61A Lecture 2

Monday, August 29

The Elements of Programming

The Elements of Programming

- Primitive Expressions and Statements
  - *The simplest building blocks of a language*

The Elements of Programming

- Primitive Expressions and Statements
  - *The simplest building blocks of a language*

- Means of Combination
  - *Compound elements are built from simpler ones*

The Elements of Programming

- Primitive Expressions and Statements
  - *The simplest building blocks of a language*

- Means of Combination
  - *Compound elements are built from simpler ones*

- Means of Abstraction
  - *Compound elements can be named and manipulated as units*

The Elements of Programming

- Primitive Expressions and Statements
  - *The simplest building blocks of a language*

- Means of Combination
  - *Compound elements are built from simpler ones*

- Means of Abstraction
  - *Compound elements can be named and manipulated as units*

Programming languages allow us to communicate, too

Functions and Data

Data: Stuff we want to manipulate

Functions and Data

Data: Stuff we want to manipulate

Functions: Rules for manipulating data

Functions and Data

Data: Stuff we want to manipulate

2

Functions: Rules for manipulating data

Functions and Data

Data: Stuff we want to manipulate

“The Art of Computer Programming”

2

Functions: Rules for manipulating data

Functions and Data

Data: Stuff we want to manipulate

“The Art of Computer Programming”

2

Donald Knuth

Functions: Rules for manipulating data

Functions and Data

Data: Stuff we want to manipulate

“The Art of Computer Programming”

2

Donald Knuth

This slide

Functions: Rules for manipulating data

Functions and Data

Data: Stuff we want to manipulate

“The Art of Computer Programming”

2

This slide

Donald Knuth

Functions: Rules for manipulating data

Add numbers

Functions and Data

Data: Stuff we want to manipulate

“The Art of Computer Programming”

2

This slide

Donald Knuth

Functions: Rules for manipulating data

Count the words in a line of text

Add numbers

Functions and Data

Data: Stuff we want to manipulate

“The Art of Computer Programming”

2

This slide

Donald Knuth

Functions: Rules for manipulating data

Count the words in a line of text

Add numbers

Pronounce someone’s name

Functions and Data

Data: Stuff we want to manipulate

“The Art of Computer Programming”

2

This slide

Donald Knuth
(Ka-NOOTH)

Functions: Rules for manipulating data

Count the words in a line of text

Add numbers

Pronounce someone’s name

Functions and Data

Data: Stuff we want to manipulate

"The Art of Computer Programming"

2

This slide

Donald Knuth
(Ka-NOOTH)

Functions: Rules for manipulating data

Count the words in a line of text

Add numbers

Load the next slide

Pronounce someone's name

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

$18 + 69$

$\frac{6}{23}$

$\sin \pi$

$f(x)$

$\sqrt{3493161}$

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

$|-1869|$

$\binom{69}{18}$

Call Expressions in Python

All expressions can use function call notation

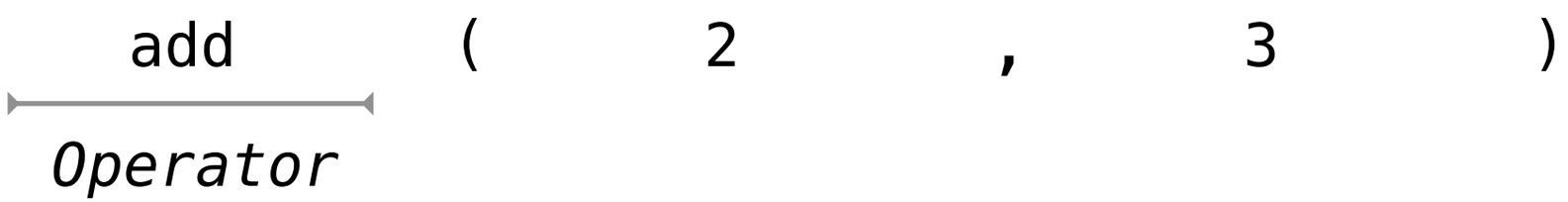
(Demo)

Anatomy of a Call Expression

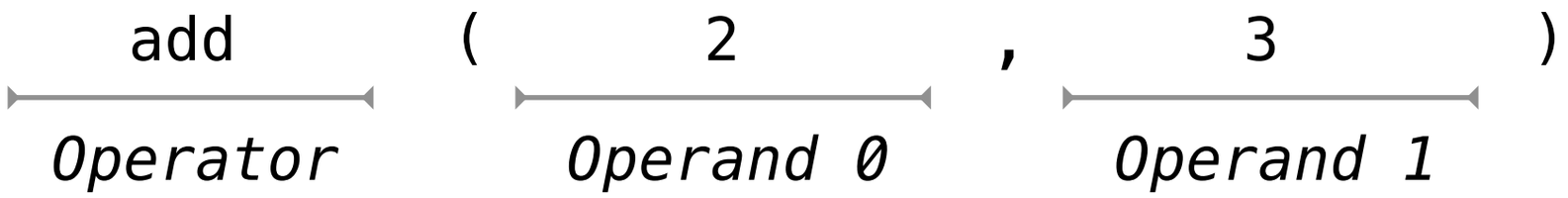
Anatomy of a Call Expression

```
add     (     2     ,     3     )
```

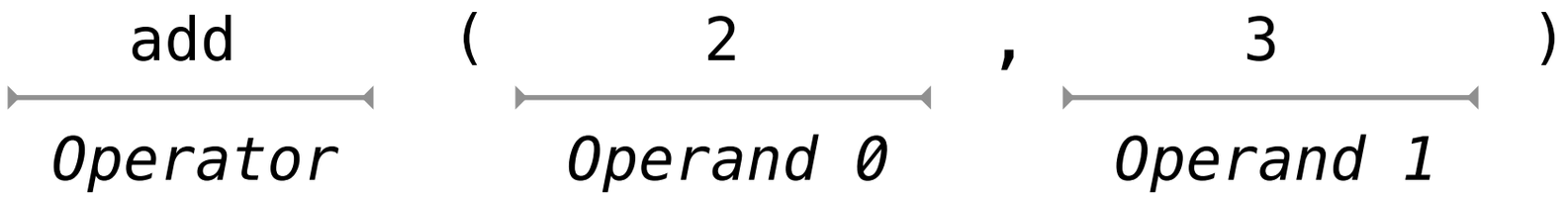
Anatomy of a Call Expression



Anatomy of a Call Expression

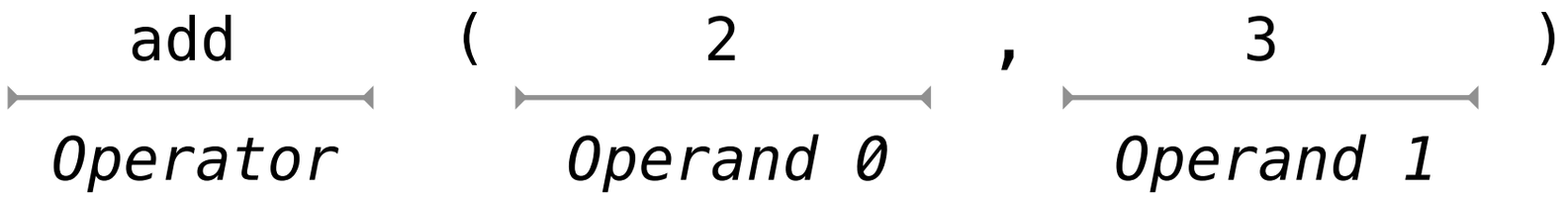


Anatomy of a Call Expression



Operators and operands are expressions

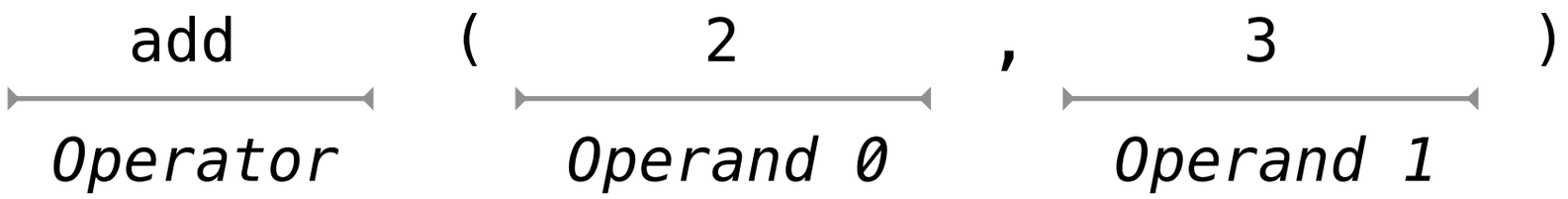
Anatomy of a Call Expression



Operators and operands are expressions

So they evaluate to values

Anatomy of a Call Expression

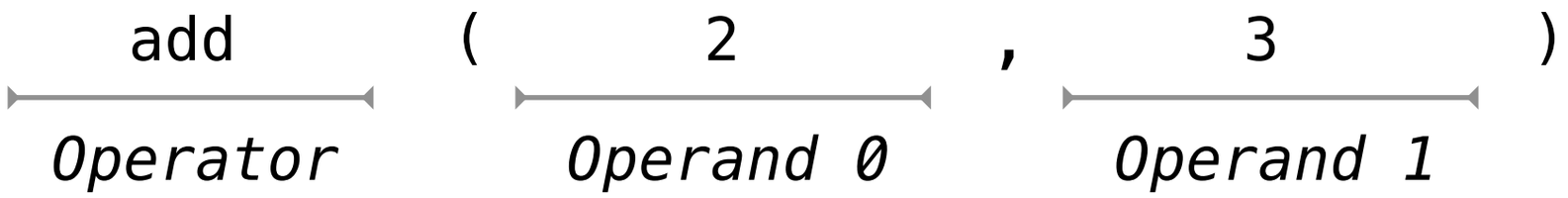


Operators and operands are expressions

So they evaluate to values

Evaluation procedure for call expressions:

Anatomy of a Call Expression



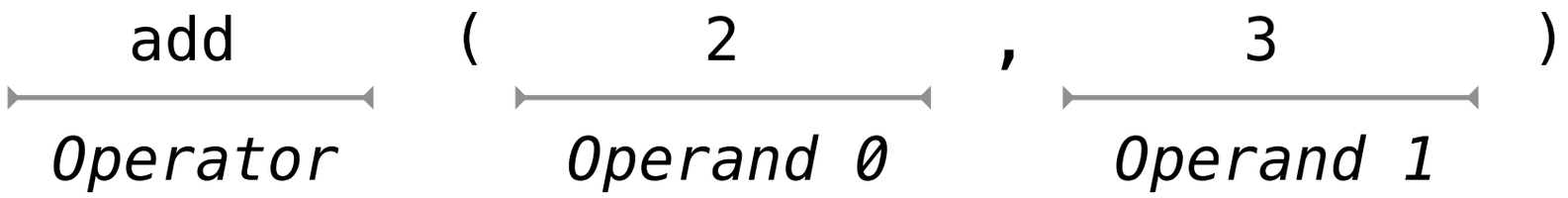
Operators and operands are expressions

So they evaluate to values

Evaluation procedure for call expressions:

1. Evaluate the operator and operand subexpressions

Anatomy of a Call Expression



Operators and operands are expressions

So they evaluate to values

Evaluation procedure for call expressions:

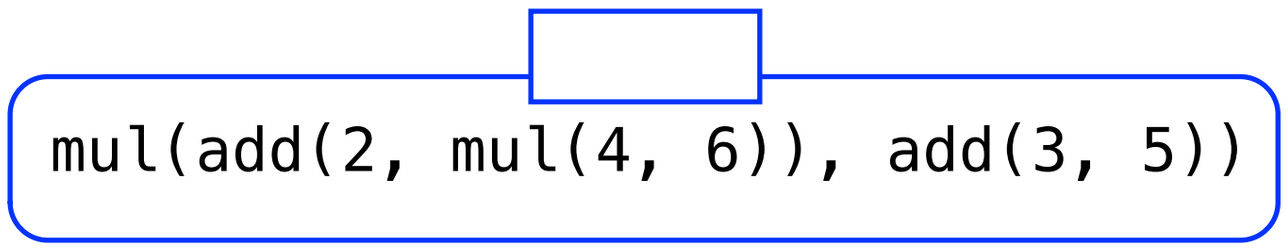
1. Evaluate the operator and operand subexpressions

2. Apply the function that is the value of the operator subexpression to the arguments that are the values of the operand subexpression

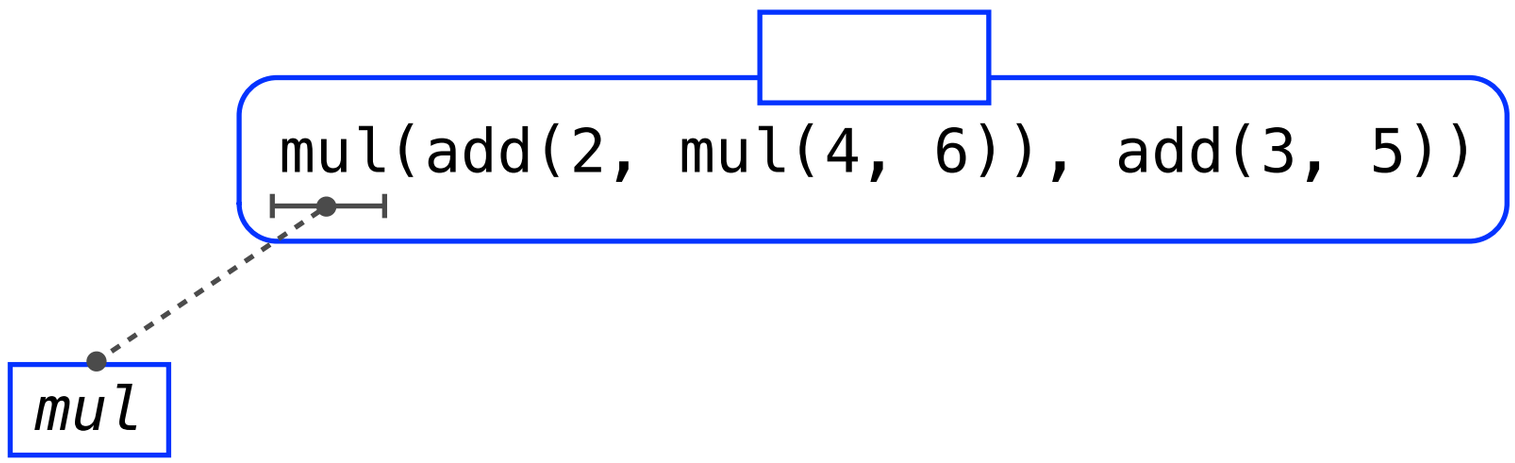
Evaluating Nested Expressions

```
mul(add(2, mul(4, 6)), add(3, 5))
```

