

# Functions

---

# Announcements

# Expressions

## Types of expressions

---

## Types of expressions

---

An expression describes a computation and evaluates to a value

## Types of expressions

---

An expression describes a computation and evaluates to a value

$$18 + 69$$

## Types of expressions

---

An expression describes a computation and evaluates to a value

$$18 + 69$$

$$\frac{6}{23}$$

## Types of expressions

---

An expression describes a computation and evaluates to a value

$$18 + 69$$

$$\frac{6}{23}$$

$$\sqrt{3493161}$$



## Types of expressions

---

An expression describes a computation and evaluates to a value

$$18 + 69$$

$$\frac{6}{23}$$

$$\sin \pi$$

$$\sqrt{3493161}$$

## Types of expressions

---

An expression describes a computation and evaluates to a value

$$18 + 69$$

$$\frac{6}{23}$$

$$\sin \pi$$

$$\sqrt{3493161}$$

$$|-1869|$$

## Types of expressions

---

An expression describes a computation and evaluates to a value

$$18 + 69$$

$$\frac{6}{23}$$

$$\sin \pi$$

$$\sqrt{3493161}$$

$$\sum_{i=1}^{100} i$$

$$|-1869|$$

## Types of expressions

---

An expression describes a computation and evaluates to a value

$$18 + 69$$

$$\frac{6}{23}$$

$$\sin \pi$$

$$\sqrt{3493161}$$

$$\sum_{i=1}^{100} i$$

$$|-1869|$$

$$\binom{69}{18}$$

## Types of expressions

---

An expression describes a computation and evaluates to a value

$$18 + 69$$

$$\frac{6}{23}$$

$$\sin \pi$$

$$f(x)$$

$$\sqrt{3493161}$$

$$\sum_{i=1}^{100} i$$

$$|-1869|$$

$$\binom{69}{18}$$

## Types of expressions

---

An expression describes a computation and evaluates to a value

$$18 + 69$$

$$\frac{6}{23}$$

$$\sin \pi$$

$$2^{100}$$

$$f(x)$$

$$\sqrt{3493161}$$

$$\sum_{i=1}^{100} i$$

$$|-1869|$$

$$\binom{69}{18}$$

## Types of expressions

---

An expression describes a computation and evaluates to a value

$$18 + 69$$

$$\frac{6}{23}$$

$$\sin \pi$$

$$\log_2 1024$$

$$2^{100}$$

$$f(x)$$

$$\sqrt{3493161}$$

$$\sum_{i=1}^{100} i$$

$$|-1869|$$

$$\binom{69}{18}$$

## Types of expressions

---

An expression describes a computation and evaluates to a value

$$18 + 69$$

$$\frac{6}{23}$$

$$\sin \pi$$

$$\log_2 1024$$

$$2^{100}$$

$$f(x)$$

$$\sqrt{3493161}$$

$$7 \bmod 2$$

$$\sum_{i=1}^{100} i$$

$$\binom{69}{18}$$

$$|-1869|$$



## Types of expressions

---

An expression describes a computation and evaluates to a value

$$18 + 69$$

$$\frac{6}{23}$$

$$\sin \pi$$

$$\log_2 1024$$

$$2^{100}$$

$$f(x)$$

$$\sqrt{3493161}$$

$$7 \bmod 2$$

$$\sum_{i=1}^{100} i$$

$$\lim_{x \rightarrow \infty} \frac{1}{x}$$

$$|-1869|$$

$$\binom{69}{18}$$

## Types of expressions

---

An expression describes a computation and evaluates to a value

$$\begin{array}{ccccccc} 18 + 69 & & \sin \pi & & \log_2 1024 & & \\ & & \frac{6}{23} & & & & \\ 2^{100} & & & & \sqrt{3493161} & & \\ & & f(x) & & & & \\ 7 \bmod 2 & & \sum_{i=1}^{100} i & & \binom{69}{18} & & \lim_{x \rightarrow \infty} \frac{1}{x} \\ & & | - 1869 | & & & & \end{array}$$

## Call Expressions in Python

---

All expressions can use function call notation  
(Demo)

## Anatomy of a Call Expression

---

## Anatomy of a Call Expression

---

add ( 2 , 3 )

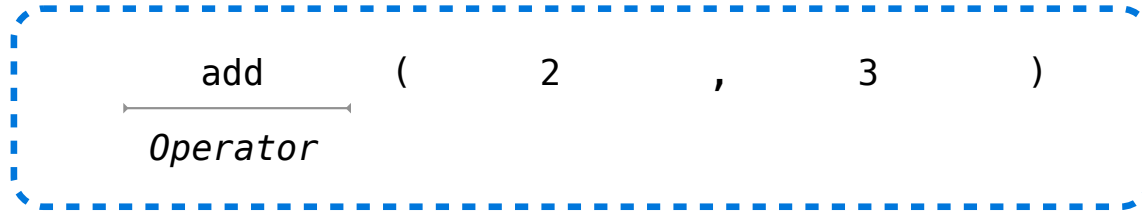
## Anatomy of a Call Expression

---

add ( 2 , 3 )

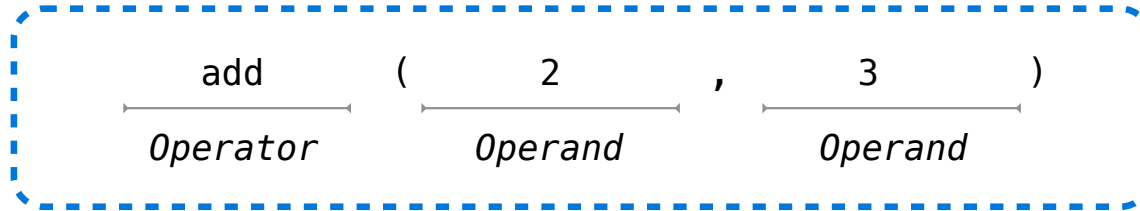
## Anatomy of a Call Expression

---



## Anatomy of a Call Expression

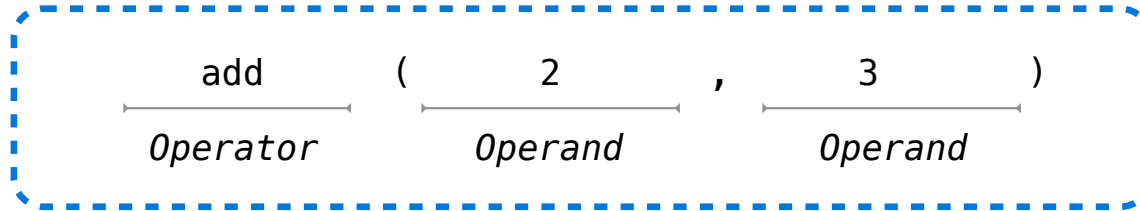
---





## Anatomy of a Call Expression

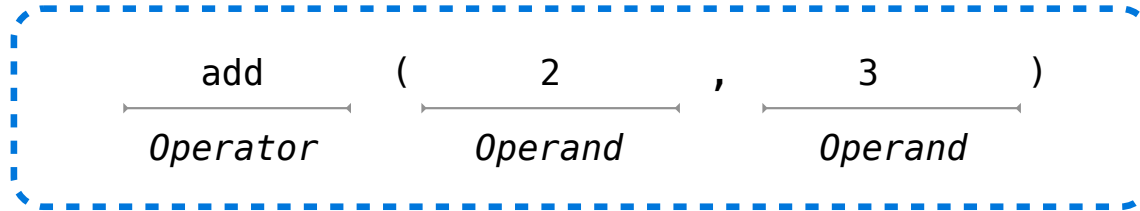
---



Operators and operands are also expressions

## Anatomy of a Call Expression

---

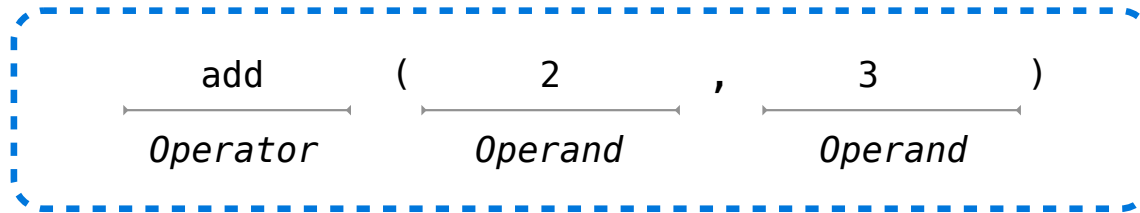


Operators and operands are also expressions

So they evaluate to values

## Anatomy of a Call Expression

---



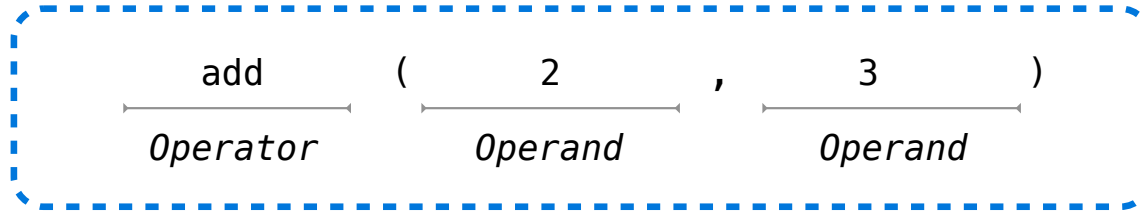
Operators and operands are also expressions

So they evaluate to values

**Evaluation procedure for call expressions:**

## Anatomy of a Call Expression

---



Operators and operands are also expressions

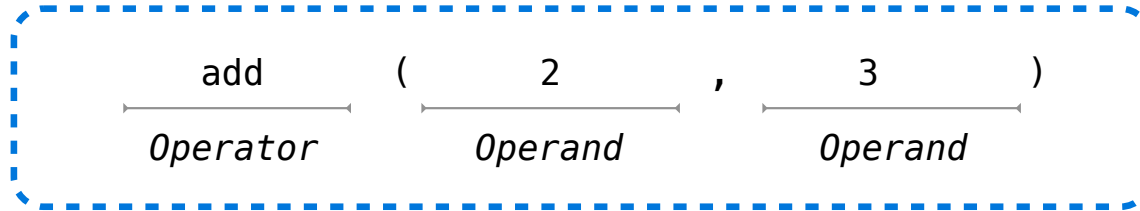
So they evaluate to values

### **Evaluation procedure for call expressions:**

1. Evaluate the operator and then the operand subexpressions

## Anatomy of a Call Expression

---



Operators and operands are also expressions

So they evaluate to values

### Evaluation procedure for call expressions:

1. Evaluate the operator and then the operand subexpressions
2. Apply the function that is the value of the operator to the arguments that are the values of the operands

