61A Lecture 5

Wednesday, September 11

Announcements

Announcements

- Take-home quiz released Wednesday 9/11 at 1pm, due Thursday 9/12 at 11:59pm.

Announcements

- Take-home quiz released Wednesday 9/11 at 1pm, due Thursday 9/12 at 11:59pm.
  - http://inst.eecs.berkeley.edu/~cs61a/fa13/hw/quiz1.html

Announcements

- Take-home quiz released Wednesday 9/11 at 1pm, due Thursday 9/12 at 11:59pm.
  - http://inst.eecs.berkeley.edu/~cs61a/fa13/hw/quiz1.html
  - 3 points; graded for correctness.

Announcements

- Take-home quiz released Wednesday 9/11 at 1pm, due Thursday 9/12 at 11:59pm.
  - http://inst.eecs.berkeley.edu/~cs61a/fa13/hw/quiz1.html
  - 3 points; graded for correctness.
  - Submit in the same way that you submit homework assignments.

Announcements

- Take-home quiz released Wednesday 9/11 at 1pm, due Thursday 9/12 at 11:59pm.
  - http://inst.eecs.berkeley.edu/~cs61a/fa13/hw/quiz1.html
  - 3 points; graded for correctness.
  - Submit in the same way that you submit homework assignments.
  - If you receive 0/3, you will need to talk to the course staff or be dropped.

Announcements

- Take-home quiz released Wednesday 9/11 at 1pm, due Thursday 9/12 at 11:59pm.
  - http://inst.eecs.berkeley.edu/~cs61a/fa13/hw/quiz1.html
  - 3 points; graded for correctness.
  - Submit in the same way that you submit homework assignments.
  - If you receive 0/3, you will need to talk to the course staff or be dropped.
  - *Open-computer*: You can use the Python interpreter, watch course videos, and read the online text (http://composingprograms.com).

Announcements

- Take-home quiz released Wednesday 9/11 at 1pm, due Thursday 9/12 at 11:59pm.
  - http://inst.eecs.berkeley.edu/~cs61a/fa13/hw/quiz1.html
  - 3 points; graded for correctness.
  - Submit in the same way that you submit homework assignments.
  - If you receive 0/3, you will need to talk to the course staff or be dropped.
  - *Open-computer*: You can use the Python interpreter, watch course videos, and read the online text (http://composingprograms.com).
  - *No external resources*: Please don't search for answers, talk to your classmates, etc.

Announcements

- Take-home quiz released Wednesday 9/11 at 1pm, due Thursday 9/12 at 11:59pm.
  - http://inst.eecs.berkeley.edu/~cs61a/fa13/hw/quiz1.html
  - 3 points; graded for correctness.
  - Submit in the same way that you submit homework assignments.
  - If you receive 0/3, you will need to talk to the course staff or be dropped.
  - *Open-computer*: You can use the Python interpreter, watch course videos, and read the online text (http://composingprograms.com).
  - *No external resources*: Please don't search for answers, talk to your classmates, etc.
- Homework 2 due Tuesday 9/17 at 5pm.

Announcements

- Take-home quiz released Wednesday 9/11 at 1pm, due Thursday 9/12 at 11:59pm.
  - http://inst.eecs.berkeley.edu/~cs61a/fa13/hw/quiz1.html
  - 3 points; graded for correctness.
  - Submit in the same way that you submit homework assignments.
  - If you receive 0/3, you will need to talk to the course staff or be dropped.
  - *Open-computer*: You can use the Python interpreter, watch course videos, and read the online text (http://composingprograms.com).
  - *No external resources*: Please don't search for answers, talk to your classmates, etc.
- Homework 2 due Tuesday 9/17 at 5pm.
- Project 1 due Thursday 9/19 at 11:59pm.

Announcements

- Take-home quiz released Wednesday 9/11 at 1pm, due Thursday 9/12 at 11:59pm.
  - http://inst.eecs.berkeley.edu/~cs61a/fa13/hw/quiz1.html
  - 3 points; graded for correctness.
  - Submit in the same way that you submit homework assignments.
  - If you receive 0/3, you will need to talk to the course staff or be dropped.
  - *Open-computer*: You can use the Python interpreter, watch course videos, and read the online text (http://composingprograms.com).
  - *No external resources*: Please don't search for answers, talk to your classmates, etc.
- Homework 2 due Tuesday 9/17 at 5pm.
- Project 1 due Thursday 9/19 at 11:59pm.
- Solutions to homeworks: http://inst.eecs.berkeley.edu/~cs61a/fa13/hw/solutions

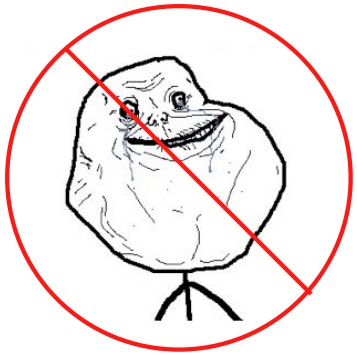
Office Hours: You Should Go!

Office Hours: You Should Go!

You are not alone!

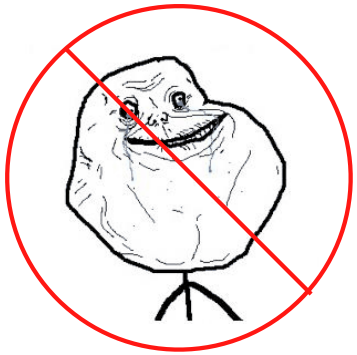
Office Hours: You Should Go!

You are not alone!



Office Hours: You Should Go!

You are not alone!



http://inst.eecs.berkeley.edu/~cs61a/fa13/staff.html

The Purpose of Higher-Order Functions

The Purpose of Higher-Order Functions

Functions are first-class: Functions can be manipulated as values in our programming language.

The Purpose of Higher-Order Functions

Functions are first-class: Functions can be manipulated as values in our programming language.

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

The Purpose of Higher-Order Functions

Functions are first-class: Functions can be manipulated as values in our programming language.

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

Higher-order functions:

The Purpose of Higher-Order Functions

Functions are first-class: Functions can be manipulated as values in our programming language.

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

Higher-order functions:

- Express general methods of computation

The Purpose of Higher-Order Functions

Functions are first-class: Functions can be manipulated as values in our programming language.

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

Higher-order functions:

- Express general methods of computation
- Remove repetition from programs

The Purpose of Higher-Order Functions

Functions are first-class: Functions can be manipulated as values in our programming language.

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

Higher-order functions:

- Express general methods of computation
- Remove repetition from programs
- Separate concerns among functions

Environments for Higher-Order Functions

Environments Enable Higher-Order Functions

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

Environments Enable Higher-Order Functions

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

Functions as arguments:

Environments Enable Higher-Order Functions

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

Functions as arguments:

Our current evaluation rules handle that case already!

Environments Enable Higher-Order Functions

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

Functions as arguments:

Our current evaluation rules handle that case already!

We'll discuss an example today

Environments Enable Higher-Order Functions

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

Functions as arguments:

Our current evaluation rules handle that case already!

We'll discuss an example today

Functions as return values:

Environments Enable Higher-Order Functions

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

Functions as arguments:

Our current evaluation rules handle that case already!

We'll discuss an example today

Functions as return values:

We need to extend our rules a little

Environments Enable Higher-Order Functions

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

Functions as arguments:

Our current evaluation rules handle that case already!

We'll discuss an example today

Functions as return values:

We need to extend our rules a little

Functions need to know where they were defined

Environments Enable Higher-Order Functions

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

Functions as arguments:

Our current evaluation rules handle that case already!

We'll discuss an example today

Functions as return values:

We need to extend our rules a little

Functions need to know where they were defined

Almost everything stays the same

Environments Enable Higher-Order Functions

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

Functions as arguments:

Our current evaluation rules handle that case already!

We'll discuss an example today

Functions as return values:

We need to extend our rules a little

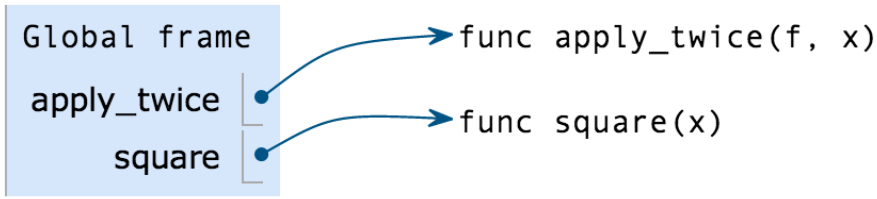
Functions need to know where they were defined

Almost everything stays the same

(demo)

Names can be Bound to Functional Arguments

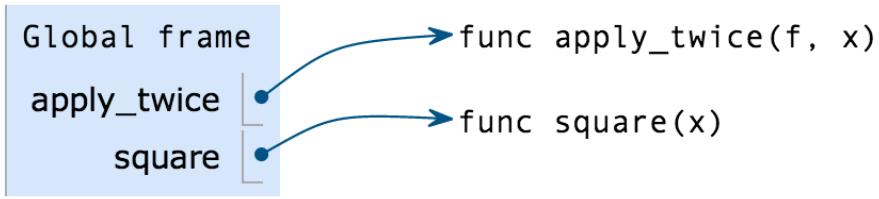
```
1  def apply_twice(f, x):
2      return f(f(x))
3
4  def square(x):
5      return x * x
6
7  result = apply_twice(square, 2)
```



Example: http://goo.gl/mwVuIF

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)
```

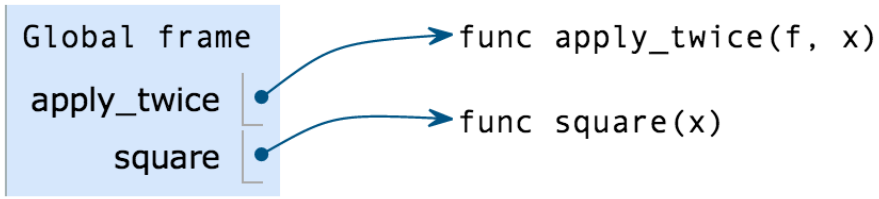


Example: http://goo.gl/mwVuIF

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)
```

func apply_twice(f, x)



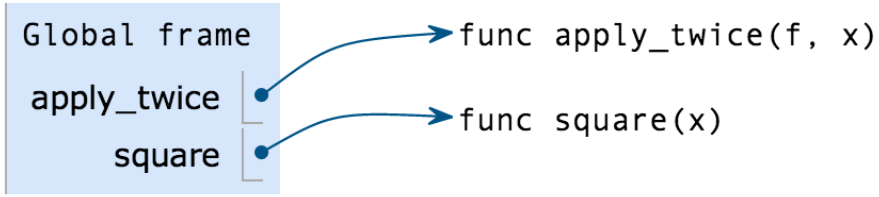
Example: http://goo.gl/mwVuIF

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)
```

func apply_twice(f, x)

func square(x)

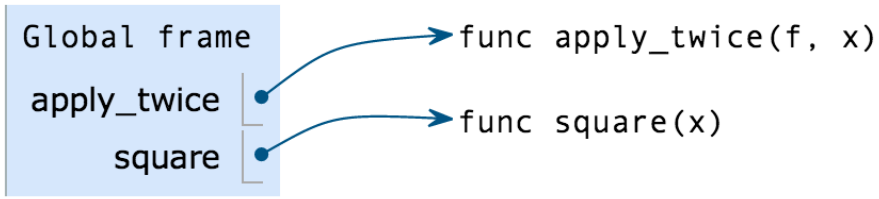


Example: http://goo.gl/mwVuIF

Names can be Bound to Functional Arguments

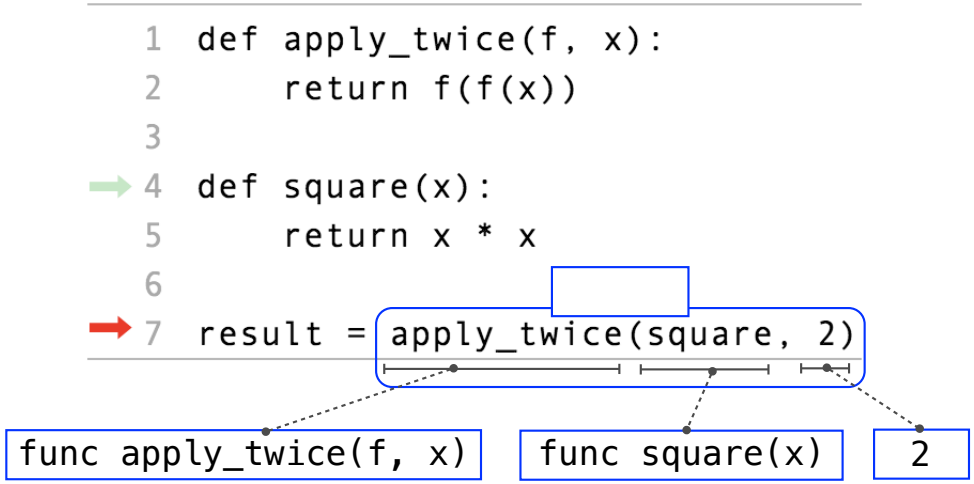
```
1  def apply_twice(f, x):
2      return f(f(x))
3
4  def square(x):
5      return x * x
6
7  result = apply_twice(square, 2)
```

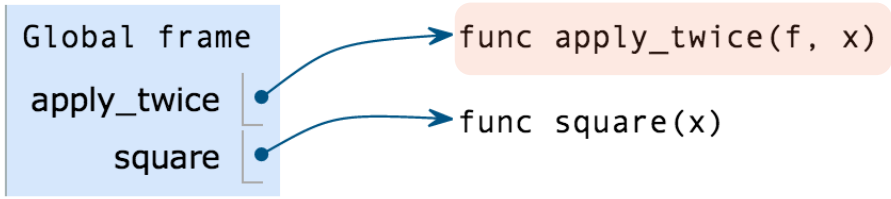
func apply_twice(f, x) | func square(x) | 2



Example: http://goo.gl/mwVuIF

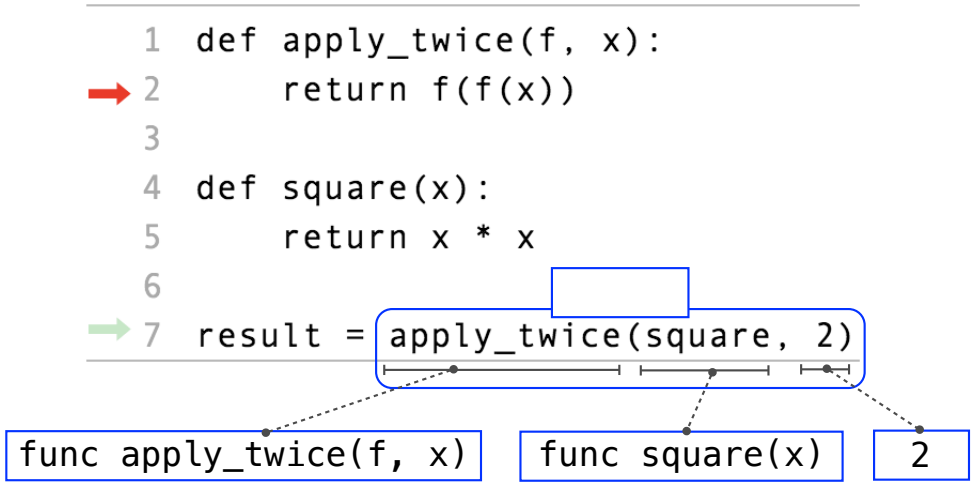
Names can be Bound to Functional Arguments

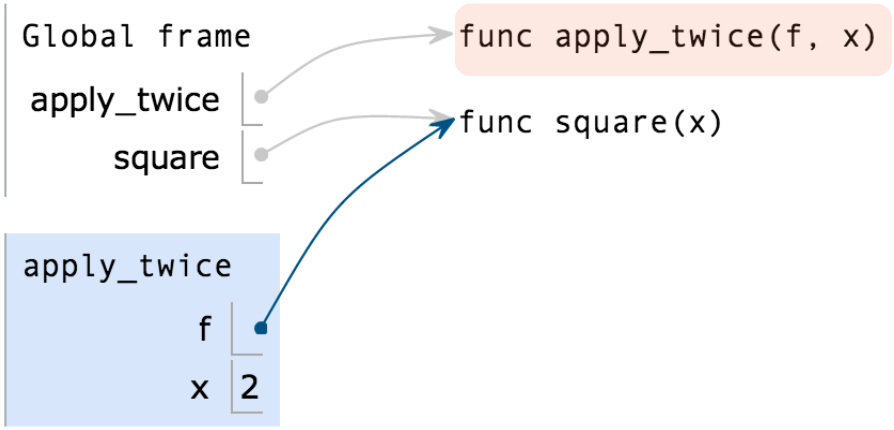




Example: http://goo.gl/mwVuIF

Names can be Bound to Functional Arguments



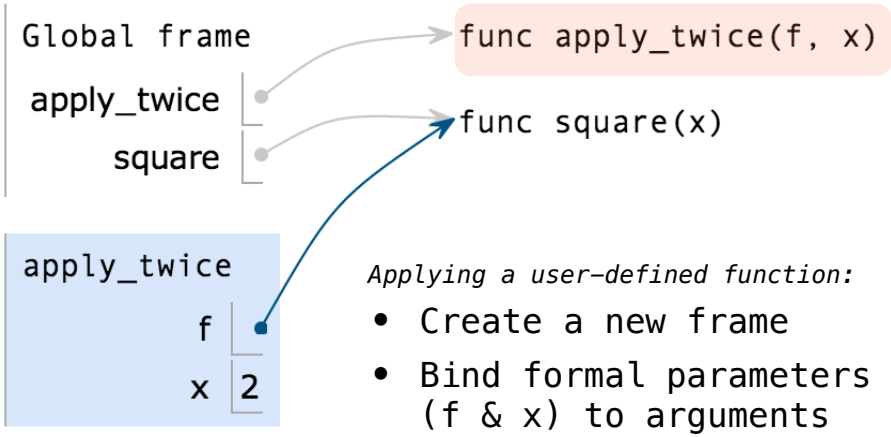


Example: http://goo.gl/mwVuIF

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)
```