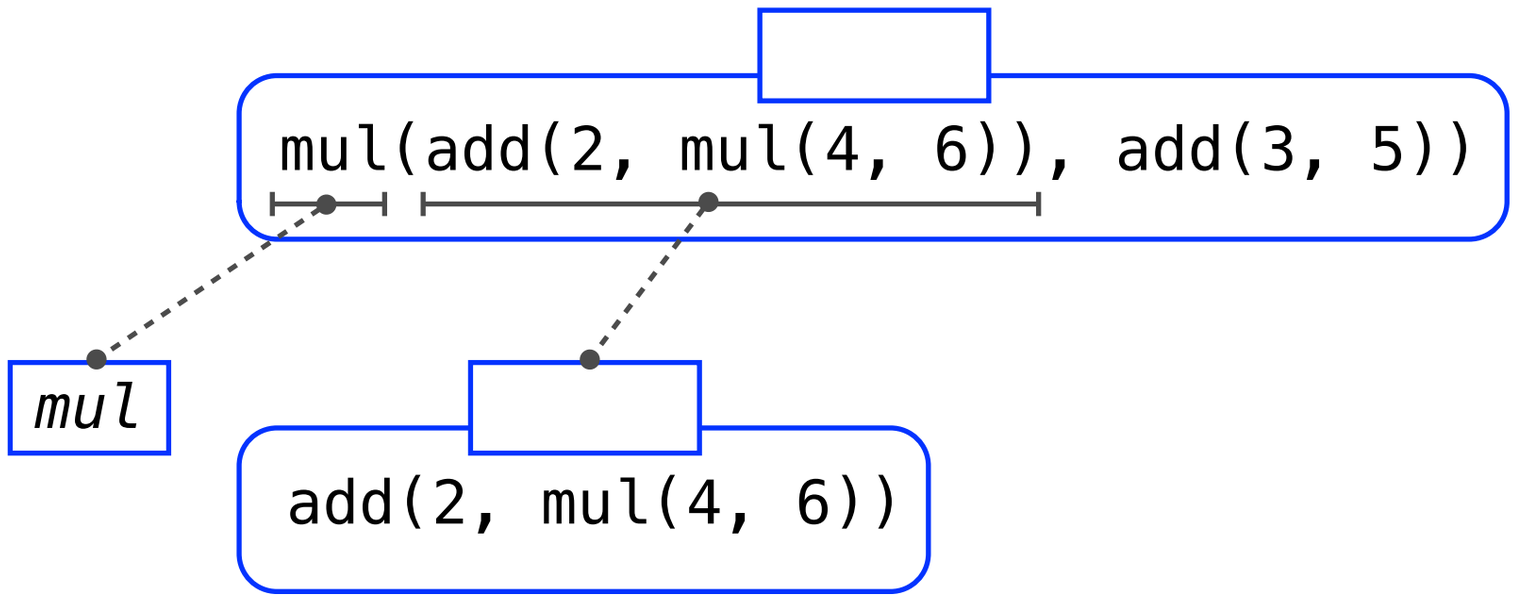
Evaluating Nested Expressions



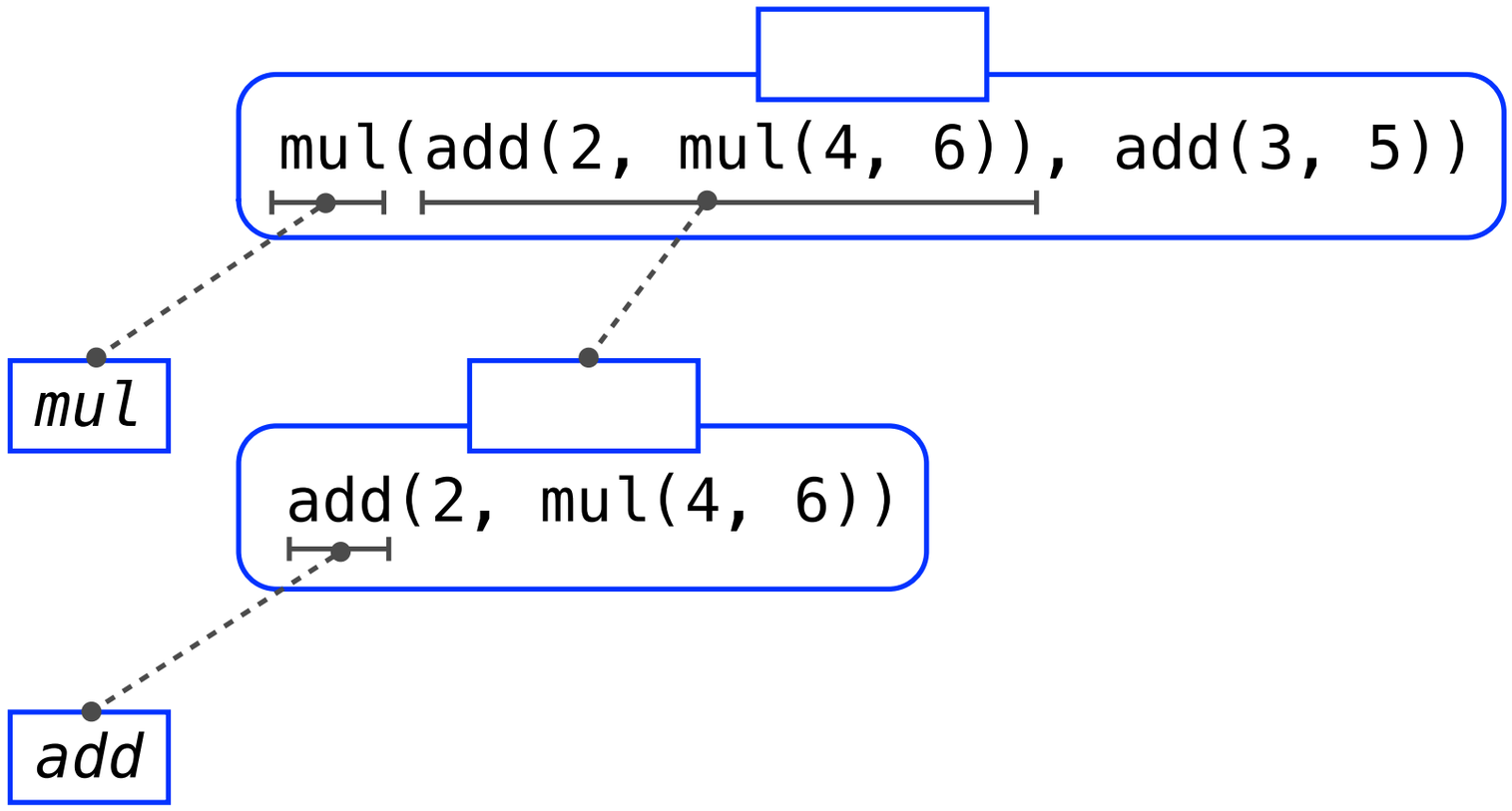
Evaluating Nested Expressions



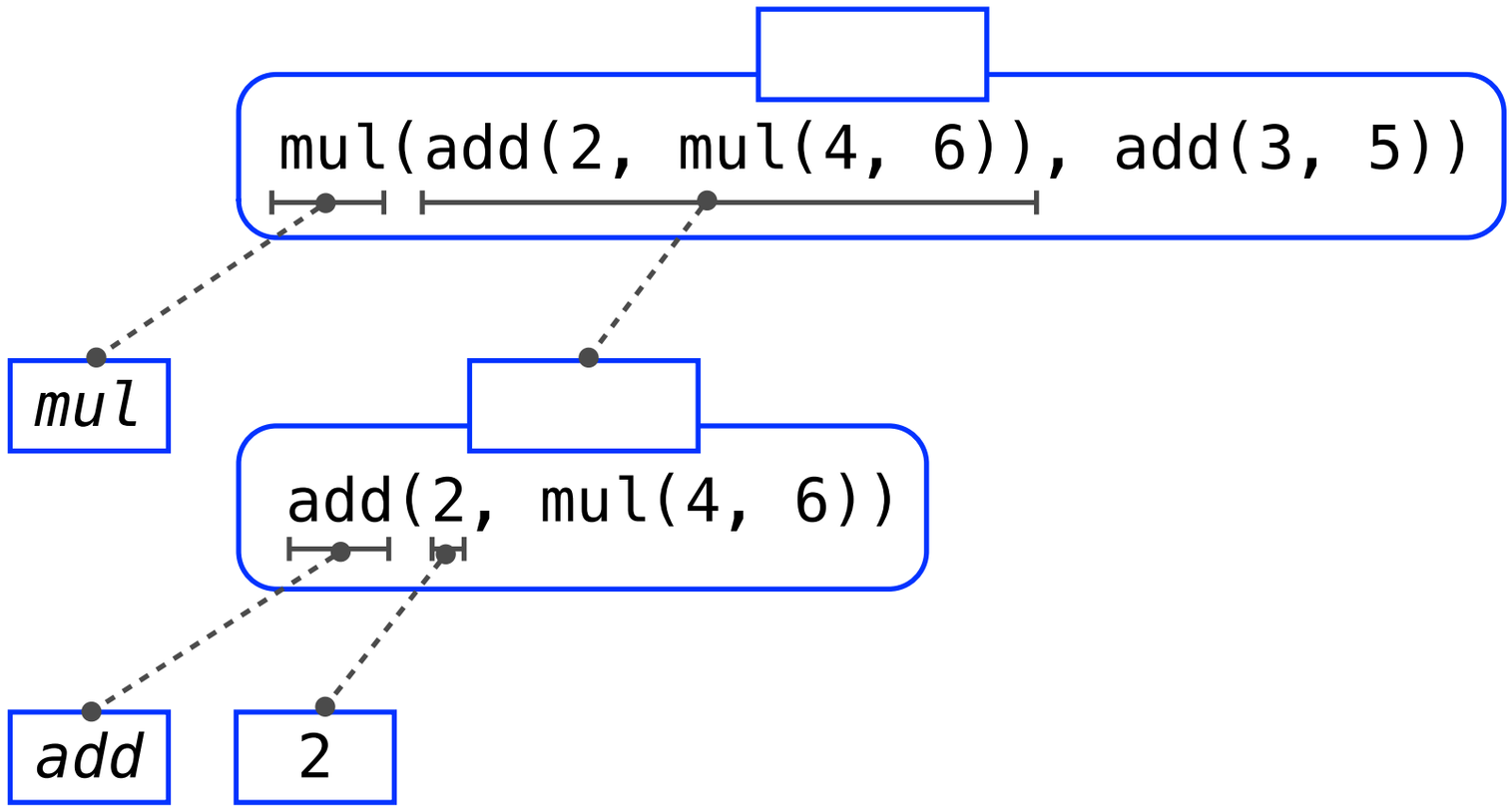
Evaluating Nested Expressions



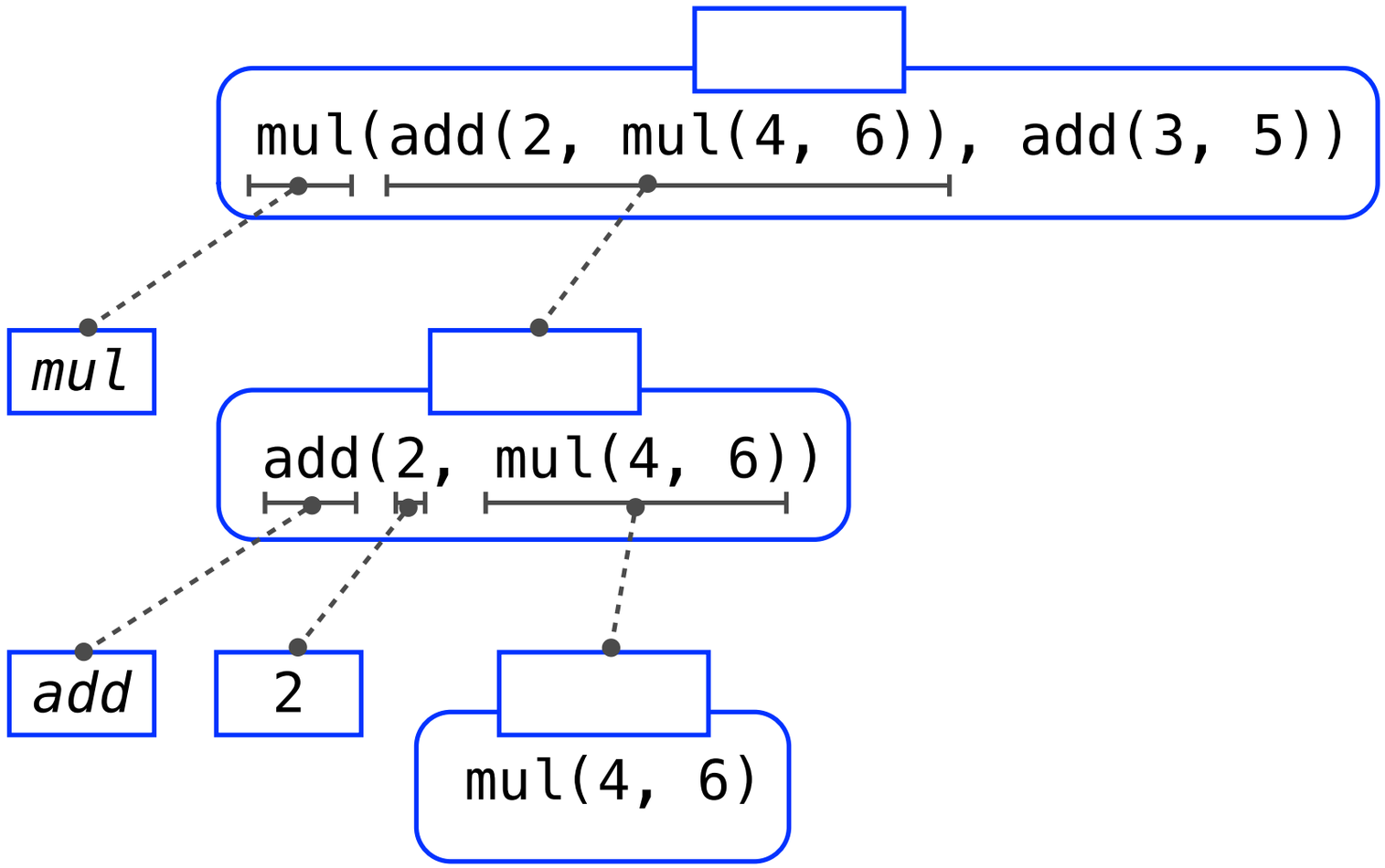
Evaluating Nested Expressions



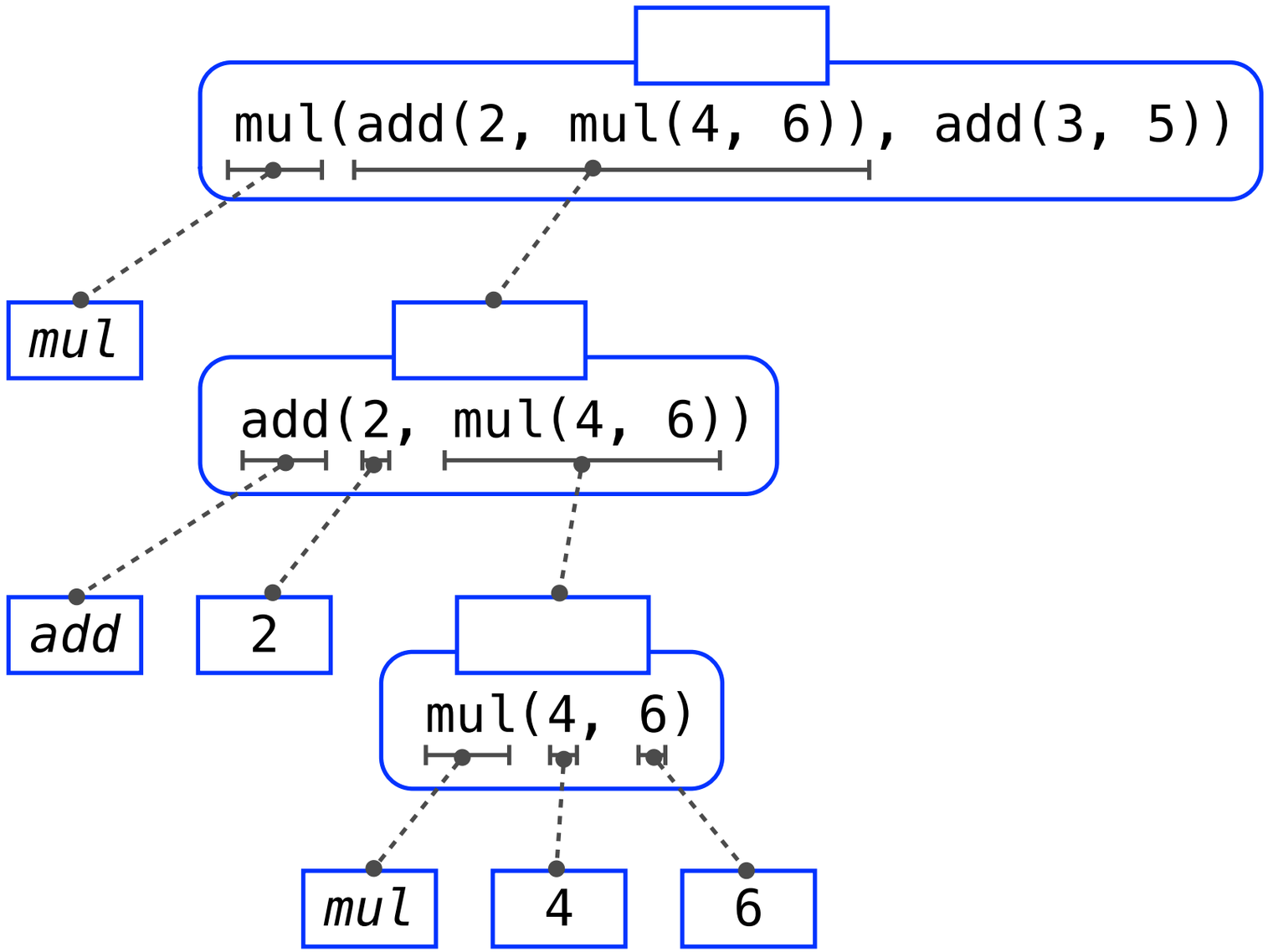
Evaluating Nested Expressions



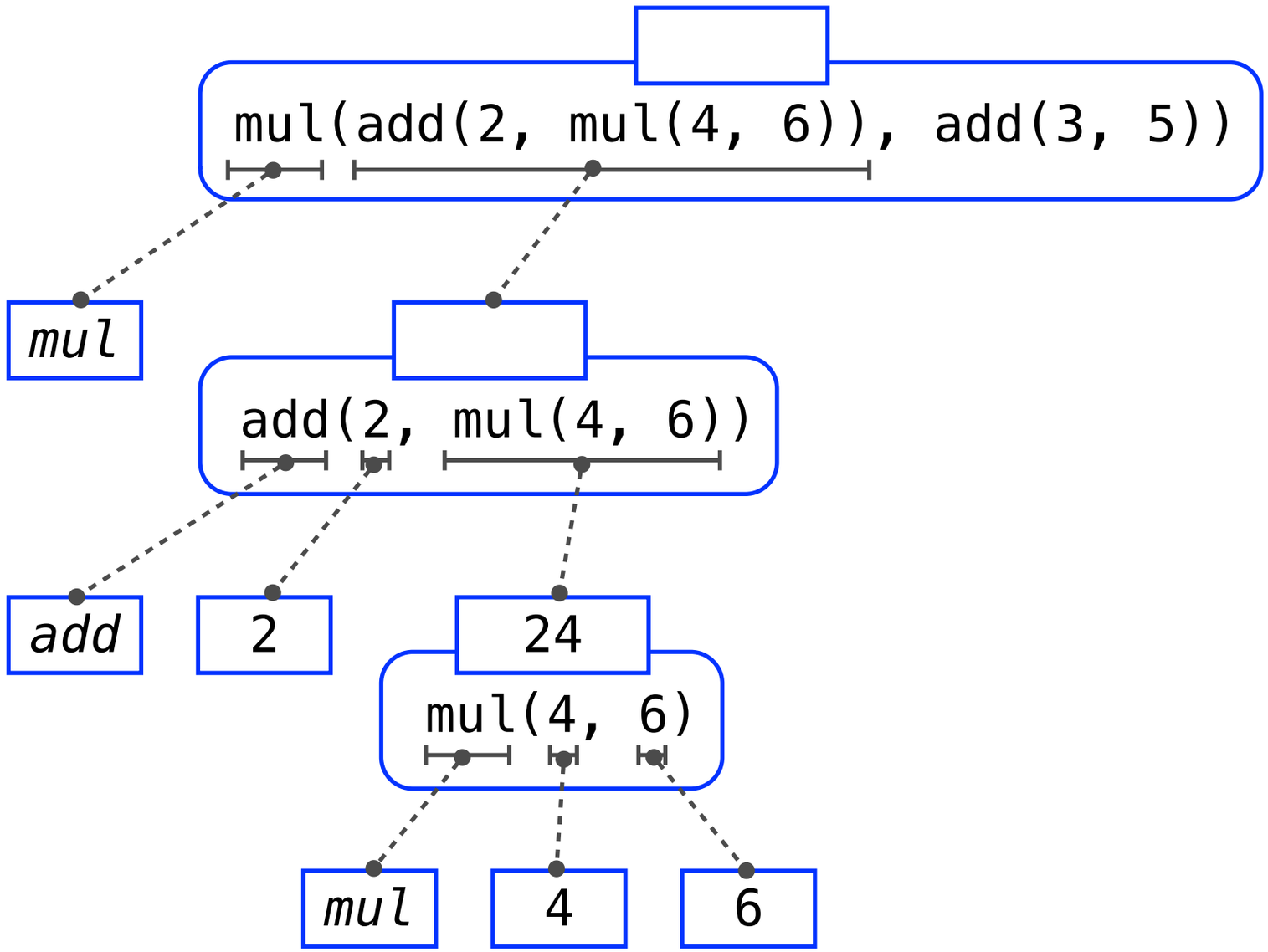
Evaluating Nested Expressions



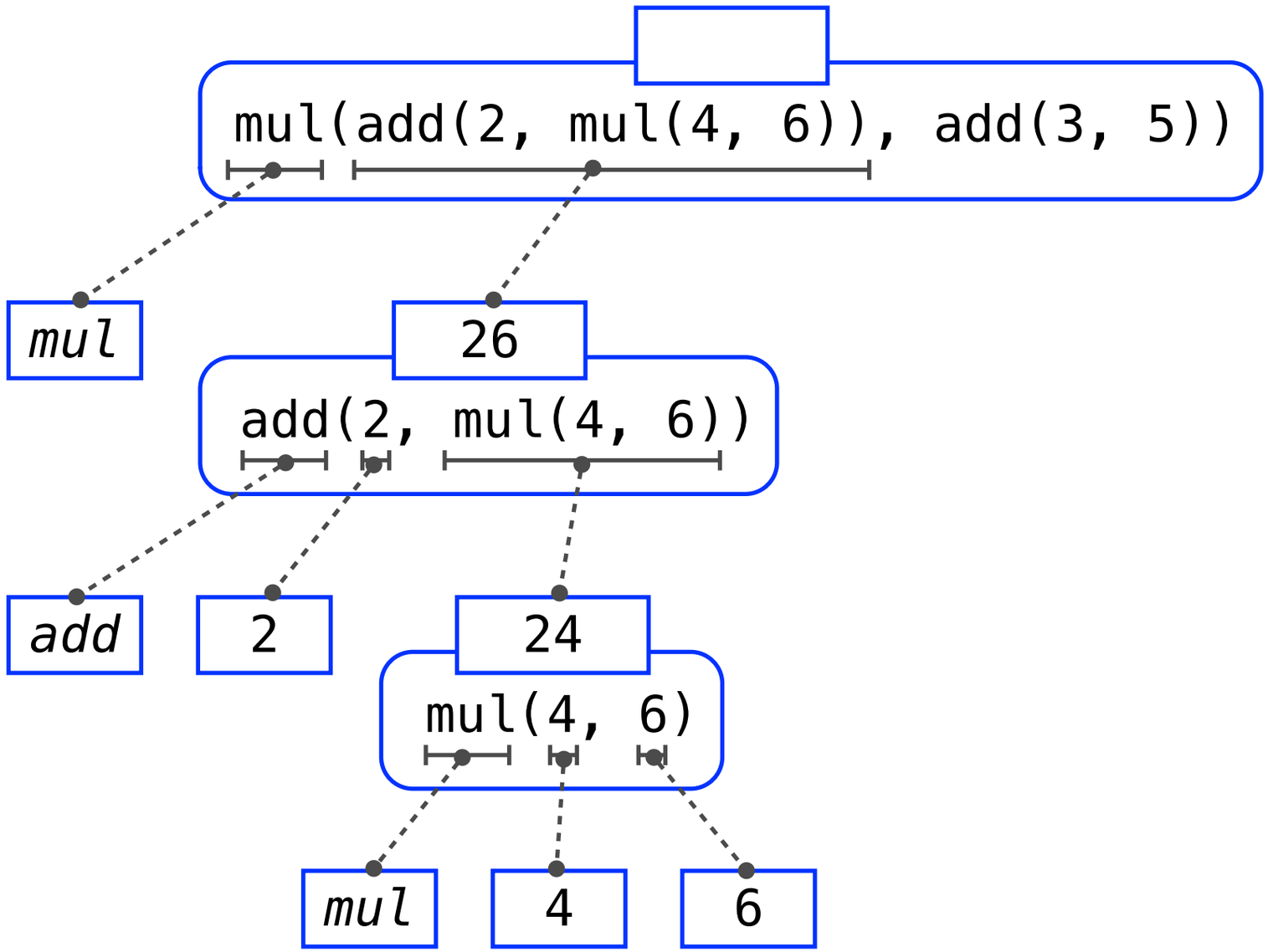
Evaluating Nested Expressions



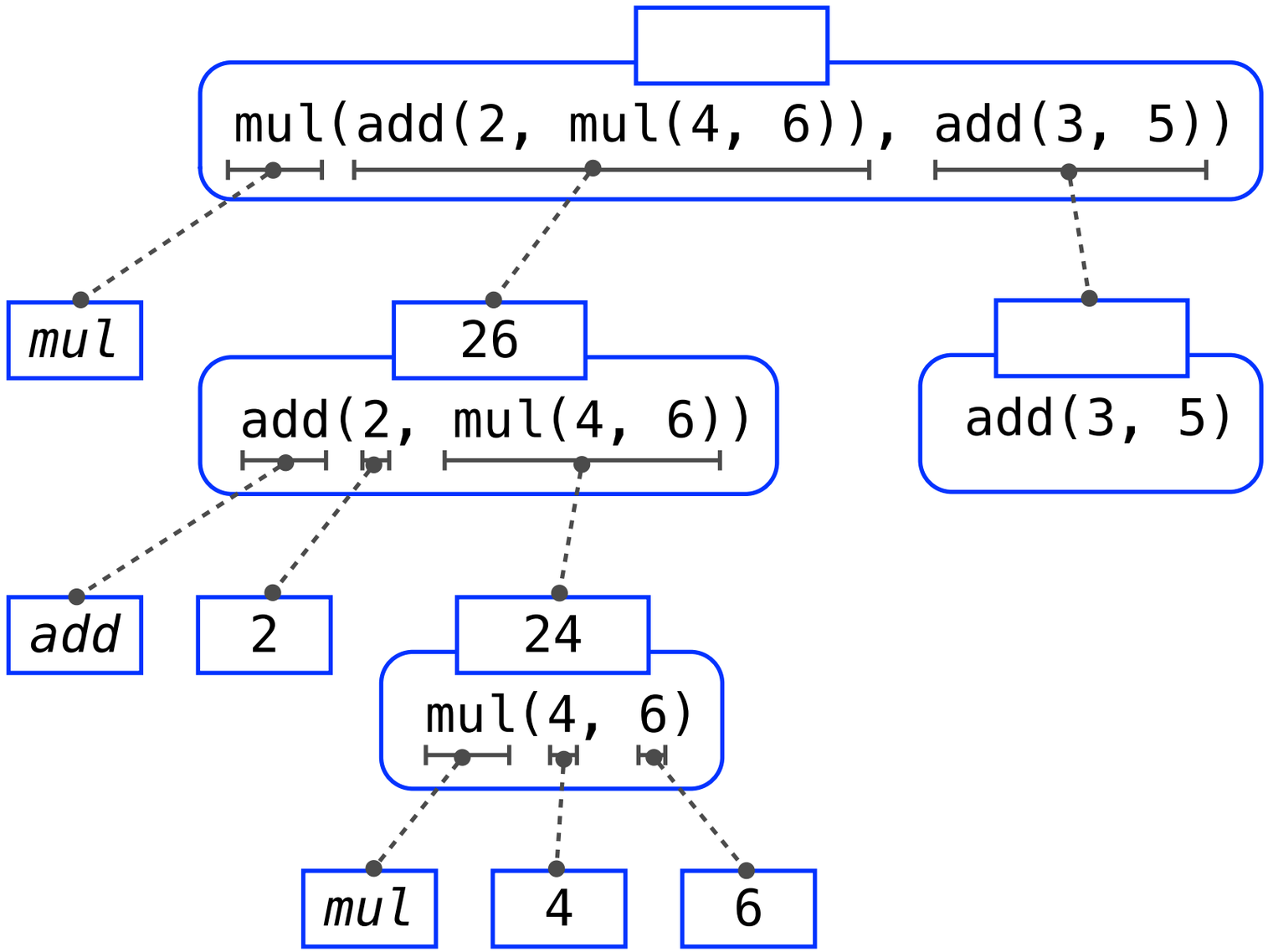
Evaluating Nested Expressions



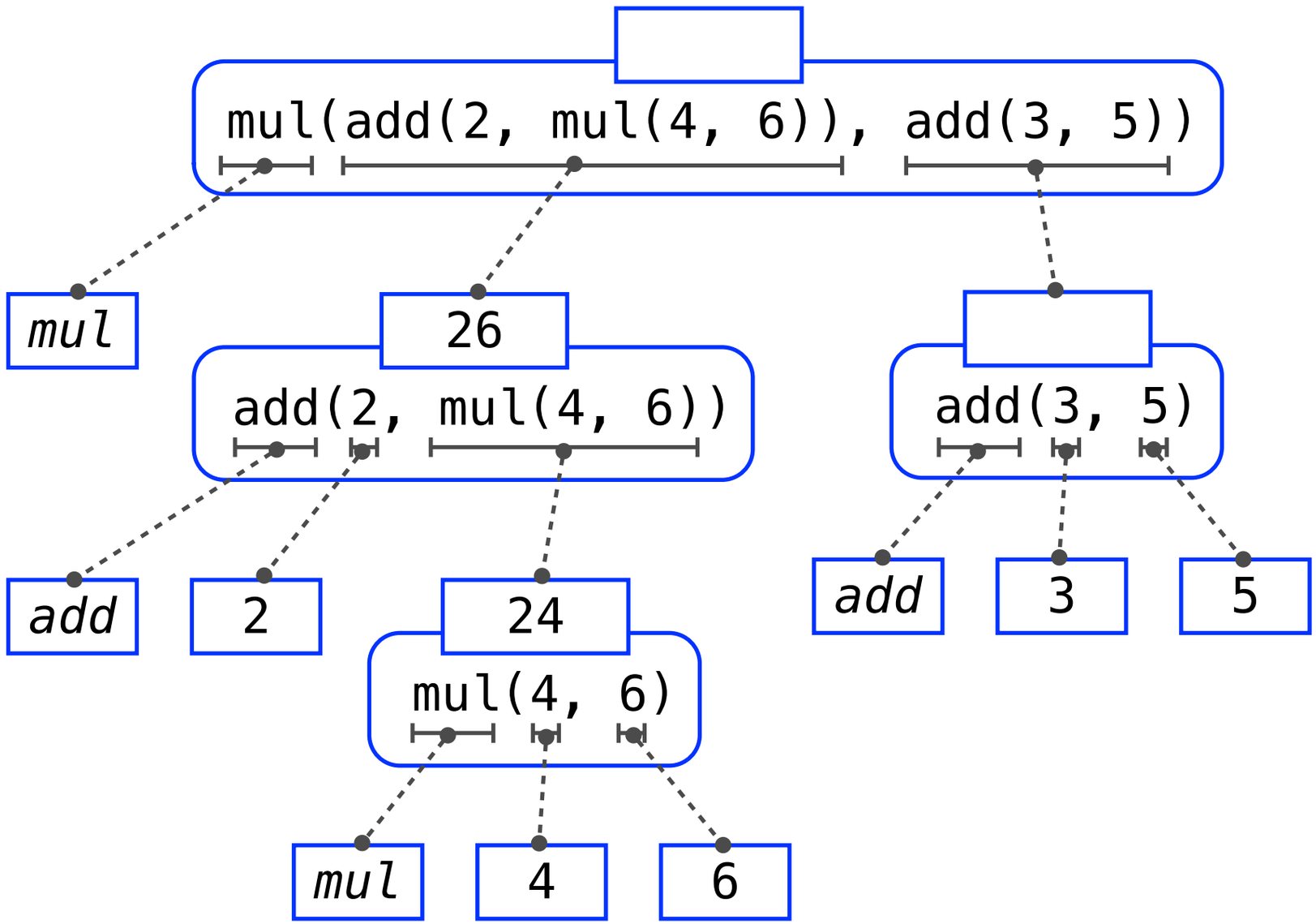
Evaluating Nested Expressions



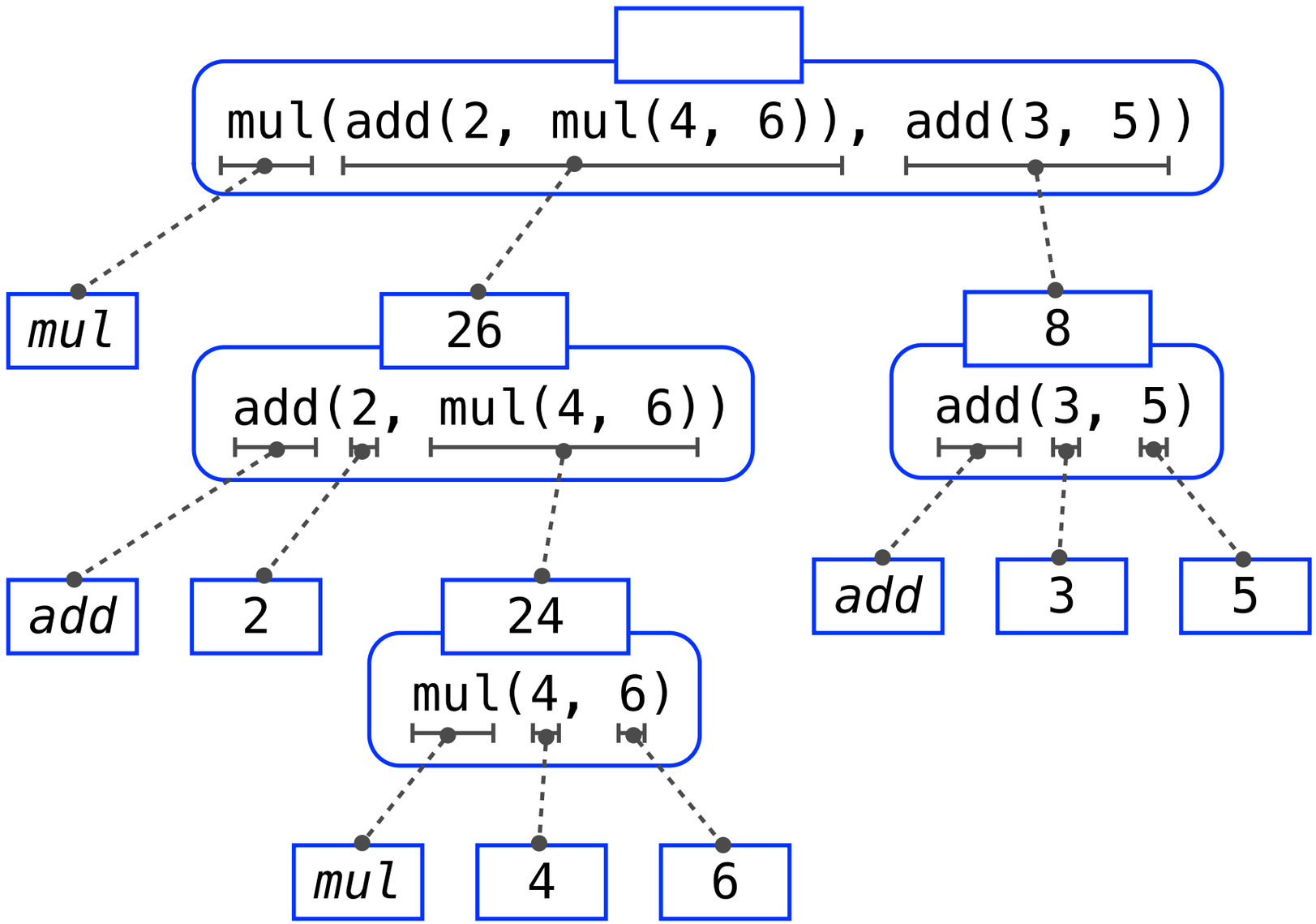
Evaluating Nested Expressions



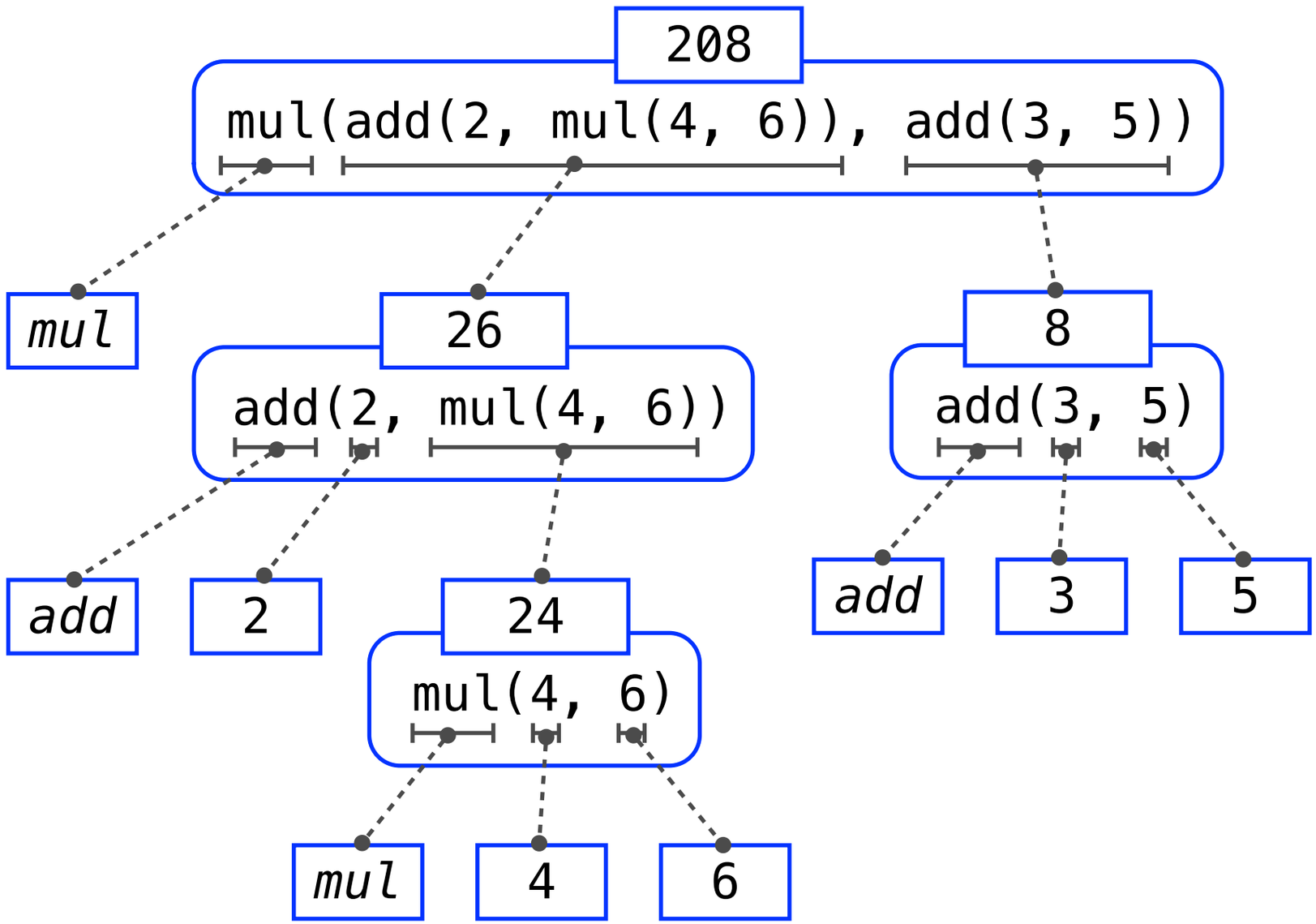
Evaluating Nested Expressions



Evaluating Nested Expressions



Evaluating Nested Expressions



The Print Function

(Demo)

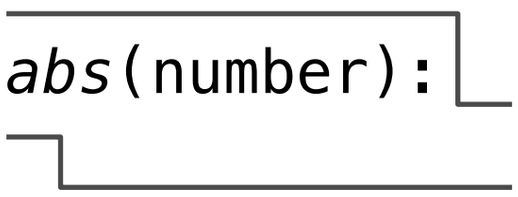
Pure Functions & Non-Pure Functions

Pure Functions

Non-Pure Functions

Pure Functions & Non-Pure Functions

Pure Functions



Non-Pure Functions

Pure Functions & Non-Pure Functions

Pure Functions

Function signature: how many parameters

abs(number):

Non-Pure Functions

Pure Functions & Non-Pure Functions

Pure Functions

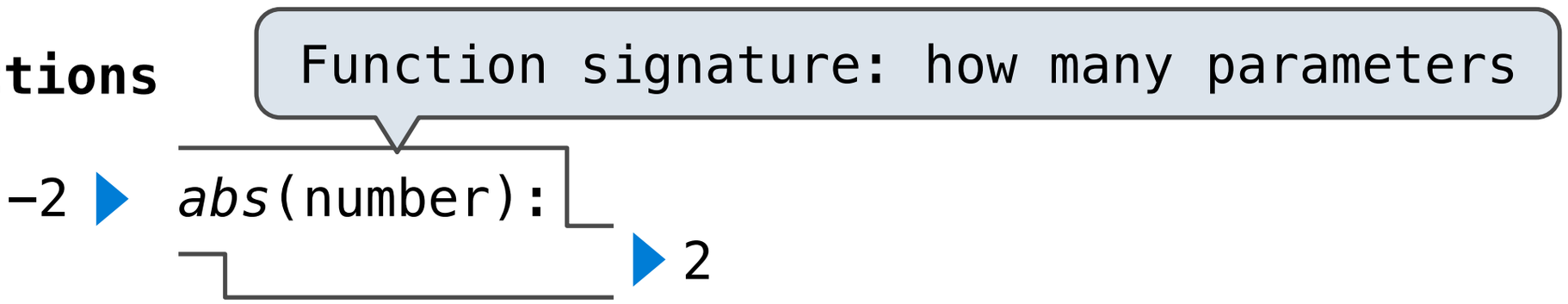
Function signature: how many parameters

```
-2 ▶ abs(number):
```