## Evaluating Nested Expressions

---

```
mul(add(4, mul(4, 6)), add(3, 5))
```

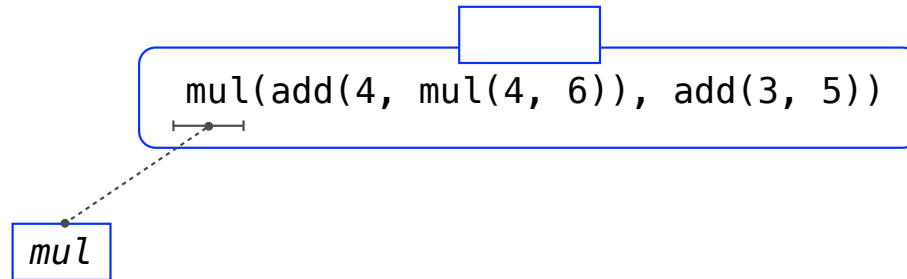
## Evaluating Nested Expressions

---

`mul(add(4, mul(4, 6)), add(3, 5))`

## Evaluating Nested Expressions

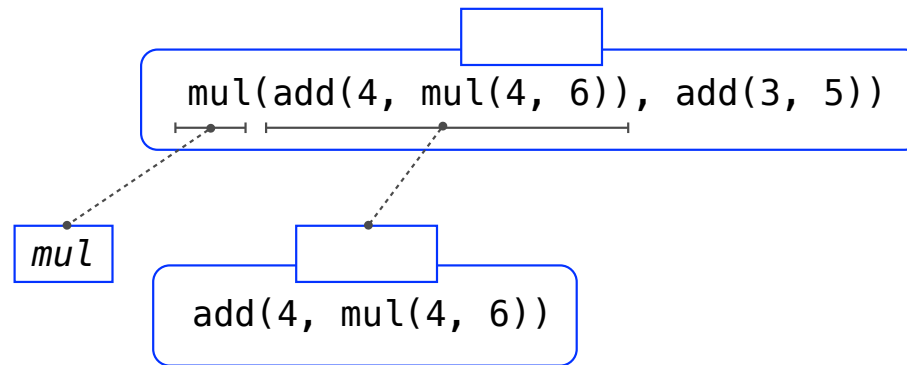
---





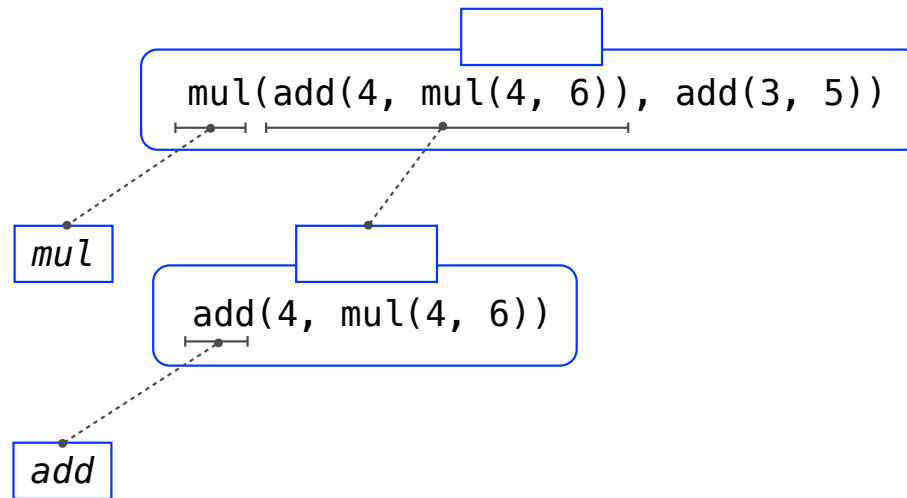
## Evaluating Nested Expressions

---



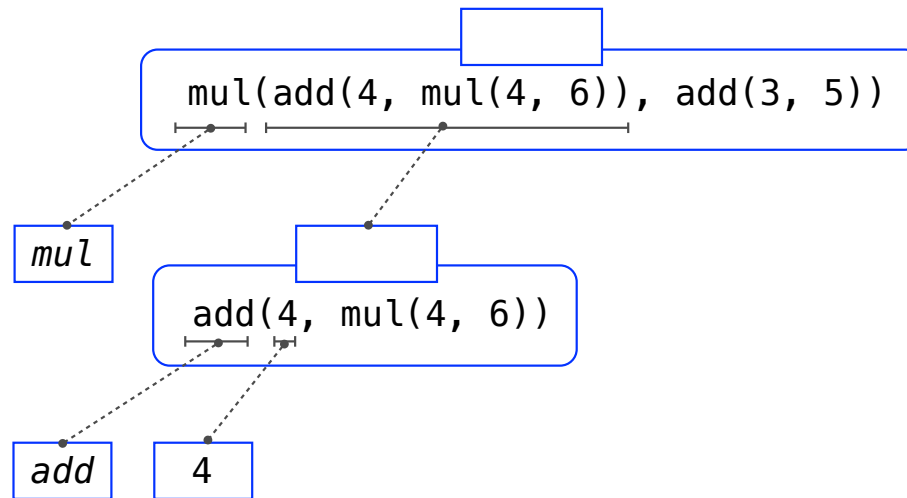
## Evaluating Nested Expressions

---



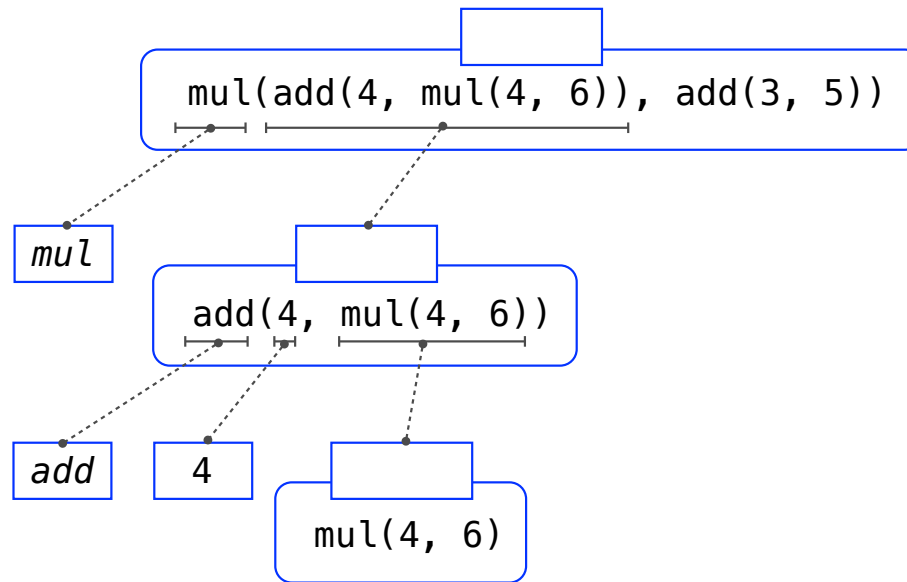
## Evaluating Nested Expressions

---

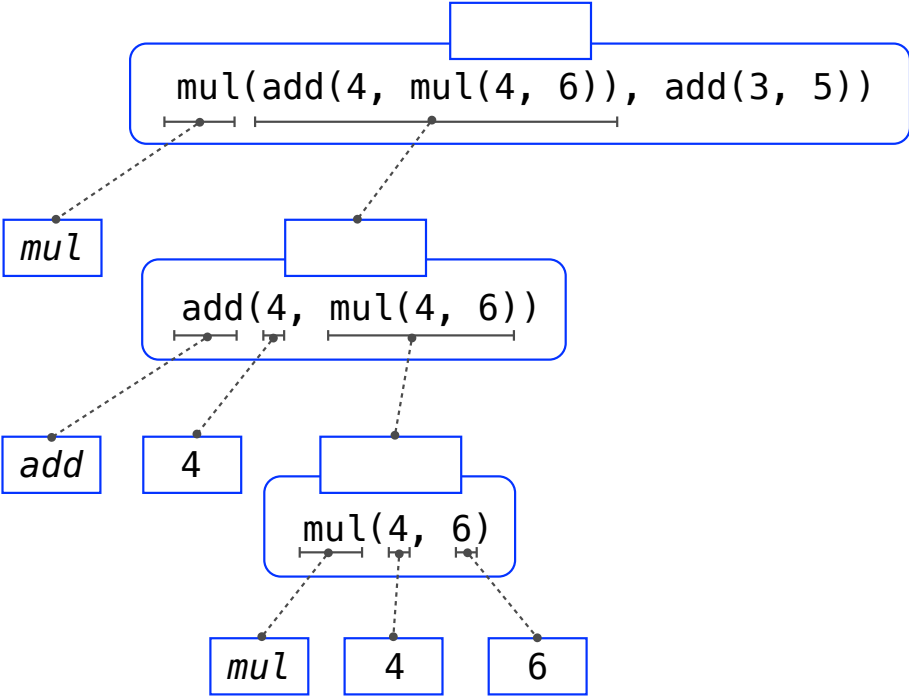


## Evaluating Nested Expressions

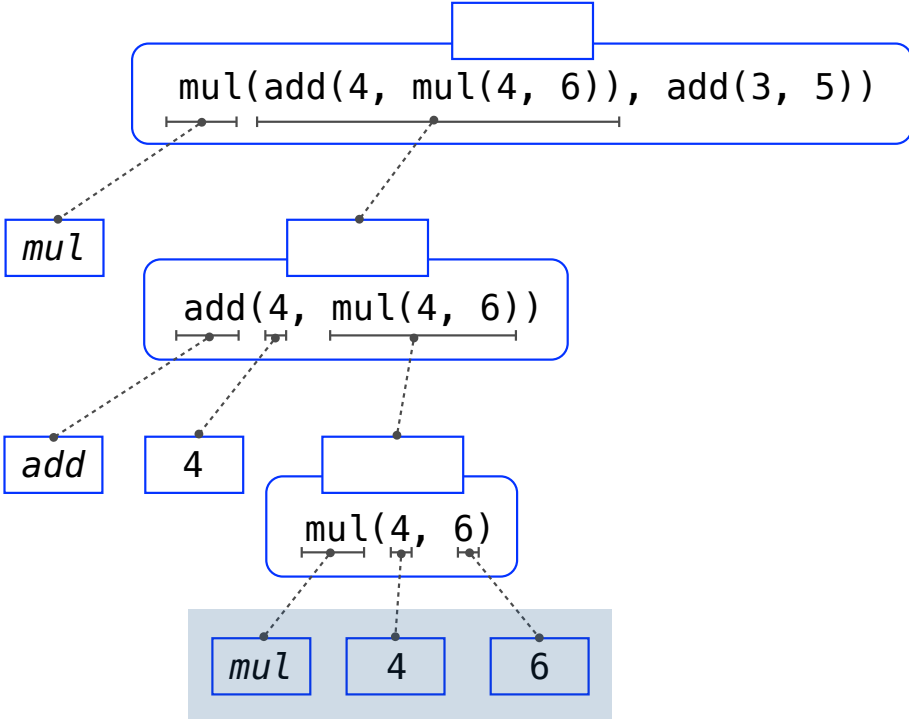
---



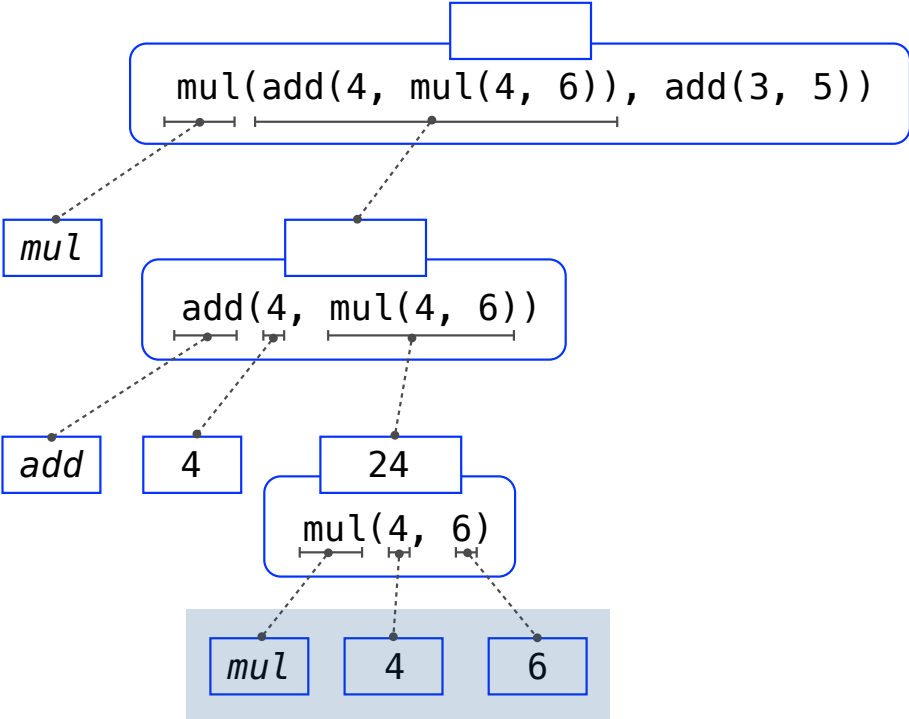
# Evaluating Nested Expressions



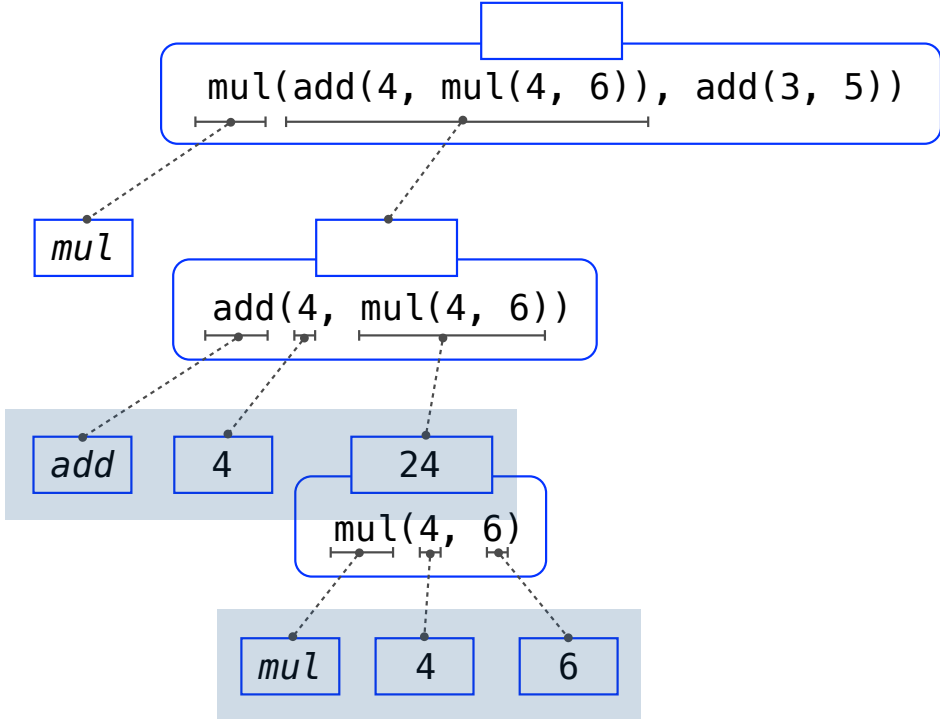
# Evaluating Nested Expressions



# Evaluating Nested Expressions

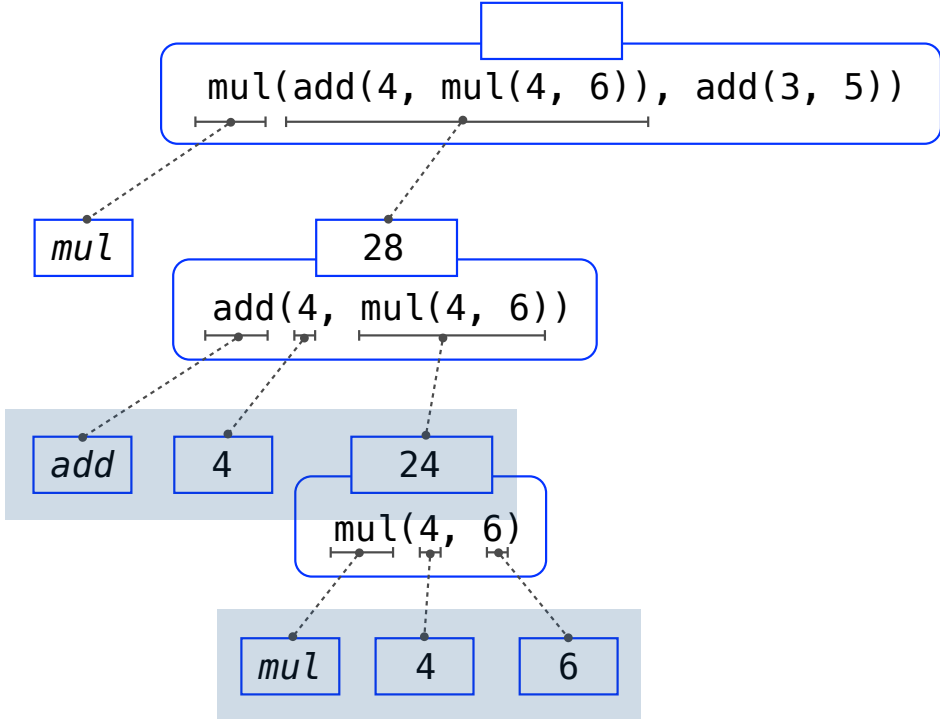


# Evaluating Nested Expressions

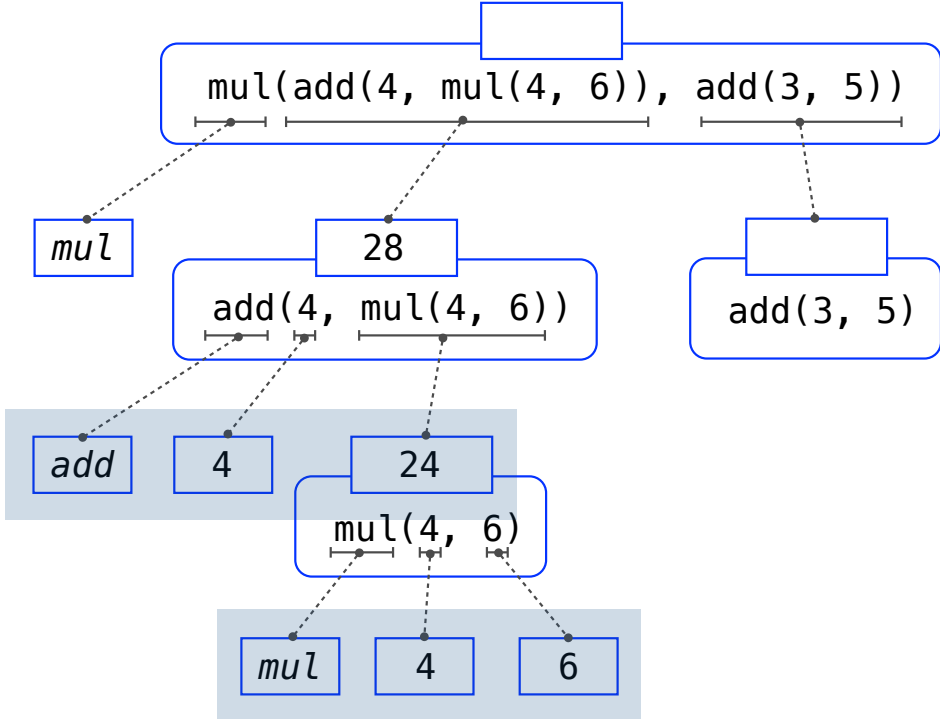




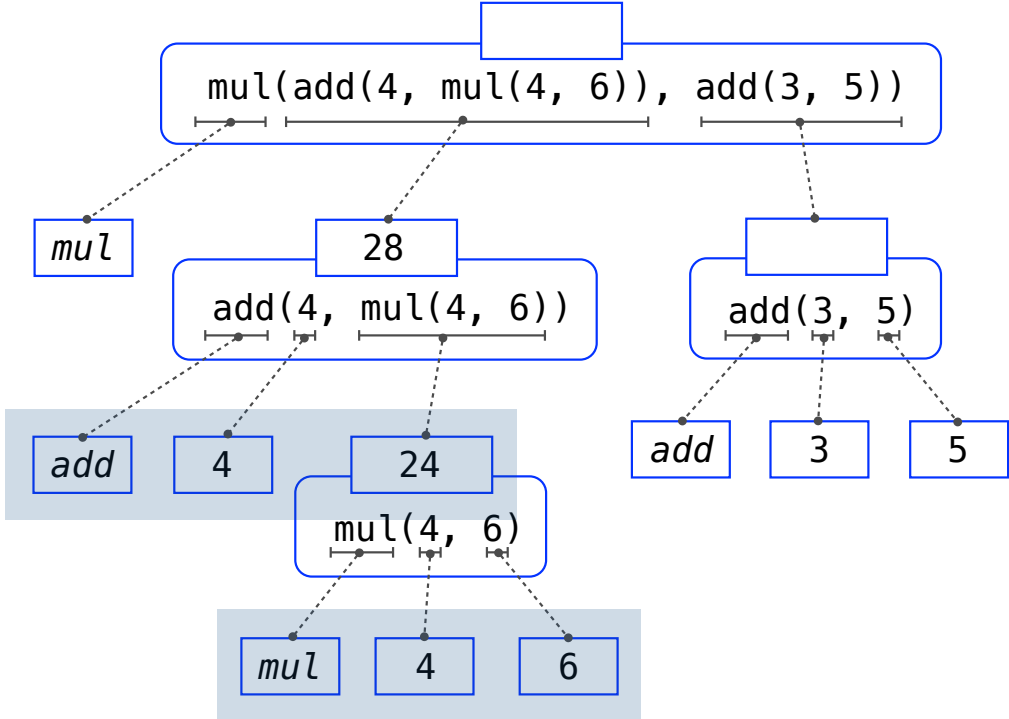
# Evaluating Nested Expressions



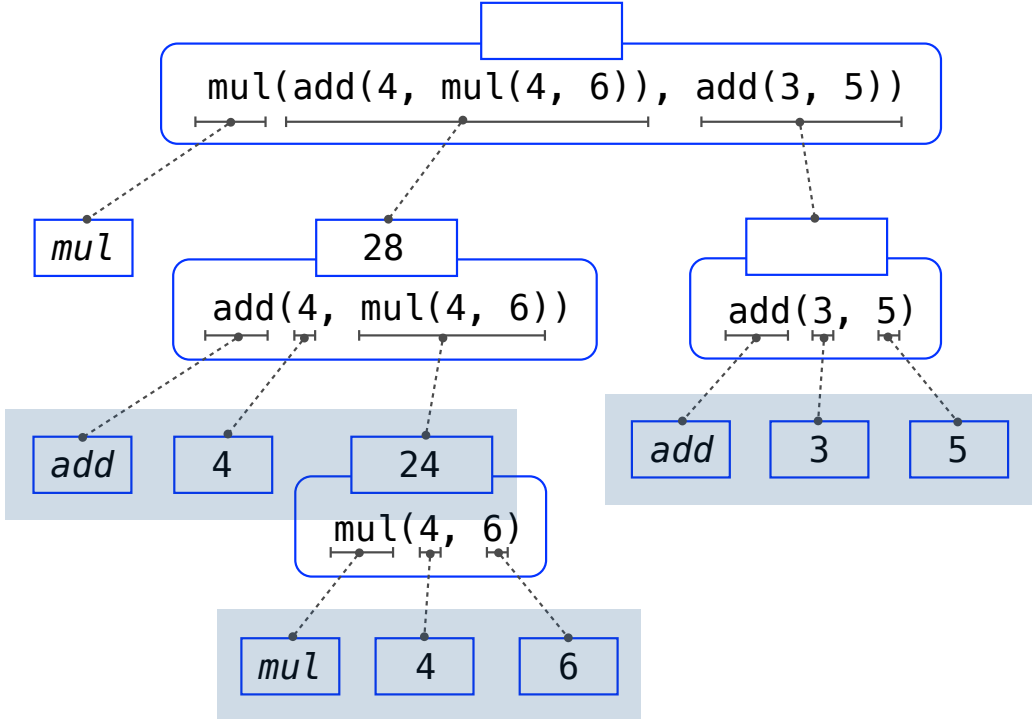
# Evaluating Nested Expressions



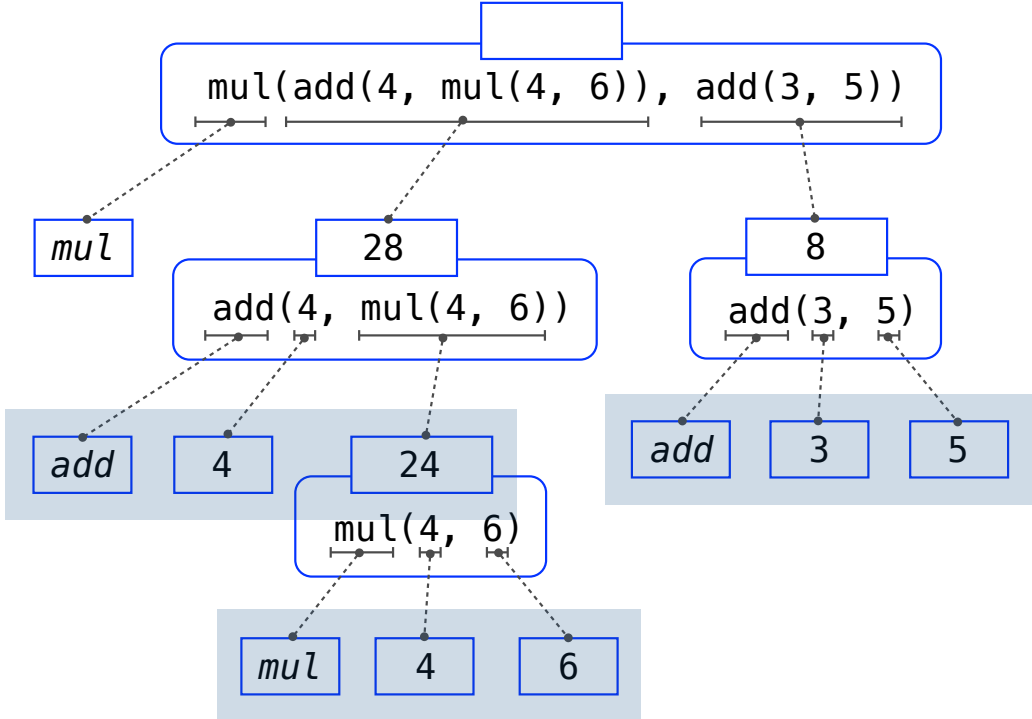
# Evaluating Nested Expressions



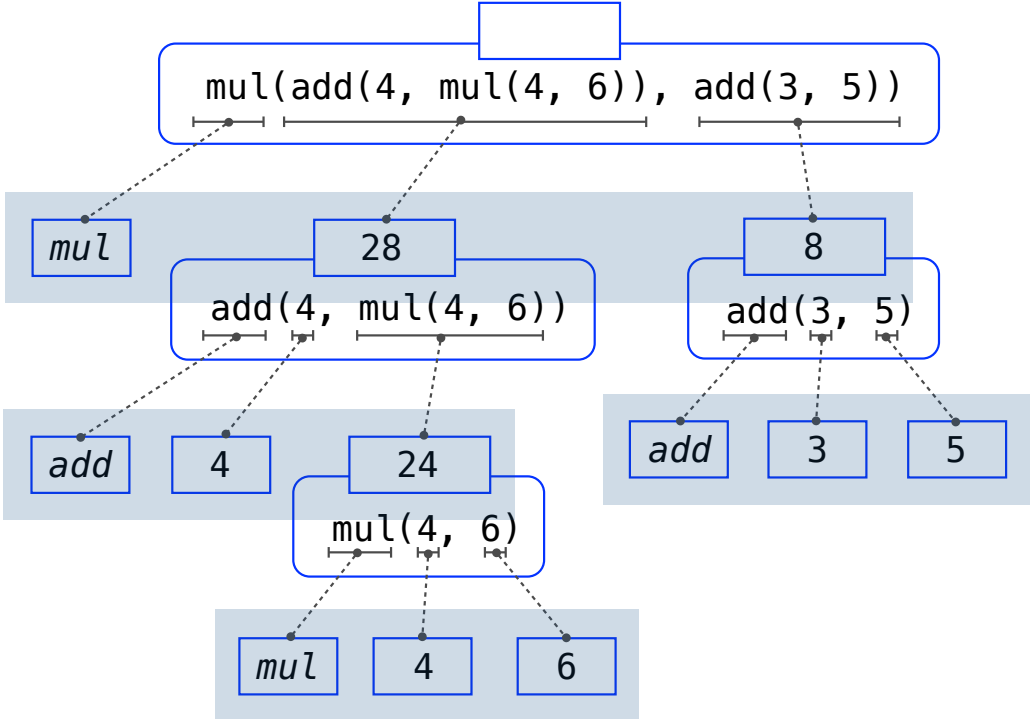
# Evaluating Nested Expressions



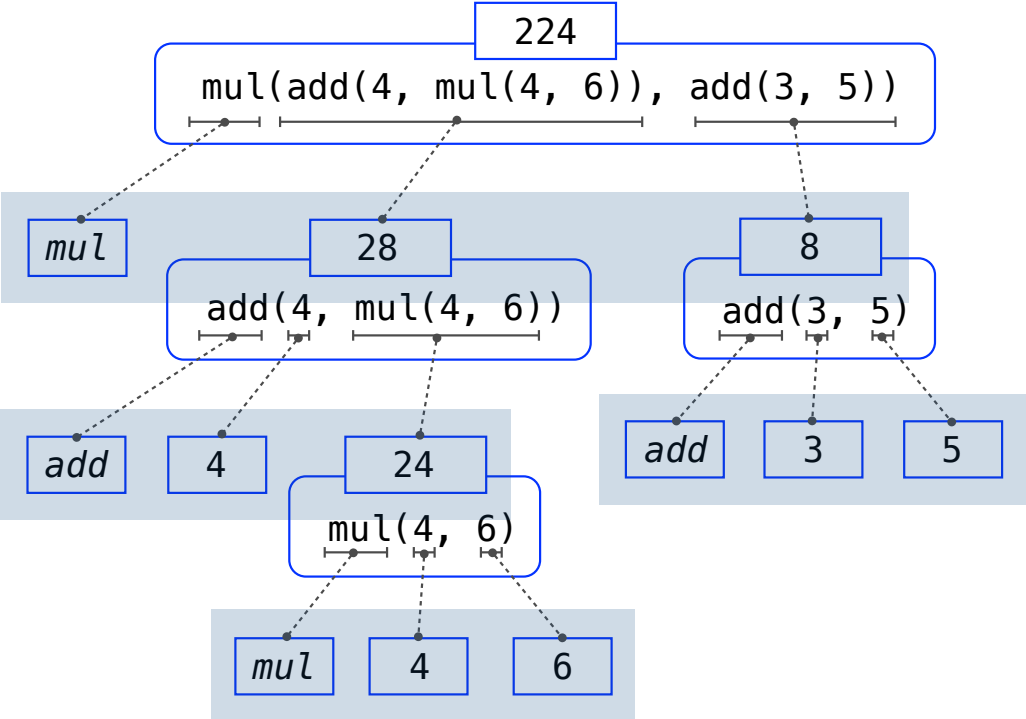
