61A Lecture 5

Announcements
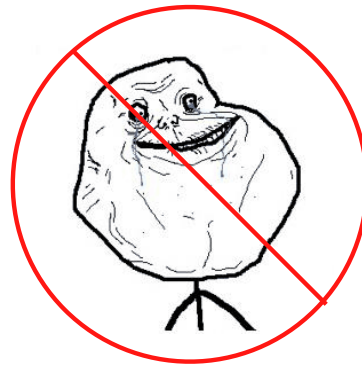
Office Hours: You Should Go!

Office Hours: You Should Go!

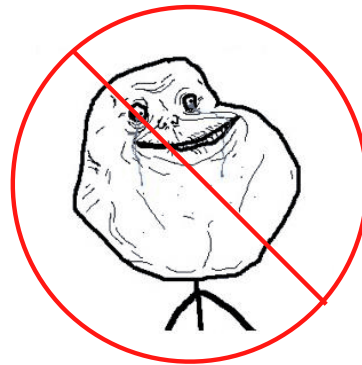`You are not alone!`

You are not alone!

Office Hours: You Should Go!

You are not alone!

http://cs61a.org/office-hours.html

Environments for Higher-Order Functions

Environments Enable Higher-Order Functions

Environments Enable Higher-Order Functions

Functions are first-class: Functions are values in our programming language

Environments Enable Higher-Order Functions

Functions are first-class: Functions are values in our programming language

Higher-order function: A function that takes a function as an argument value **or**
A function that returns a function as a return value

Environments Enable Higher-Order Functions

Functions are first-class: Functions are values in our programming language

Higher-order function: A function that takes a function as an argument value **or**
A function that returns a function as a return value

Environment diagrams describe how higher-order functions work!

Environments Enable Higher-Order Functions

Functions are first-class: Functions are values in our programming language

Higher-order function: A function that takes a function as an argument value **or**
A function that returns a function as a return value

Environment diagrams describe how higher-order functions work!

(Demo)

Names can be Bound to Functional Arguments

```
1  def apply_twice(f, x):
2      return f(f(x))
3
4  def square(x):
5      return x * x
6
7  result = apply_twice(square, 2)
```

Global frame

apply_twice

square

func apply_twice(f, x) [parent=Global]

func square(x) [parent=Global]

Interactive Diagram

Names can be Bound to Functional Arguments

```
1  def apply_twice(f, x):
2      return f(f(x))
3
4  def square(x):
5      return x * x
6
7  result = apply_twice(square, 2)
```

Global frame

apply_twice

square

func apply_twice(f, x) [parent=Global]

func square(x) [parent=Global]

Interactive Diagram

Names can be Bound to Functional Arguments

```
1  def apply_twice(f, x):
2      return f(f(x))
3
4  def square(x):
5      return x * x
6
7  result = apply_twice(square, 2)
```

Global frame

apply_twice •⟶ func apply_twice(f, x) [parent=Global]

square •⟶ func square(x) [parent=Global]

Applying a user-defined function:

- Create a new frame
- Bind formal parameters (f & x) to arguments
- Execute the body:
  return f(f(x))

Interactive Diagram

Names can be Bound to Functional Arguments

```
1  def apply_twice(f, x):
2      return f(f(x))
3
4  def square(x):
5      return x * x
6
7  result = apply_twice(square, 2)
```

Global frame

apply_twice
square

func apply_twice(f, x) [parent=Global]

func square(x) [parent=Global]

Applying a user-defined function:

- Create a new frame
- Bind formal parameters (f & x) to arguments
- Execute the body: return f(f(x))

```
1  def apply_twice(f, x):
2      return f(f(x))
3
4  def square(x):
5      return x * x
6
7  result = apply_twice(square, 2)
```

Global frame

apply_twice
square

func apply_twice(f, x) [parent=Global]

func square(x) [parent=Global]

f1: apply_twice [parent=Global]

f
x 2

Interactive Diagram

Names can be Bound to Functional Arguments

```
1  def apply_twice(f, x):
2      return f(f(x))
3
4  def square(x):
5      return x * x
6
7  result = apply_twice(square, 2)
```

Global frame

apply_twice

square

func apply_twice(f, x) [parent=Global]

func square(x) [parent=Global]

Applying a user-defined function:

- Create a new frame
- Bind formal parameters (f & x) to arguments
- Execute the body: return f(f(x))

```
1  def apply_twice(f, x):
2      return f(f(x))
3
4  def square(x):
5      return x * x
6
7  result = apply_twice(square, 2)
```

2

Global frame

apply_twice

square

func apply_twice(f, x) [parent=Global]

func square(x) [parent=Global]

1 f1: apply_twice [parent=Global]

f

x   2

Interactive Diagram

Names can be Bound to Functional Arguments

```
1  def apply_twice(f, x):
2      return f(f(x))
3
4  def square(x):
5      return x * x
6
7  result = apply_twice(square, 2)
```

Global frame

apply_twice ●⟶ func apply_twice(f, x) [parent=Global]

square ●⟶ func square(x) [parent=Global]

Applying a user-defined function:

- Create a new frame
- Bind formal parameters (f & x) to arguments
- Execute the body:
  return f(f(x))

```
1  def apply_twice(f, x):
2      return f(f(x))
3
4  def square(x):
5      return x * x
6
7  result = apply_twice(square, 2)
```

2 Global frame ⟶ func apply_twice(f, x) [parent=Global]

apply_twice ●

square ● ⟶ func square(x) [parent=Global]

1 f1: apply_twice [parent=Global]

f ●

x 2

Interactive Diagram

Environments for Nested Definitions

(Demo)

Environment Diagrams for Nested Def Statements

```
1   def make_adder(n):
2       def adder(k):
3           return k + n
4       return adder
5
6   add_three = make_adder(3)
7   add_three(4)
```


Global frame

make_adder
add_three

func make_adder(n) [parent=Global]

func adder(k) [parent=f1]

f1: make_adder [parent=G]

n 3
adder
Return value

f2: adder [parent=f1]

k 4
Return value 7

Environment Diagrams for Nested Def Statements


```
    Nested def

1   def make_adder(n):
2       def adder(k):
3           return k + n
4       return adder
5
6   add_three = make_adder(3)
7   add_three(4)
```

Global frame
    make_adder   func make_adder(n) [parent=Global]
    add_three   func adder(k) [parent=f1]

f1: make_adder [parent=G]
    n   3
    adder
    Return
    value

f2: adder [parent=f1]
    k   4
    Return
    value   7

<u>Interactive Diagram</u>

Environment Diagrams for Nested Def Statements

Nested def

```
1  def make_adder(n):
2      def adder(k):
3          return k + n
4      return adder
5
6  add_three = make_adder(3)
7  add_three(4)
```

Global frame   func make_adder(n) [parent=Global]
            make_adder   func adder(k) [parent=f1]
            add_three

f1: make_adder [parent=G]
                    n   3
                adder
            Return
            value

f2: adder [parent=f1]
                    k   4
            Return   7
            value

Interactive Diagram

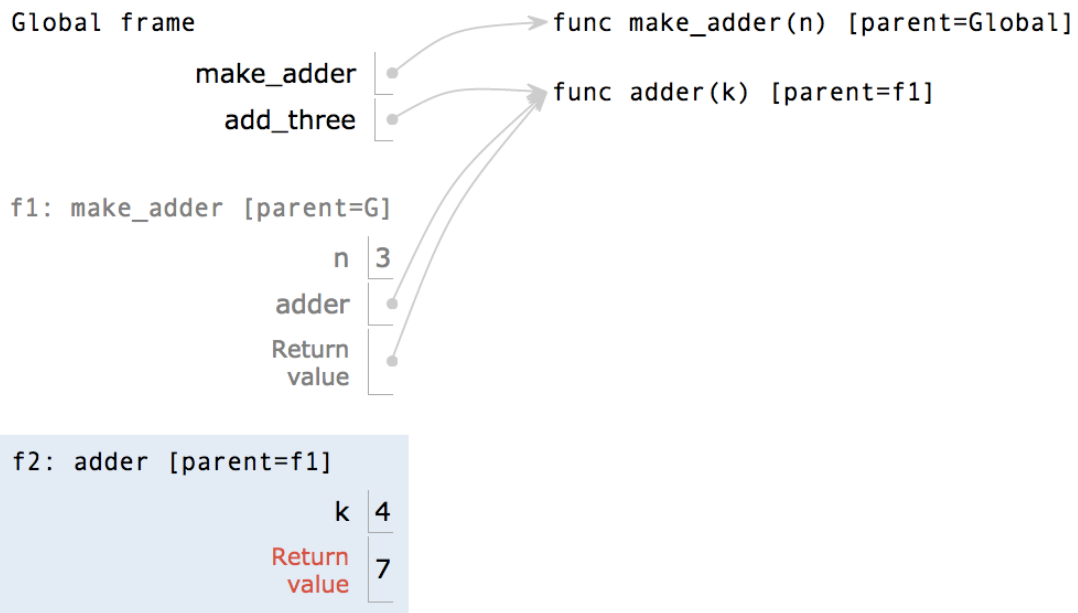# Environment Diagrams for Nested Def Statements

Nested def

```
1  def make_adder(n):
2      def adder(k):
3          return k + n
4      return adder
5
6  add_three = make_adder(3)
7  add_three(4)
```

Global frame   func make_adder(n) [parent=Global]

make_adder
add_three   func adder(k) [parent=f1]

f1: make_adder [parent=G]

n   3

adder

Return
value

f2: adder [parent=f1]

k   4

Return   7
value

Interactive Diagram

Environment Diagrams for Nested Def Statements


```
     Nested def

1    def make_adder(n):
2        def adder(k):
3            return k + n
4        return adder
5
6    add_three = make_adder(3)
7    add_three(4)
```

Global frame   func make_adder(n) [parent=Global]

        make_adder   func adder(k) [parent=f1]
        add_three

f1: make_adder [parent=G]

                n   3
            adder
           Return
            value

f2: adder [parent=f1]

                k   4
           Return   7
            value
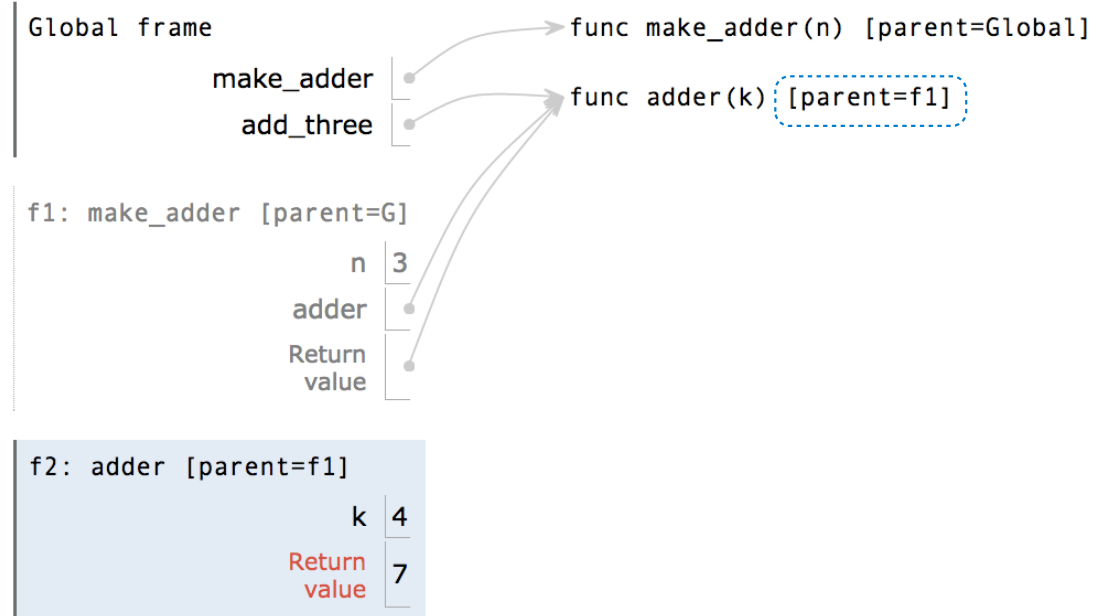
Interactive Diagram

