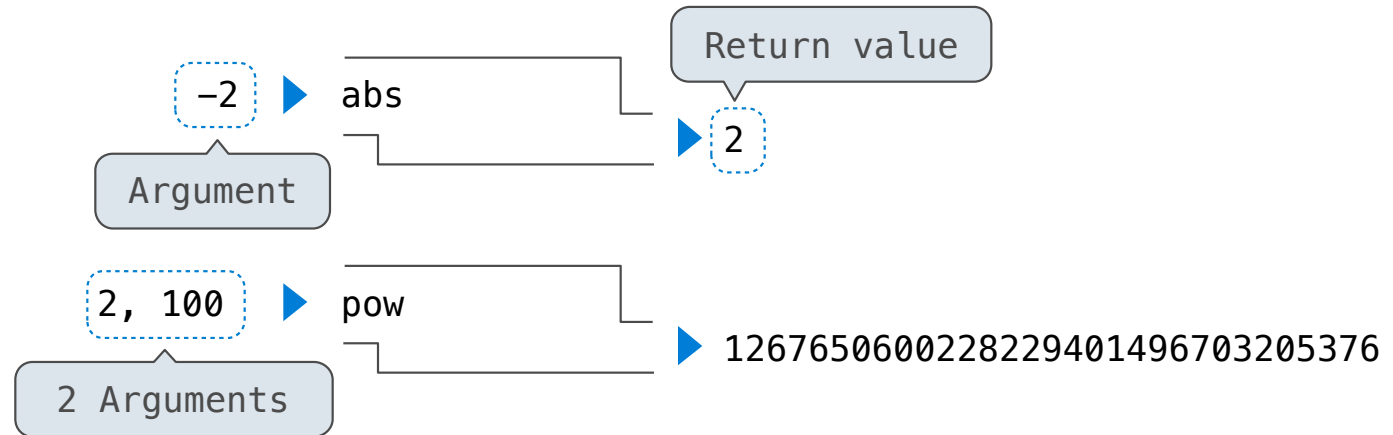
Pure Functions
just return values



Non-Pure Functions
have side effects

Pure Functions & Non-Pure Functions

Pure Functions
just return values



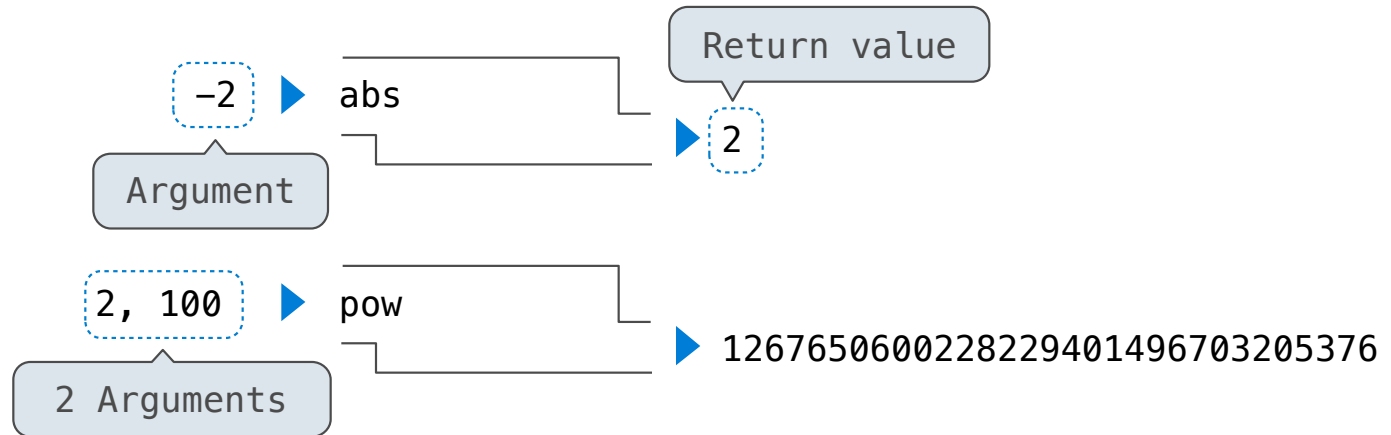
Non-Pure Functions
have side effects

`print`

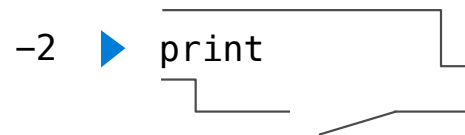
The diagram shows the `print` function. It has a callout box pointing to it, but no arguments or return values are shown, indicating it performs a side effect (output) instead of returning a value.