func apply_twice(f, x) | func square(x) | 2



Applying a user-defined function:

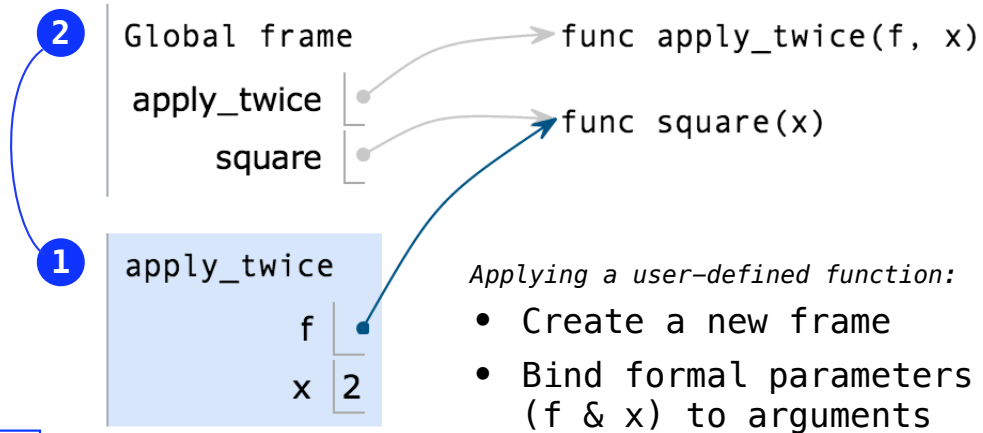
- Create a new frame
- Bind formal parameters (f & x) to arguments
- Execute the body: **return** f(f(x))

Example: http://goo.gl/mwVuIF

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)
```

func apply_twice(f, x) | func square(x) | 2

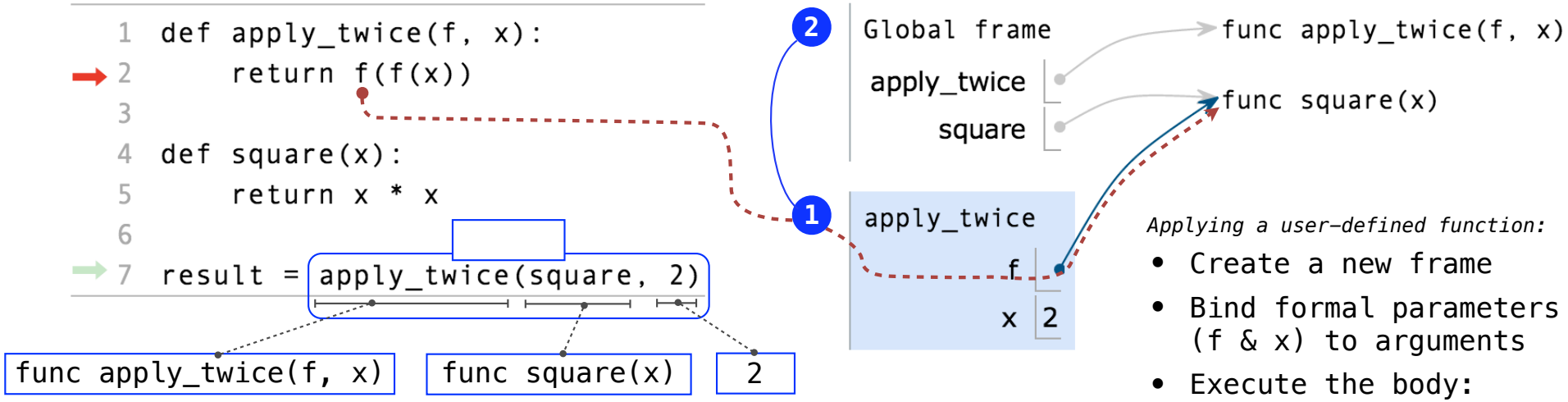


Applying a user-defined function:

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

Example: http://goo.gl/mwVuIF

Names can be Bound to Functional Arguments

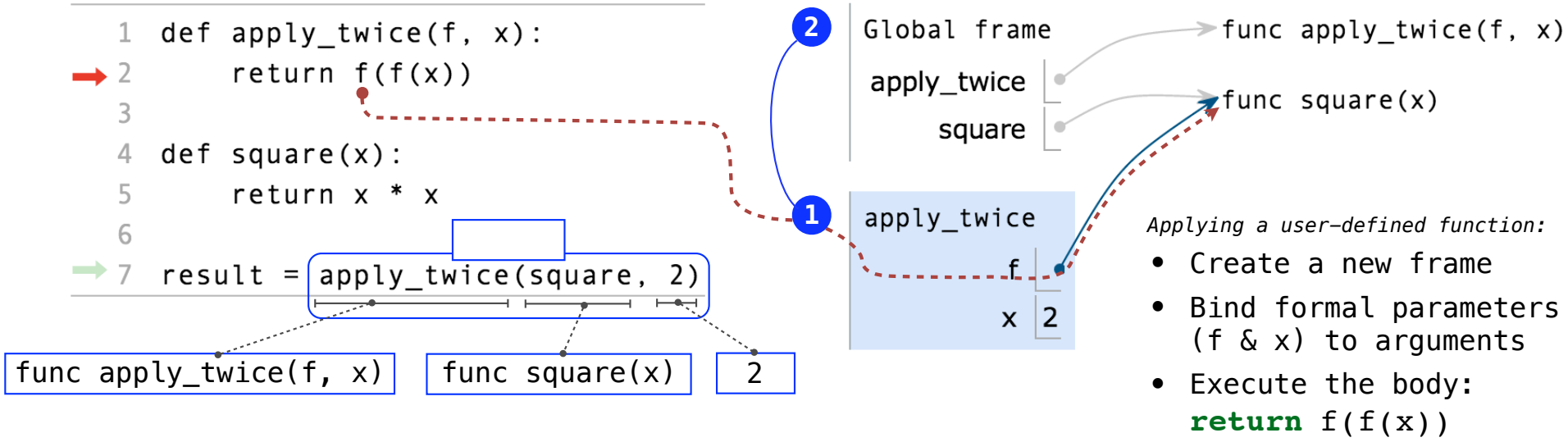


Applying a user-defined function:

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

Example: http://goo.gl/mwVuIF

Names can be Bound to Functional Arguments



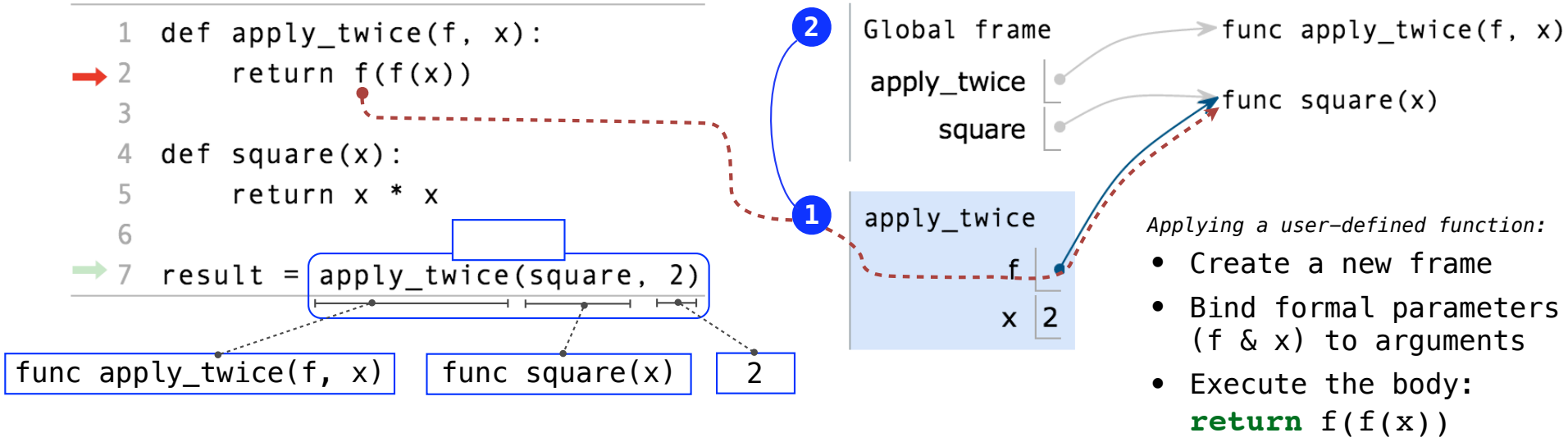
Applying a user-defined function:

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

- Functions are values.

Example: http://goo.gl/mwVuIF

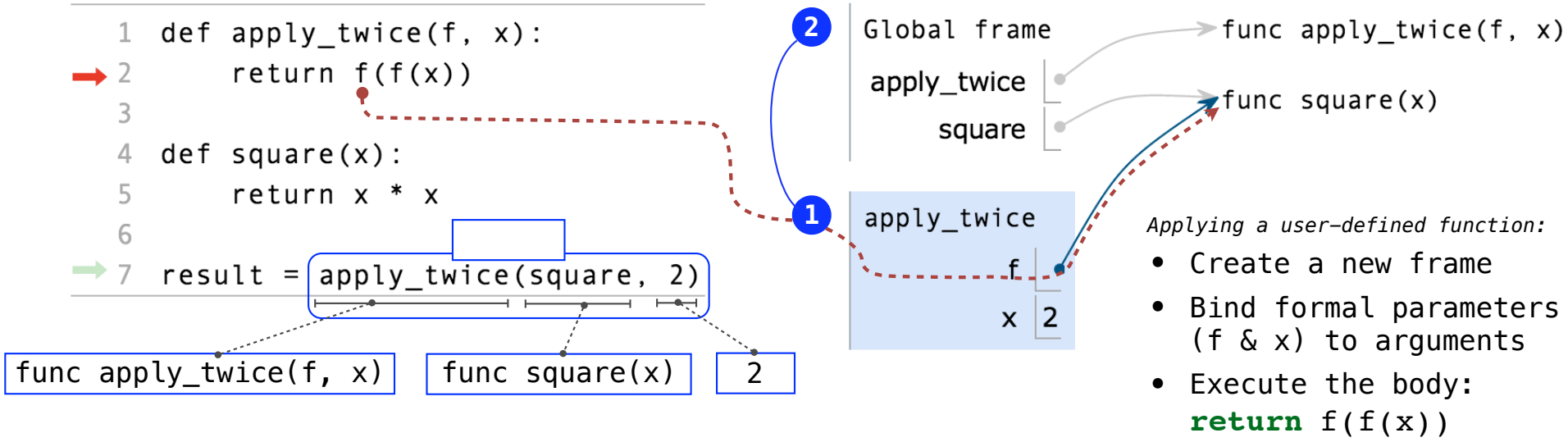
Names can be Bound to Functional Arguments



- Functions are values.
- Names can refer to functions (just as they can refer to any values).

Example: http://goo.gl/mwVuIF

Names can be Bound to Functional Arguments



- Functions are values.
- Names can refer to functions (just as they can refer to any values).
- Multiple names can all refer to the same function, even in different frames.

Example: http://goo.gl/mwVuIF

Discussion Question

What is the value of the final expression below?