Environment Diagrams for Nested Def Statements

Nested def

```
1  def make_adder(n):
2      def adder(k):
3          return k + n
4      return adder
5
6  add_three = make_adder(3)
7  add_three(4)
```

Global frame

make_adder

add_three

func make_adder(n) [parent=Global]

func adder(k) [parent=f1]

f1: make_adder [parent=G]

n   3

adder

Return
value

f2: adder [parent=f1]

k   4

Return
value   7

<u>Interactive Diagram</u>

Environment Diagrams for Nested Def Statements

Nested def

```
1  def make_adder(n):
2      def adder(k):
3          return k + n
4      return adder
5
6  add_three = make_adder(3)
7  add_three(4)
```
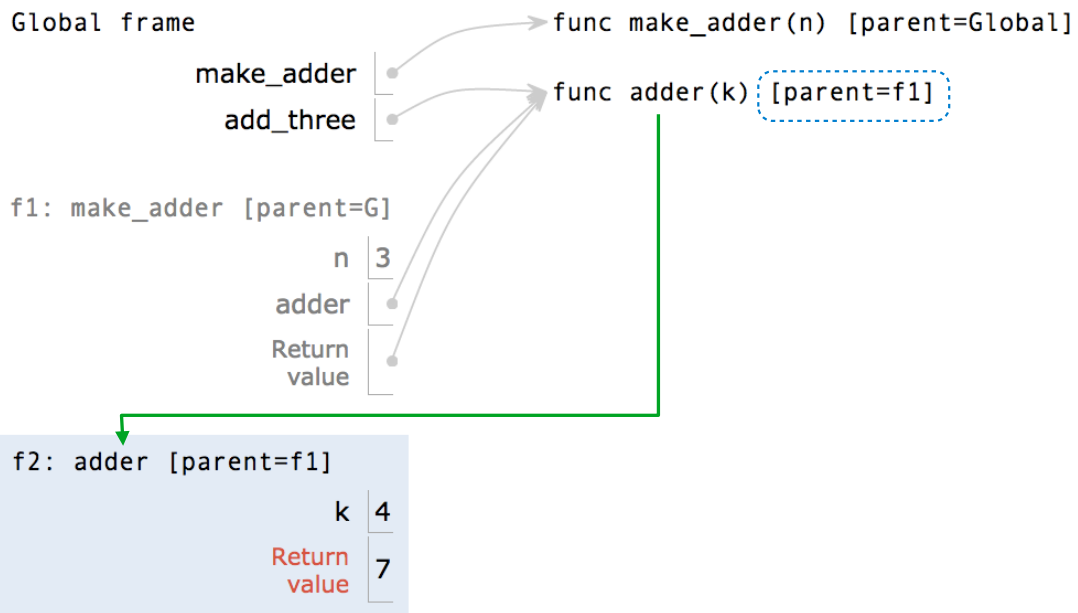

Global frame

make_adder
add_three

func make_adder(n) [parent=Global]

func adder(k) [parent=f1]

f1: make_adder [parent=G]

n 3
adder
Return value

f2: adder [parent=f1]

k 4
Return value 7

Interactive Diagram

8

Environment Diagrams for Nested Def Statements
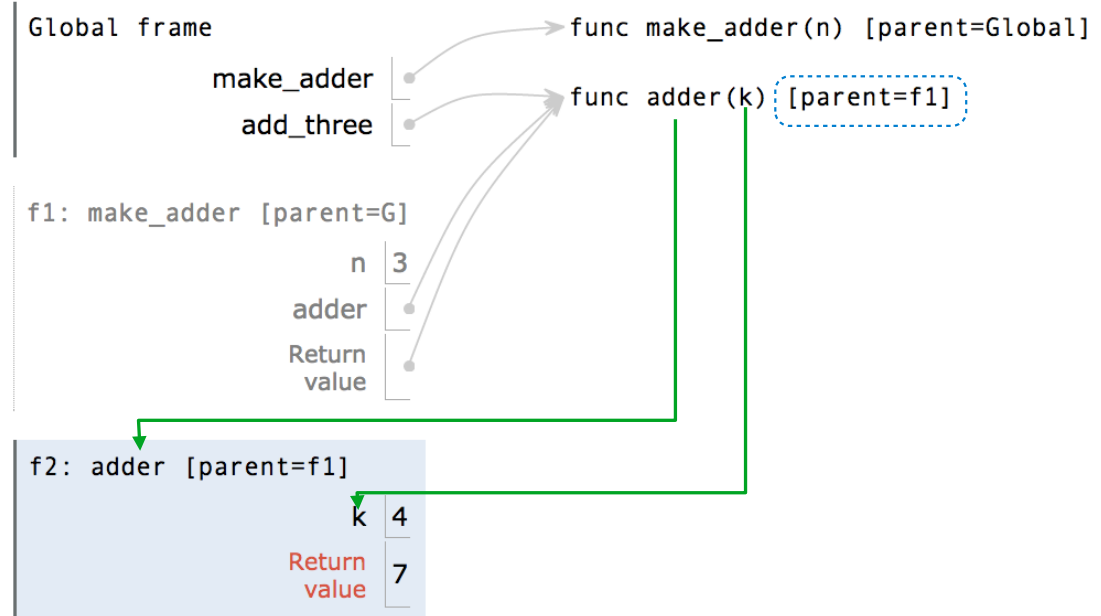
Environment Diagrams for Nested Def Statements

Nested def

```
1   def make_adder(n):
2       def adder(k):
3           return k + n
4       return adder
5
6   add_three = make_adder(3)
7   add_three(4)
```

- Every user-defined function has a parent frame (often global)

3 Global frame

make_adder   func make_adder(n) [parent=Global]

add_three   func adder(k) [parent=f1]

f1: make_adder [parent=G]

     n   3

2   adder

     Return value

f2: adder [parent=f1]

     k   4

1   Return value   7

Interactive Diagram

8

Environment Diagrams for Nested Def Statements

Nested def

```
1  def make_adder(n):
2      def adder(k):
3          return k + n
4      return adder
5
6  add_three = make_adder(3)
7  add_three(4)
```
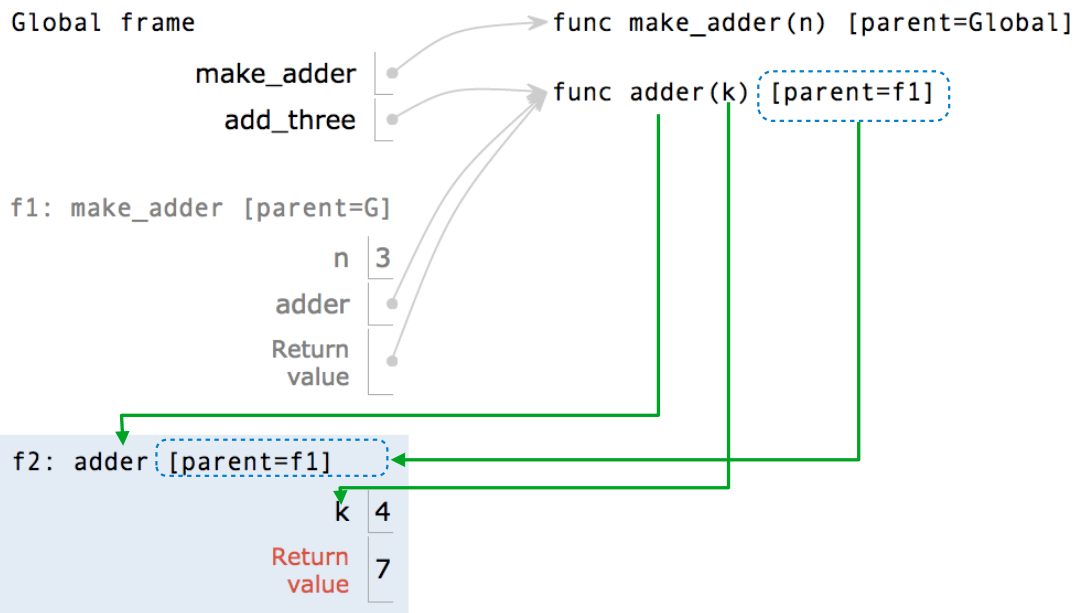
- Every user–defined function has a parent frame (often global)

- The parent of a function is the frame in which it was defined

Global frame
make_adder
add_three
func make_adder(n) [parent=Global]
func adder(k) [parent=f1]

f1: make_adder [parent=G]
n 3
adder
Return value

f2: adder [parent=f1]
k 4
Return value 7

Environment Diagrams for Nested Def Statements

Nested def

```
1  def make_adder(n):
2      def adder(k):
3          return k + n
4      return adder
5
6  add_three = make_adder(3)
7  add_three(4)
```
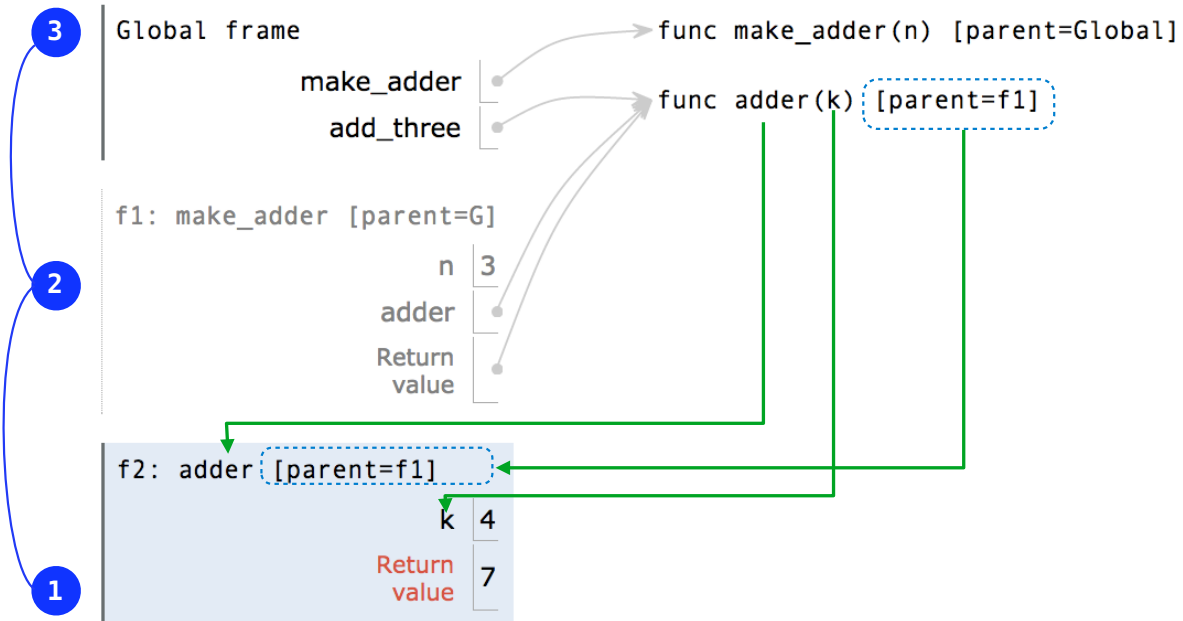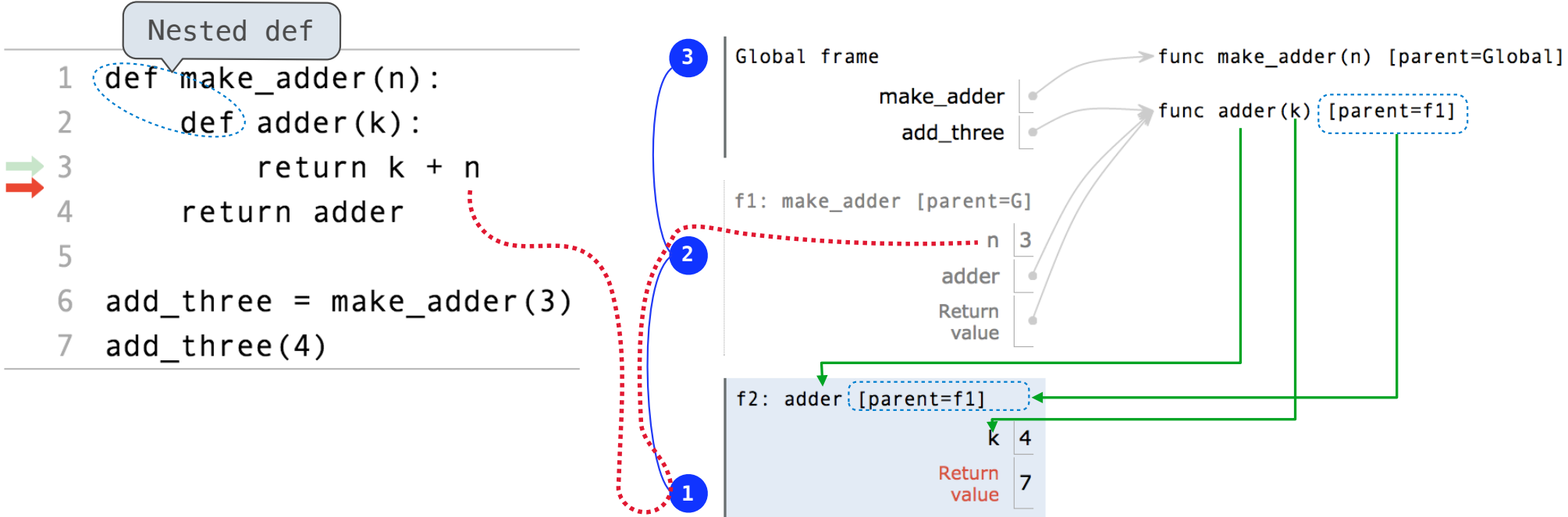
- Every user-defined function has a parent frame (often global)

