61A Lecture 2

Wednesday, September 3, 2014

Announcements

Announcements

- Lab 1 is due Wednesday 9/3 at 11:59pm

Announcements

- Lab 1 is due Wednesday 9/3 at 11:59pm

- Submitting labs and attending section may help your grade

Announcements

- Lab 1 is due Wednesday 9/3 at 11:59pm

- Submitting labs and attending section may help your grade

- Homework 1 is due next Wednesday 9/10 at 11:59pm

Announcements

- Lab 1 is due Wednesday 9/3 at 11:59pm
- Submitting labs and attending section may help your grade
- Homework 1 is due next Wednesday 9/10 at 11:59pm
- Office hours are a great place to ask questions about lab and homework assignments (demo)

Announcements

- Lab 1 is due Wednesday 9/3 at 11:59pm
- Submitting labs and attending section may help your grade
- Homework 1 is due next Wednesday 9/10 at 11:59pm
- Office hours are a great place to ask questions about lab and homework assignments (demo)
- You can switch to sections with open space. http://goo.gl/nWfv7Z

Announcements

- Lab 1 is due Wednesday 9/3 at 11:59pm
- Submitting labs and attending section may help your grade
- Homework 1 is due next Wednesday 9/10 at 11:59pm
- Office hours are a great place to ask questions about lab and homework assignments (demo)
- You can switch to sections with open space. http://goo.gl/nWfv7Z
- Michelle Hwang's sections (15, 18) are for students with little prior CS experience

Announcements

- Lab 1 is due Wednesday 9/3 at 11:59pm
- Submitting labs and attending section may help your grade
- Homework 1 is due next Wednesday 9/10 at 11:59pm
- Office hours are a great place to ask questions about lab and homework assignments (demo)
- You can switch to sections with open space. http://goo.gl/nWfv7Z
- Michelle Hwang's sections (15, 18) are for students with little prior CS experience
- Videos are a mix of Fall 2013 and new content

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

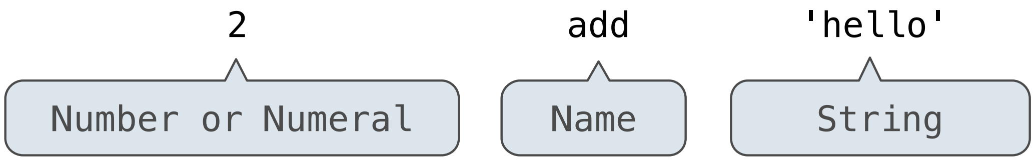
Primitive expressions: `2` `add`

Number or Numeral

Name

Types of Expressions

Primitive expressions:



2 add 'hello'

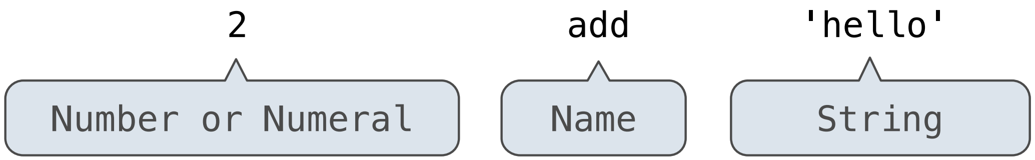
Number or Numeral

Name

String

Types of Expressions

Primitive expressions:



2

Number or Numeral

add

Name

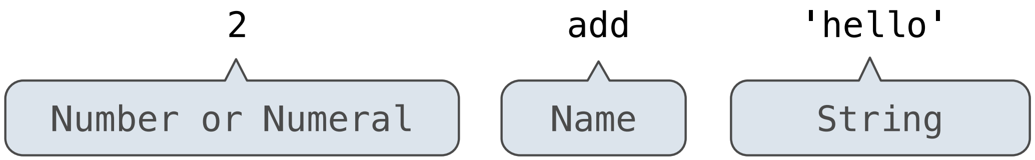
'hello'

String

Call expressions:

Types of Expressions

Primitive expressions:



`2` `add` `'hello'`

Number or Numeral

Name

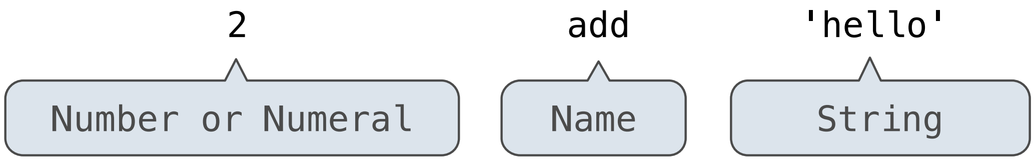
String

Call expressions:

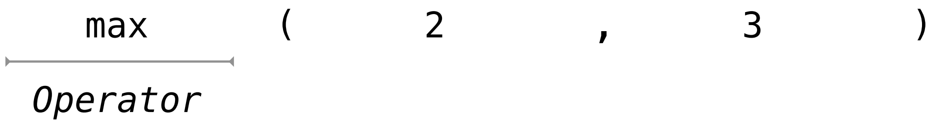
```
max     (     2     ,     3     )
```

Types of Expressions

Primitive expressions:

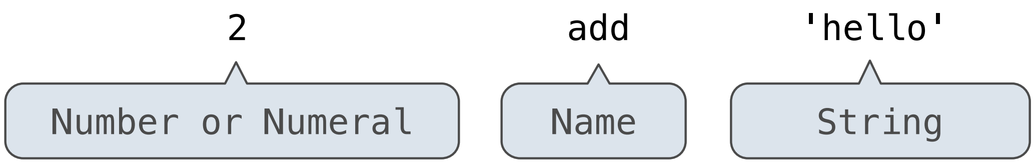


Call expressions:



Types of Expressions

Primitive expressions:



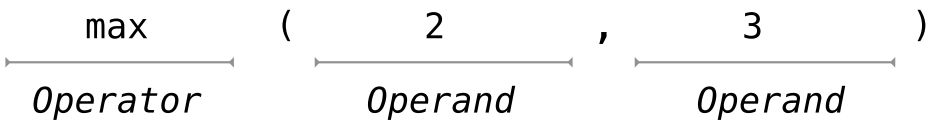
2 add 'hello'

Number or Numeral

Name

String

Call expressions:



max (2 , 3)

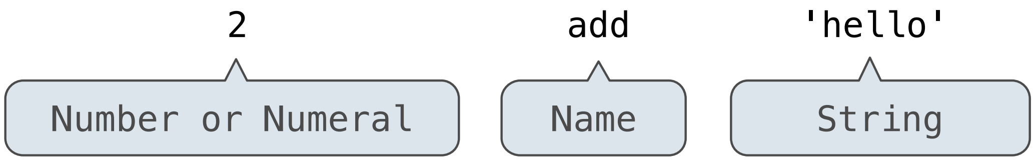
Operator

Operand

Operand

Types of Expressions

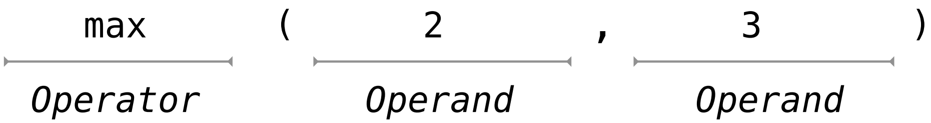
Primitive expressions:



2 add 'hello'

Number or Numeral Name String

Call expressions:



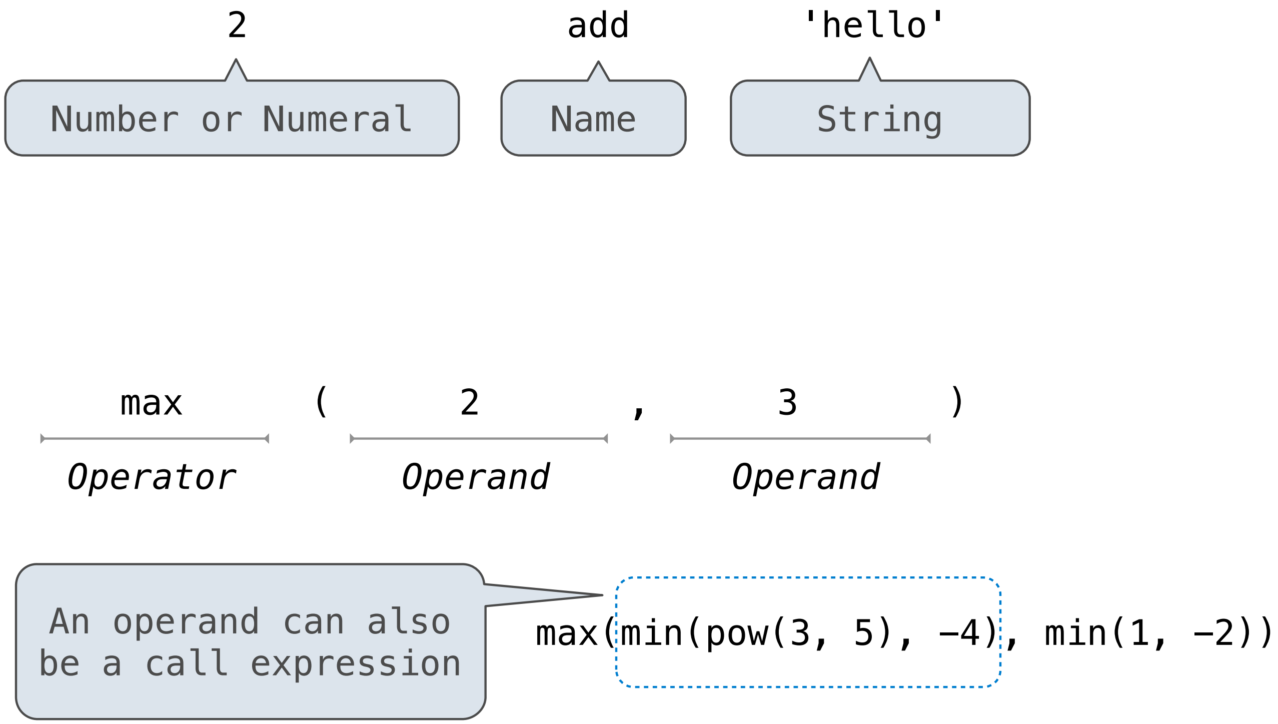
max (2 , 3)

Operator *Operand* *Operand*

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

Types of Expressions

Primitive expressions:



`2` — Number or Numeral

`add` — Name

`'hello'` — String

Call expressions:

`max ( 2 , 3 )`

max — *Operator*

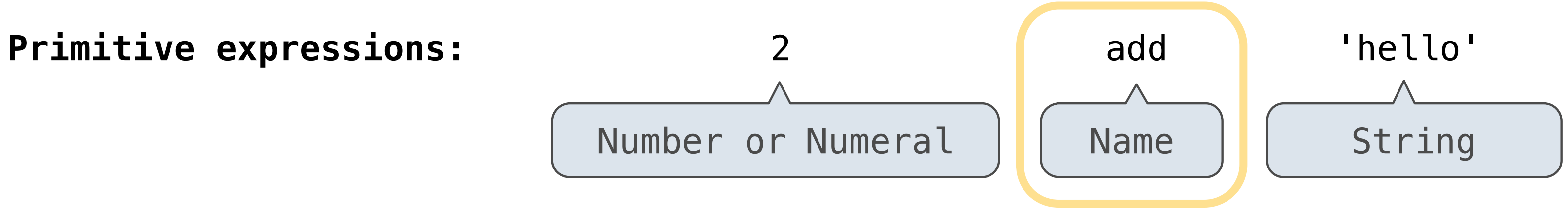
2 — *Operand*

3 — *Operand*

An operand can also be a call expression

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

Types of Expressions

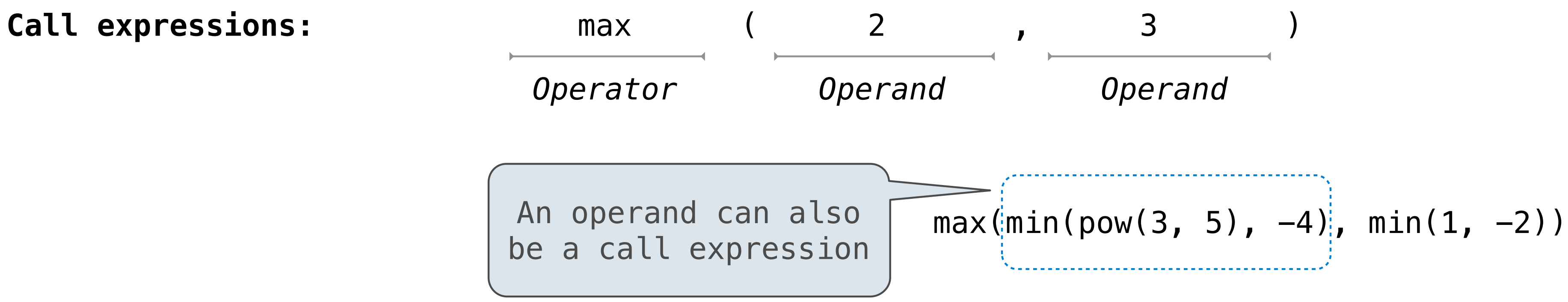


Primitive expressions:

`2` — Number or Numeral

`add` — Name

`'hello'` — String



Call expressions:

```
max      (     2      ,      3      )
Operator       Operand       Operand
```

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)
```

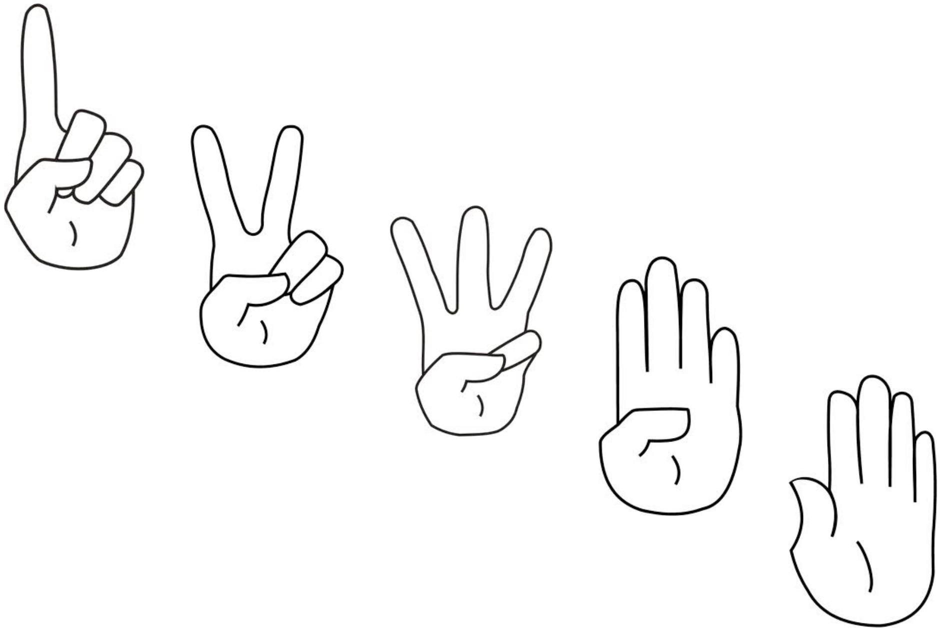
???

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

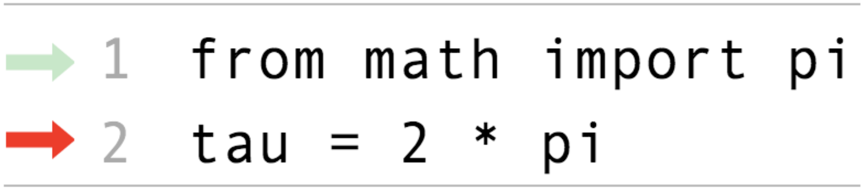
Environment Diagrams

Environment diagrams visualize the interpreter’s process.

Interactive Diagram

Environment Diagrams

Environment diagrams visualize the interpreter’s process.

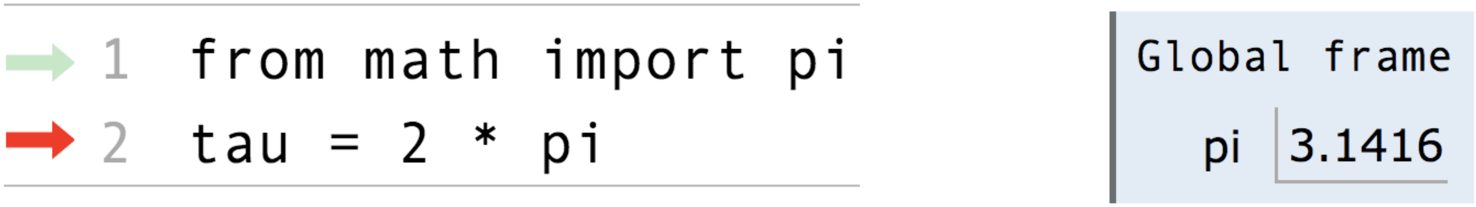