Non-Pure Functions

Pure Functions & Non-Pure Functions

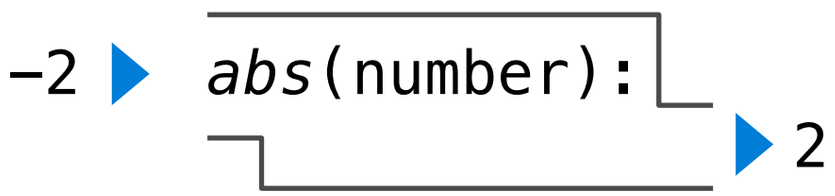
Pure Functions



Non-Pure Functions

Pure Functions & Non-Pure Functions

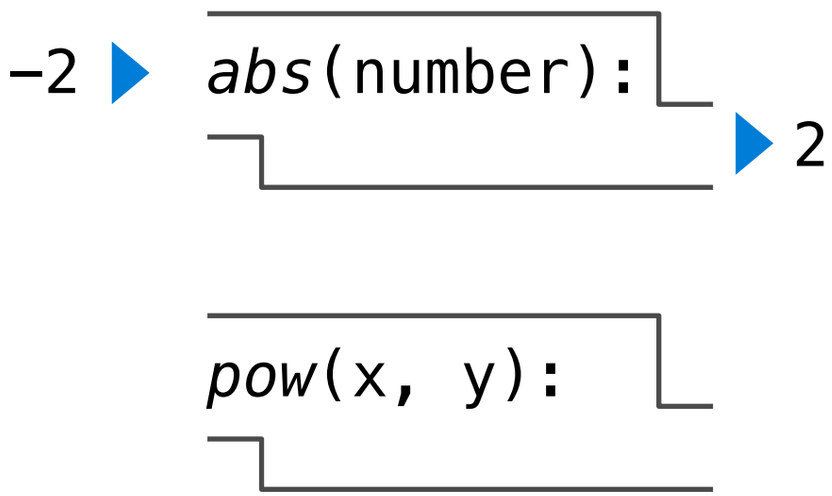
Pure Functions



Non-Pure Functions

Pure Functions & Non-Pure Functions

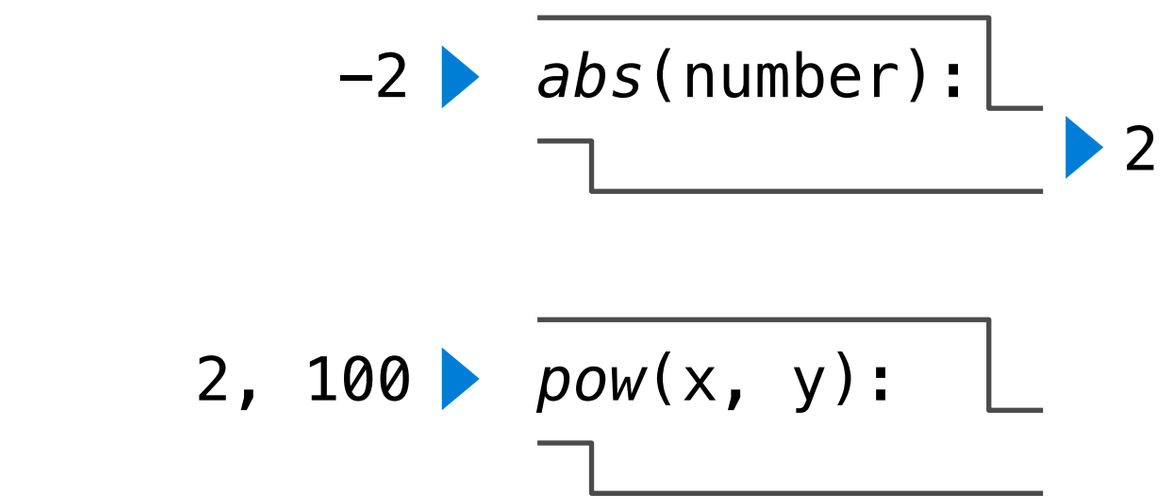
Pure Functions



Non-Pure Functions

Pure Functions & Non-Pure Functions

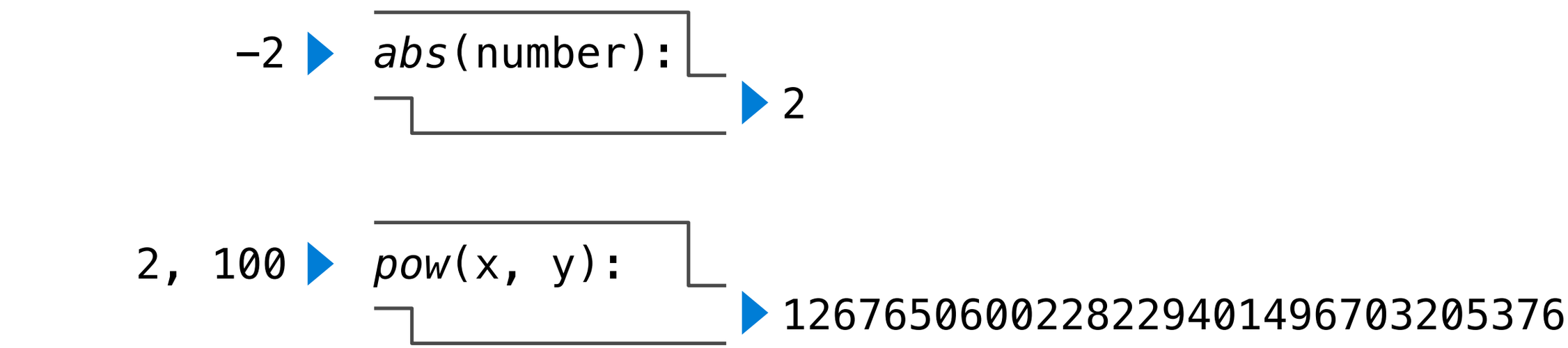
Pure Functions



Non-Pure Functions

Pure Functions & Non-Pure Functions

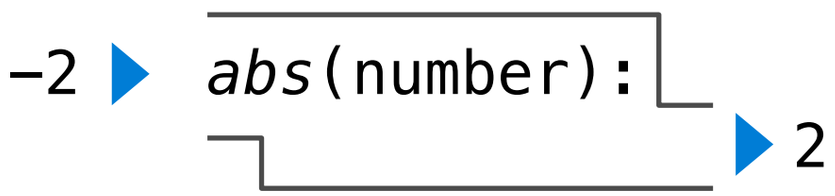
Pure Functions



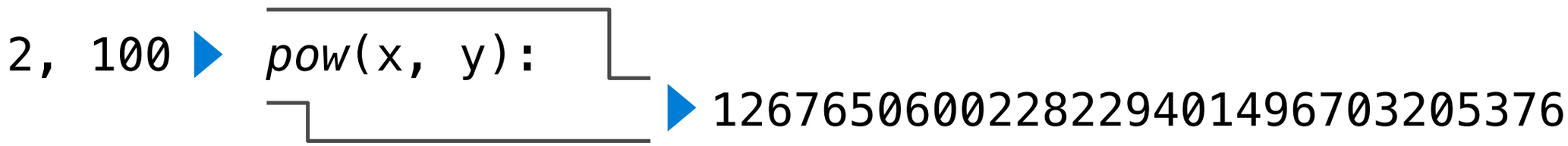
Non-Pure Functions

Pure Functions & Non-Pure Functions

Pure Functions



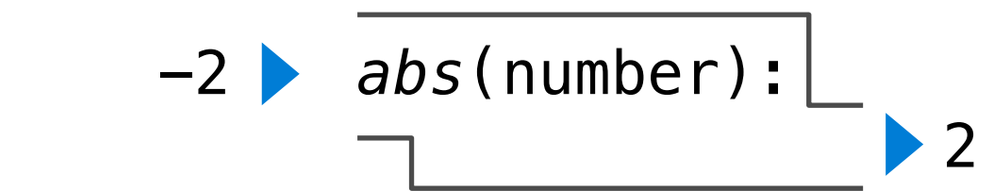
Only produces return values



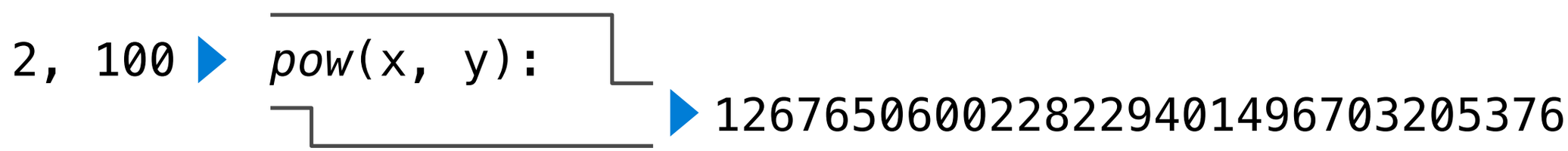
Non-Pure Functions

Pure Functions & Non-Pure Functions

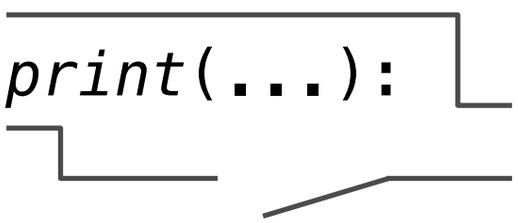
Pure Functions



Only produces return values

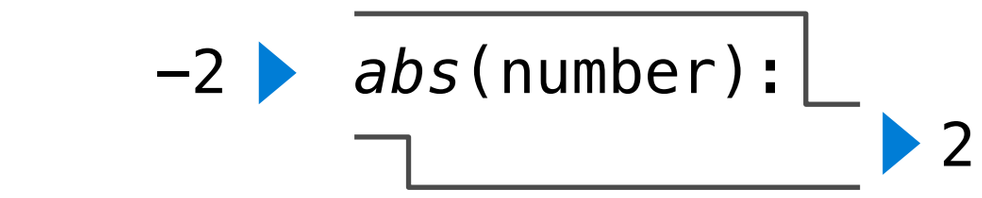


Non-Pure Functions

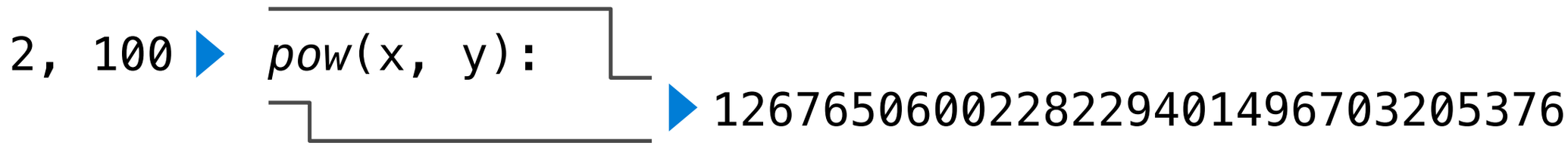


Pure Functions & Non-Pure Functions

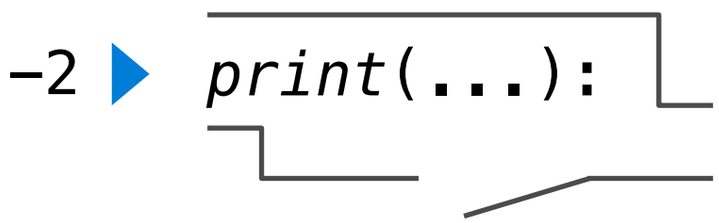
Pure Functions



Only produces return values

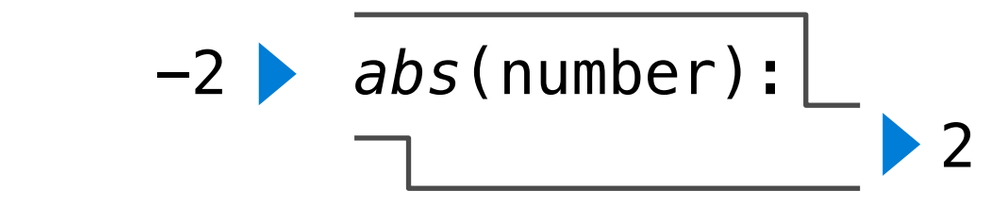


Non-Pure Functions

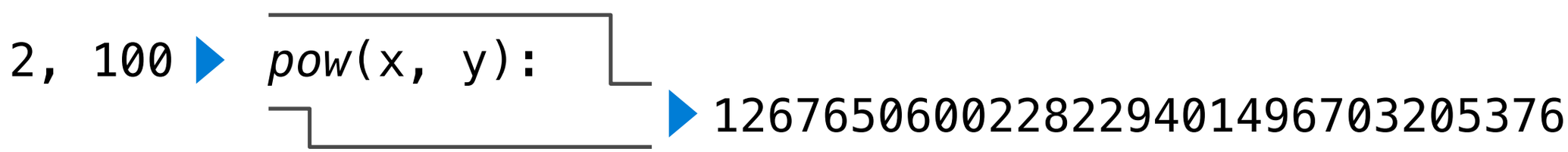


Pure Functions & Non-Pure Functions

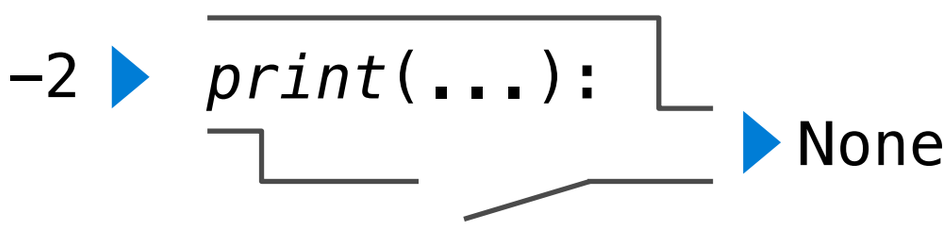
Pure Functions



Only produces return values

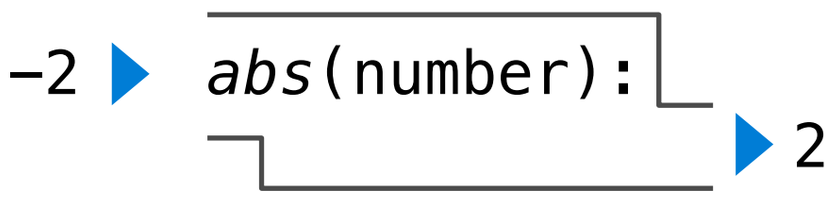


Non-Pure Functions

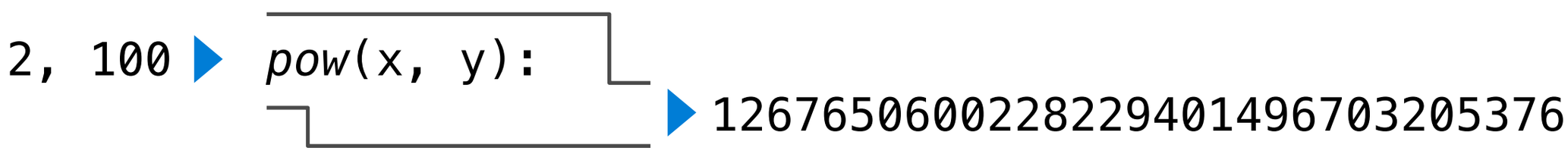


Pure Functions & Non-Pure Functions

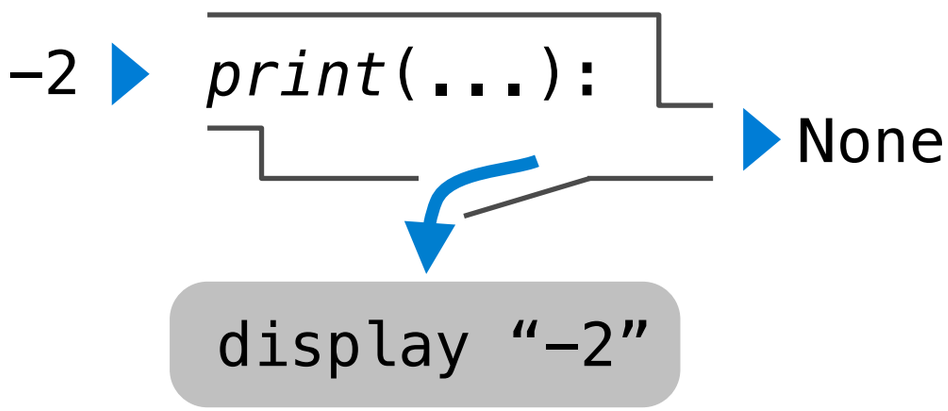
Pure Functions



Only produces return values

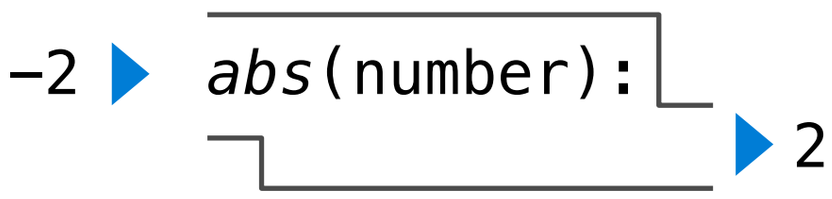


Non-Pure Functions

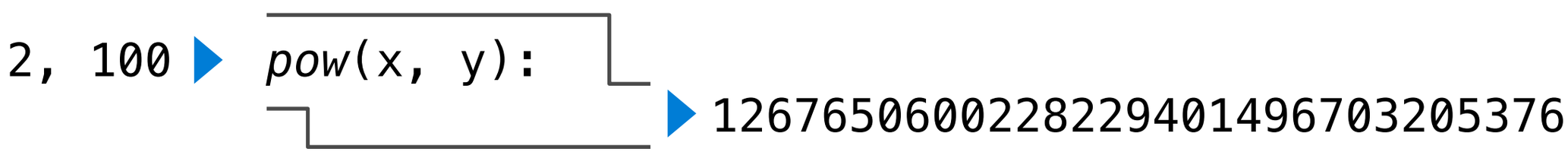


Pure Functions & Non-Pure Functions

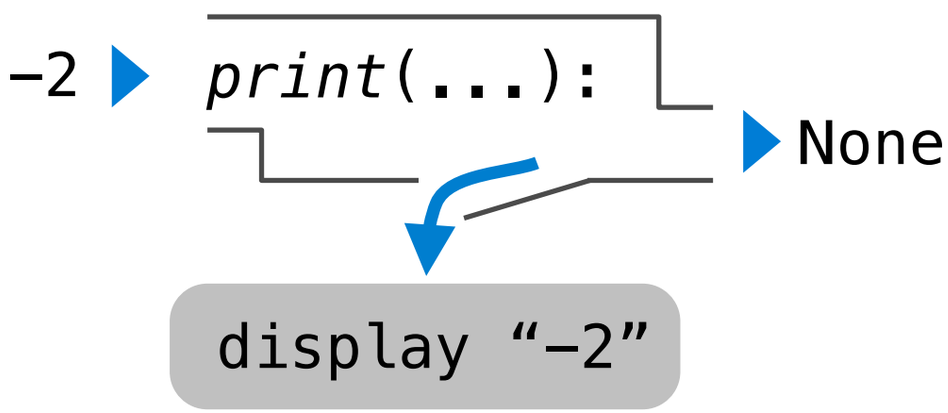
Pure Functions



Only produces return values



Non-Pure Functions

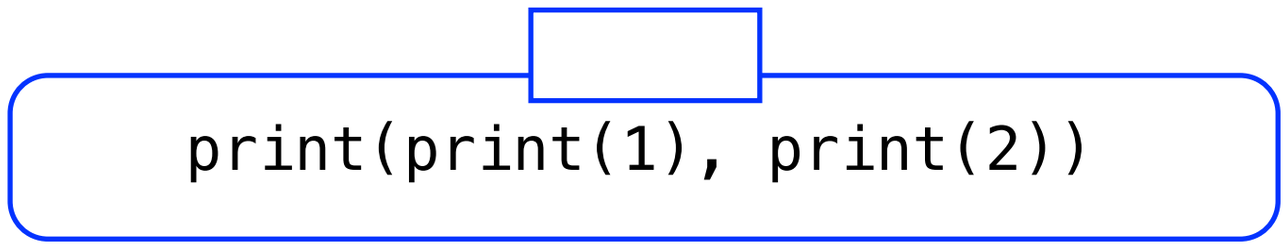


May create side effects

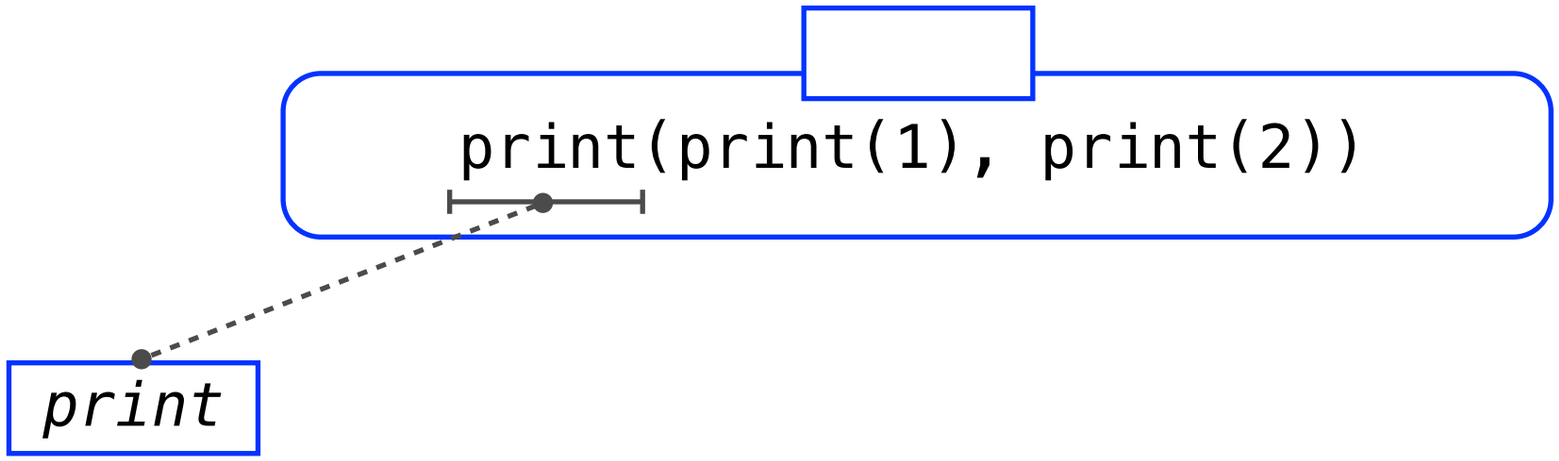
Nested Expressions with Print

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

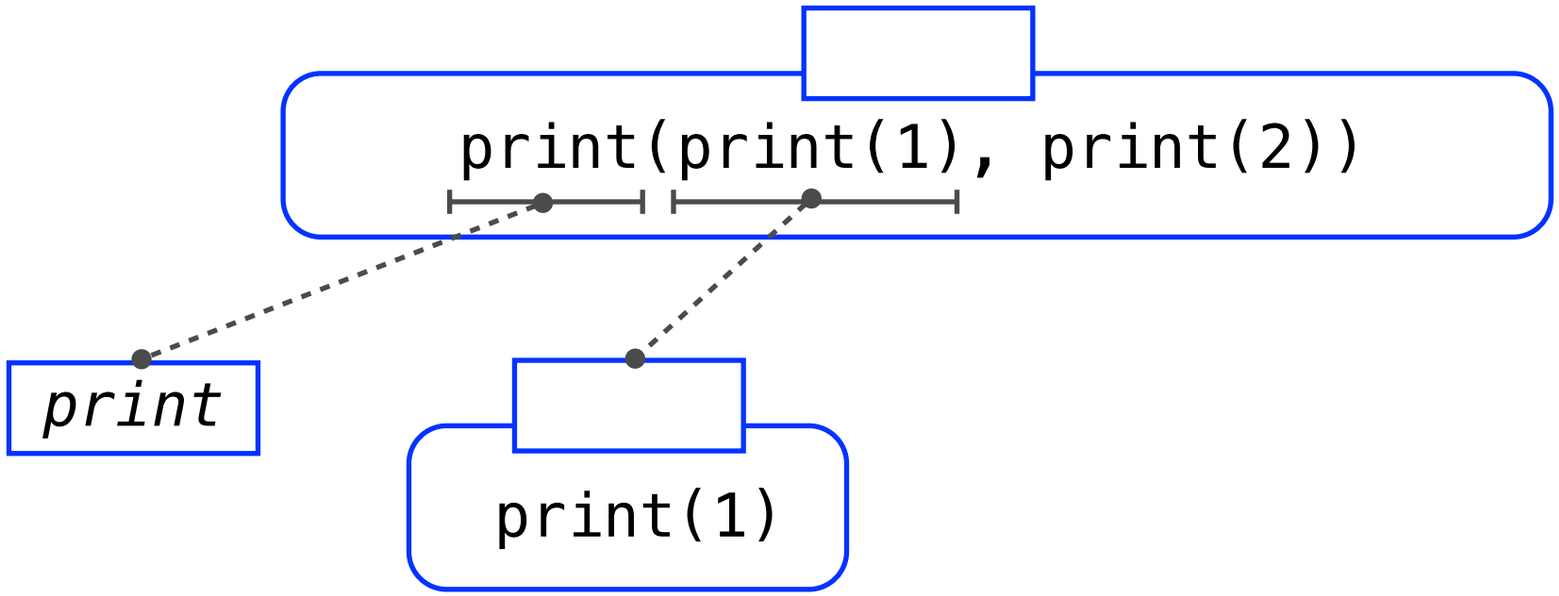
Nested Expressions with Print



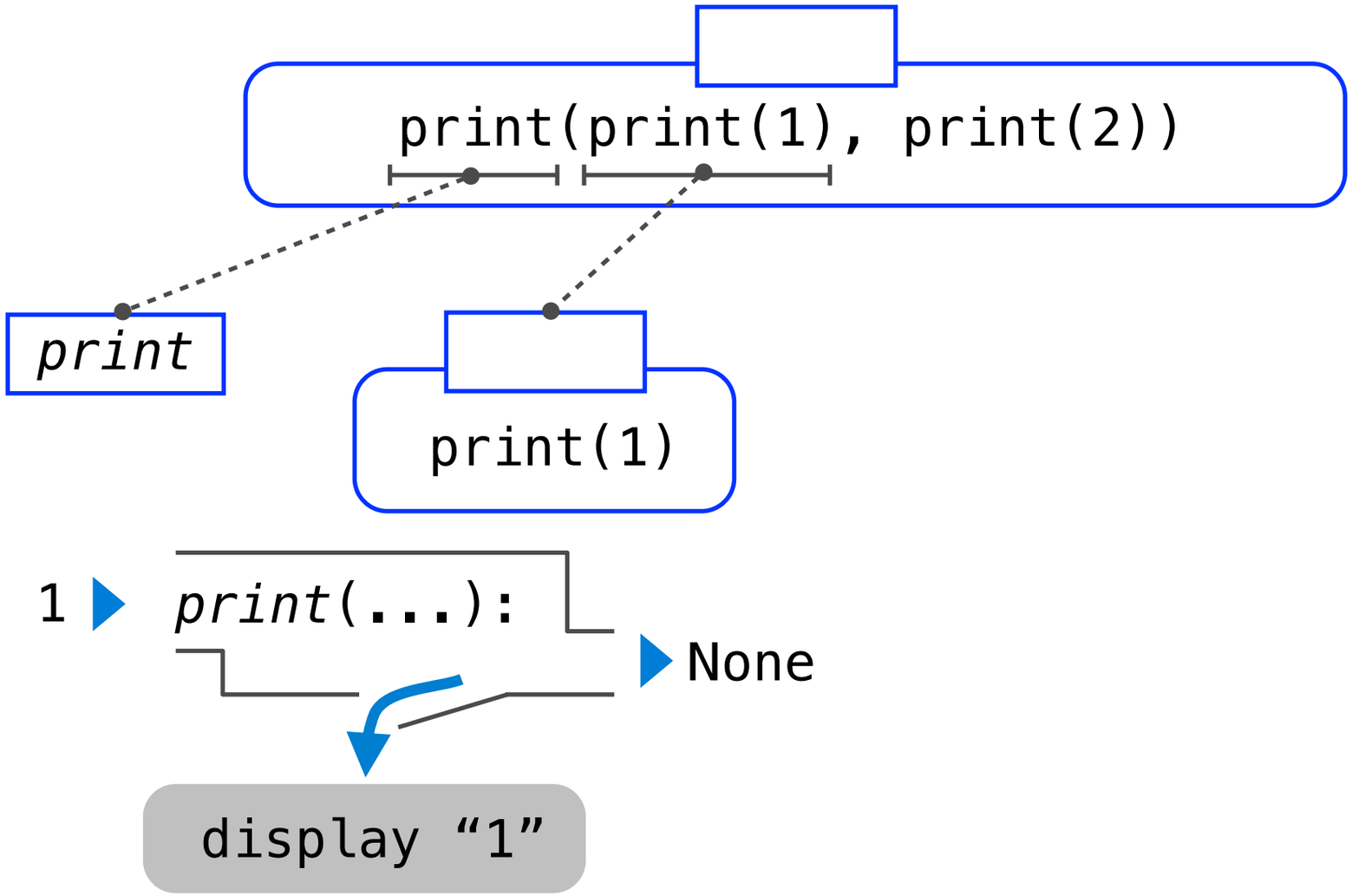
Nested Expressions with Print



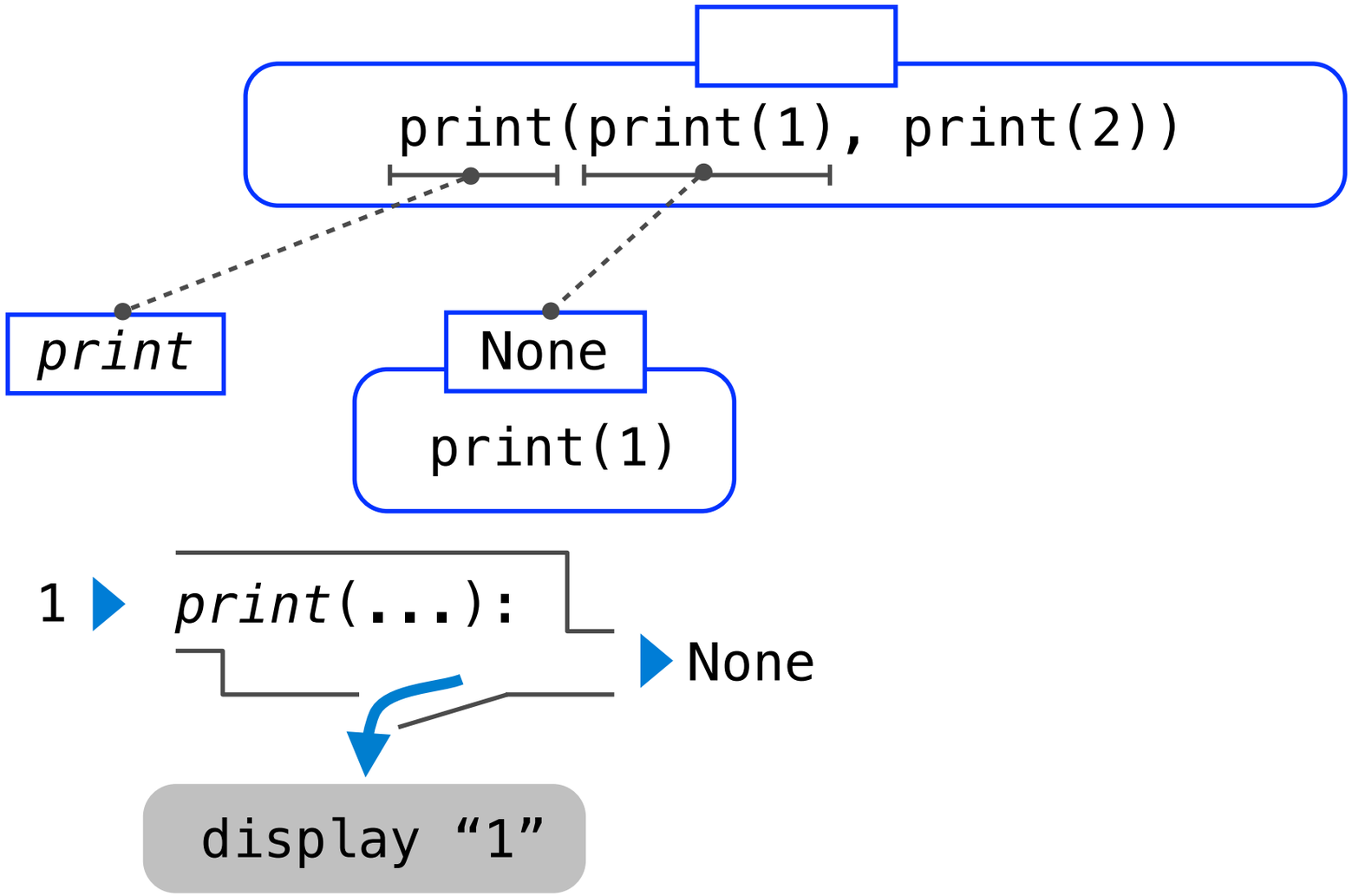
Nested Expressions with Print



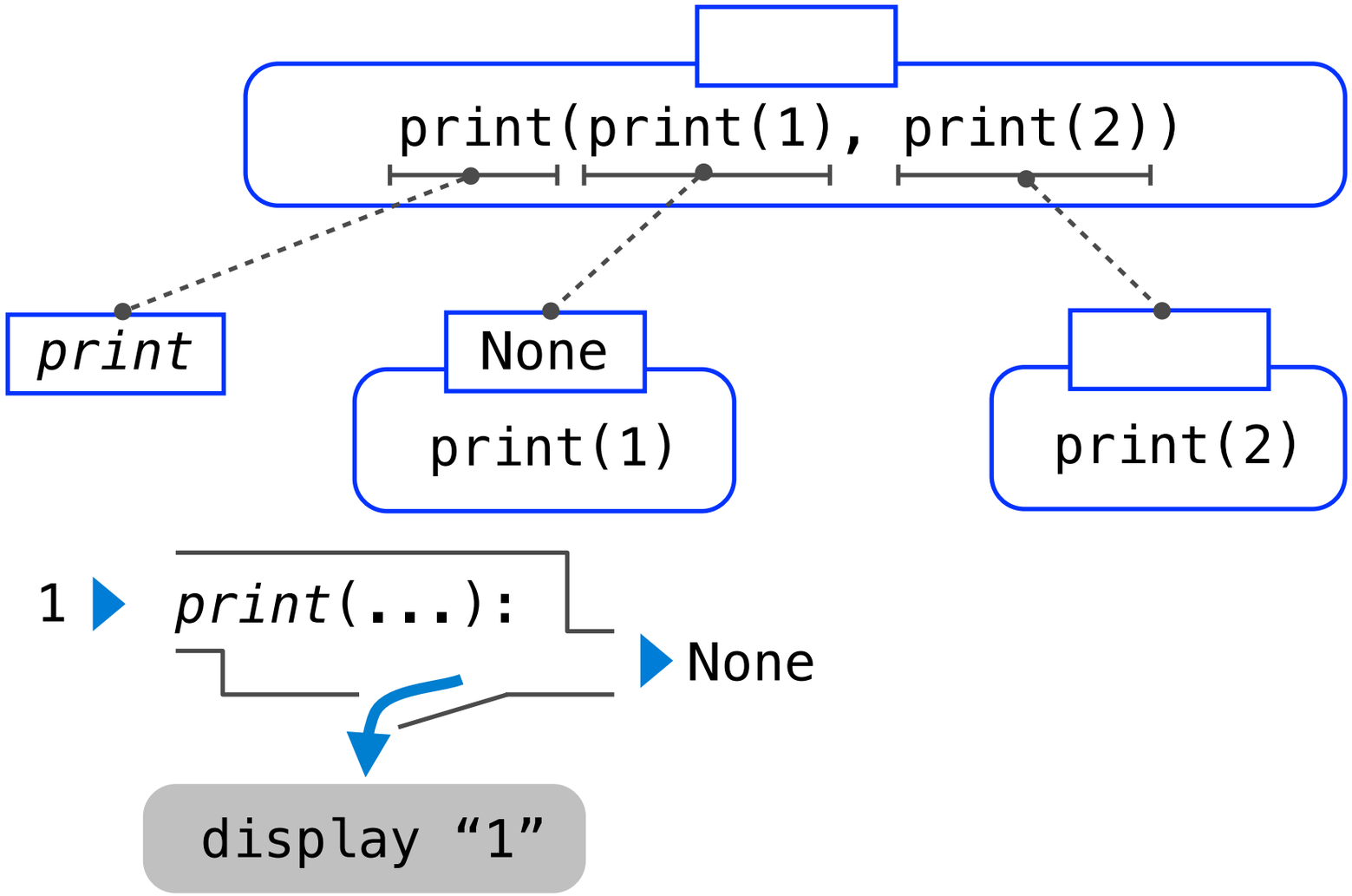
Nested Expressions with Print



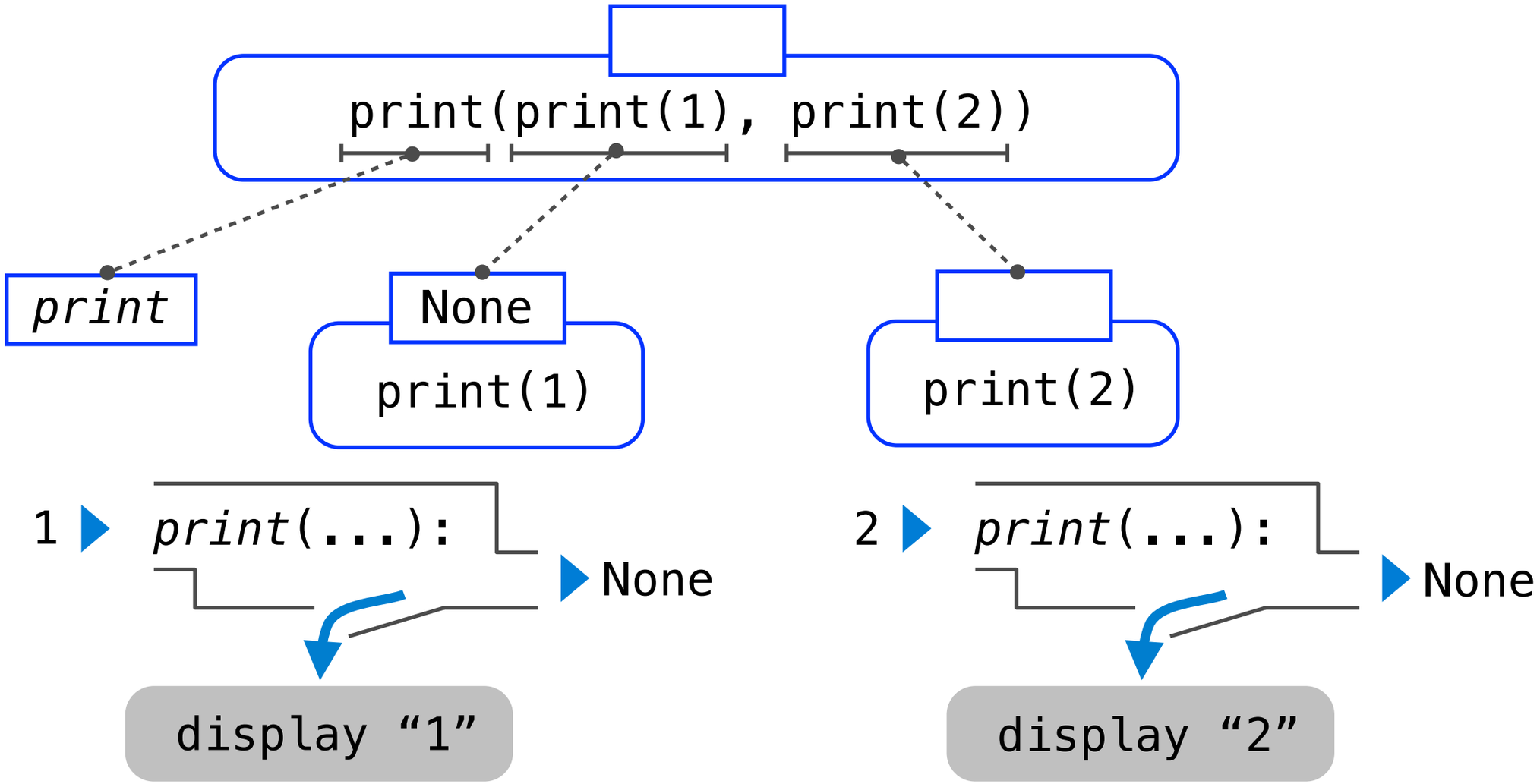
Nested Expressions with Print



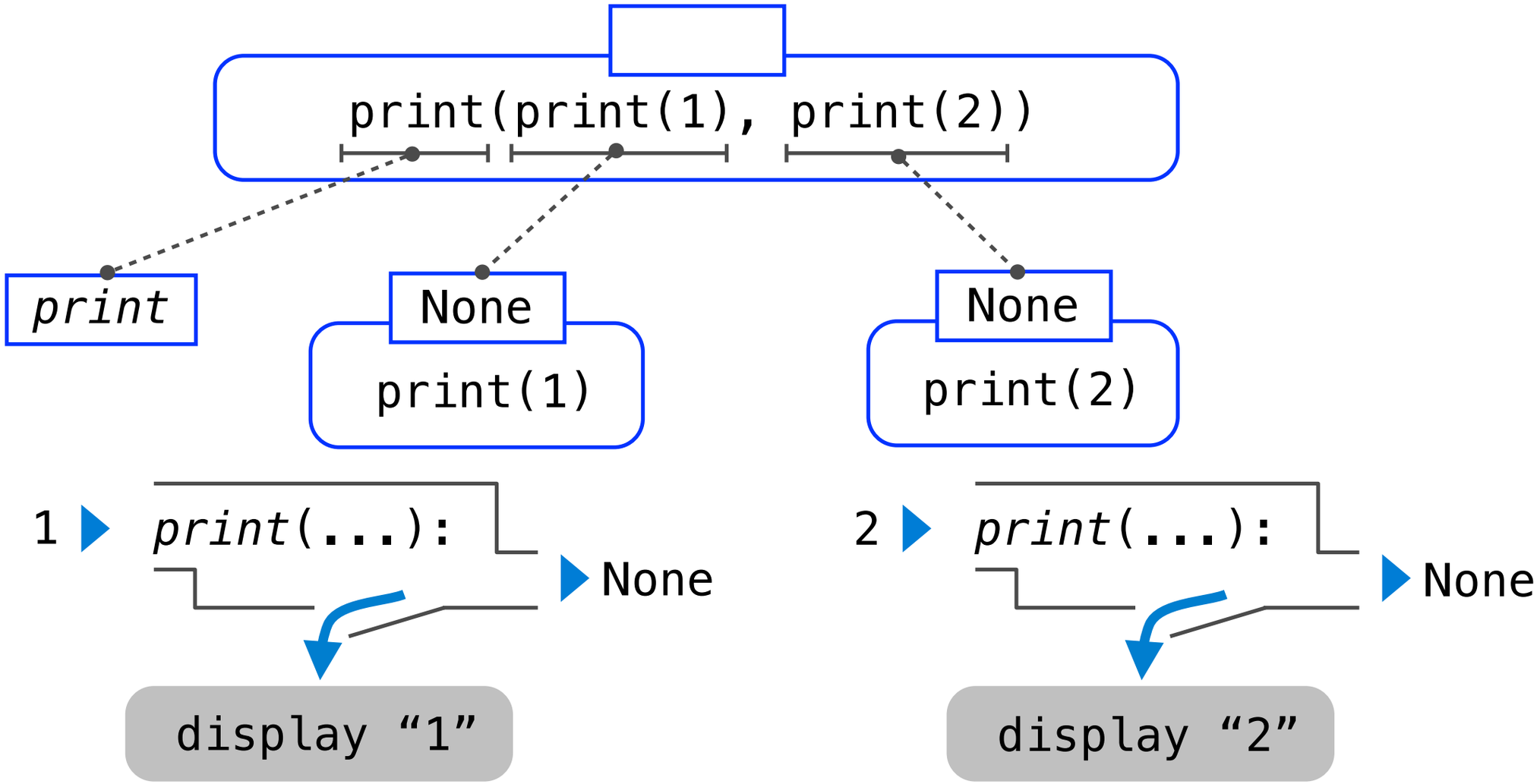
Nested Expressions with Print



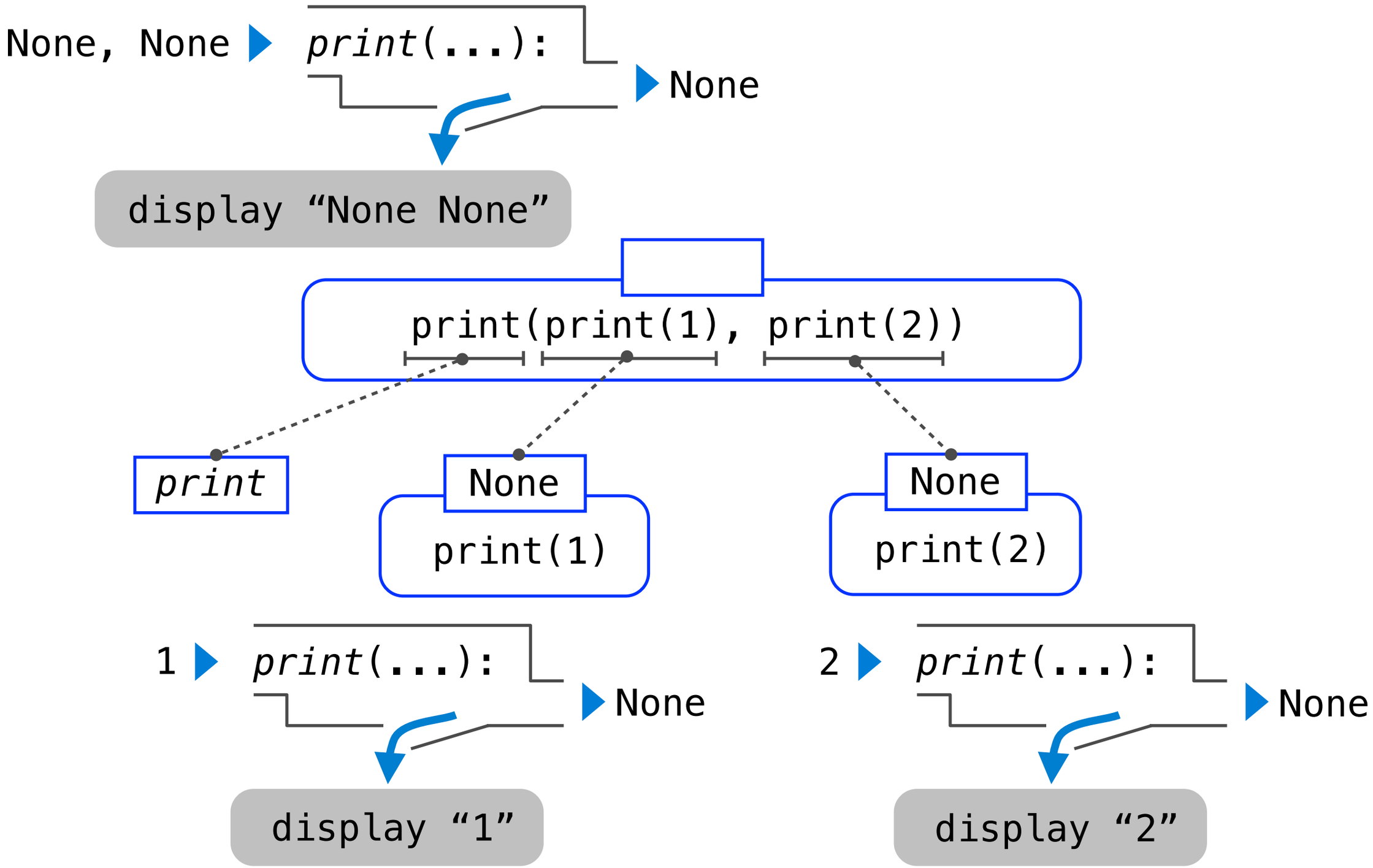
Nested Expressions with Print



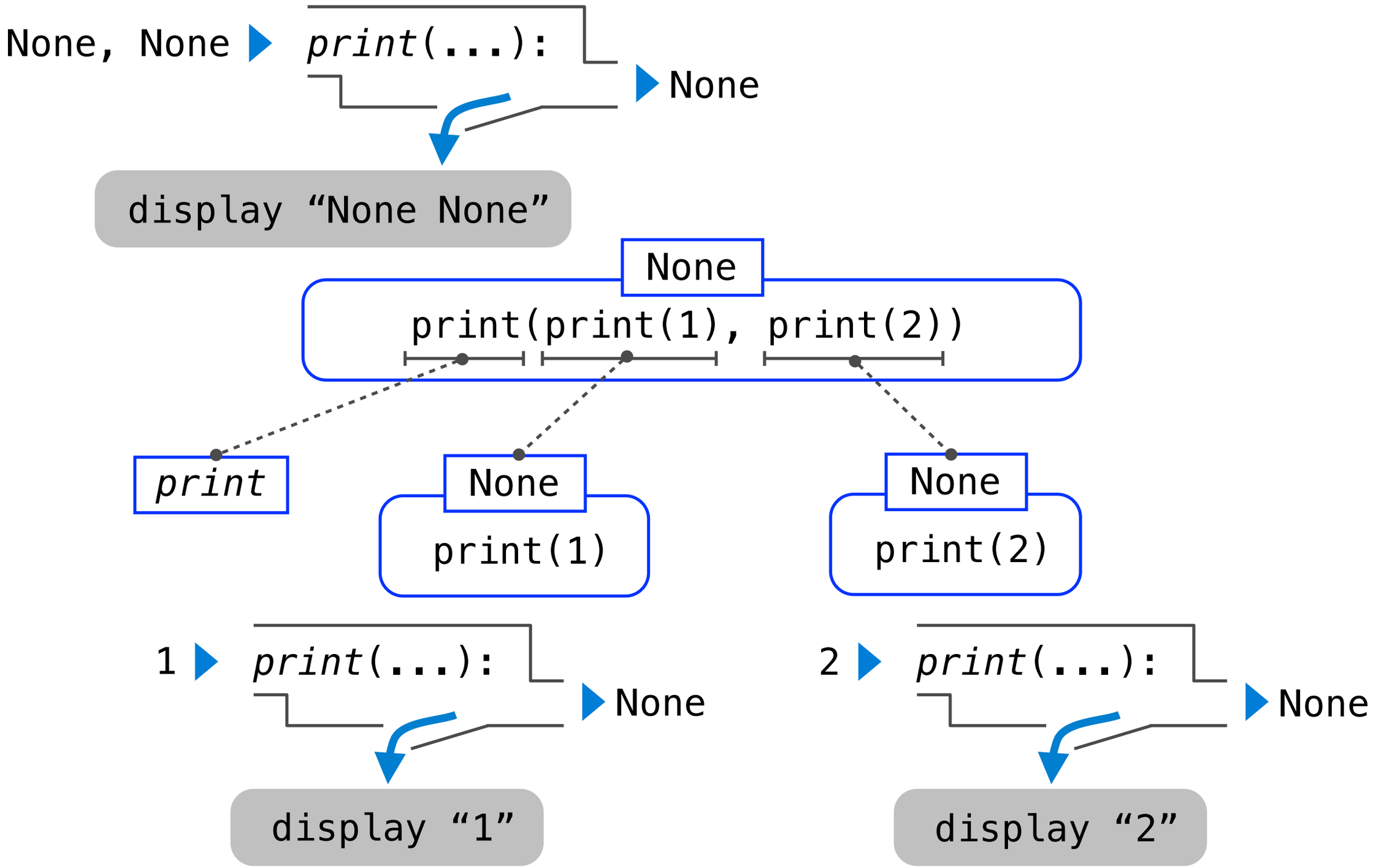
Nested Expressions with Print



Nested Expressions with Print



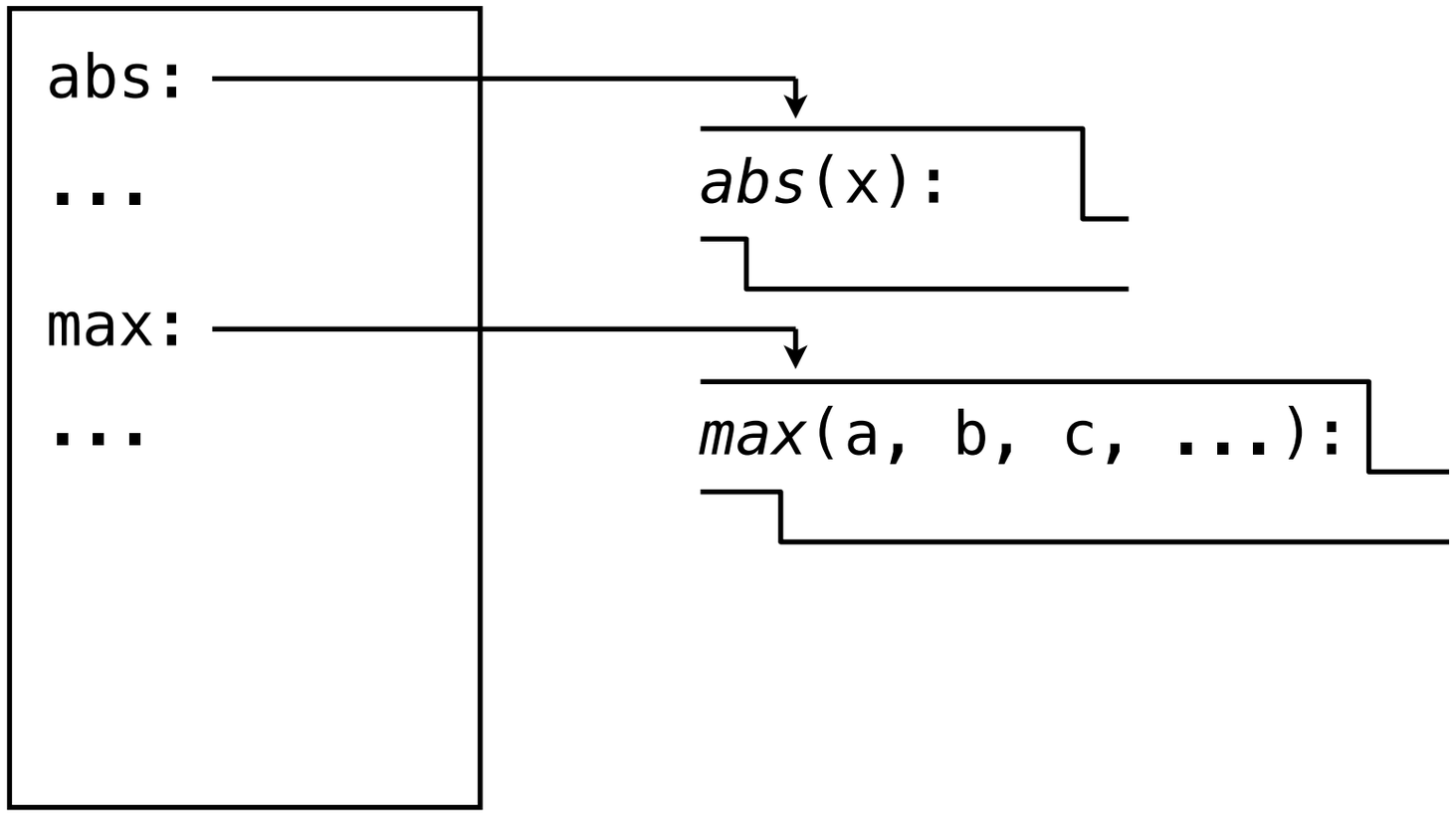
Nested Expressions with Print



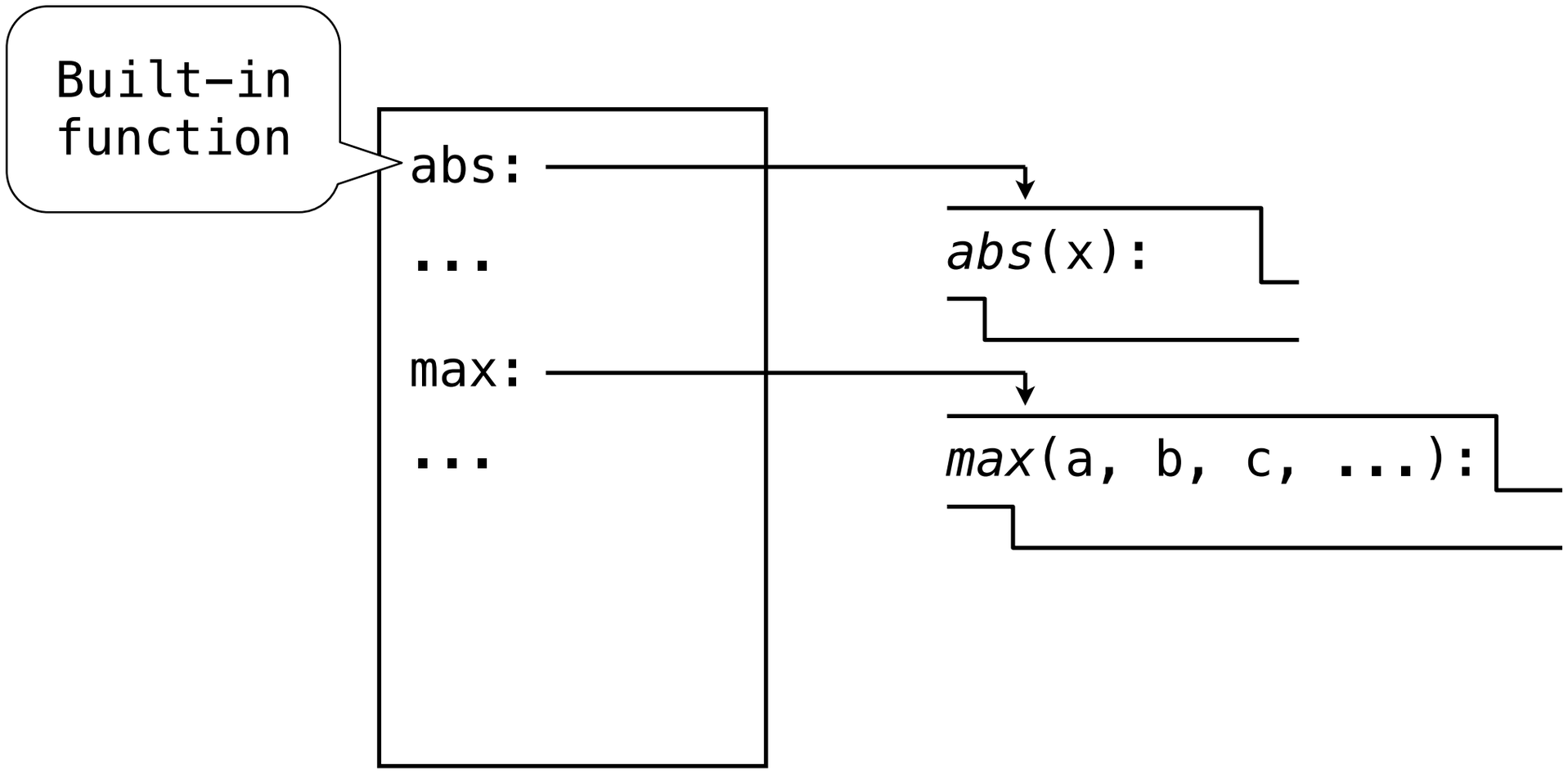
Names and Assignment

(Demo)