- The parent of a function is the frame in which it was defined

- Every local frame has a parent frame (often global)

Interactive Diagram

Environment Diagrams for Nested Def Statements

Nested def

```
1  def make_adder(n):
2      def adder(k):
3          return k + n
4      return adder
5
6  add_three = make_adder(3)
7  add_three(4)
```
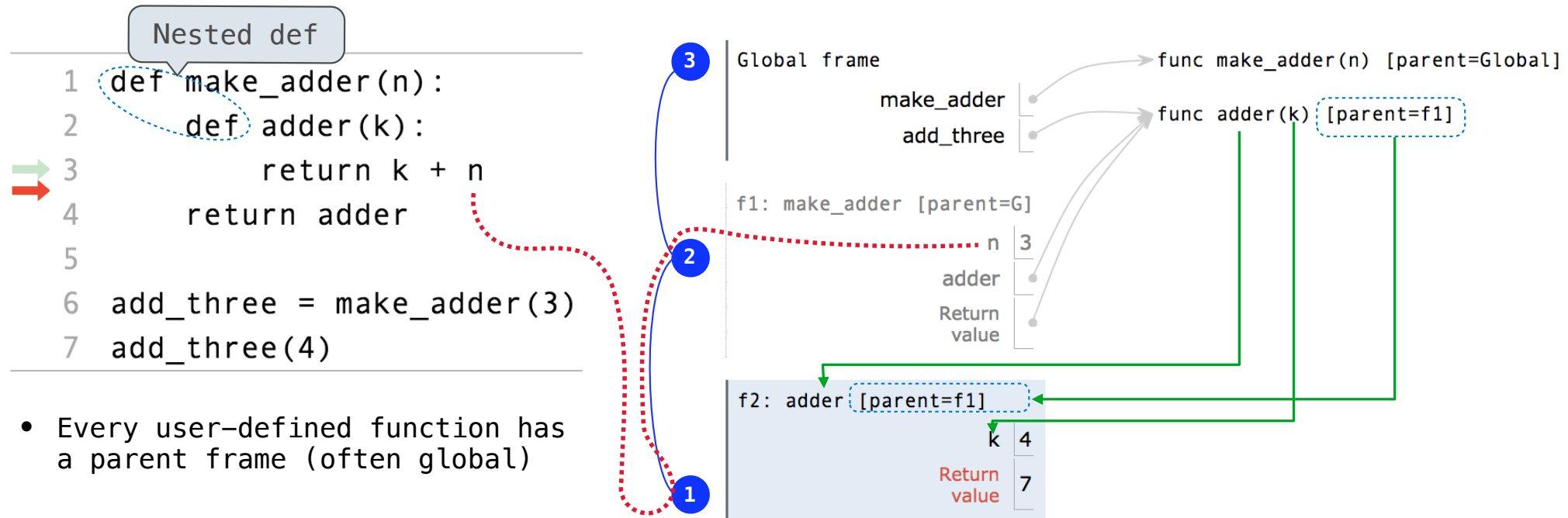
- Every user-defined function has a parent frame (often global)

- The parent of a function is the frame in which it was defined

- Every local frame has a parent frame (often global)

- The parent of a frame is the parent of the function called

Global frame

make_adder
add_three

func make_adder(n) [parent=Global]
func adder(k) [parent=f1]

f1: make_adder [parent=G]

n 3
adder
Return value

f2: adder [parent=f1]

k 4
Return value 7

Interactive Diagram

8
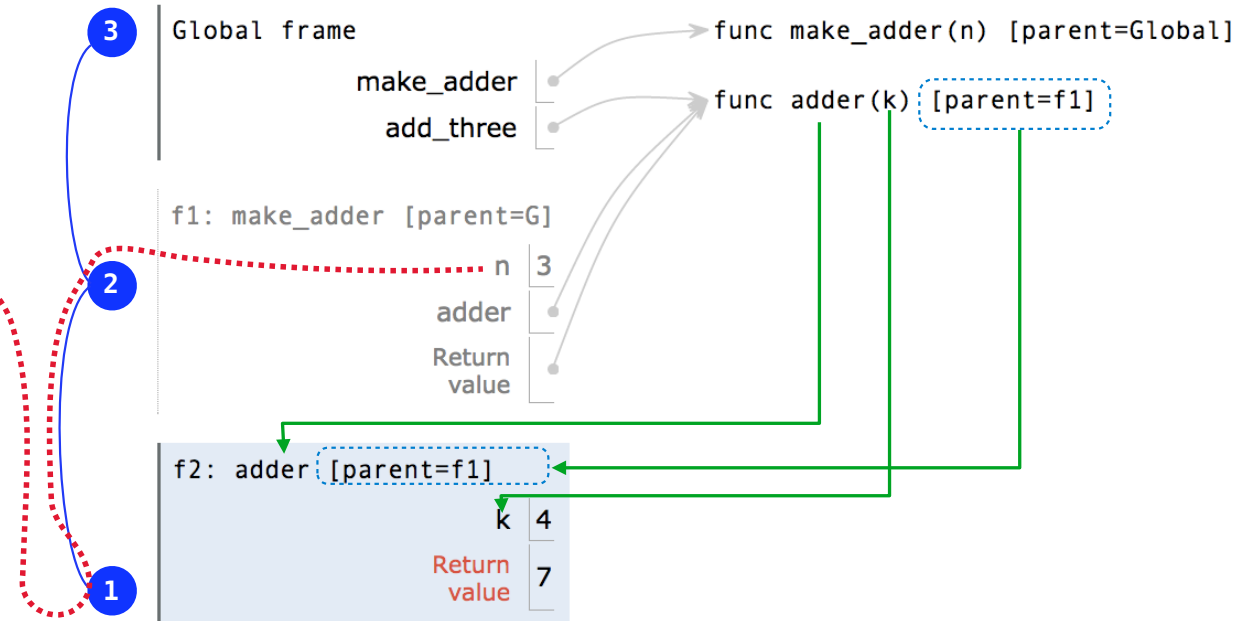
How to Draw an Environment Diagram

How to Draw an Environment Diagram

`When a function is defined:`

How to Draw an Environment Diagram

When a function is defined:

Create a function value: func <name>(<formal parameters>) [parent=<label>]

How to Draw an Environment Diagram

When a function is defined:

Create a function value: func <name>(<formal parameters>) [parent=<label>]

Its parent is the current frame.

How to Draw an Environment Diagram

When a function is defined:

Create a function value: func <name>(<formal parameters>) [parent=<label>]

Its parent is the current frame.

f1: make_adder   func adder(k) [parent=f1]

How to Draw an Environment Diagram

When a function is defined:

Create a function value: func <name>(<formal parameters>) [parent=<label>]

Its parent is the current frame.

f1: make_adder   func adder(k) [parent=f1]

Bind <name> to the function value in the current frame

How to Draw an Environment Diagram

When a function is defined:

Create a function value: func <name>(<formal parameters>) [parent=<label>]

Its parent is the current frame.

f1: make_adder   func adder(k) [parent=f1]

Bind <name> to the function value in the current frame
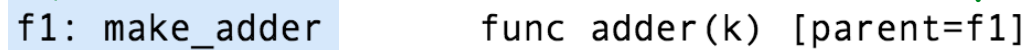
When a function is called:

How to Draw an Environment Diagram

When a function is defined:

Create a function value: func <name>(<formal parameters>) [parent=<label>]

Its parent is the current frame.

f1: make_adder   func adder(k) [parent=f1]

Bind <name> to the function value in the current frame

When a function is called:
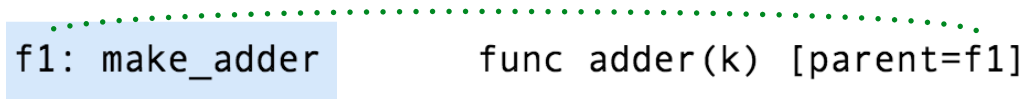
1. Add a local frame, titled with the <name> of the function being called.

How to Draw an Environment Diagram

When a function is defined:

Create a function value: func <name>(<formal parameters>) [parent=<label>]

Its parent is the current frame.

f1: make_adder   func adder(k) [parent=f1]

Bind <name> to the function value in the current frame

When a function is called:

1. Add a local frame, titled with the <name> of the function being called.

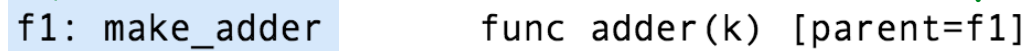2. Copy the parent of the function to the local frame: [parent=<label>]

How to Draw an Environment Diagram

When a function is defined:

Create a function value: func <name>(<formal parameters>) [parent=<label>]

Its parent is the current frame.

f1: make_adder   func adder(k) [parent=f1]

Bind <name> to the function value in the current frame

When a function is called:

1. Add a local frame, titled with the <name> of the function being called.
2. Copy the parent of the function to the local frame: [parent=<label>]
3. Bind the <formal parameters> to the arguments in the local frame.