```
from math import pi
tau = 2 * pi
```

Interactive Diagram

Environment Diagrams

Environment diagrams visualize the interpreter’s process.



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

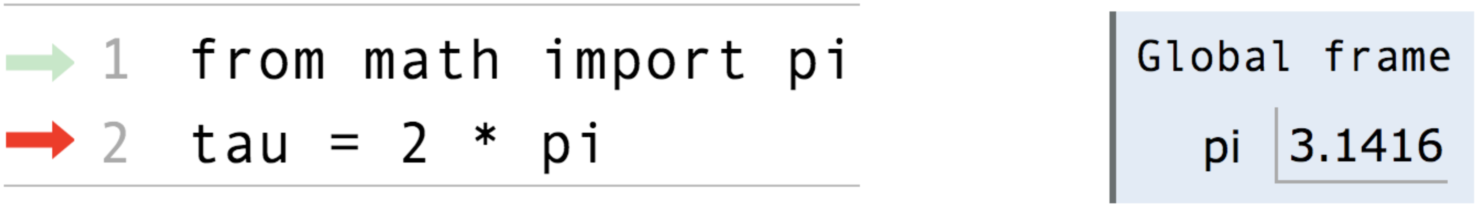
Global frame

pi 3.1416

Interactive Diagram

Environment Diagrams

Environment diagrams visualize the interpreter’s process.



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

Global frame

pi 3.1416

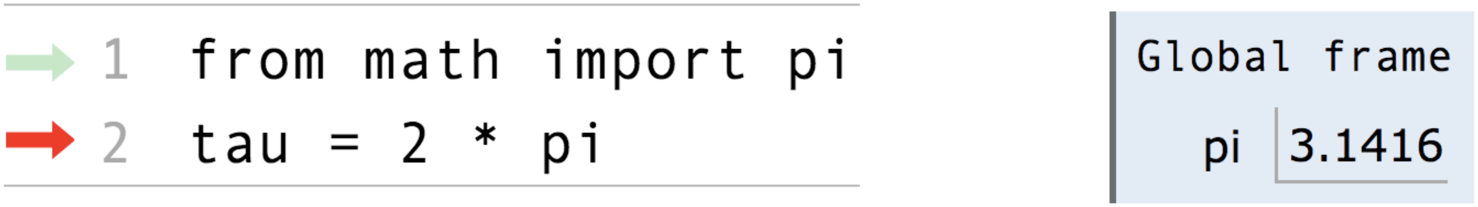
Code (left):

Frames (right):

Interactive Diagram

Environment Diagrams

Environment diagrams visualize the interpreter’s process.



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

Global frame

pi	3.1416

Code (left):

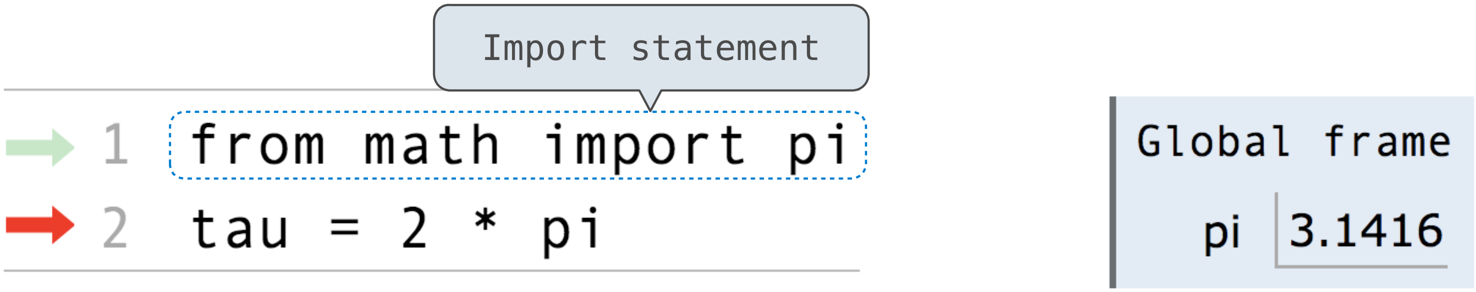
Statements and expressions

Frames (right):

Interactive Diagram

Environment Diagrams

Environment diagrams visualize the interpreter’s process.



Import statement

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

Code (left):

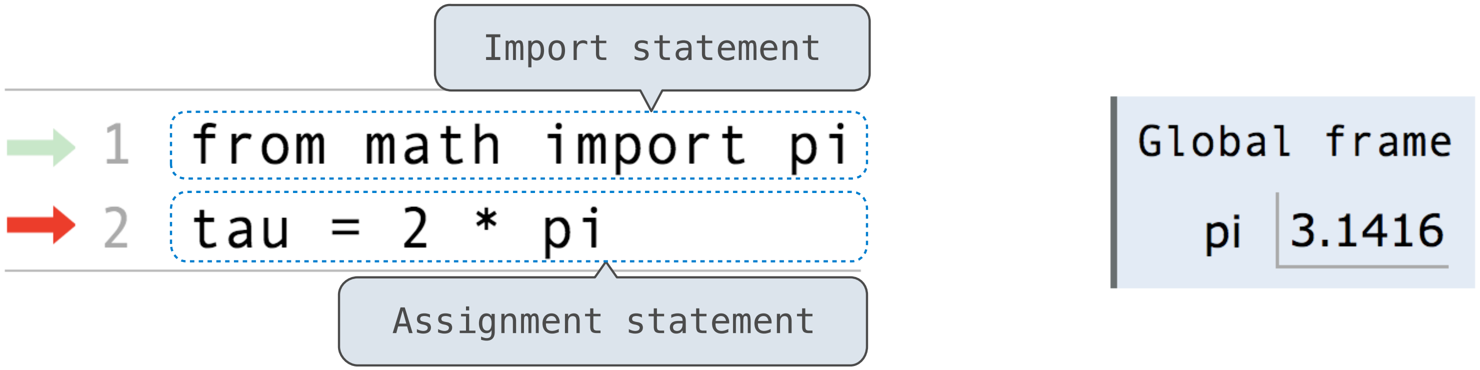
Statements and expressions

Frames (right):

Interactive Diagram

Environment Diagrams

Environment diagrams visualize the interpreter's process.



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

Code (left):

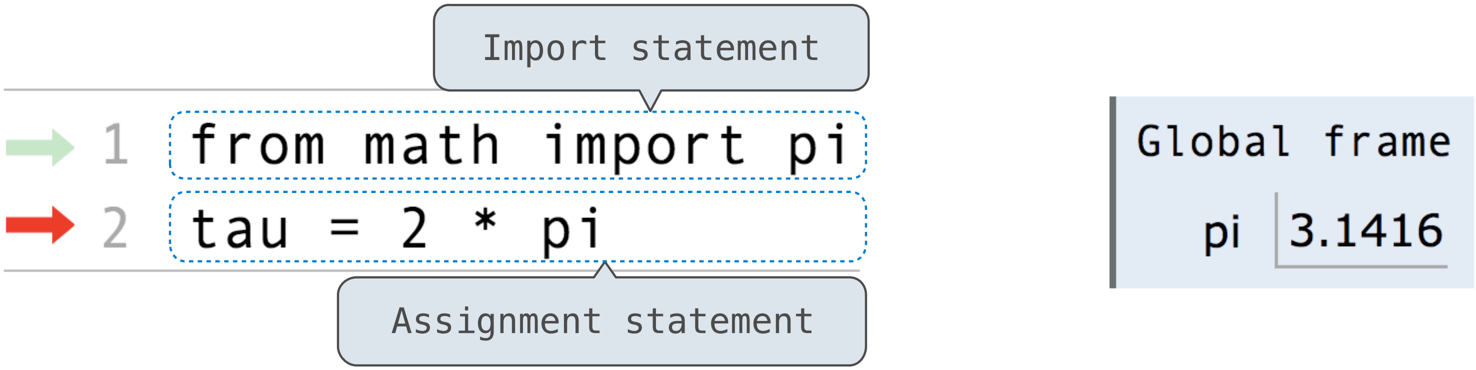
Frames (right):

Statements and expressions

Interactive Diagram

Environment Diagrams

Environment diagrams visualize the interpreter’s process.



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

Code (left):

Statements and expressions

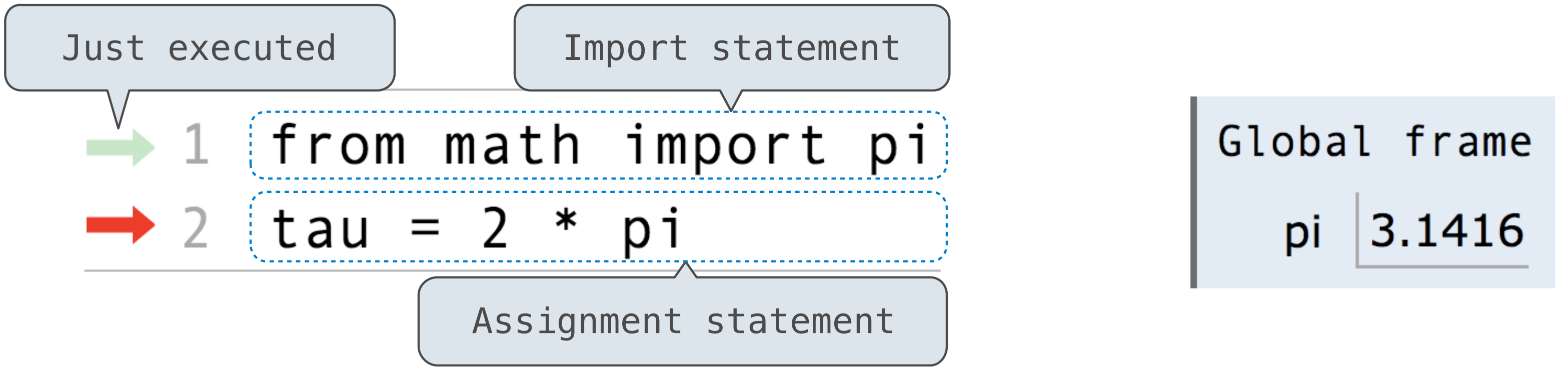
Arrows indicate evaluation order

Frames (right):

Interactive Diagram

Environment Diagrams

Environment diagrams visualize the interpreter’s process.



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

Code (left):

Statements and expressions

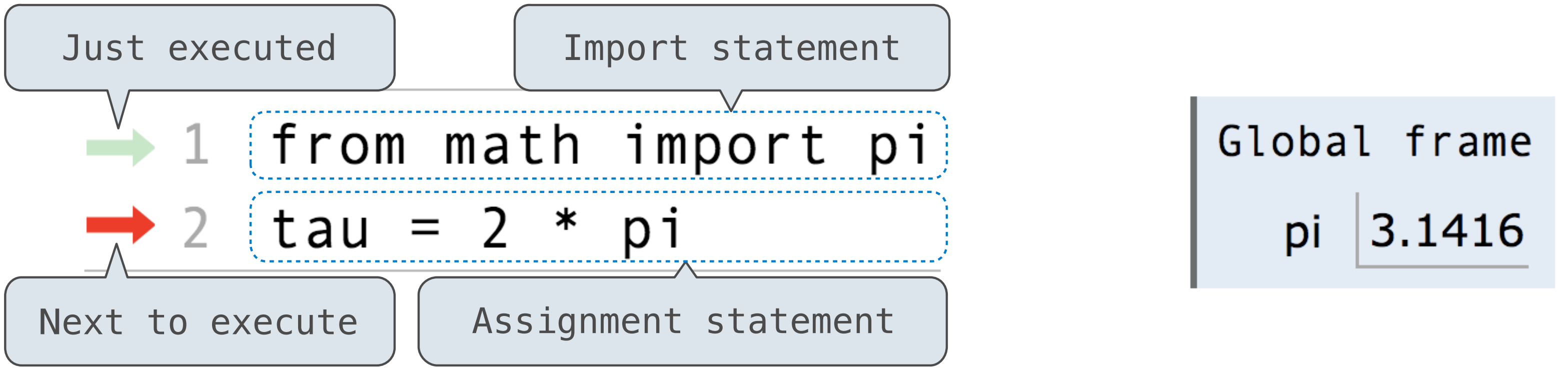
Arrows indicate evaluation order

Frames (right):

Interactive Diagram

Environment Diagrams

Environment diagrams visualize the interpreter’s process.



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

Code (left):

Statements and expressions

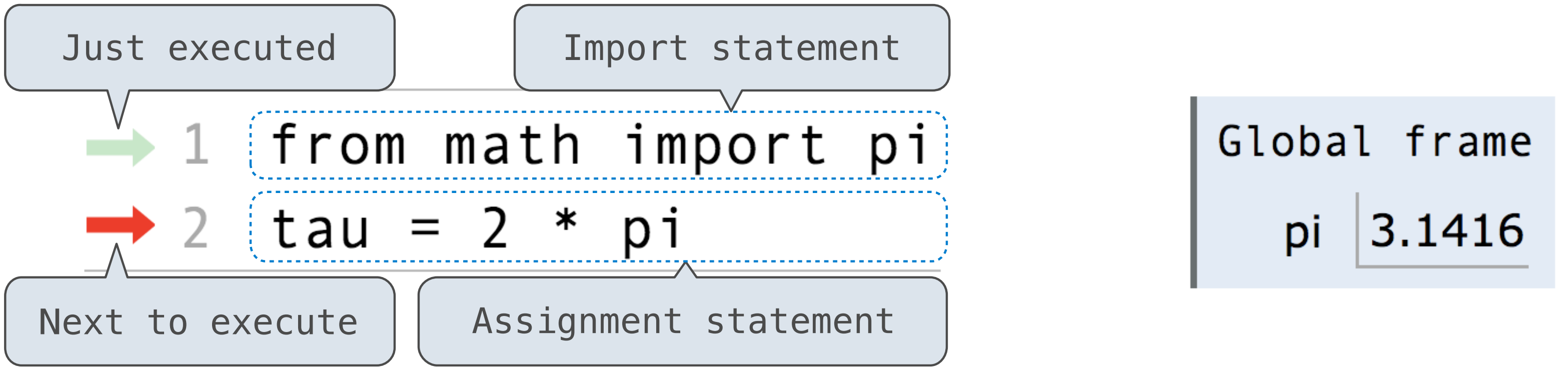
Arrows indicate evaluation order

Frames (right):

Interactive Diagram

Environment Diagrams

Environment diagrams visualize the interpreter’s process.



Just executed

Import statement

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

Next to execute

Assignment statement

Global frame

pi	3.1416

Code (left):

Statements and expressions

Arrows indicate evaluation order

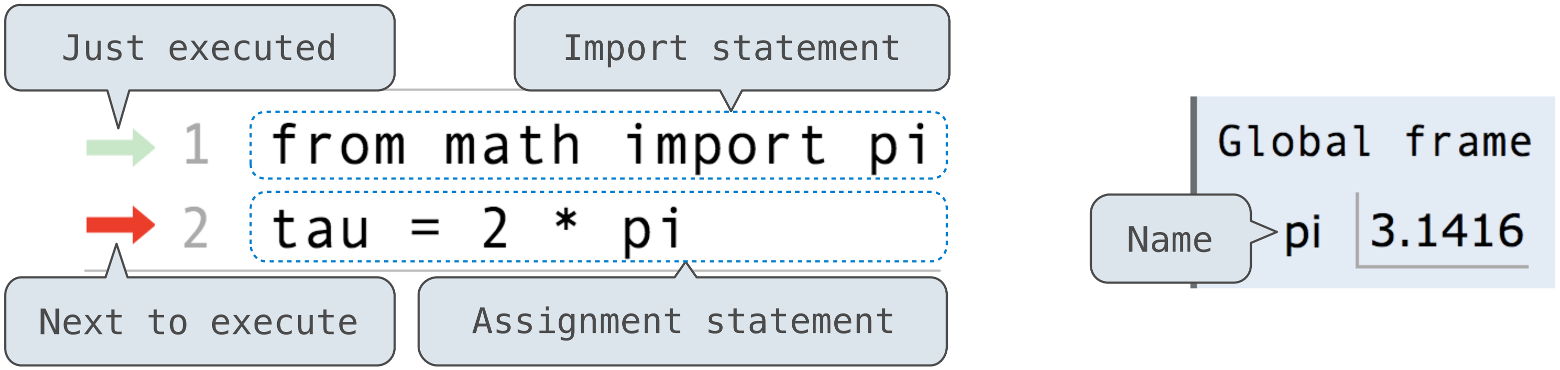
Frames (right):

Each name is bound to a value

Interactive Diagram

Environment Diagrams

Environment diagrams visualize the interpreter’s process.



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

Code (left):

Statements and expressions

Arrows indicate evaluation order

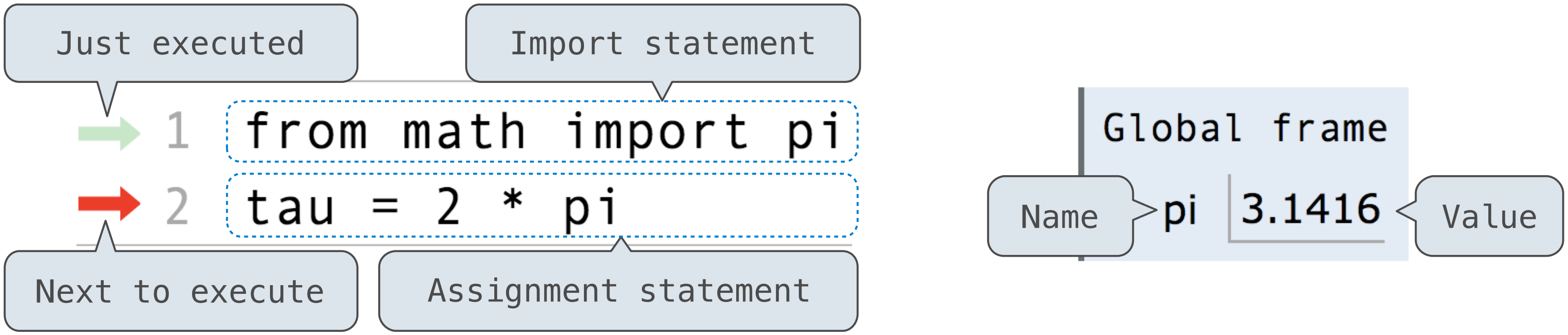
Frames (right):

Each name is bound to a value

Interactive Diagram

Environment Diagrams

Environment diagrams visualize the interpreter’s process.



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

Code (left):

Statements and expressions

Arrows indicate evaluation order

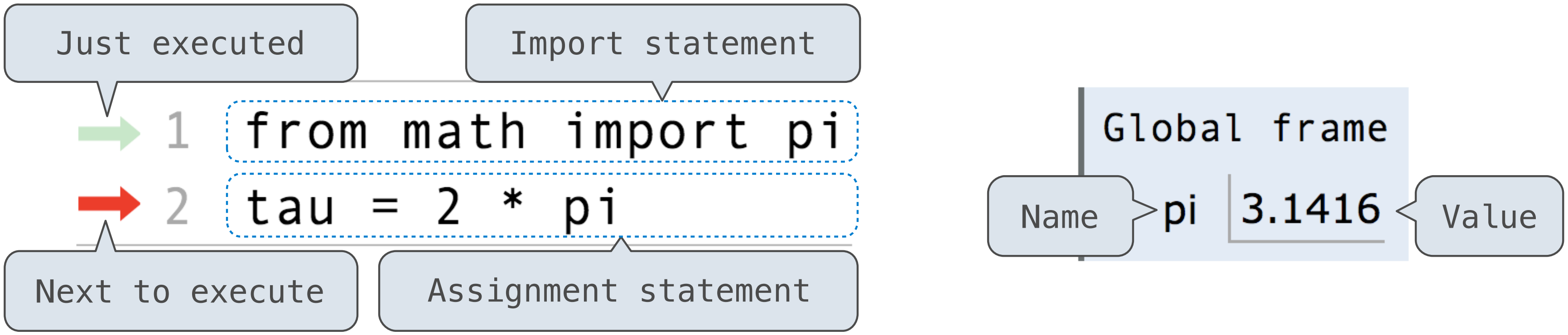
Frames (right):

Each name is bound to a value

Interactive Diagram

Environment Diagrams

Environment diagrams visualize the interpreter’s process.



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

Code (left):

Statements and expressions

Arrows indicate evaluation order

Frames (right):

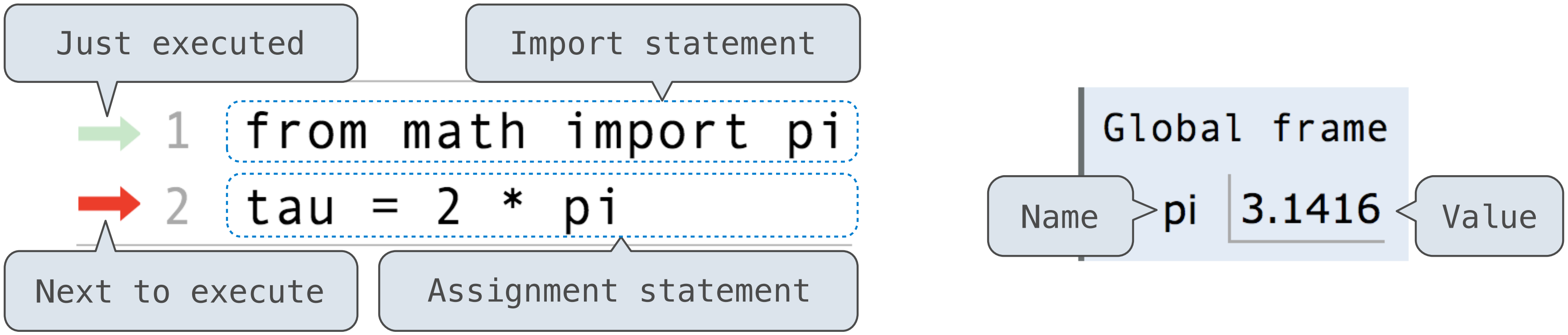
Each name is bound to a value

Within a frame, a name cannot be repeated

Interactive Diagram

Environment Diagrams

Environment diagrams visualize the interpreter’s process.



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

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

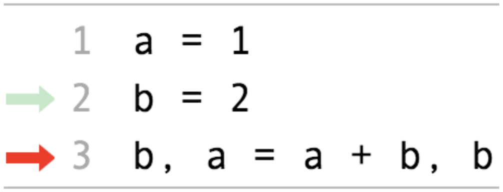
(Demo)

Interactive Diagram

Assignment Statements

Interactive Diagram

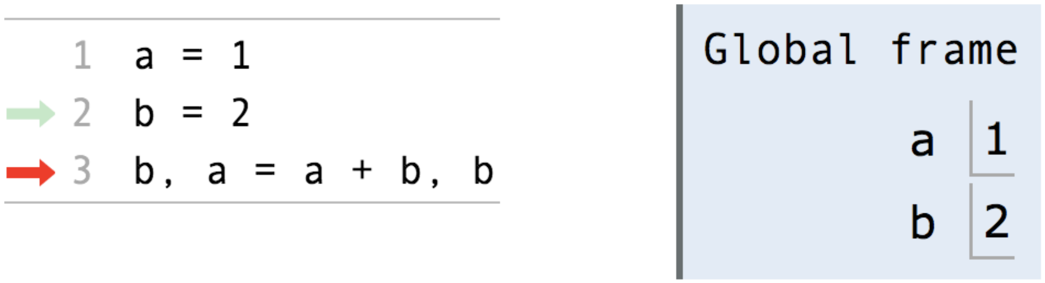
Assignment Statements



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

Interactive Diagram

Assignment Statements



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

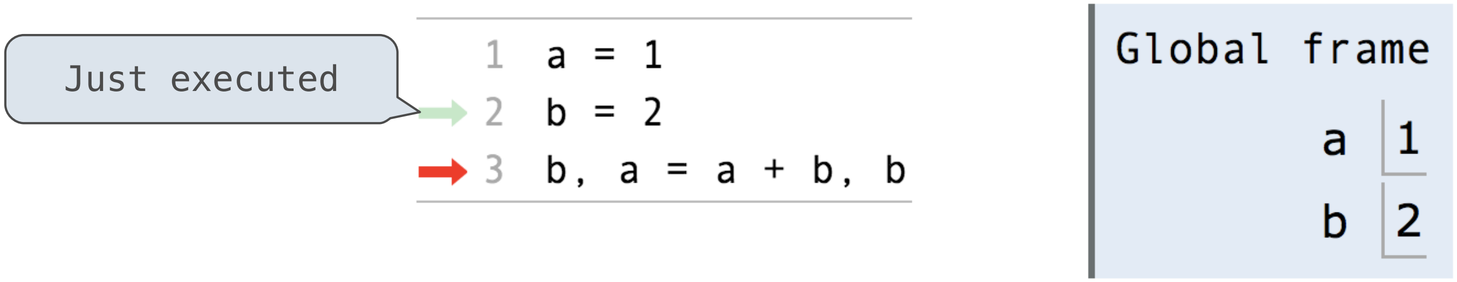
Global frame

a 1

b 2

Interactive Diagram

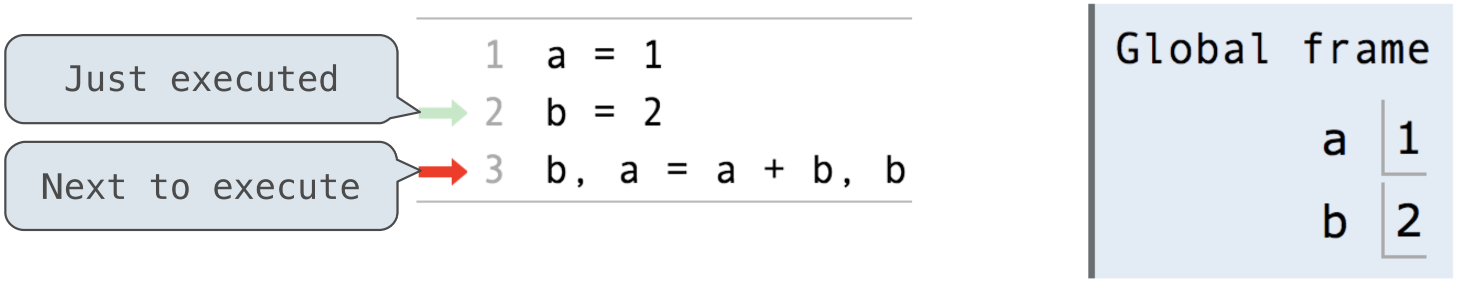
Assignment Statements



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

Interactive Diagram

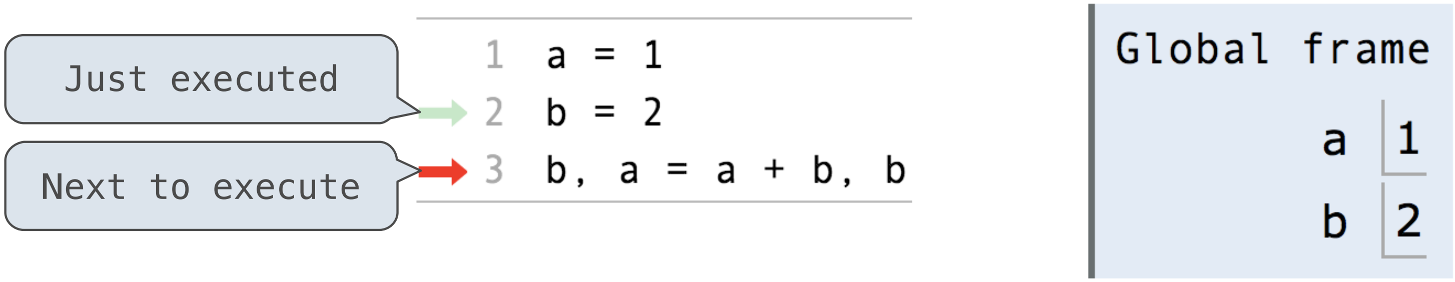
Assignment Statements



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

Interactive Diagram

Assignment Statements

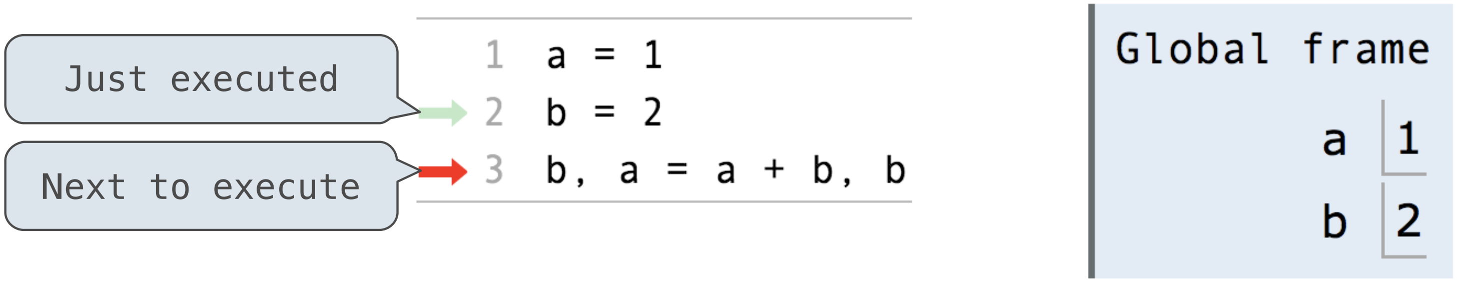


```
a = 1
b = 2
b, a = a + b, b
```

Execution rule for assignment statements:

Interactive Diagram

Assignment Statements



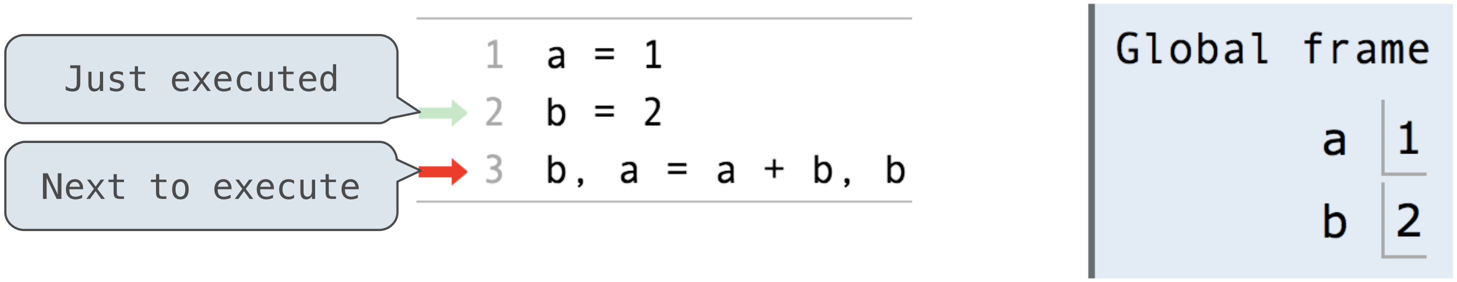
```
a = 1
b = 2
b, a = a + b, b
```

Execution rule for assignment statements:

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

Interactive Diagram

Assignment Statements



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

Global frame

a	1
b	2

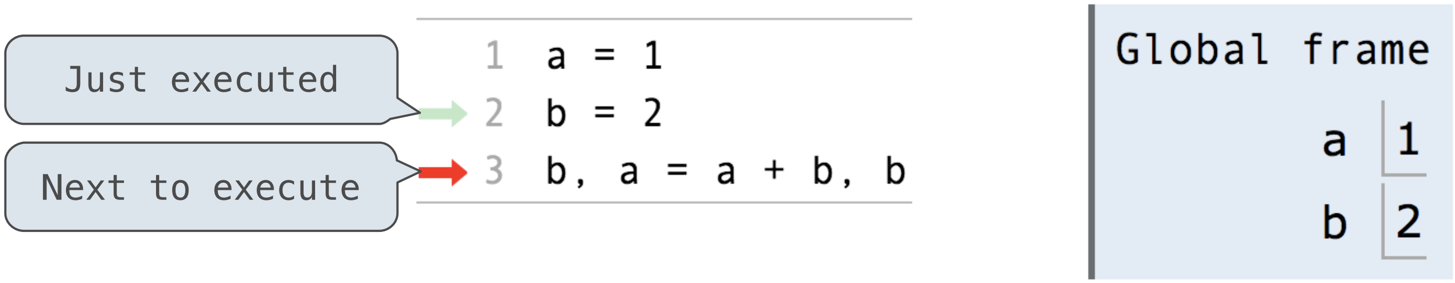
Execution rule for assignment statements:

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

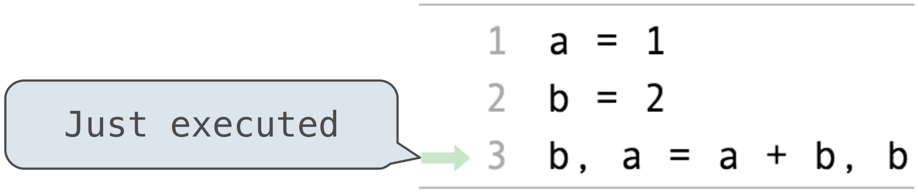
2. Bind all names to the left of = to the resulting values in the current frame.

Interactive Diagram

Assignment Statements



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



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

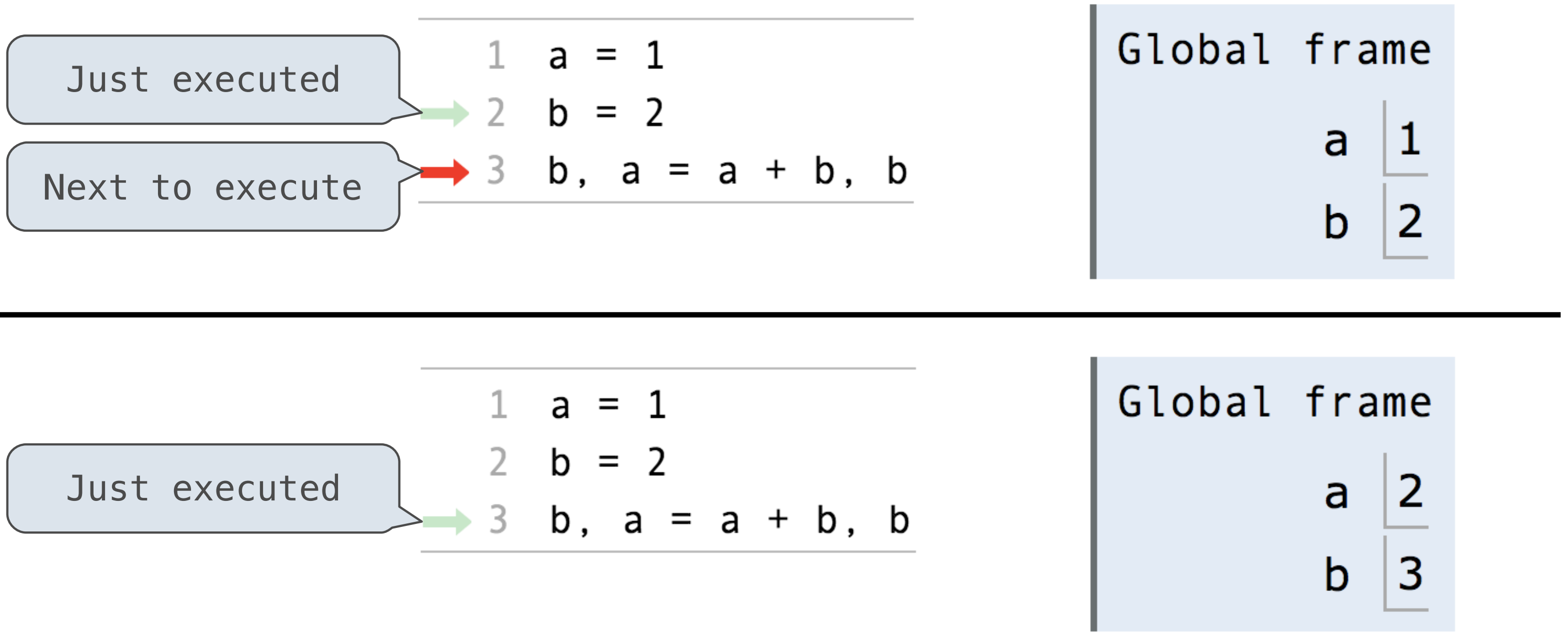
Execution rule for assignment statements:

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

2. Bind all names to the left of = to the resulting values in the current frame.

Interactive Diagram

Assignment Statements



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

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

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 the resulting values in the current frame.

Interactive Diagram

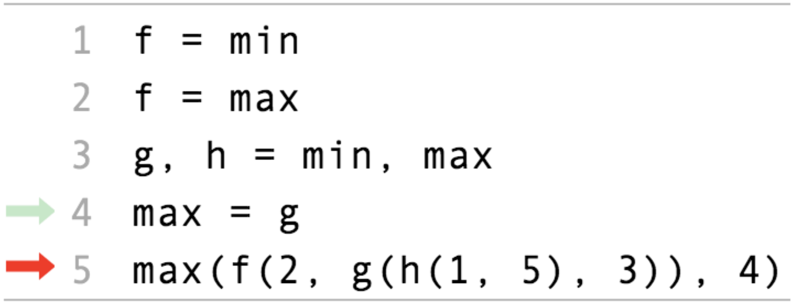
Discussion Question 1 Solution

(Demo)

Interactive Diagram

Discussion Question 1 Solution

(Demo)

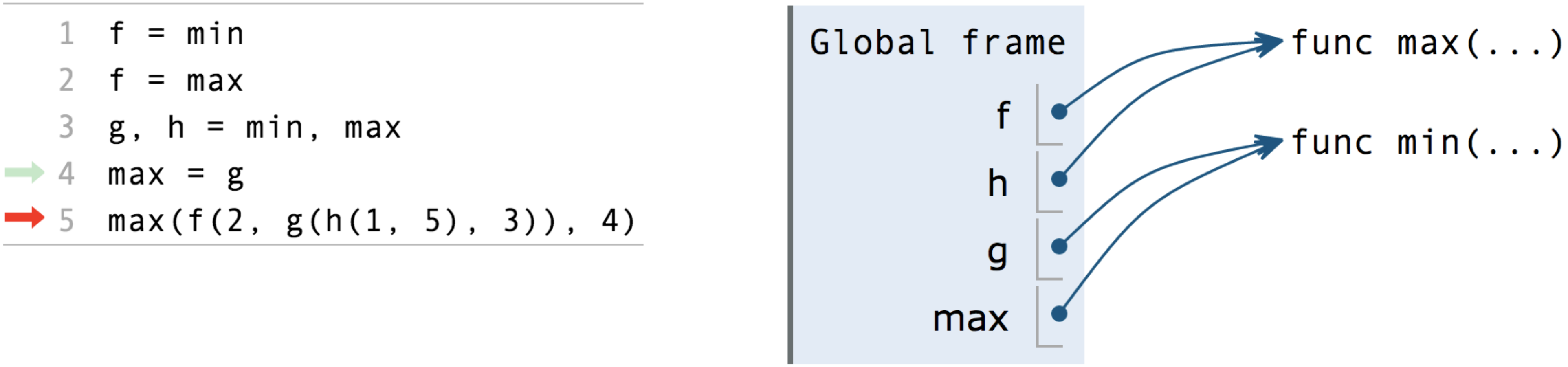