Pure Functions & Non-Pure Functions

Pure Functions
just return values

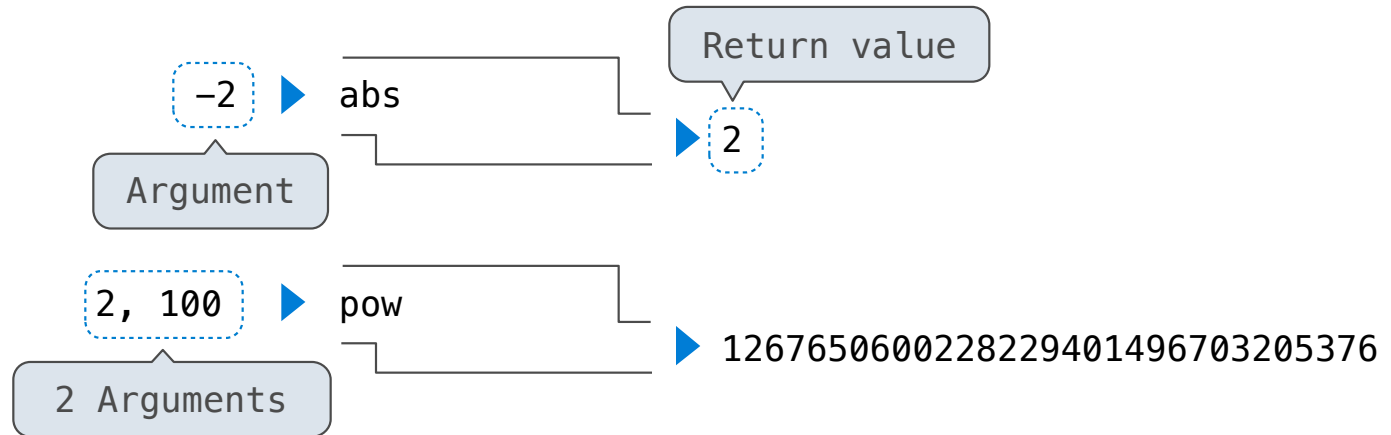


Non-Pure Functions
have side effects



Pure Functions & Non-Pure Functions

Pure Functions
just return values

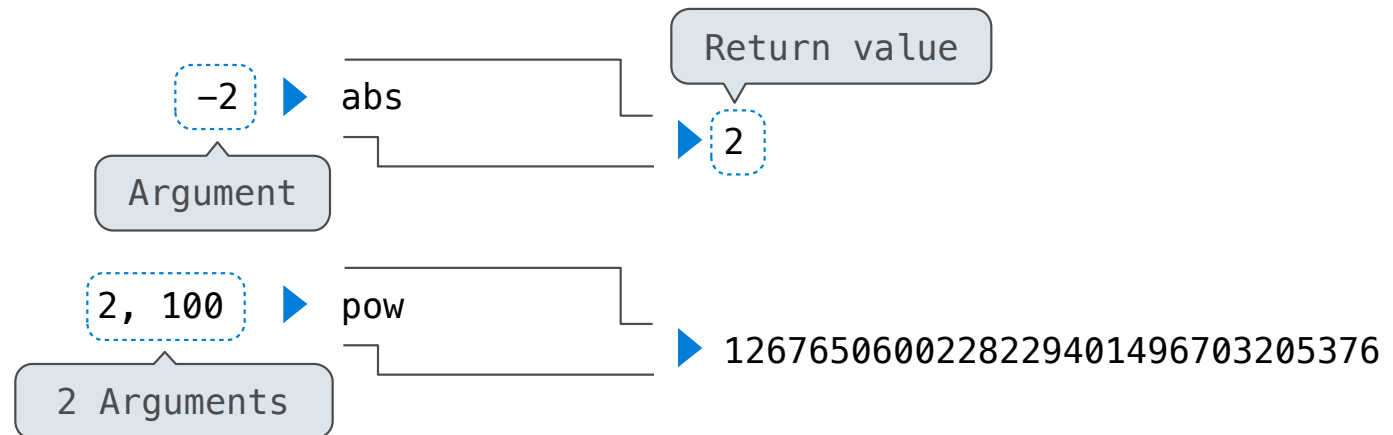


Non-Pure Functions
have side effects

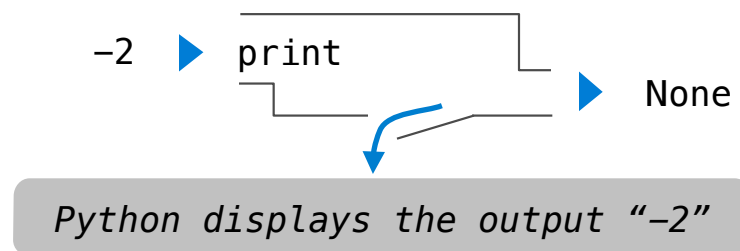


Pure Functions & Non-Pure Functions

Pure Functions
just return values

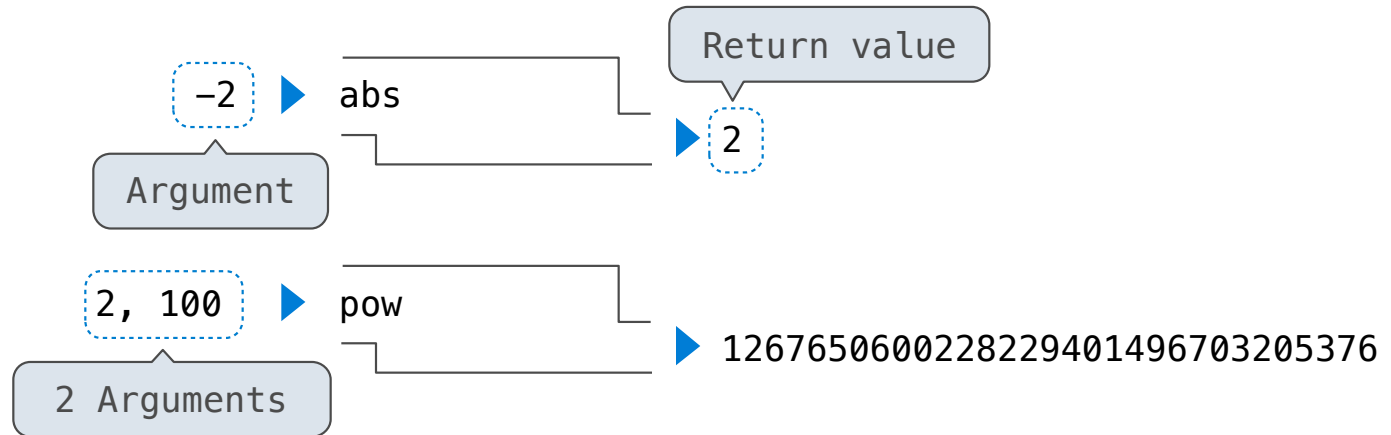