How to Draw an Environment Diagram

When a function is defined:

Create a function value: func <name>(<formal parameters>) [parent=<label>]

Its parent is the current frame.
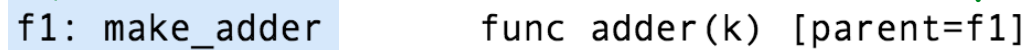
f1: make_adder   func adder(k) [parent=f1]

Bind <name> to the function value in the current frame

When a function is called:

1. Add a local frame, titled with the <name> of the function being called.

2. Copy the parent of the function to the local frame: [parent=<label>]

3. Bind the <formal parameters> to the arguments in the local frame.

4. Execute the body of the function in the environment that starts with the local frame.

Local Names

(Demo)

Local Names are not Visible to Other (Non-Nested) Functions

```
1  def f(x, y):
2      return g(x)
3
4  def g(a):
5      return a + y
6
7  result = f(1, 2)
```

Global frame   func f(x, y) [parent=Global]

                           f   func g(a) [parent=Global]

                           g

f1: f [parent=Global]

                         x 1

                         y 2

f2: g [parent=Global]

                         a 1

Local Names are not Visible to Other (Non-Nested) Functions

```
1  def f(x, y):
2      return g(x)
3
4  def g(a):
5      return a + y
6
7  result = f(1, 2)
```

Global frame   func f(x, y) [parent=Global]

                        f   func g(a) [parent=Global]
                        g

f1: f [parent=Global]

                        x 1
                        y 2

f2: g [parent=Global]

                        a 1

Interactive Diagram

Local Names are not Visible to Other (Non-Nested) Functions

```
1   def f(x, y):
2       return g(x)
3
4   def g(a):
5       return a + y
6
7   result = f(1, 2)
```

Global frame

func f(x, y) [parent=Global]

f

g

func g(a) [parent=Global]

f1: f [parent=Global]

x 1

y 2

f2: g [parent=Global]

a 1

Interactive Diagram

Local Names are not Visible to Other (Non-Nested) Functions


```
1   def f(x, y):
2       return g(x)
3
4   def g(a):
5       return a + y
6
7   result = f(1, 2)
```

"y" is not found

Global frame

f
g

func f(x, y) [parent=Global]

func g(a) [parent=Global]

f1: f [parent=Global]
x 1
y 2

f2: g [parent=Global]
a 1

Interactive Diagram

Local Names are not Visible to Other (Non-Nested) Functions

Local Names are not Visible to Other (Non-Nested) Functions

Local Names are not Visible to Other (Non-Nested) Functions


```
1   def f(x, y):
2       return g(x)
3
4   def g(a):
5       return a + y
6
7   result = f(1, 2)
```

"y" is not found, again

Error

"y" is not found

Global frame → func f(x, y) [parent=Global]

f
g → func g(a) [parent=Global]

f1: f [parent=Global]
x 1
y 2

f2: g [parent=Global]
a 1

- An environment is a sequence of frames.
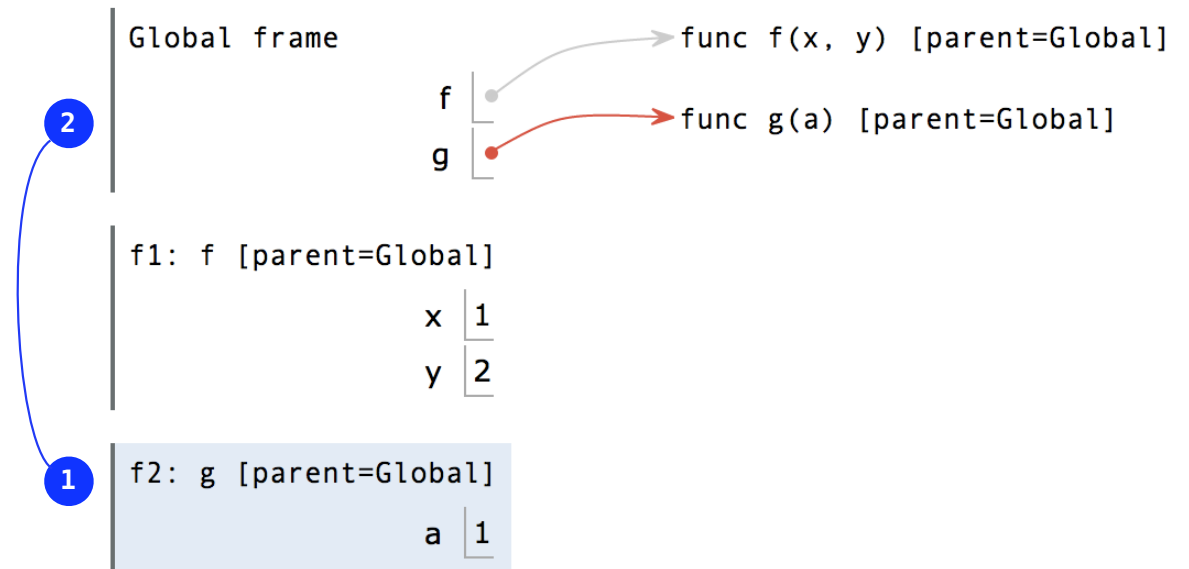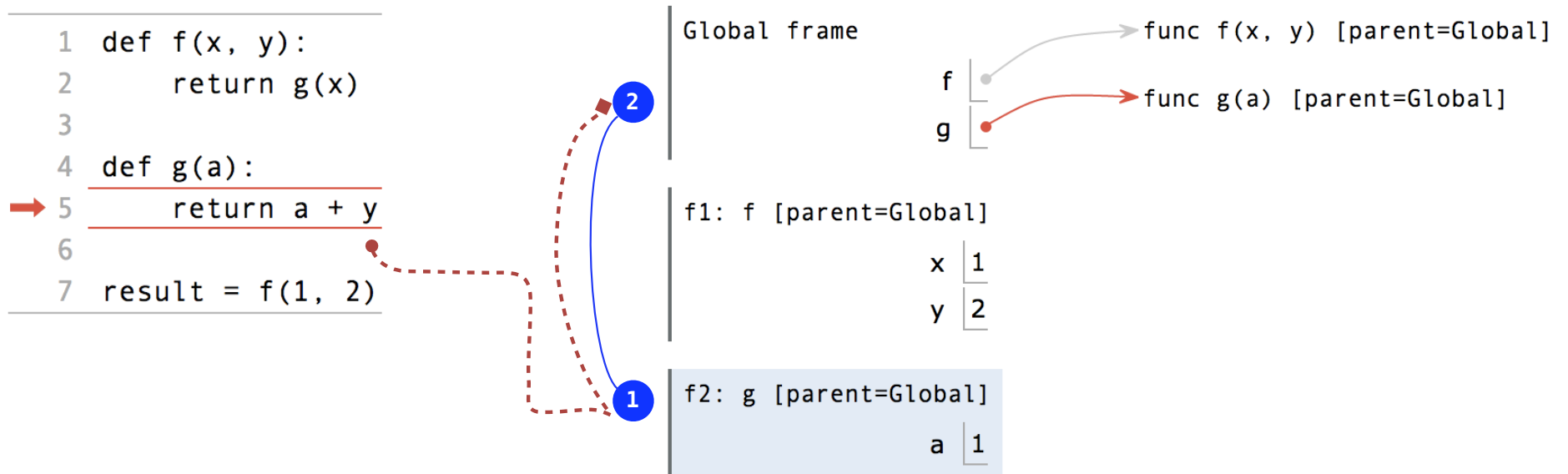
Local Names are not Visible to Other (Non-Nested) Functions
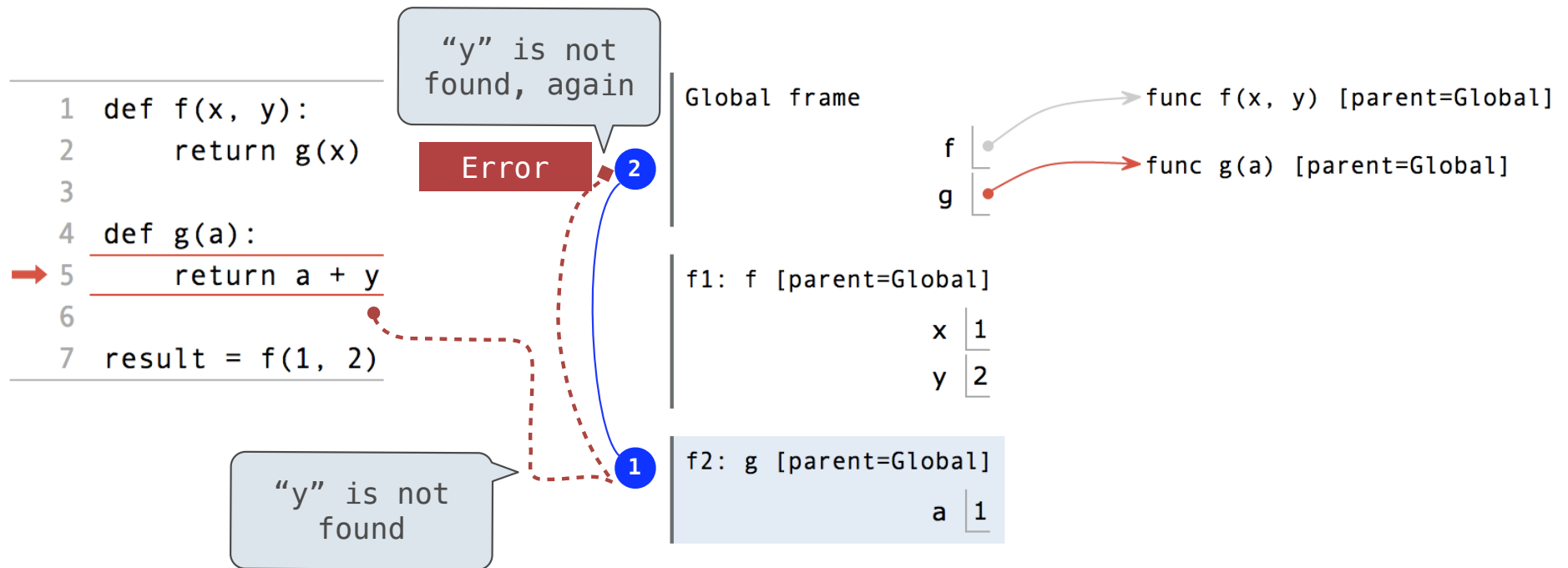
- An environment is a sequence of frames.
- The environment created by calling a top-level function (no def within def) consists of one local frame, followed by the global frame.

Function Composition

(Demo)

The Environment Diagram for Function Composition

```
1   def square(x):
2       return x * x
3
4   def make_adder(n):
5       def adder(k):
6           return k + n
7       return adder
8
9   def compose1(f, g):
10      def h(x):
11          return f(g(x))
12      return h
13
14  compose1(square, make_adder(2))(3)
```

Global frame

      square

  make_adder

    compose1

func square(x) [parent=Global]

func make_adder(n) [parent=Global]

func compose1(f, g) [parent=Global]

func adder(k) [parent=f1]

func h(x) [parent=f2]

f1: make_adder [parent=Global]

         n 2

      adder

    Return
     value

f2: compose1 [parent=Global]

         f

         g

         h

    Return
     value

f3: h [parent=f2]

         x 3

f4: adder [parent=f1]

         k 3

Interactive Diagram

13

The Environment Diagram for Function Composition

```
1   def square(x):
2       return x * x
3
4   def make_adder(n):
5       def adder(k):
6           return k + n
7       return adder
8
9   def compose1(f, g):
10      def h(x):
11          return f(g(x))
12      return h
13
14  compose1(square, make_adder(2))(3)
```

```
Global frame                           func square(x) [parent=Global]
                        square         func make_adder(n) [parent=Global]
                     make_adder
                      compose1         func compose1(f, g) [parent=Global]

                                       func adder(k) [parent=f1]
f1: make_adder [parent=Global]
                                       func h(x) [parent=f2]
                           n   2
                        adder
                      Return
                      value

f2: compose1 [parent=Global]
                           f
                           g
                           h
                      Return
                      value

f3: h [parent=f2]
                           x   3

f4: adder [parent=f1]
                           k   3
```