```
f = min
f = max
g, h = min, max
max = g
max(f(2, g(h(1, 5), 3)), 4)
```

Interactive Diagram

Discussion Question 1 Solution

(Demo)

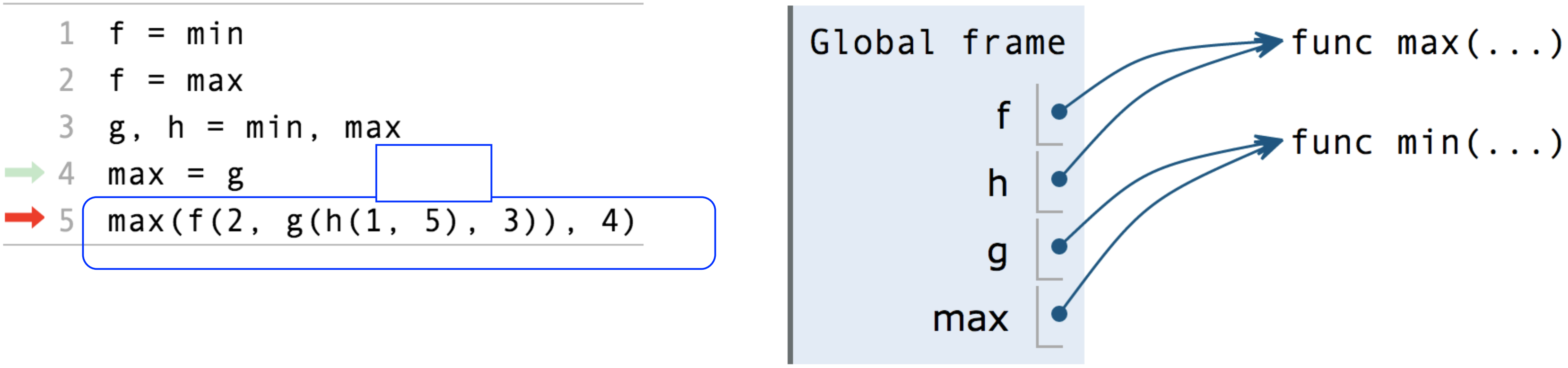


```python
f = min
f = max
g, h = min, max
max = g
max(f(2, g(h(1, 5), 3)), 4)
```

Interactive Diagram

Discussion Question 1 Solution

(Demo)

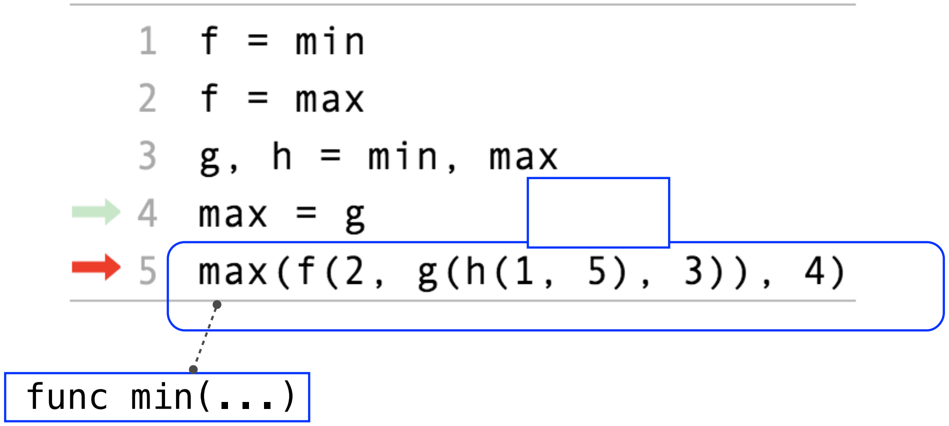


```
f = min
f = max
g, h = min, max
max = g
max(f(2, g(h(1, 5), 3)), 4)
```

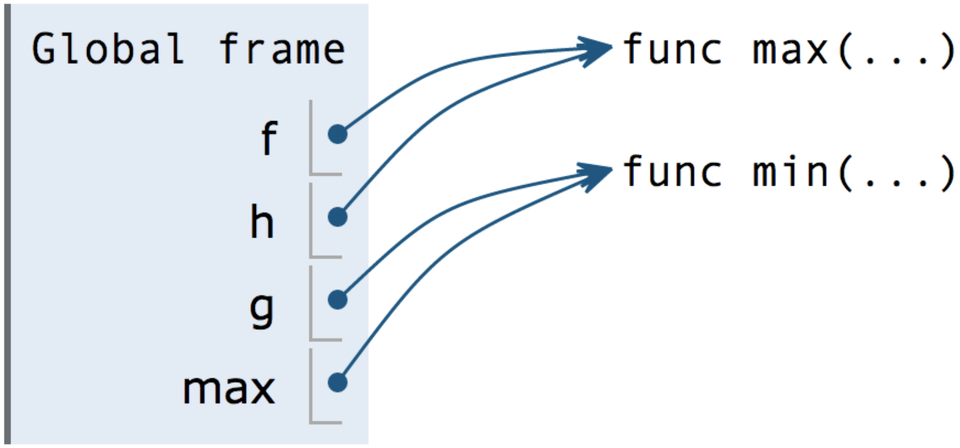
Interactive Diagram

Discussion Question 1 Solution

(Demo)



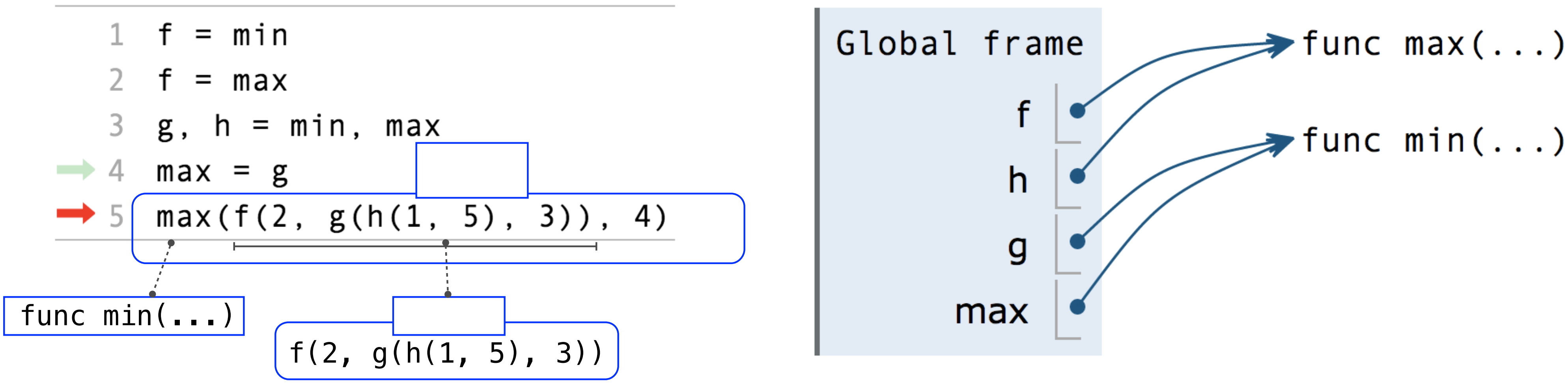
```
f = min
f = max
g, h = min, max
max = g
max(f(2, g(h(1, 5), 3)), 4)
```



Interactive Diagram

Discussion Question 1 Solution

(Demo)

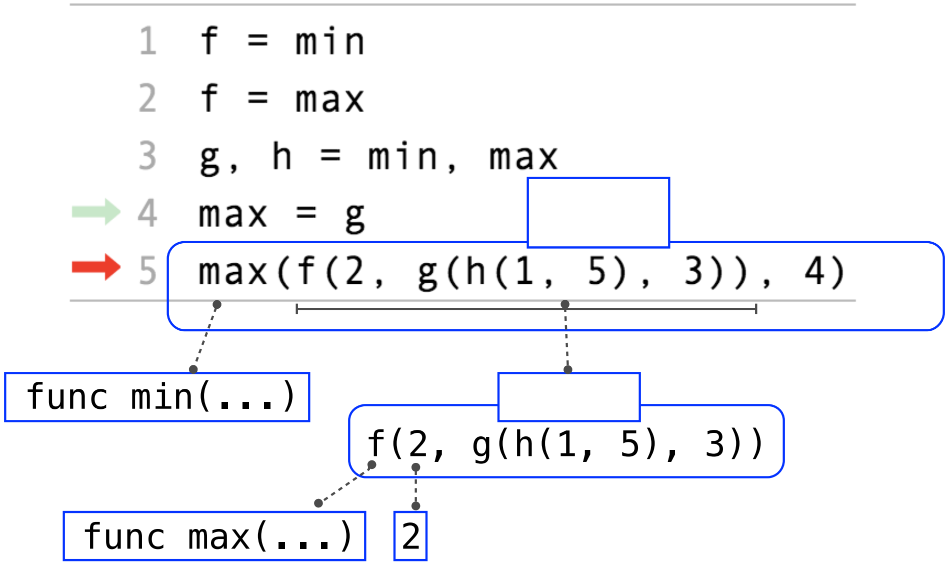