# Evaluating Nested Expressions



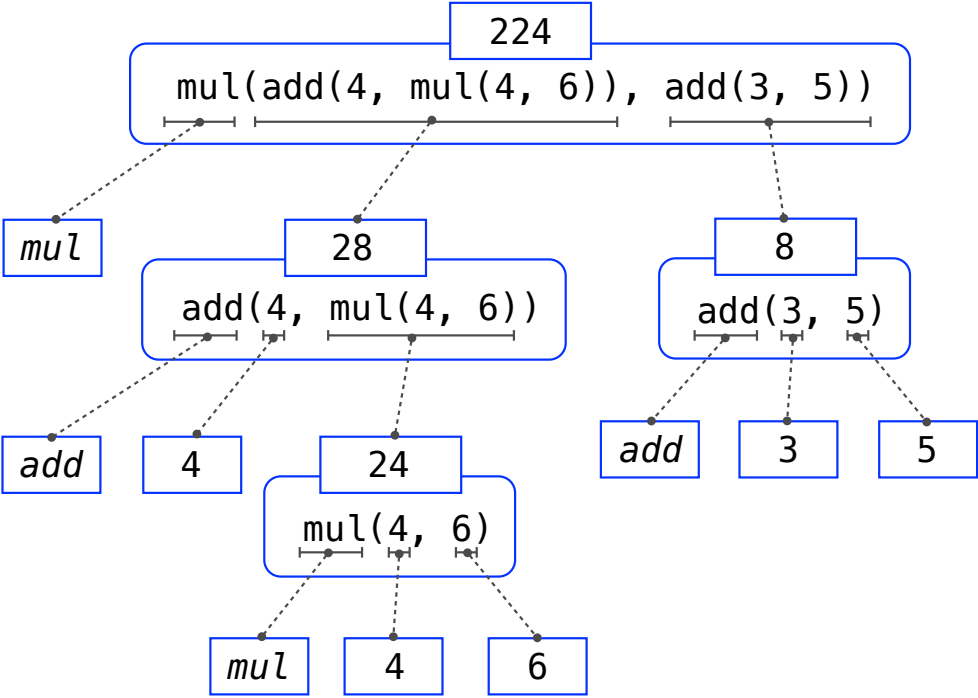
# Evaluating Nested Expressions



# Evaluating Nested Expressions

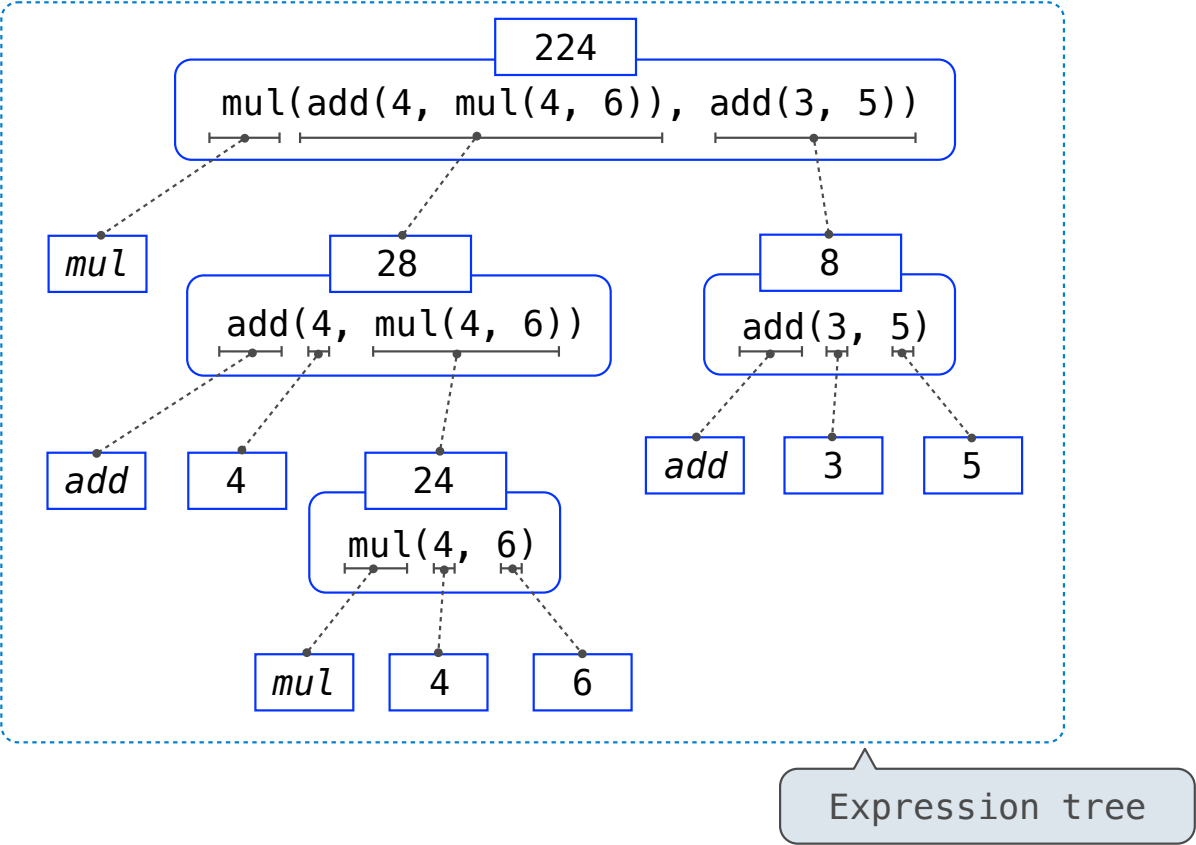


# Evaluating Nested Expressions

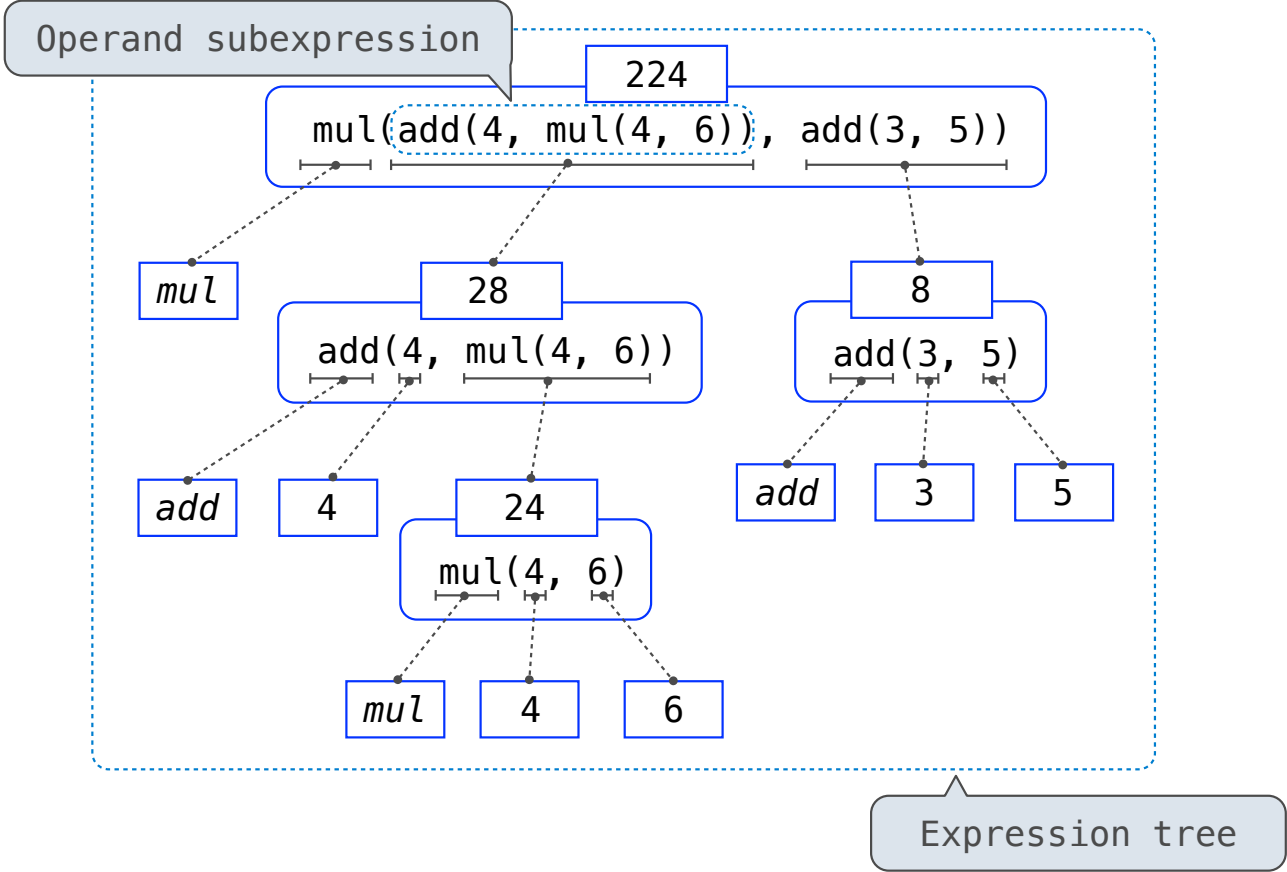




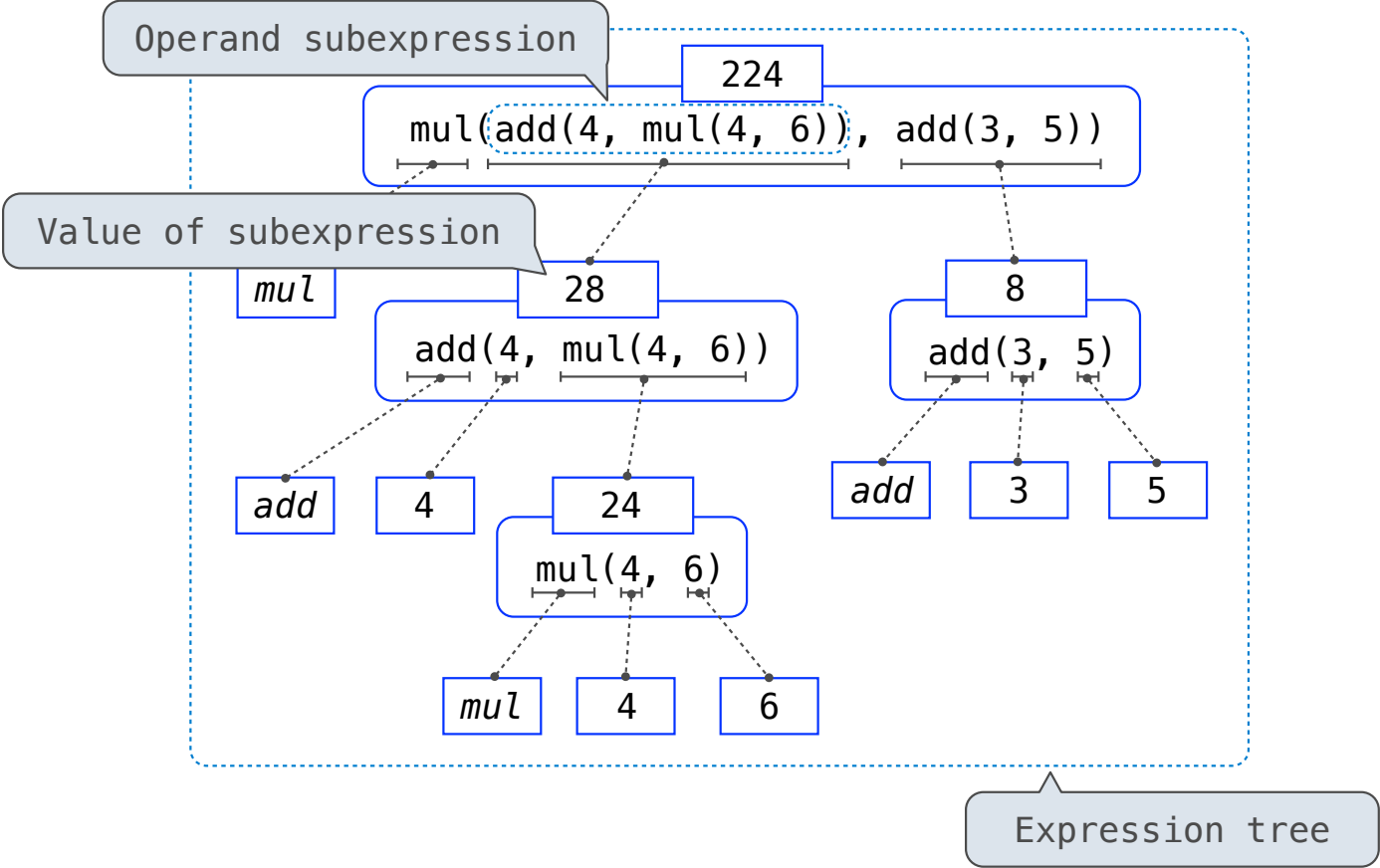
# Evaluating Nested Expressions



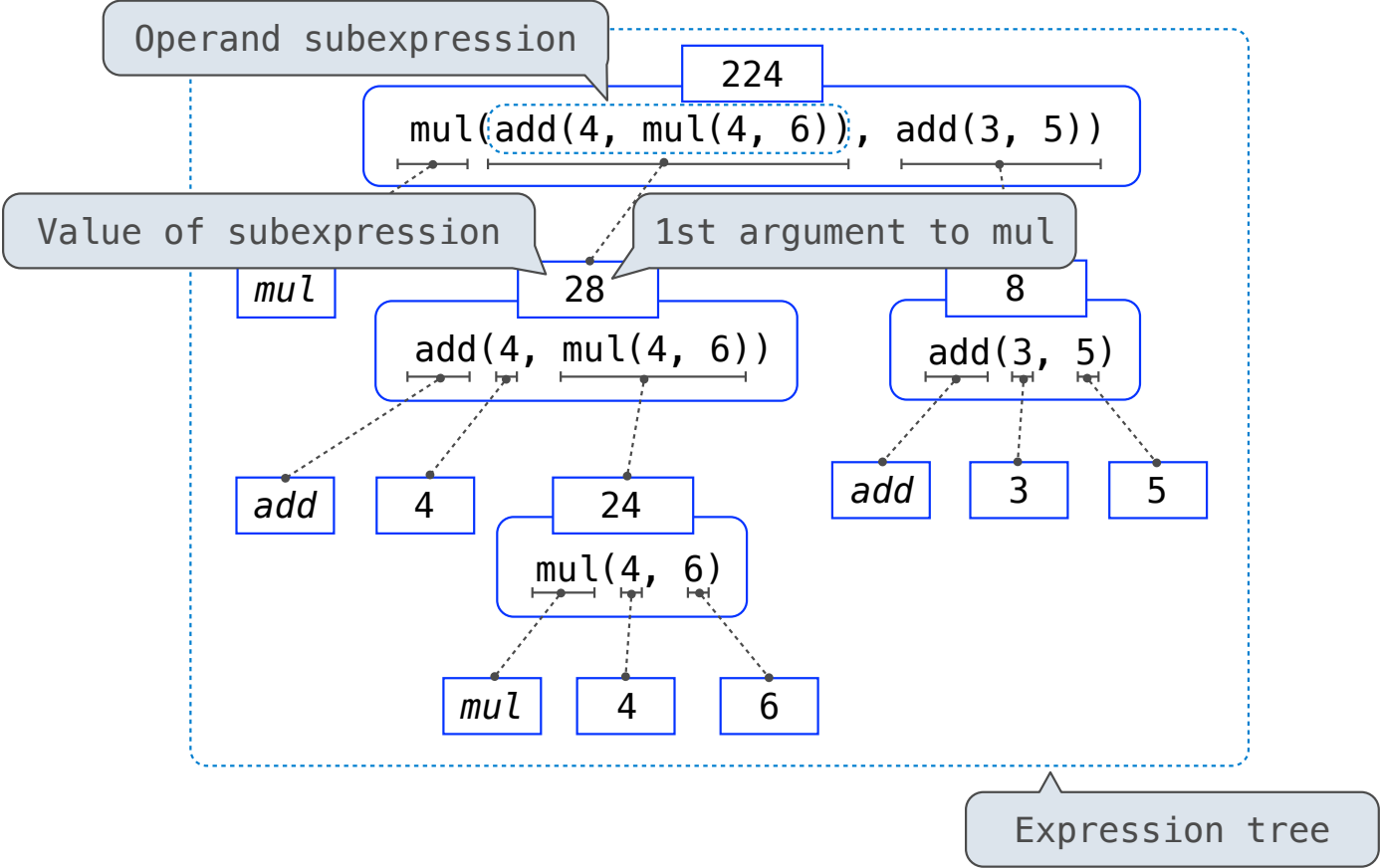
# Evaluating Nested Expressions



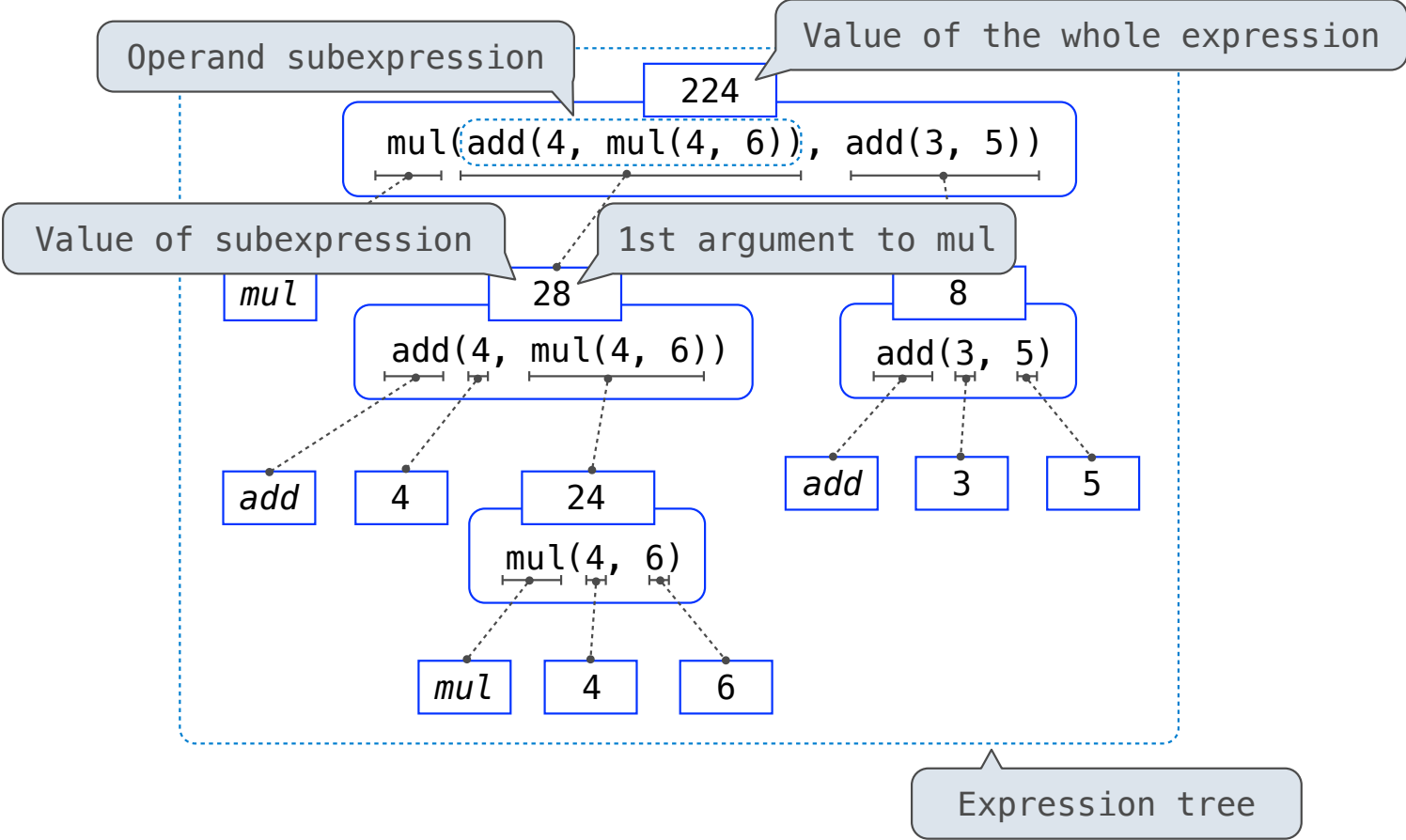
# Evaluating Nested Expressions



# Evaluating Nested Expressions



# Evaluating Nested Expressions



# Names, Assignment, and User-Defined Functions

(Demo)

## Types of Expressions

---

## Types of Expressions

---

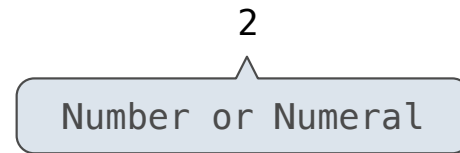
**Primitive expressions:**



## Types of Expressions

---

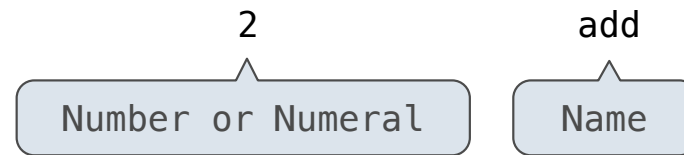
**Primitive expressions:**



## Types of Expressions

---

**Primitive expressions:**



## Types of Expressions

---

**Primitive expressions:**



## Types of Expressions

---

**Primitive expressions:**



**Call expressions:**

## Types of Expressions

---

**Primitive expressions:**



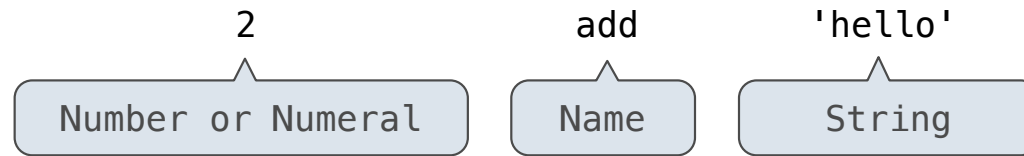
**Call expressions:**

max ( 2 , 3 )

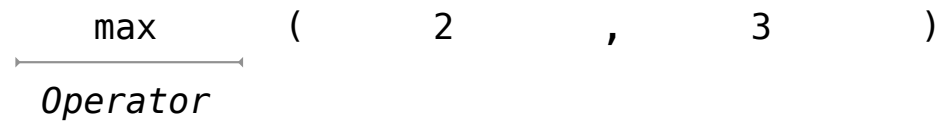
## Types of Expressions

---