Interactive Diagram

The Environment Diagram for Function Composition

```
1   def square(x):
2       return x * x
3
4   def make_adder(n):
5       def adder(k):
6           return k + n
7       return adder
8
9   def compose1(f, g):
10      def h(x):
11          return f(g(x))
12      return h
13
14  compose1(square, make_adder(2))(3)
```

```
Global frame                          func square(x) [parent=Global]
                    square            func make_adder(n) [parent=Global]
               make_adder
                  compose1            func compose1(f, g) [parent=Global]

f1: make_adder [parent=Global]        func adder(k) [parent=f1]

                        n  2          func h(x) [parent=f2]
                    adder
                   Return
                    value

f2: compose1 [parent=Global]

                        f
                        g
                        h
                   Return
                    value

f3: h [parent=f2]

                        x  3

f4: adder [parent=f1]

                        k  3
```
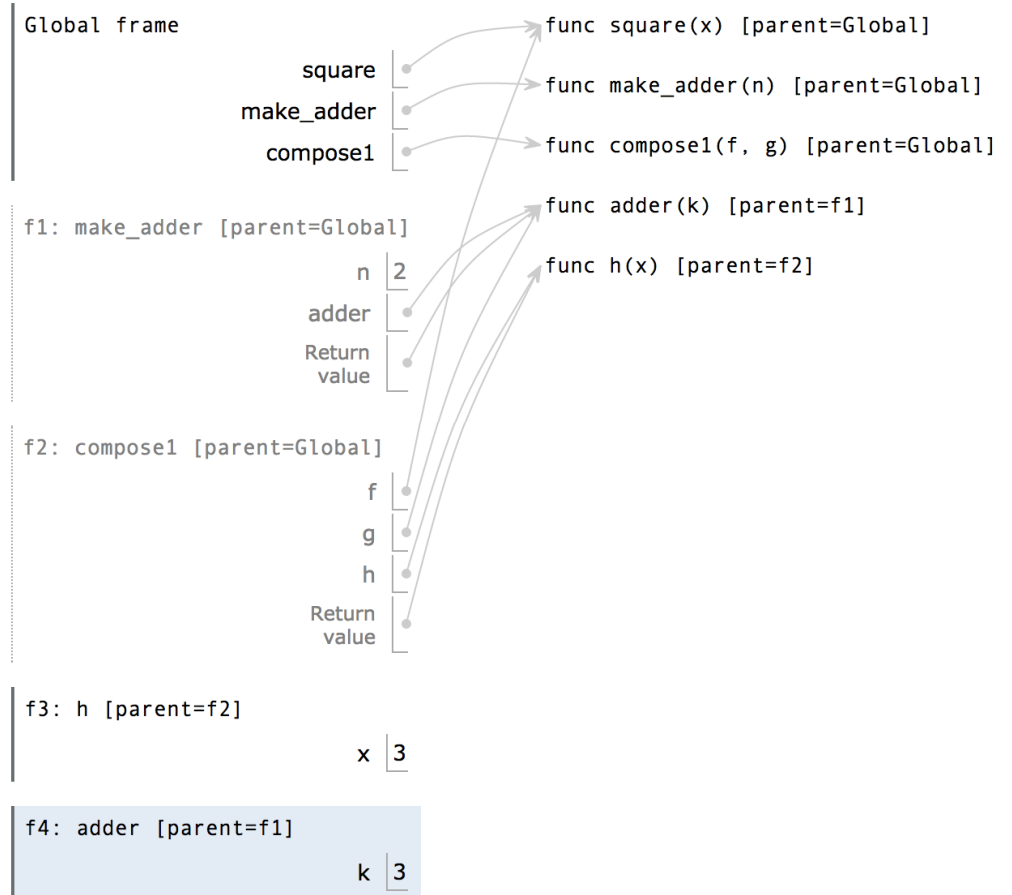
Interactive Diagram

The Environment Diagram for Function Composition

```
1   def square(x):
2       return x * x
3
4   def make_adder(n):
5       def adder(k):
6           return k + n
7       return adder
8
9   def compose1(f, g):
10      def h(x):
11          return f(g(x))
12      return h
13
14  compose1(square, make_adder(2))(3)
```

```
Global frame                              func square(x) [parent=Global]
                        square
                    make_adder            func make_adder(n) [parent=Global]
                      compose1
                                          func compose1(f, g) [parent=Global]

f1: make_adder [parent=Global]
                                          func adder(k) [parent=f1]
                           n   2
                                          func h(x) [parent=f2]
                        adder
                       Return
                       value

f2: compose1 [parent=Global]
                           f
                           g
                           h
                       Return
                       value

f3: h [parent=f2]
                           x   3

f4: adder [parent=f1]
                           k   3
```

Interactive Diagram

13

The Environment Diagram for Function Composition
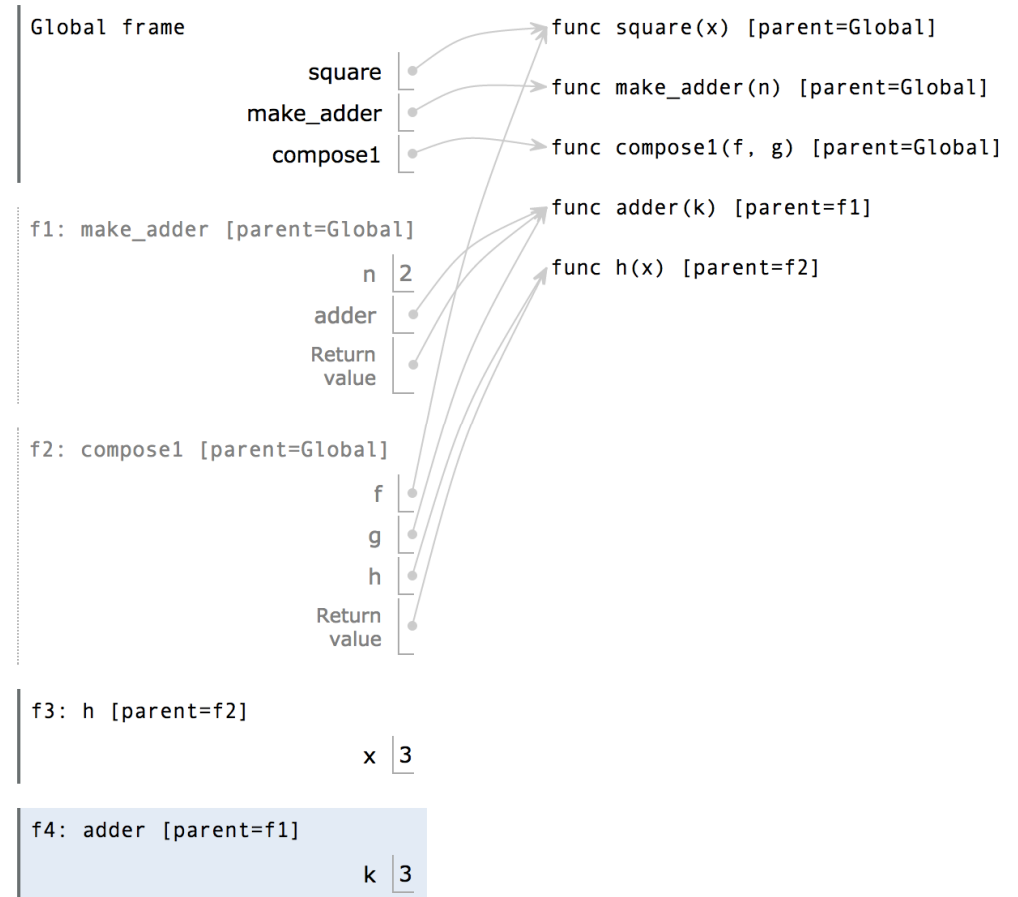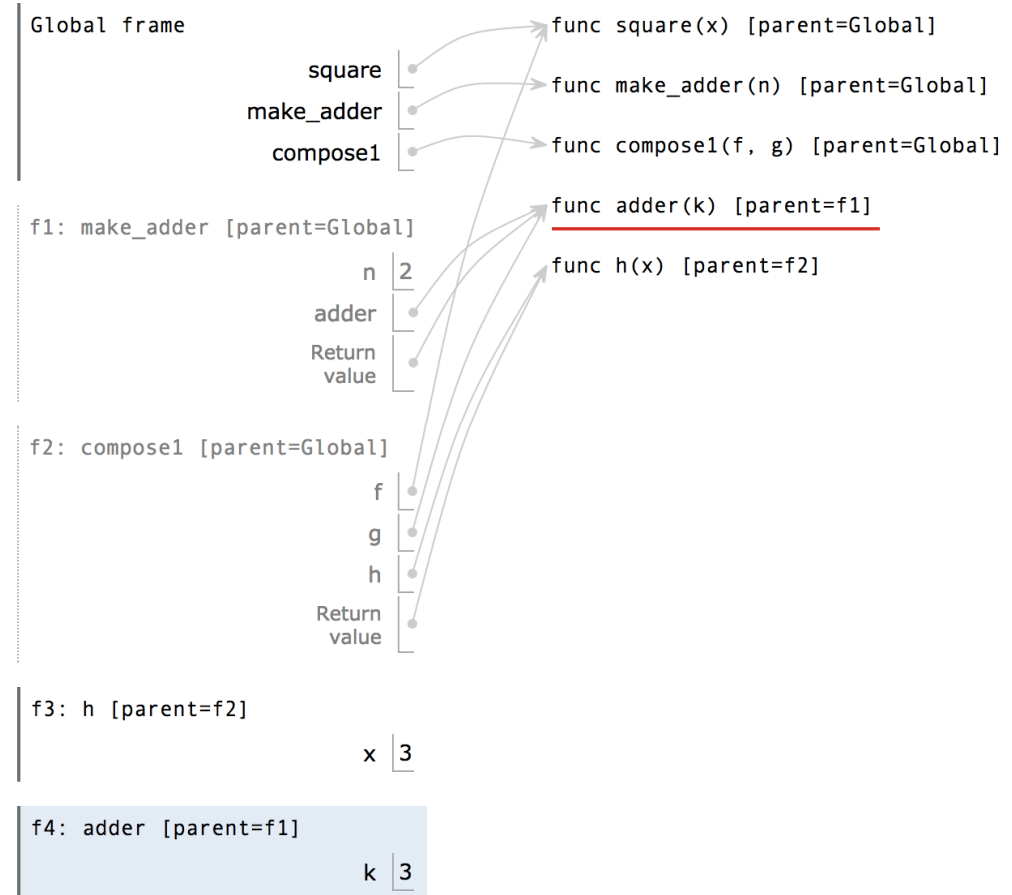
```
1  def square(x):
2      return x * x
3
4  def make_adder(n):
5      def adder(k):
6          return k + n
7      return adder
8
9  def compose1(f, g):
10     def h(x):
11         return f(g(x))
12     return h
13
14 compose1(square, make_adder(2))(3)
```

Return value of make_adder is an argument to compose1

Global frame

square
make_adder
compose1

func square(x) [parent=Global]
func make_adder(n) [parent=Global]
func compose1(f, g) [parent=Global]
func adder(k) [parent=f1]
func h(x) [parent=f2]

f1: make_adder [parent=Global]

n | 2
adder
Return value

f2: compose1 [parent=Global]

f
g
h
Return value

f3: h [parent=f2]

x | 3

f4: adder [parent=f1]

k | 3

Interactive Diagram

The Environment Diagram for Function Composition

```
1   def square(x):
2       return x * x
3
4   def make_adder(n):
5       def adder(k):
6           return k + n
7       return adder
8
9   def compose1(f, g):
10      def h(x):
11          return f(g(x))
12      return h
13
14  compose1(square, make_adder(2))(3)
```
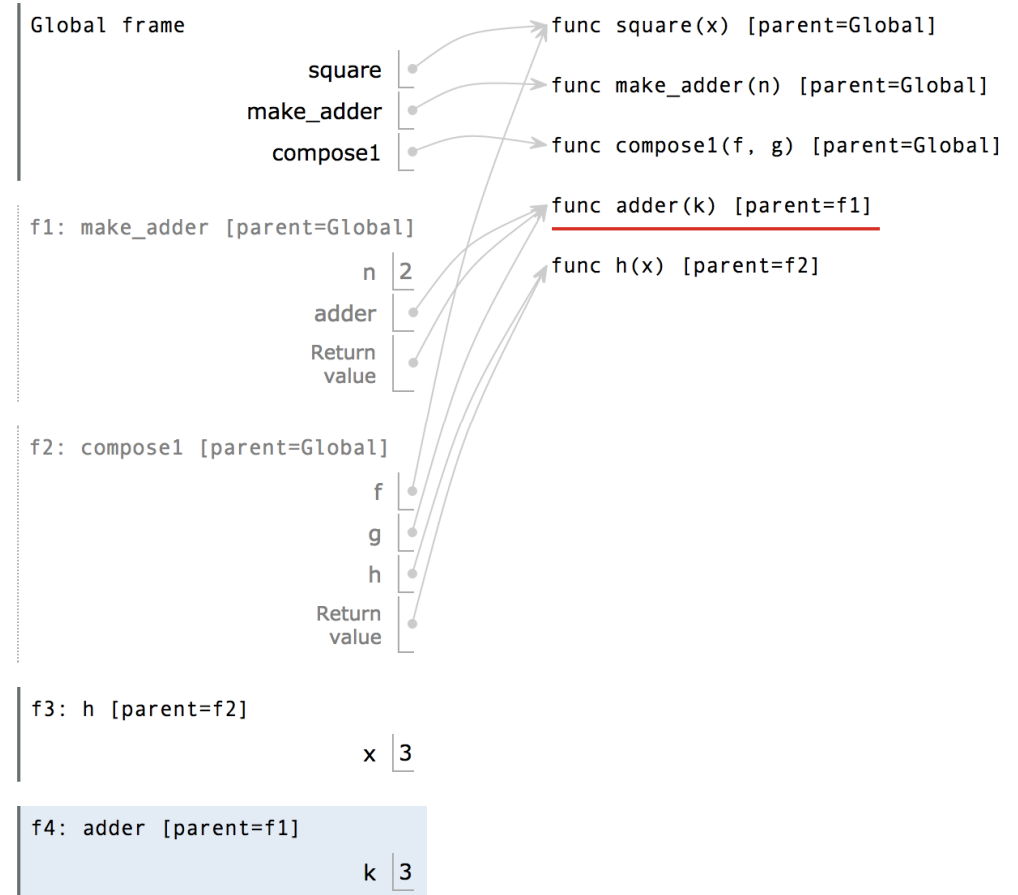
Return value of make_adder is an argument to compose1

```
Global frame                           func square(x) [parent=Global]
                    square  •
              make_adder  •            func make_adder(n) [parent=Global]
                compose1  •
                                       func compose1(f, g) [parent=Global]

f1: make_adder [parent=Global]
                                       func adder(k) [parent=f1]
                       n  2
                                       func h(x) [parent=f2]
                   adder  •
              Return  •
              value

f2: compose1 [parent=Global]
                       f  •
                       g  •
                       h  •
              Return  •
              value

f3: h [parent=f2]
                       x  3

f4: adder [parent=f1]
                       k  3
```