```
def repeat(f, x):
    while f(x) != x:
        x = f(x)
    return x

def g(y):
    return (y + 5) // 3

repeat(g, 5)
```

Example: http://goo.gl/EDiOIr

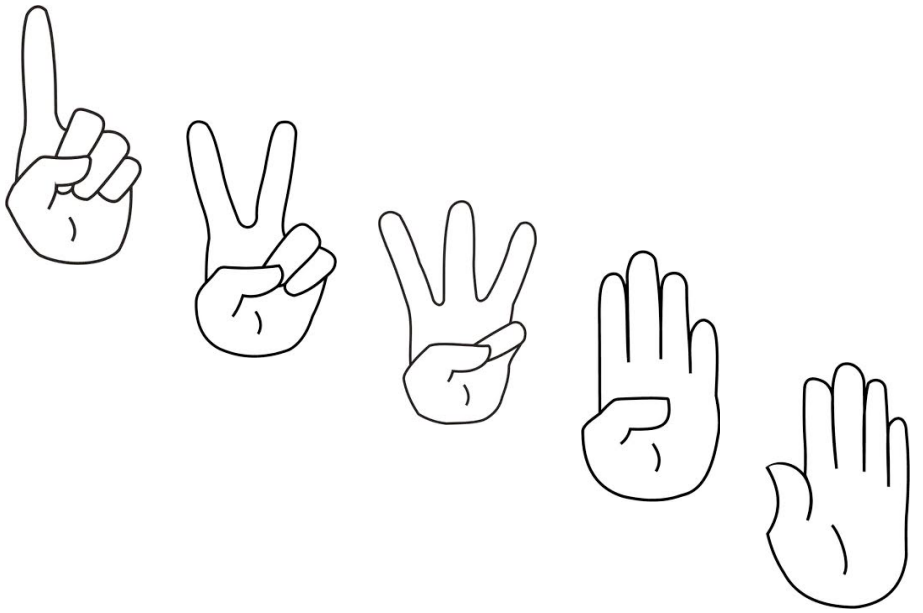
Discussion Question

What is the value of the final expression below?

```
def repeat(f, x):
    while f(x) != x:
        x = f(x)
    return x

def g(y):
    return (y + 5) // 3

repeat(g, 5)
```



Example: http://goo.gl/EDiOIr

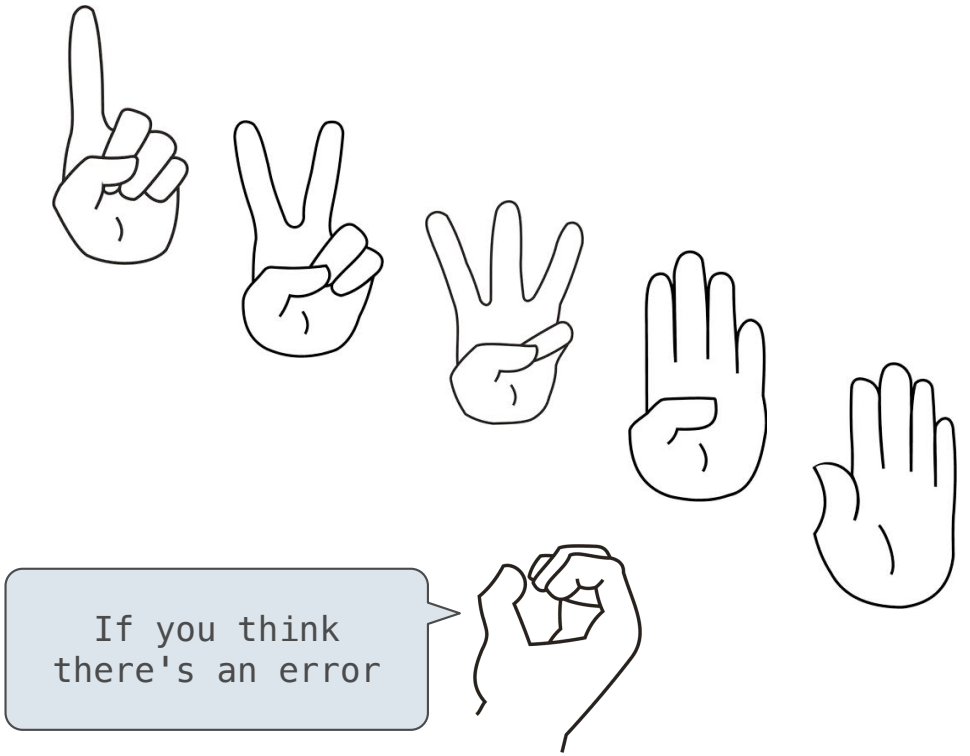
Discussion Question

What is the value of the final expression below?

```
def repeat(f, x):
    while f(x) != x:
        x = f(x)
    return x

def g(y):
    return (y + 5) // 3

repeat(g, 5)
```



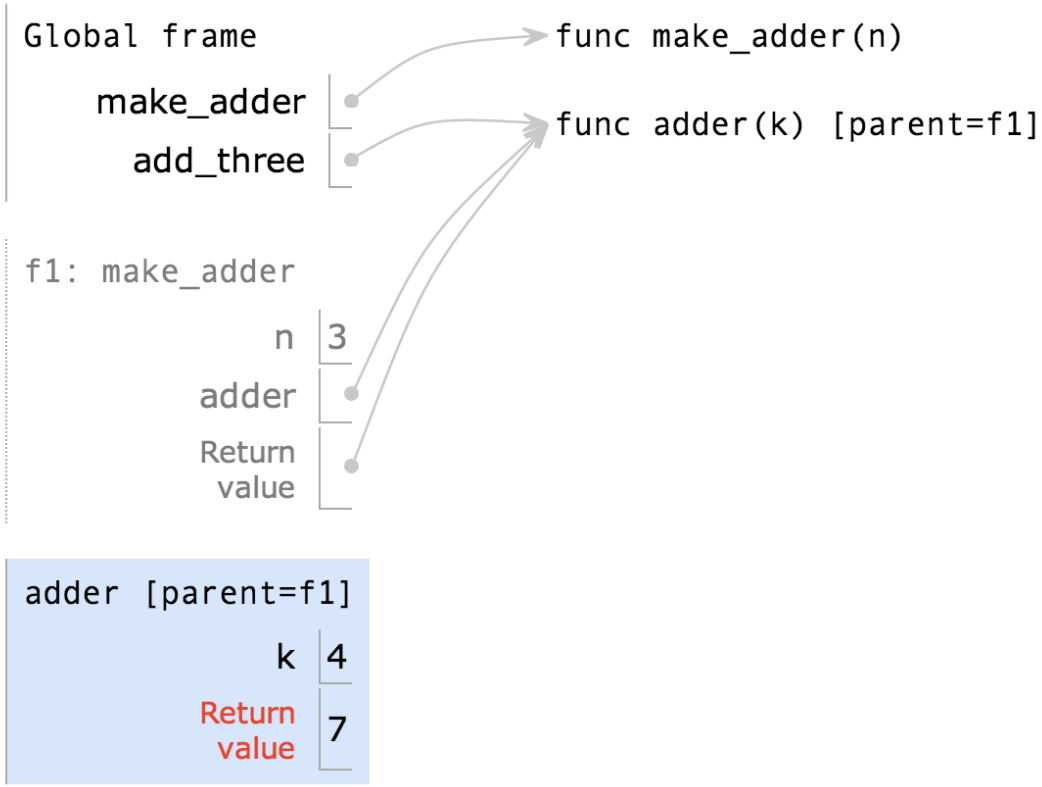
Example: http://goo.gl/EDiOIr

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)
```

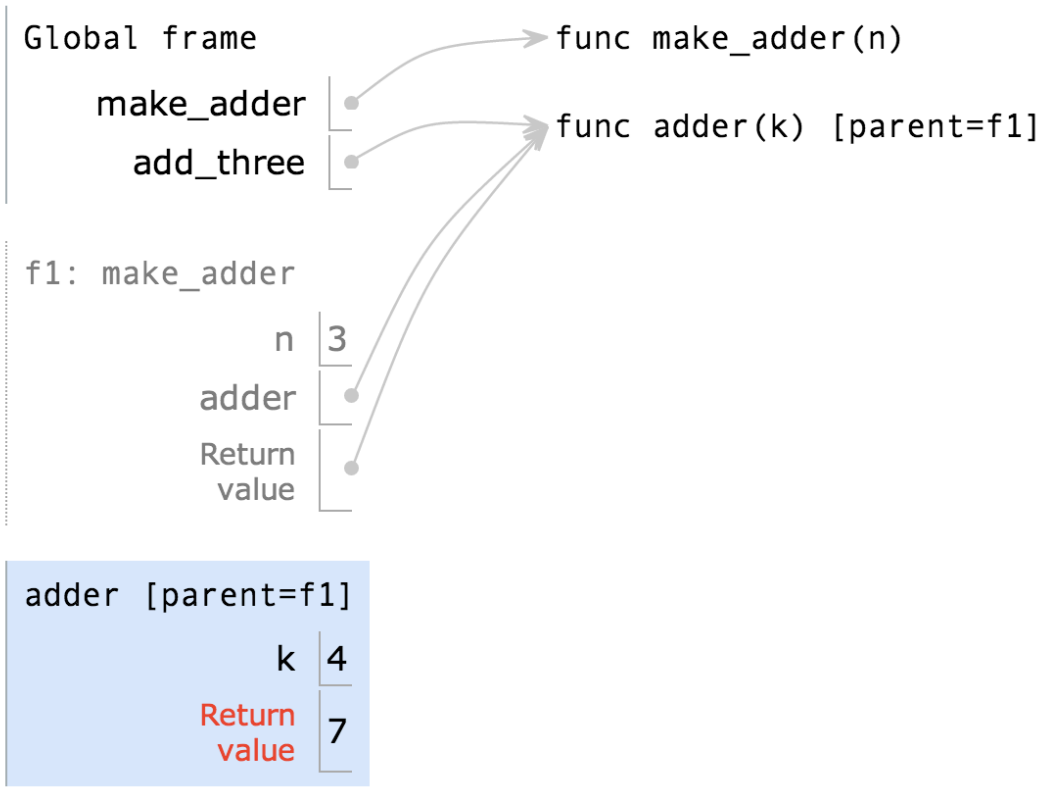


Example:

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)
```

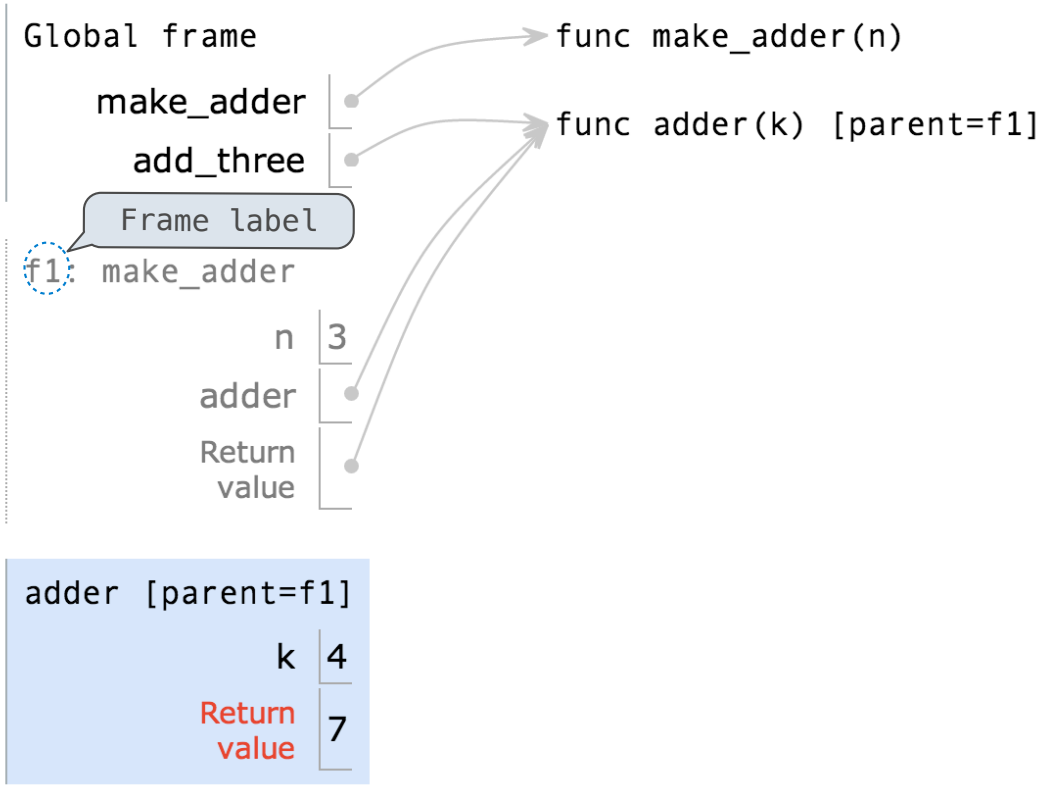


Example:

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)
```

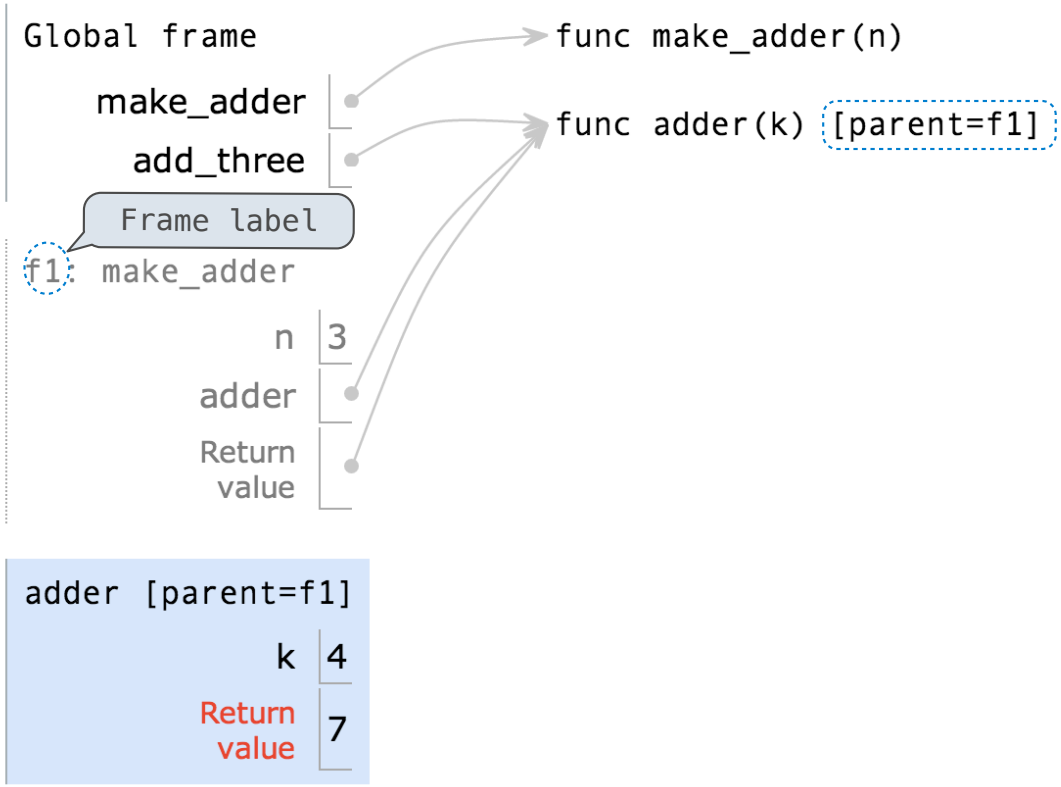


Example:

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)
```

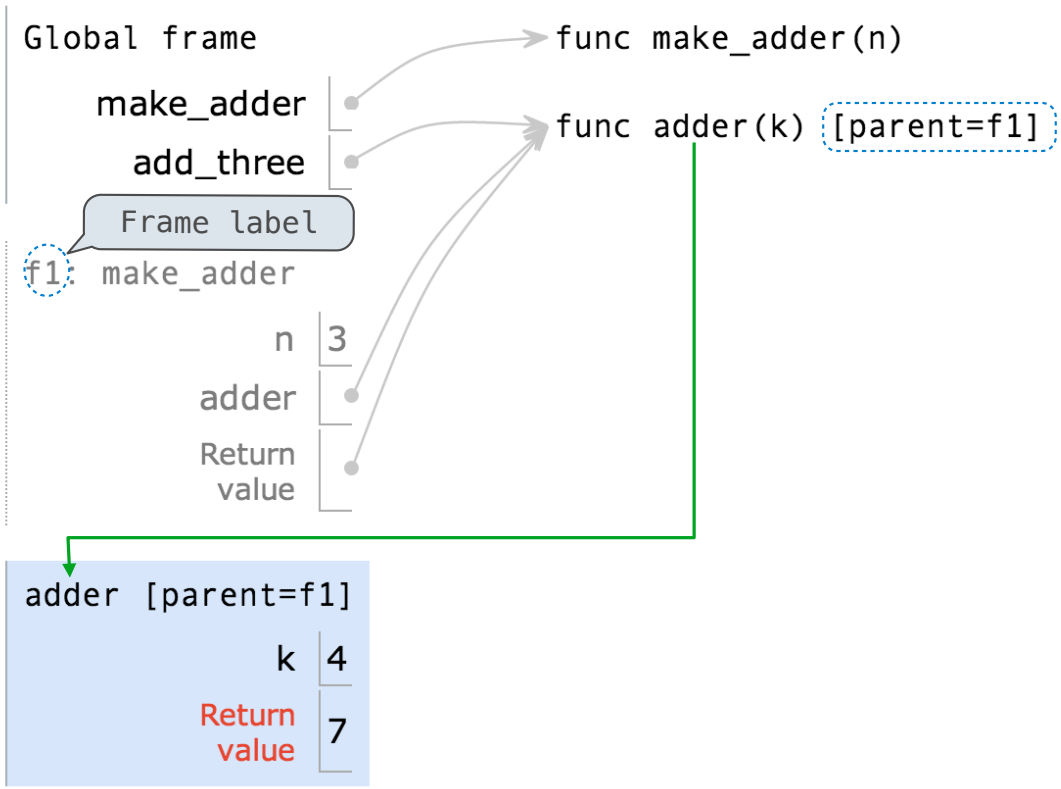


Example:

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)
```

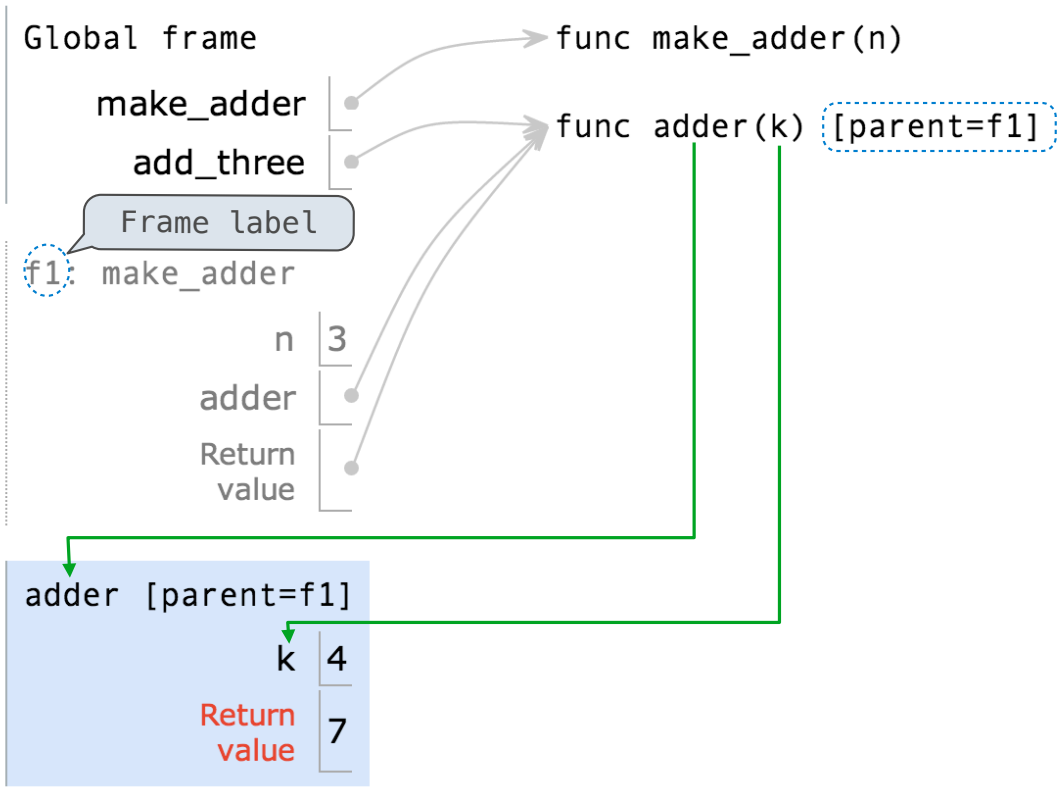


Example:

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)
```

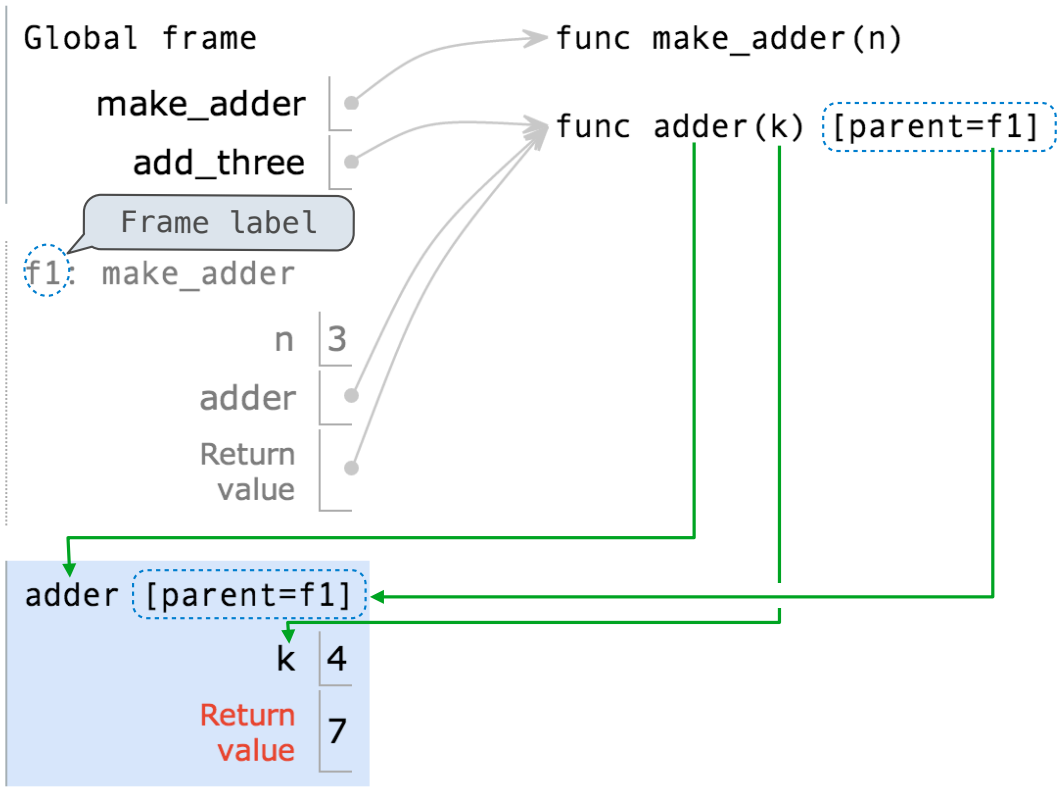


Example:

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)
```

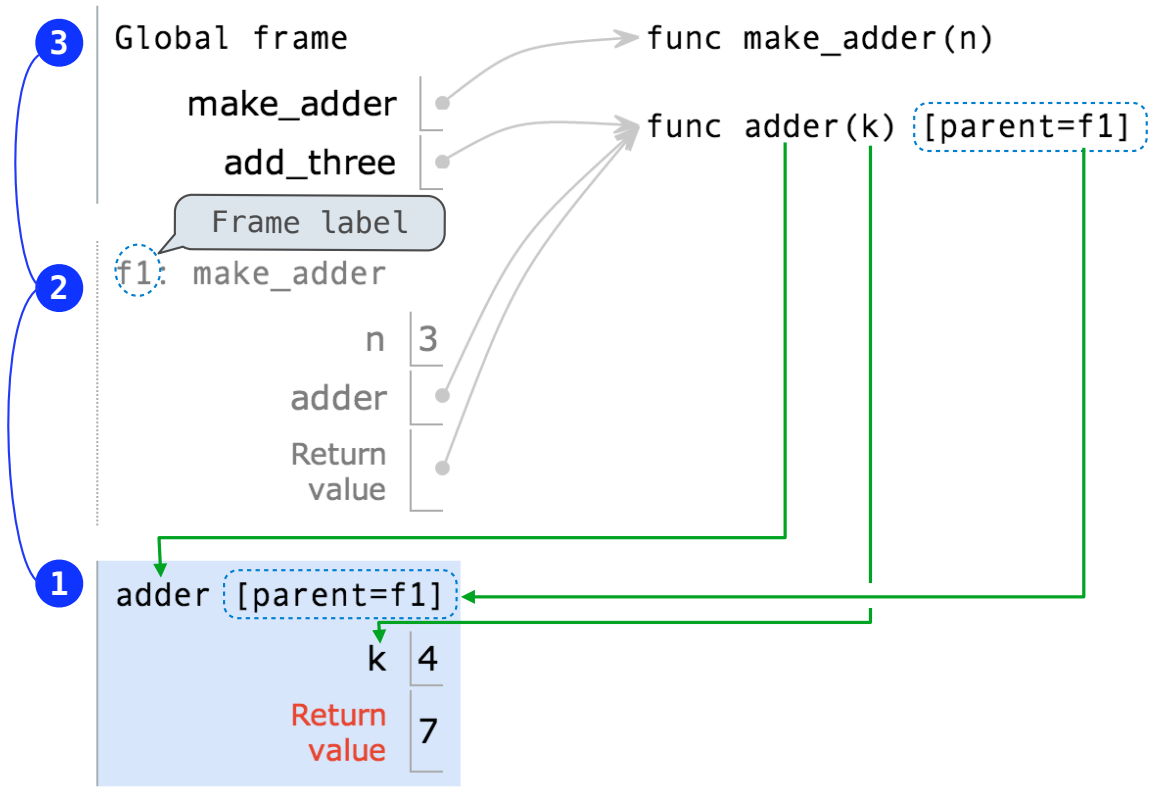


Example:

Environment Diagrams for Nested Def Statements

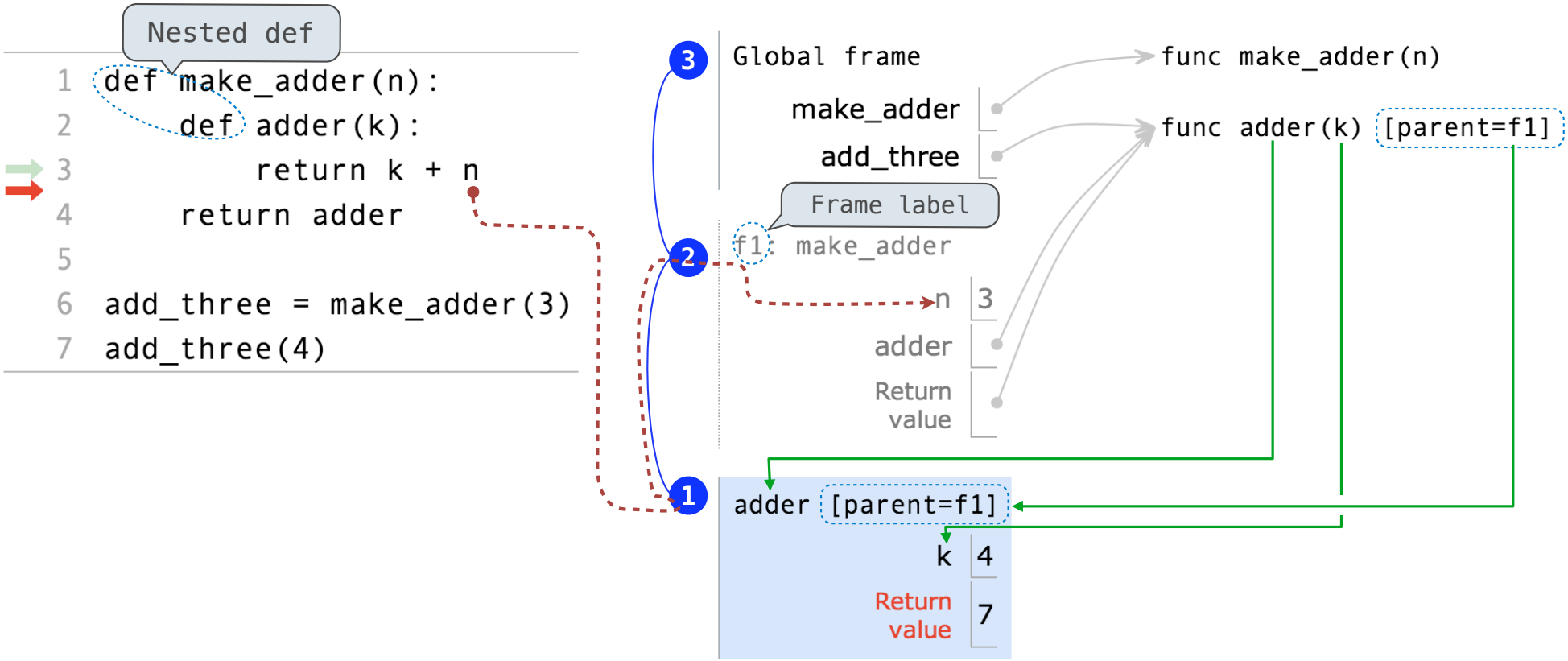
Nested def

```
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)
```