```
f = min
f = max
g, h = min, max
max = g
max(f(2, g(h(1, 5), 3)), 4)
```

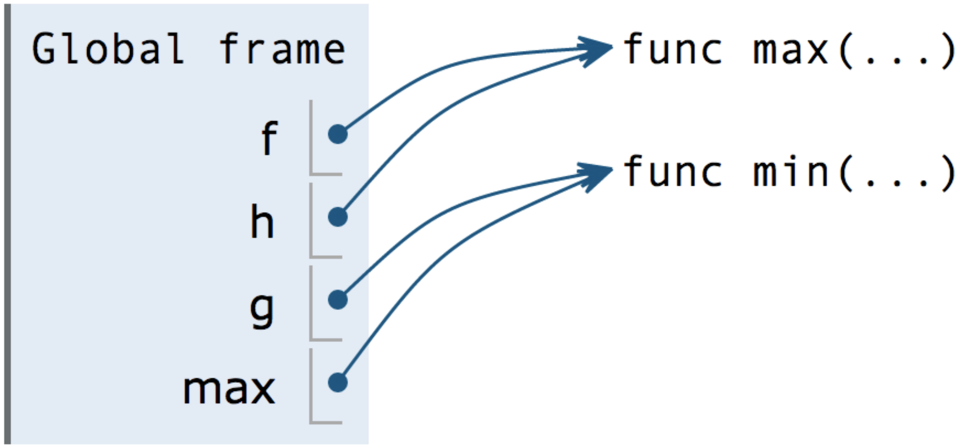
Interactive Diagram

Discussion Question 1 Solution

(Demo)



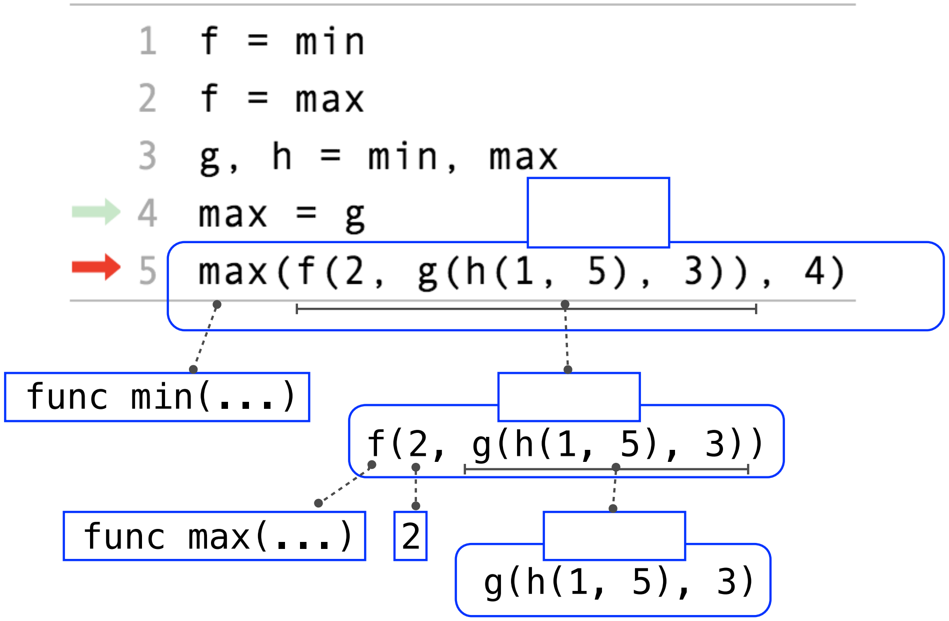
```
f = min
f = max
g, h = min, max
max = g
max(f(2, g(h(1, 5), 3)), 4)
```



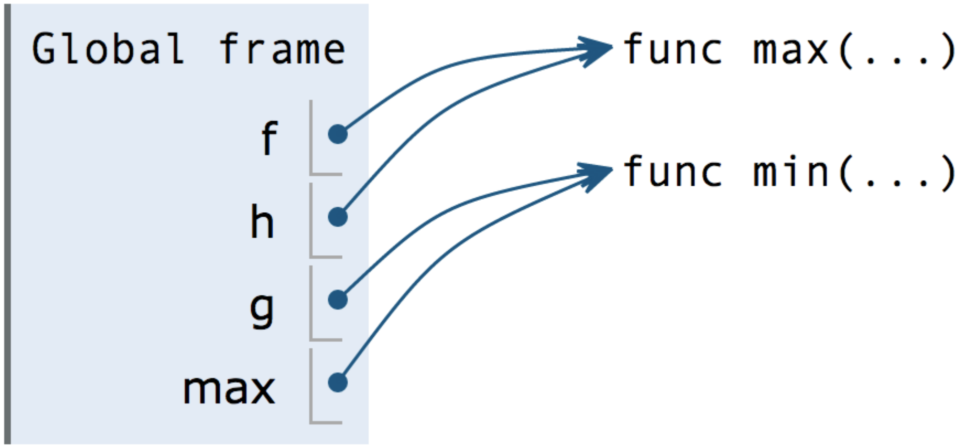
Interactive Diagram

Discussion Question 1 Solution

(Demo)



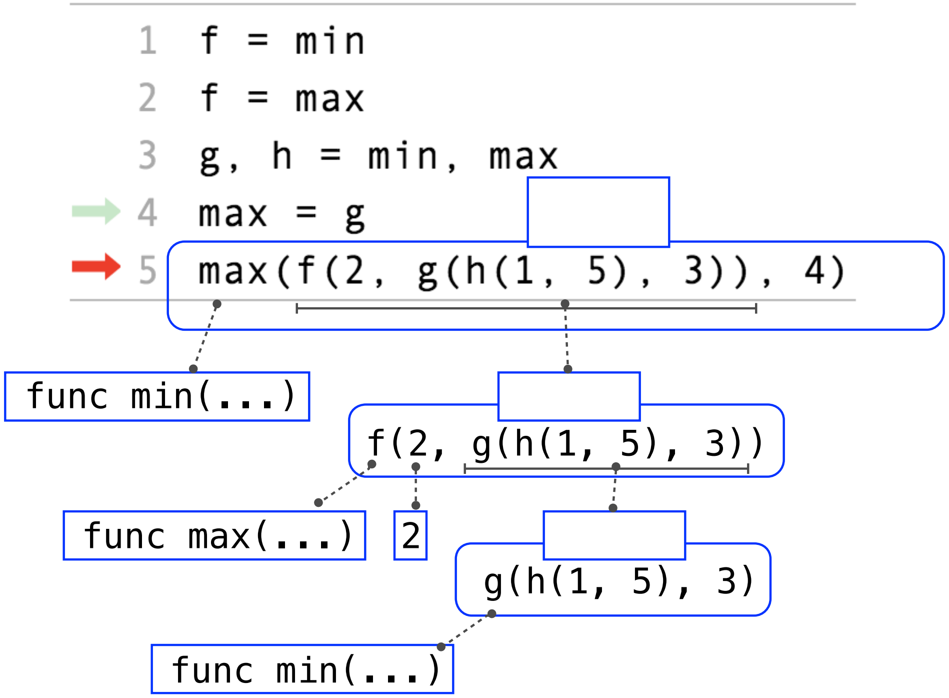
```
f = min
f = max
g, h = min, max
max = g
max(f(2, g(h(1, 5), 3)), 4)
```



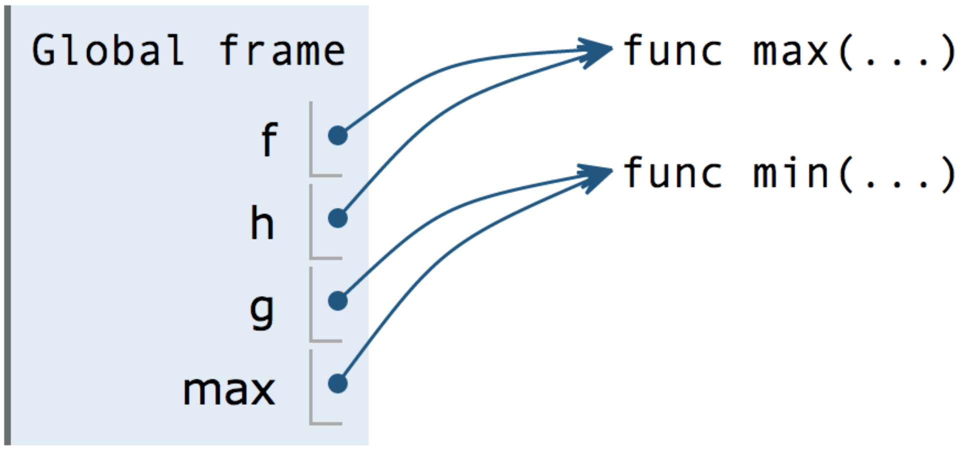
Interactive Diagram

Discussion Question 1 Solution

(Demo)



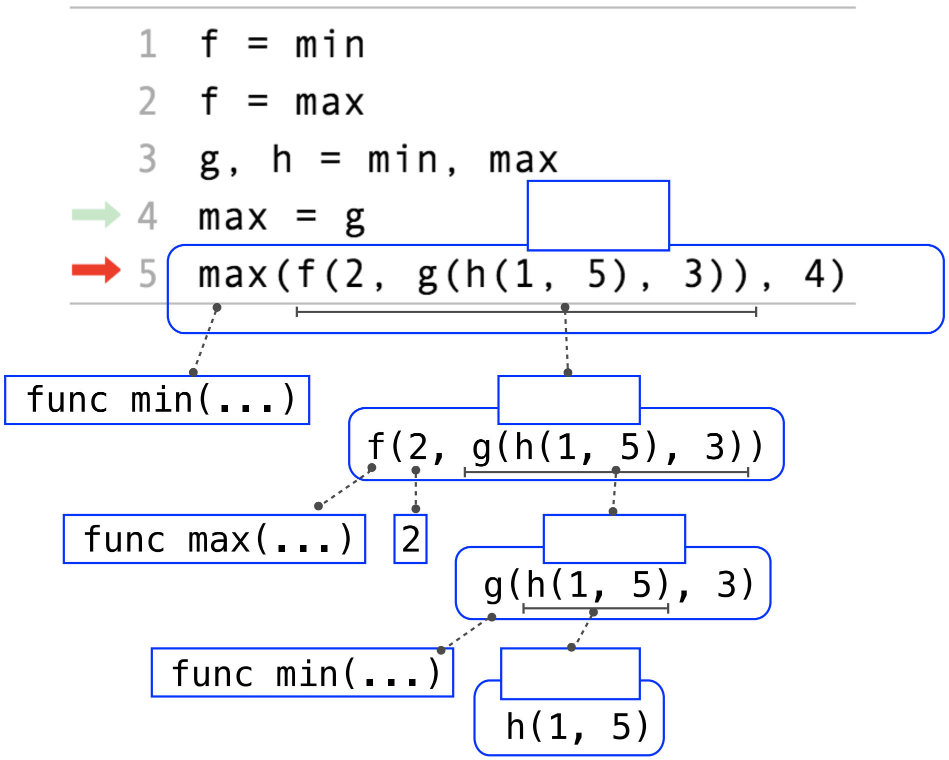
```
f = min
f = max
g, h = min, max
max = g
max(f(2, g(h(1, 5), 3)), 4)
```



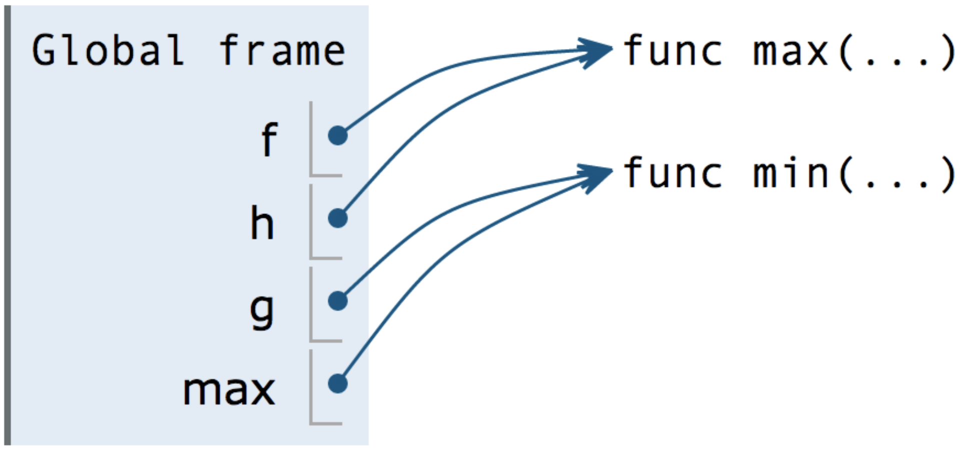
Interactive Diagram

Discussion Question 1 Solution

(Demo)



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

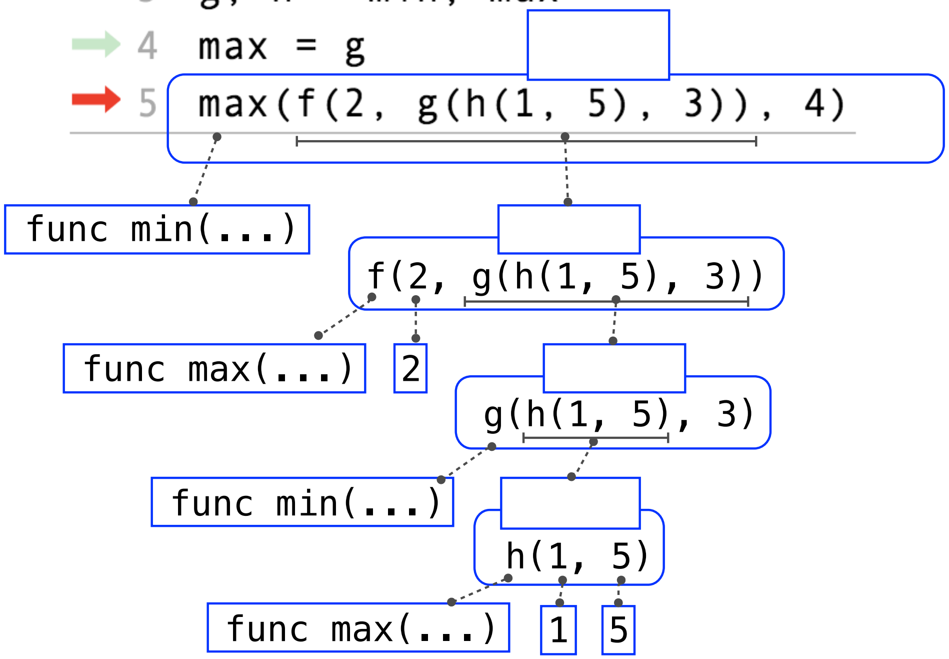


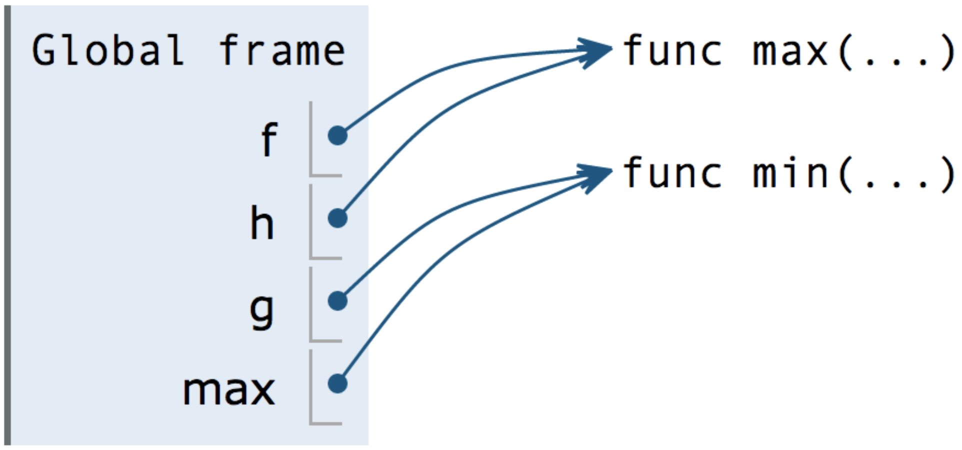
Interactive Diagram

Discussion Question 1 Solution

(Demo)

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

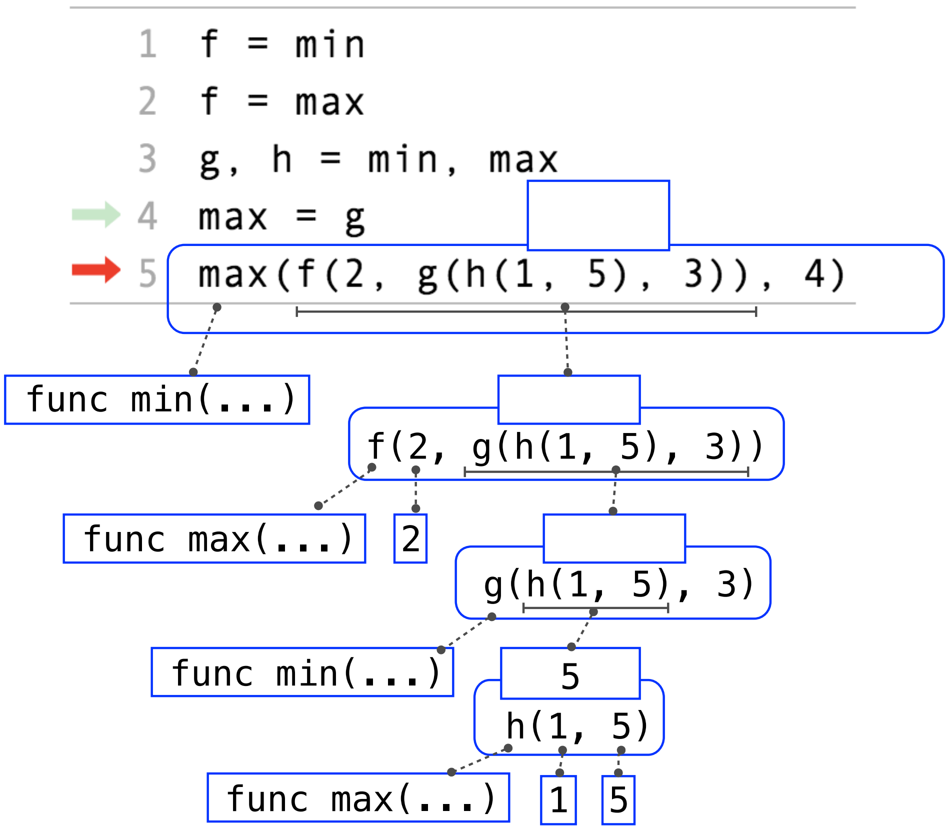




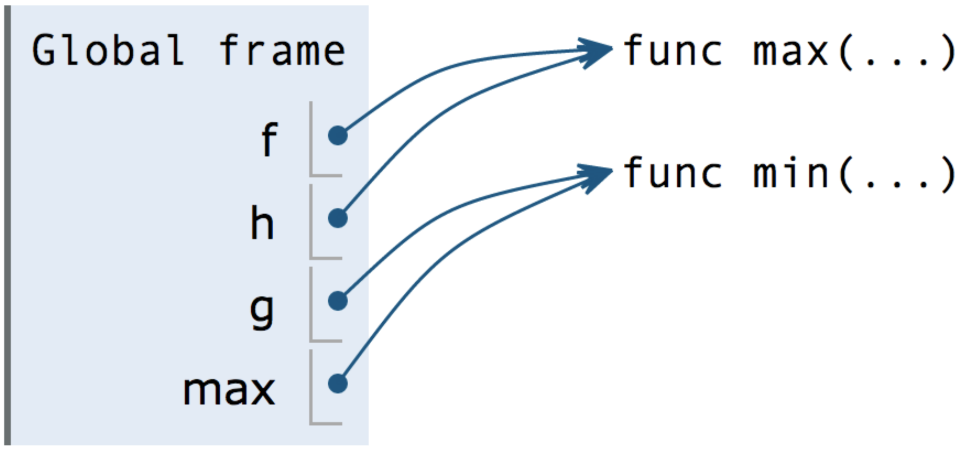
Interactive Diagram

Discussion Question 1 Solution

(Demo)



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

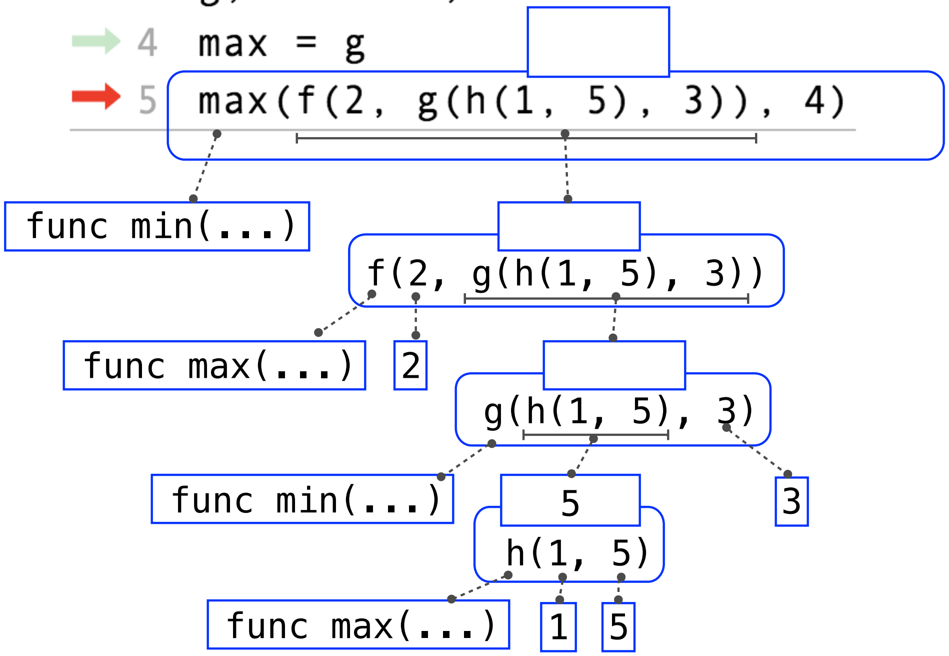


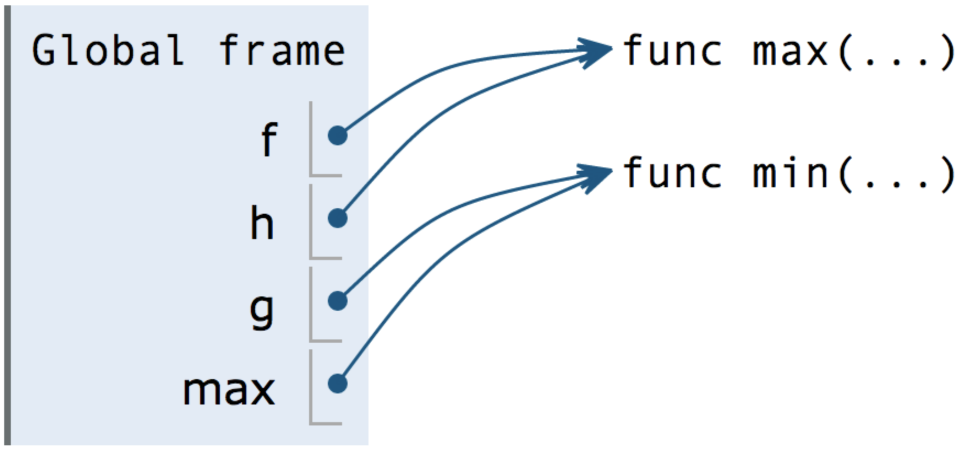