Interactive Diagram

The Environment Diagram for Function Composition

```
1   def square(x):
2       return x * x
3
4   def make_adder(n):
5       def adder(k):
6           return k + n
7       return adder
8
9   def compose1(f, g):
10      def h(x):
11          return f(g(x))
12      return h
13
14  compose1(square, make_adder(2))(3)
```
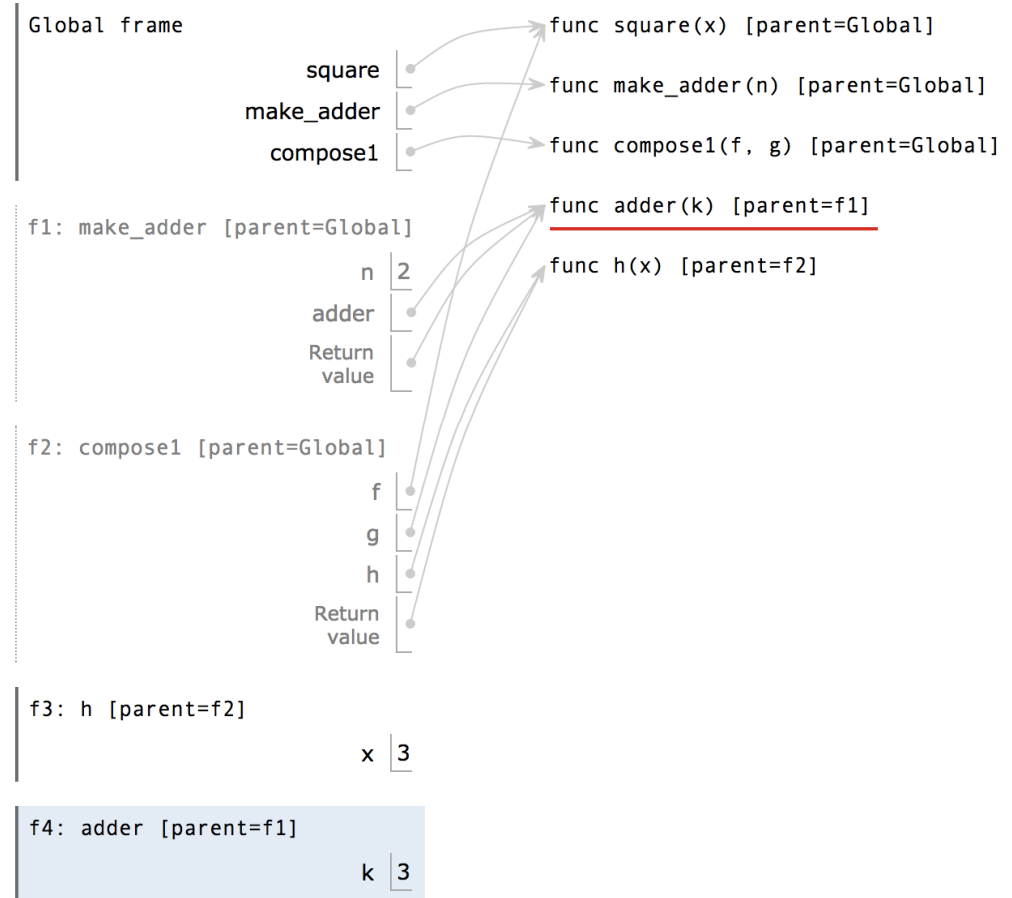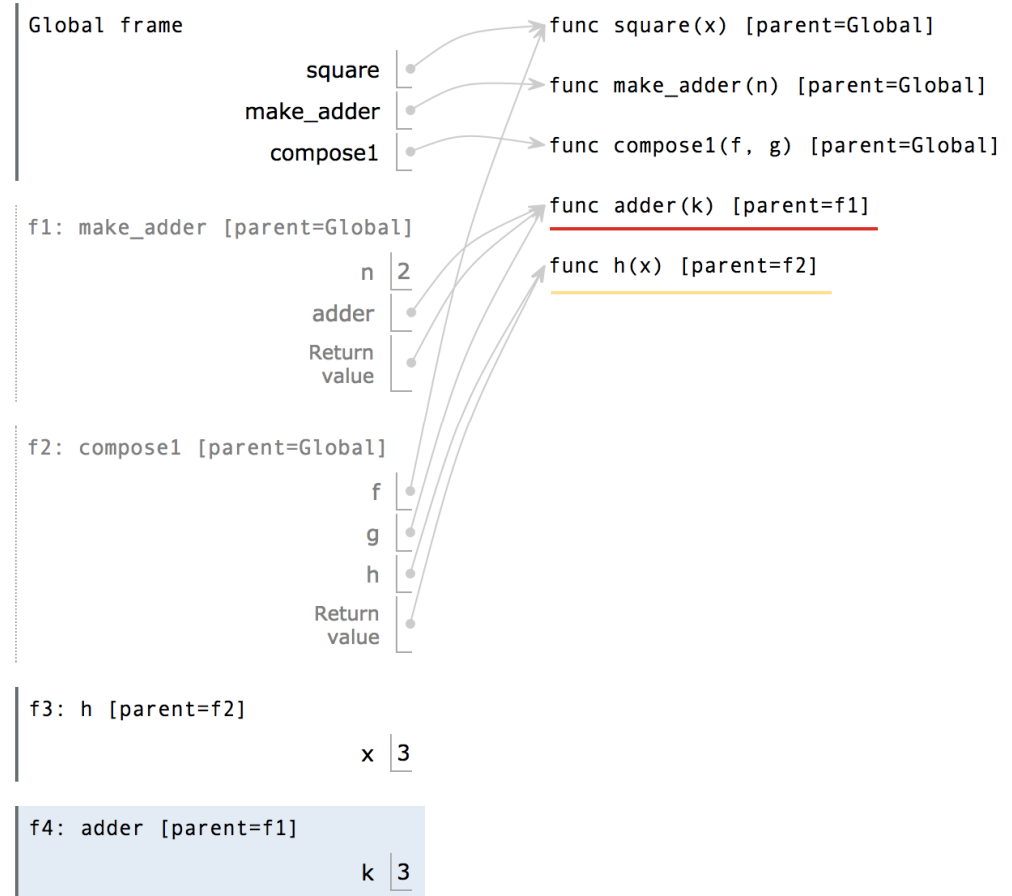
Return value of make_adder is an argument to compose1

```
Global frame                          func square(x) [parent=Global]
                square                func make_adder(n) [parent=Global]
          make_adder                  func compose1(f, g) [parent=Global]
            compose1
                                      func adder(k) [parent=f1]
f1: make_adder [parent=Global]
                                      func h(x) [parent=f2]
                      n  2
                  adder
              Return
               value

f2: compose1 [parent=Global]
                      f
                      g
                      h
              Return
               value

f3: h [parent=f2]
                      x  3

f4: adder [parent=f1]
                      k  3
```

Interactive Diagram

The Environment Diagram for Function Composition

```
1   def square(x):
2       return x * x
3
4   def make_adder(n):
5       def adder(k):
6           return k + n
7       return adder
8
9   def compose1(f, g):
10      def h(x):
11          return f(g(x))
12      return h
13
14  compose1(square, make_adder(2))(3)
```

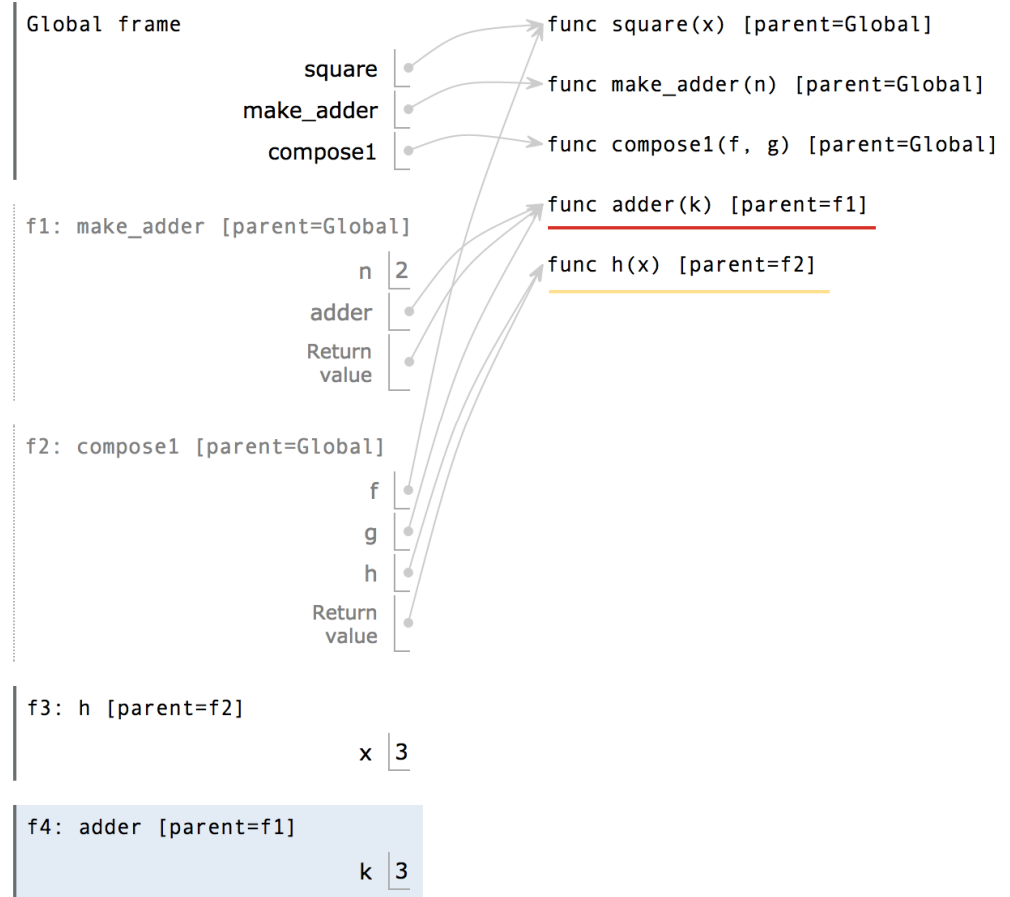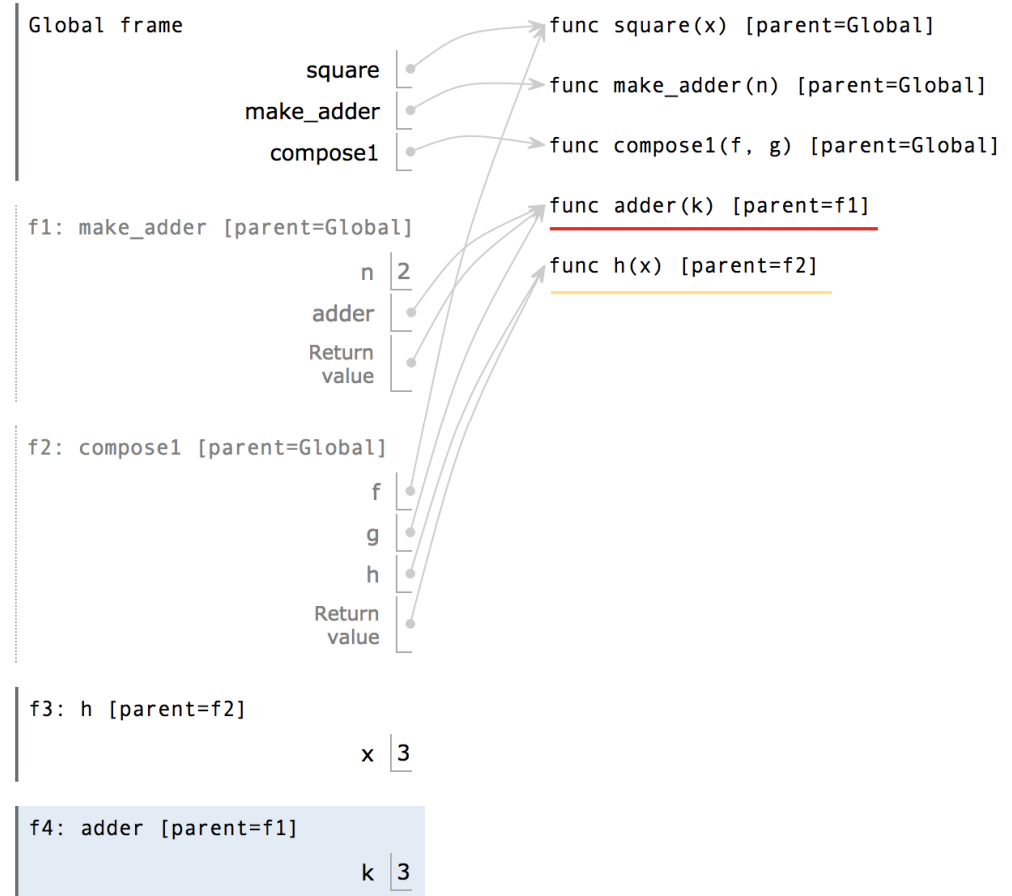Return value of make_adder is an argument to compose1

```
Global frame                          func square(x) [parent=Global]
              square                   func make_adder(n) [parent=Global]
         make_adder                    func compose1(f, g) [parent=Global]
           compose1
                                       func adder(k) [parent=f1]
f1: make_adder [parent=Global]
                                       func h(x) [parent=f2]
                    n    2
                adder
             Return
             value

f2: compose1 [parent=Global]
                    f
                    g
                    h
             Return
             value

f3: h [parent=f2]
                    x    3

f4: adder [parent=f1]
                    k    3
```