Example:

Environment Diagrams for Nested Def Statements



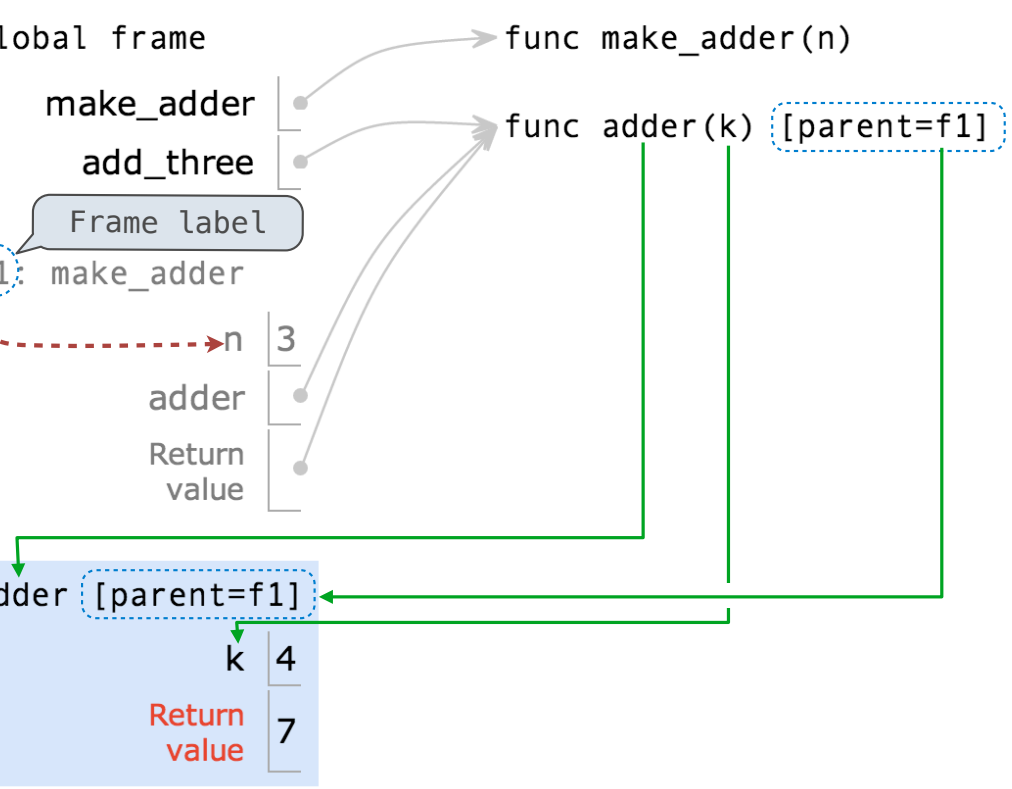
Example:

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)



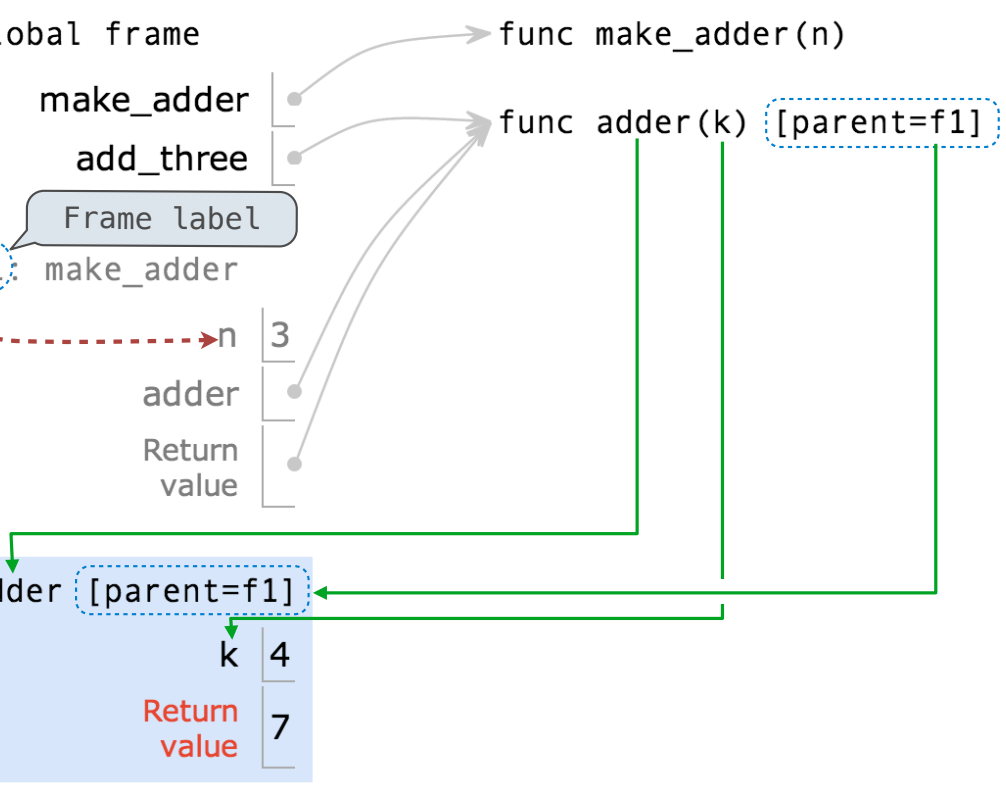
Example:

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)
```

Nested def

- Every user-defined **function** has a ***parent frame*** (often global)
- The parent of a **function** is the frame in which it was *defined*

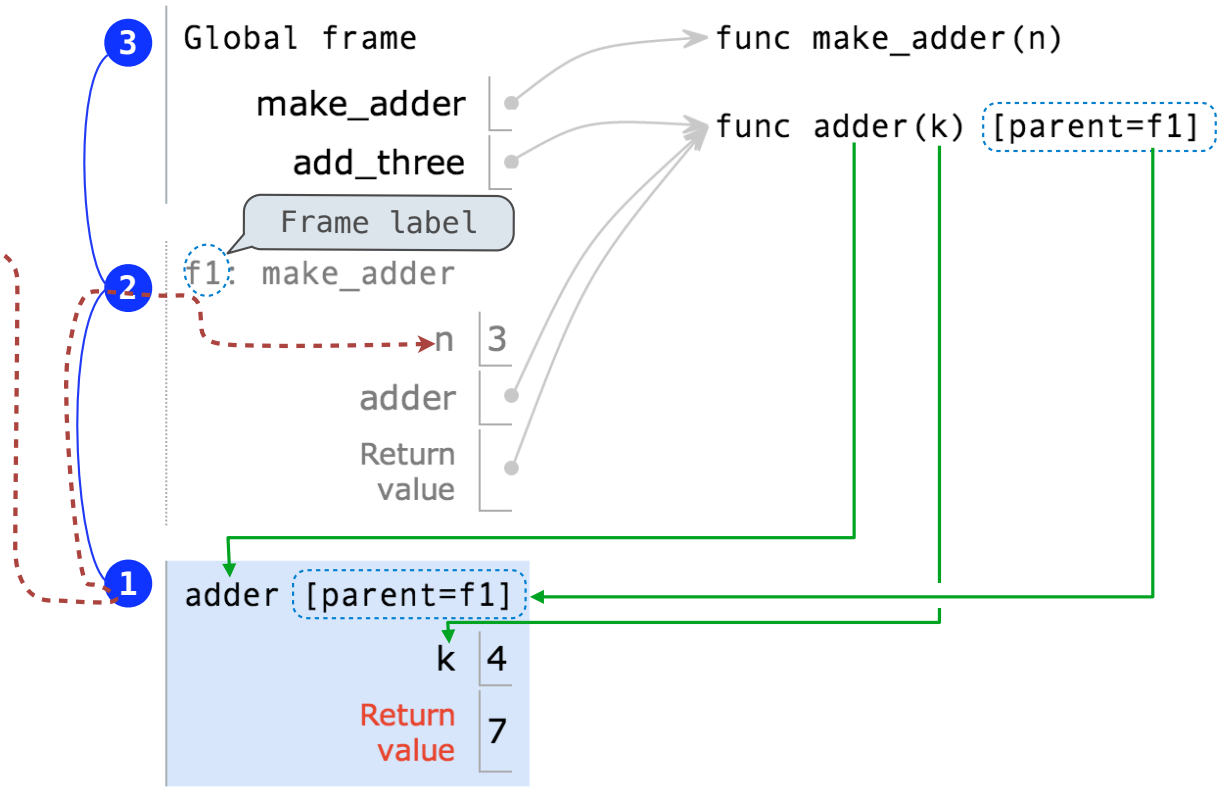


Example:

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)
```



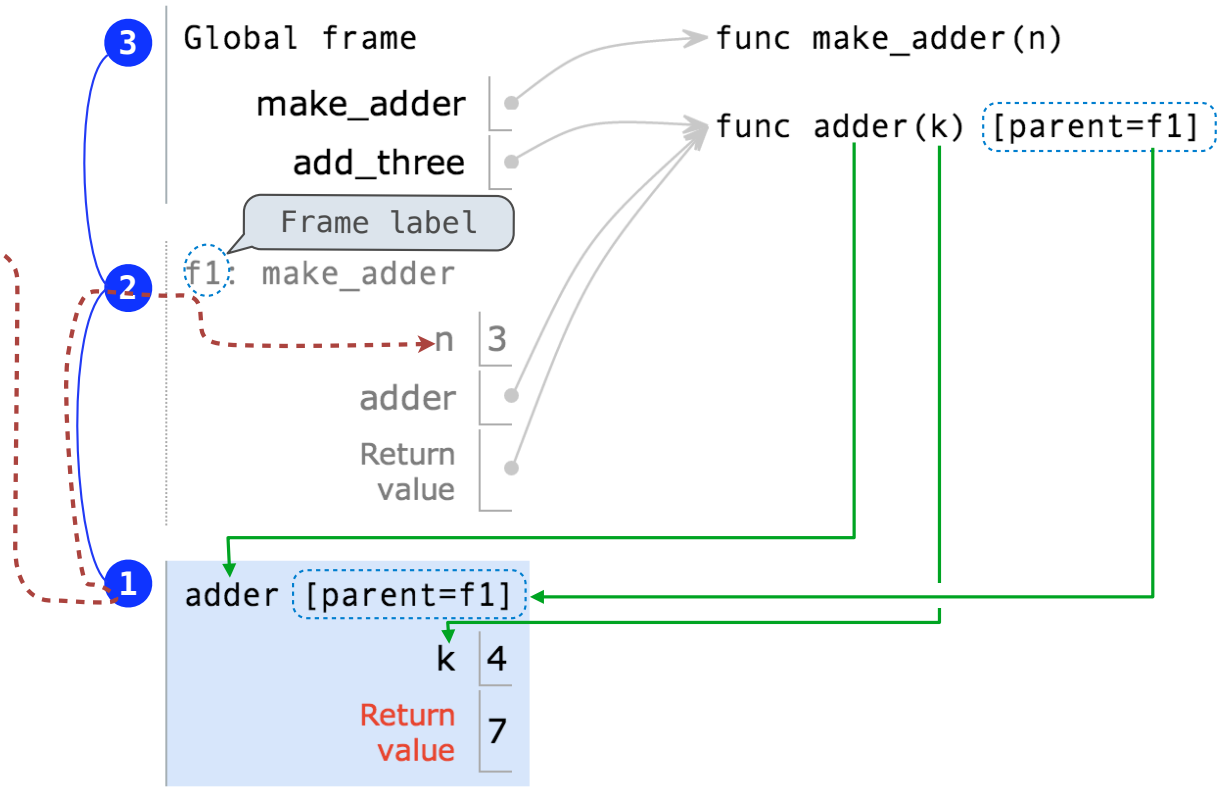
- Every user-defined **function** has a ***parent frame*** (often global)
- The parent of a **function** is the frame in which it was *defined*
- Every local **frame** has a ***parent frame*** (often global)

Example:

Environment Diagrams for Nested Def Statements

Nested def

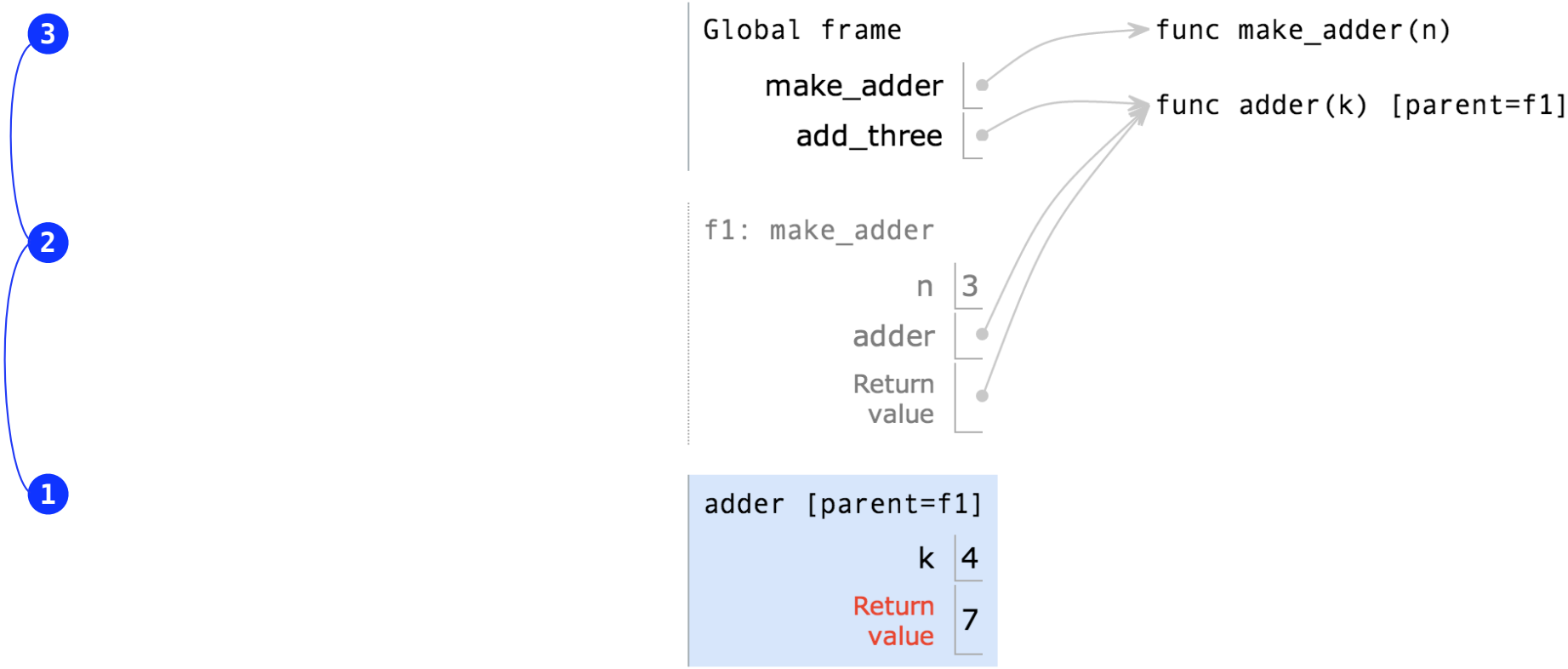
```
1 def make_adder(n):
2     def adder(k):
3         return k + n
4     return adder
5
6 add_three = make_adder(3)
7 add_three(4)
```