Interactive Diagram

Discussion Question 1 Solution

(Demo)

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

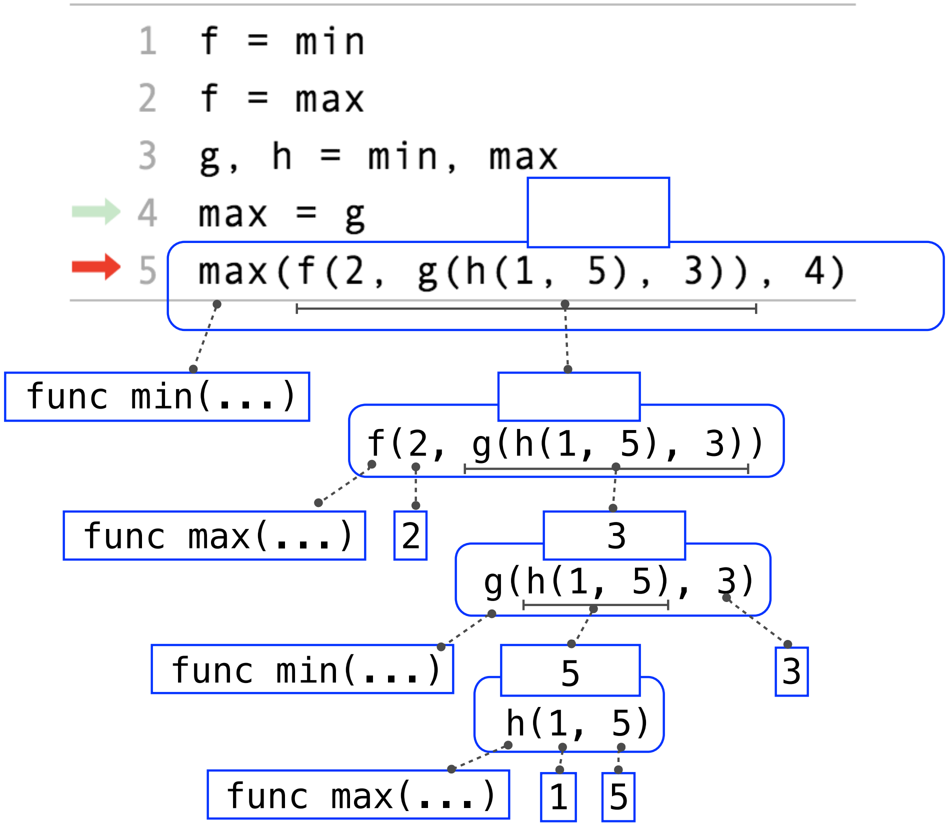




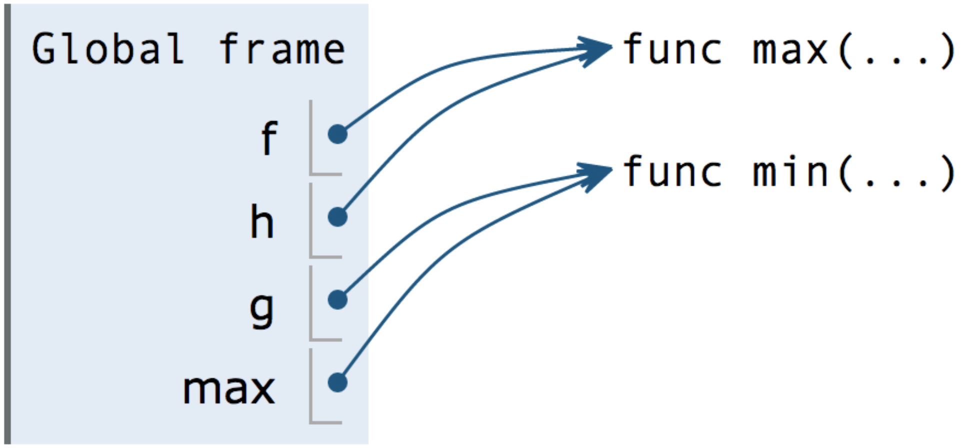
Interactive Diagram

Discussion Question 1 Solution

(Demo)



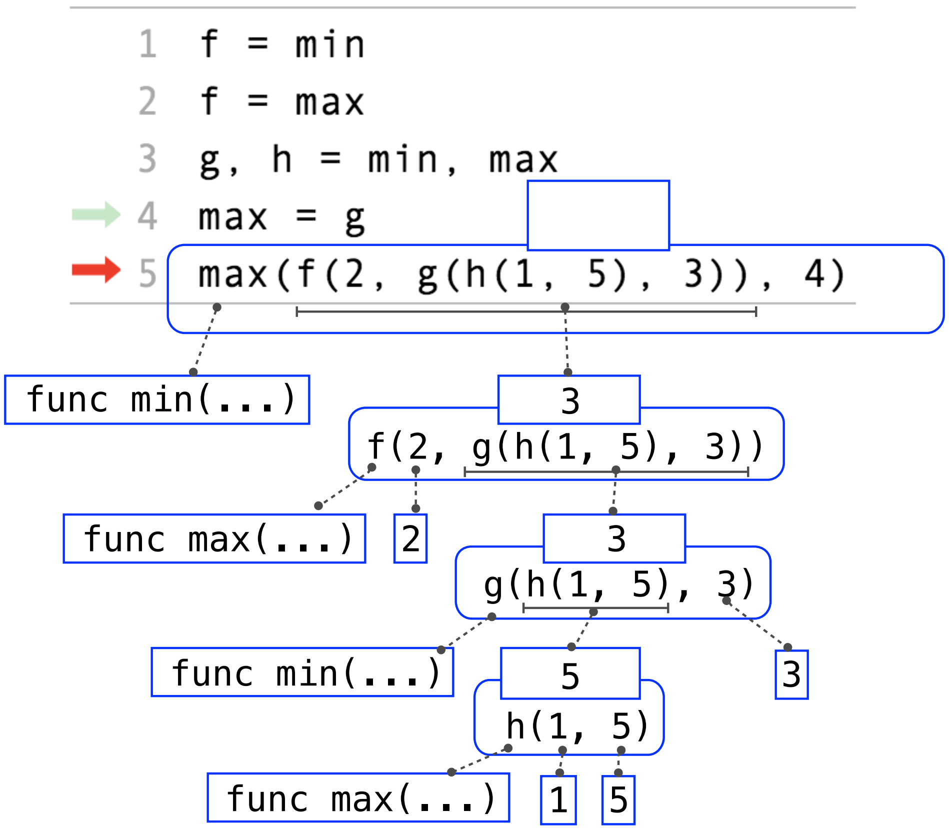
```
f = min
f = max
g, h = min, max
max = g
max(f(2, g(h(1, 5), 3)), 4)
```



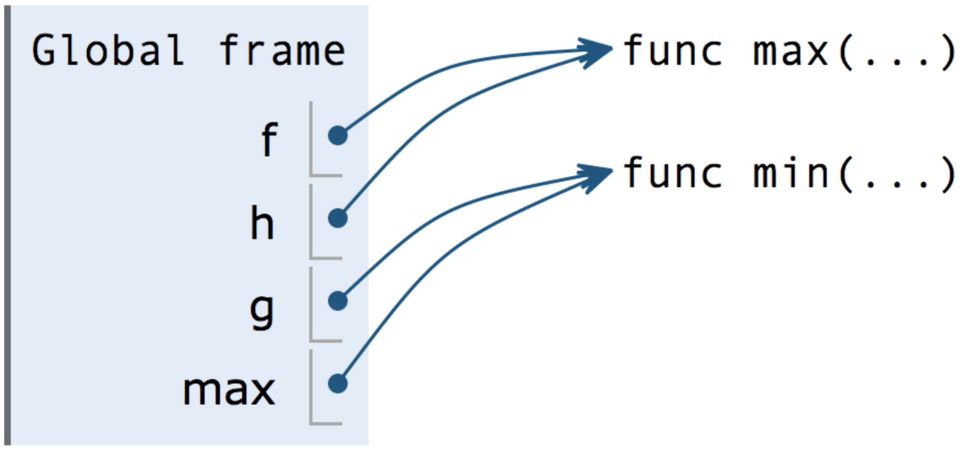
Interactive Diagram

Discussion Question 1 Solution

(Demo)



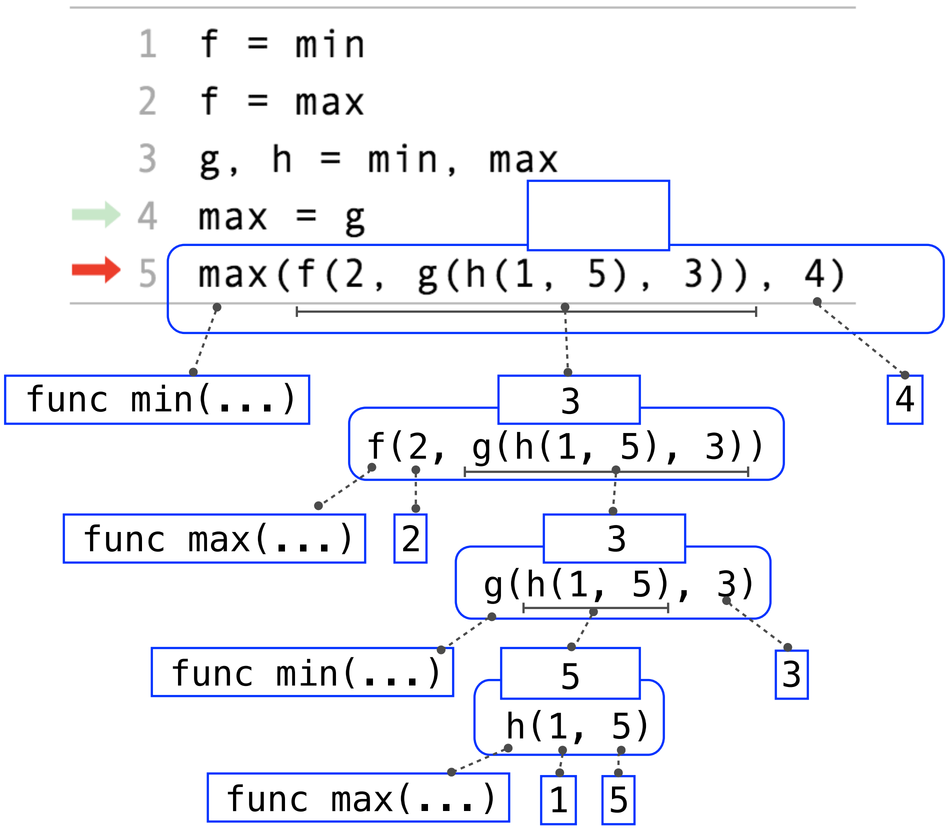
```
f = min
f = max
g, h = min, max
max = g
max(f(2, g(h(1, 5), 3)), 4)
```



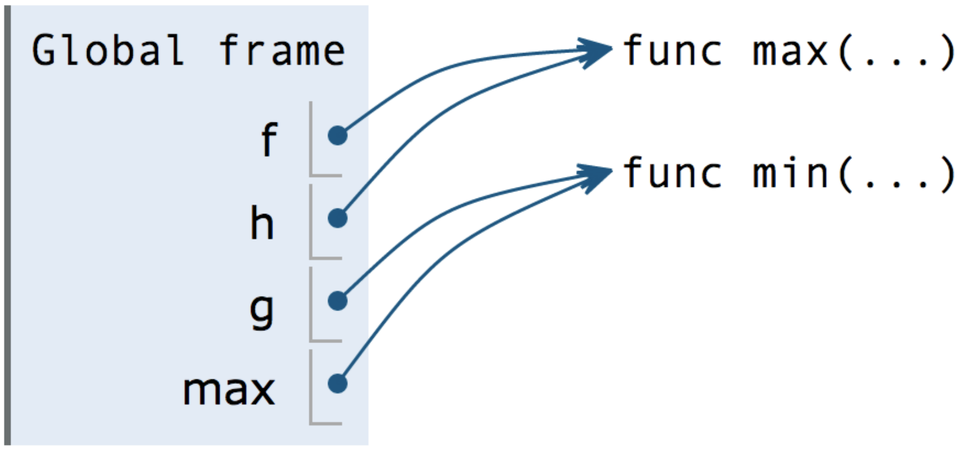
Interactive Diagram

Discussion Question 1 Solution

(Demo)



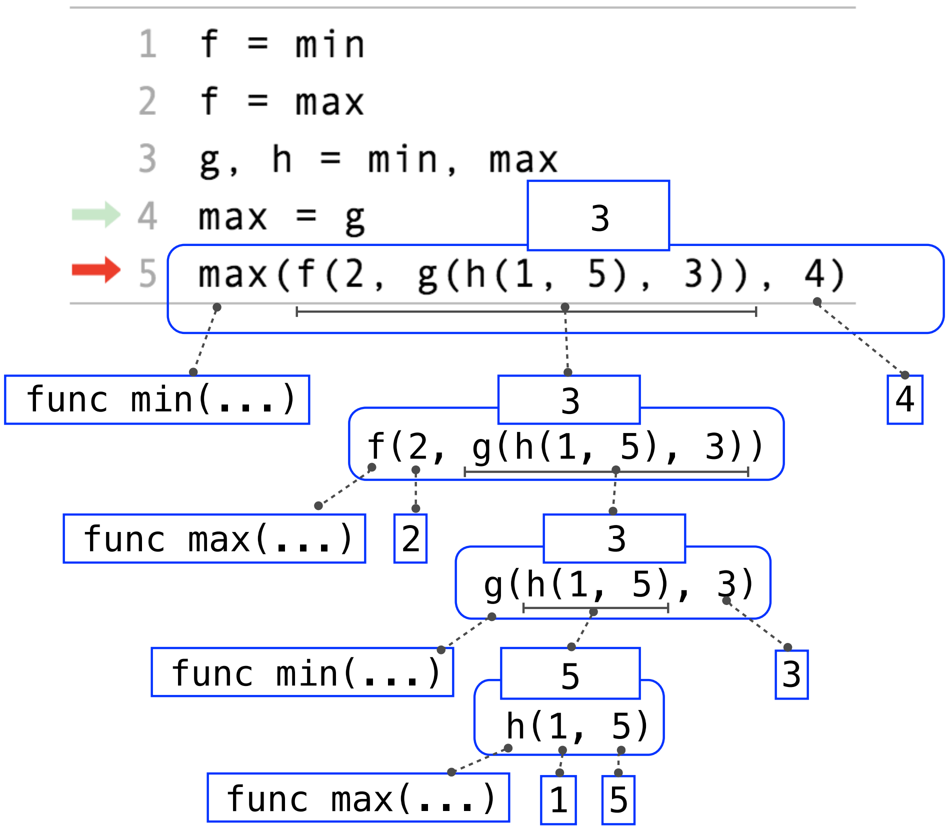
```
f = min
f = max
g, h = min, max
max = g
max(f(2, g(h(1, 5), 3)), 4)
```



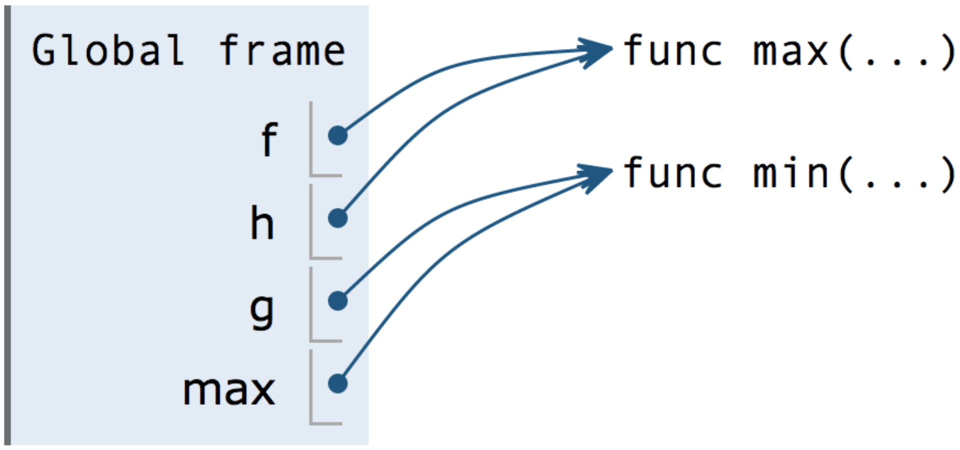
Interactive Diagram

Discussion Question 1 Solution

(Demo)



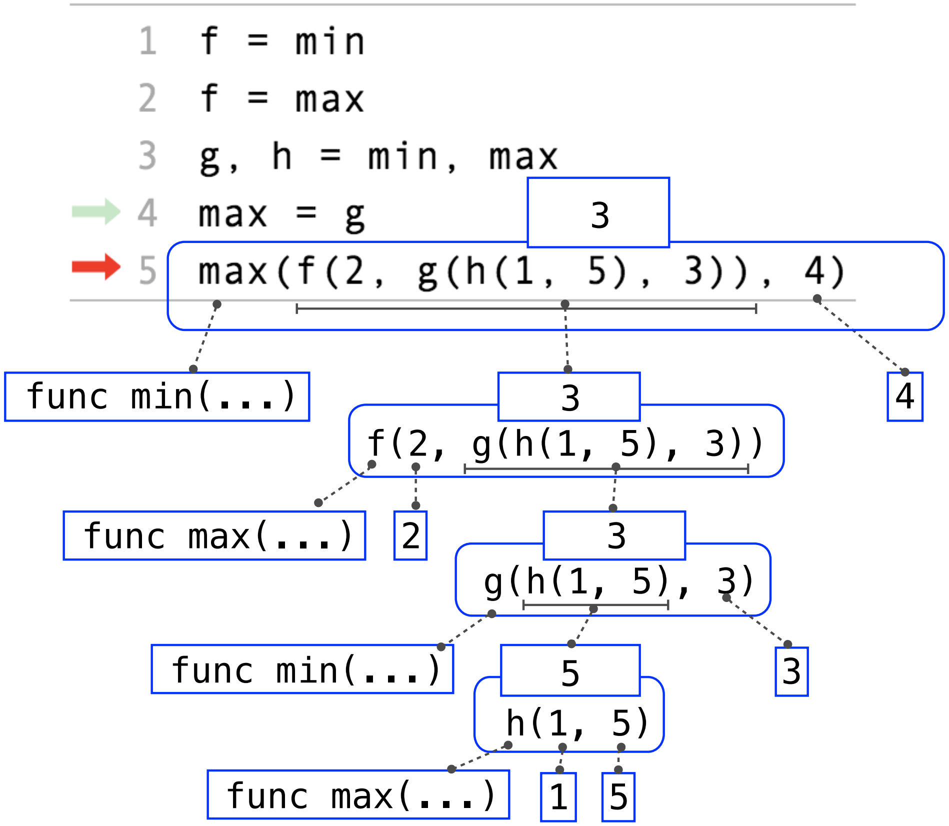
```
f = min
f = max
g, h = min, max
max = g
max(f(2, g(h(1, 5), 3)), 4)
```



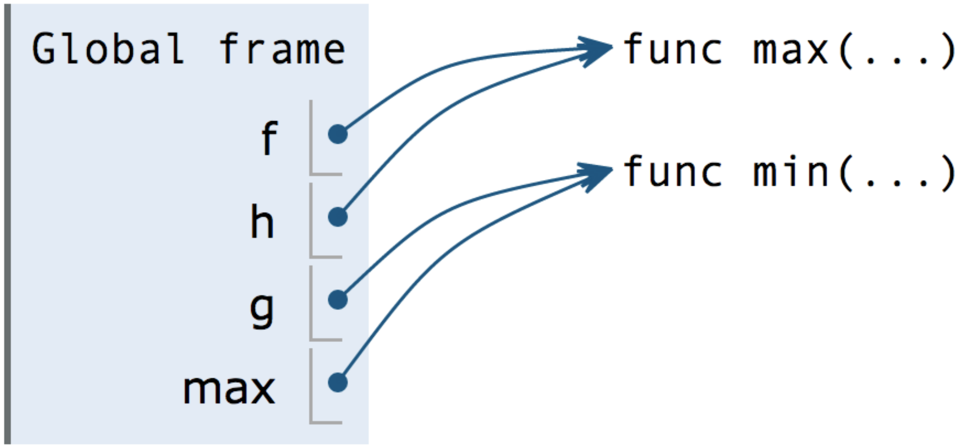
Interactive Diagram

Discussion Question 1 Solution

(Demo)



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



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

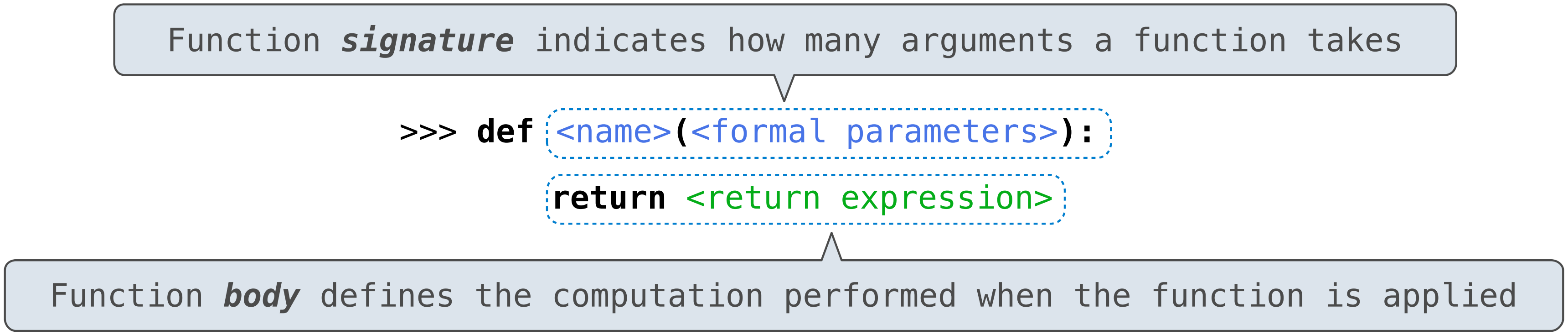
Function ***signature*** indicates how many arguments a function takes