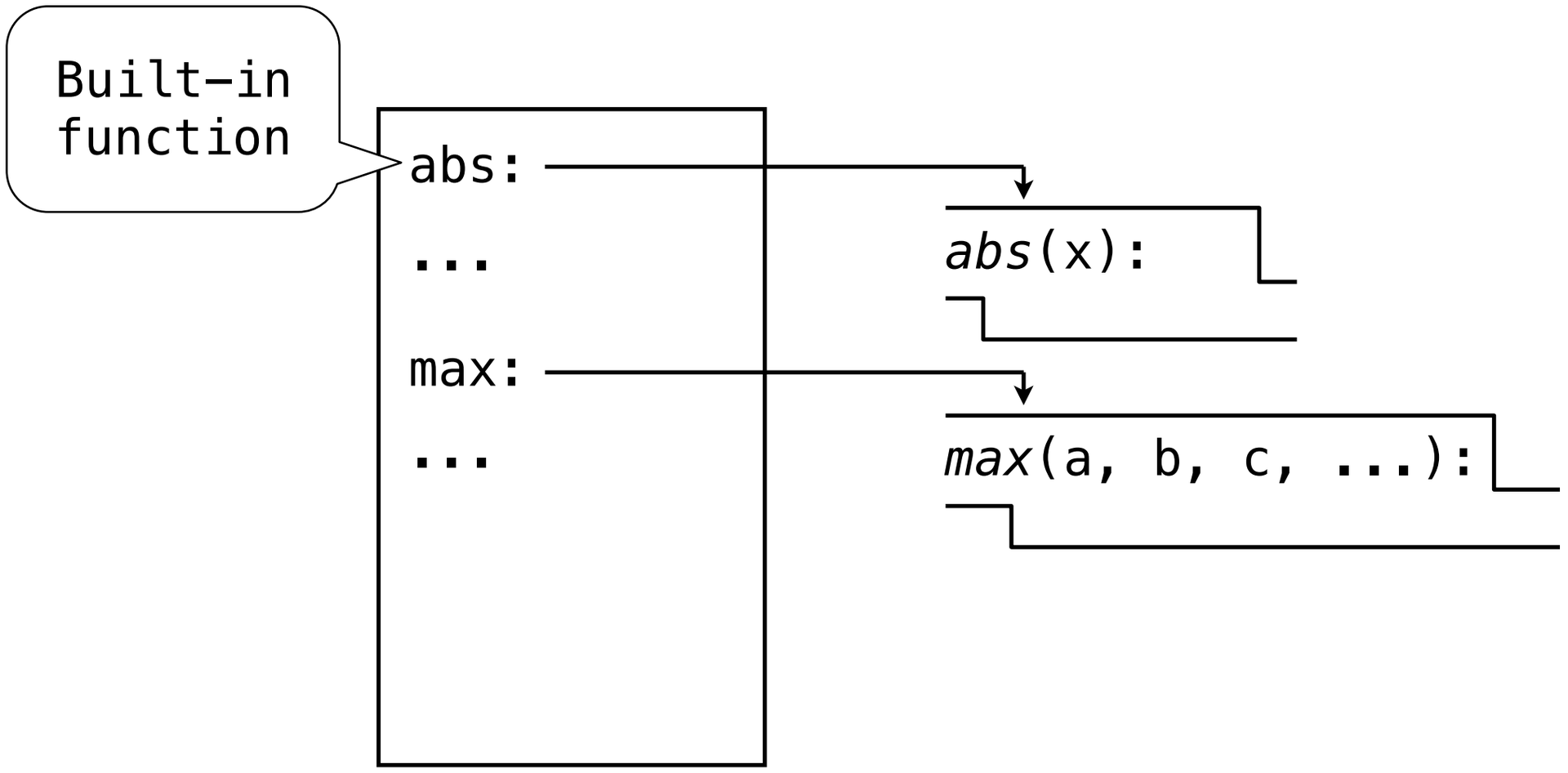
Environments



Environments

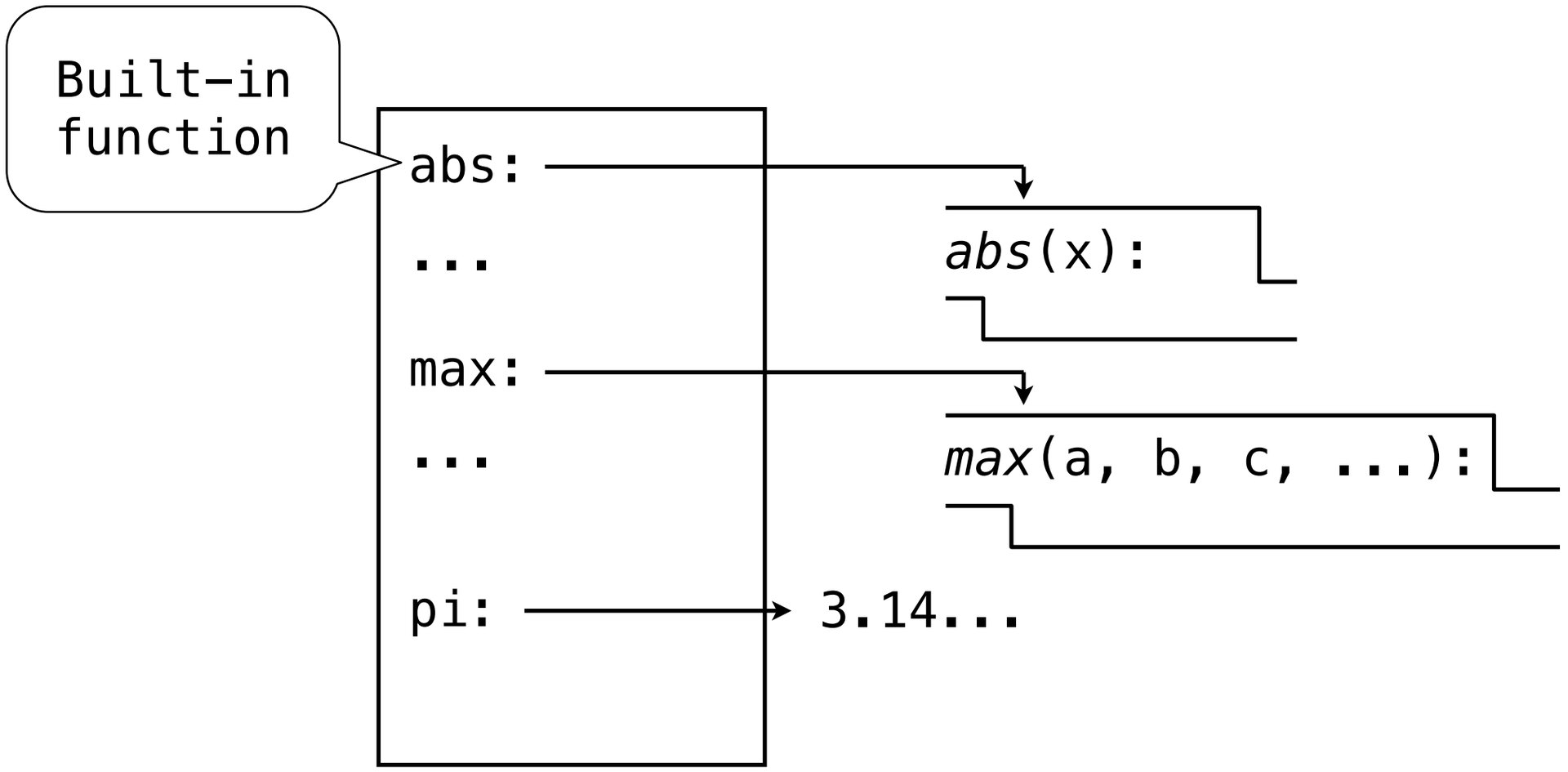


Environments



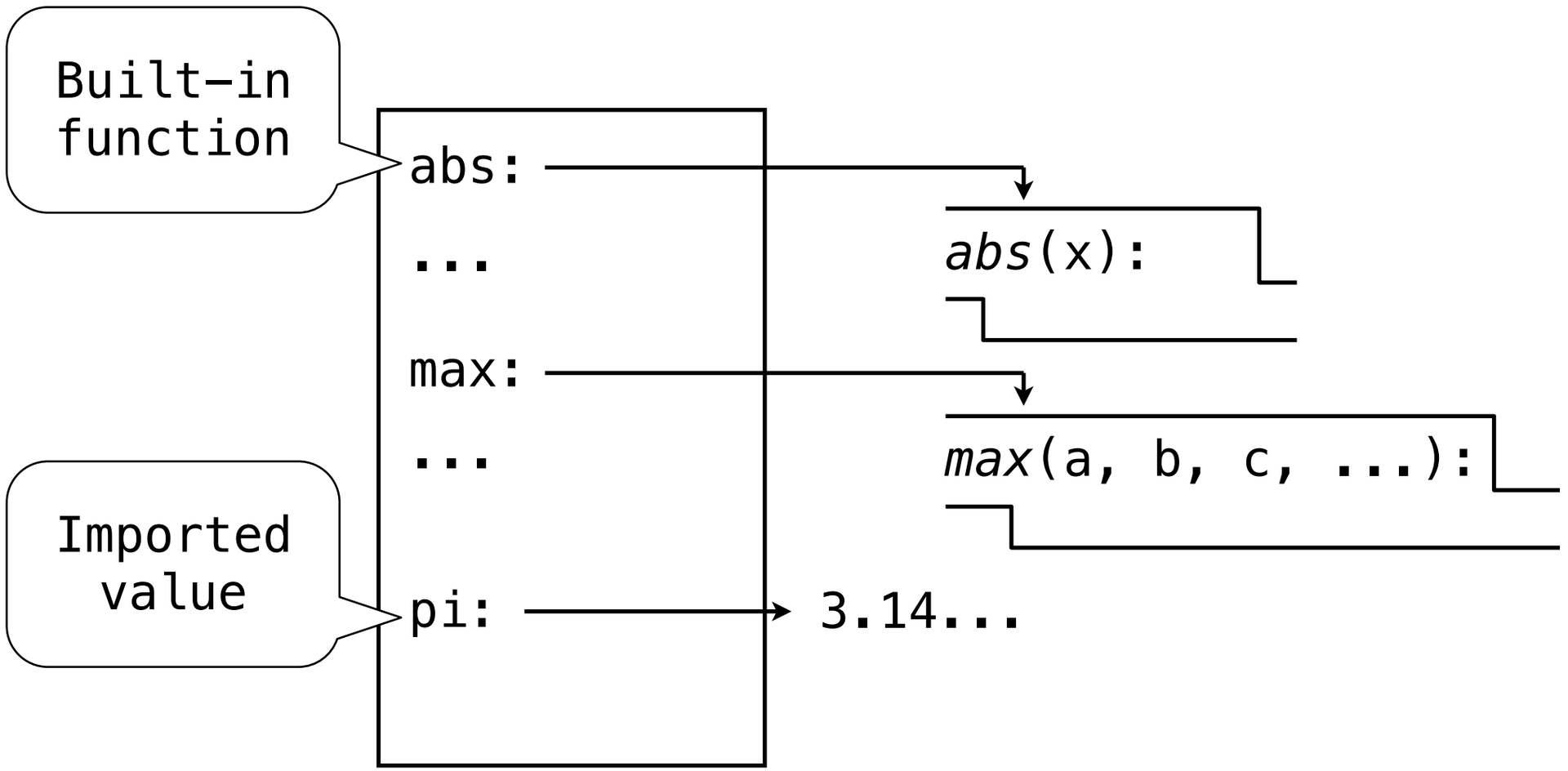

```
from math import pi
```

Environments



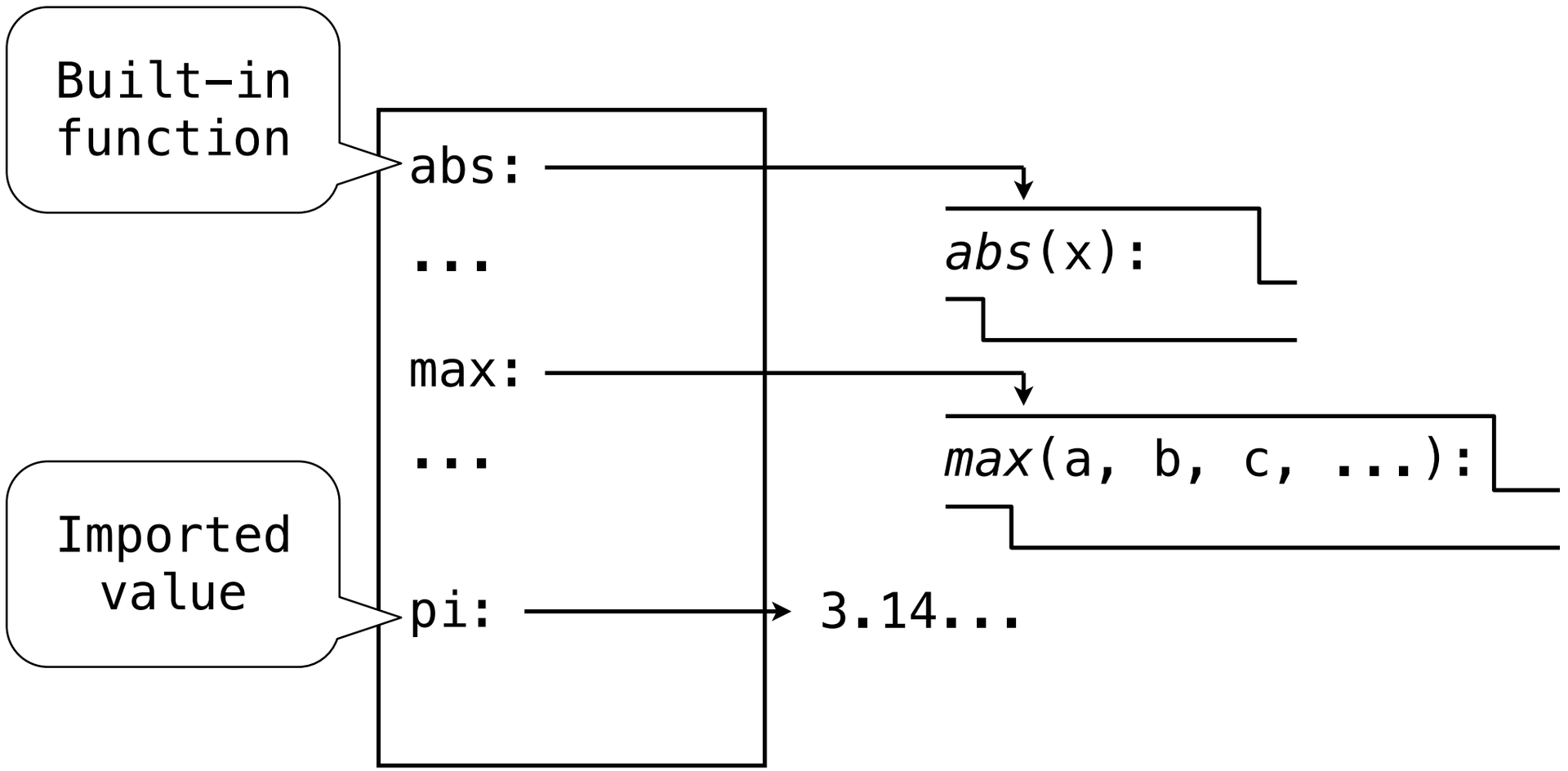

```
from math import pi
```

Environments



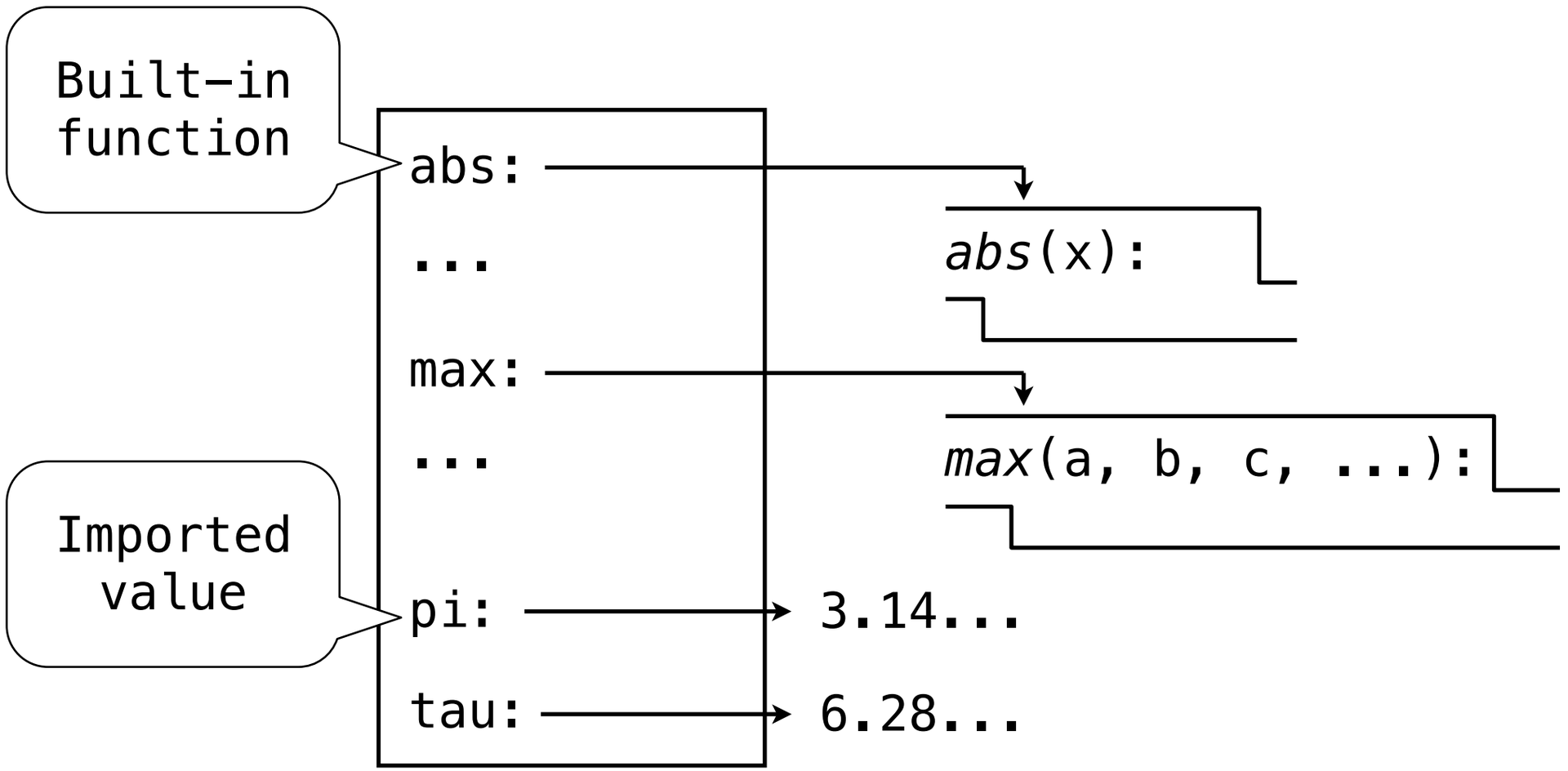

```
from math import pi
```

Environments



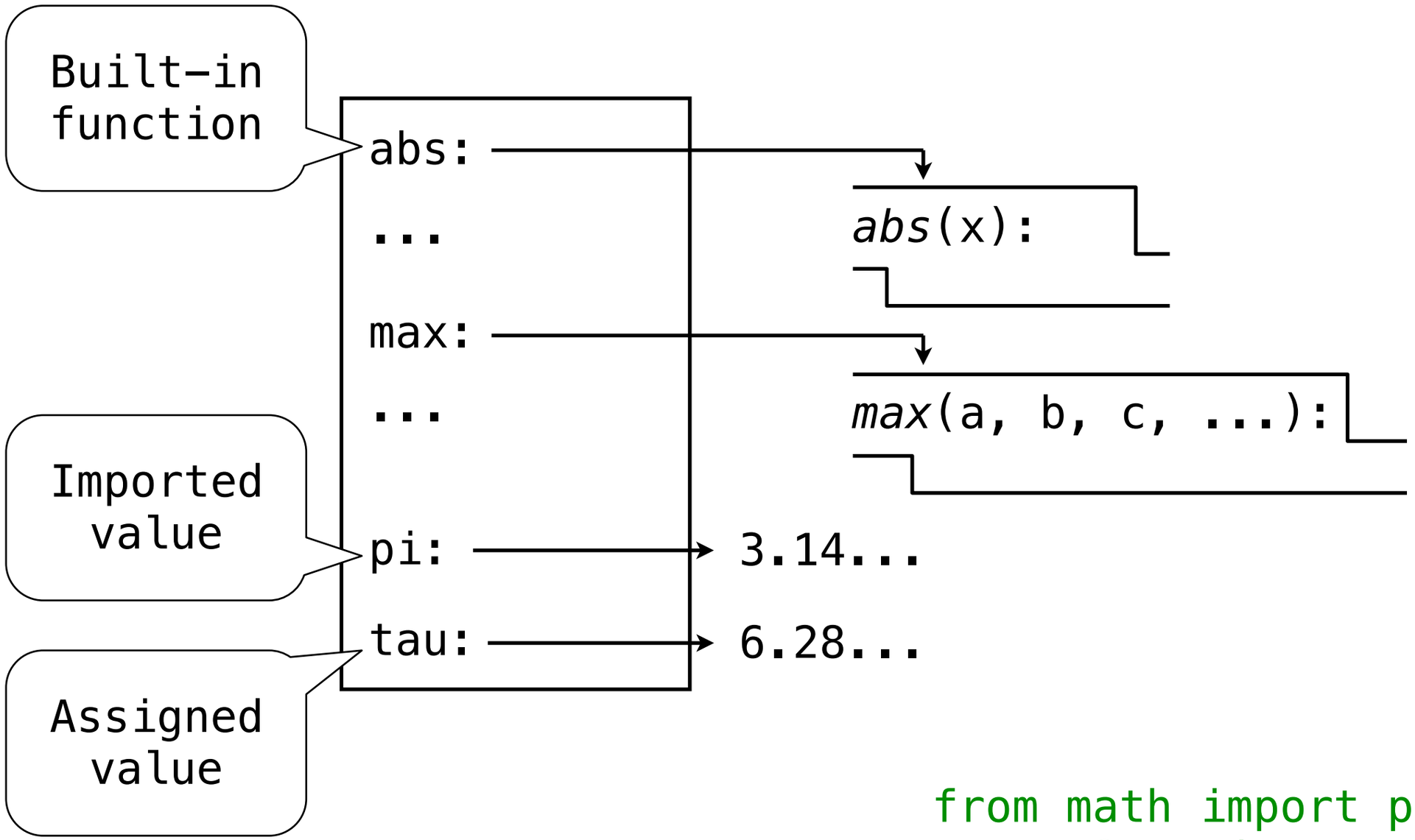

```
from math import pi
tau = 2 * pi
```

Environments



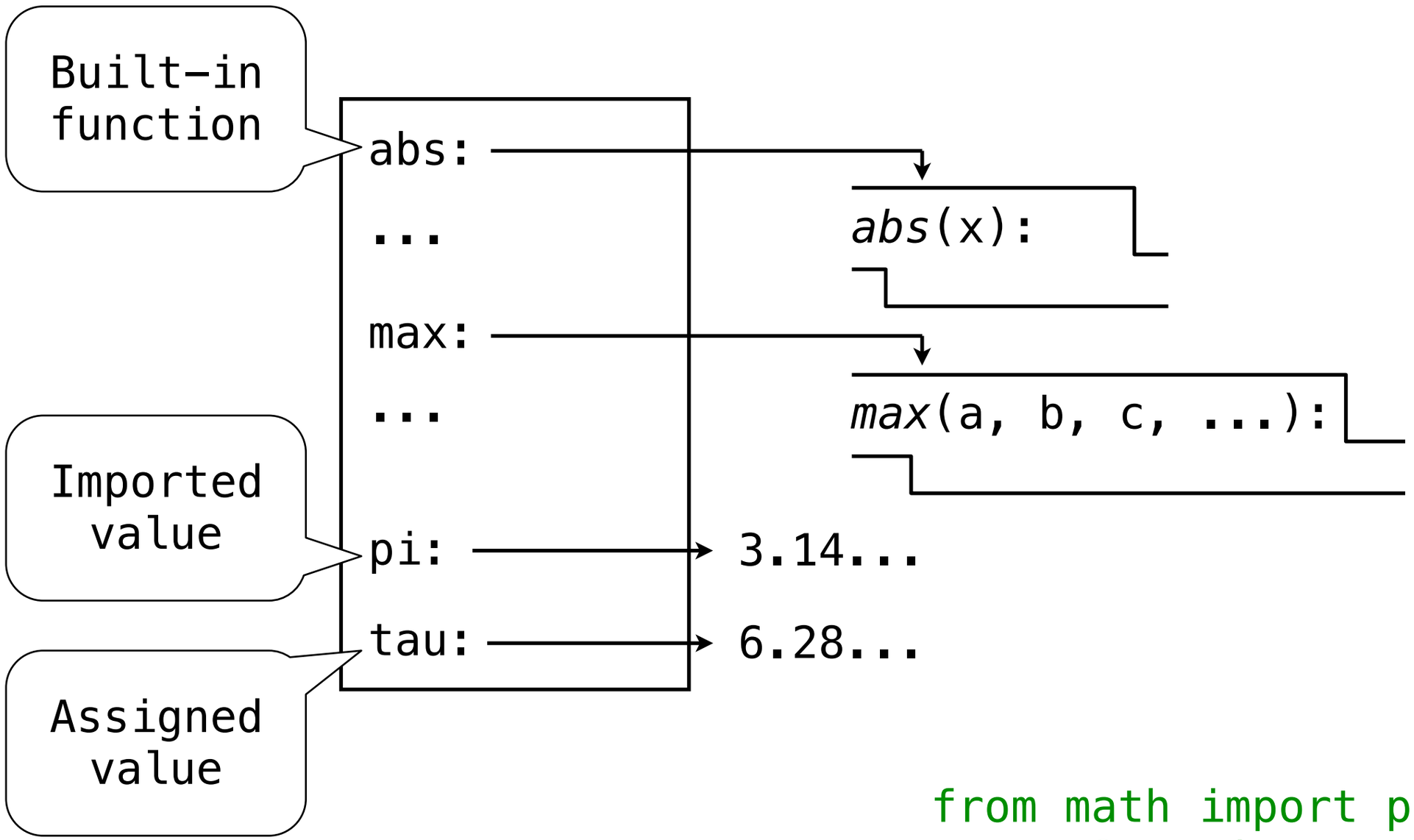

```
from math import pi
tau = 2 * pi
```

Environments




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

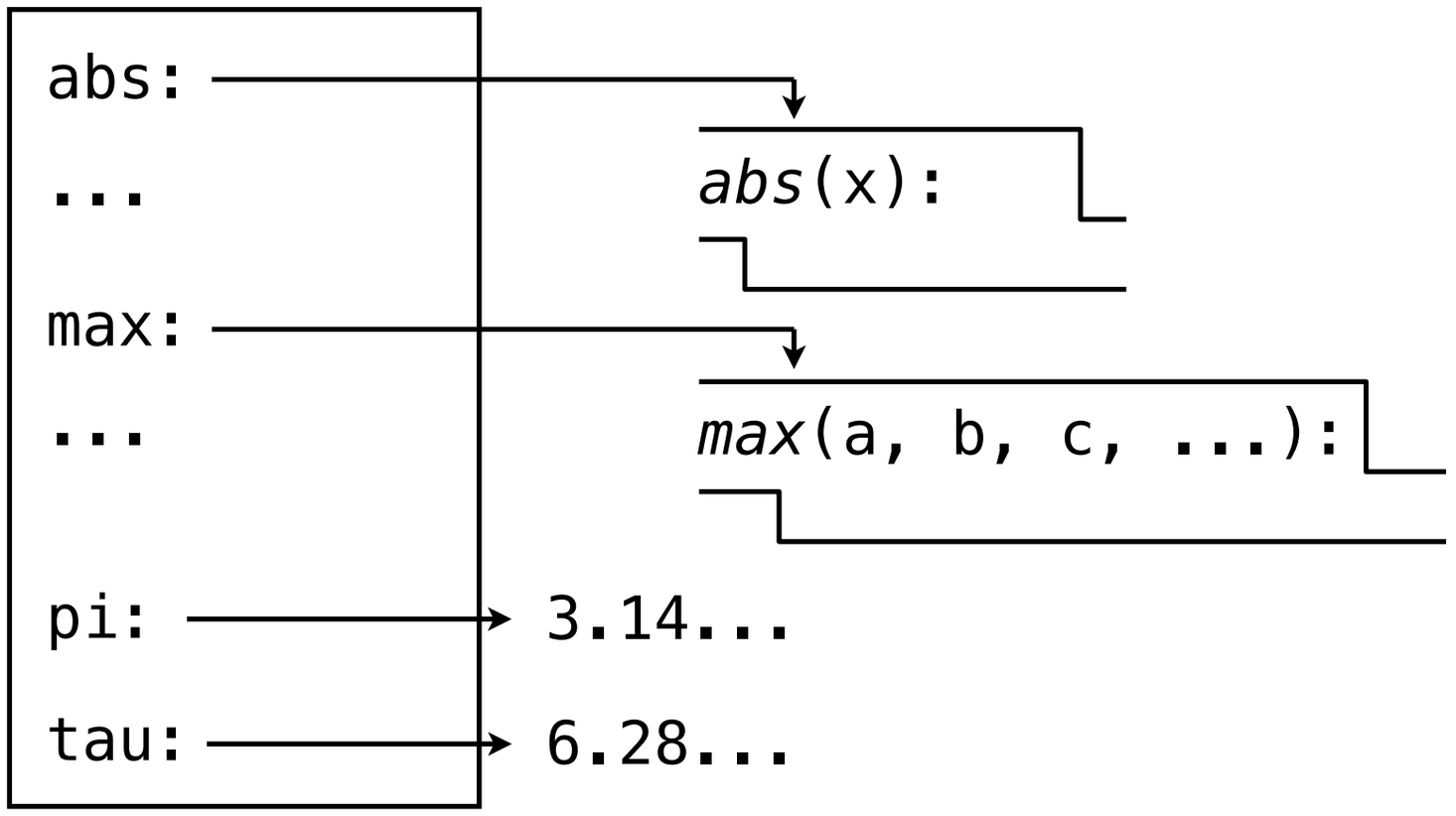
Environments




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

The environment does not track where names came from

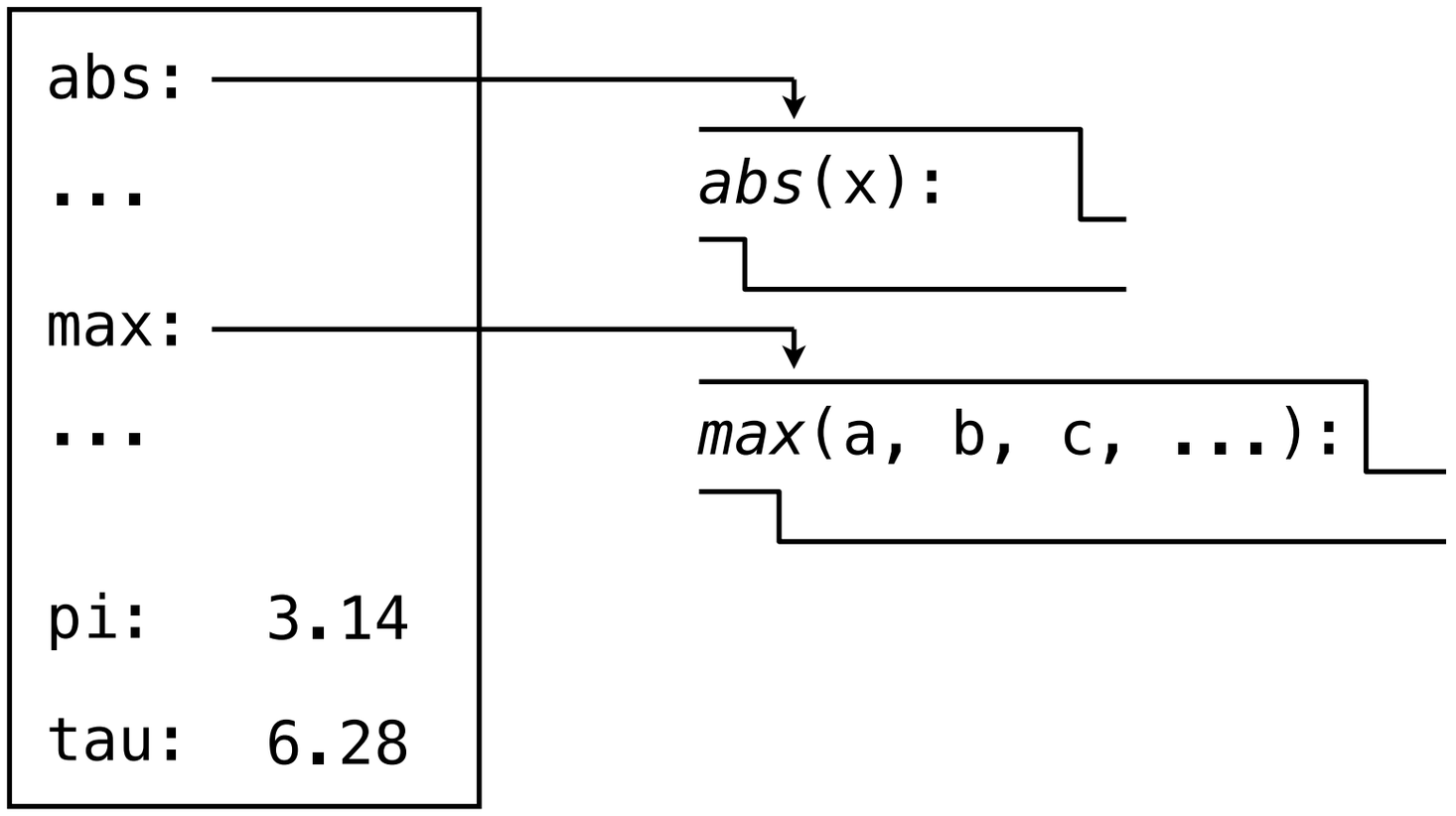
Environments




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

The environment does not track where names came from

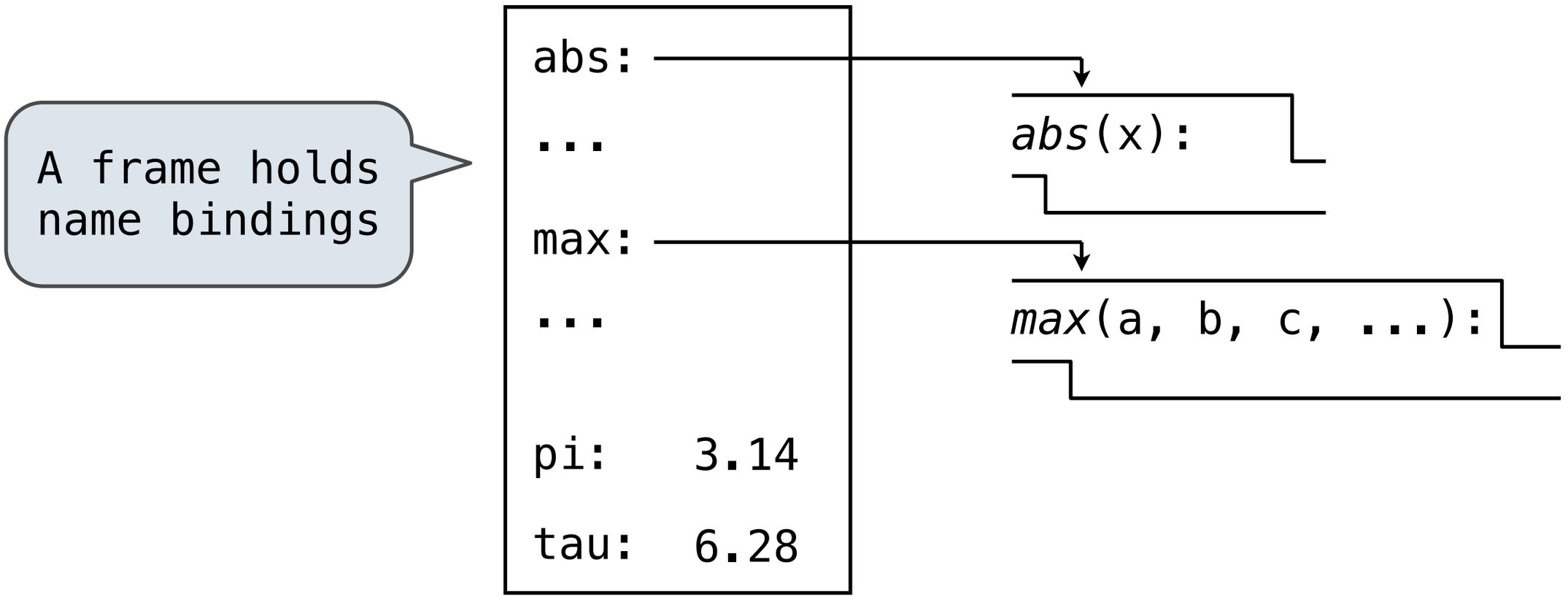
Environments




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

The environment does not track where names came from

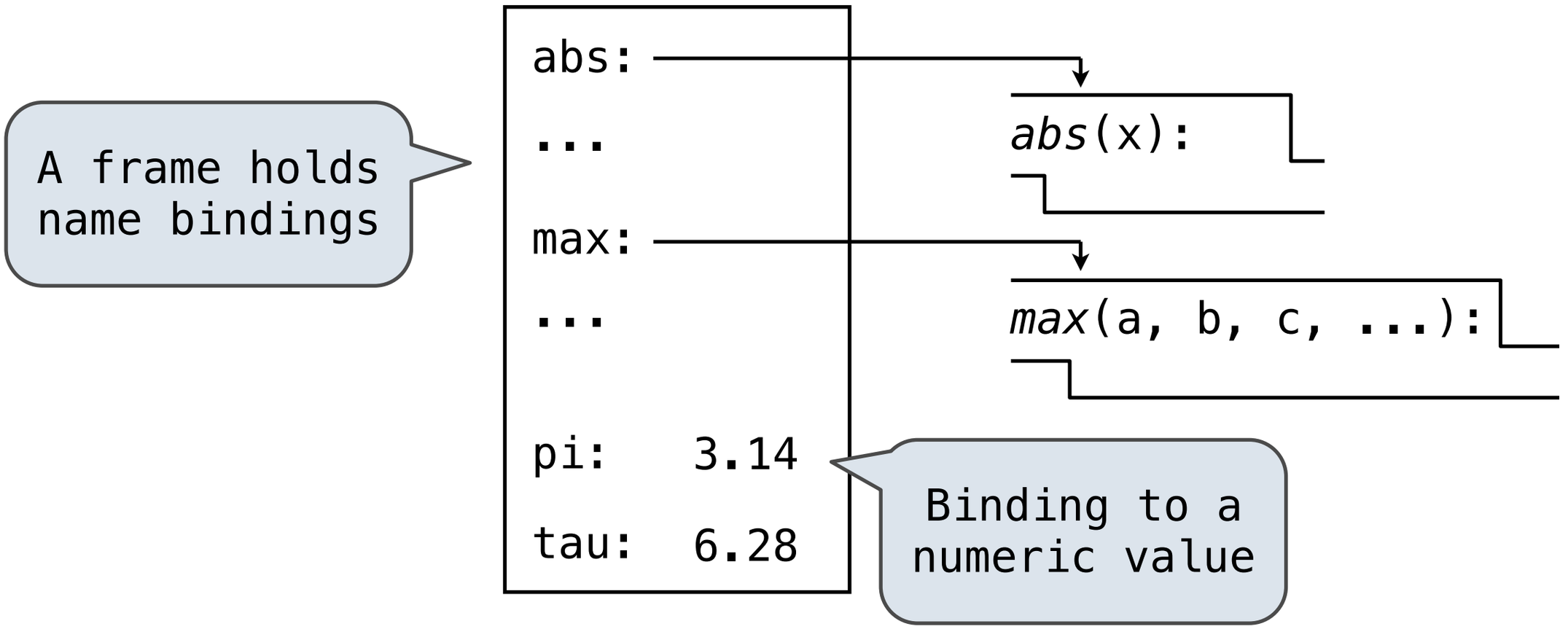
Environments




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

The environment does not track where names came from

Environments




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

The environment does not track where names came from

User-Defined Functions

Named values are a simple means of abstraction

Named *expressions* are a more powerful means of abstraction

User-Defined Functions

Named values are a simple means of abstraction

Named *expressions* are a more powerful means of abstraction

```
>>> def
```

User-Defined Functions

Named values are a simple means of abstraction

Named *expressions* are a more powerful means of abstraction

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

User-Defined Functions

Named values are a simple means of abstraction

Named *expressions* are a more powerful means of abstraction

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

User-Defined Functions

Named values are a simple means of abstraction

Named *expressions* are a more powerful means of abstraction

def expressions:

- Create a new function
- Bind a name to it

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

User-Defined Functions

Named values are a simple means of abstraction

Named *expressions* are a more powerful means of abstraction

```
def expressions:

• Create a new function

• Bind a name to it
```

User-Defined Functions

Named values are a simple means of abstraction

Named *expressions* are a more powerful means of abstraction

```
>>> def
```

def expressions:

- Create a new function
- Bind a name to it

User-Defined Functions

Named values are a simple means of abstraction

Named *expressions* are a more powerful means of abstraction

```
def expressions:
```

- Create a new function
- Bind a name to it

```
>>> def square(x):
```

User-Defined Functions

Named values are a simple means of abstraction

Named *expressions* are a more powerful means of abstraction

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

def expressions:

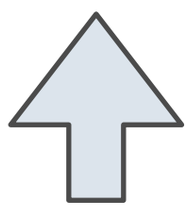
- Create a new function
- Bind a name to it

User-Defined Functions

Named values are a simple means of abstraction

Named *expressions* are a more powerful means of abstraction

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



def expressions:

- Create a new function
- Bind a name to it

User-Defined Functions

Named values are a simple means of abstraction

Named *expressions* are a more powerful means of abstraction

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

def expressions:

- Create a new function
- Bind a name to it

User-Defined Functions

Named values are a simple means of abstraction

Named *expressions* are a more powerful means of abstraction

```
def expressions:
```

- Create a new function
- Bind a name to it

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

User-Defined Functions

Named values are a simple means of abstraction

Named *expressions* are a more powerful means of abstraction

def expressions:

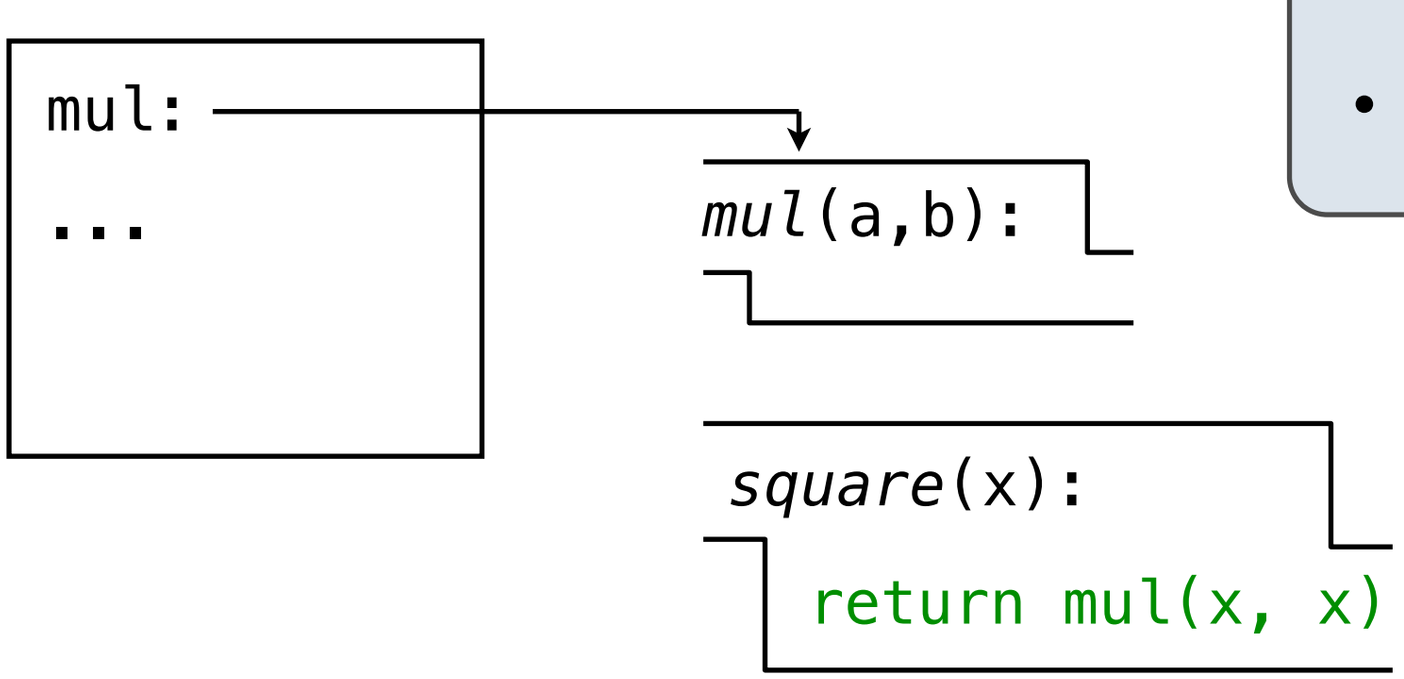
- Create a new function
- Bind a name to it

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

User-Defined Functions

Named values are a simple means of abstraction

Named *expressions* are a more powerful means of abstraction



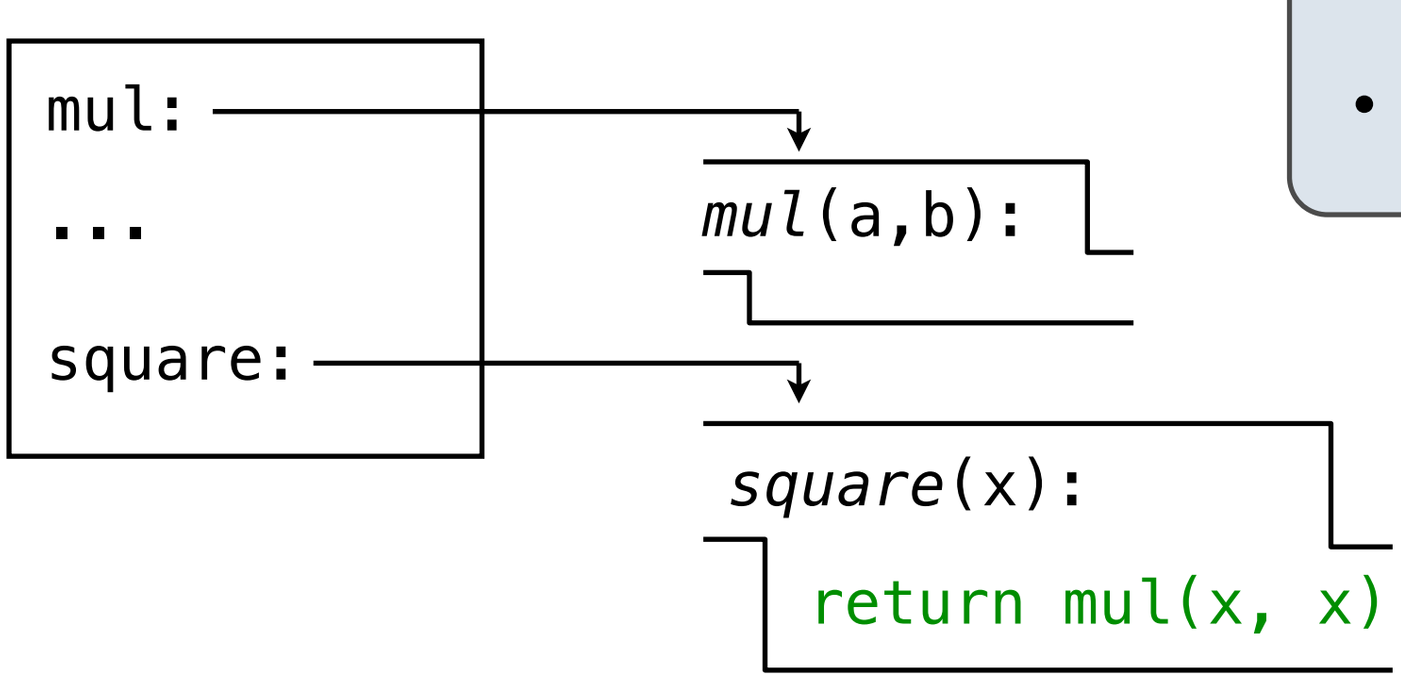
def expressions:

- Create a new function
- Bind a name to it

User-Defined Functions

Named values are a simple means of abstraction

Named *expressions* are a more powerful means of abstraction



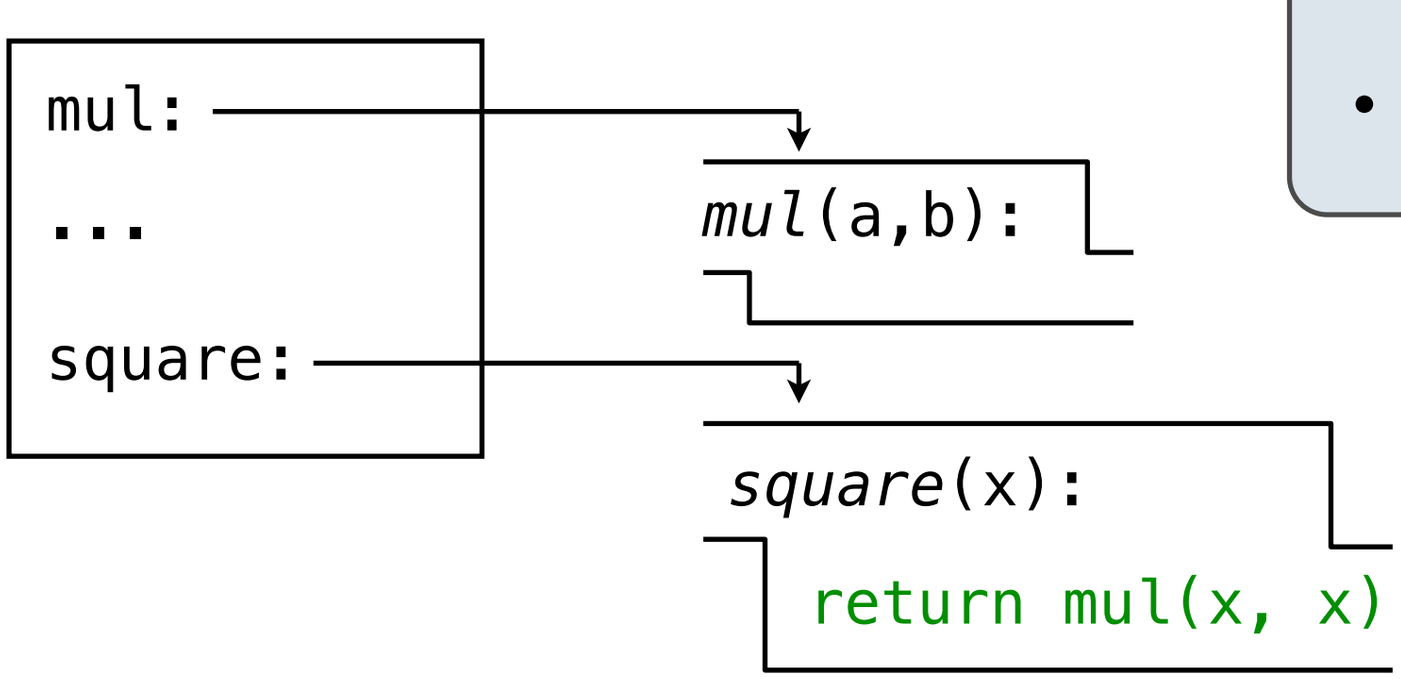
def expressions:

- Create a new function
- Bind a name to it

User-Defined Functions

Named values are a simple means of abstraction

Named *expressions* are a more powerful means of abstraction

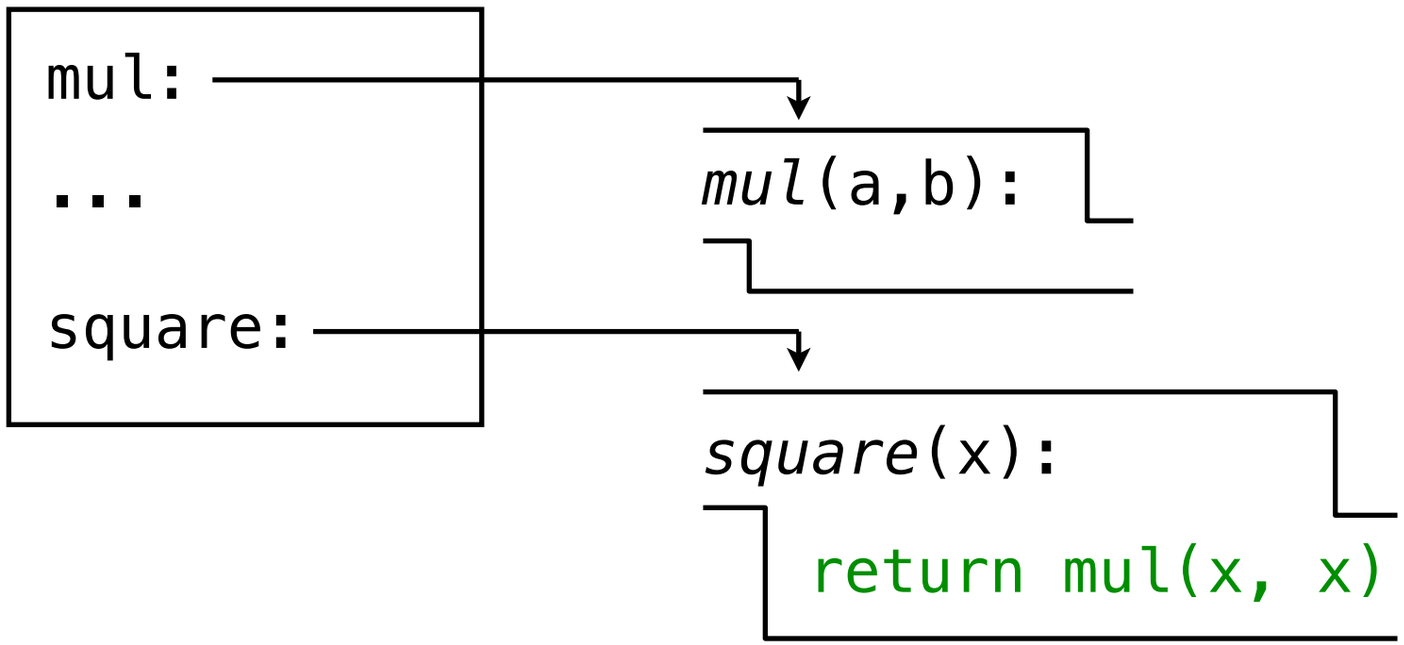


def expressions:

- Create a new function
- Bind a name to it

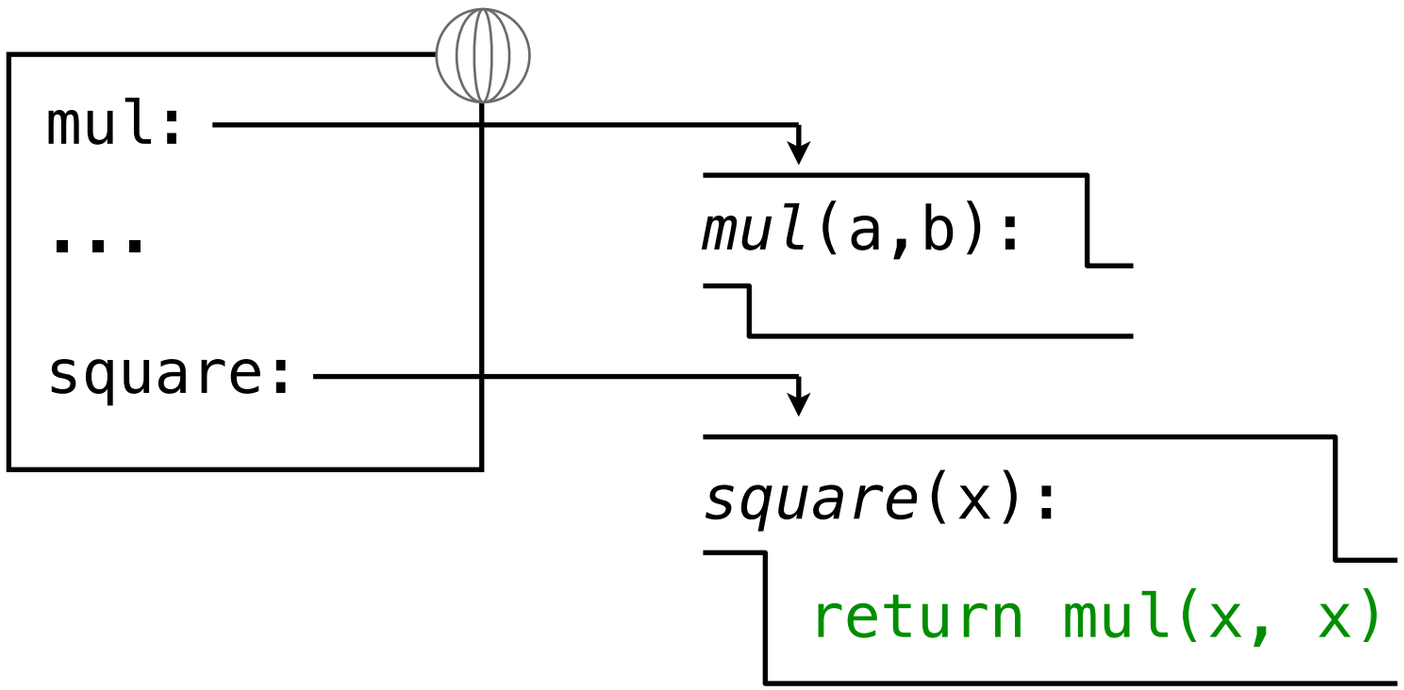
(Demo)