Non-Pure Functions
have side effects

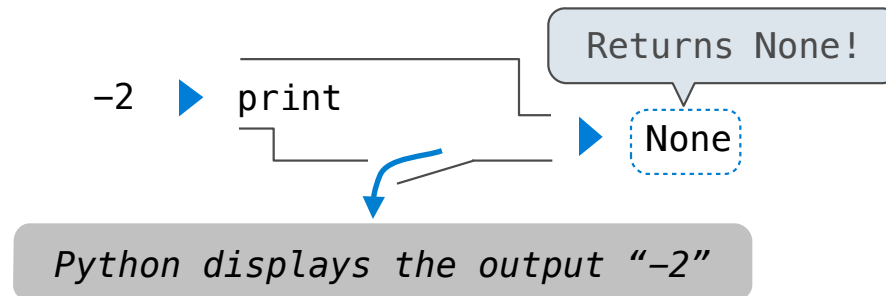


Pure Functions & Non-Pure Functions

Pure Functions
just return values

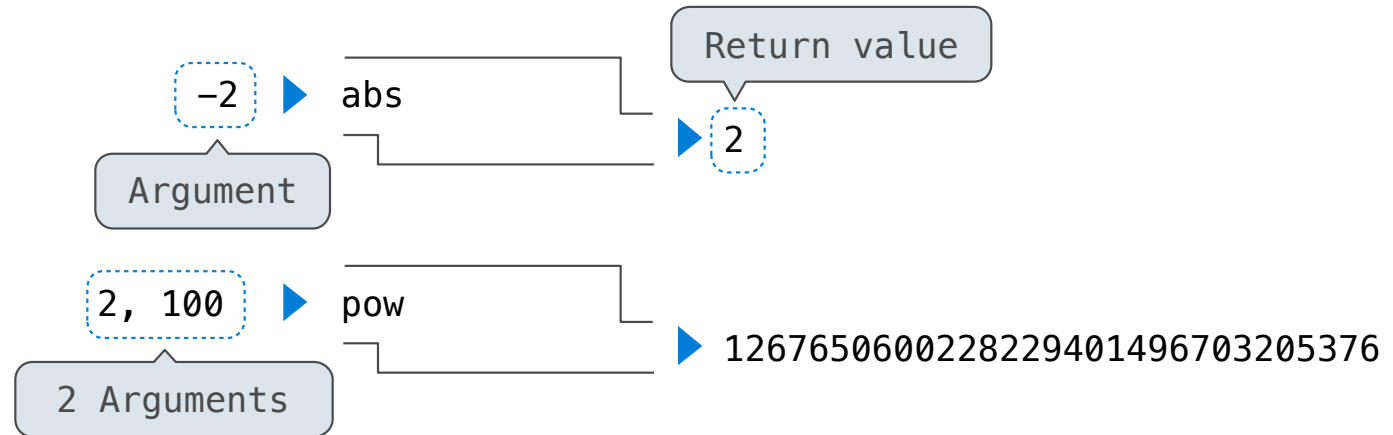


Non-Pure Functions
have side effects

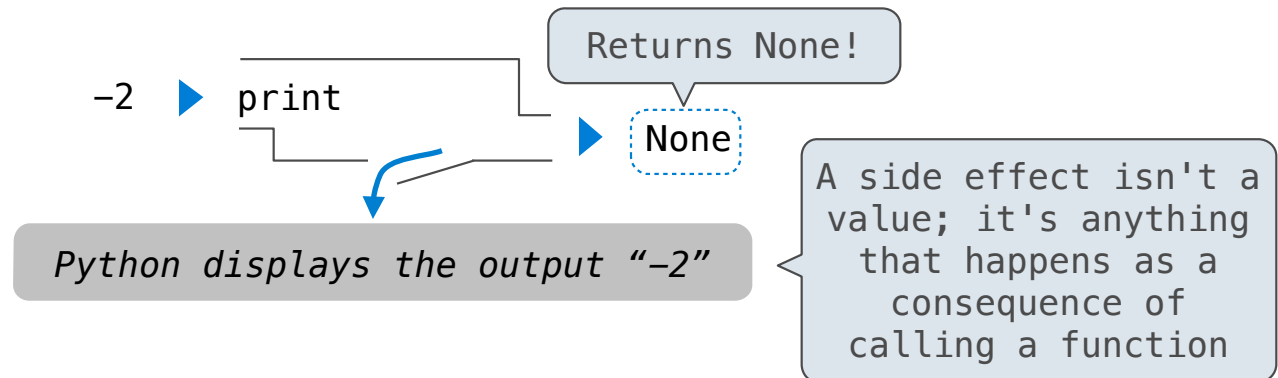


Pure Functions & Non-Pure Functions

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

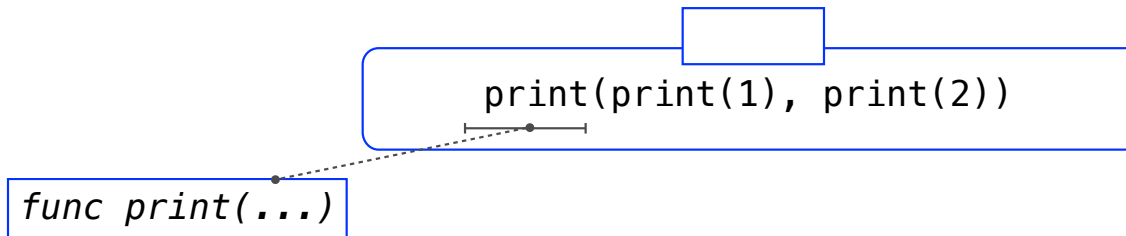
```
>>> print(print(1), print(2))  
1  
2  
None None
```