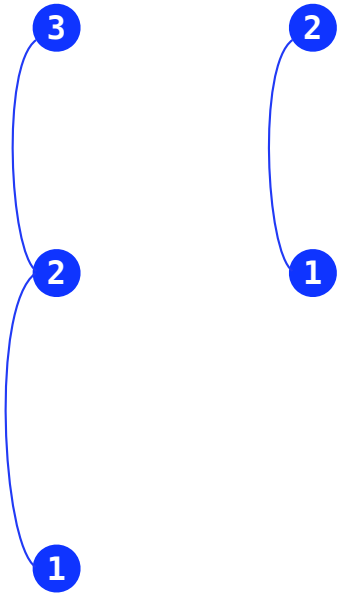
- Every user-defined **function** has a ***parent frame*** (often global)
- The parent of a **function** is the frame in which it was *defined*
- Every local **frame** has a ***parent frame*** (often global)
- The parent of a **frame** is the parent of the function *called*

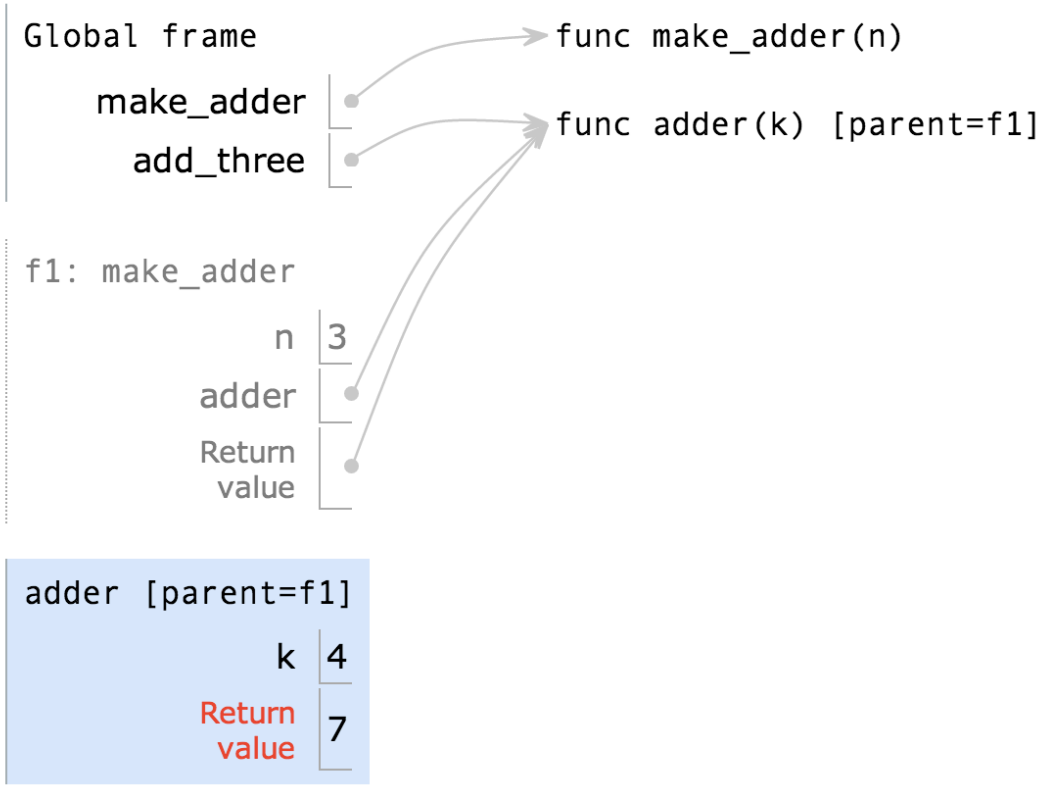
Example:

An Environment is a Sequence of Frames

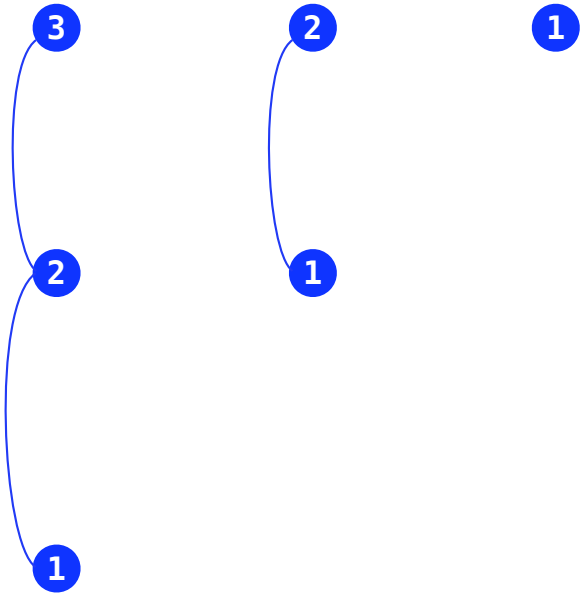


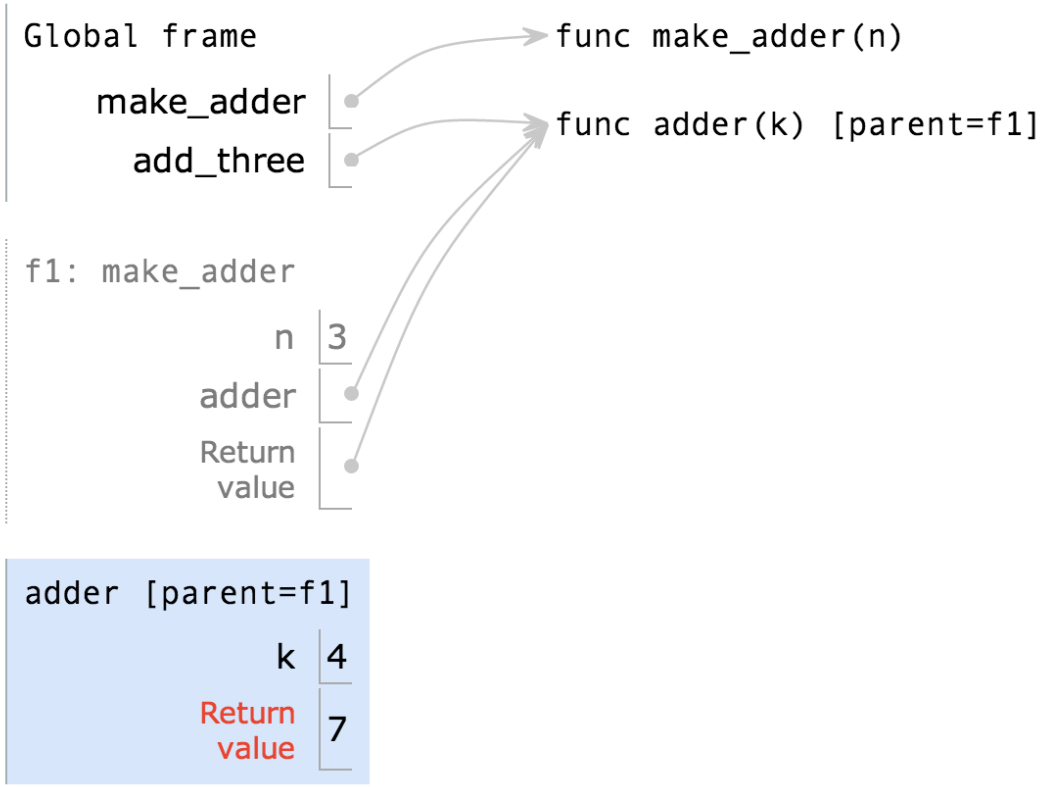
An Environment is a Sequence of Frames



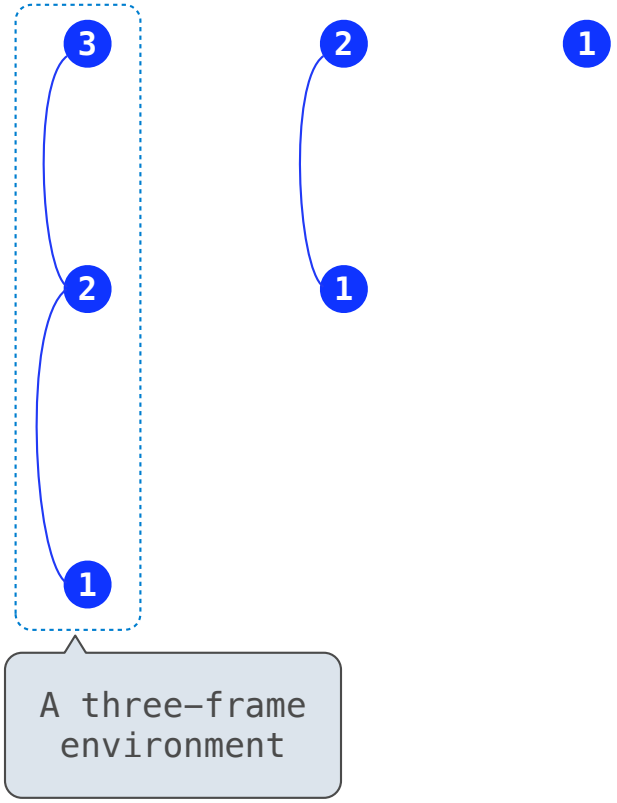


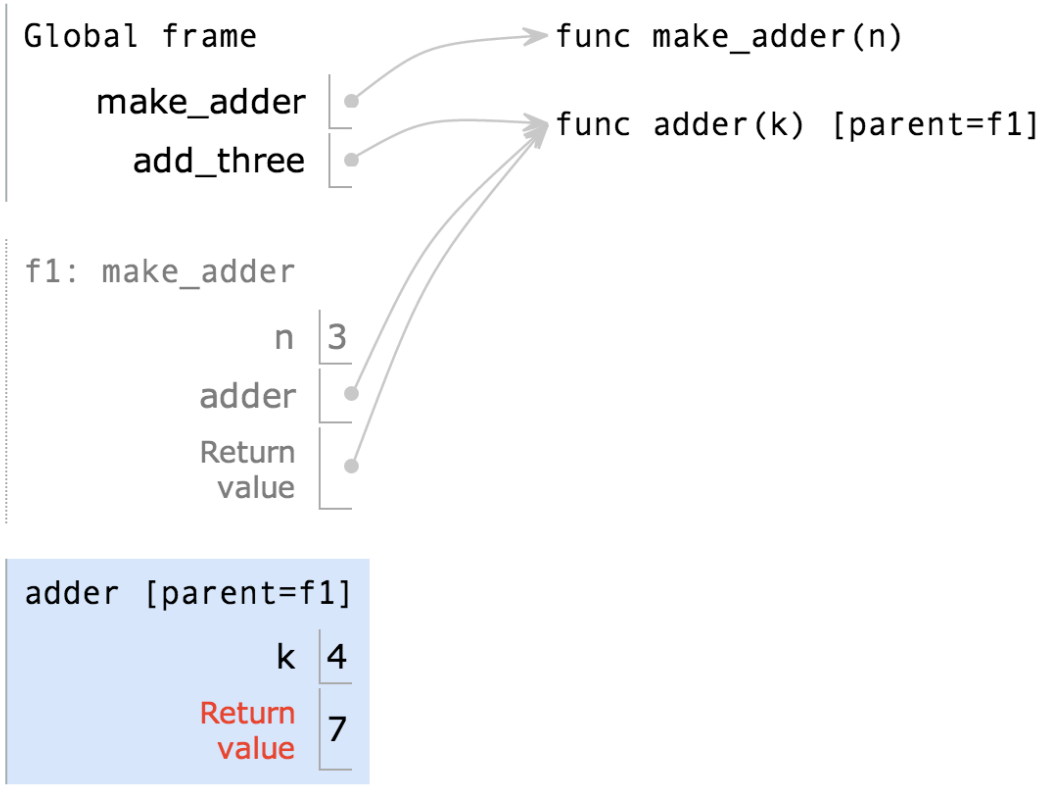
An Environment is a Sequence of Frames



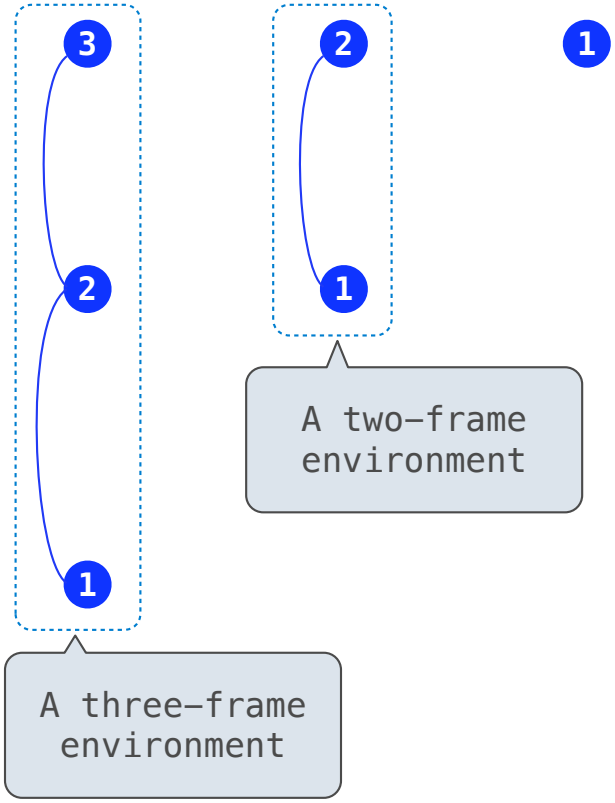


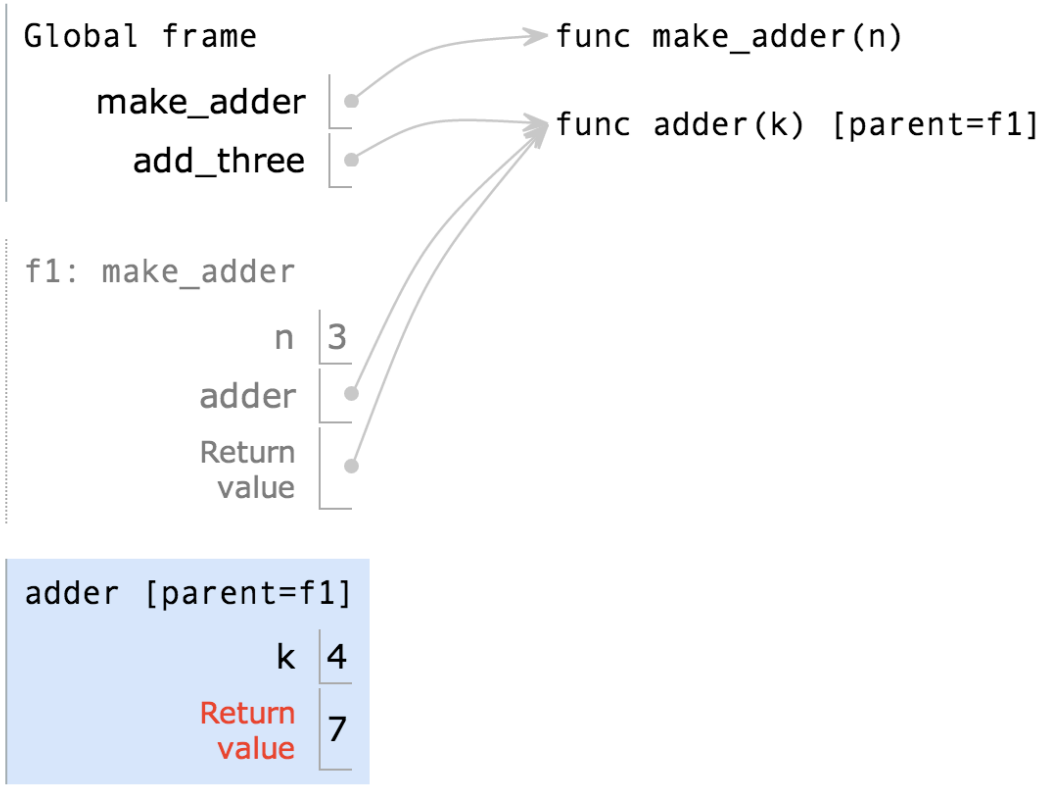
An Environment is a Sequence of Frames



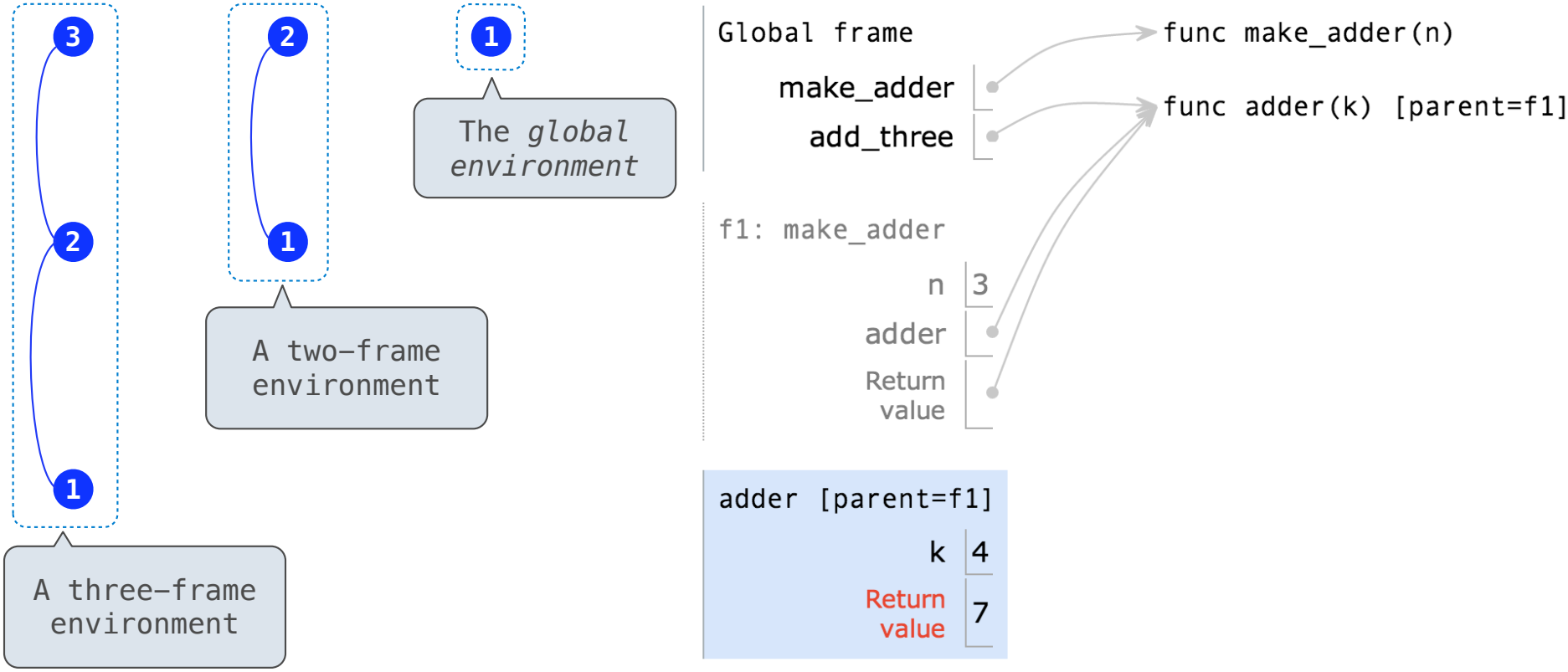


An Environment is a Sequence of Frames

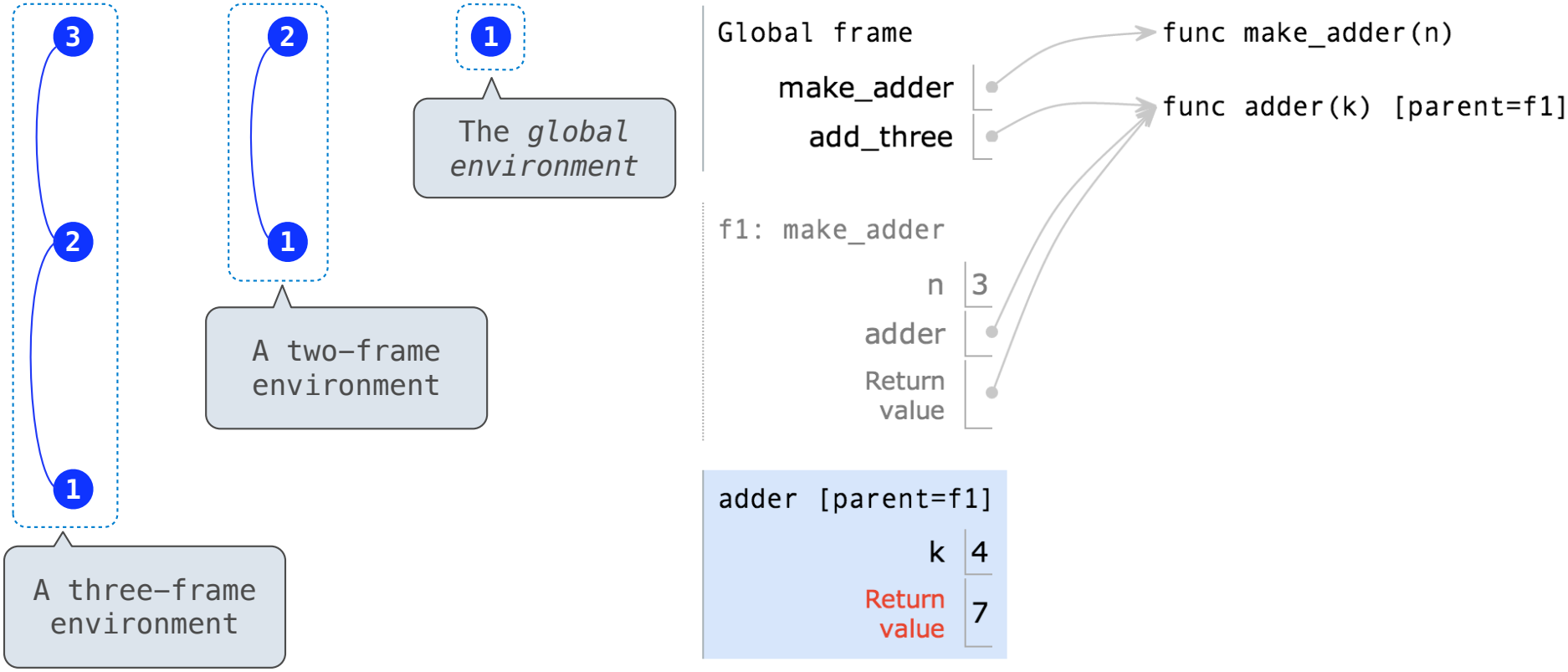




An Environment is a Sequence of Frames

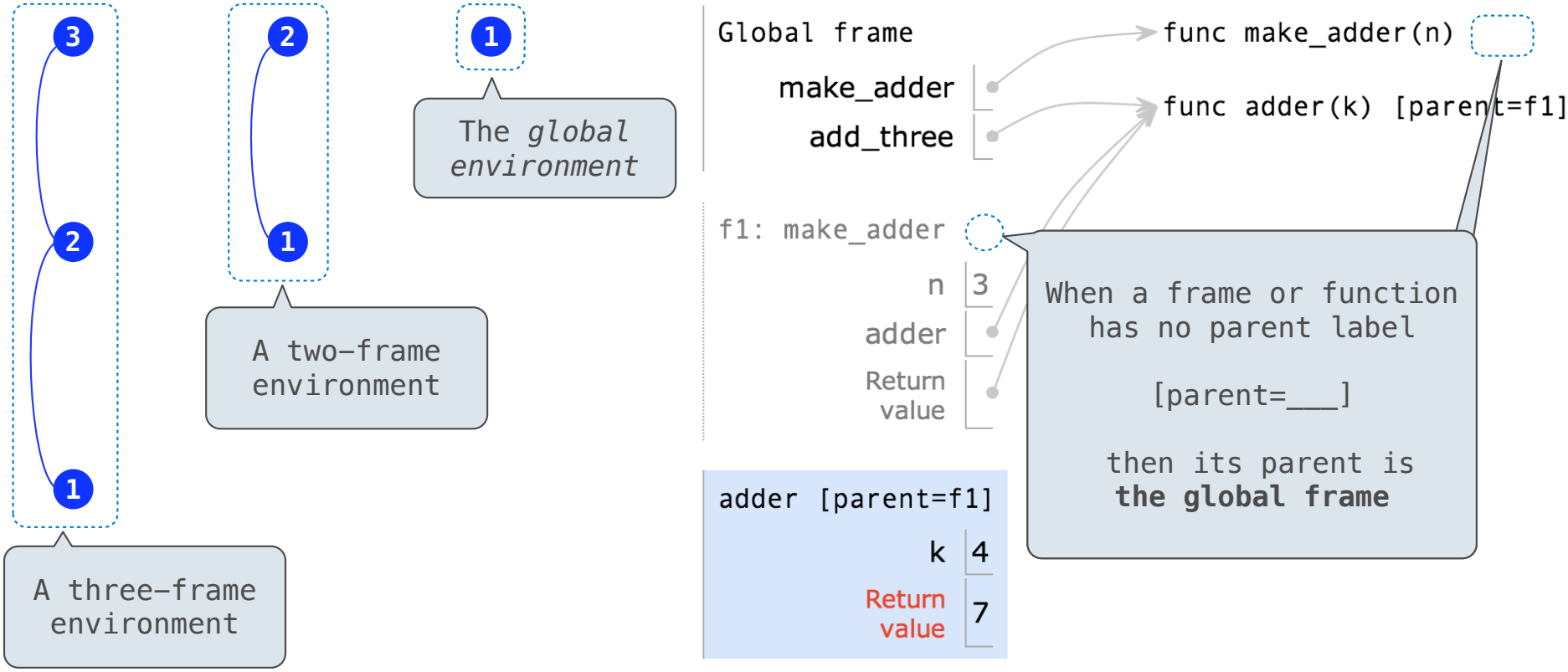


An Environment is a Sequence of Frames



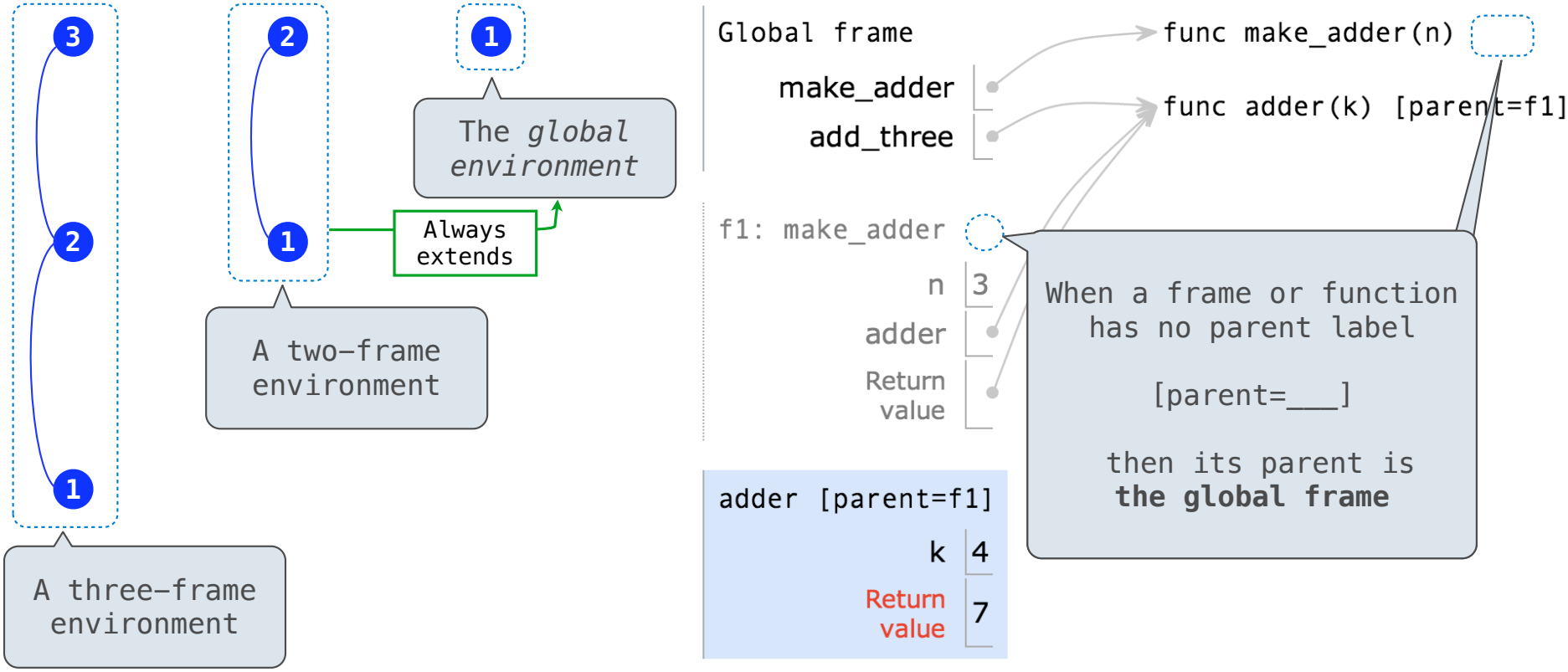
A local frame extends the environment that begins with its parent.