Calling User-Defined Functions



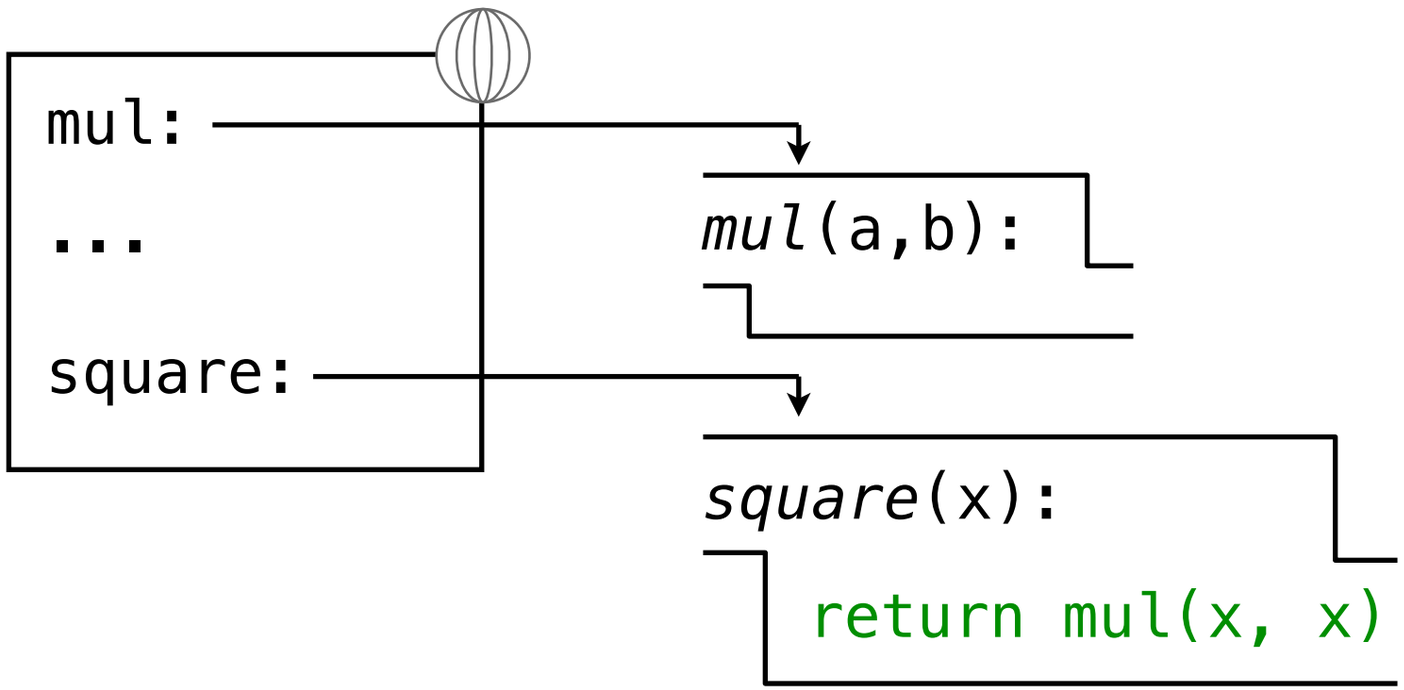

```
from operator import mul
def square(x):
    return mul(x, x)
```

Calling User-Defined Functions



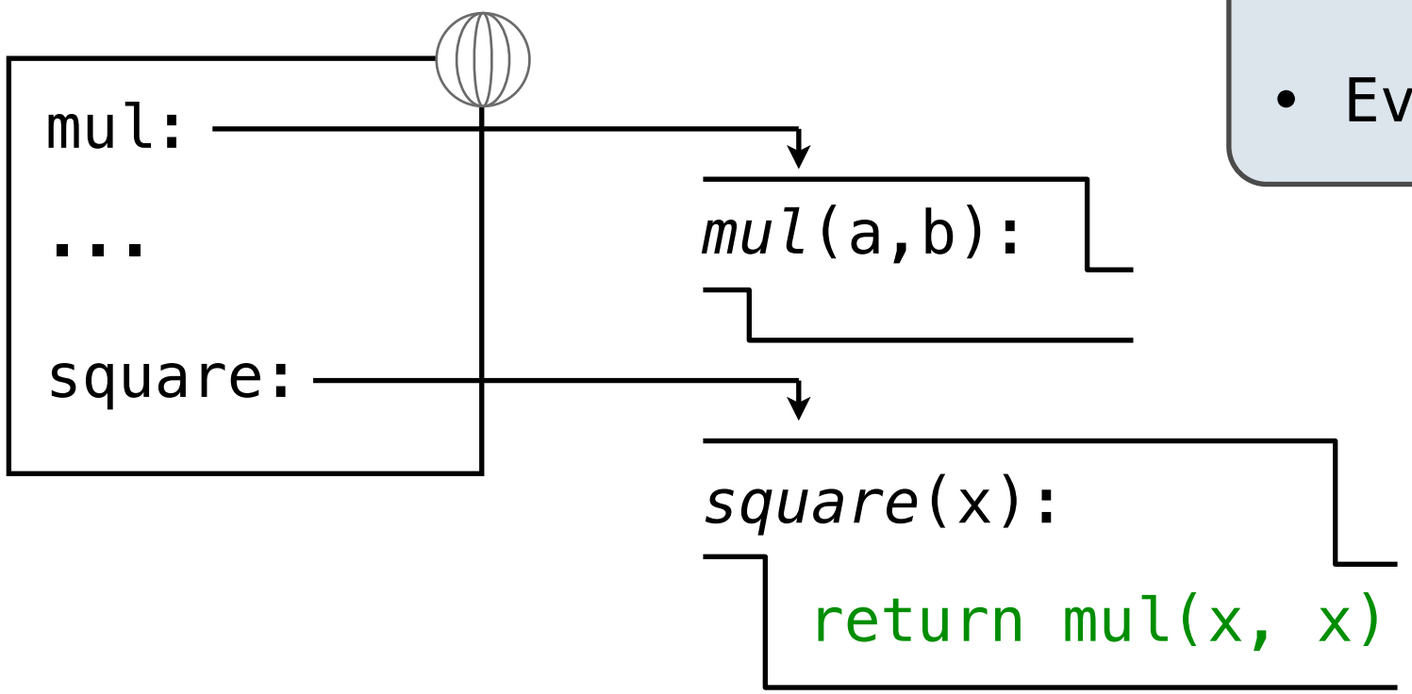

```
from operator import mul
def square(x):
    return mul(x, x)
```

Calling User-Defined Functions




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

Calling User-Defined Functions

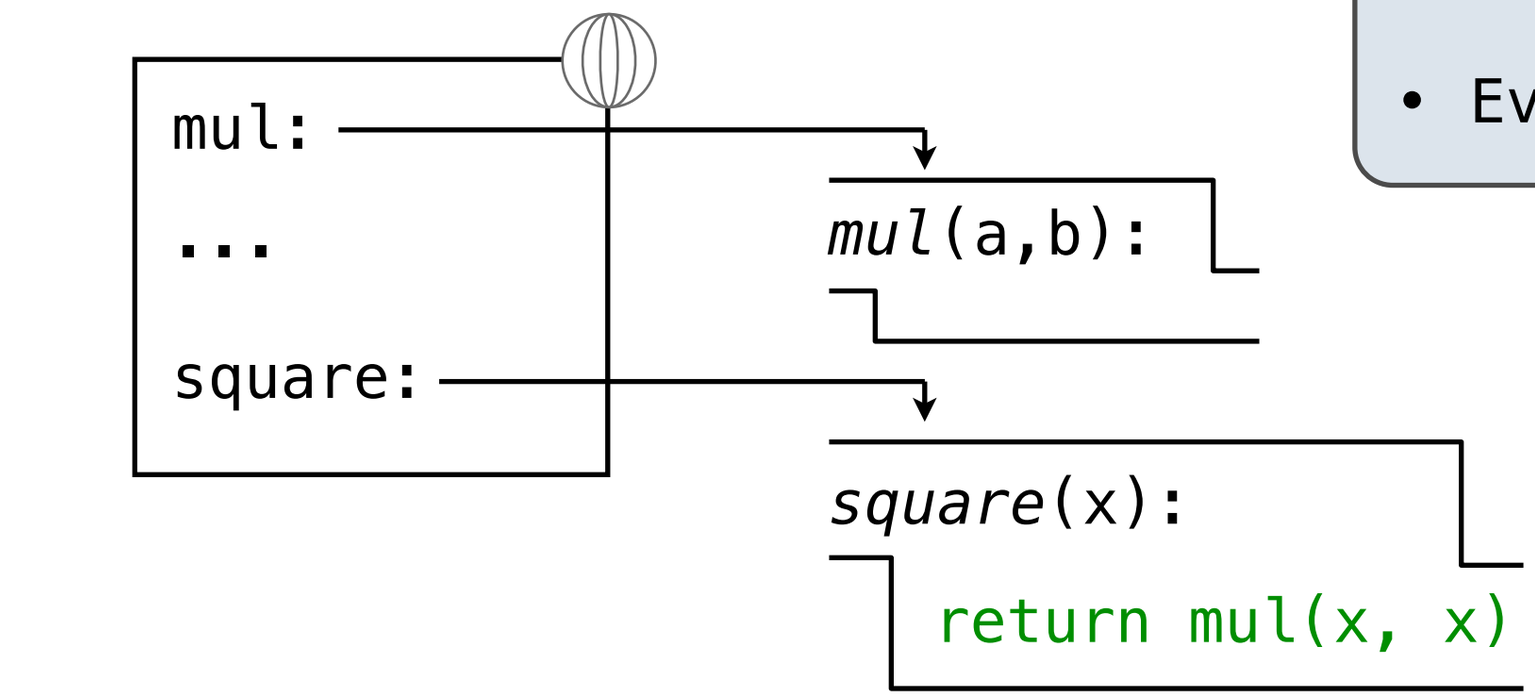


- Bind formal parameters
- Eval return expression

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

Calling User-Defined Functions

- Bind formal parameters
- Eval return expression



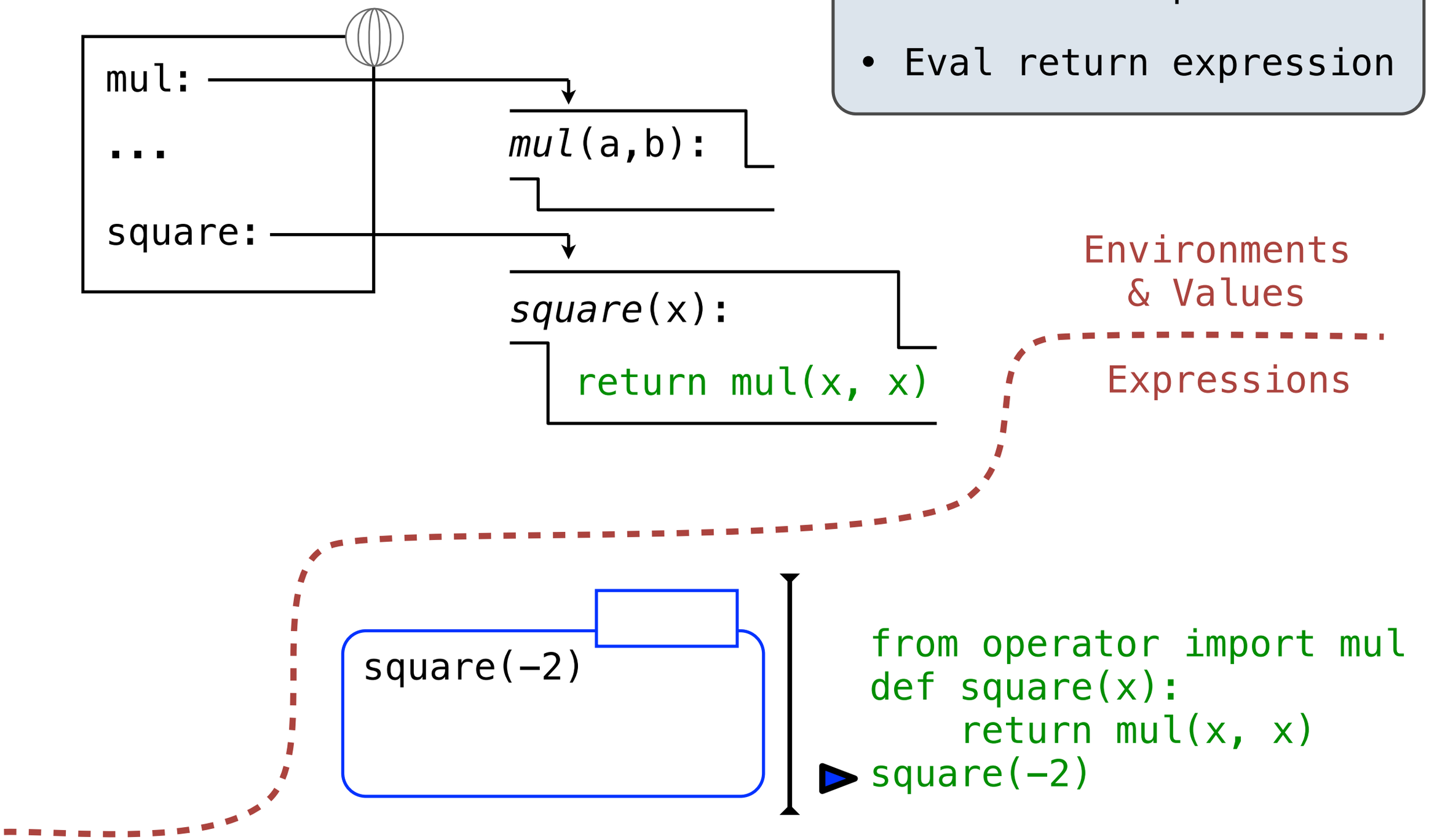
Environments & Values

Expressions

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

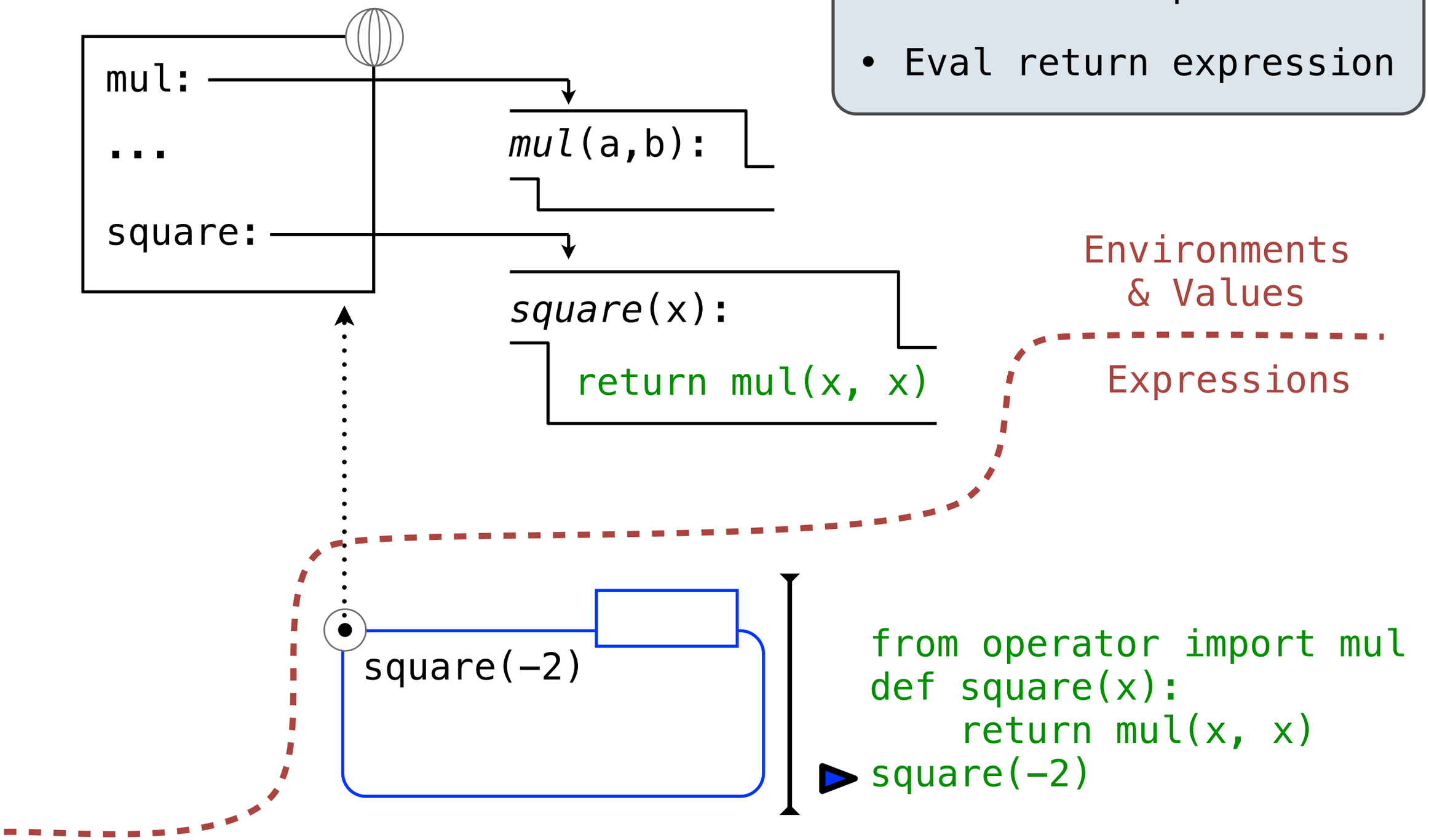
Calling User-Defined Functions

- Bind formal parameters
- Eval return expression



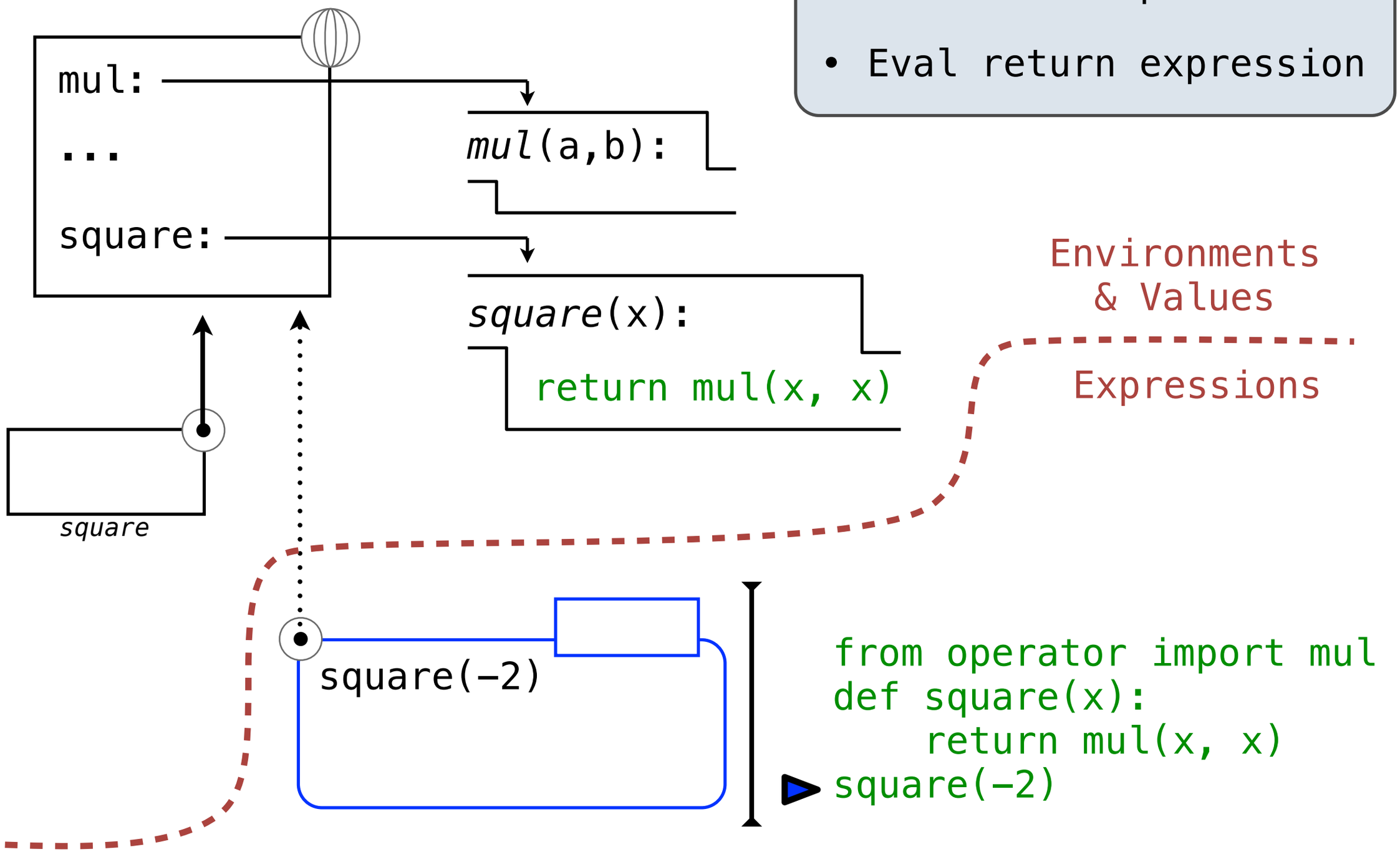
Calling User-Defined Functions

- Bind formal parameters
- Eval return expression



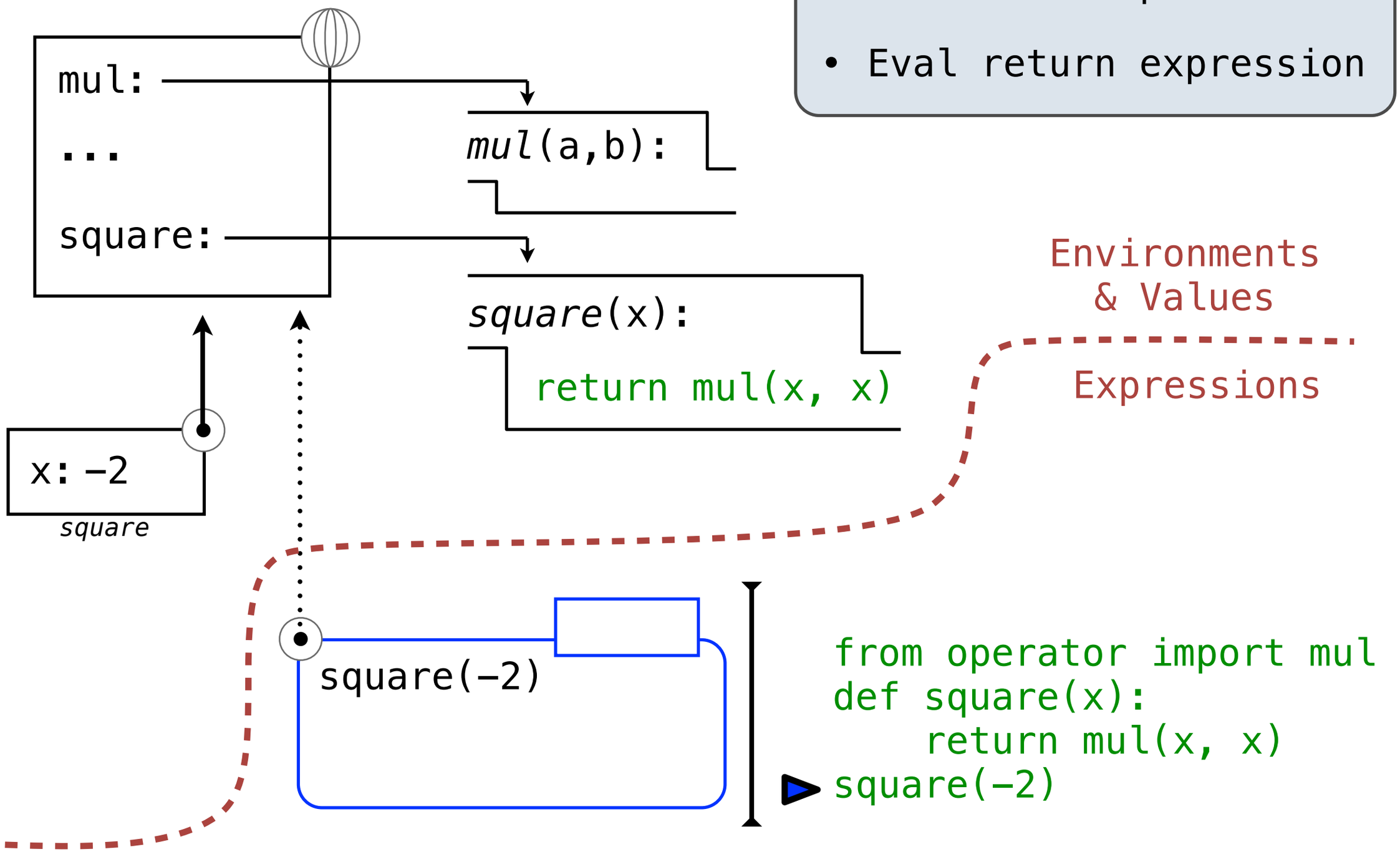
Calling User-Defined Functions

- Bind formal parameters
- Eval return expression



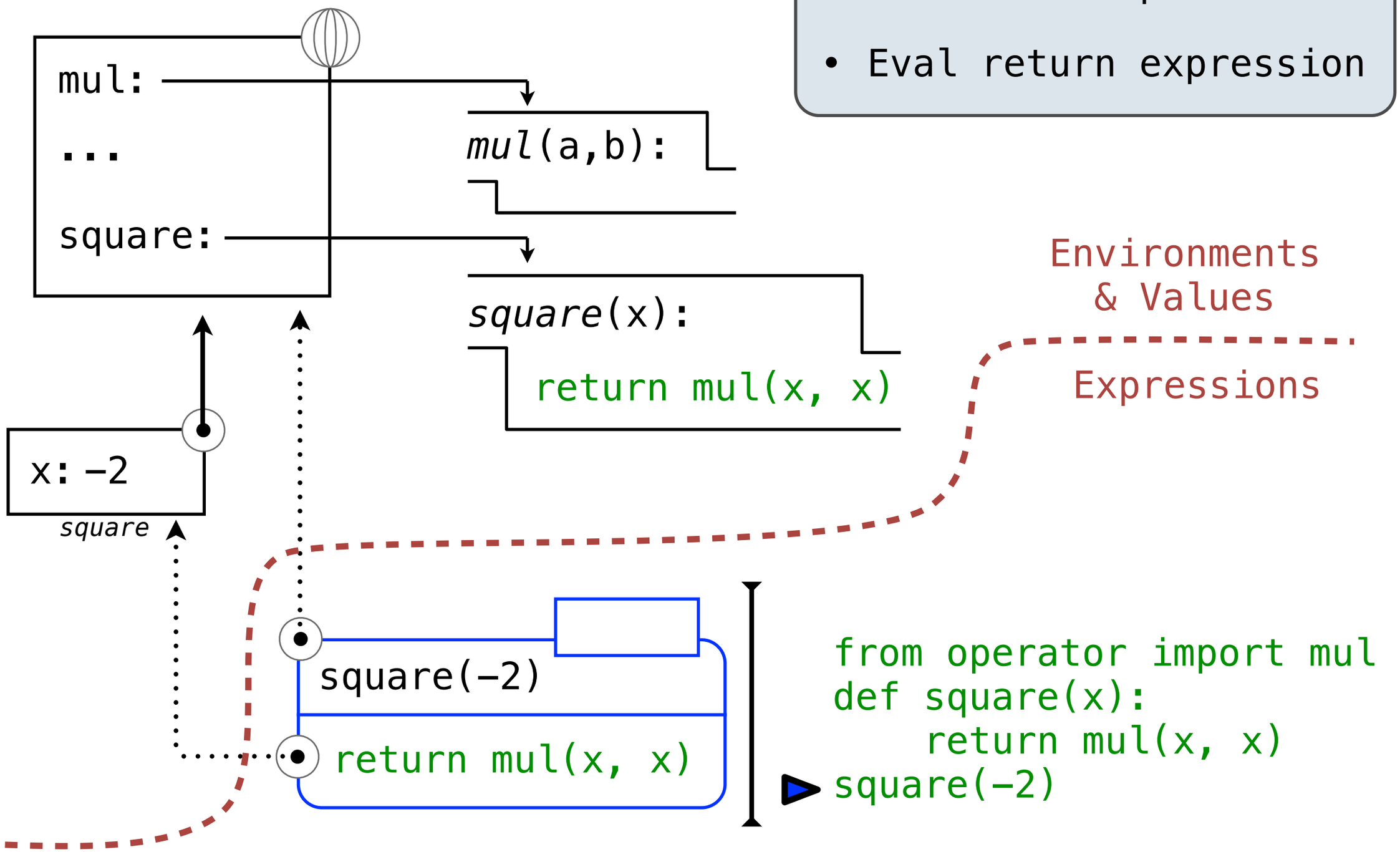
Calling User-Defined Functions

- Bind formal parameters
- Eval return expression



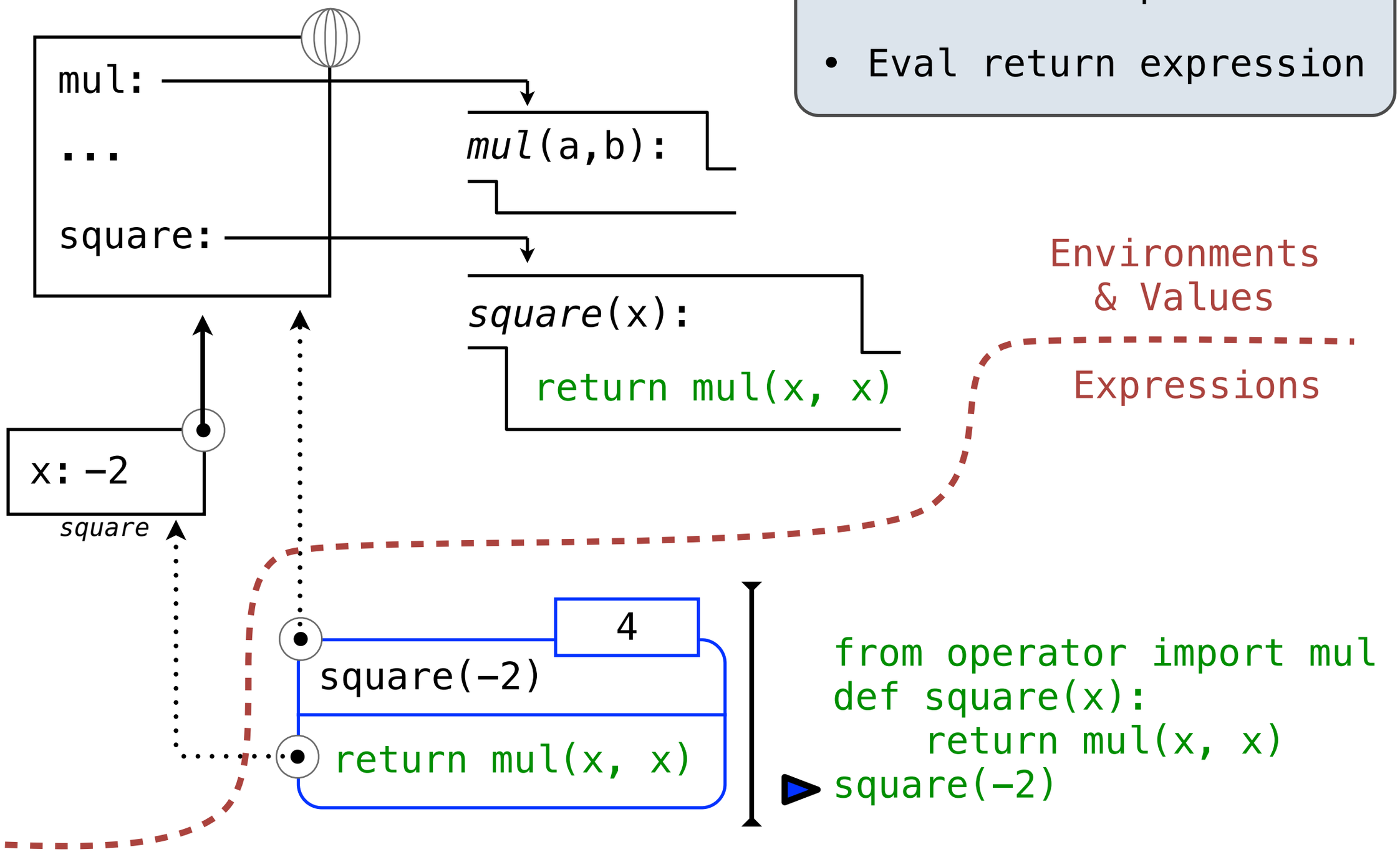
Calling User-Defined Functions

- Bind formal parameters
- Eval return expression



Calling User-Defined Functions

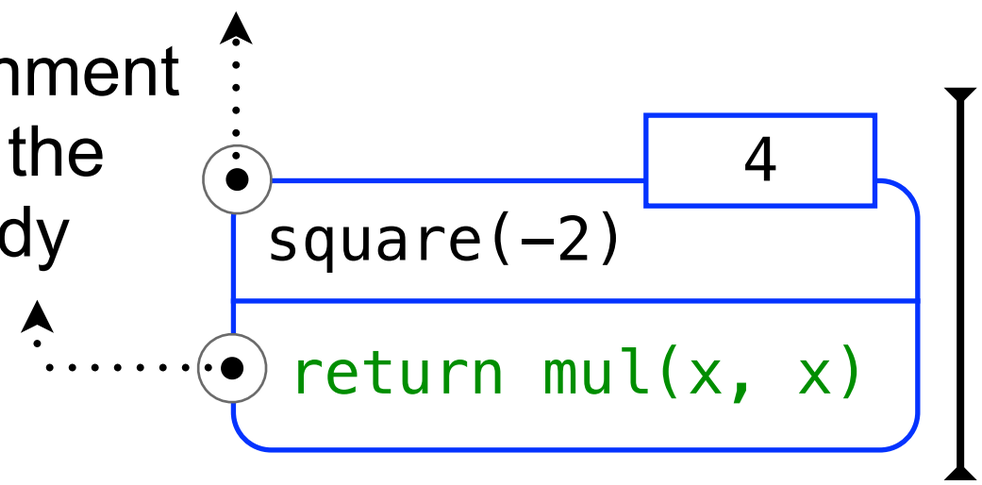
- Bind formal parameters
- Eval return expression



Calling User-Defined Functions

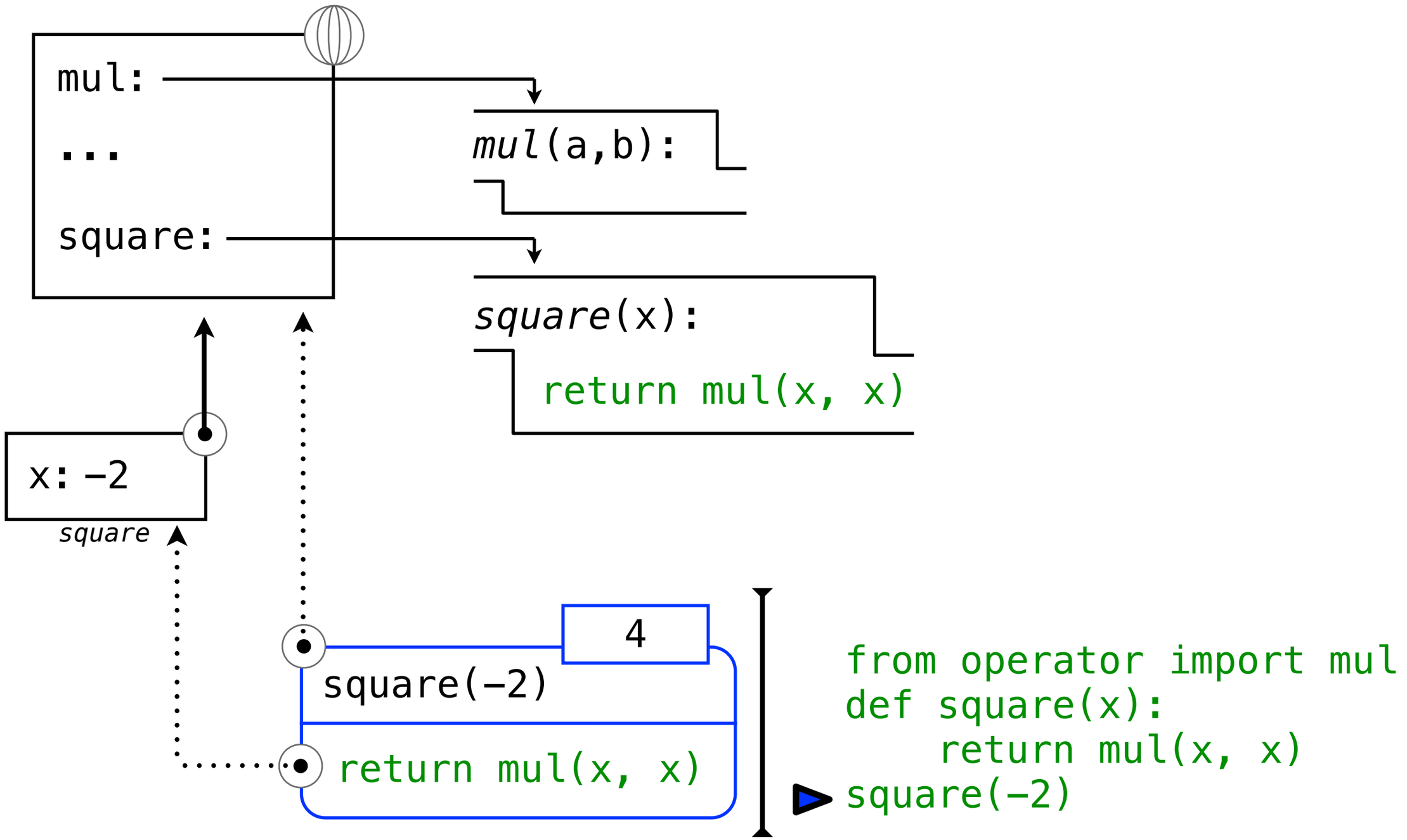
The existing environment in which the call expression is evaluated

The environment created for the function body

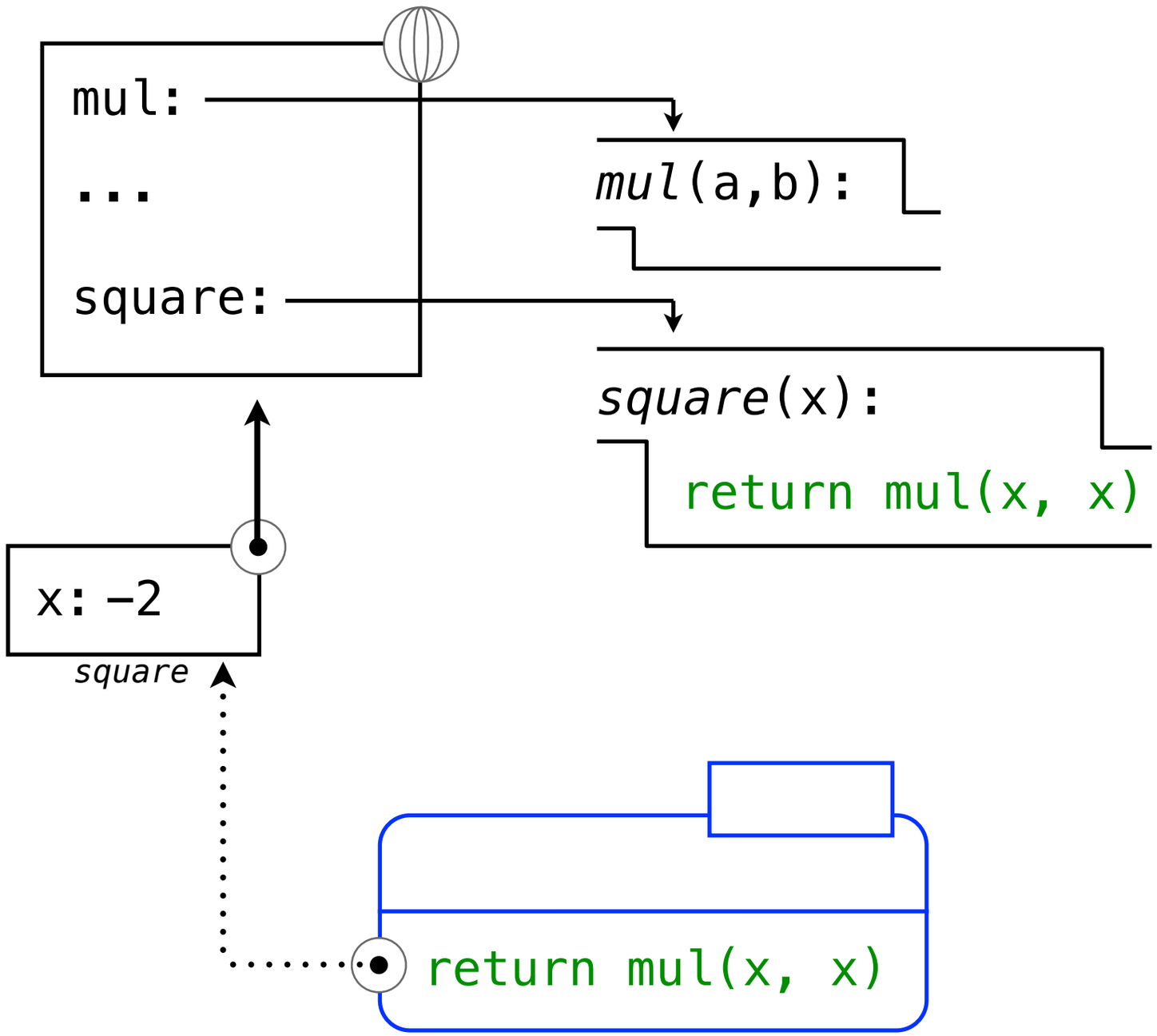



```
from operator import mul
def square(x):
    return mul(x, x)
square(-2)
```