```
>>> 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

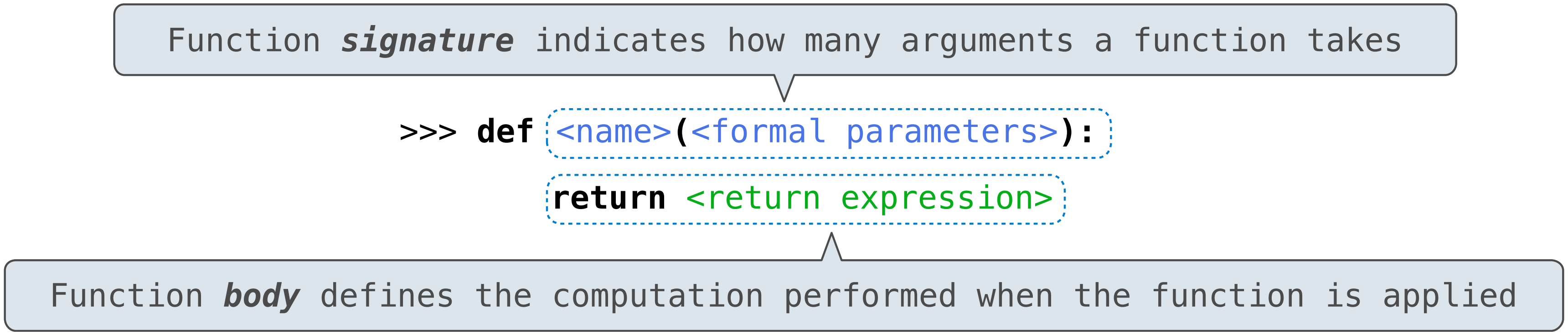
```
>>> def <name>(<formal parameters>):
        return <return expression>
```

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>
```

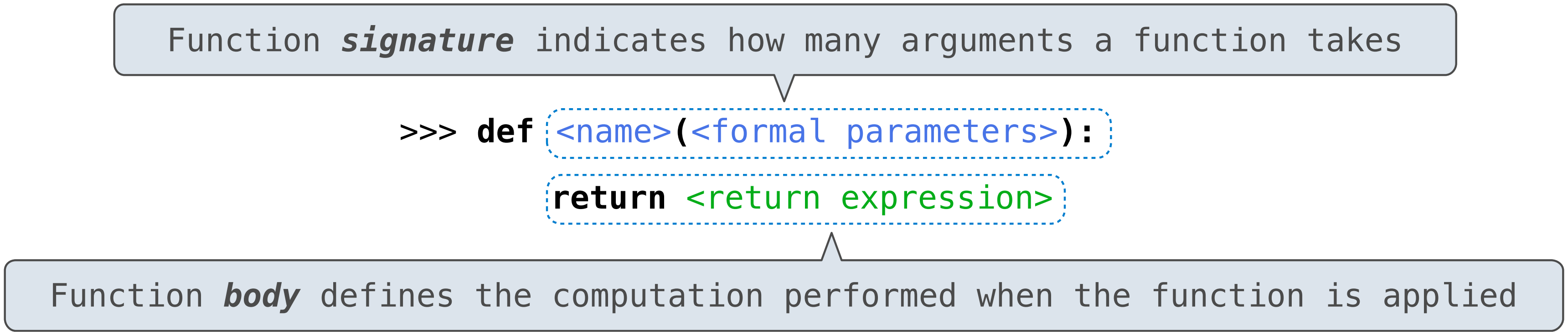
Function ***body*** defines the computation performed when the function is applied

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

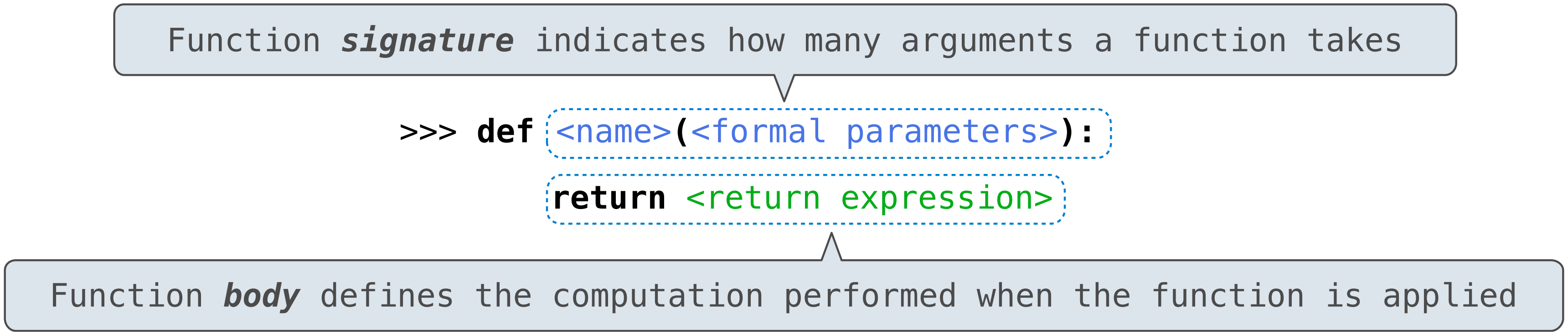
Execution procedure for def statements:

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

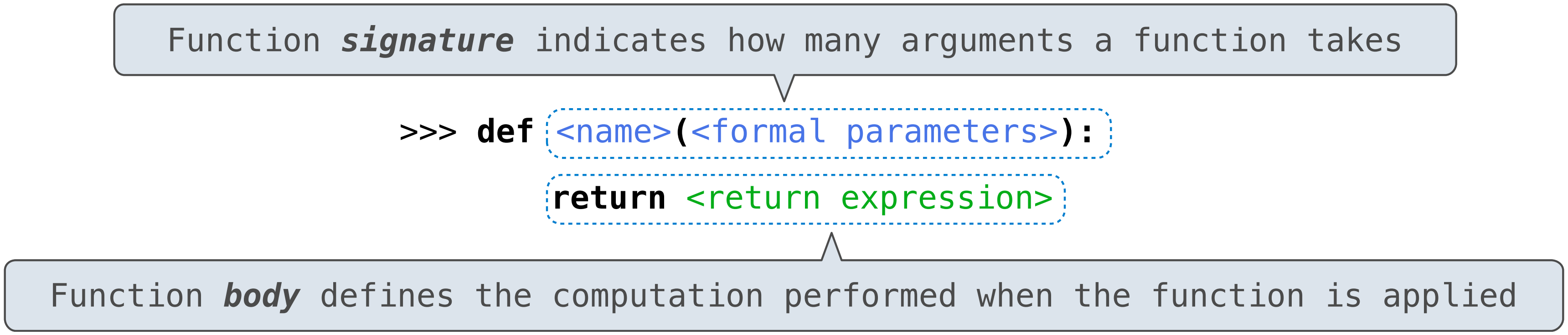
Execution procedure for def statements:

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

Interactive Diagram

Calling User-Defined Functions

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

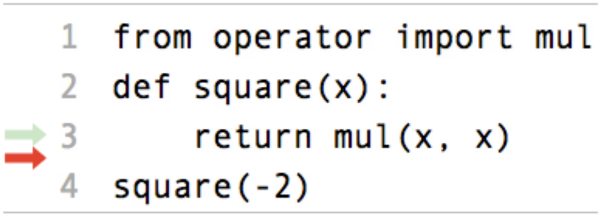
1. Add a local frame, forming a new environment
2. Bind the function's formal parameters to its arguments in that frame
3. Execute the body of the function in that new environment

Interactive Diagram

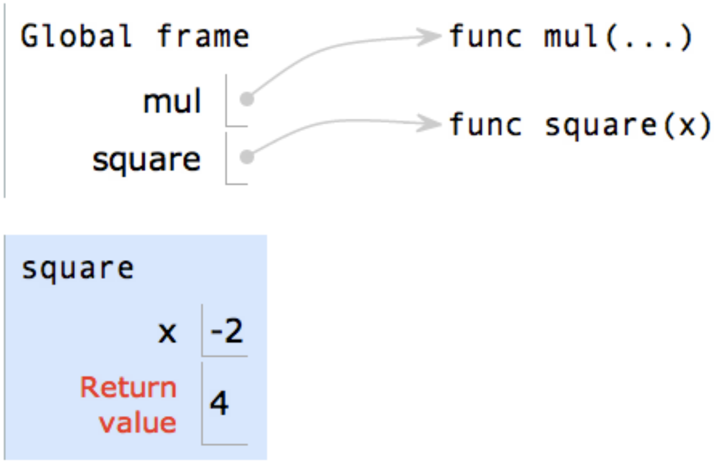
Calling User-Defined Functions

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



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

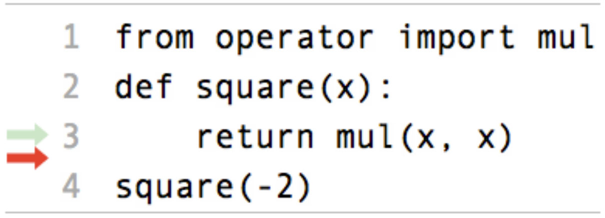


Interactive Diagram

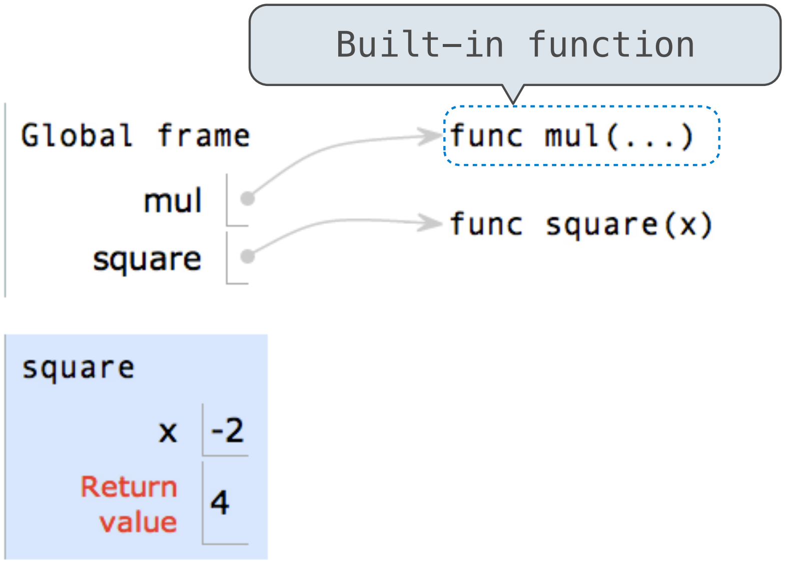
Calling User-Defined Functions

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



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

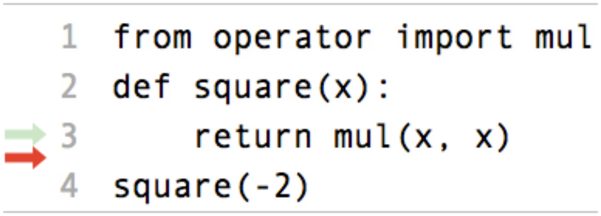


Interactive Diagram

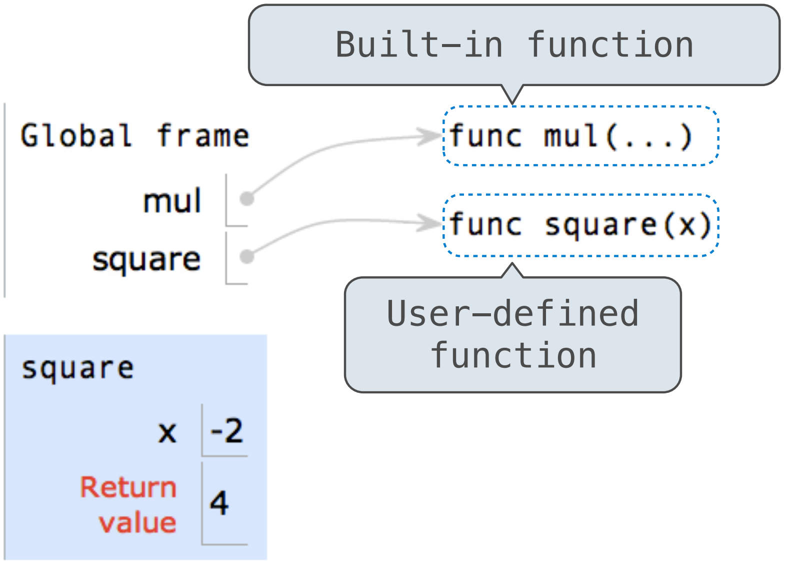
Calling User-Defined Functions

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



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



Built-in function

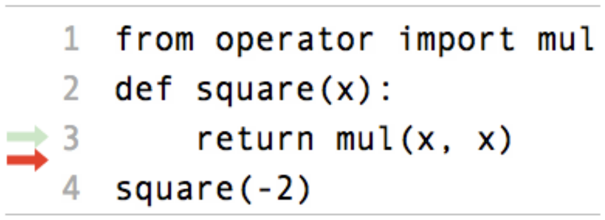
User-defined function

Interactive Diagram

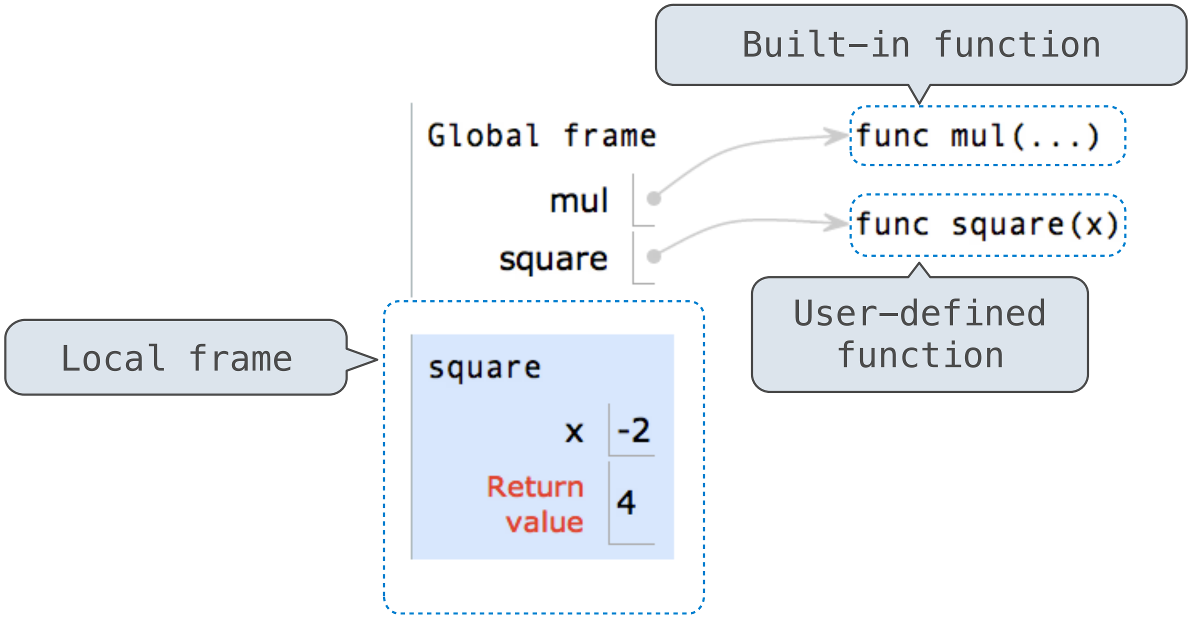
Calling User-Defined Functions

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



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

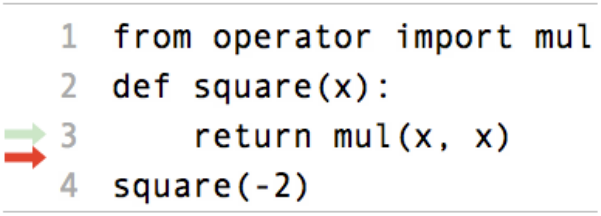


Interactive Diagram

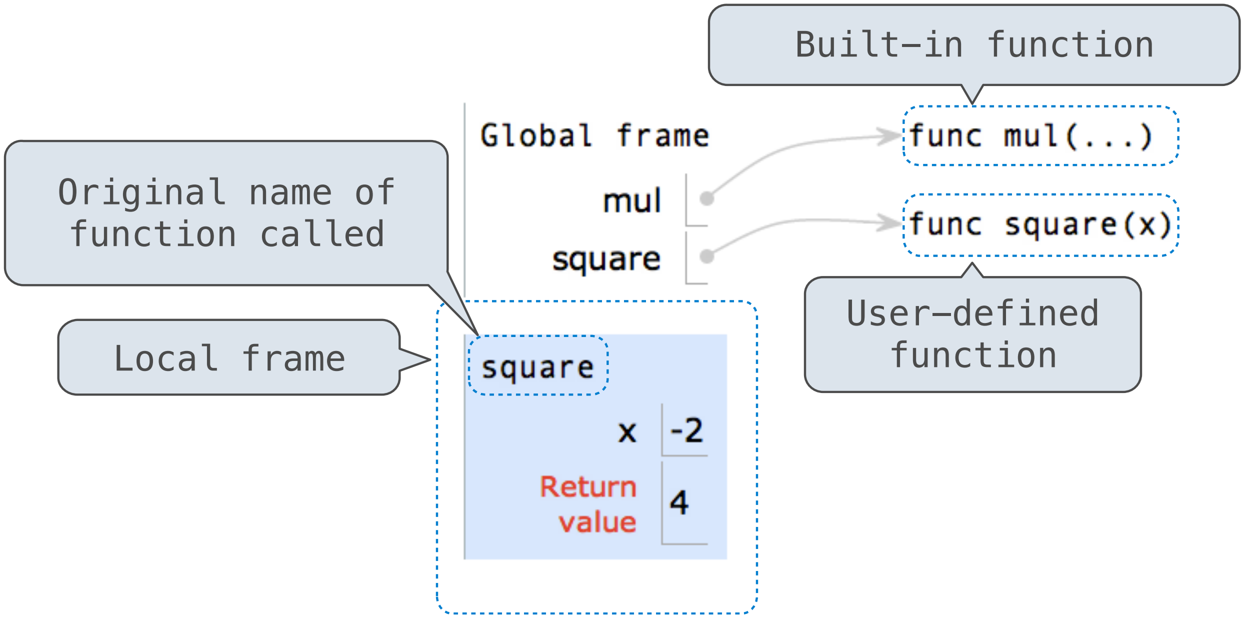
Calling User-Defined Functions

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



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

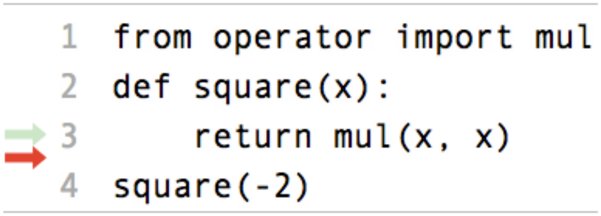


Interactive Diagram

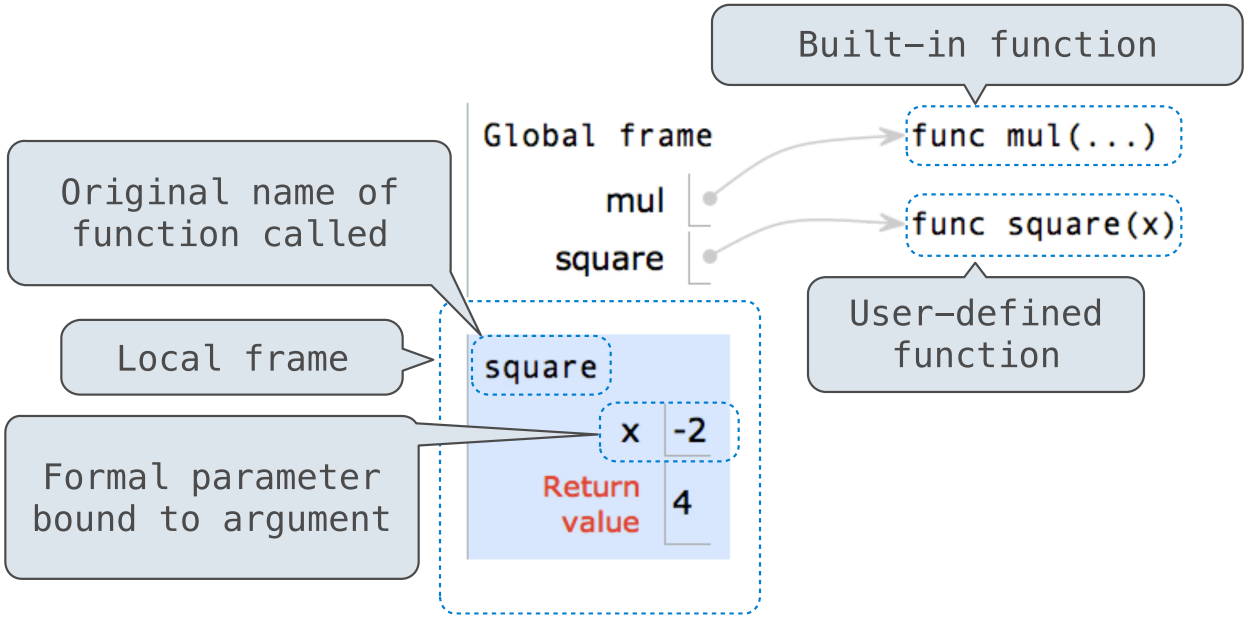
Calling User-Defined Functions

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



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



Built-in function

Original name of function called

Local frame

Formal parameter bound to argument

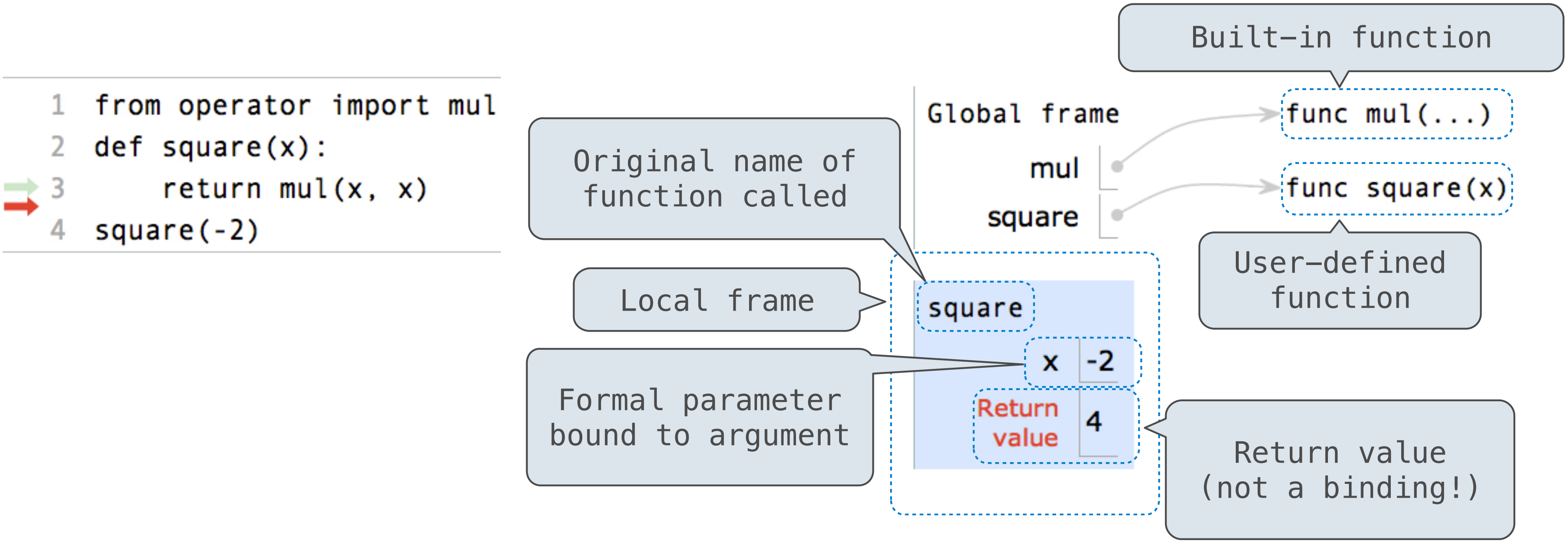
User-defined function

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



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

Built-in function

Original name of function called

User-defined function

Local frame

Formal parameter bound to argument

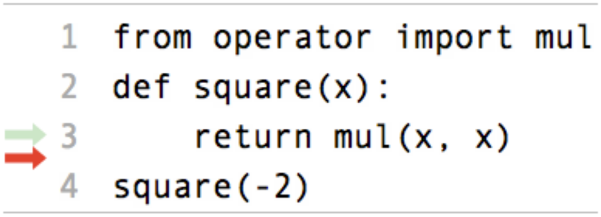
Return value (not a binding!)

Interactive Diagram

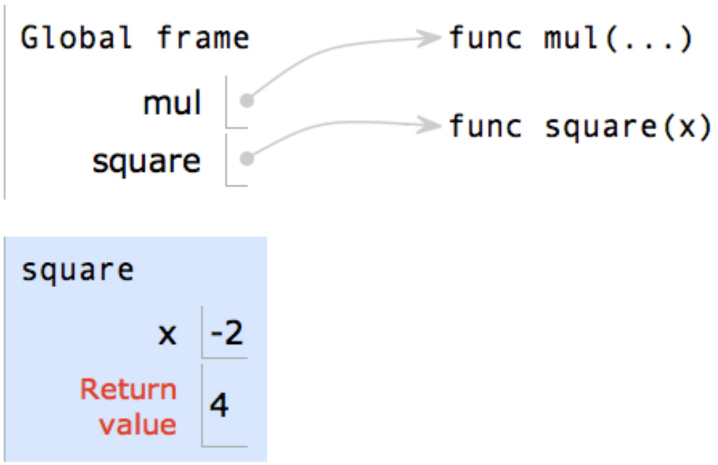
Calling User-Defined Functions

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



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

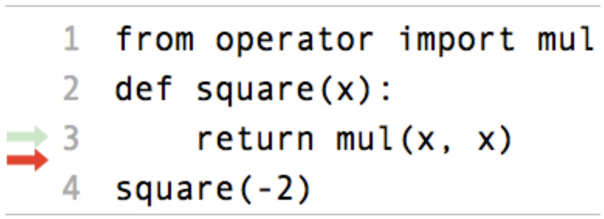


Interactive Diagram

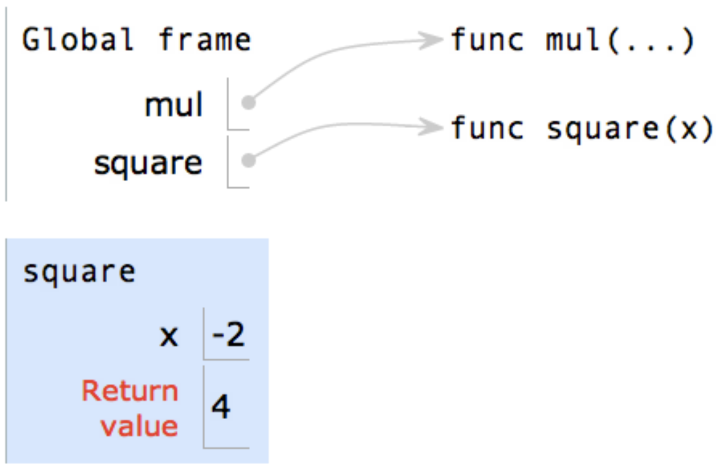
Calling User-Defined Functions

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



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



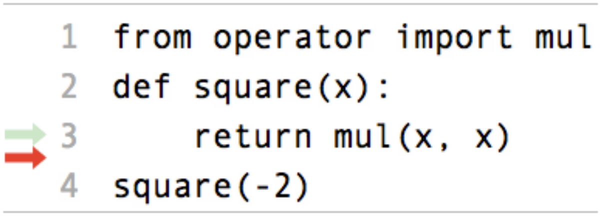
A function's signature has all the information needed to create a local frame

Interactive Diagram

Calling User-Defined Functions

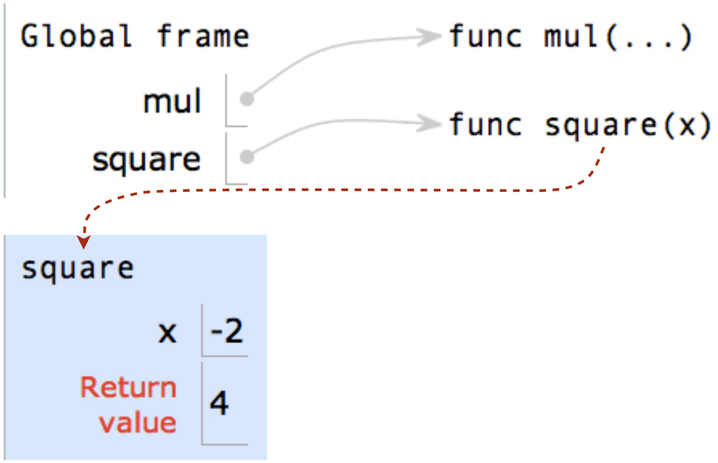
Procedure for calling/applying user-defined functions (version 1):

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



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

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

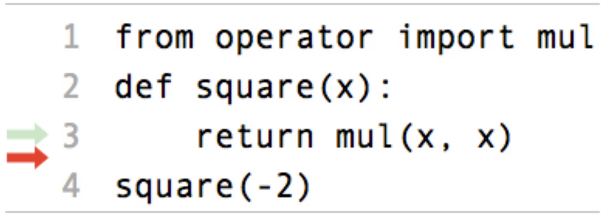


Interactive Diagram

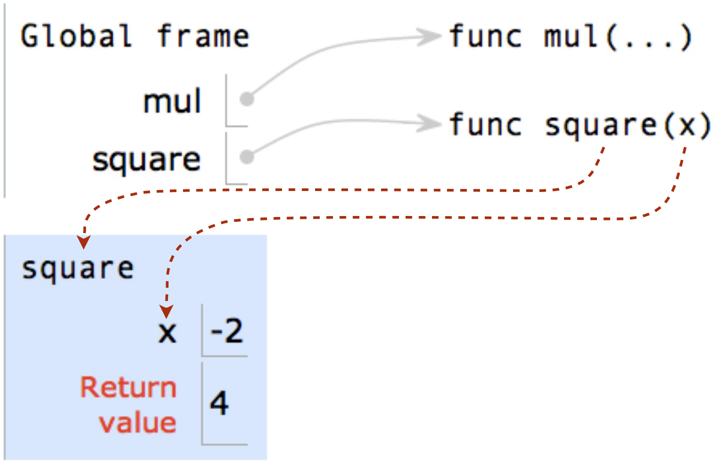
Calling User-Defined Functions

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



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



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

Interactive Diagram

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
...
```