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

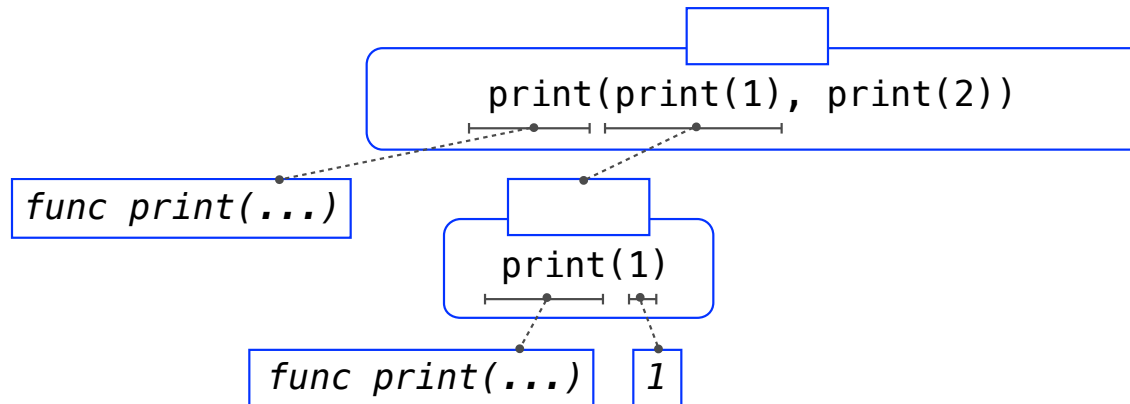
Nested Expressions with Print

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



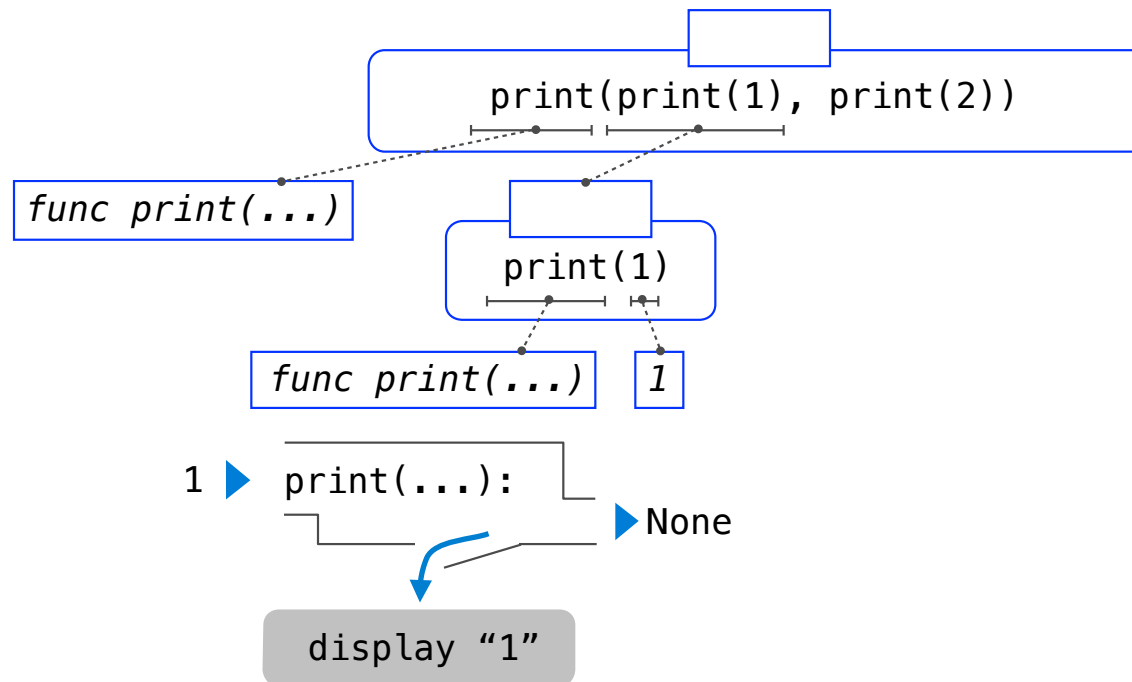
Nested Expressions with Print

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