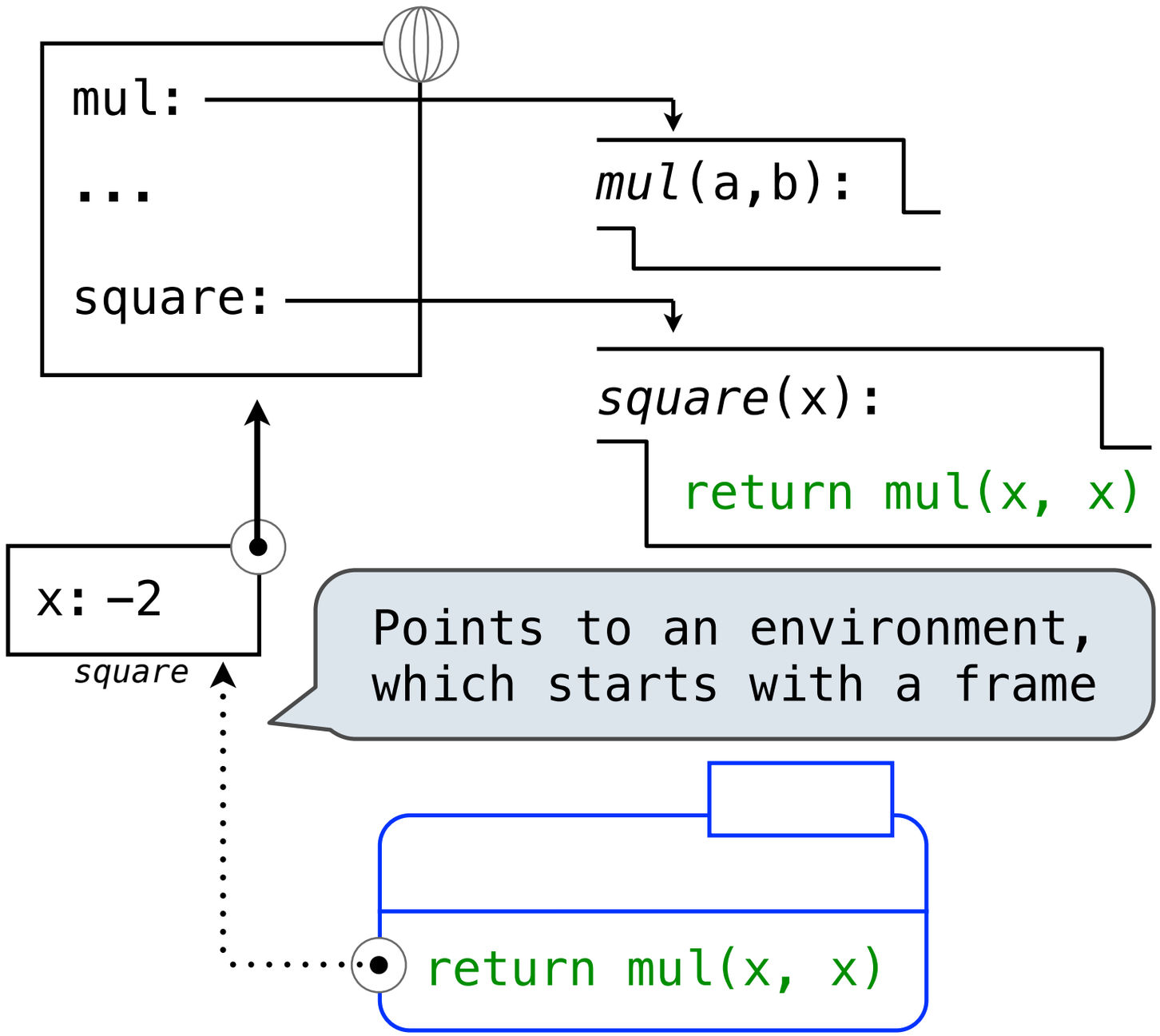
Calling User-Defined Functions



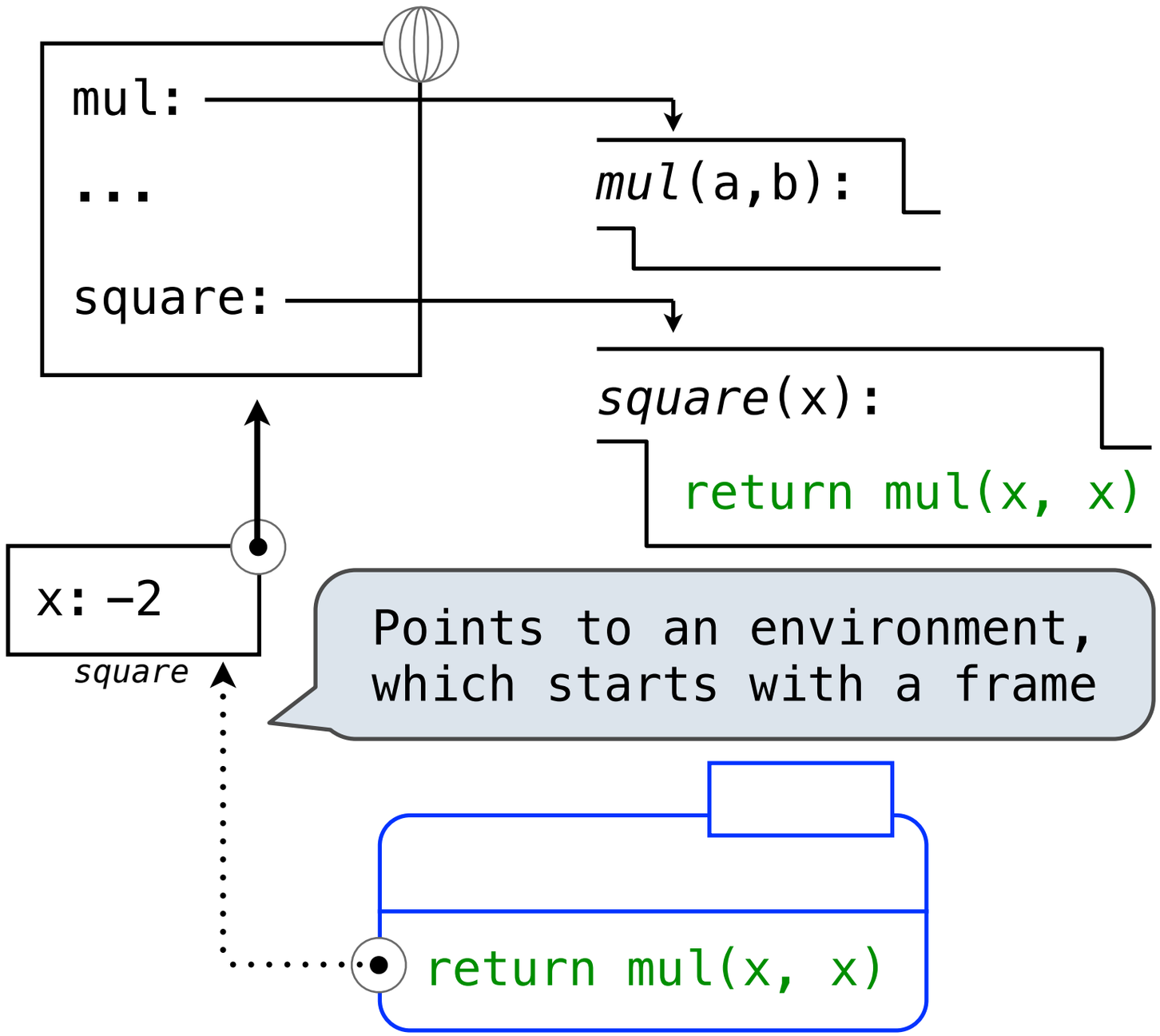
Evaluating a Name in an Environment



Evaluating a Name in an Environment

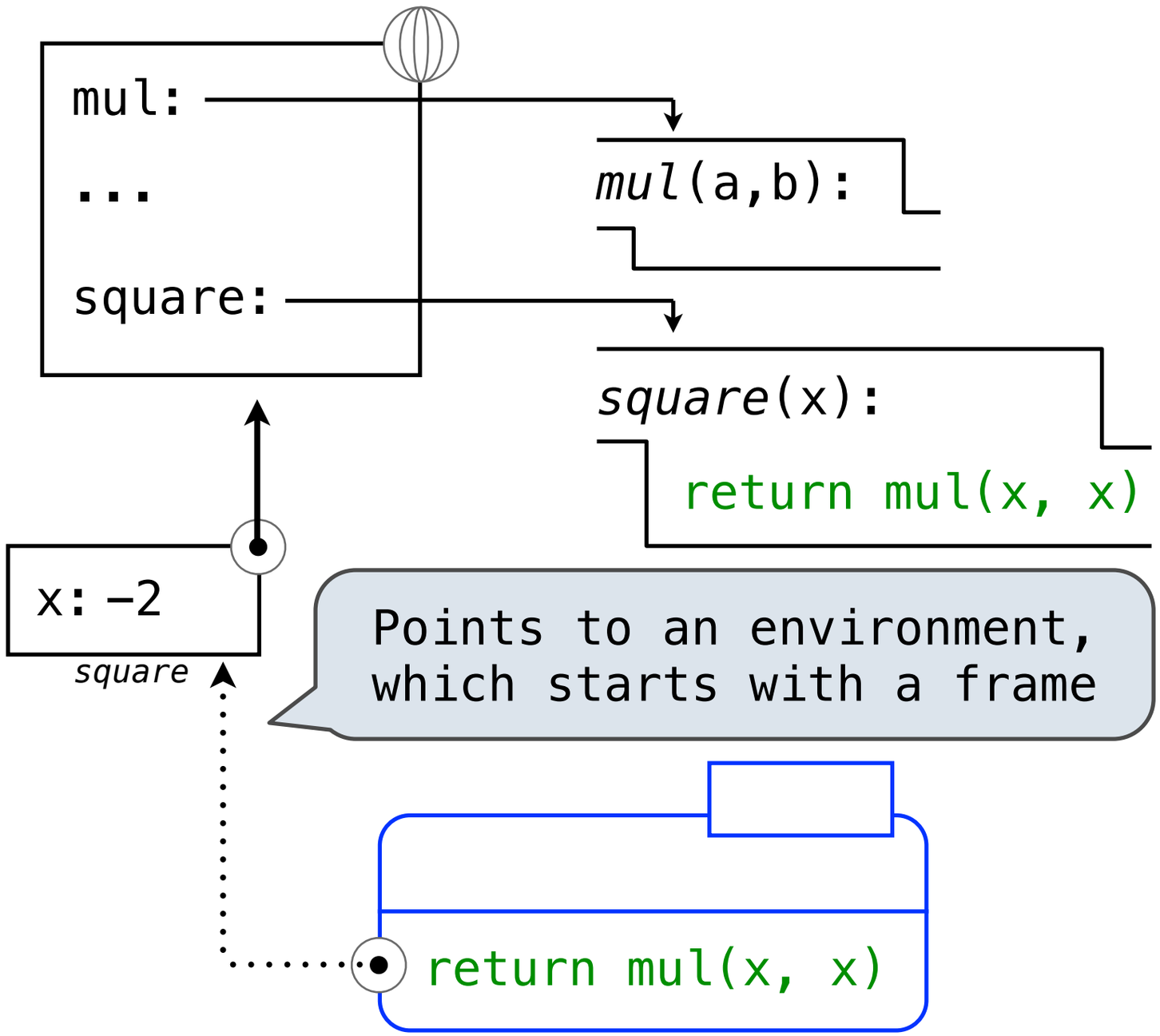


Evaluating a Name in an Environment



A name evaluates to the **value bound** to that name

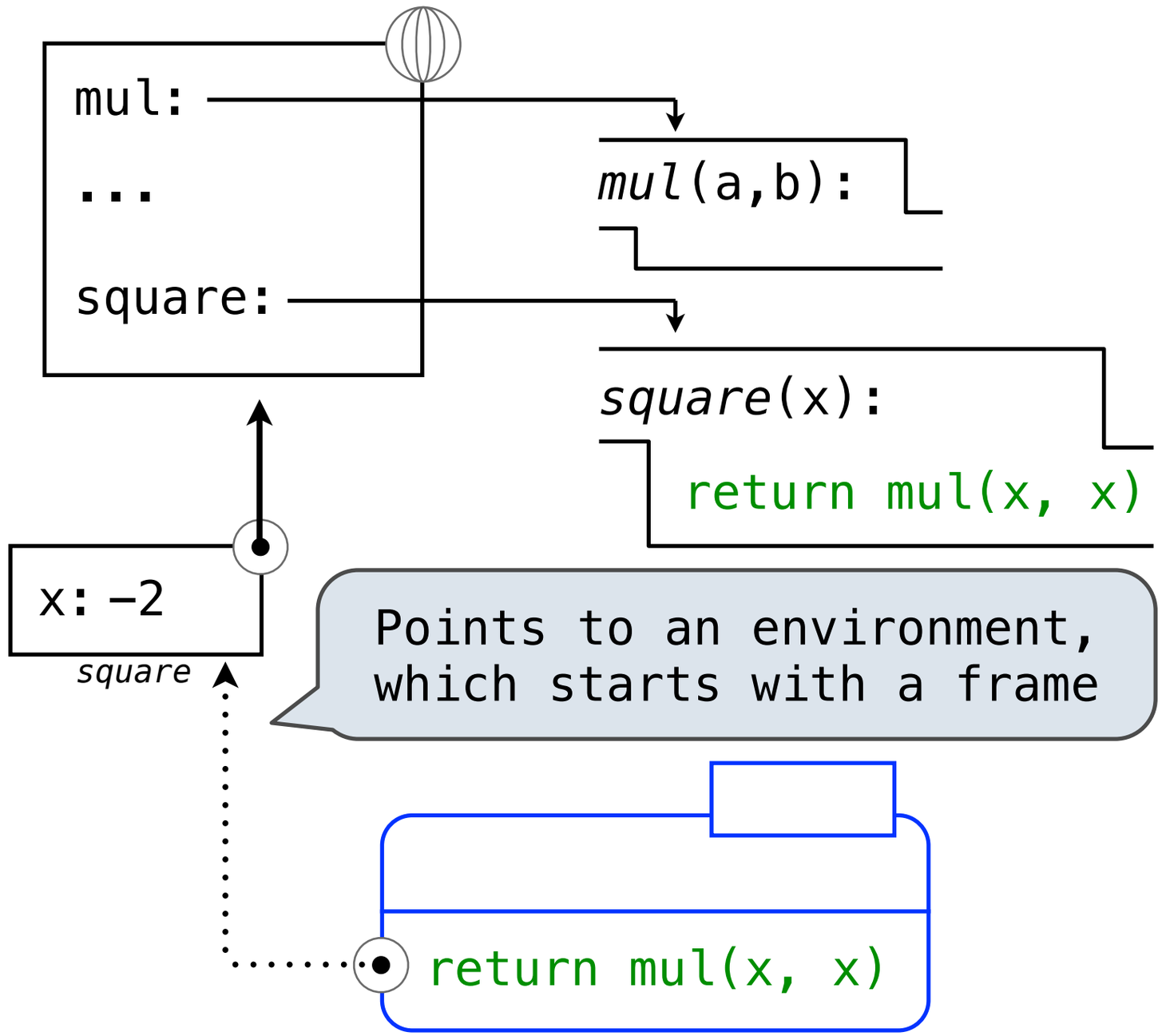
Evaluating a Name in an Environment



A name evaluates to the **value bound** to that name

...in the **earliest** frame of the **current environment**

Evaluating a Name in an Environment

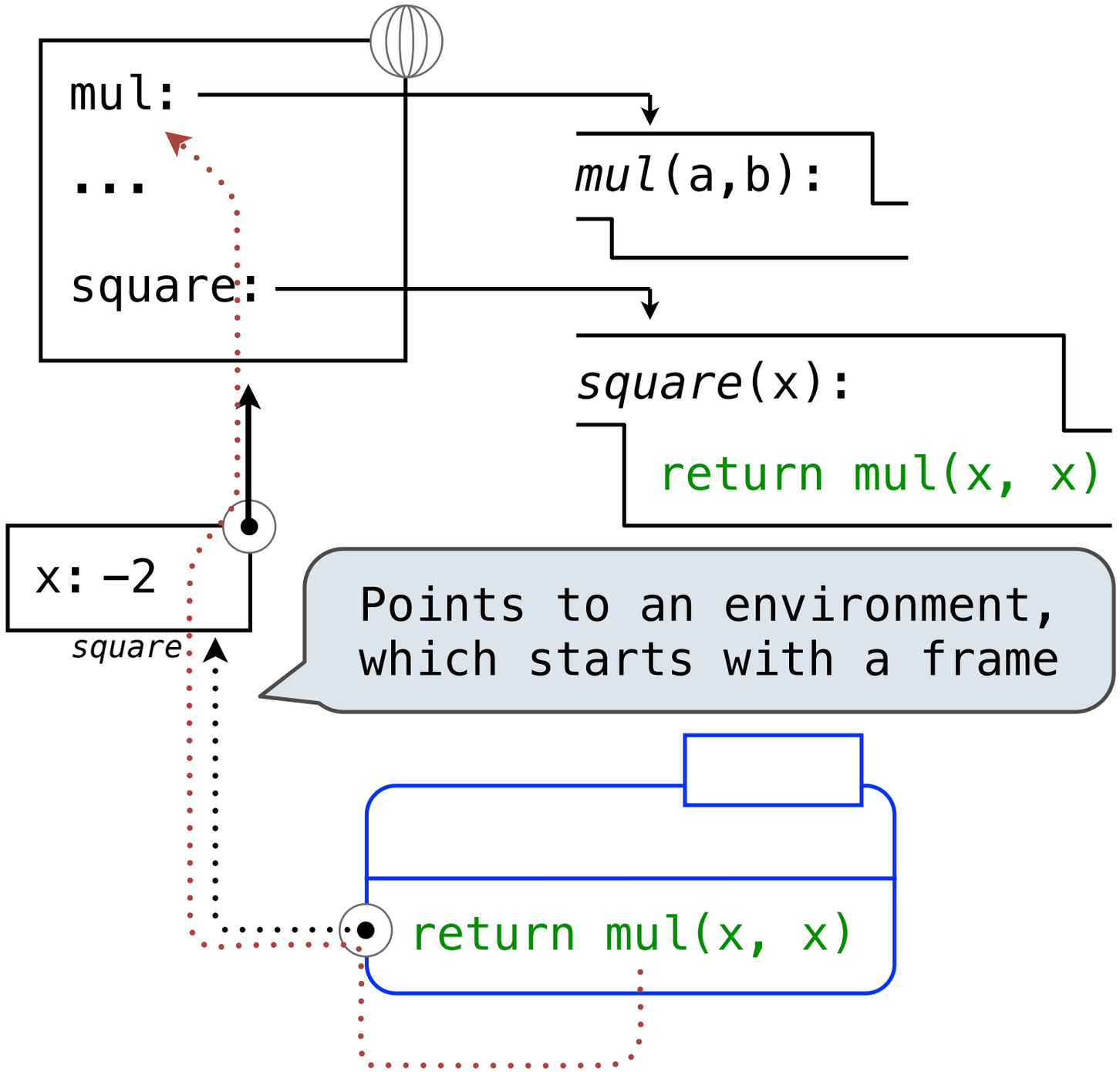


A name evaluates to the **value bound** to that name

...in the **earliest** frame of the **current environment**

...in which that **name is found**

Evaluating a Name in an Environment

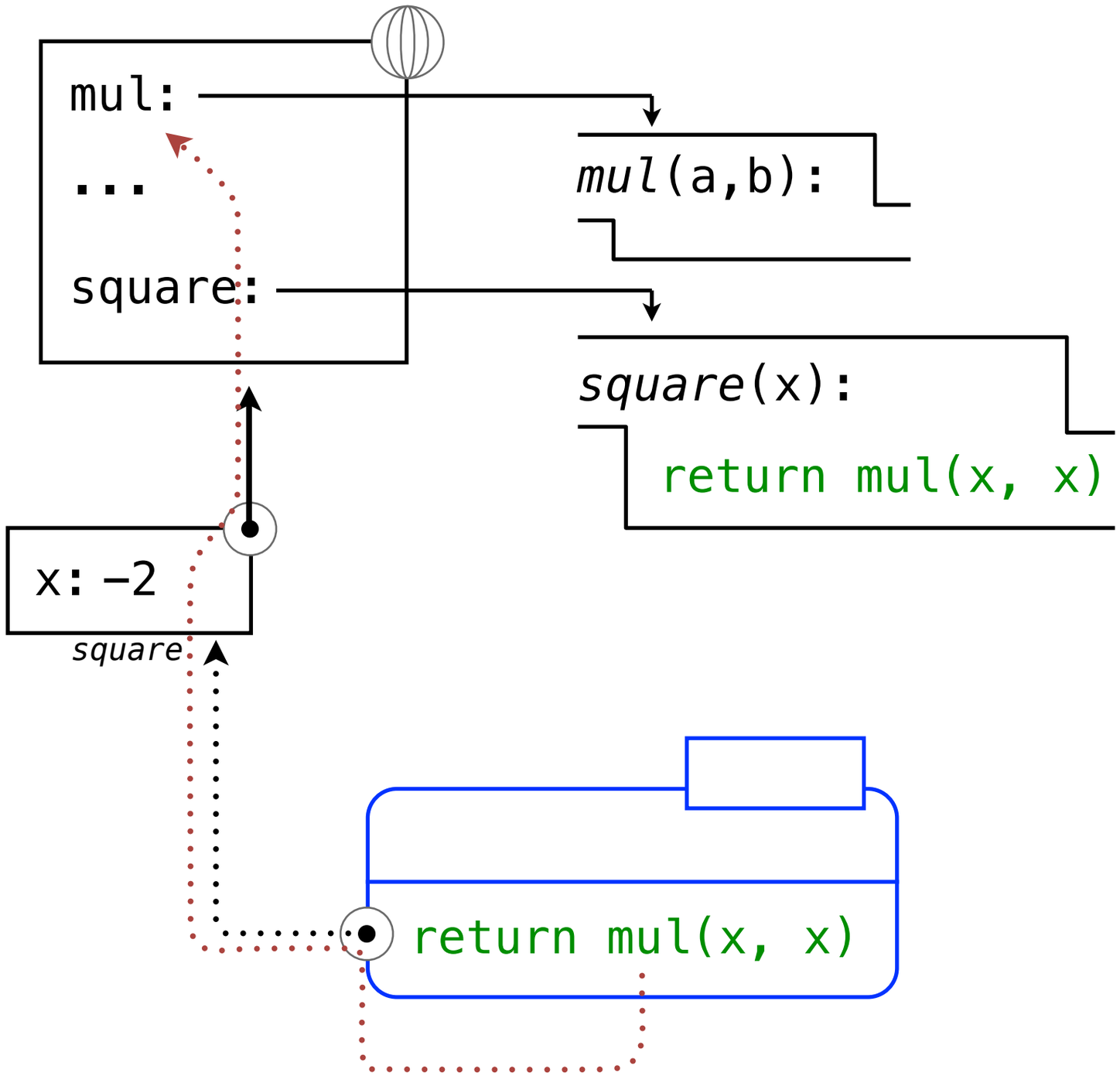


A name evaluates to the **value bound** to that name

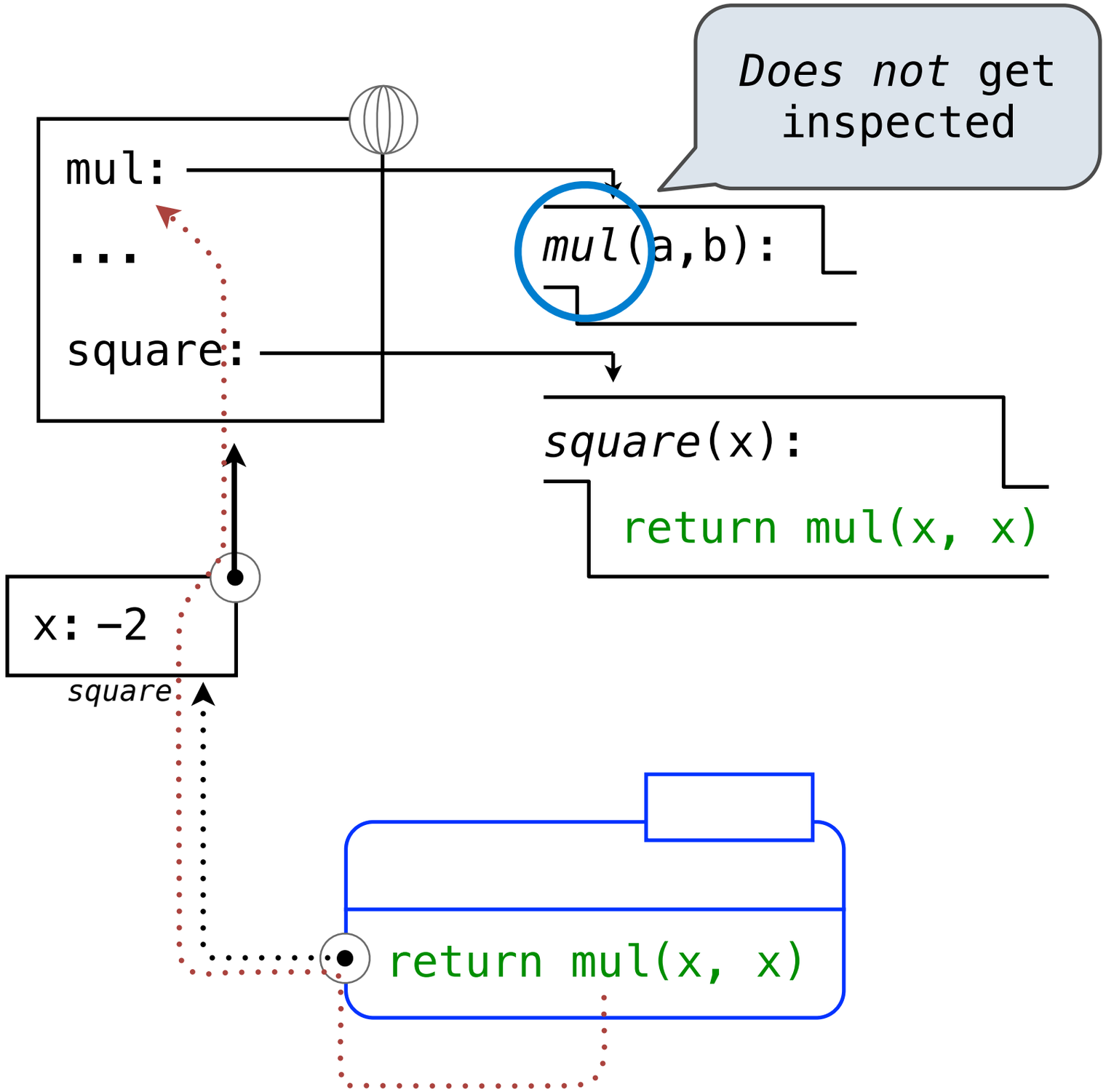
...in the **earliest** frame of the **current environment**

...in which that **name is found**

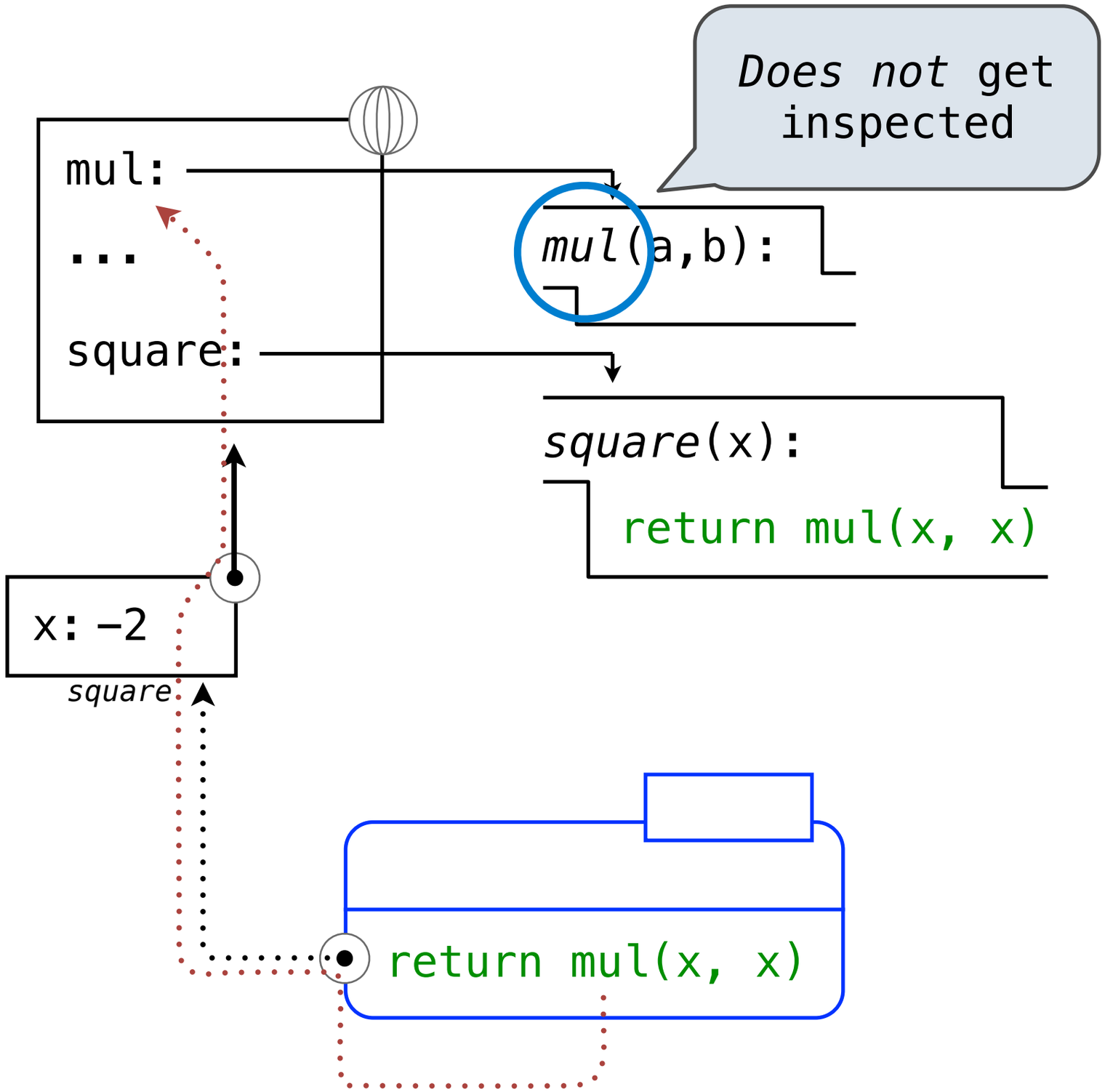
Intrinsic Function Names Don't Play a Role



Intrinsic Function Names Don't Play a Role

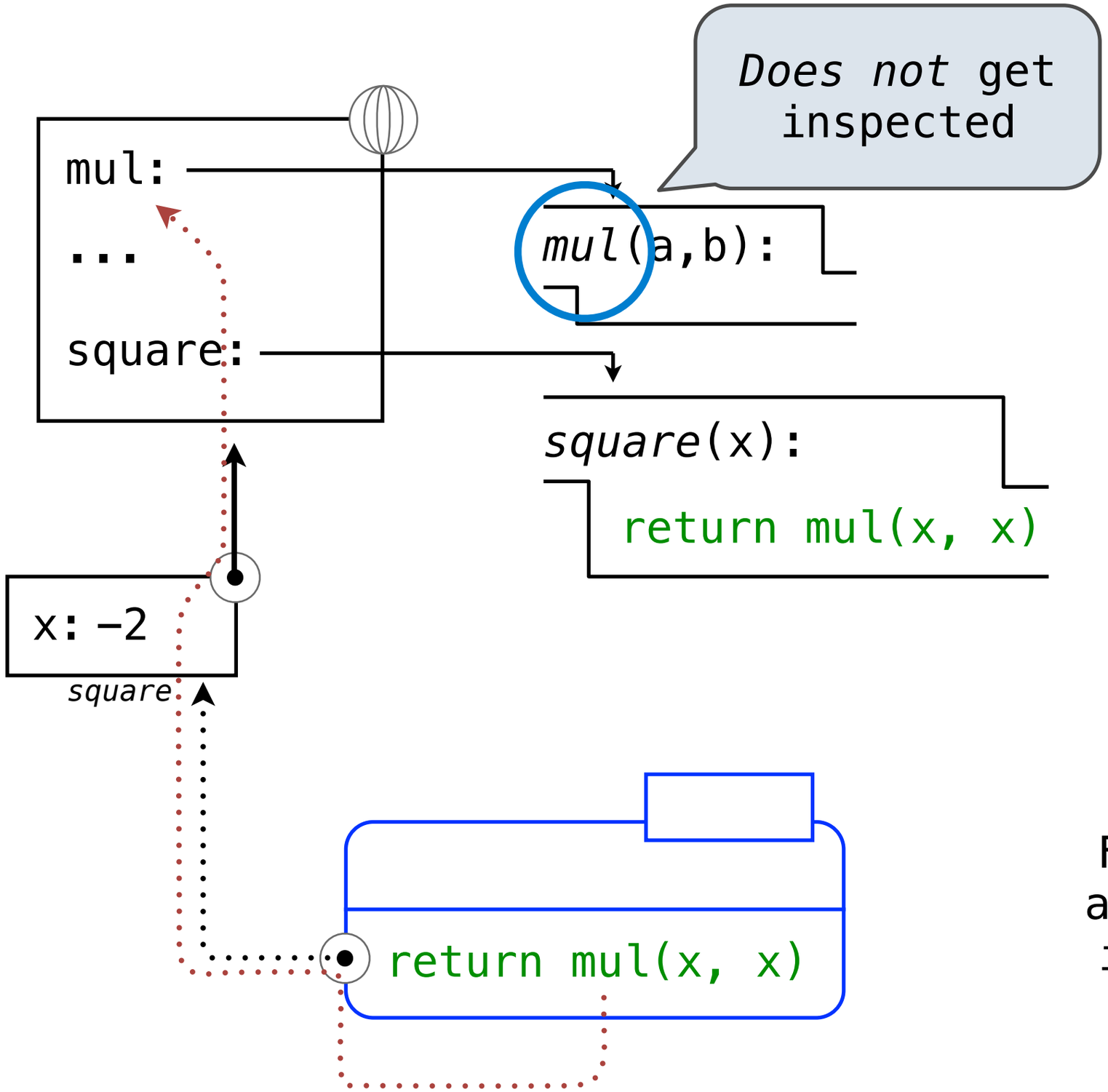


Intrinsic Function Names Don't Play a Role



Witness Protection Program

Intrinsic Function Names Don't Play a Role

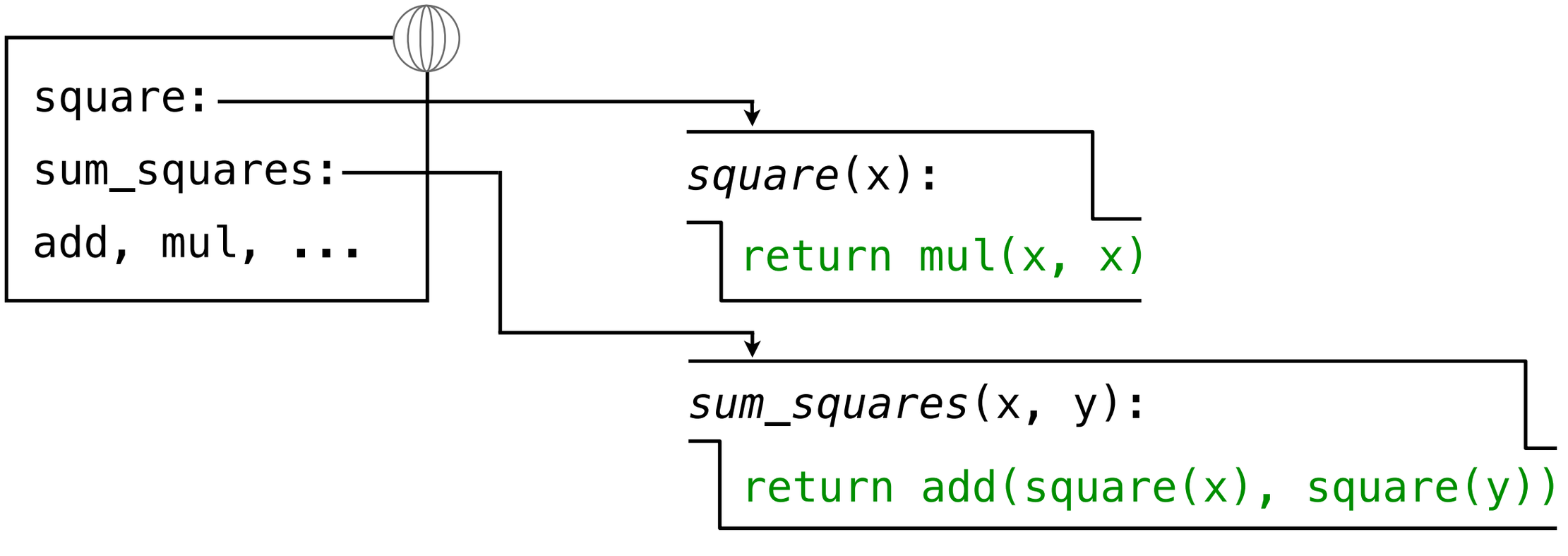


Witness Protection Program

Functions aren't accessed by their intrinsic names!

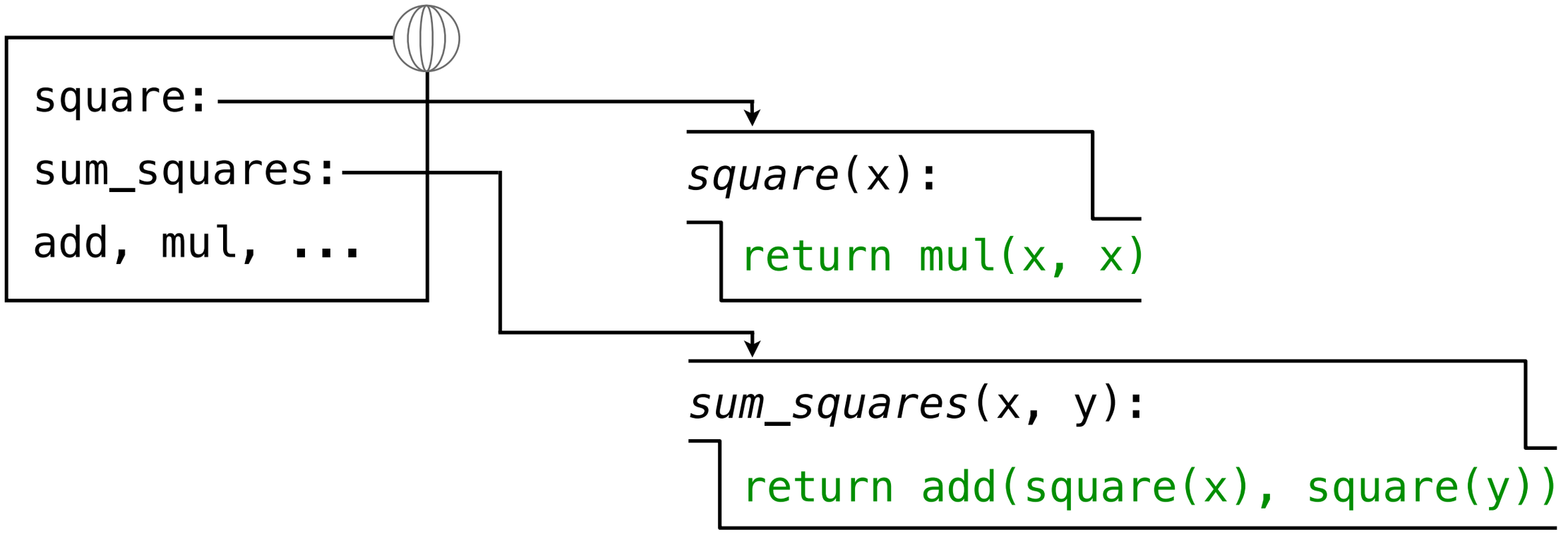
(Demo)

Example: Function Application



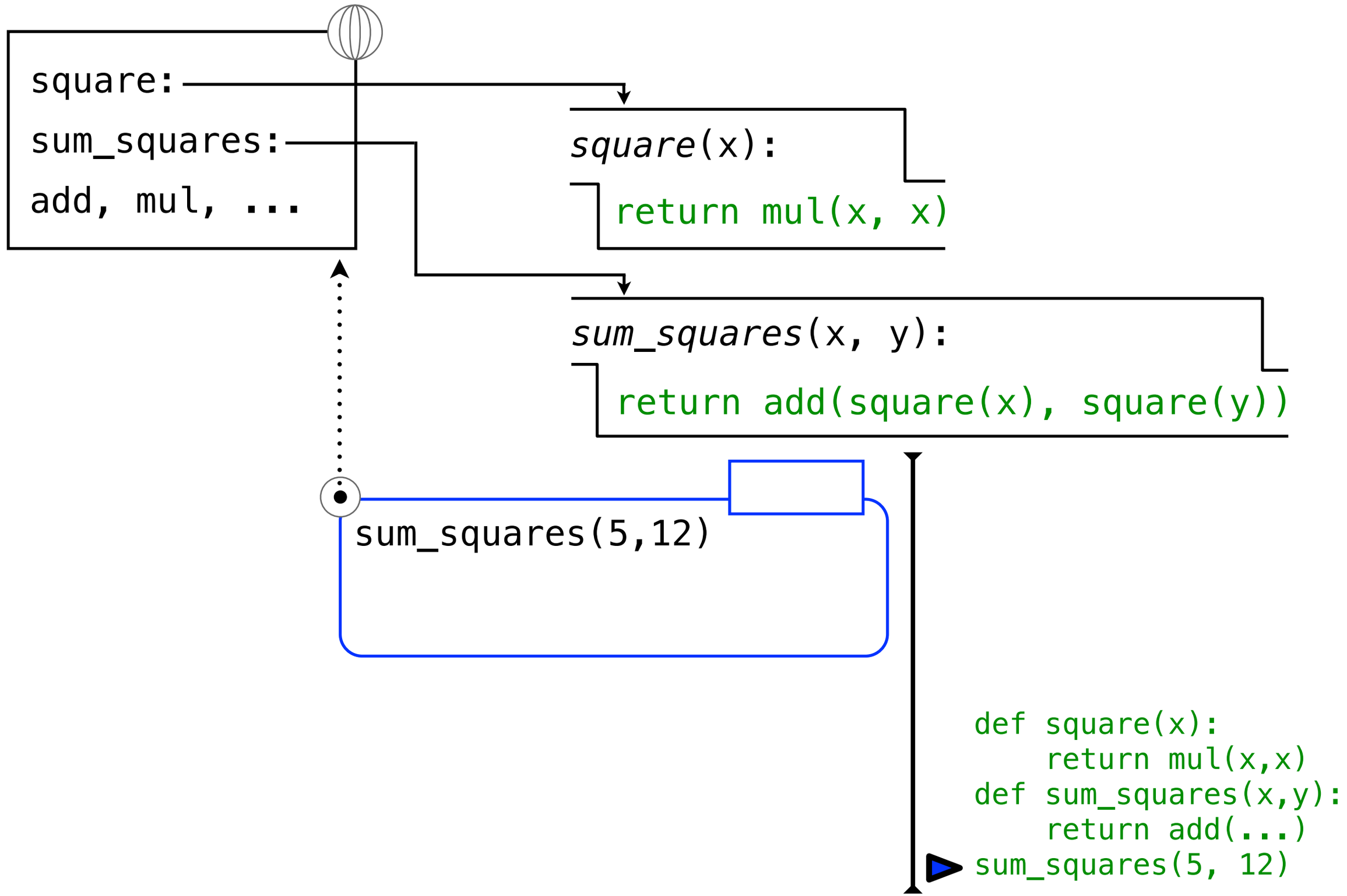

```
def square(x):
    return mul(x,x)
def sum_squares(x, y):
    return add(square(x),square(y))
```

Example: Function Application

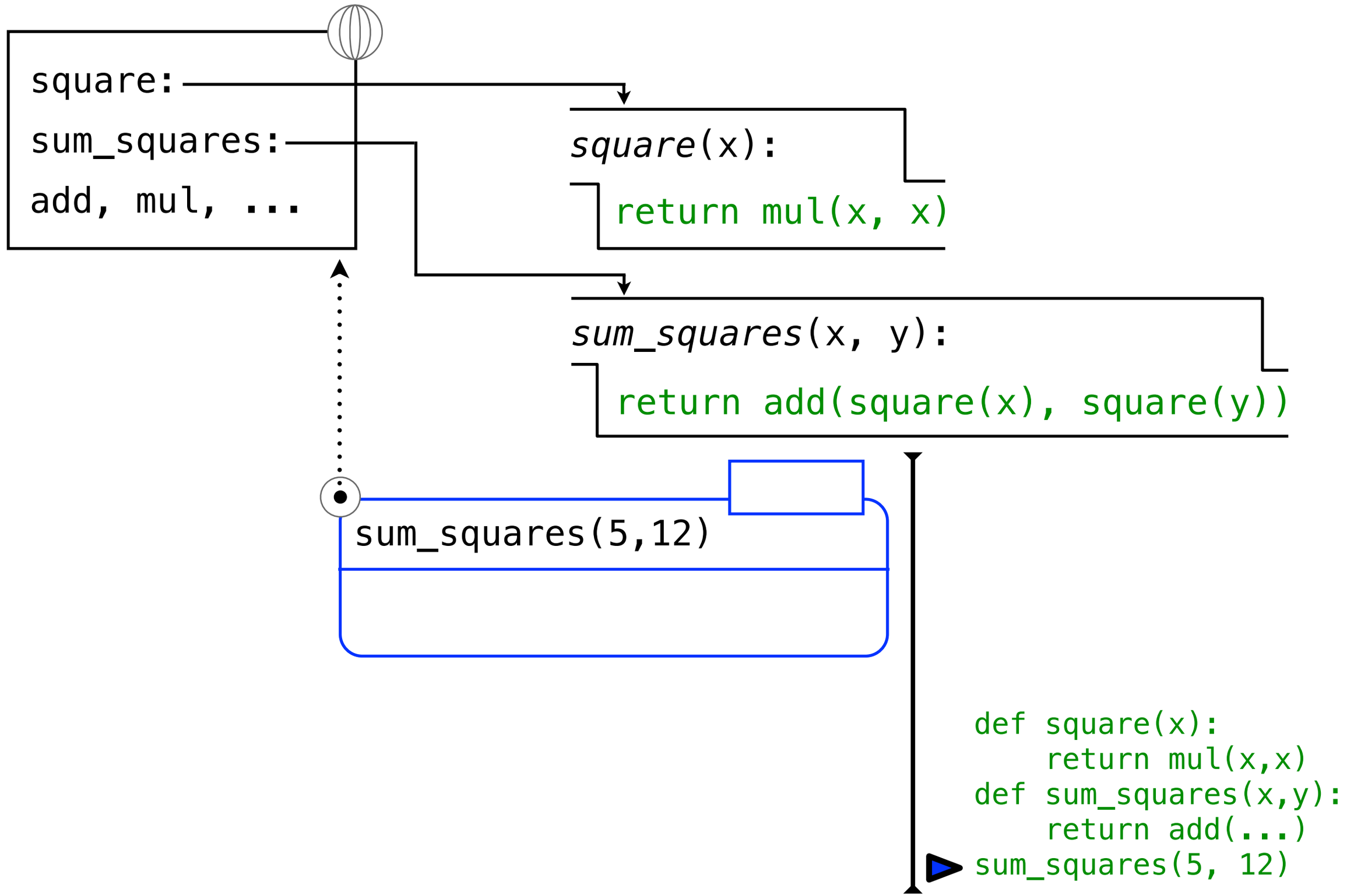



```
def square(x):
    return mul(x,x)
def sum_squares(x,y):
    return add(...)
sum_squares(5, 12)
```