No **return**

None Indicates that Nothing is Returned

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

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

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

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

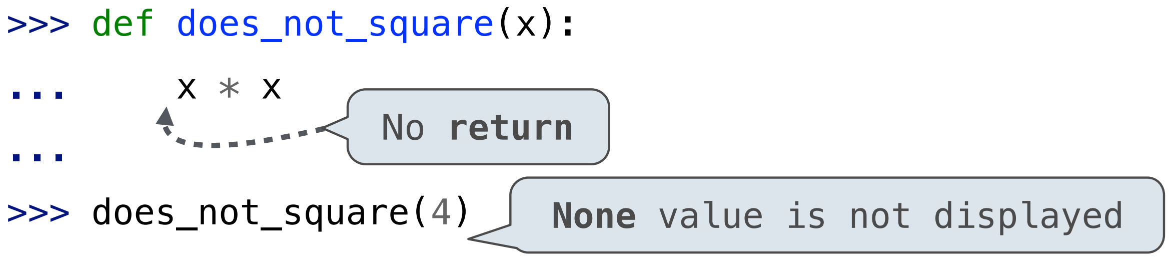
No **return**

None Indicates that Nothing is Returned

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

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

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



```
>>> def does_not_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)
```

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)
```

No **return**

None value is not displayed

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

None Indicates that Nothing is Returned

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

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

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

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

No **return**

None value is not displayed

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

Pure Functions & Non-Pure Functions

Pure Functions
just return values

Non-Pure Functions
have side effects

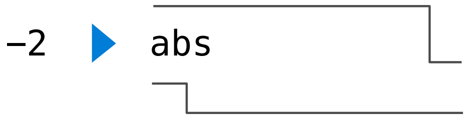
Pure Functions & Non-Pure Functions

Pure Functions
just return values

Non-Pure Functions
have side effects

Pure Functions & Non-Pure Functions

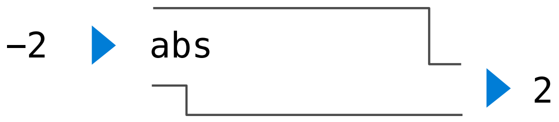
Pure Functions
just return values



Non-Pure Functions
have side effects

Pure Functions & Non-Pure Functions

Pure Functions
just return values



-2

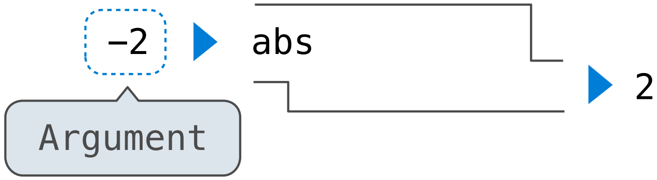
abs

2

Non-Pure Functions
have side effects

Pure Functions & Non-Pure Functions

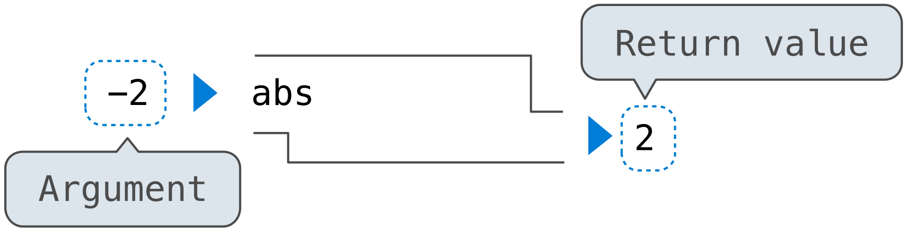
Pure Functions
just return values



Non-Pure Functions
have side effects

Pure Functions & Non-Pure Functions

Pure Functions
just return values



Return value

-2

abs

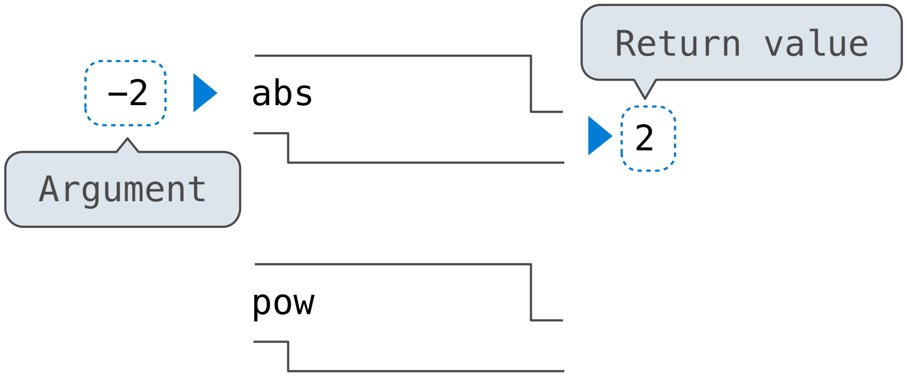
2

Argument

Non-Pure Functions
have side effects

Pure Functions & Non-Pure Functions

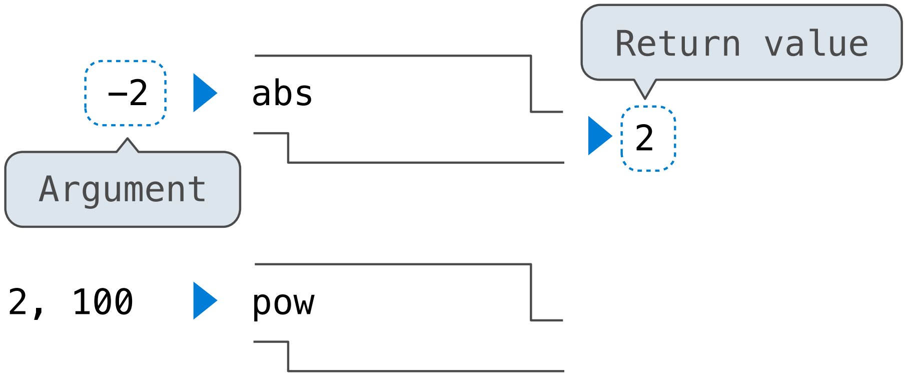
Pure Functions
just return values



Non-Pure Functions
have side effects

Pure Functions & Non-Pure Functions

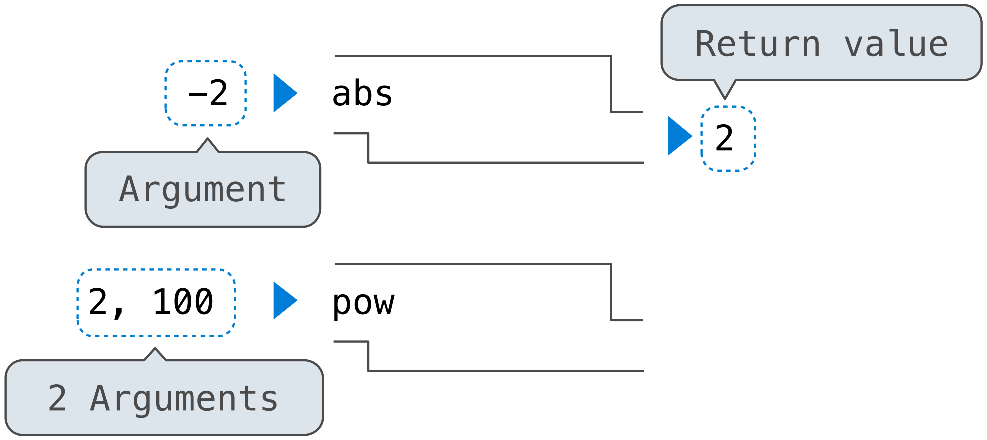
Pure Functions
just return values



Non-Pure Functions
have side effects

Pure Functions & Non-Pure Functions

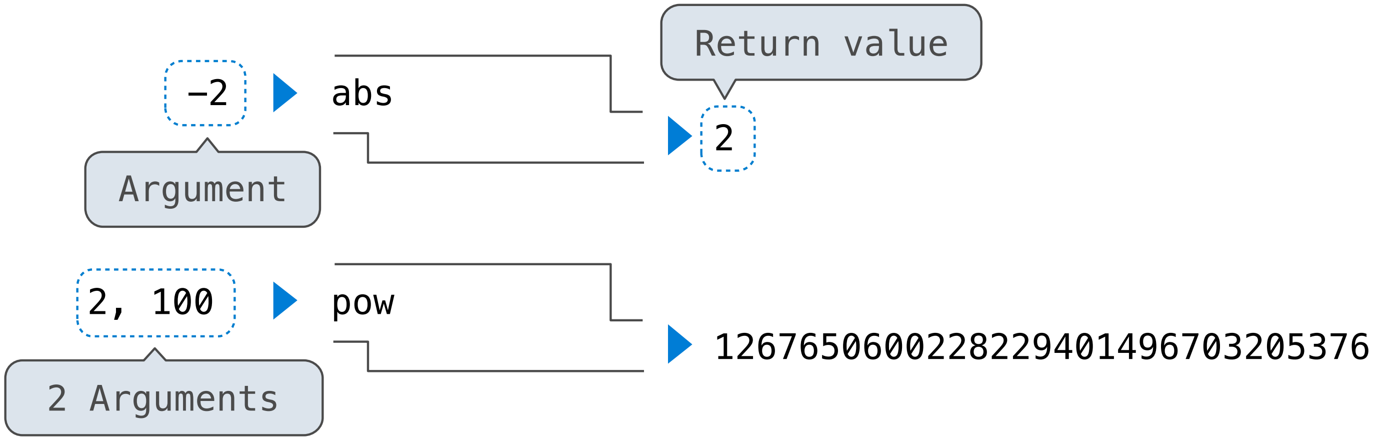
Pure Functions
just return values



Non-Pure Functions
have side effects

Pure Functions & Non-Pure Functions

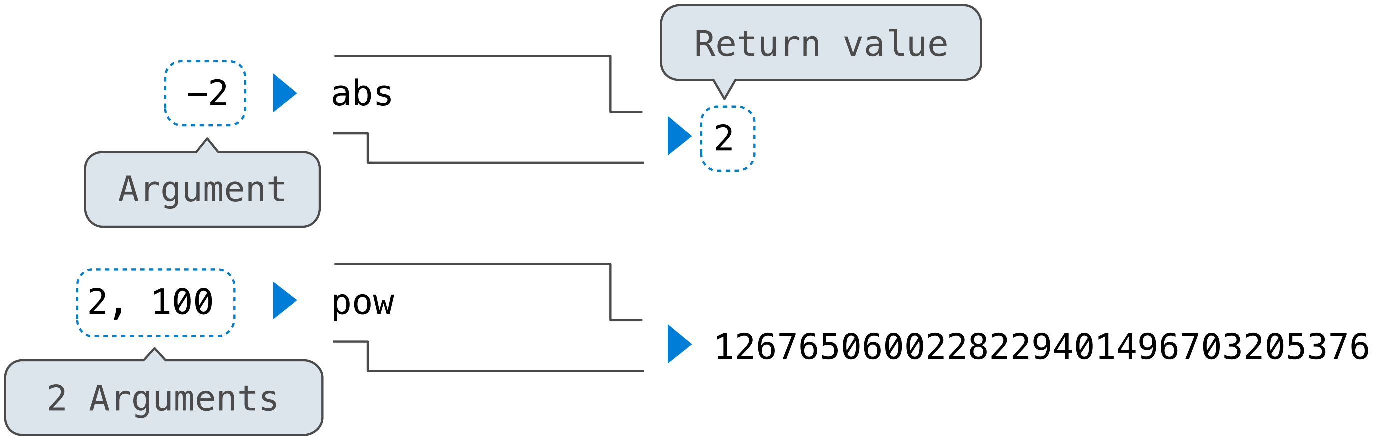
Pure Functions
just return values



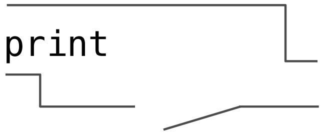
Non-Pure Functions
have side effects

Pure Functions & Non-Pure Functions

Pure Functions
just return values



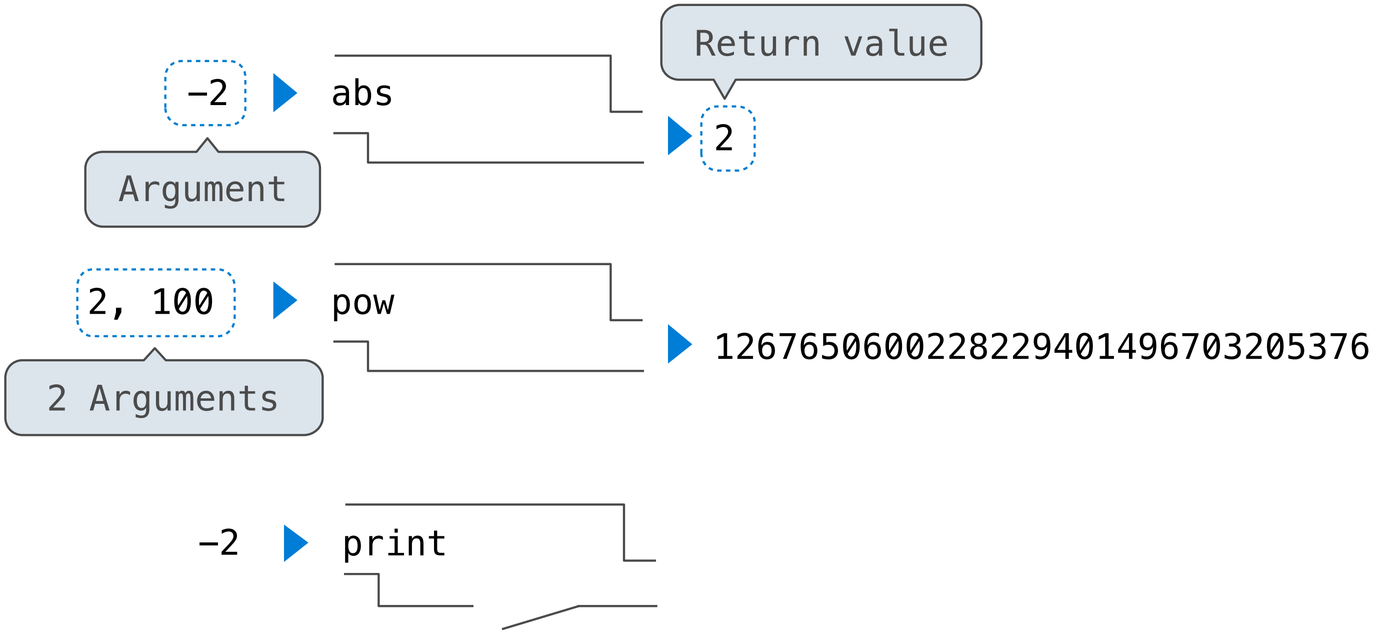
Non-Pure Functions
have side effects



Pure Functions & Non-Pure Functions

Pure Functions
just return values

Non-Pure Functions
have side effects



Return value

-2 abs 2

Argument

2, 100 pow 1267650600228229401496703205376

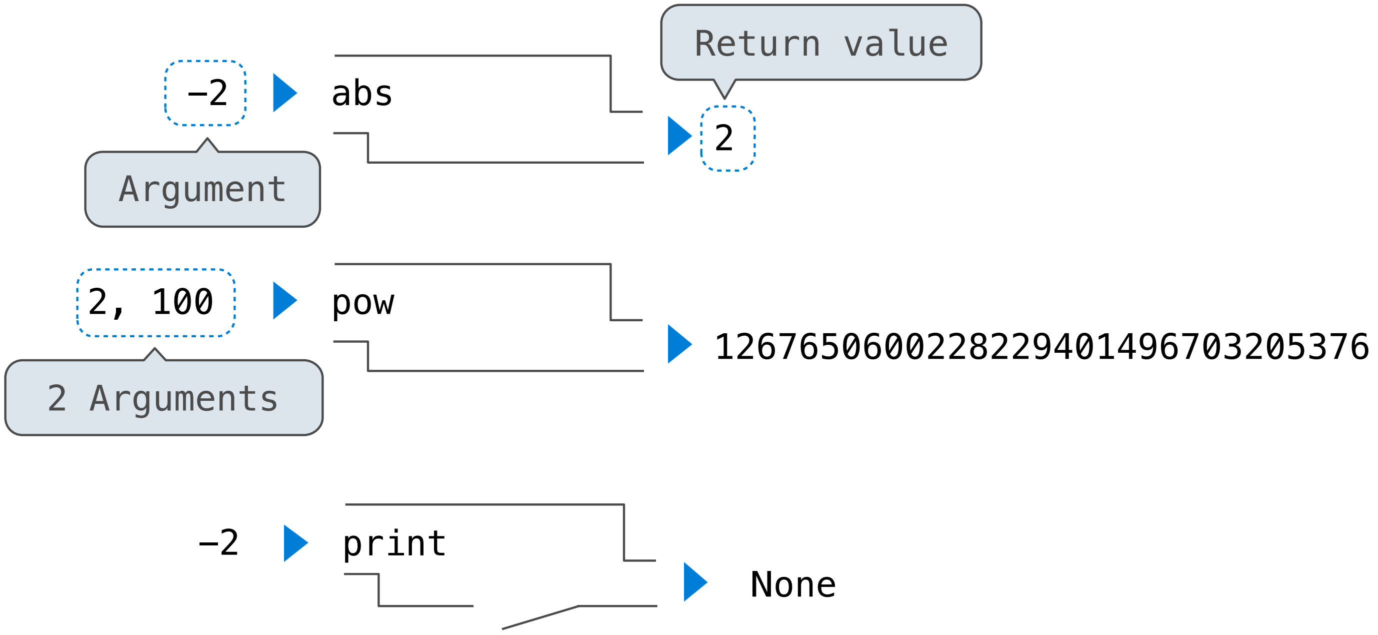
2 Arguments

-2 print

Pure Functions & Non-Pure Functions

Pure Functions
just return values

Non-Pure Functions
have side effects



-2 abs 2

Argument

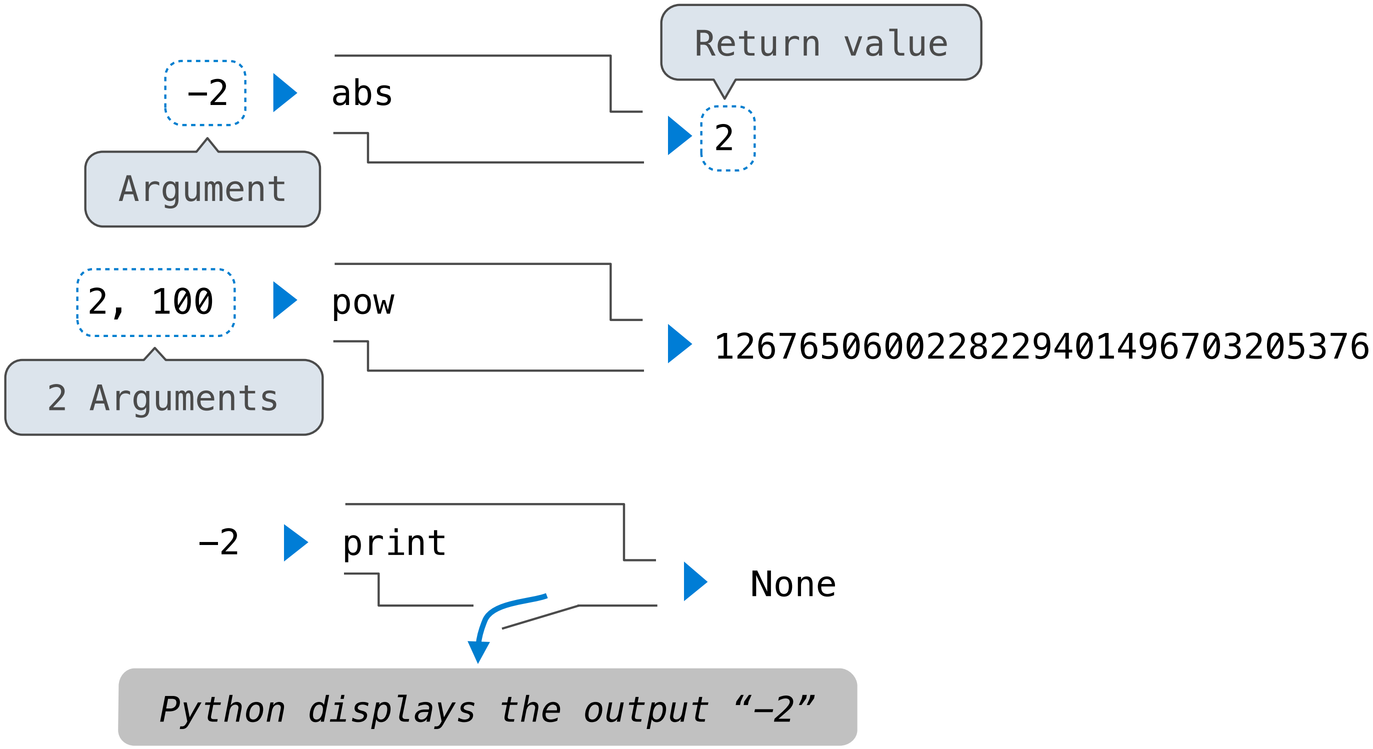
Return value

2, 100 pow 1267650600228229401496703205376

2 Arguments

-2 print None

Pure Functions & Non-Pure Functions



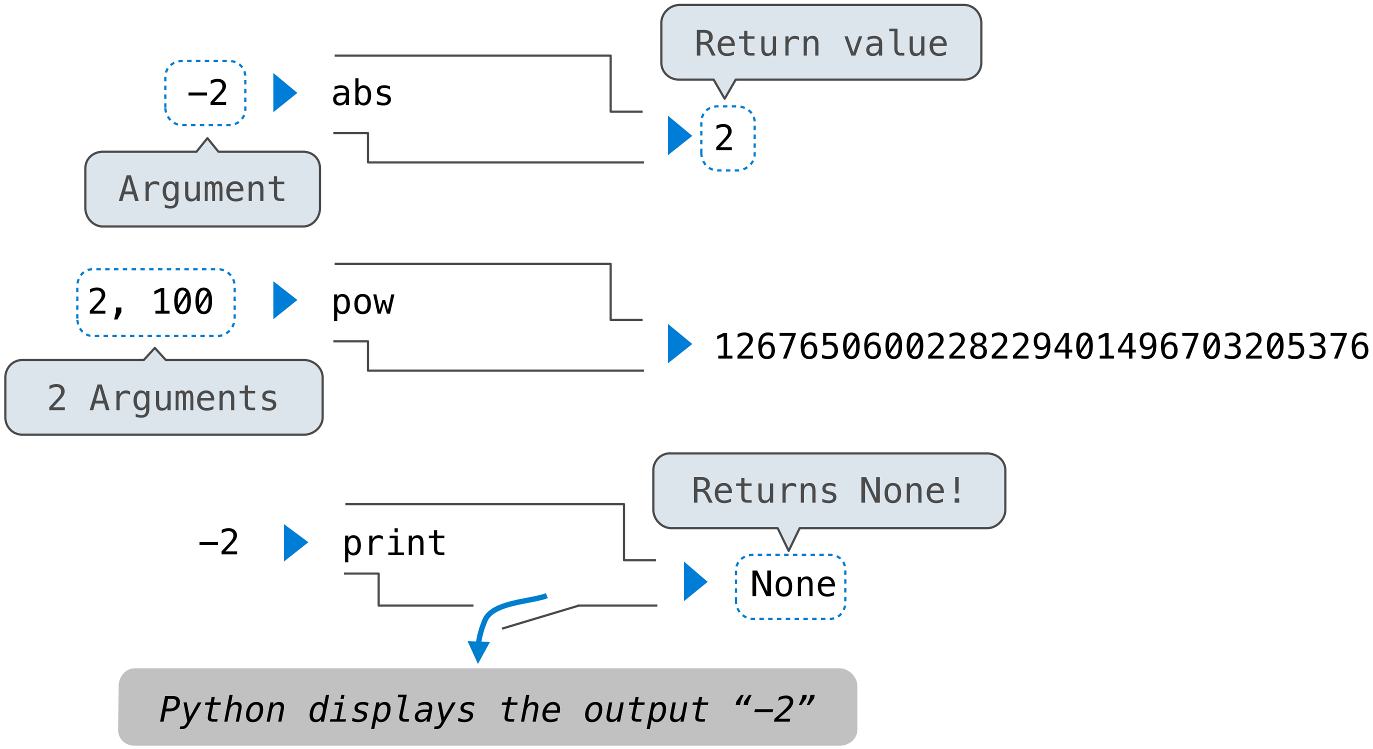
Pure Functions
just return values

Non-Pure Functions
have side effects

Pure Functions & Non-Pure Functions

Pure Functions
just return values

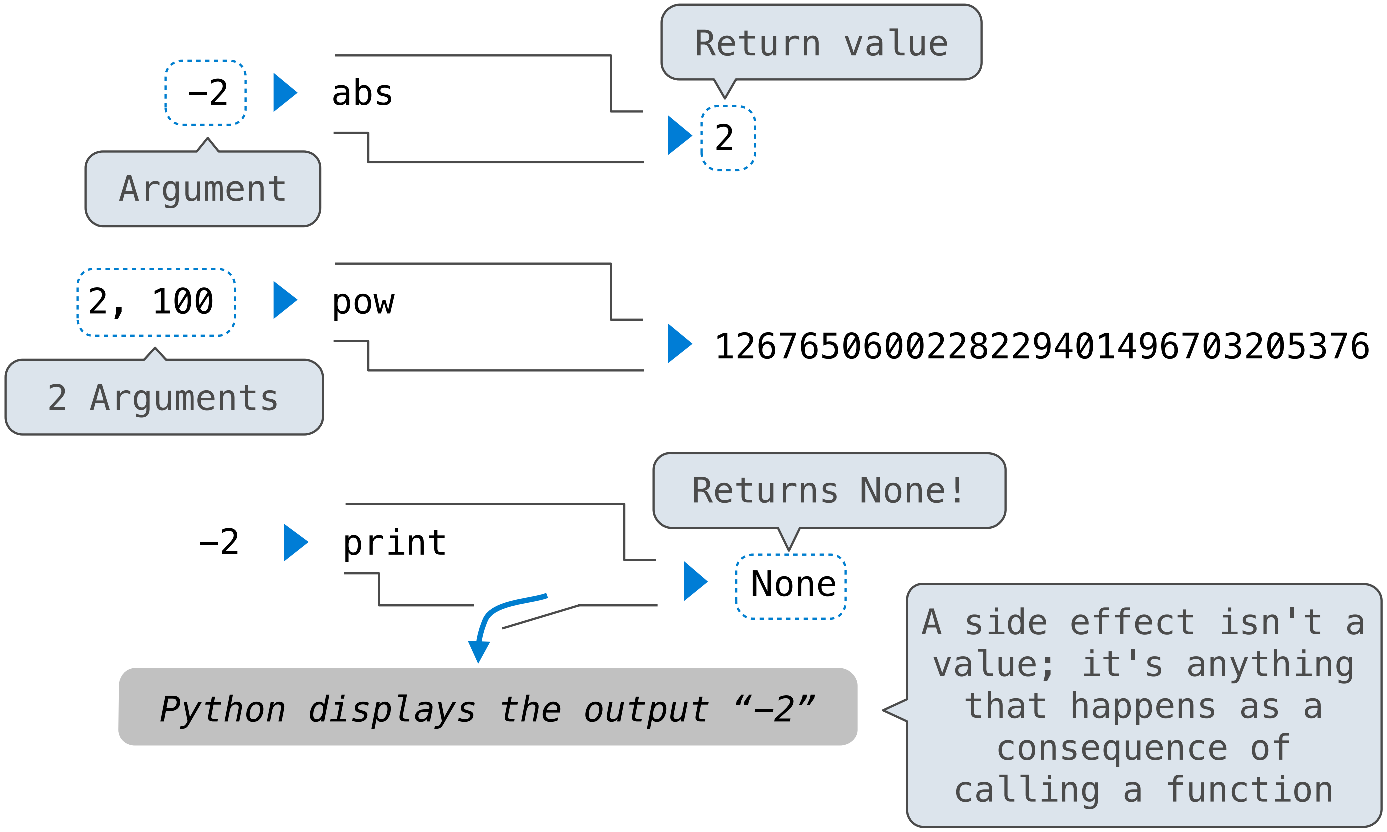
Non-Pure Functions
have side effects



Pure Functions & Non-Pure Functions

Pure Functions
just return values

Non-Pure Functions
have side effects



-2 abs 2

Return value

Argument

2, 100 pow 1267650600228229401496703205376

2 Arguments

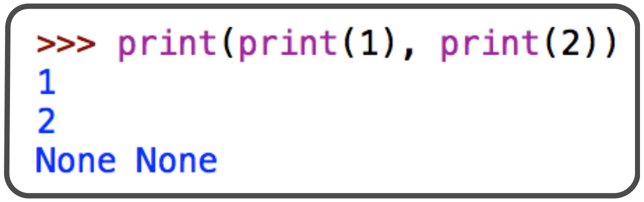
-2 print None

Returns None!

Python displays the output “-2”

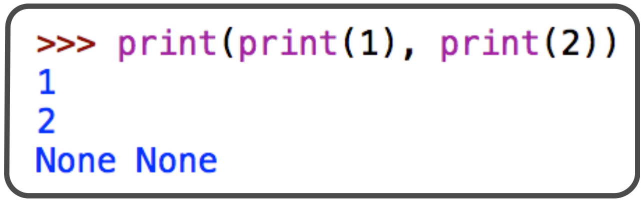
A side effect isn't a value; it's anything that happens as a consequence of calling a function

Nested Expressions with Print



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

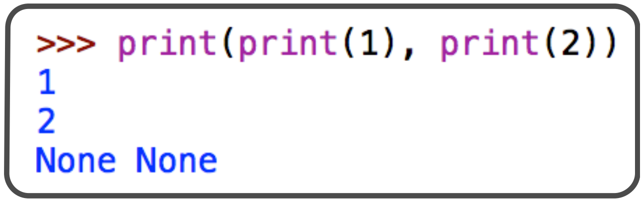
Nested Expressions with Print



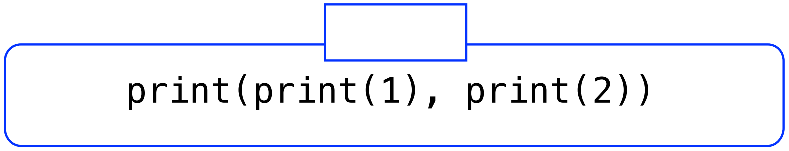
```
>>> print(print(1), print(2))
1
2
None None
```

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

Nested Expressions with Print

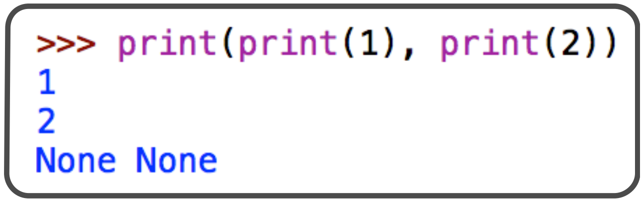


```
>>> print(print(1), print(2))
1
2
None None
```

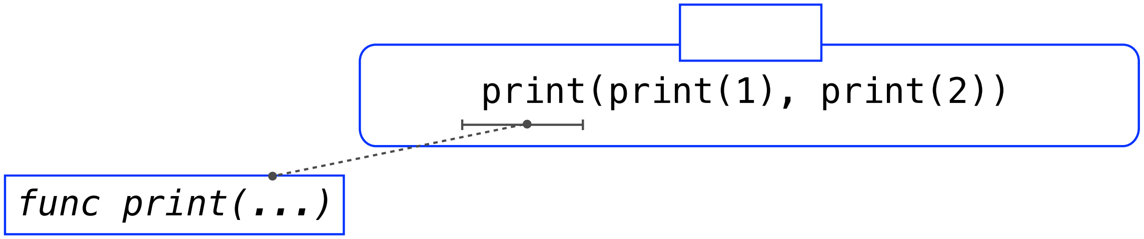


print(print(1), print(2))

Nested Expressions with Print

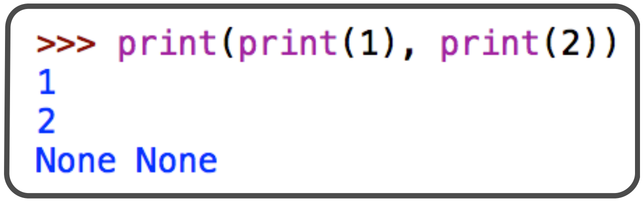


```
>>> print(print(1), print(2))
1
2
None None
```

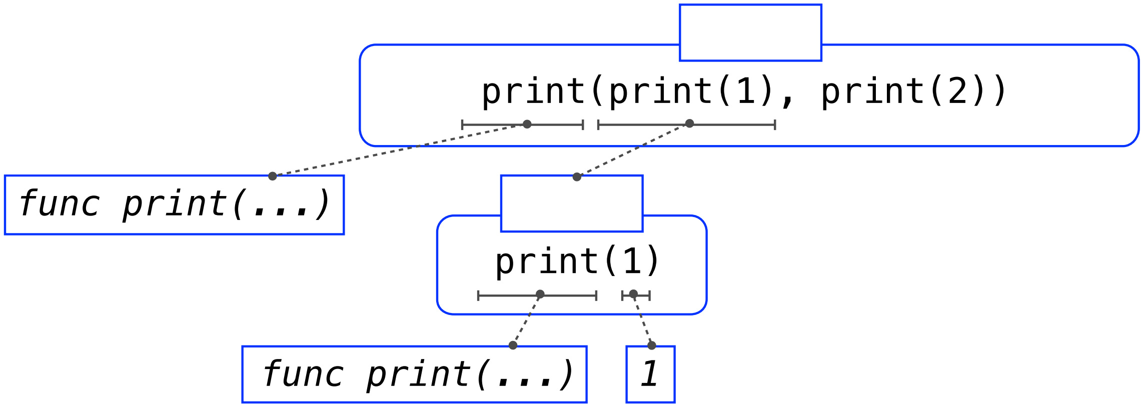