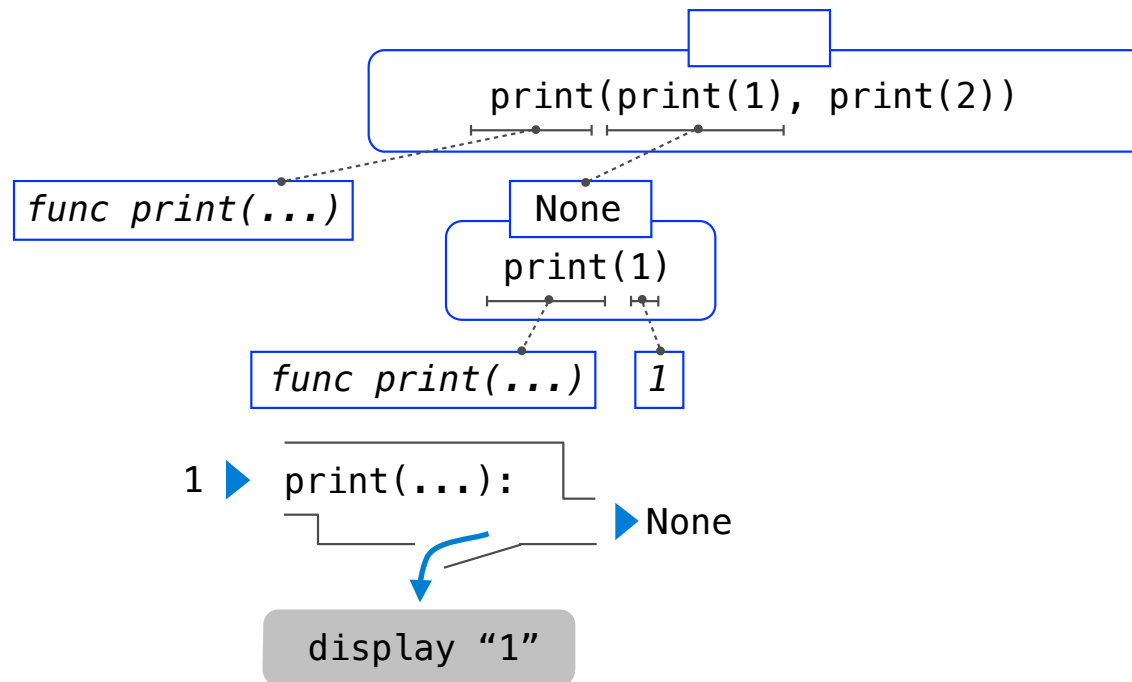
Nested Expressions with Print

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



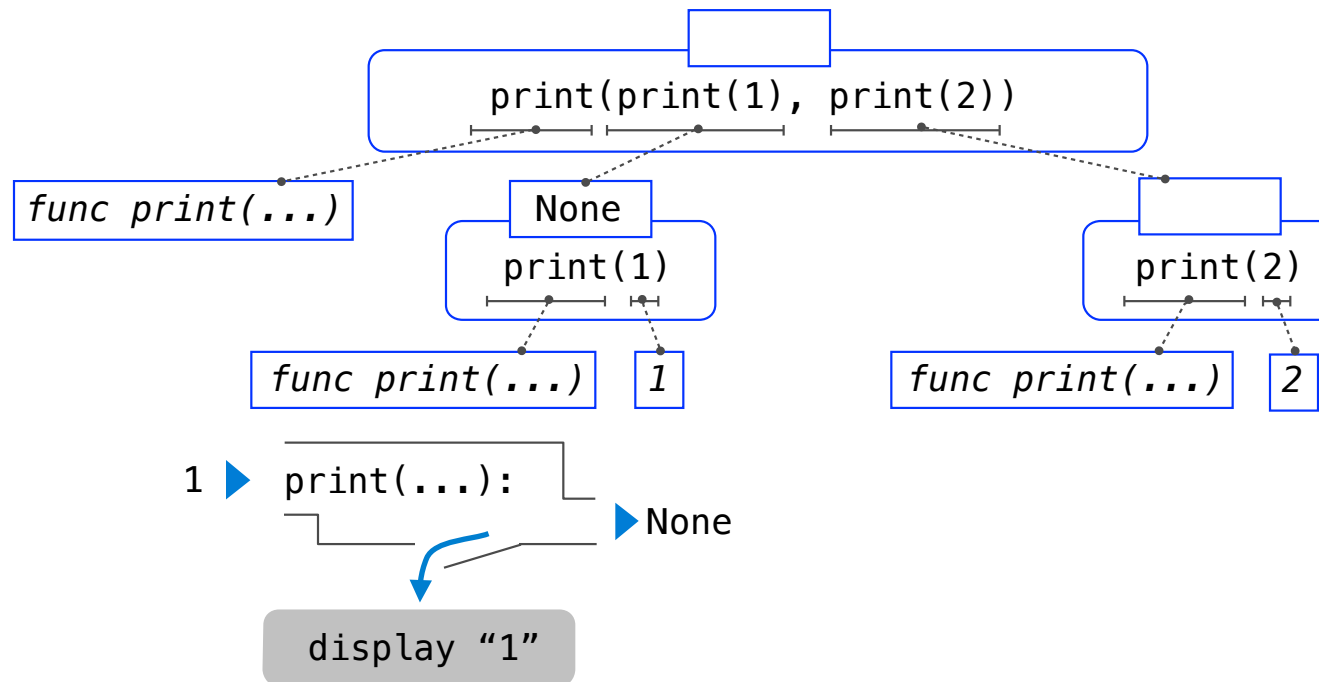
Nested Expressions with Print

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