**Primitive expressions:**



**Call expressions:**



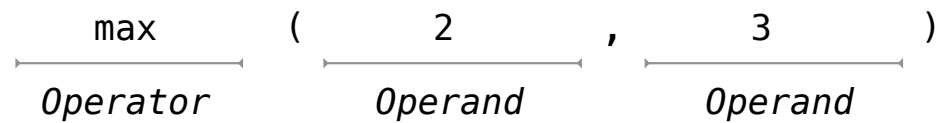
## Types of Expressions

---

**Primitive expressions:**



**Call expressions:**



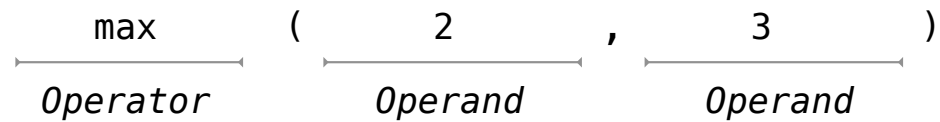
## Types of Expressions

---

**Primitive expressions:**



**Call expressions:**



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



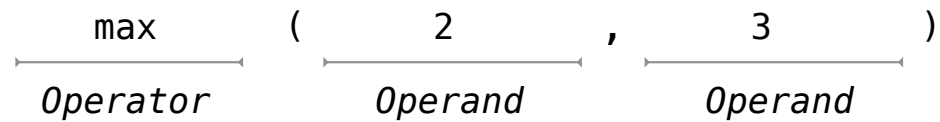
## Types of Expressions

---

### Primitive expressions:



### Call expressions:



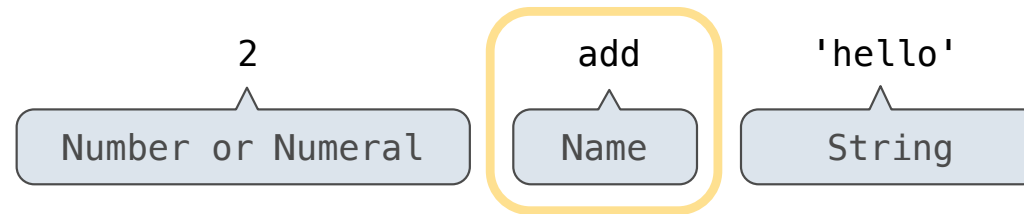
An operand can also be a call expression

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

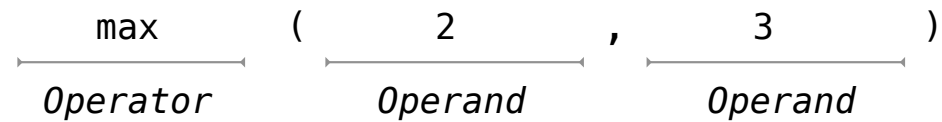
## Types of Expressions

---

**Primitive expressions:**



**Call expressions:**



An operand can also be a call expression

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

## Discussion Question 1

---

## Discussion Question 1

---

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

## Discussion Question 1

---

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

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

## Discussion Question 1

---

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

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

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