Interactive Diagram

The Environment Diagram for Function Composition

```
1   def square(x):
2       return x * x
3
4   def make_adder(n):
5       def adder(k):
6           return k + n
7       return adder
8
9   def compose1(f, g):
10      def h(x):
11          return f(g(x))
12      return h
13
14  compose1(square, make_adder(2))(3)
```
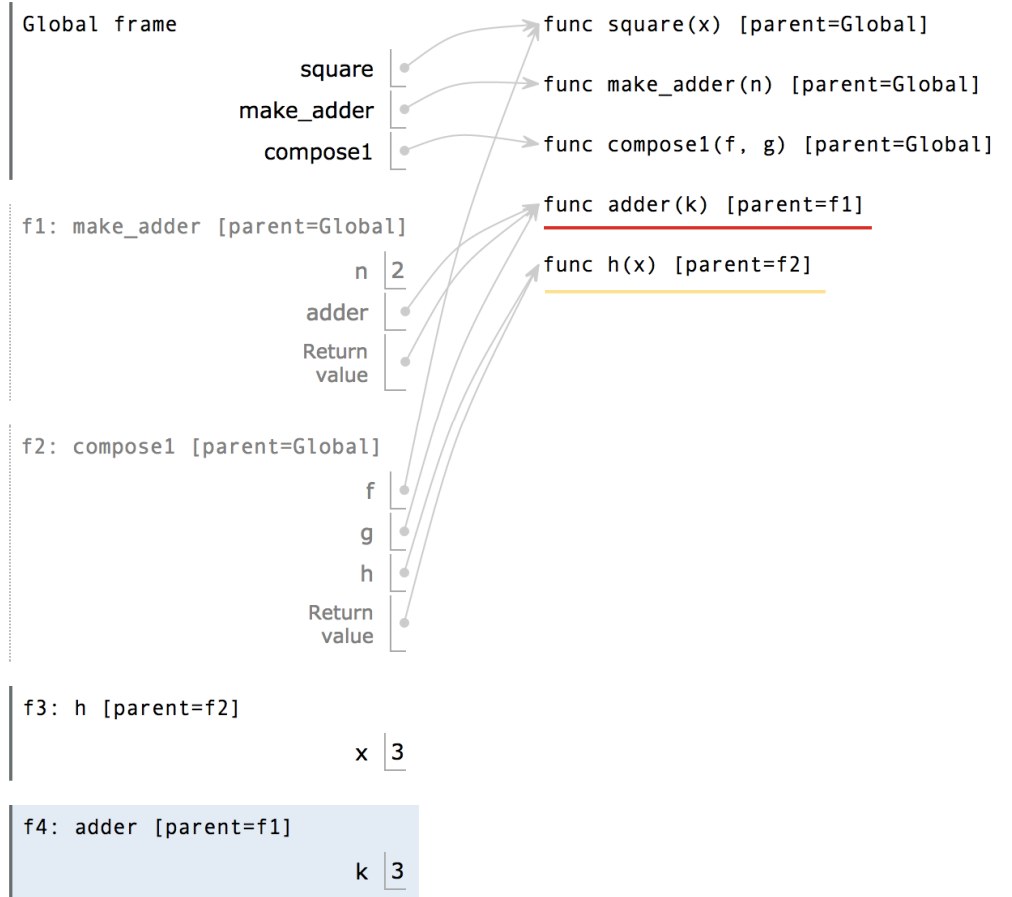
Return value of make_adder is an argument to compose1

Global frame

square	•
make_adder	•
compose1	•

func square(x) [parent=Global]

func make_adder(n) [parent=Global]

func compose1(f, g) [parent=Global]

func adder(k) [parent=f1]

func h(x) [parent=f2]

f1: make_adder [parent=Global]

n	2
adder	•
Return value	

f2: compose1 [parent=Global]

f	•
g	•
h	•
Return value	

f3: h [parent=f2]

x	3

f4: adder [parent=f1]

k	3

Interactive Diagram

13

The Environment Diagram for Function Composition
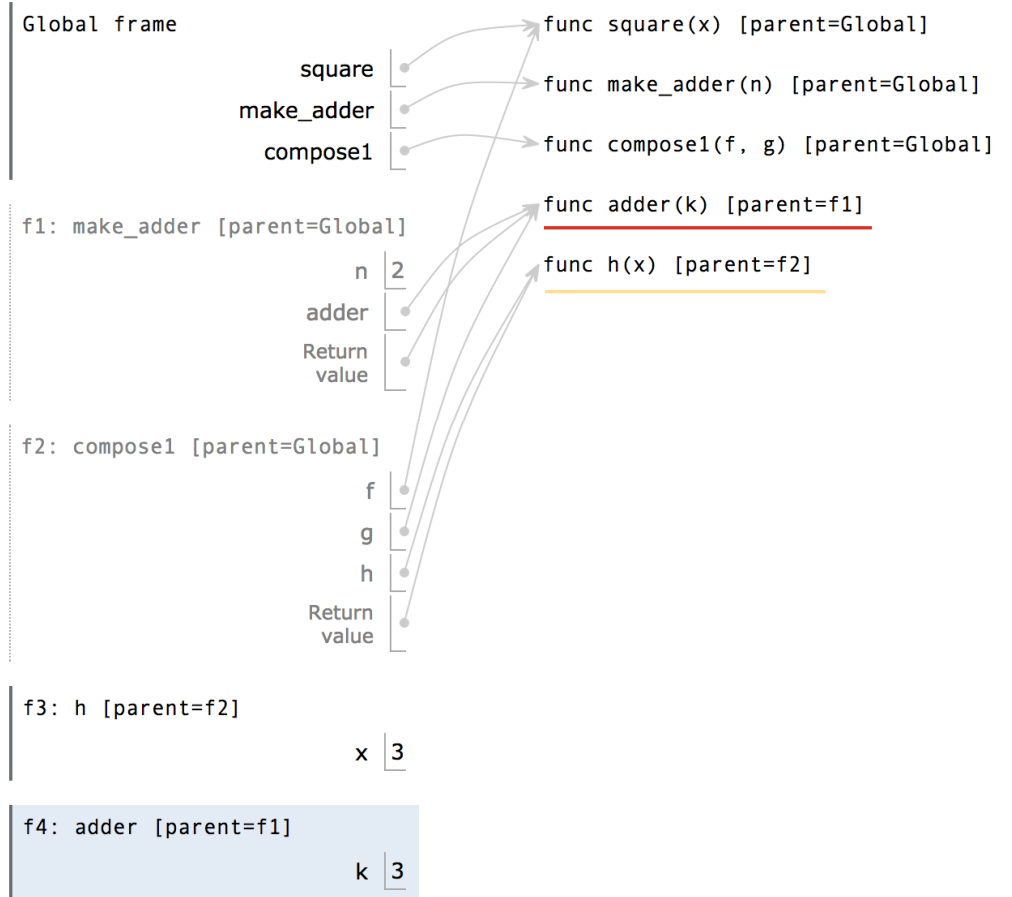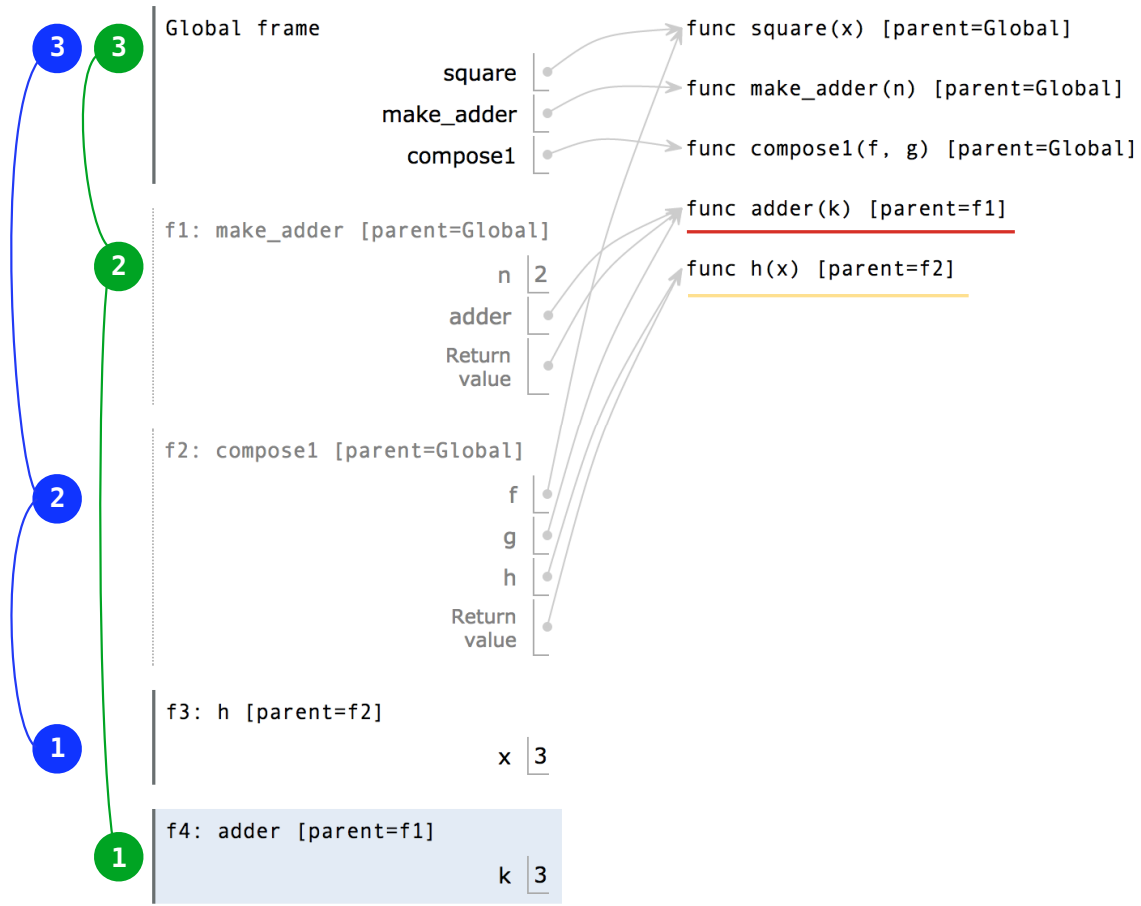
```
1  def square(x):
2      return x * x
3
4  def make_adder(n):
5      def adder(k):
6          return k + n
7      return adder
8
9  def compose1(f, g):
10      def h(x):
11          return f(g(x))
12      return h
13
14  compose1(square, make_adder(2))(3)
```