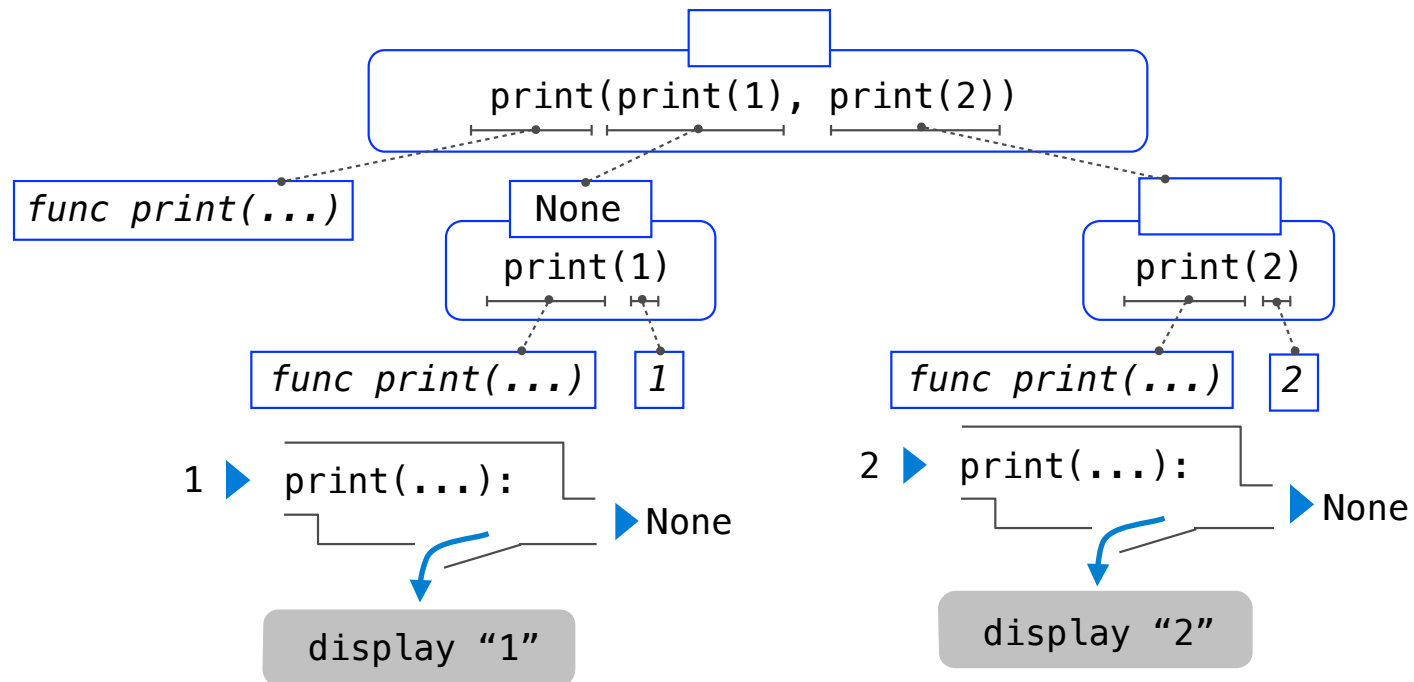
Nested Expressions with Print

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



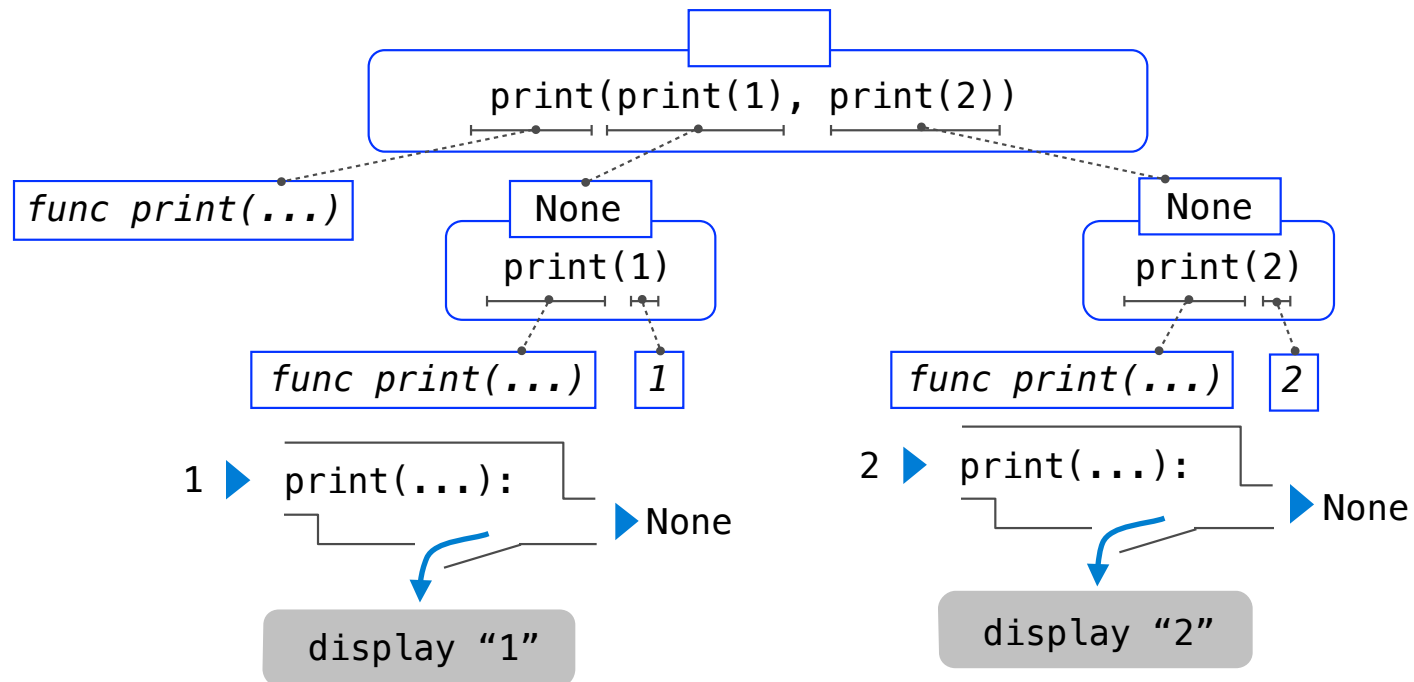
Nested Expressions with Print

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

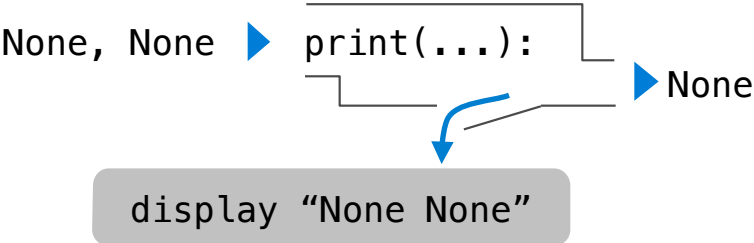