## Discussion Question 1

---

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

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

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

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

## Discussion Question 1

---

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

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

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

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

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



## Discussion Question 1

---

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

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

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

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

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

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

## Discussion Question 1

---

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

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

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

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

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

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

**???**

## Discussion Question 1

---

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

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

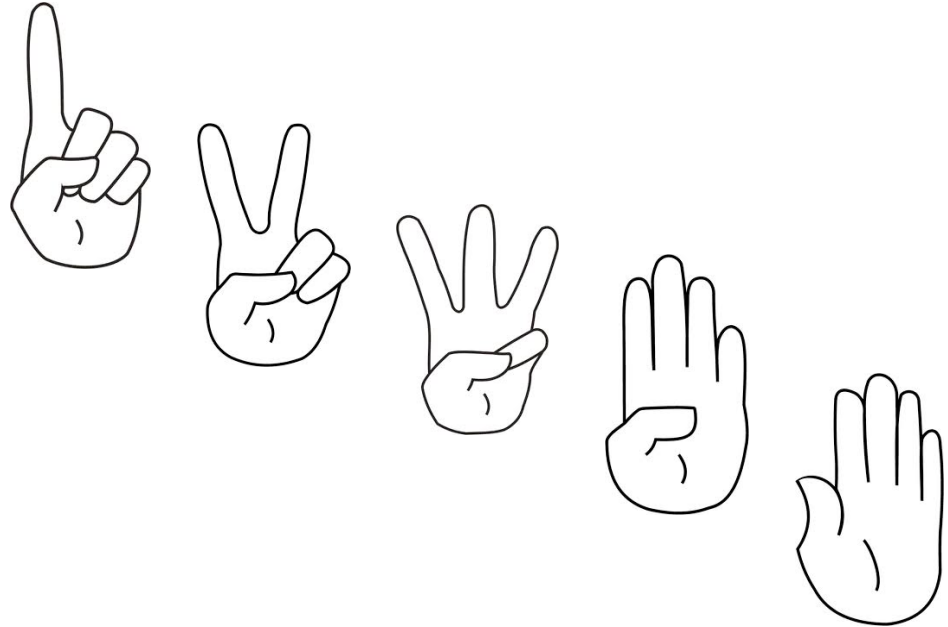
```
>>> f = max
```

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

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

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

**???**



# Environment Diagrams



## Environment Diagrams

---

Environment diagrams visualize the interpreter's process.

---

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

---

## Environment Diagrams

---

Environment diagrams visualize the interpreter's process.

---

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

---

```
Global frame  
pi 3.1416
```

## 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):**



## 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):**

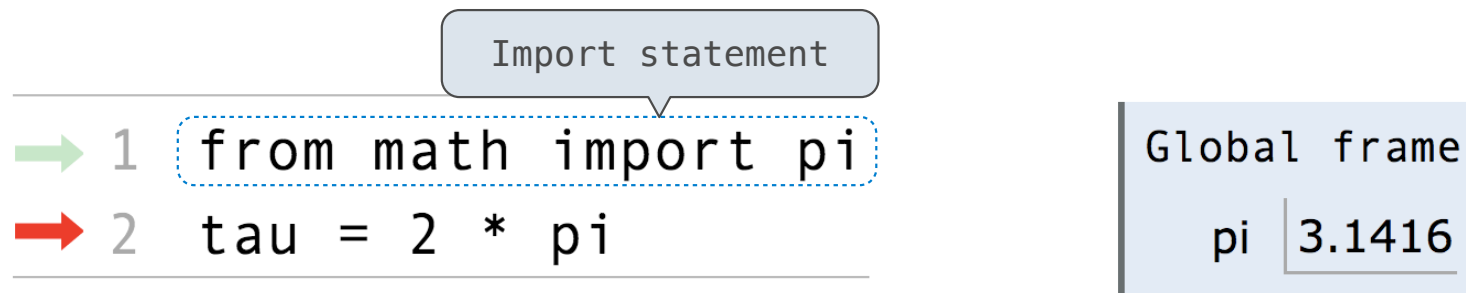
Statements and expressions

**Frames (right):**

## Environment Diagrams

---

Environment diagrams visualize the interpreter's process.



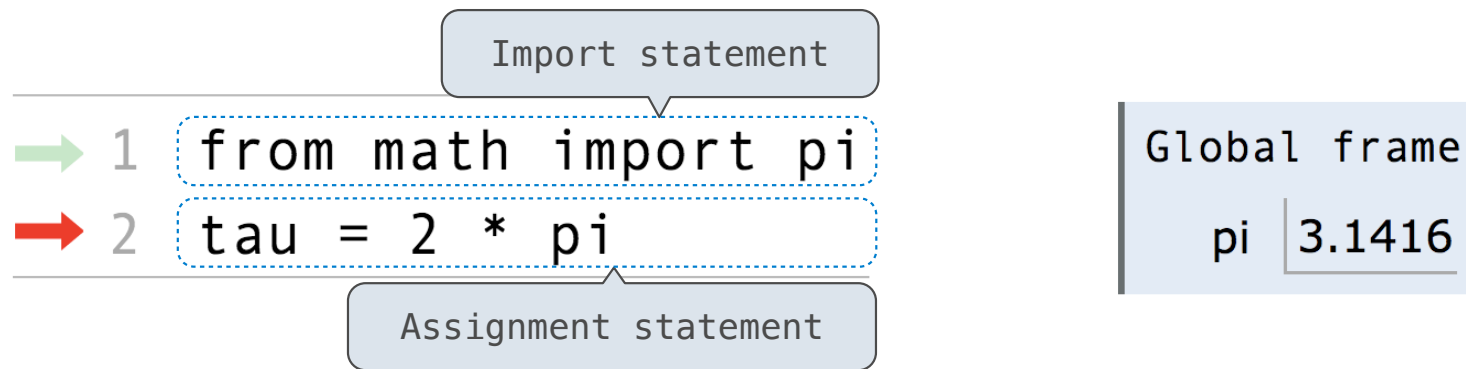
**Code (left):**

Statements and expressions

**Frames (right):**

## Environment Diagrams

Environment diagrams visualize the interpreter's process.