Return value of make_adder is an argument to compose1

```
Global frame                          func square(x) [parent=Global]
                    square ●─────────▶
               make_adder ●──────────▶ func make_adder(n) [parent=Global]
                 compose1 ●─────────▶ func compose1(f, g) [parent=Global]

f1: make_adder [parent=Global]        func adder(k) [parent=f1]
                        n │ 2
                    adder ●            func h(x) [parent=f2]
             Return
              value ●

f2: compose1 [parent=Global]
                        f ●
                        g ●
                        h ●
             Return
              value ●

f3: h [parent=f2]
                        x │ 3

f4: adder [parent=f1]
                        k │ 3
```

The Environment Diagram for Function Composition

```
1   def square(x):
2       return x * x
3
4   def make_adder(n):
5       def adder(k):
6           return k + n
7       return adder
8
9   def compose1(f, g):
10      def h(x):
11          return f(g(x))
12      return h
13
14  compose1(square, make_adder(2))(3)
```

Return value of make_adder is
an argument to compose1

Global frame

square
make_adder
compose1

func square(x) [parent=Global]
func make_adder(n) [parent=Global]
func compose1(f, g) [parent=Global]
func adder(k) [parent=f1]
func h(x) [parent=f2]

f1: make_adder [parent=Global]

n   2
adder
Return
value

f2: compose1 [parent=Global]

f
g
h
Return
value

f3: h [parent=f2]

x   3

f4: adder [parent=f1]

k   3

Interactive Diagram

13

Lambda Expressions

(Demo)

Lambda Expressions

Lambda Expressions

```
>>> x = 10
```

Lambda Expressions

```
>>> x = 10

>>> square = x * x
```

Lambda Expressions

```
>>> x = 10

>>> square = x * x
```

An expression: this one evaluates to a number

Lambda Expressions

```
>>> x = 10
```

An expression: this one
evaluates to a number

```
>>> square = x * x
```

```
>>> square = lambda x: x * x
```

Lambda Expressions

```
>>> x = 10
```

An expression: this one evaluates to a number

```
>>> square = x * x
```

Also an expression: evaluates to a function

```
>>> square = lambda x: x * x
```

Lambda Expressions

```
>>> x = 10
```

An expression: this one evaluates to a number

```
>>> square = x * x
```

Also an expression: evaluates to a function

```
>>> square = lambda x: x * x
```

A function

Lambda Expressions

```
>>> x = 10
```

An expression: this one evaluates to a number

```
>>> square = x * x
```

Also an expression: evaluates to a function

```
>>> square = lambda x: x * x
```

A function

with formal parameter x

Lambda Expressions

```
>>> x = 10
```

An expression: this one evaluates to a number

```
>>> square = x * x
```

Also an expression: evaluates to a function

```
>>> square = lambda x: x * x
```

A function

with formal parameter x

that returns the value of "x * x"

Lambda Expressions

```
>>> x = 10
```

An expression: this one evaluates to a number

```
>>> square = x * x
```

Also an expression: evaluates to a function

```
>>> square = lambda x: x * x
```

Important: No "return" keyword!

A function

with formal parameter x

that returns the value of "x * x"

Lambda Expressions

```
>>> x = 10
```

An expression: this one evaluates to a number

```
>>> square = x * x
```

Also an expression: evaluates to a function

```
>>> square = lambda x: x * x
```

Important: No "return" keyword!

A function

with formal parameter x

that returns the value of "x * x"

Must be a single expression

Lambda Expressions

```
>>> x = 10
```

An expression: this one evaluates to a number

```
>>> square = x * x
```

Also an expression: evaluates to a function

```
>>> square = lambda x: x * x
```

A function

with formal parameter x

that returns the value of "x * x"

Important: No "return" keyword!

Must be a single expression

```
>>> square(4)
16
```

Lambda Expressions

```
>>> x = 10
```

An expression: this one evaluates to a number

```
>>> square = x * x
```

Also an expression: evaluates to a function

```
>>> square = lambda x: x * x
```

A function

with formal parameter x

that returns the value of "x * x"

Important: No "return" keyword!

Must be a single expression

```
>>> square(4)
16
```

Lambda expressions are not common in Python, but important in general

Lambda Expressions

```
>>> x = 10
```

An expression: this one evaluates to a number

```
>>> square = x * x
```

Also an expression: evaluates to a function

```
>>> square = lambda x: x * x
```

Important: No "return" keyword!

A function

with formal parameter x

that returns the value of "x * x"

```
>>> square(4)
16
```

Must be a single expression

Lambda expressions are not common in Python, but important in general

Lambda expressions in Python cannot contain statements at all!

Lambda Expressions Versus Def Statements

VS

Lambda Expressions Versus Def Statements

`square = lambda x: x * x`   **VS**

Lambda Expressions Versus Def Statements


```
square = lambda x: x * x
```

VS

```
def square(x):
    return x * x
```

Lambda Expressions Versus Def Statements

```
square = lambda x: x * x
```

VS

```
def square(x):
    return x * x
```

- Both create a function with the same domain, range, and behavior.

Lambda Expressions Versus Def Statements

`square = lambda x: x * x`

VS

```
def square(x):
    return x * x
```

- Both create a function with the same domain, range, and behavior.

- Both functions have as their parent the frame in which they were defined.

Lambda Expressions Versus Def Statements

square = lambda x: x * x   **VS**   def square(x):
                                                      return x * x

- Both create a function with the same domain, range, and behavior.

- Both functions have as their parent the frame in which they were defined.

- Both bind that function to the name square.

Lambda Expressions Versus Def Statements

`square = lambda x: x * x`   **VS**   `def square(x):`
`    return x * x`

- Both create a function with the same domain, range, and behavior.

- Both functions have as their parent the frame in which they were defined.

- Both bind that function to the name square.

- Only the def statement gives the function an intrinsic name.

Lambda Expressions Versus Def Statements

$$square = lambda\ x: x * x$$

VS

```
def square(x):
    return x * x
```

- Both create a function with the same domain, range, and behavior.

- Both functions have as their parent the frame in which they were defined.

- Both bind that function to the name square.

- Only the def statement gives the function an intrinsic name.


```
Global frame
                           square  ●─────►func λ(x) <line 1> [parent=Global]

f1: λ <line 1> [parent=Global]
                      x  4
                 Return  16
                  value
```

Lambda Expressions Versus Def Statements
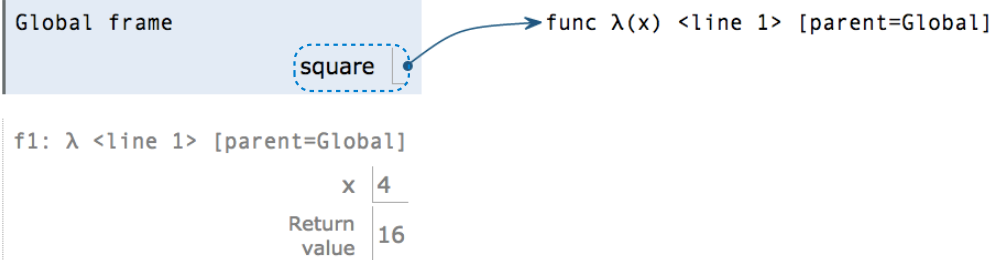
square = lambda x: x * x   **VS**   def square(x):
                                                      return x * x

- Both create a function with the same domain, range, and behavior.

- Both functions have as their parent the frame in which they were defined.

- Both bind that function to the name square.

- Only the def statement gives the function an intrinsic name.

```
Global frame                            ──→ func λ(x) <line 1> [parent=Global]
                    square  ●

f1: λ <line 1> [parent=Global]
                         x   4
                    Return
                    value   16
```

Lambda Expressions Versus Def Statements

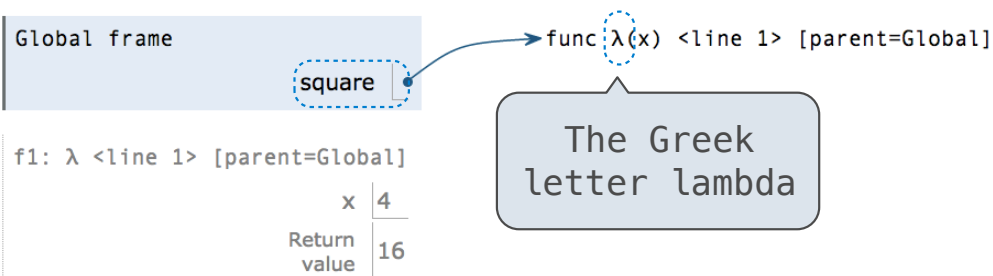square = lambda x: x * x   **VS**   def square(x):
                                                  return x * x

- Both create a function with the same domain, range, and behavior.

- Both functions have as their parent the frame in which they were defined.

- Both bind that function to the name square.

- Only the def statement gives the function an intrinsic name.

Global frame

square

func λ(x) <line 1> [parent=Global]

f1: λ <line 1> [parent=Global]

x 4

Return value 16

The Greek letter lambda

Lambda Expressions Versus Def Statements

square = lambda x: x * x   **VS**   def square(x):
                                                  return x * x

- Both create a function with the same domain, range, and behavior.

- Both functions have as their parent the frame in which they were defined.

- Both bind that function to the name square.

- Only the def statement gives the function an intrinsic name.


```
Global frame
                    square
f1: λ <line 1> [parent=Global]
              x   4
     Return
     value     16
```

func λ(x) <line 1> [parent=Global]

The Greek
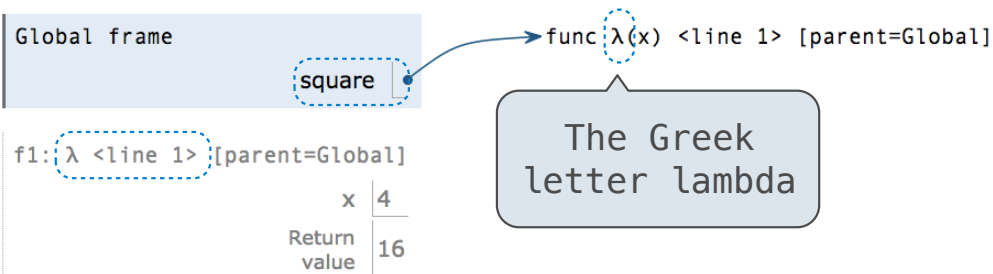letter lambda

Lambda Expressions Versus Def Statements

square = lambda x: x * x

VS

```
def square(x):
    return x * x
```

- Both create a function with the same domain, range, and behavior.

- Both functions have as their parent the frame in which they were defined.

- Both bind that function to the name square.

- Only the def statement gives the function an intrinsic name.

Global frame → func λ(x) <line 1> [parent=Global]

square

f1: λ <line 1> [parent=Global]

x | 4

Return value | 16

The Greek letter lambda

Global frame → func square(x) [parent=Global]

square

f1: square [parent=Global]

x | 4

Return value | 16