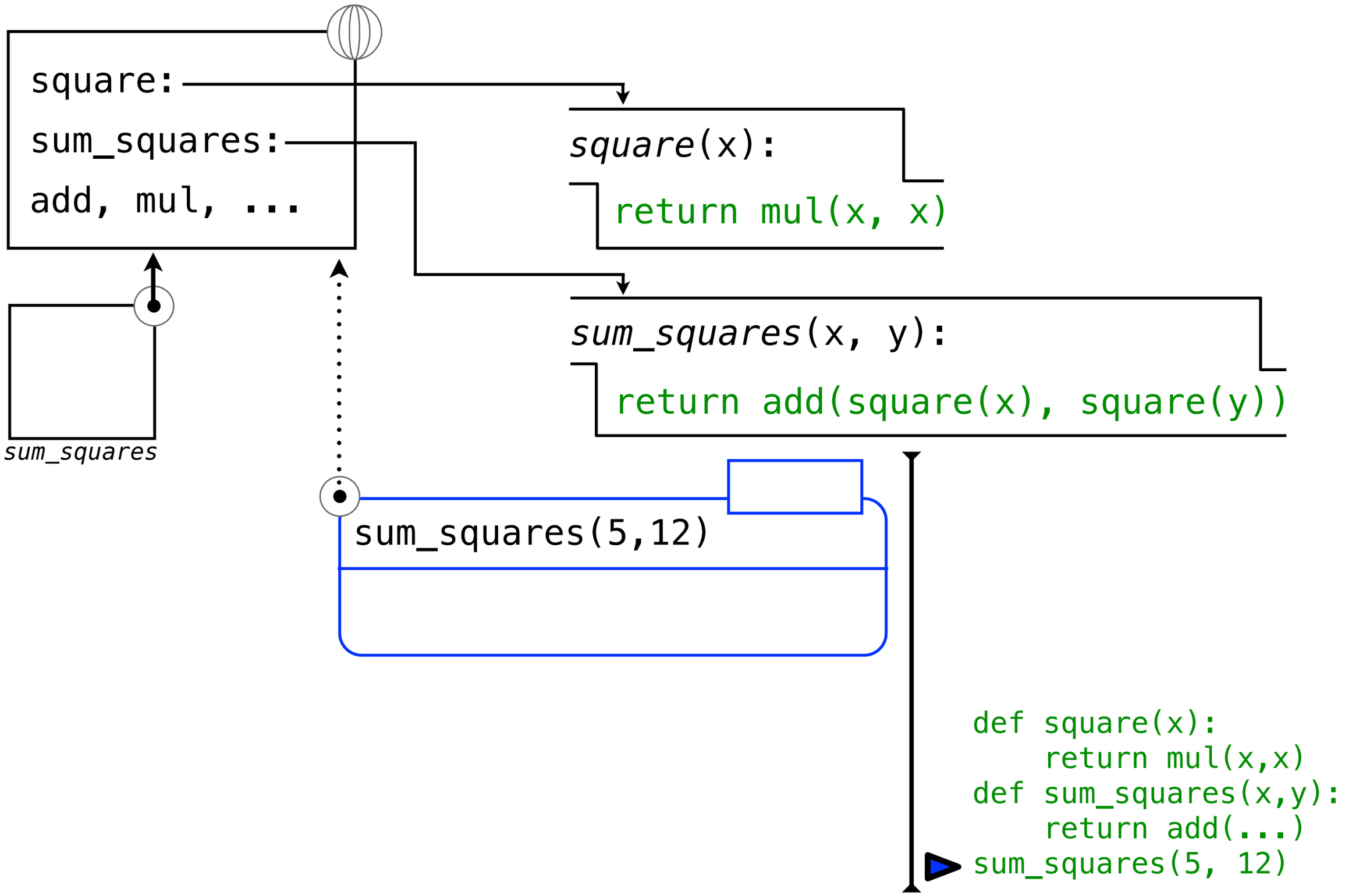
Example: Function Application



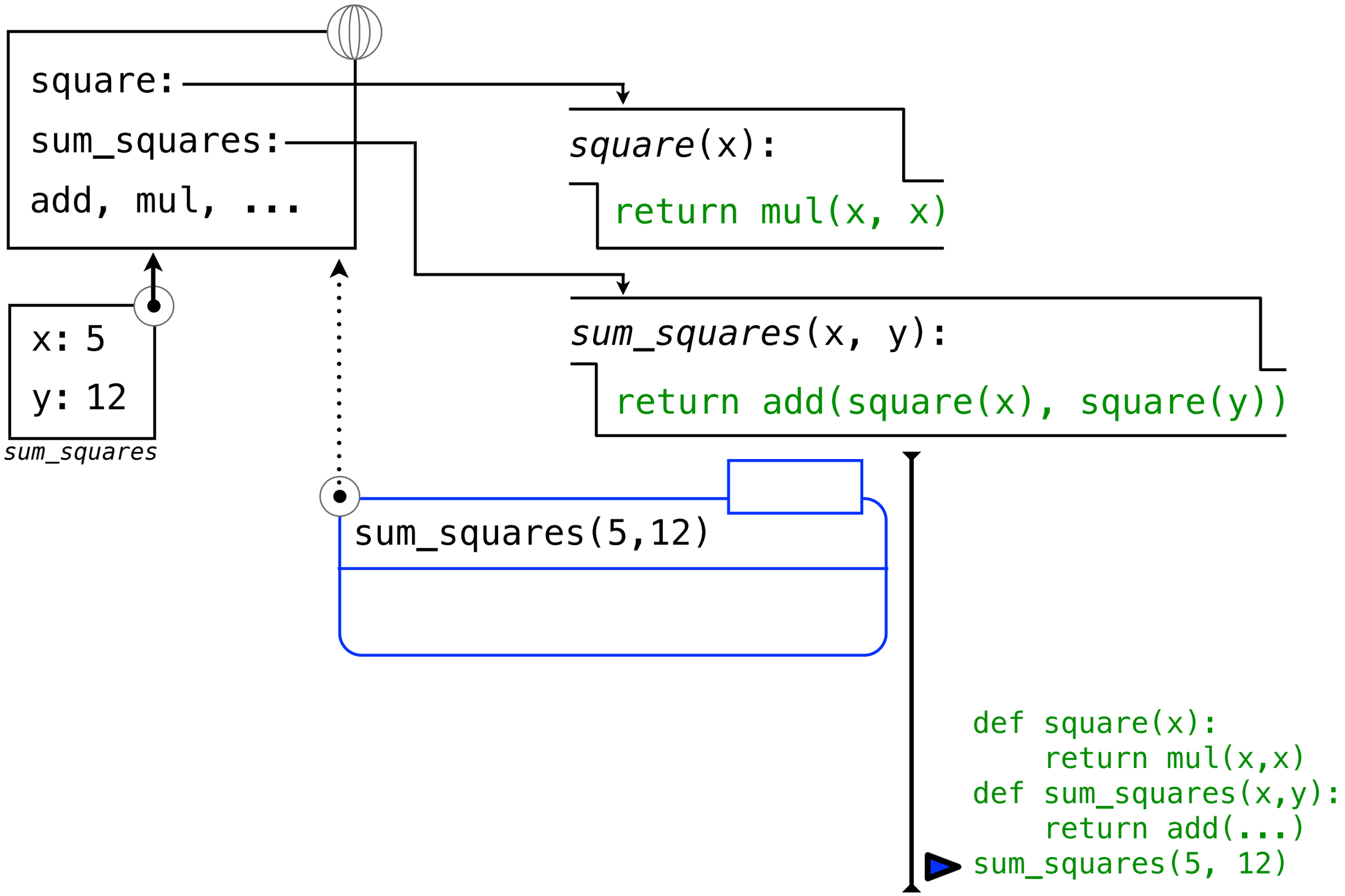
Example: Function Application



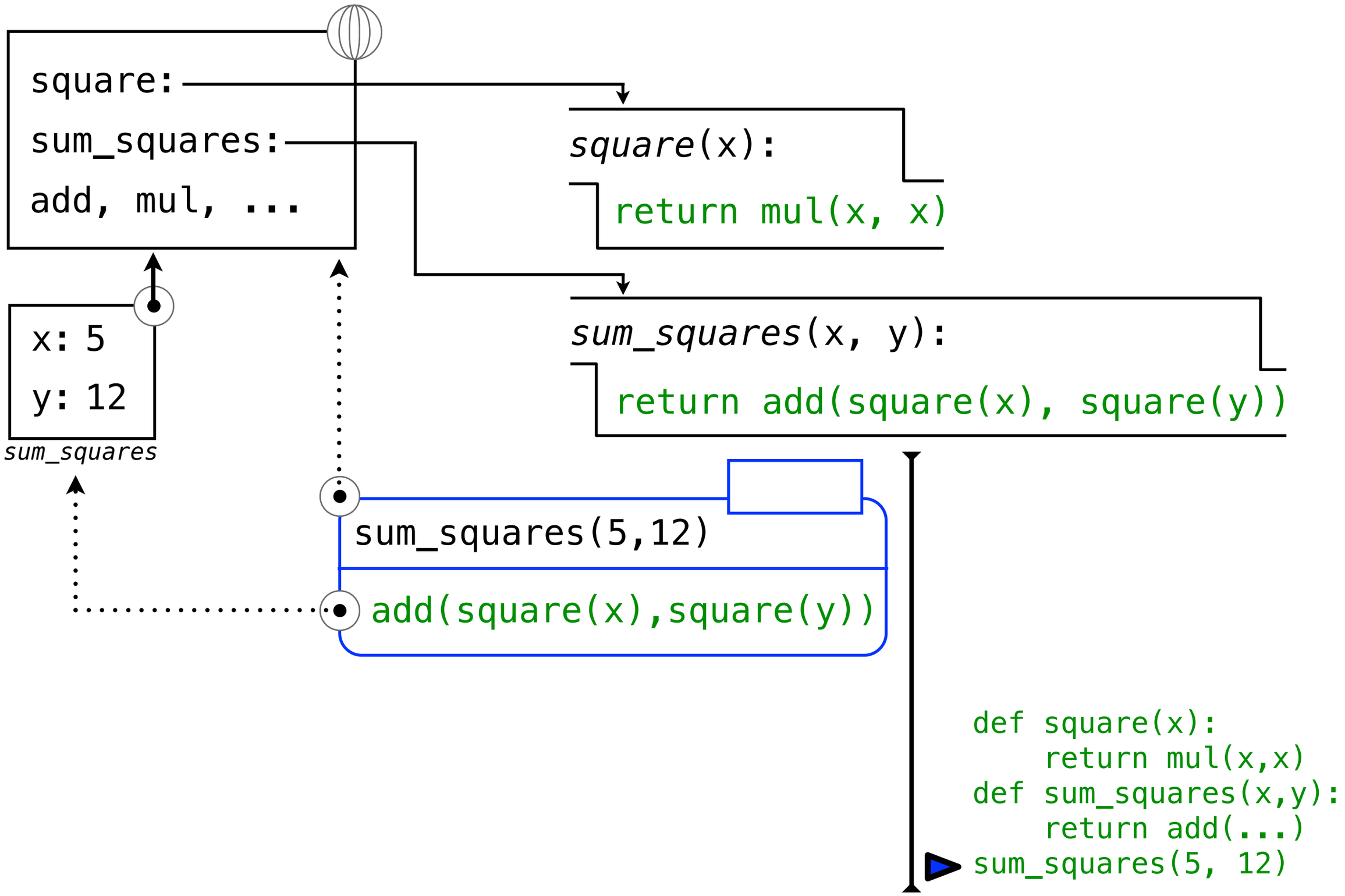
Example: Function Application



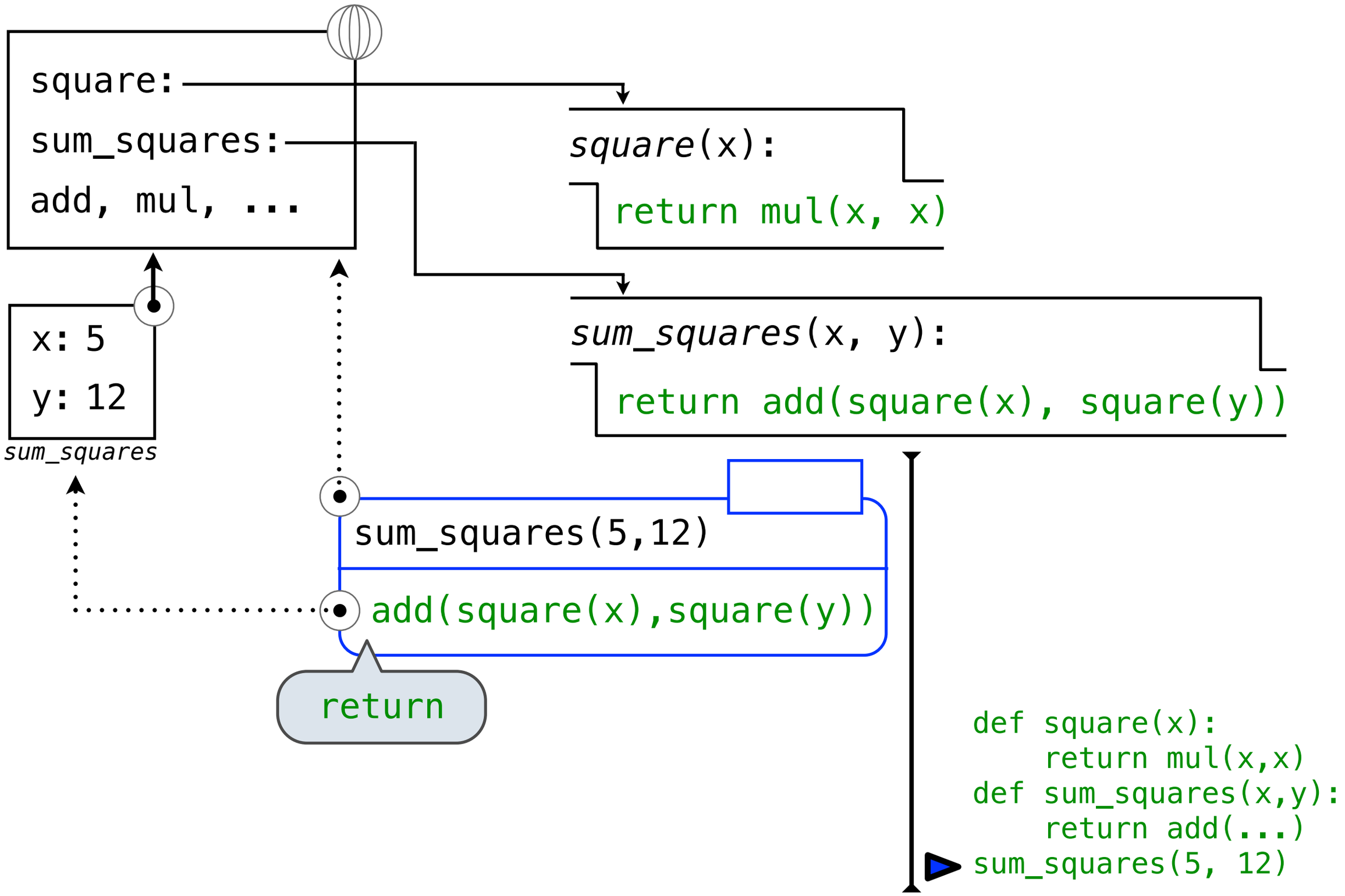
Example: Function Application



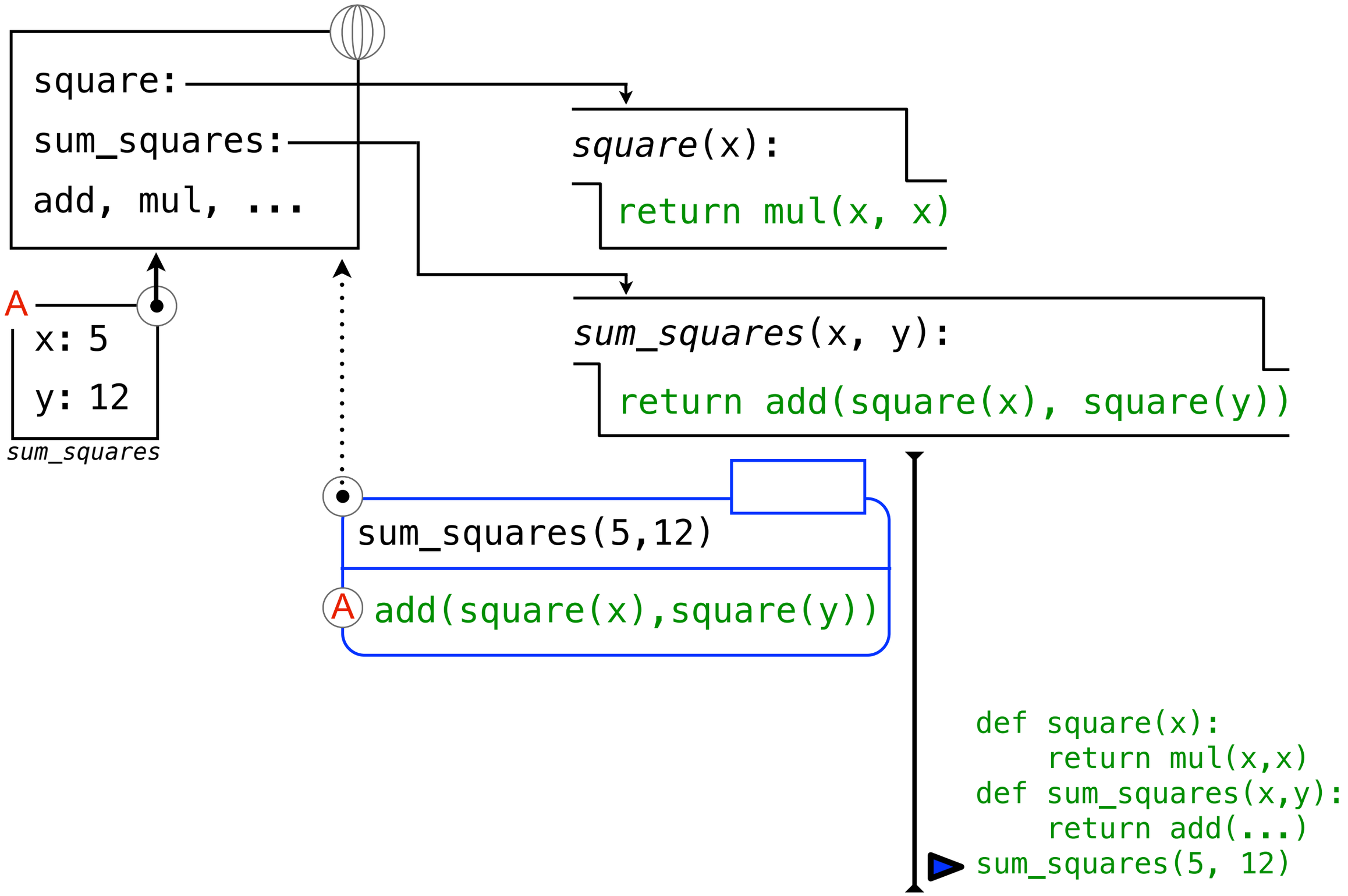
Example: Function Application



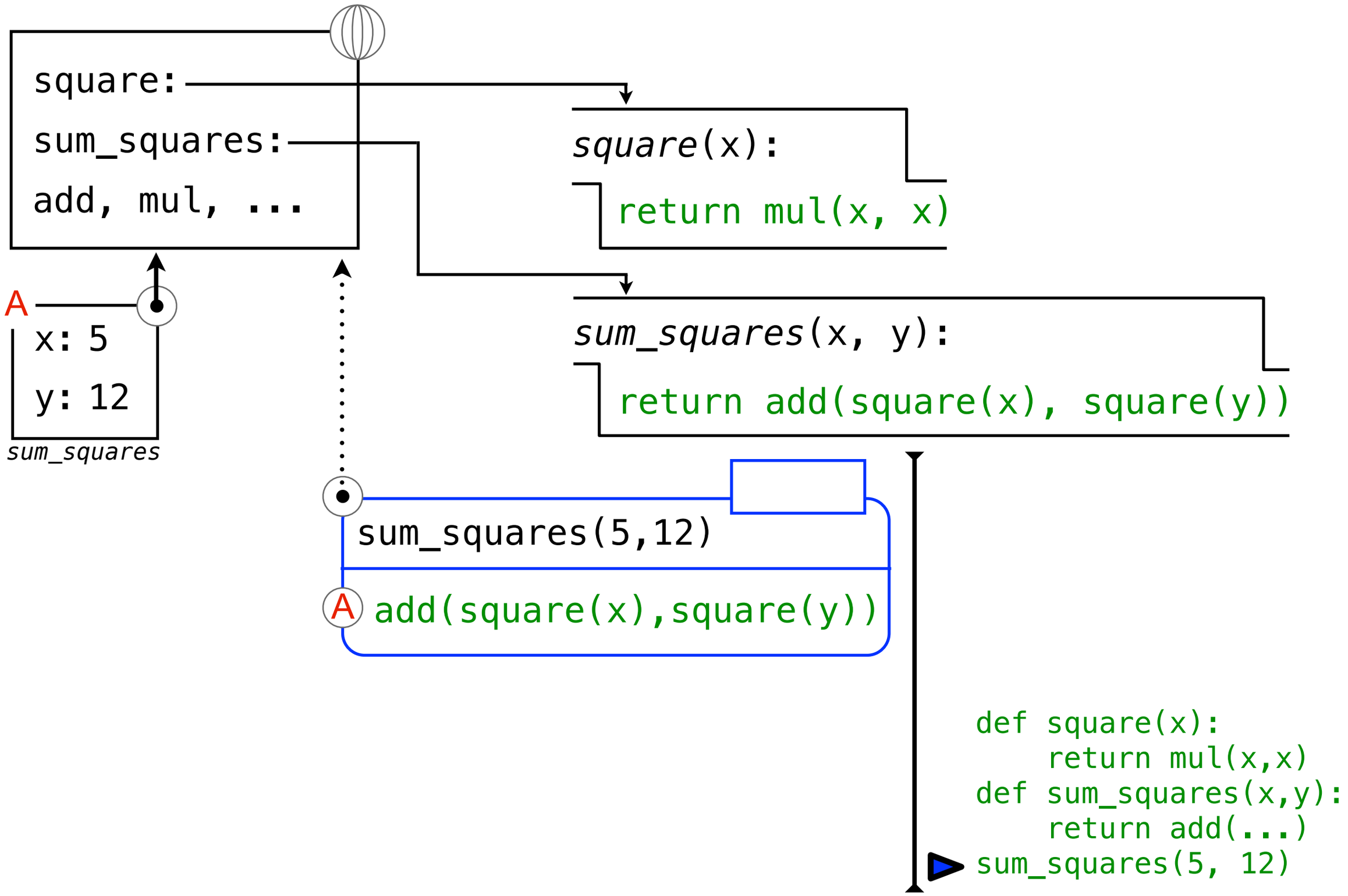
Example: Function Application



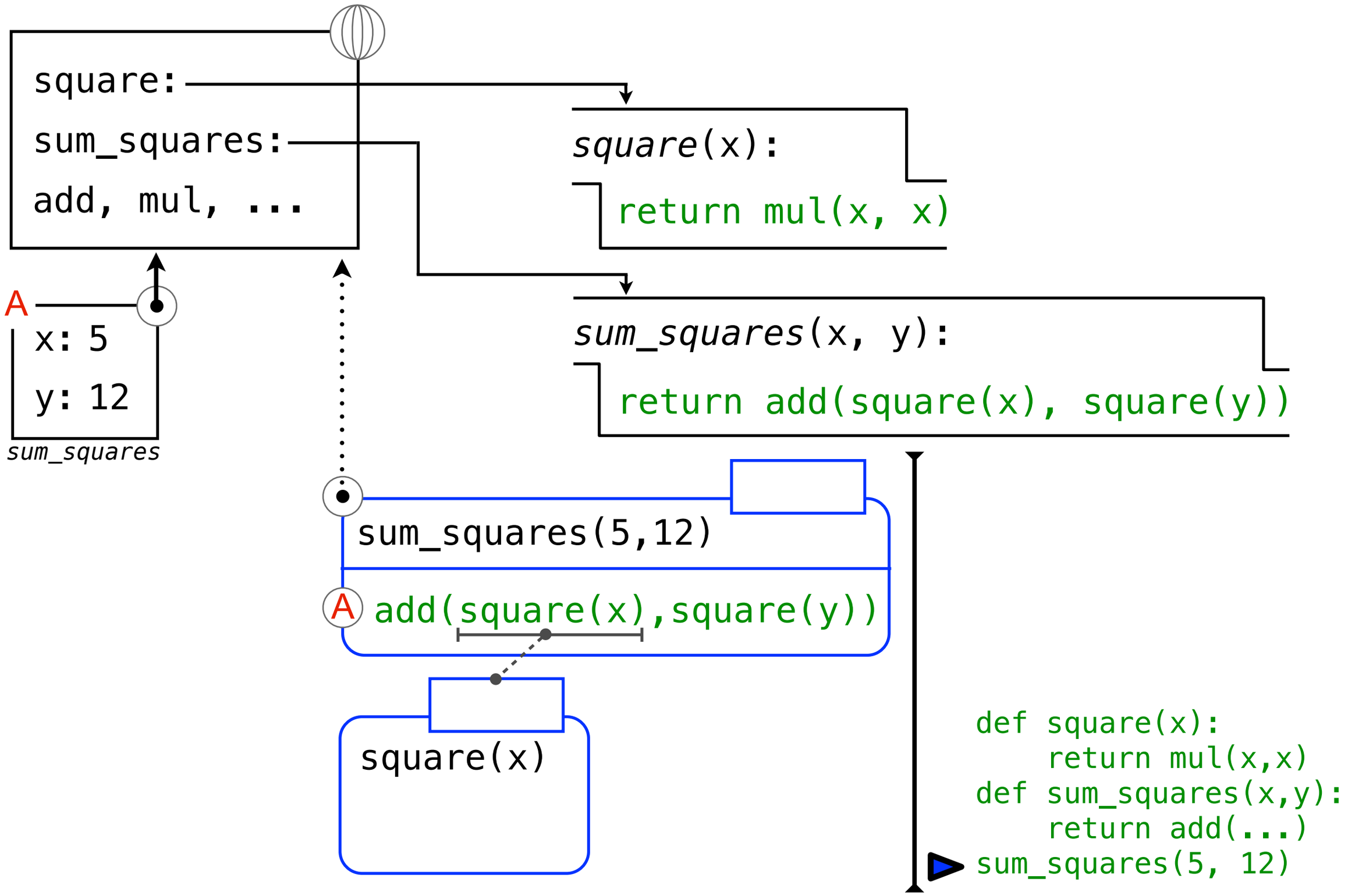
Example: Function Application



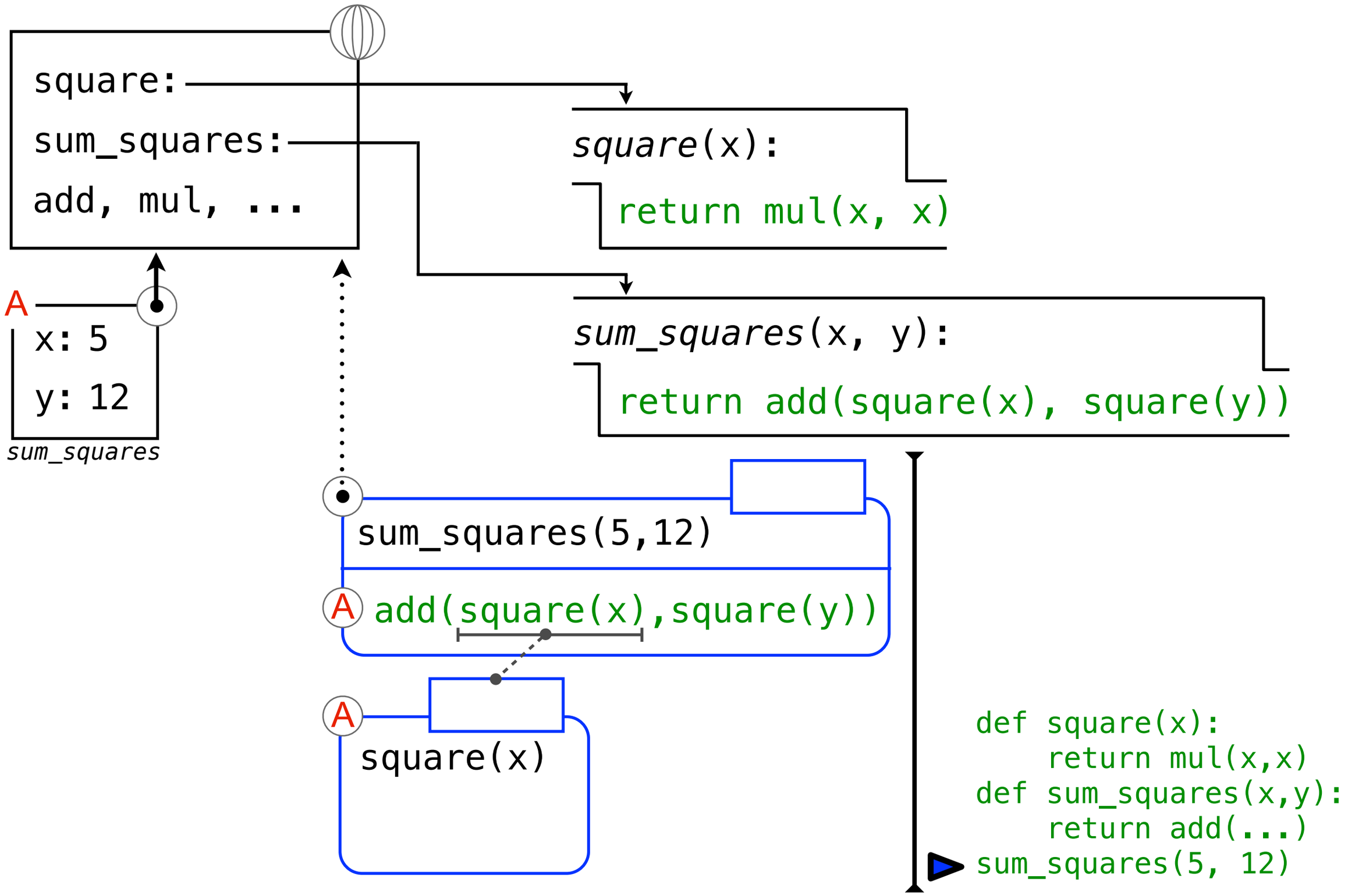
Example: Function Application



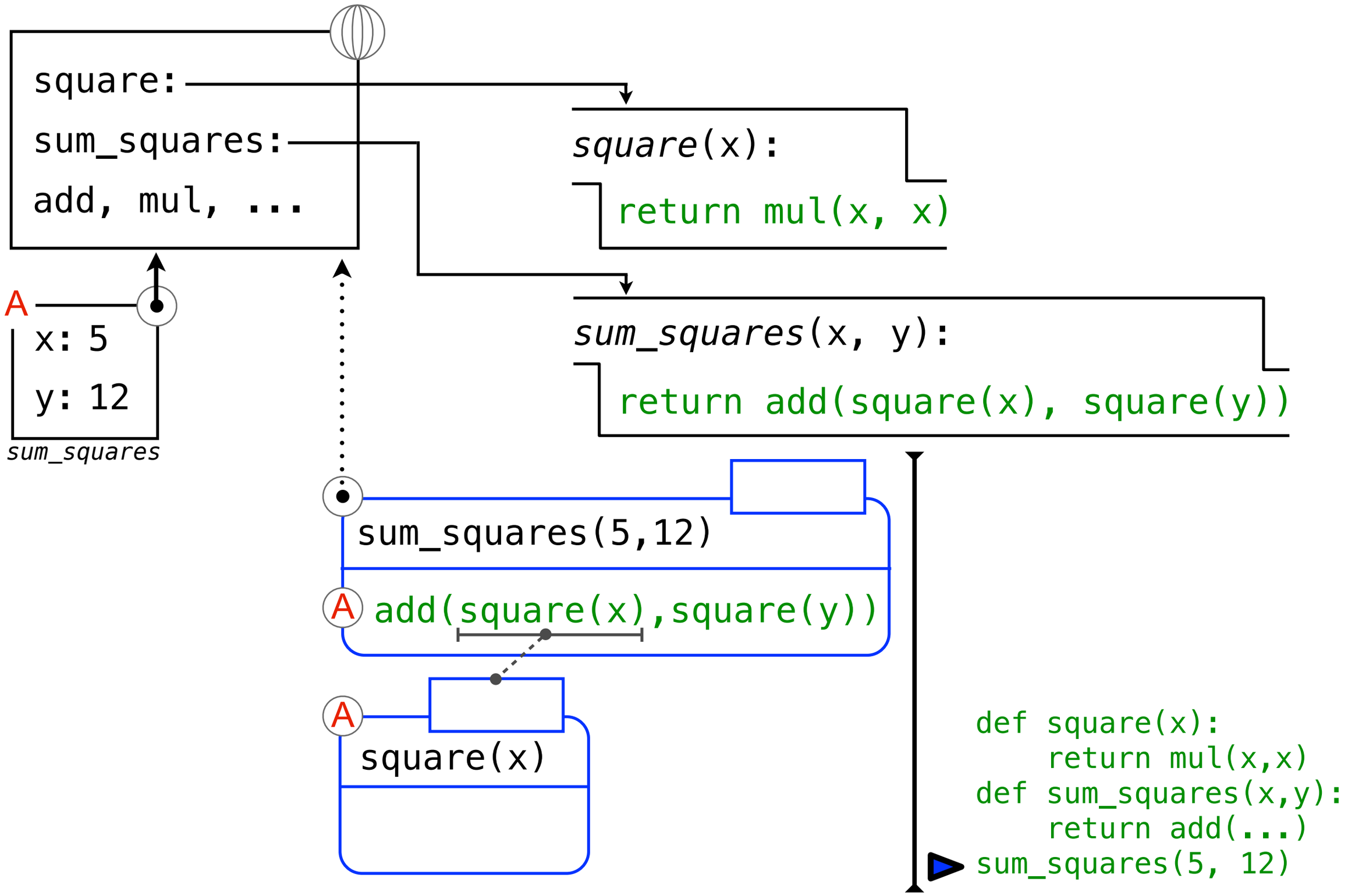
Example: Function Application



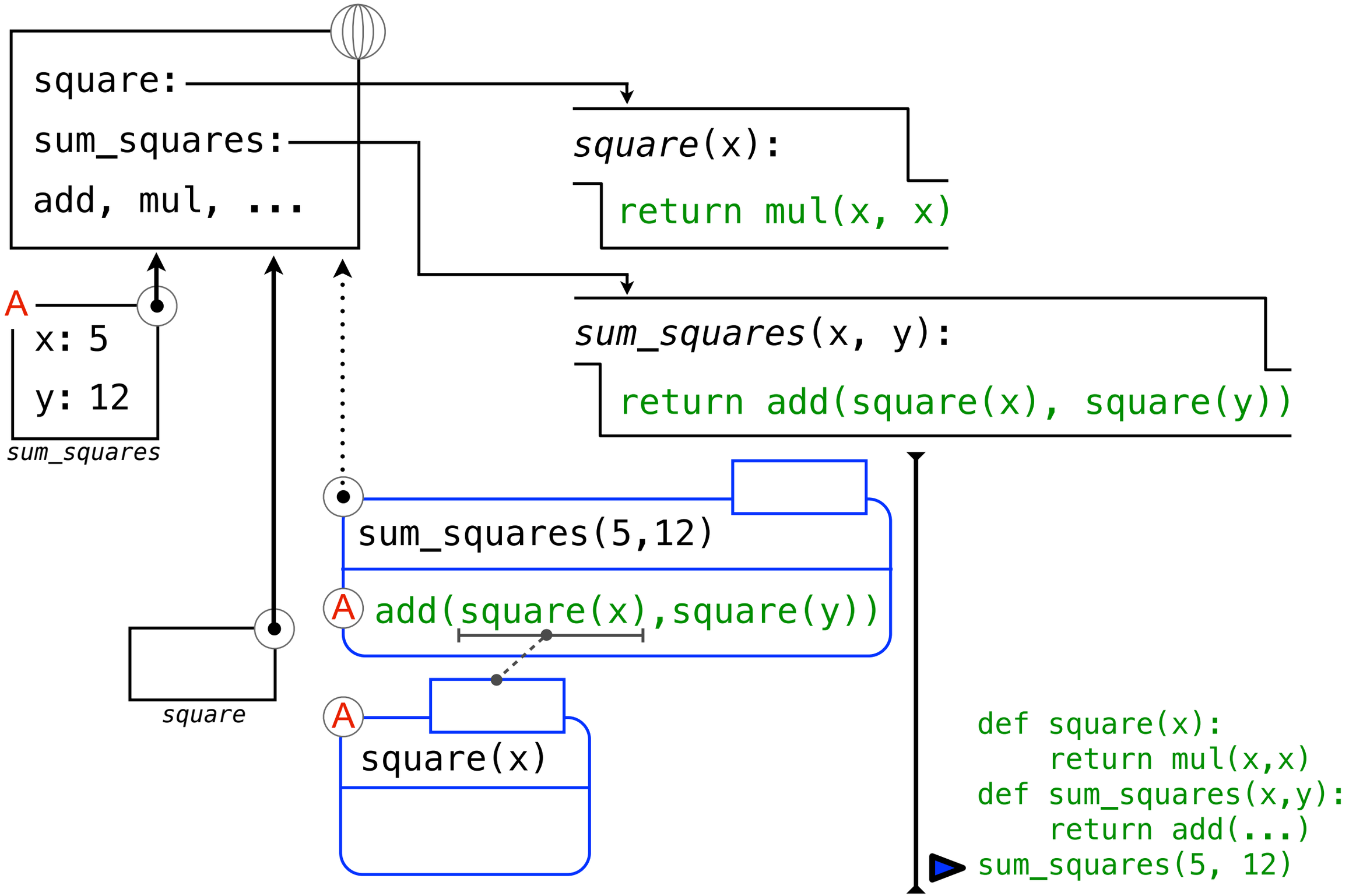
Example: Function Application



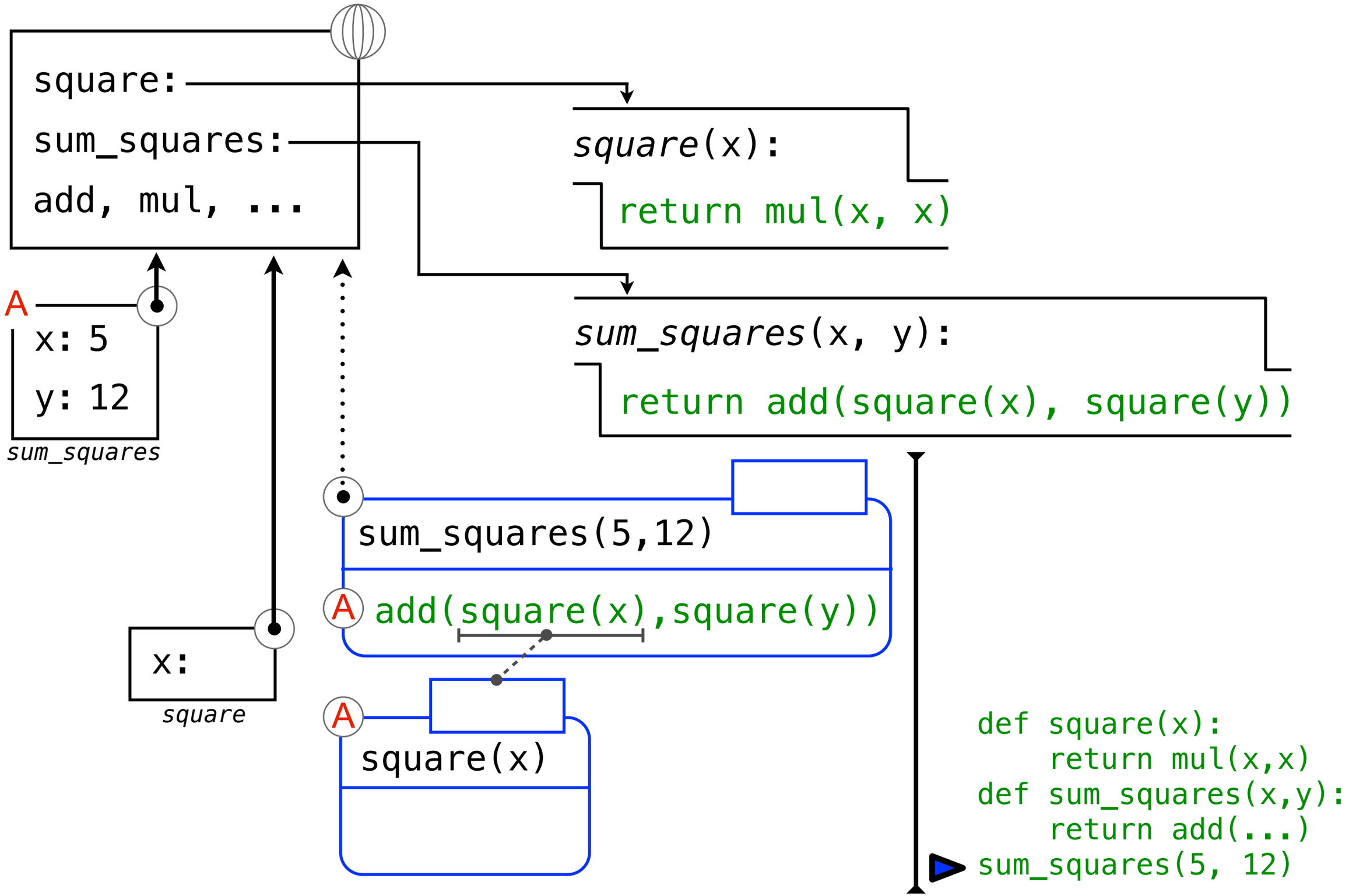
Example: Function Application



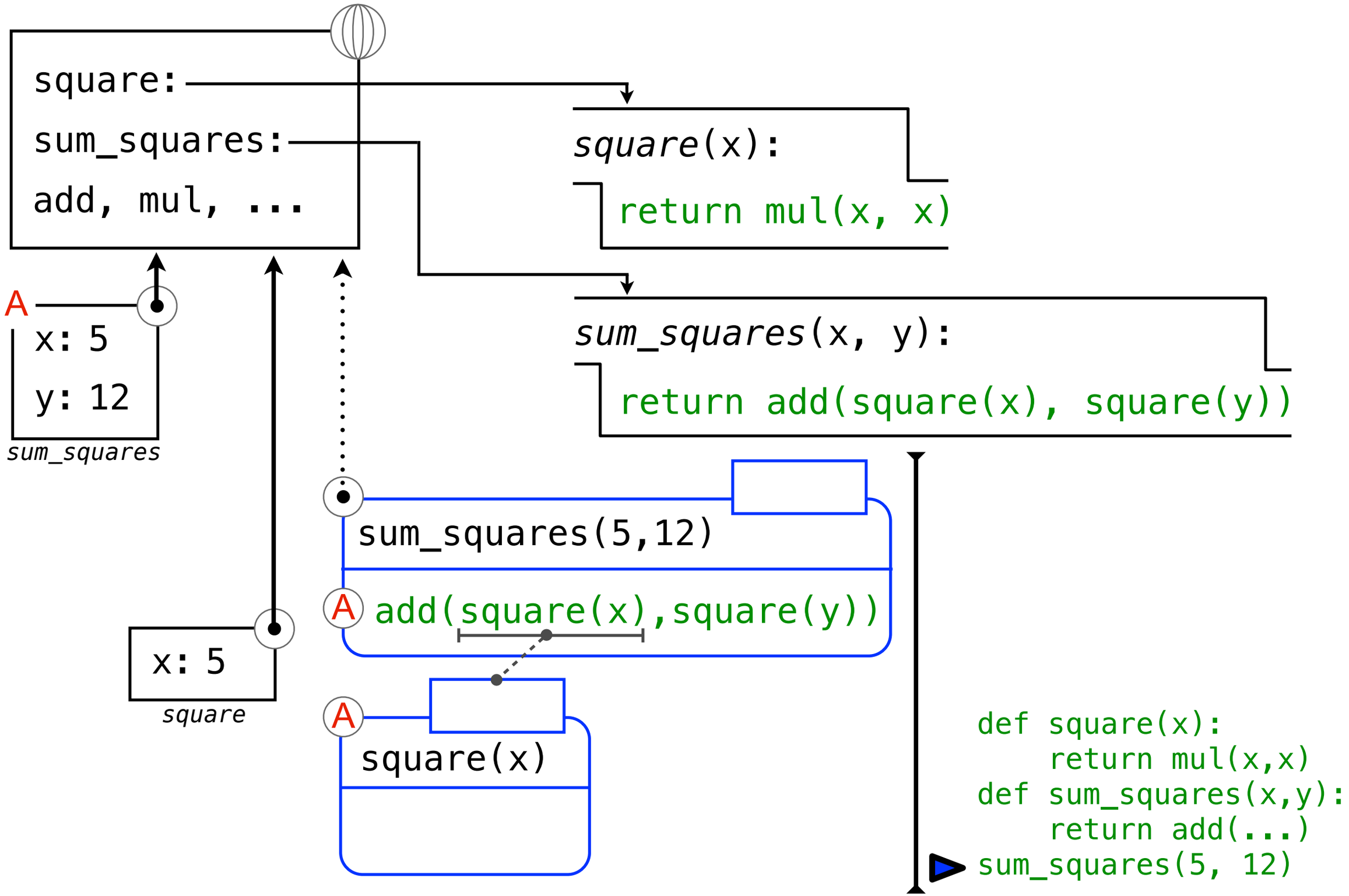
Example: Function Application