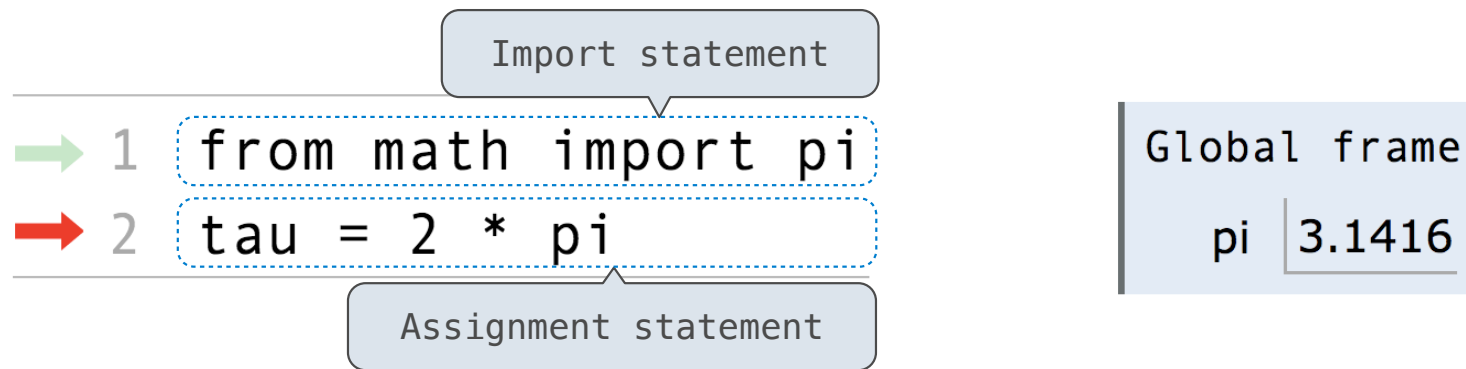
**Code (left):**

Statements and expressions

**Frames (right):**

## Environment Diagrams

Environment diagrams visualize the interpreter's process.



### Code (left):

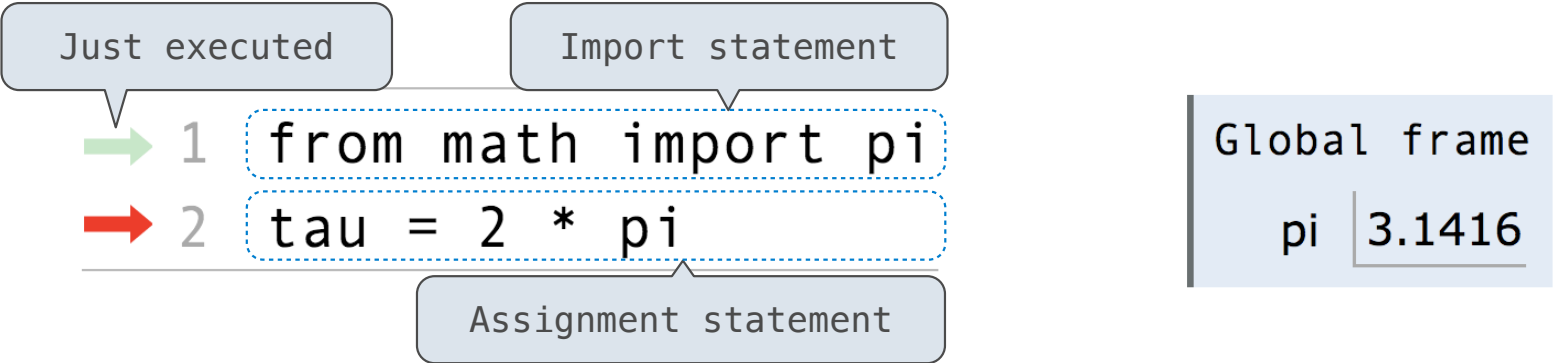
Statements and expressions

Arrows indicate evaluation order

### Frames (right):

# Environment Diagrams

Environment diagrams visualize the interpreter's process.



### Code (left):

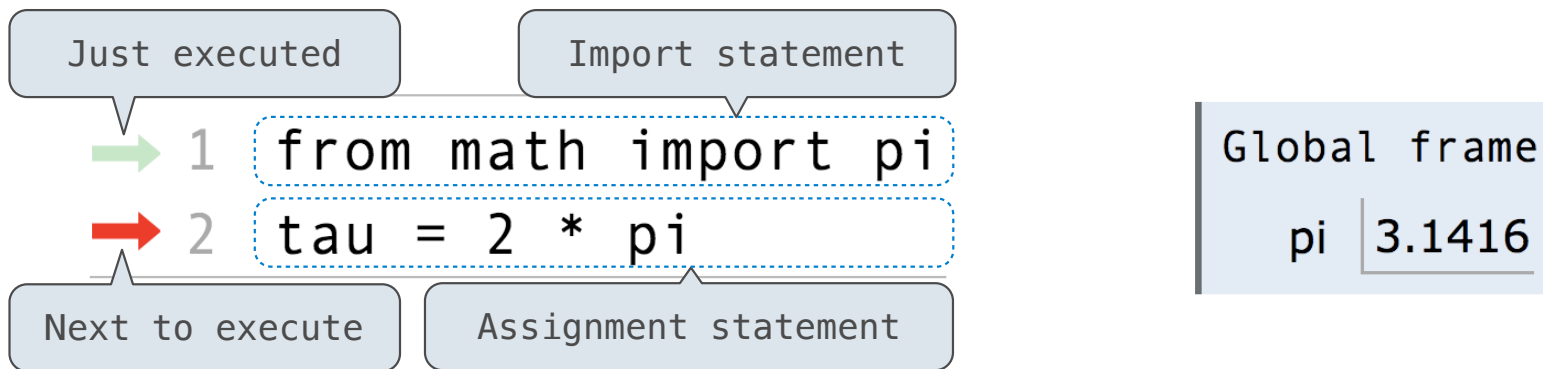
Statements and expressions

Arrows indicate evaluation order

### Frames (right):

## Environment Diagrams

Environment diagrams visualize the interpreter's process.



### Code (left):

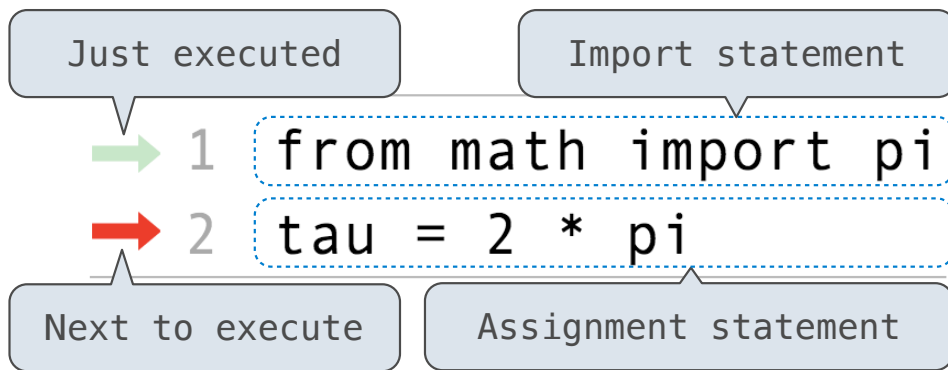
Statements and expressions

Arrows indicate evaluation order

### Frames (right):

## Environment Diagrams

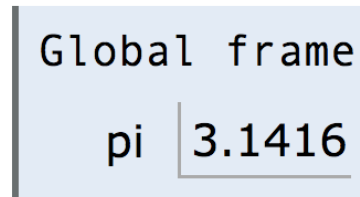
Environment diagrams visualize the interpreter's process.



### Code (left):

Statements and expressions

Arrows indicate evaluation order

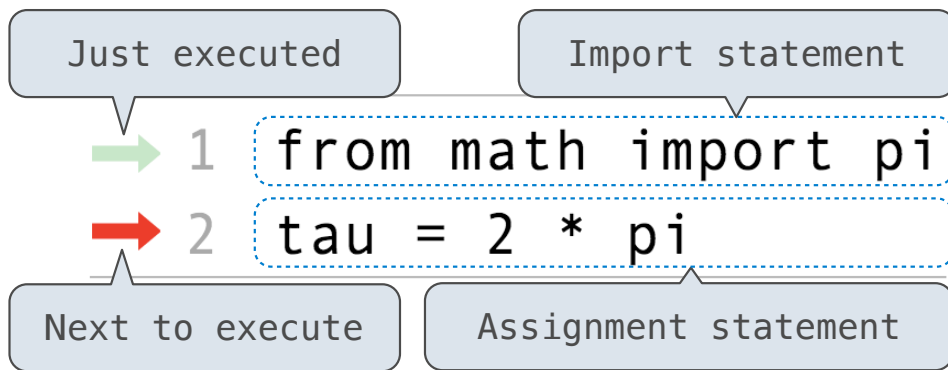


### Frames (right):

Each name is bound to a value

## Environment Diagrams

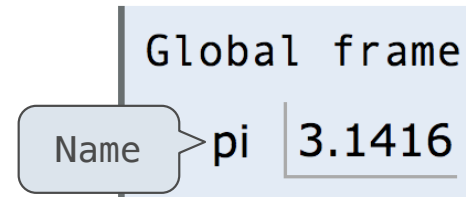
Environment diagrams visualize the interpreter's process.



### Code (left):

Statements and expressions

Arrows indicate evaluation order



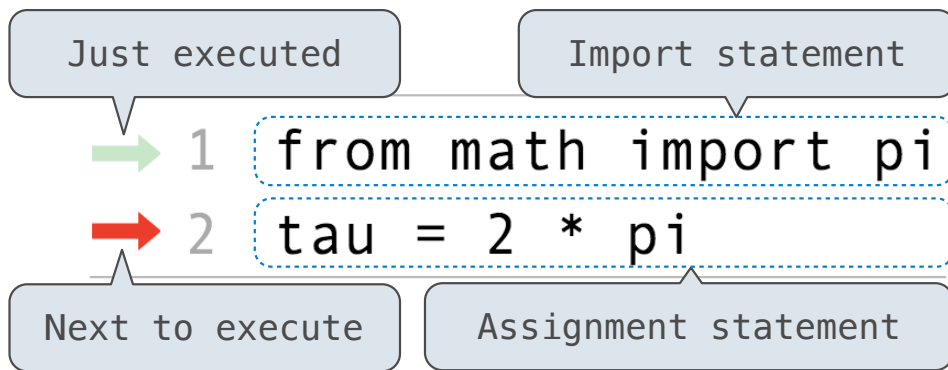
### Frames (right):

Each name is bound to a value



## Environment Diagrams

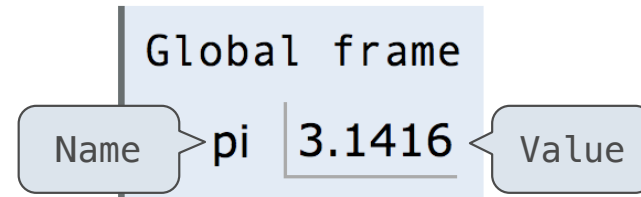
Environment diagrams visualize the interpreter's process.