Nested Expressions with Print

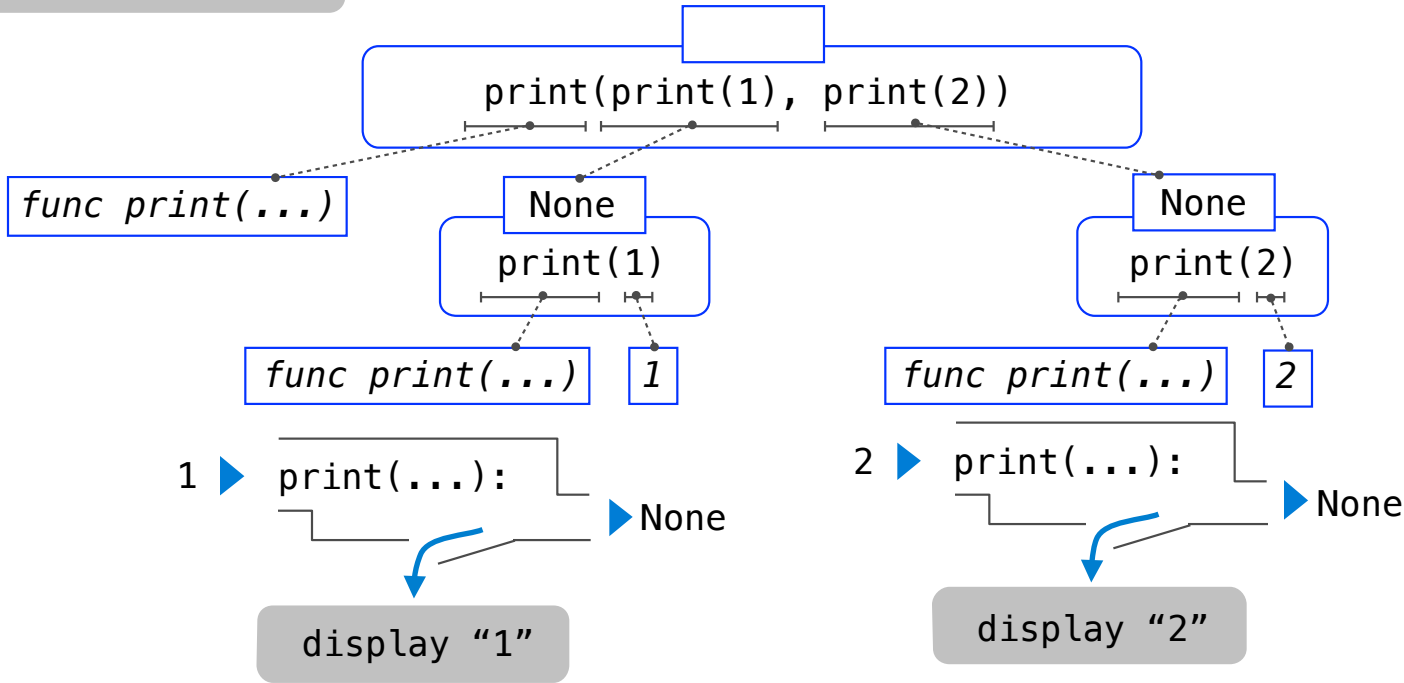
```
>>> print(print(1), print(2))  
1  
2  
None None
```



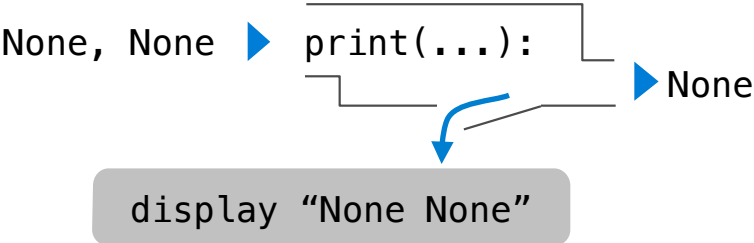
Nested Expressions with Print



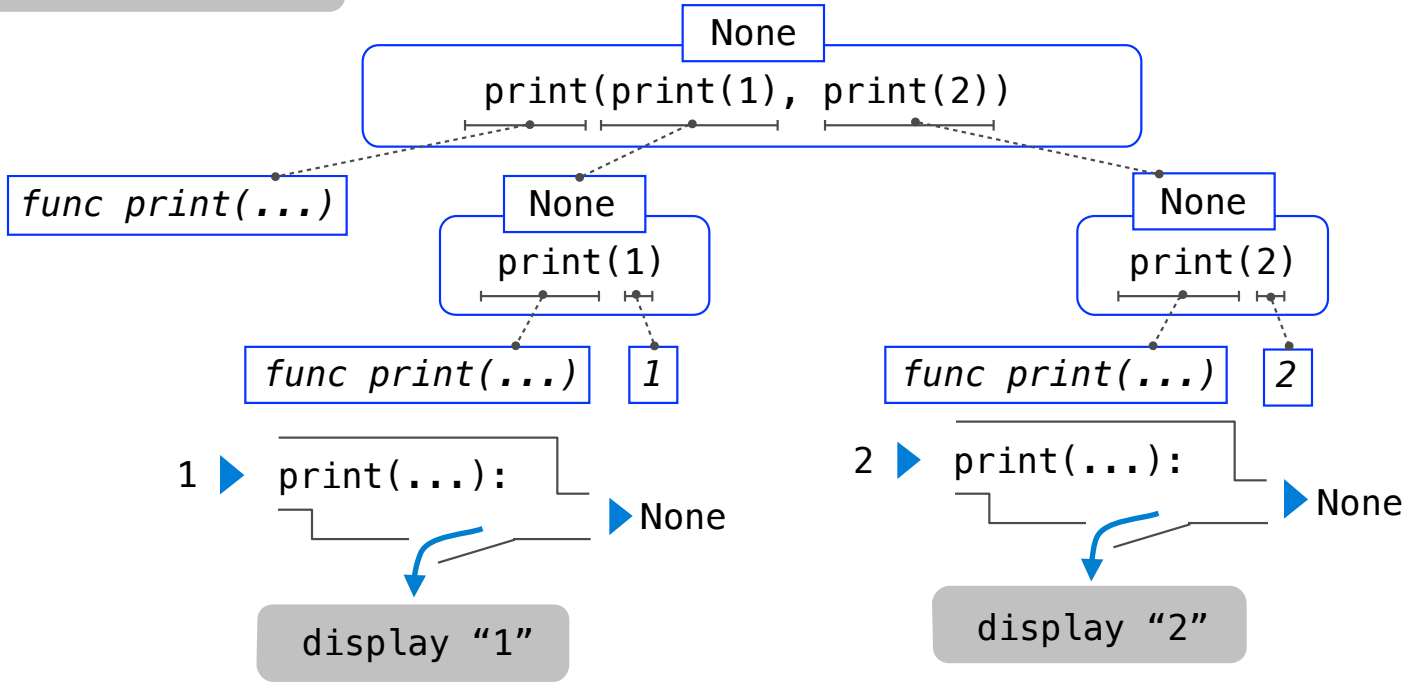
```
>>> print(print(1), print(2))
1
2
None None
```