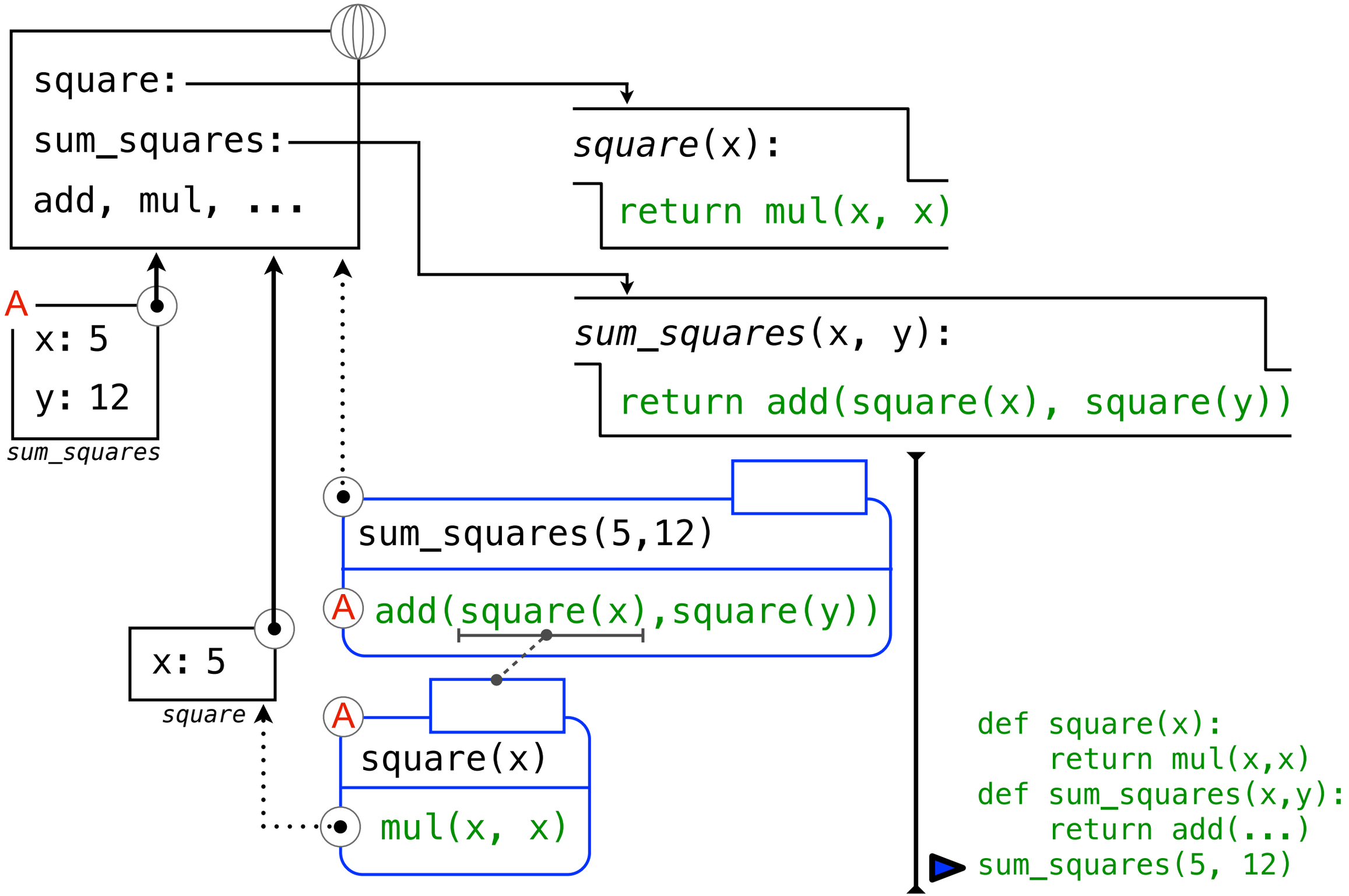
Example: Function Application



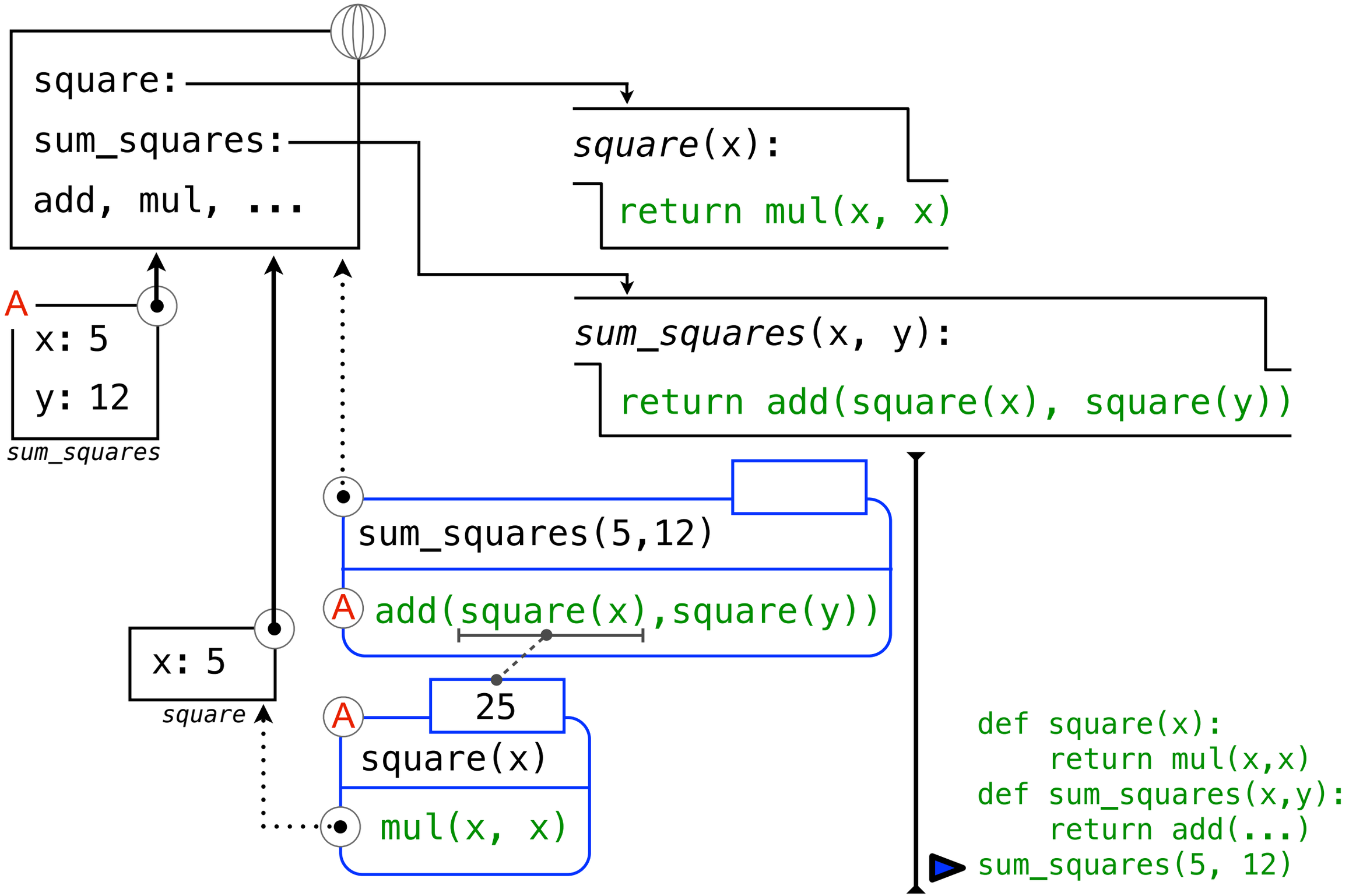
Example: Function Application



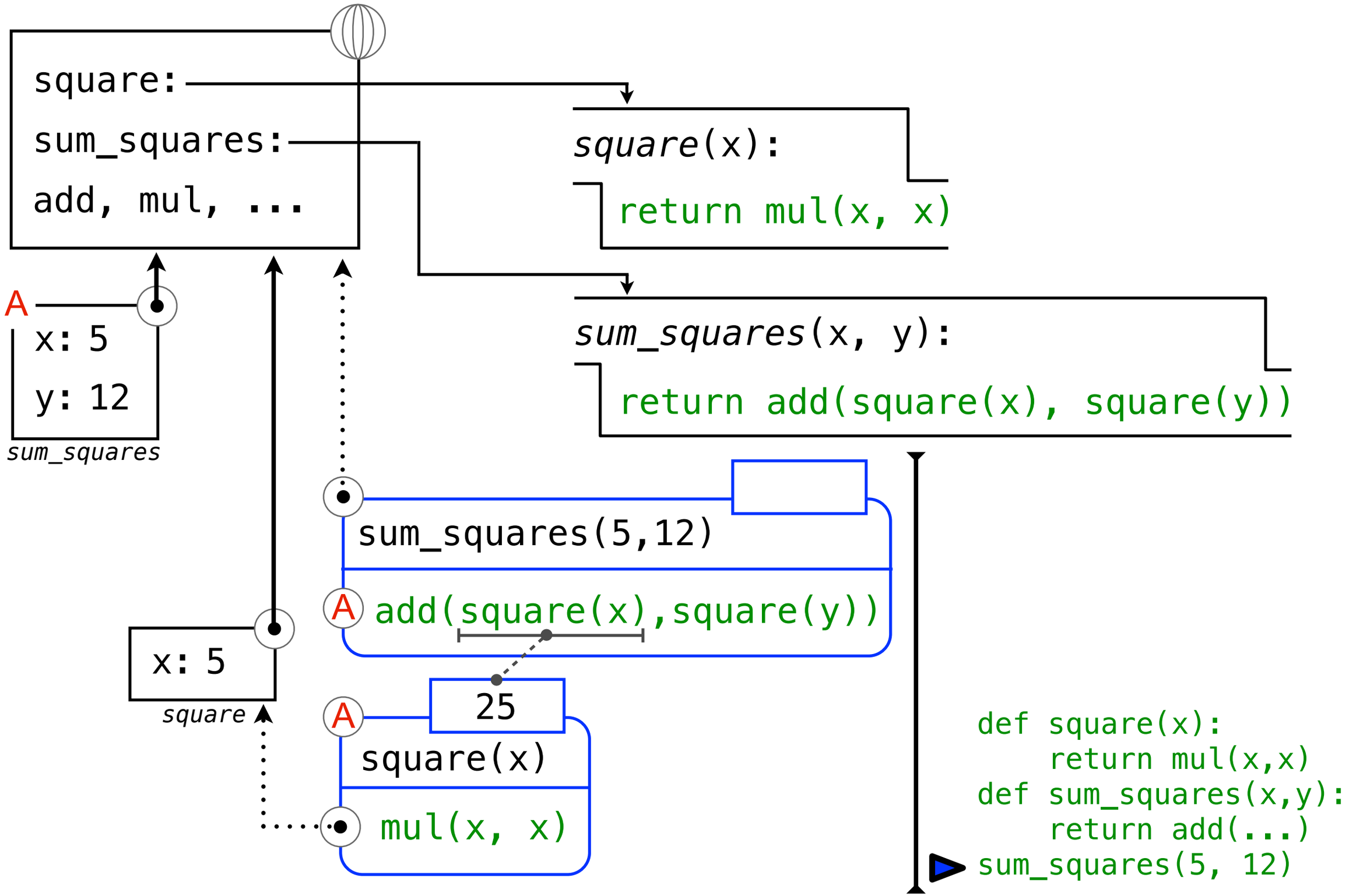
Example: Function Application



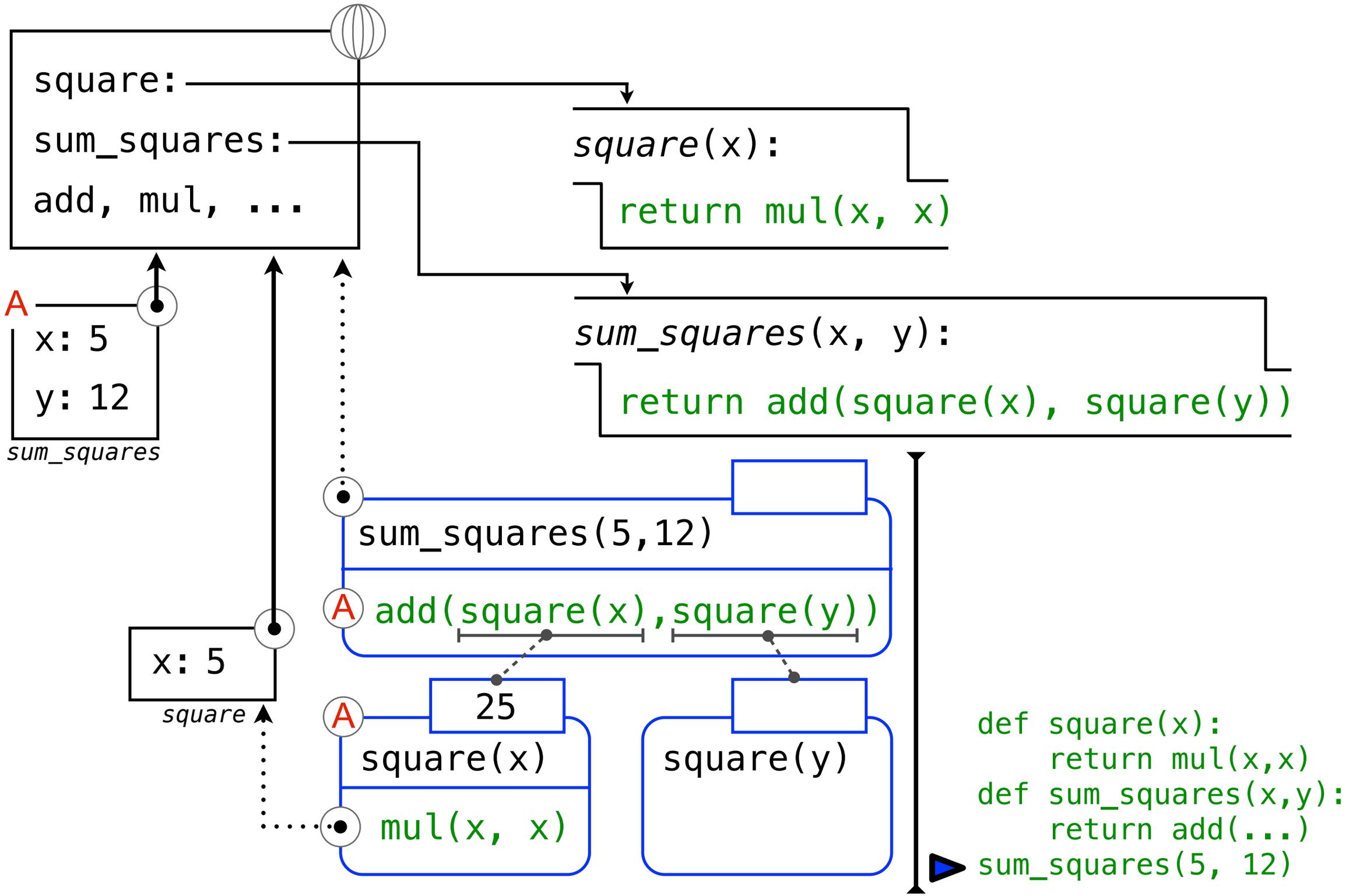
Example: Function Application



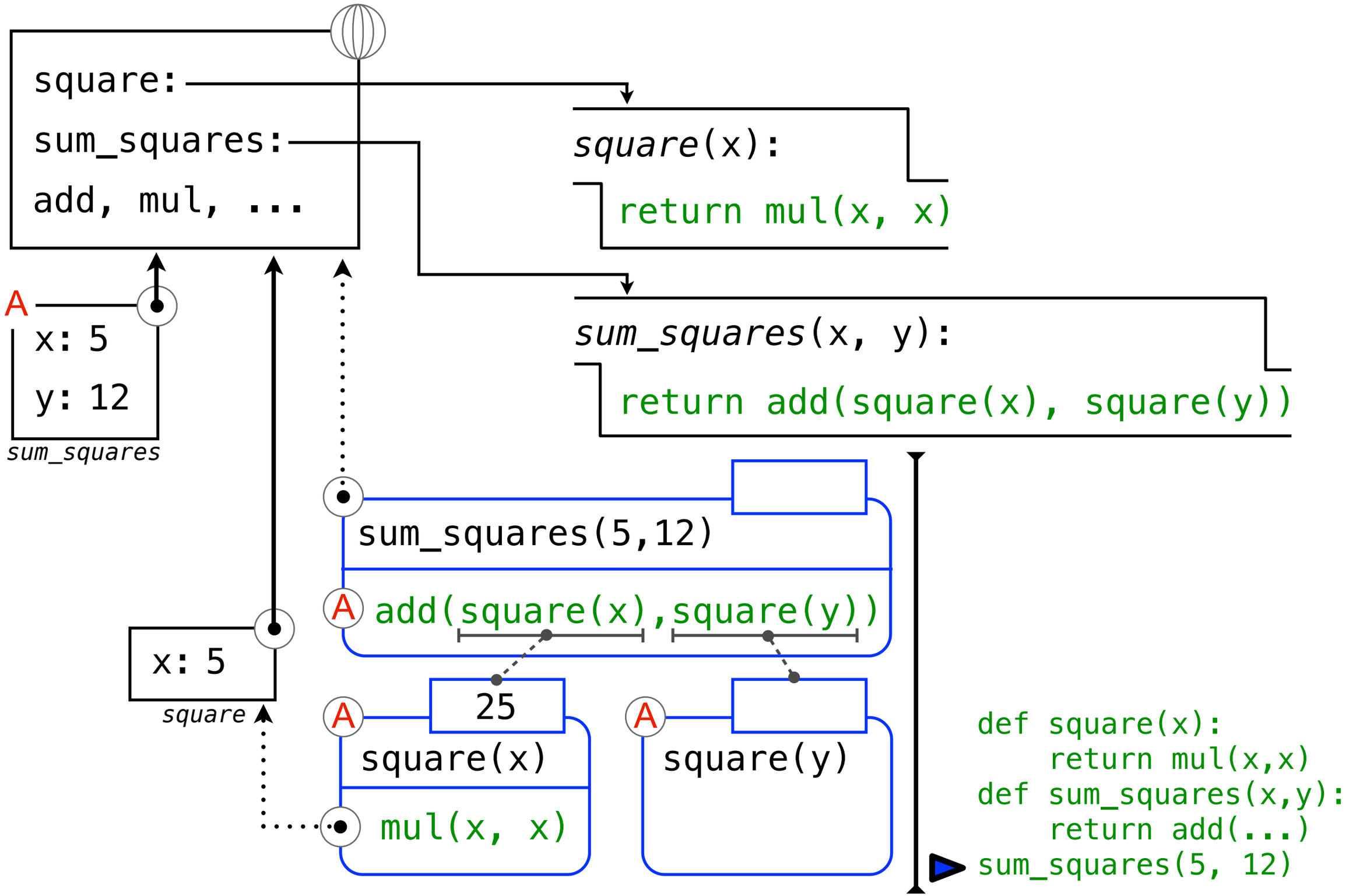
Example: Function Application



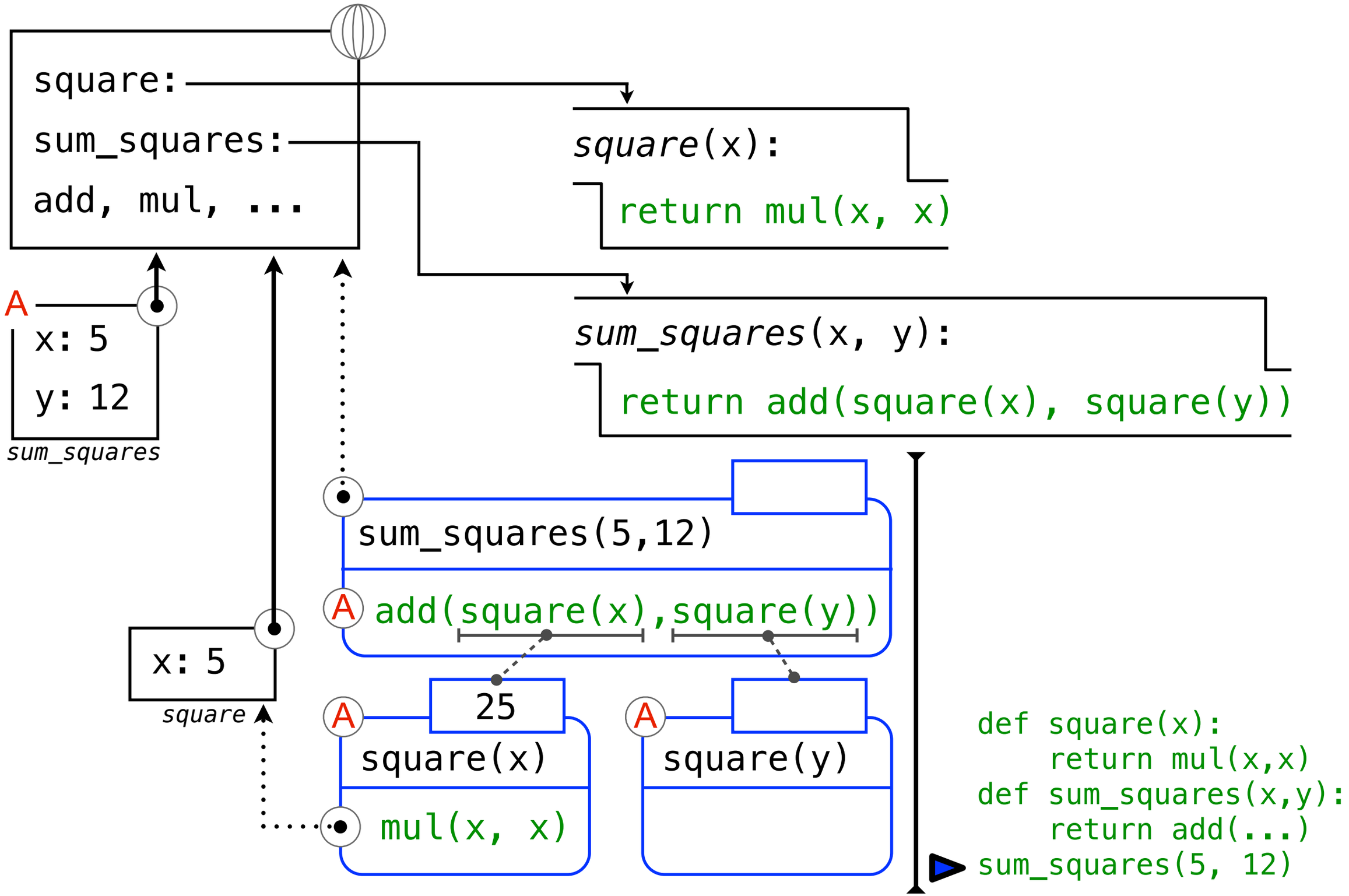
Example: Function Application



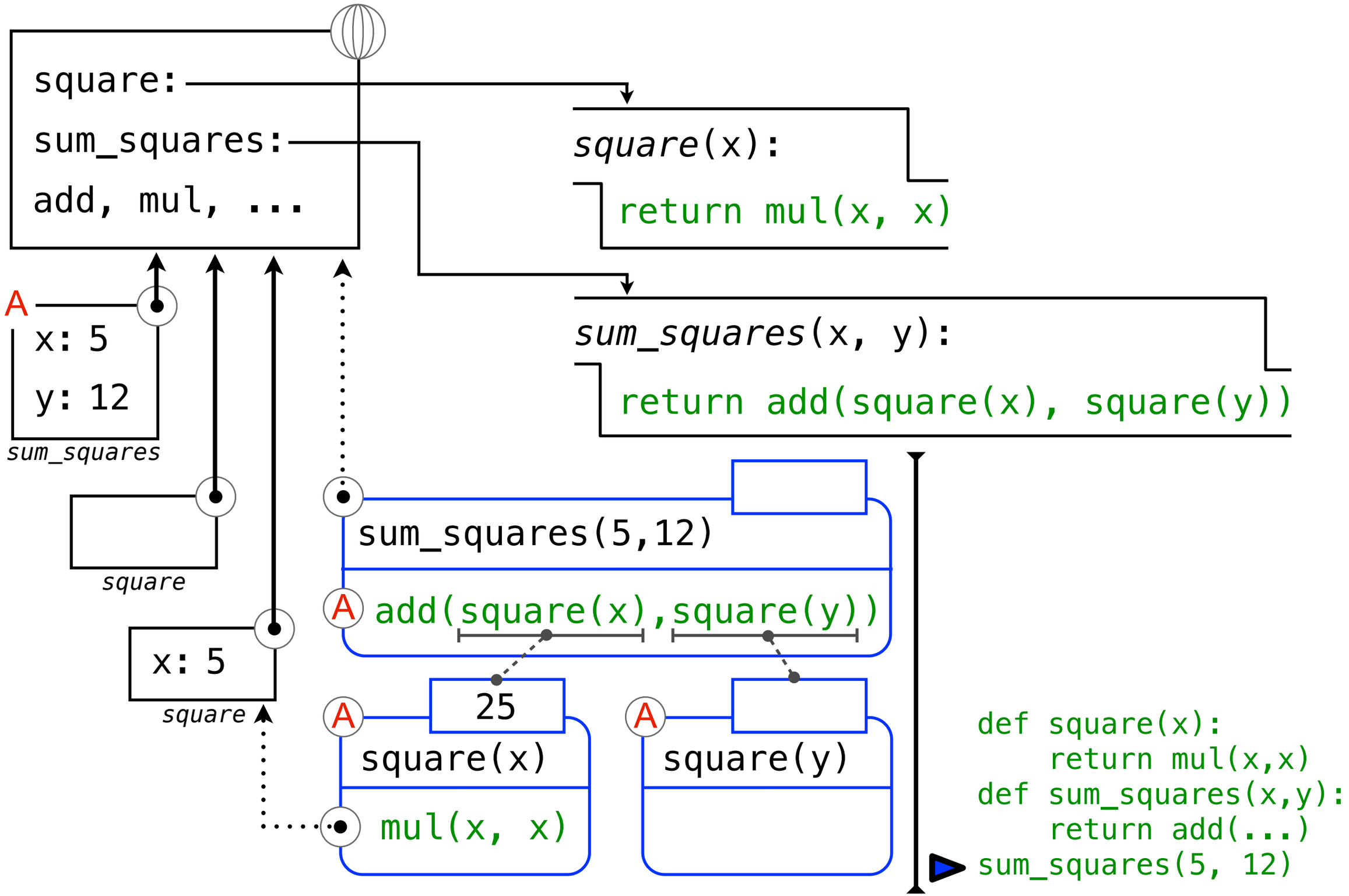
Example: Function Application



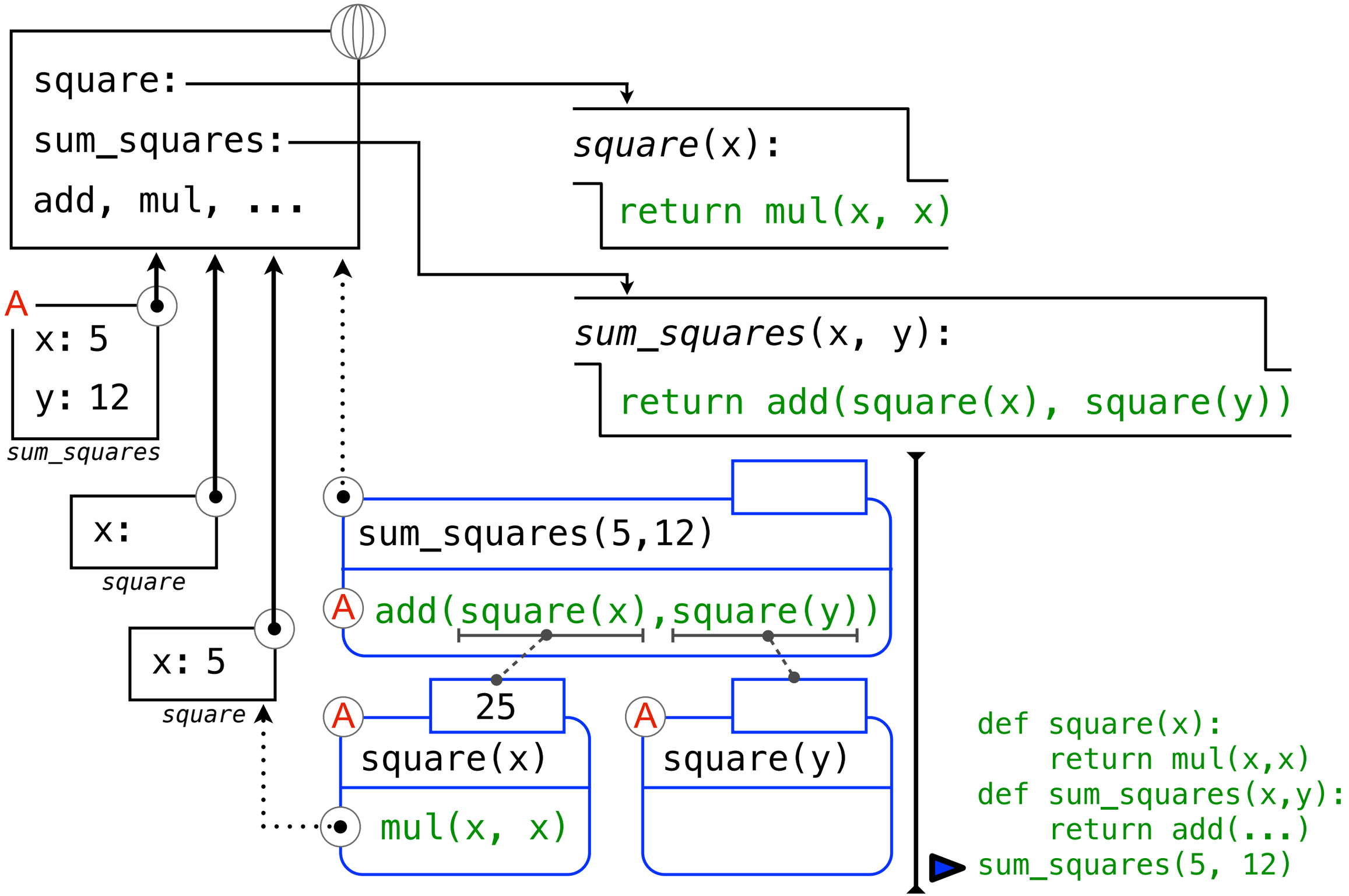
Example: Function Application



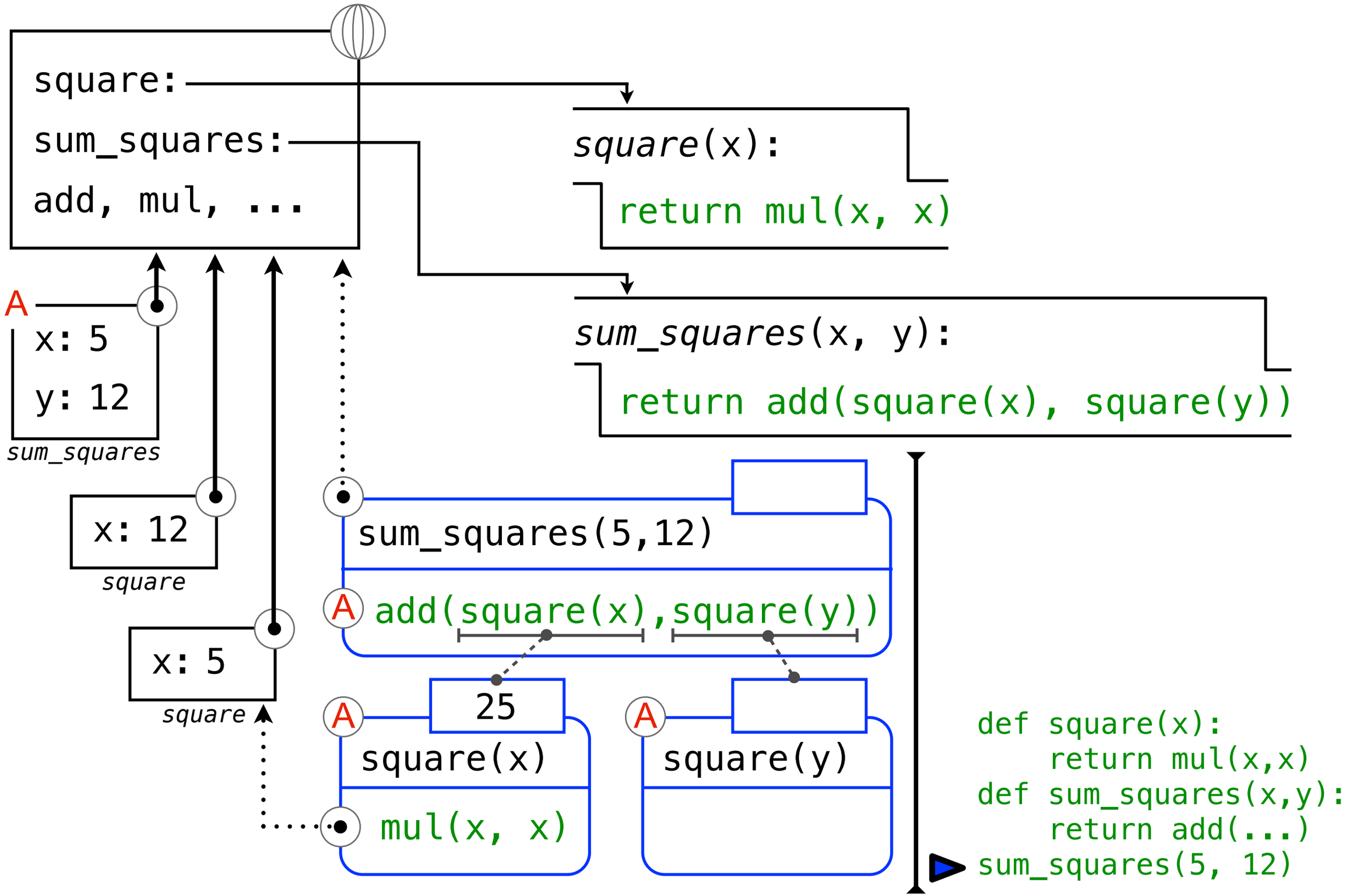
Example: Function Application



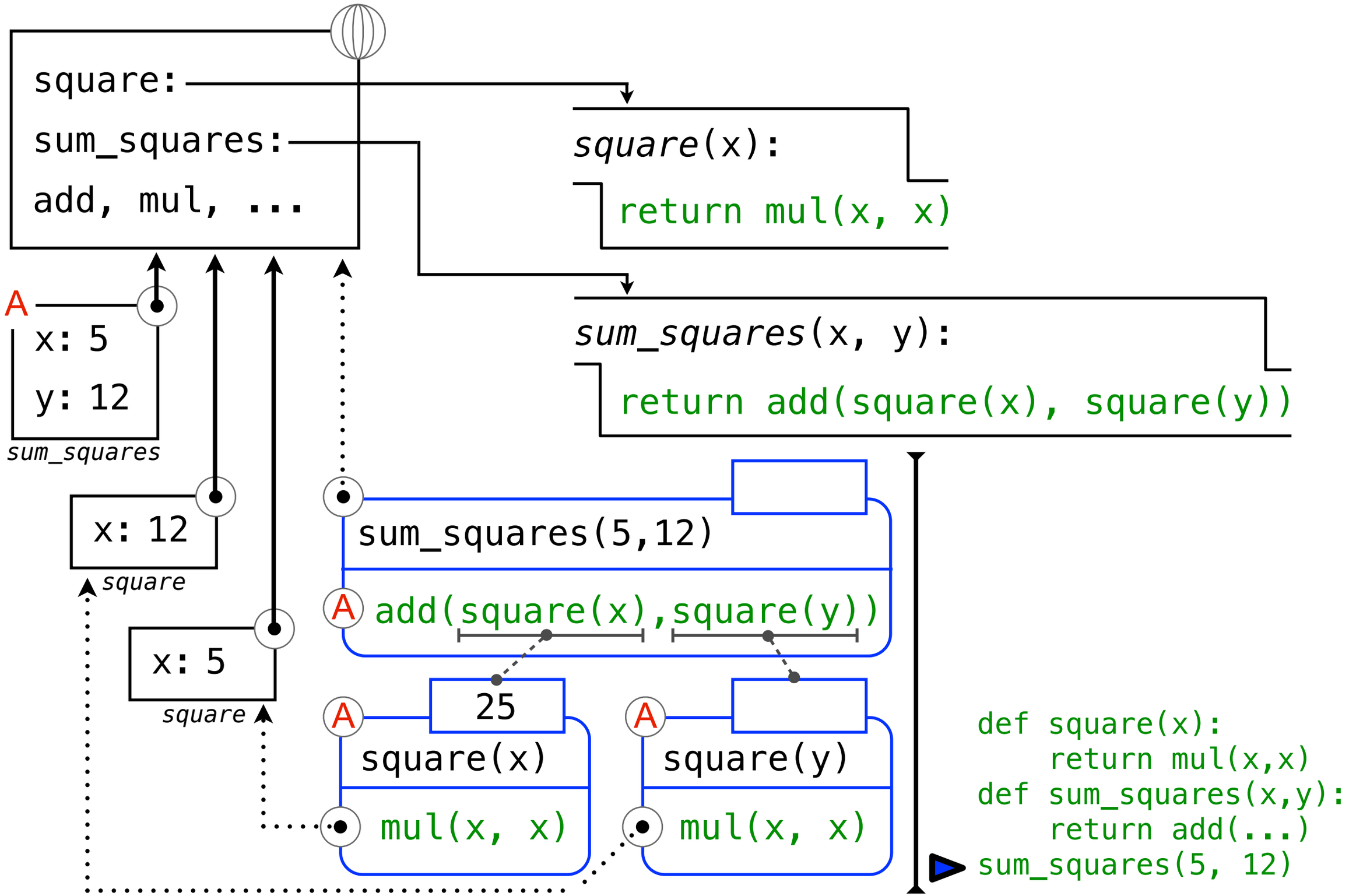
Example: Function Application



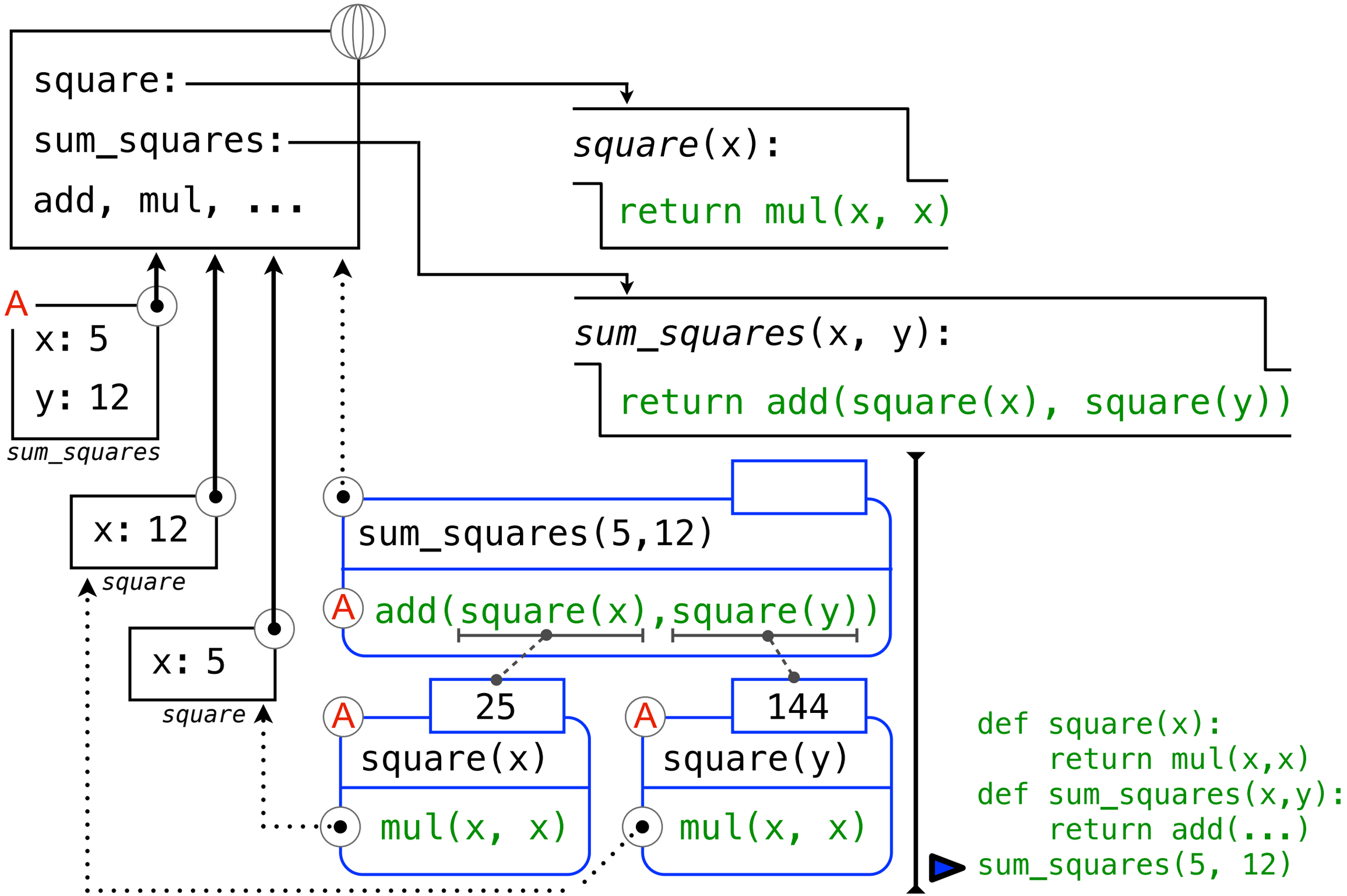
Example: Function Application



Example: Function Application



Example: Function Application



Example: Function Application