```
print(print(1), print(2))
```

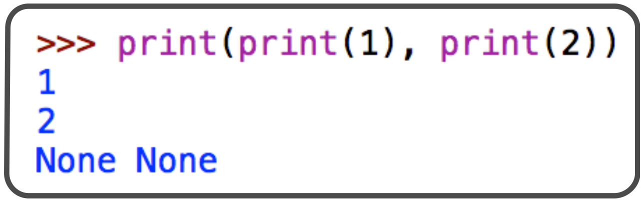
Nested Expressions with Print



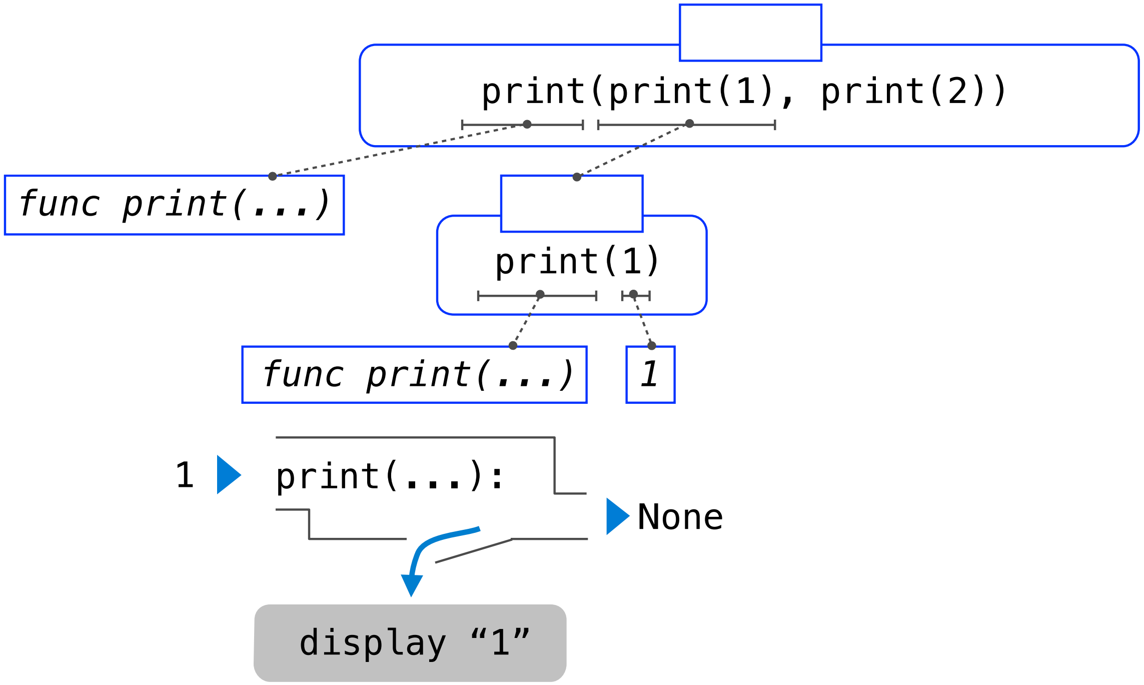
```
>>> print(print(1), print(2))
1
2
None None
```



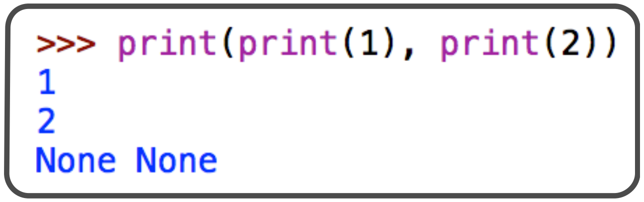
Nested Expressions with Print



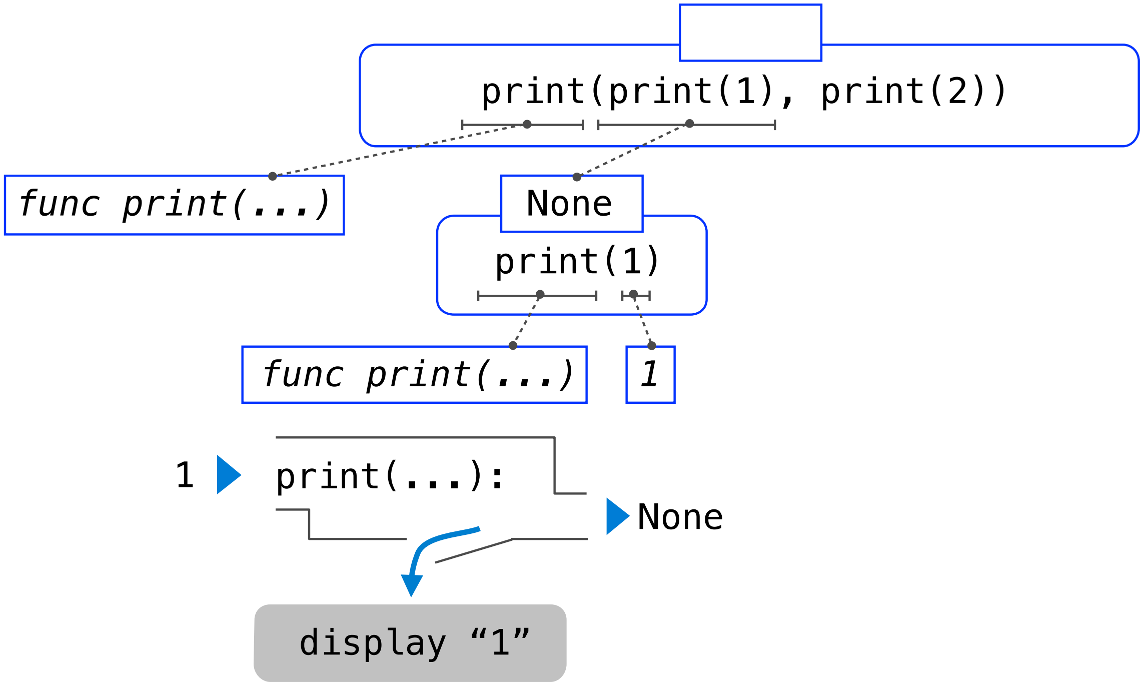
```
>>> print(print(1), print(2))
1
2
None None
```



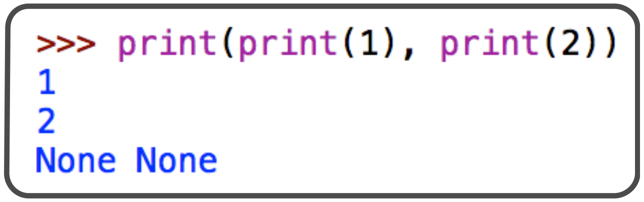
Nested Expressions with Print



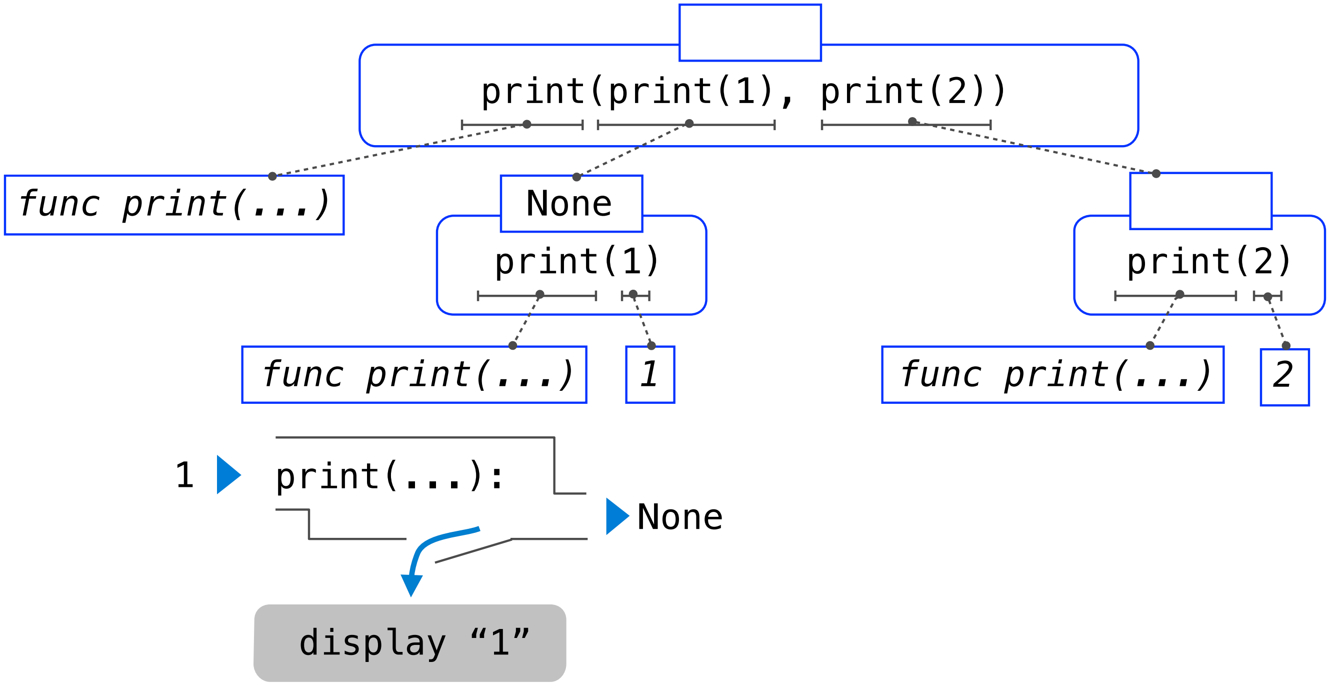
```
>>> print(print(1), print(2))
1
2
None None
```



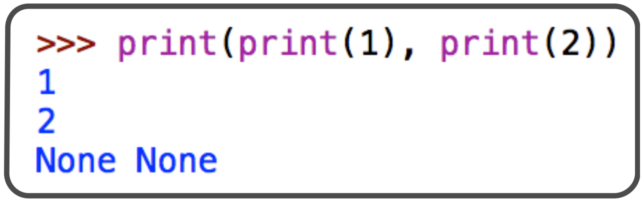
Nested Expressions with Print



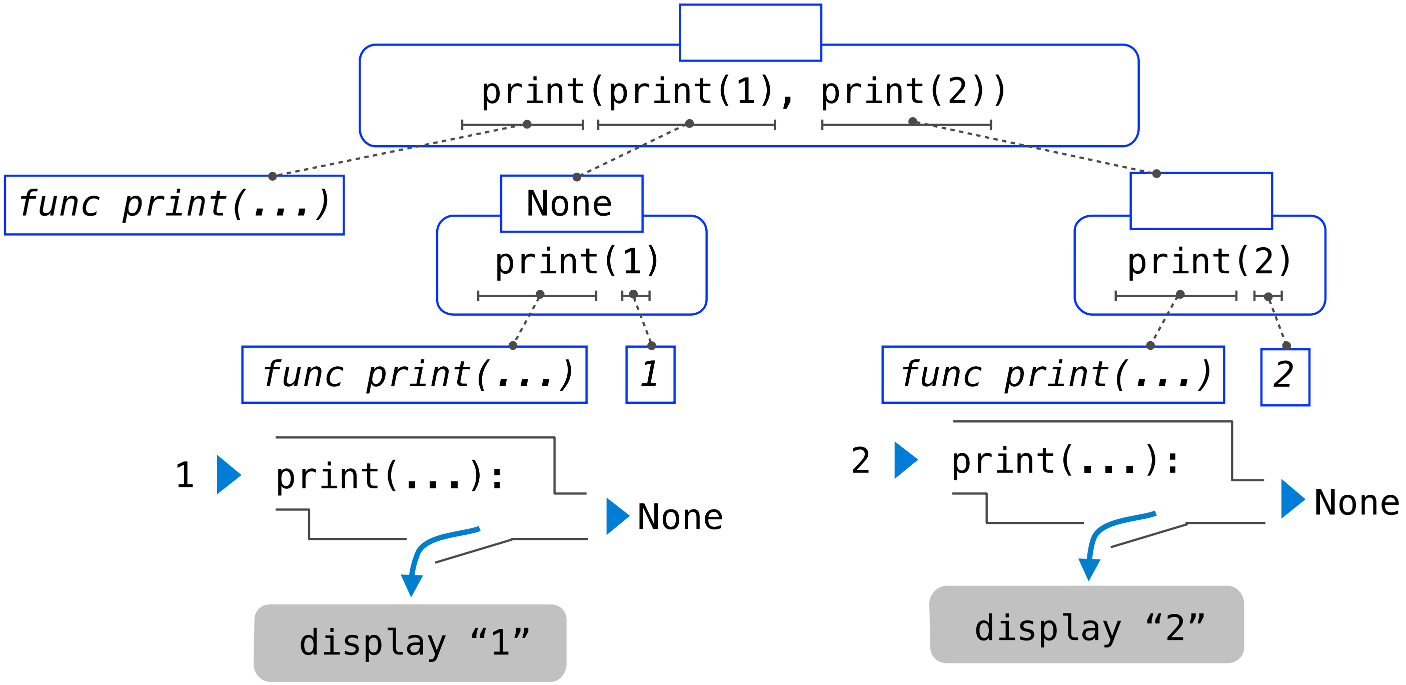
```
>>> print(print(1), print(2))
1
2
None None
```



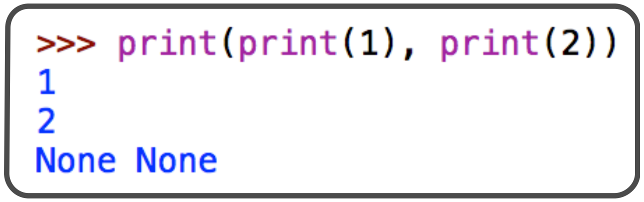
Nested Expressions with Print



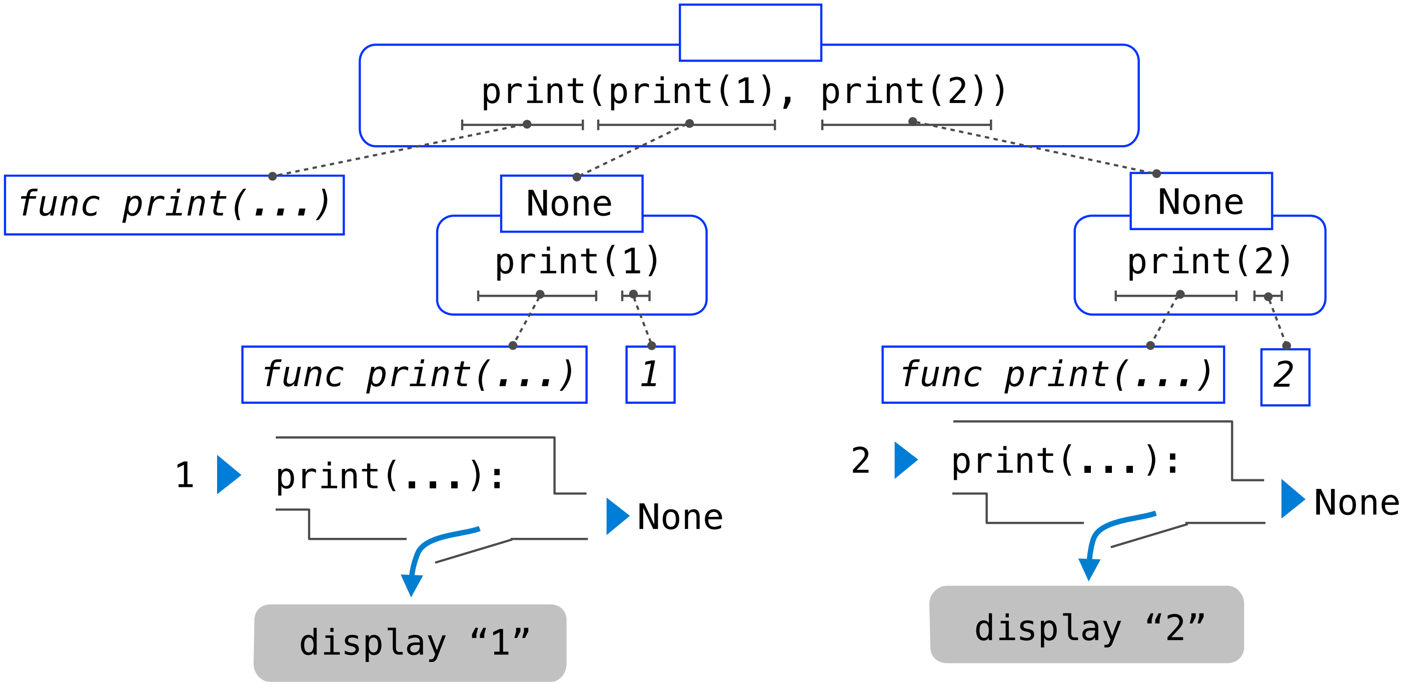
```
>>> print(print(1), print(2))
1
2
None None
```



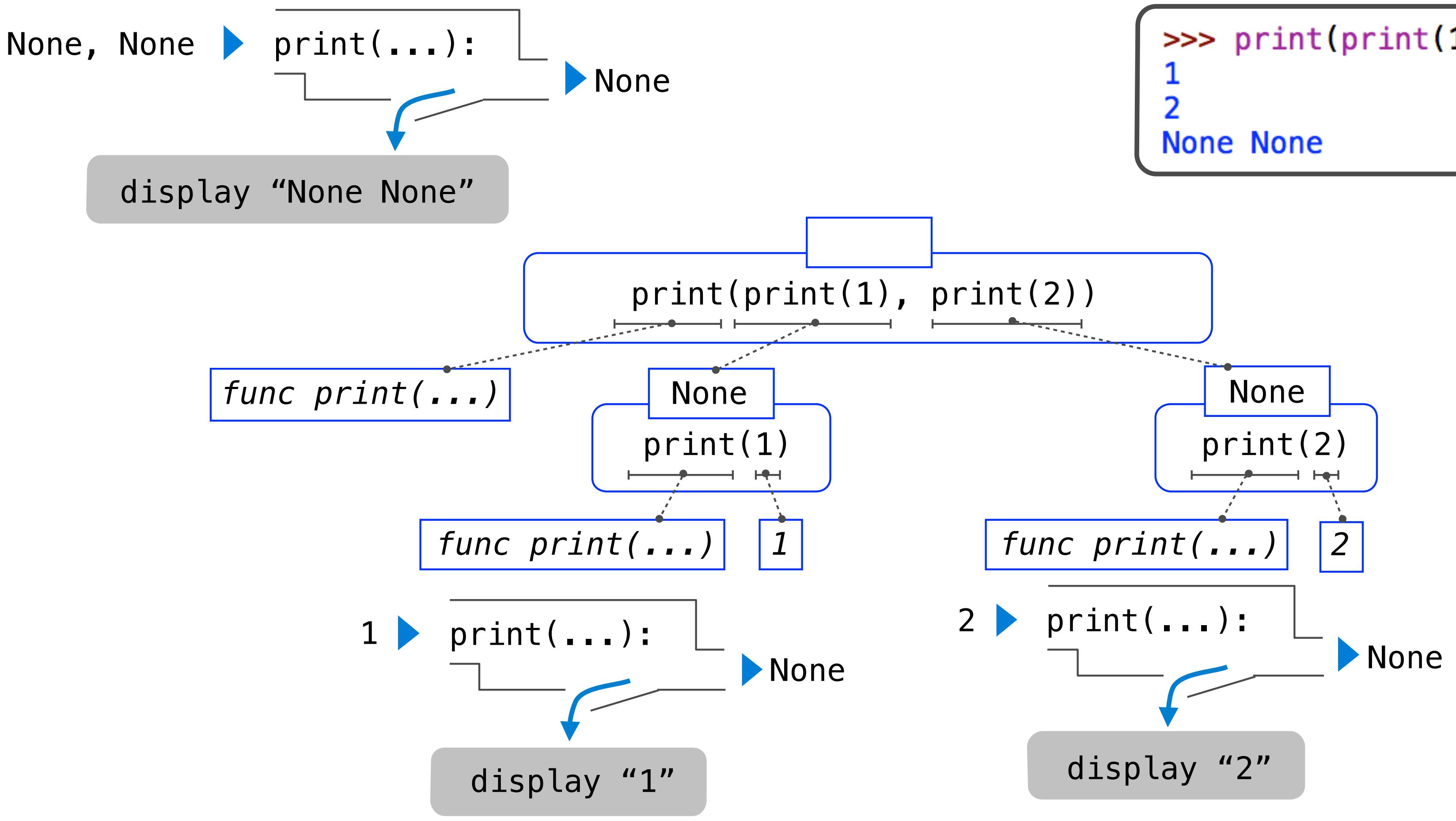
Nested Expressions with Print



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

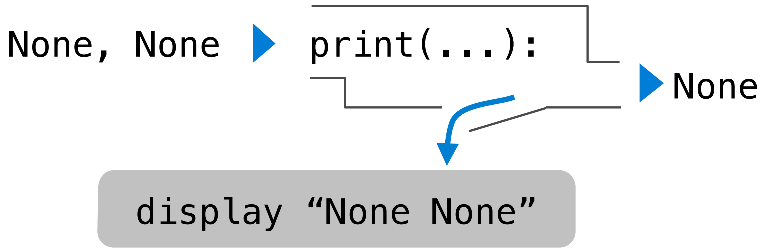


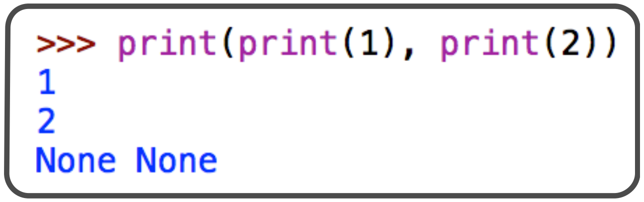
Nested Expressions with Print



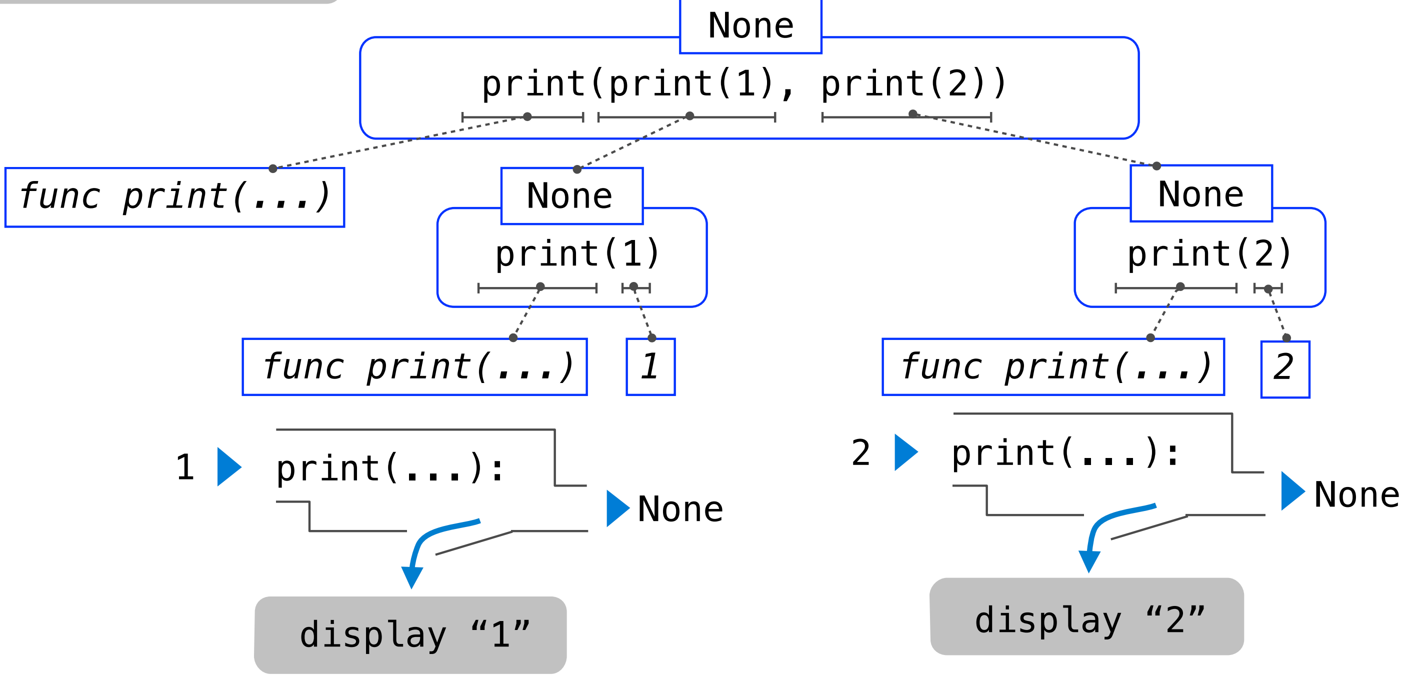
```
>>> print(print(1), print(2))
1
2
None None
```

Nested Expressions with Print



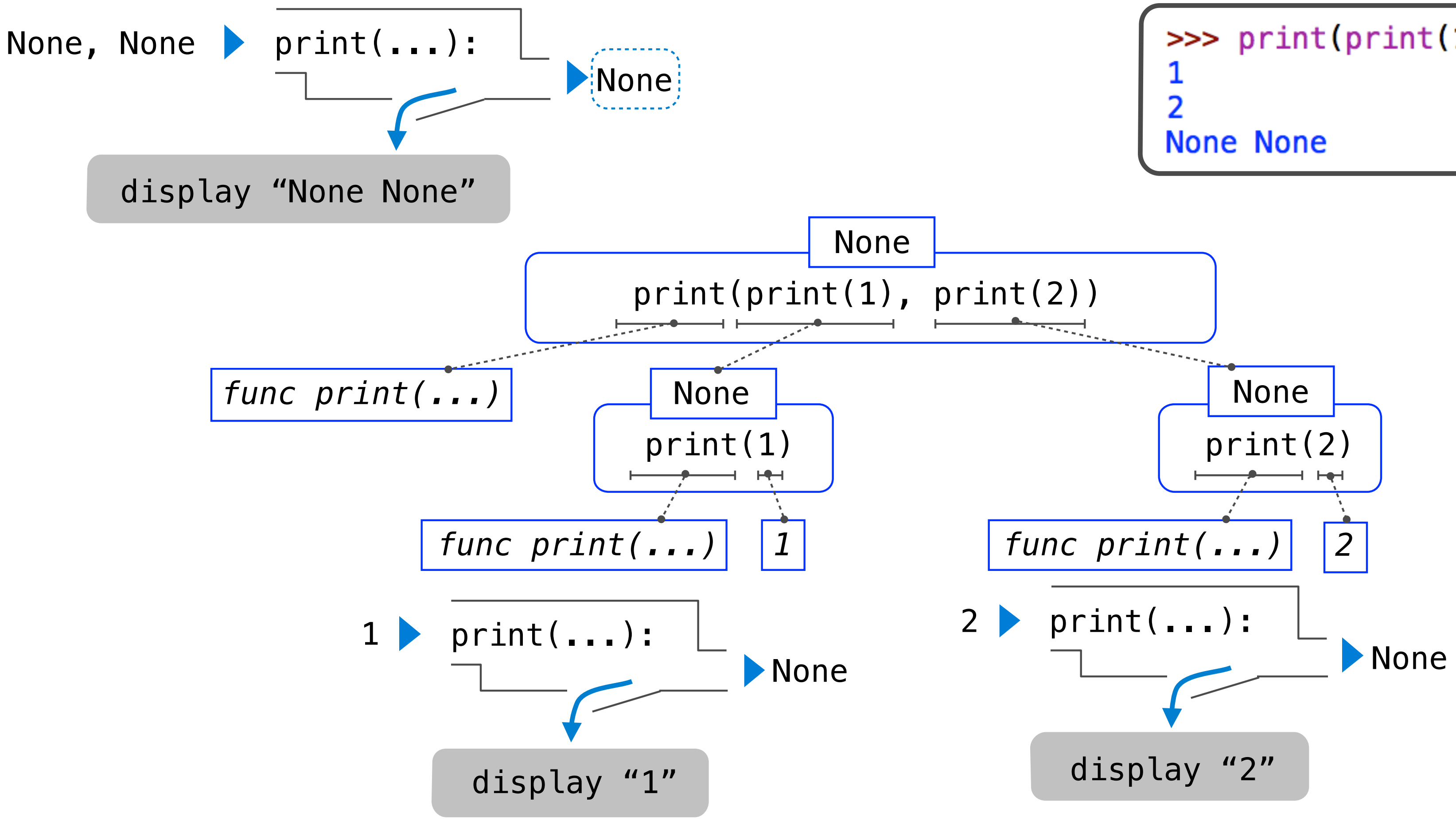


```
>>> print(print(1), print(2))
1
2
None None
```



Nested Expressions with Print

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



None, None ▶ print(...): ▶ None

display “None None”

None

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

func print(...)

None

print(1)

func print(...)

1

1 ▶ print(...): ▶ None

display “1”

None

print(2)

func print(...)

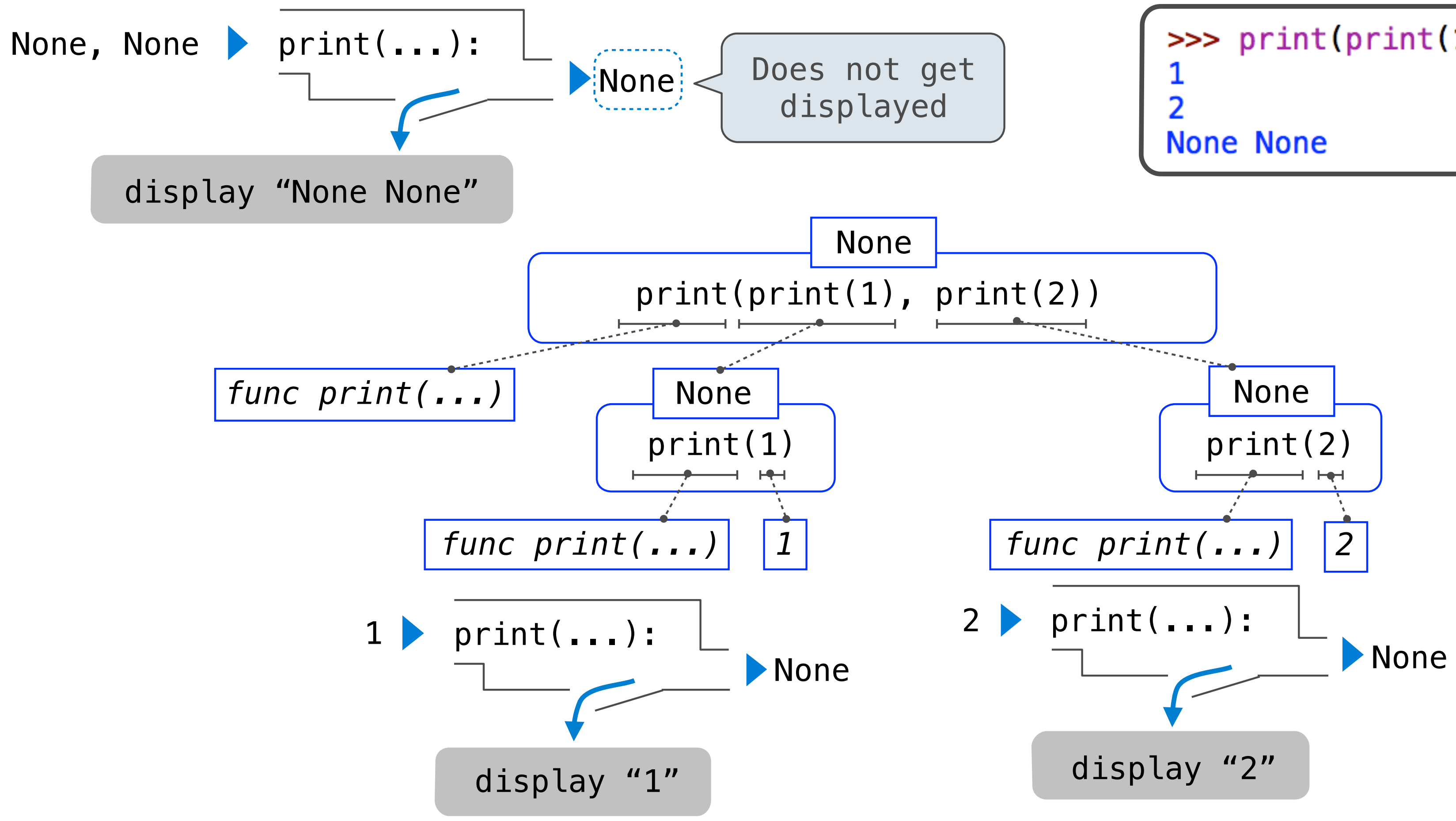
2

2 ▶ print(...): ▶ None

display “2”

Nested Expressions with Print

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


None, None
print(...):
None
Does not get displayed
display "None None"
None
print(print(1), print(2))
func print(...)
None
print(1)
None
print(2)
func print(...)
1
func print(...)
2
1
print(...):
None
display "1"
2
print(...):
None
display "2"