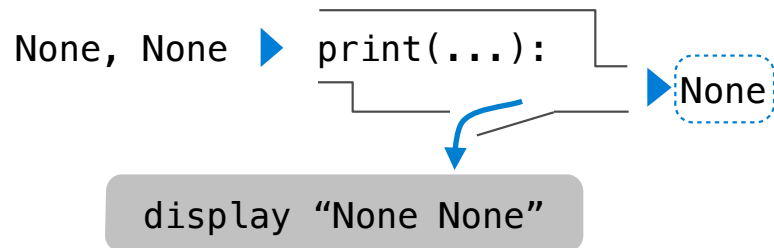
Nested Expressions with Print



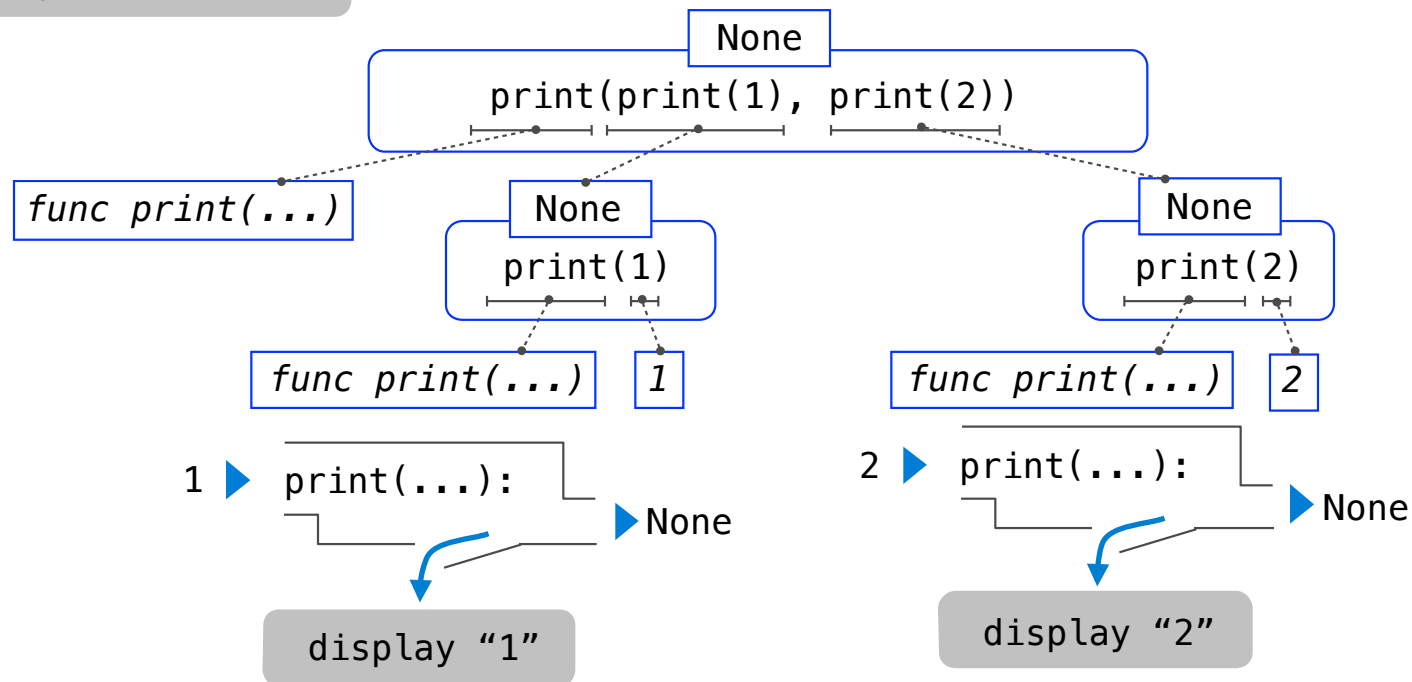
```
>>> print(print(1), print(2))
1
2
None None
```



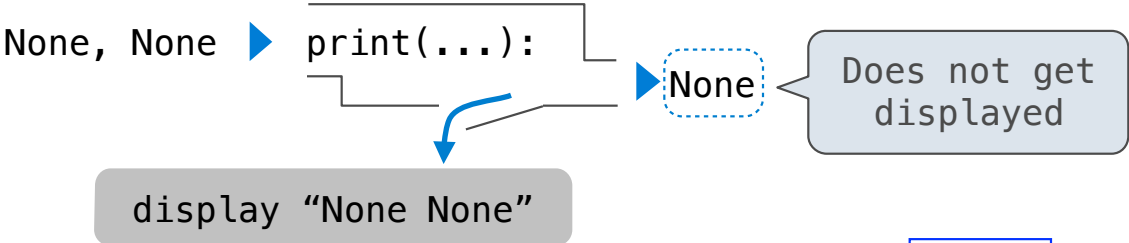
Nested Expressions with Print



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



Nested Expressions with Print



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