### Code (left):

Statements and expressions

Arrows indicate evaluation order

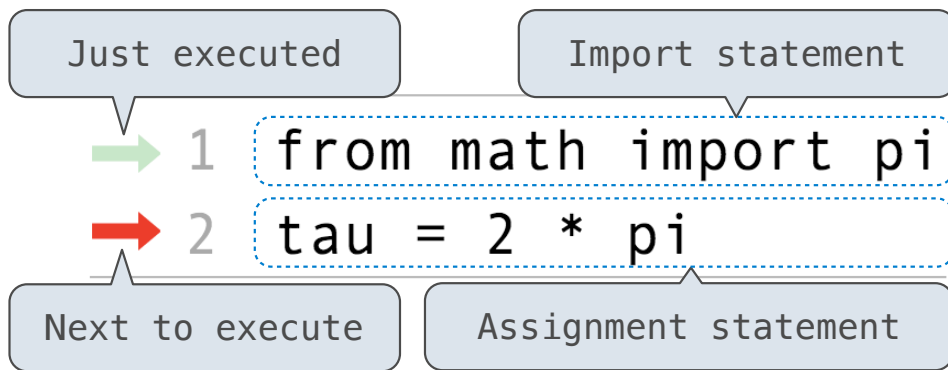


### Frames (right):

Each name is bound to a value

## Environment Diagrams

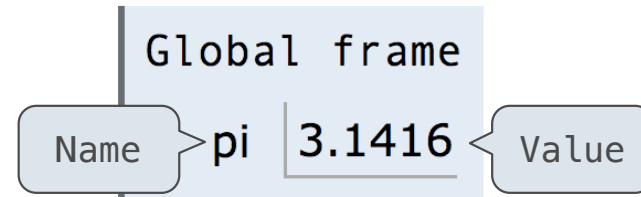
Environment diagrams visualize the interpreter's process.



### Code (left):

Statements and expressions

Arrows indicate evaluation order



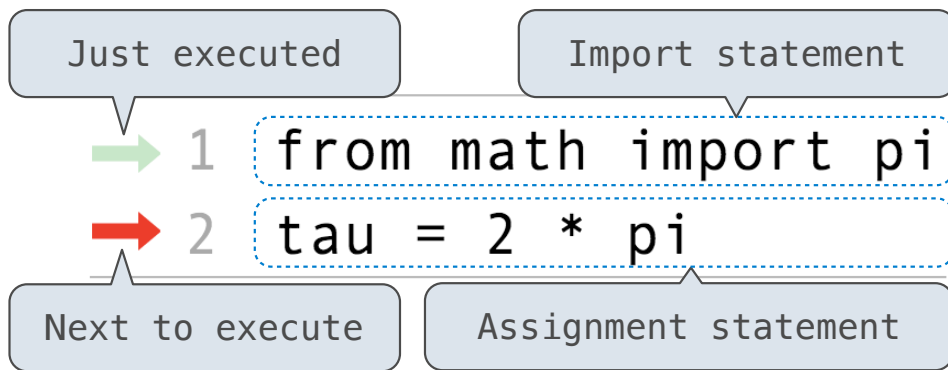
### Frames (right):

Each name is bound to a value

Within a frame, a name cannot be repeated

## Environment Diagrams

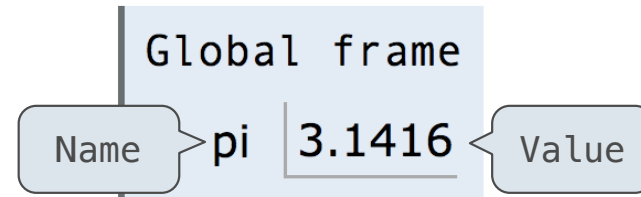
Environment diagrams visualize the interpreter's process.



### 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)

# Assignment Statements

---





















## Discussion Question 1 Solution

---

(Demo)







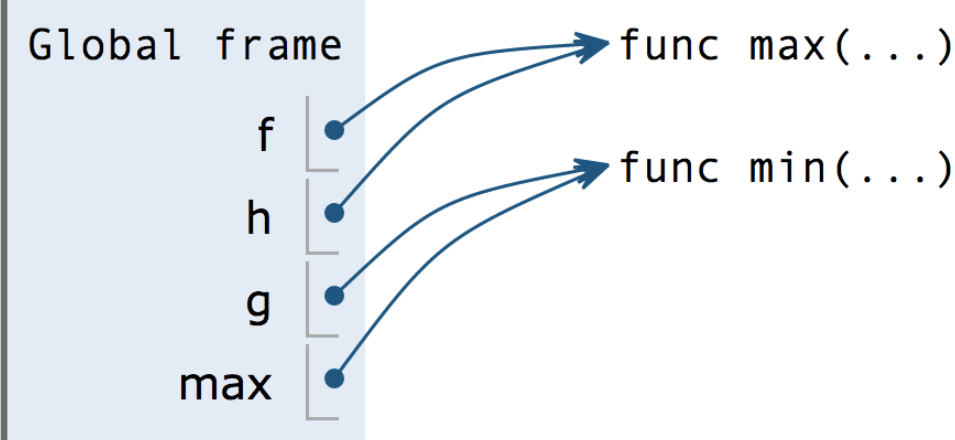




# Discussion Question 1 Solution

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

(Demo)





























## 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

---

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

## Calling User-Defined Functions

---

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

1. Add a local frame, forming a new environment

## Calling User-Defined Functions

---

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

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

## Calling User-Defined Functions

---

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

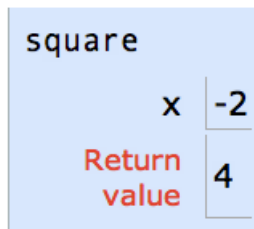
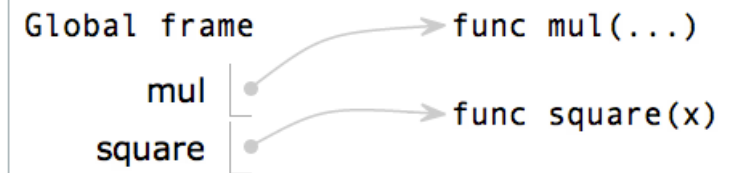
1. Add a local frame, forming a new environment
2. Bind the function's formal parameters to its arguments in that frame
3. Execute the body of the function in that new environment

## Calling User-Defined Functions

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

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

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







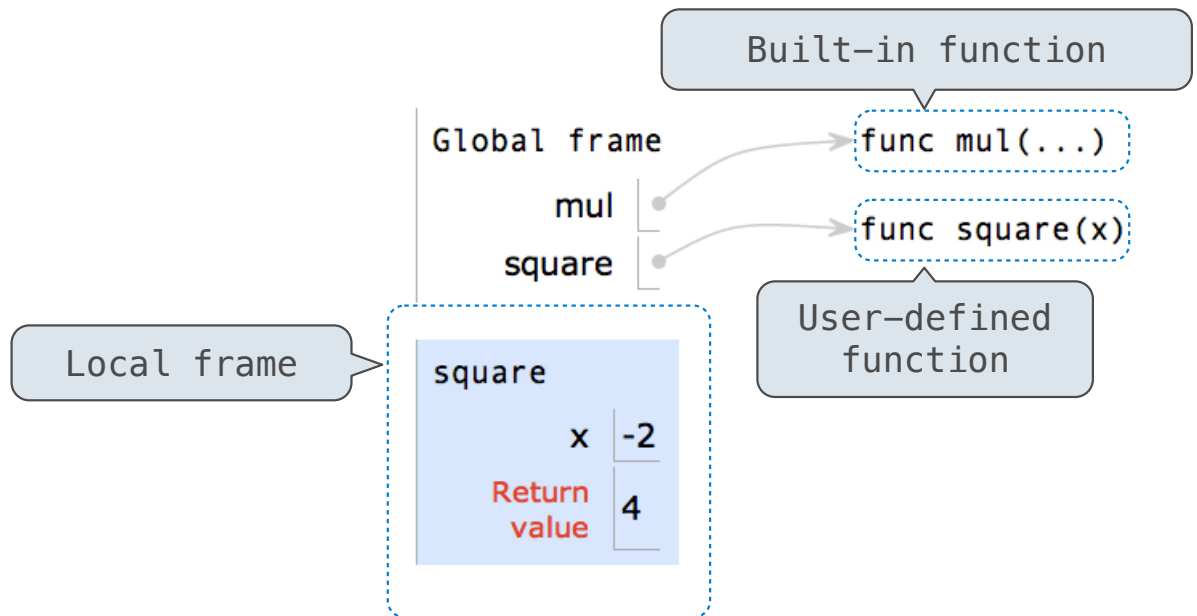


## Calling User-Defined Functions

### 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)
```

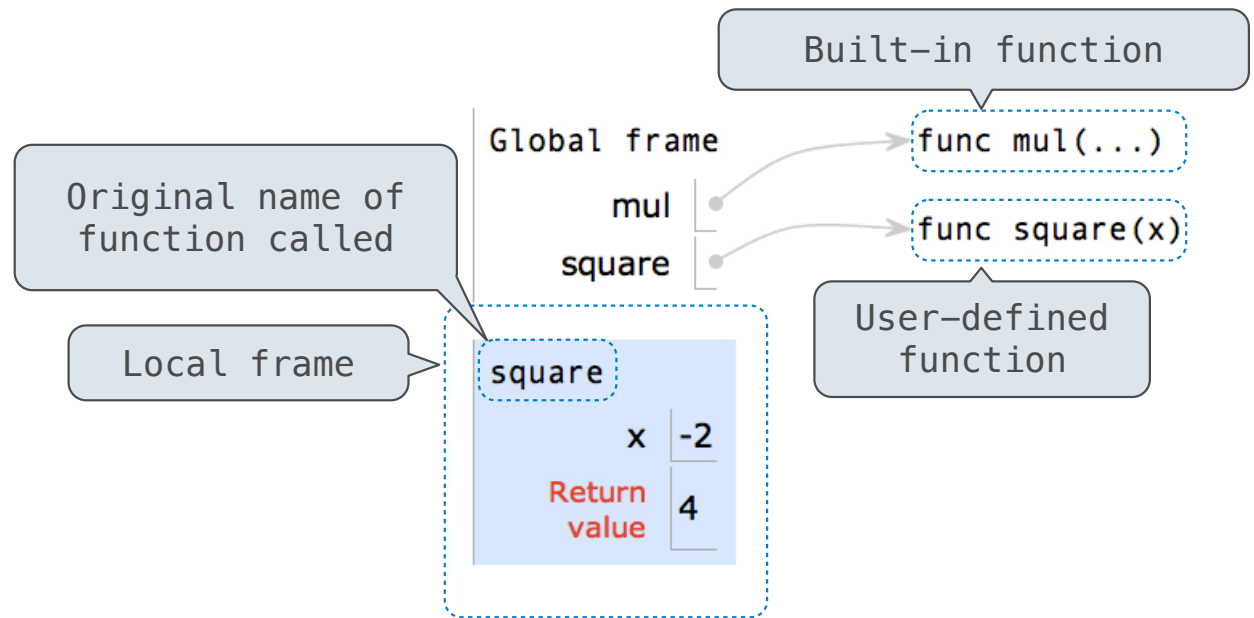


## Calling User-Defined Functions

### 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)
```





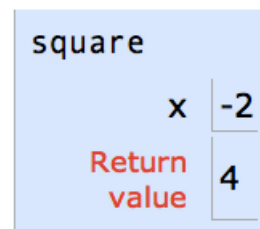
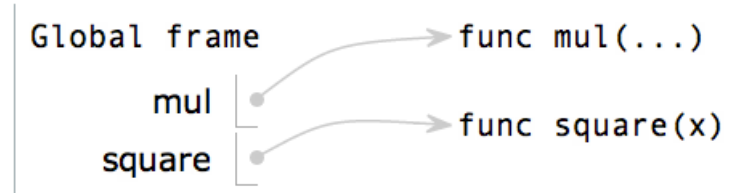


## Calling User-Defined Functions

### 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)
```



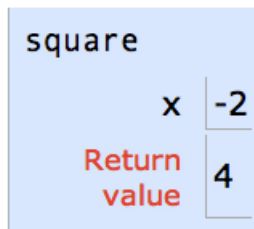
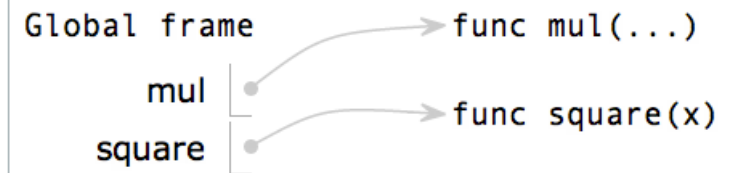
## Calling User-Defined Functions

### 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)
```

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)