An Environment is a Sequence of Frames



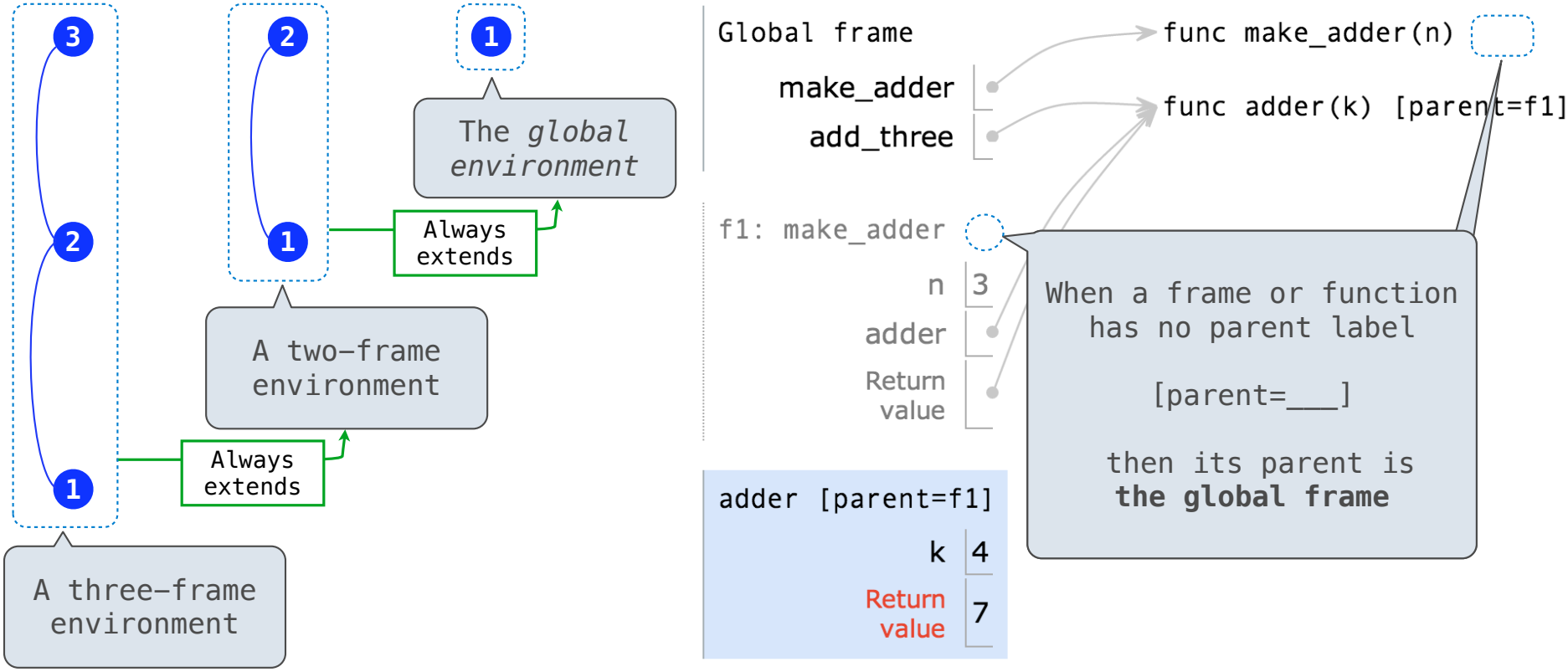
A local frame extends the environment that begins with its parent.

An Environment is a Sequence of Frames



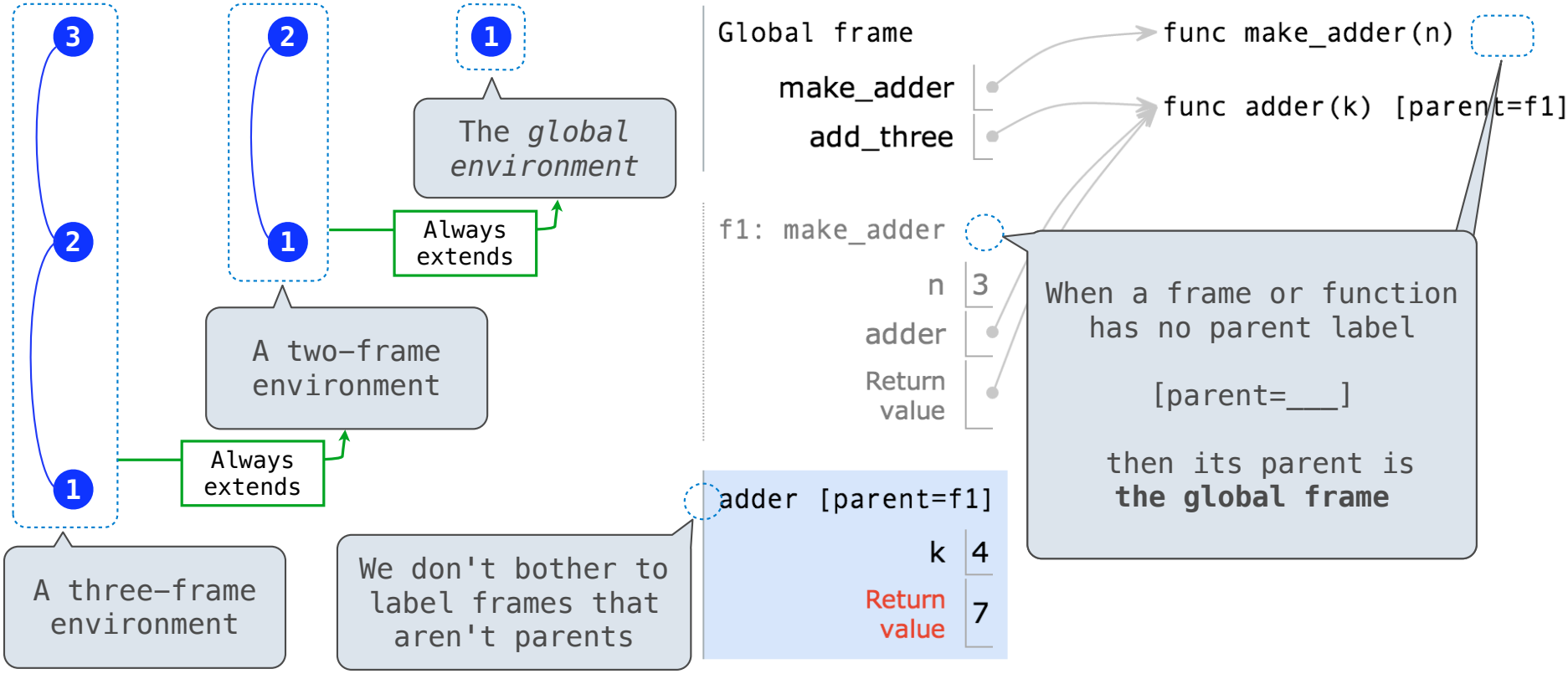
A local frame extends the environment that begins with its parent.

An Environment is a Sequence of Frames



A local frame extends the environment that begins with its parent.

An Environment is a Sequence of Frames



A local frame extends the environment that begins with its parent.

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:

1. Create a **function value:** func *<name>*(*<formal parameters>*)

How to Draw an Environment Diagram

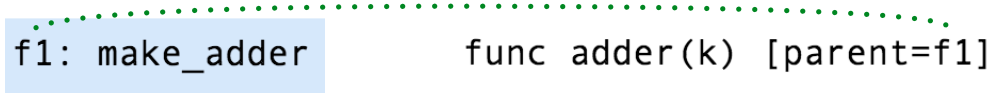
When a function is defined:

1. Create a **function value**: func *<name>*(*<formal parameters>*)
2. If the **parent frame** of that function is not the global frame, add matching labels to the **parent frame** and the **function value** (such as *f1*, *f2*, or *f3*).

How to Draw an Environment Diagram

When a function is defined:

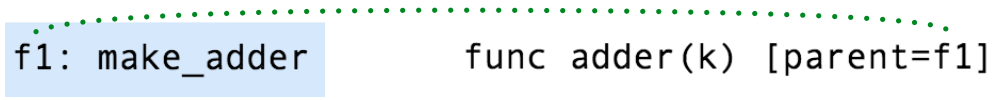
1. Create a **function value**: func <*name*>(<*formal parameters*>)
2. If the **parent frame** of that function is not the global frame, add matching labels to the **parent frame** and the **function value** (such as *f1*, *f2*, or *f3*).



How to Draw an Environment Diagram

When a function is defined:

1. Create a **function value**: func <*name*>(<*formal parameters*>)
2. If the **parent frame** of that function is not the global frame, add matching labels to the **parent frame** and the **function value** (such as *f1*, *f2*, or *f3*).

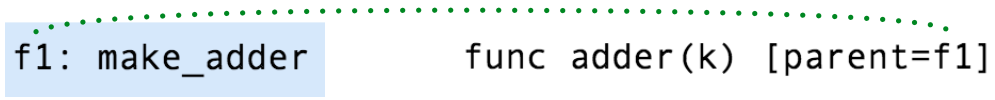


3. Bind <*name*> to the **function value** in the first frame of the current environment.

How to Draw an Environment Diagram

When a function is defined:

1. Create a **function value**: func <*name*>(<*formal parameters*>)
2. If the **parent frame** of that function is not the global frame, add matching labels to the **parent frame** and the **function value** (such as *f1*, *f2*, or *f3*).



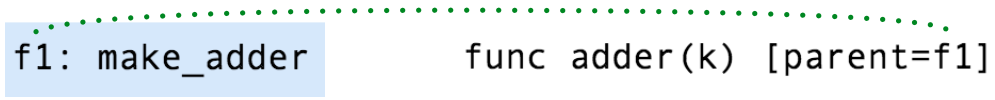
3. Bind <*name*> to the **function value** in the first frame of the current environment.

When a function is called:

How to Draw an Environment Diagram

When a function is defined:

1. Create a **function value**: func <*name*>(<*formal parameters*>)
2. If the **parent frame** of that function is not the global frame, add matching labels to the **parent frame** and the **function value** (such as *f1*, *f2*, or *f3*).



3. Bind <*name*> to the **function value** in the first frame of the current environment.

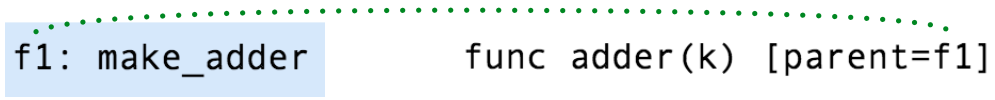
When a function is called:

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:

1. Create a **function value**: func <*name*>(<*formal parameters*>)
2. If the **parent frame** of that function is not the global frame, add matching labels to the **parent frame** and the **function value** (such as *f1*, *f2*, or *f3*).



3. Bind <*name*> to the **function value** in the first frame of the current environment.

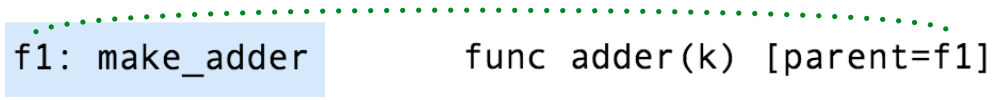
When a function is called:

1. Add a **local frame**, titled with the <*name*> of the function being called.
2. If the function has a parent label, copy it to the **local frame**: [parent=<*label*>]

How to Draw an Environment Diagram

When a function is defined:

1. Create a **function value**: func <*name*>(<*formal parameters*>)
2. If the **parent frame** of that function is not the global frame, add matching labels to the **parent frame** and the **function value** (such as *f1*, *f2*, or *f3*).



3. Bind <*name*> to the **function value** in the first frame of the current environment.

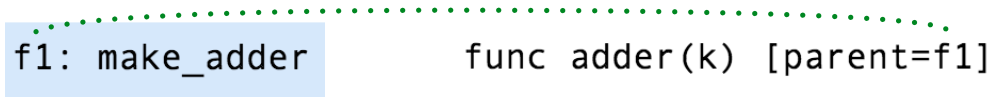
When a function is called:

1. Add a **local frame**, titled with the <*name*> of the function being called.
2. If the function has a parent label, copy it 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:

1. Create a **function value**: func <*name*>(<*formal parameters*>)
2. If the **parent frame** of that function is not the global frame, add matching labels to the **parent frame** and the **function value** (such as *f1*, *f2*, or *f3*).



3. Bind <*name*> to the **function value** in the first frame of the current environment.

When a function is called:

1. Add a **local frame**, titled with the <*name*> of the function being called.
2. If the function has a parent label, copy it 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)

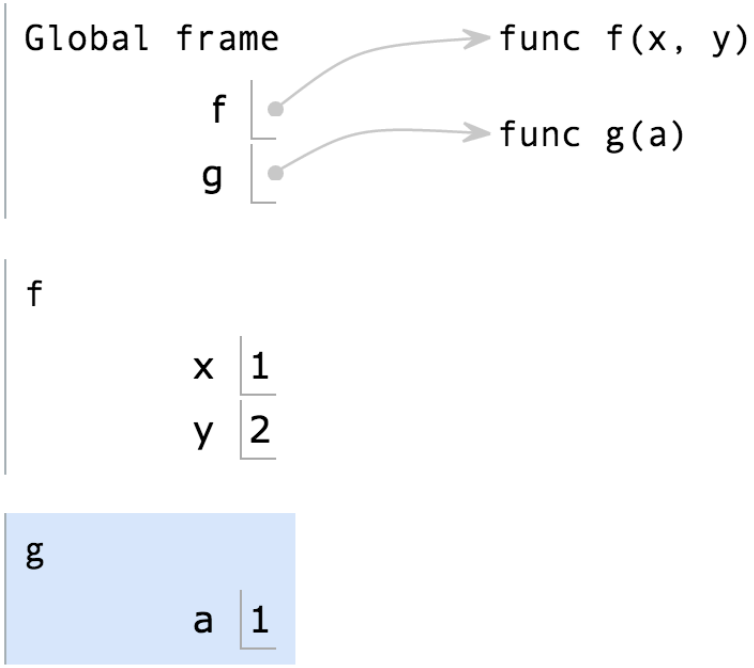
Local Names are *not* Visible to Other (Non-Nested) Functions

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

Example: http://goo.gl/b6WvUc

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

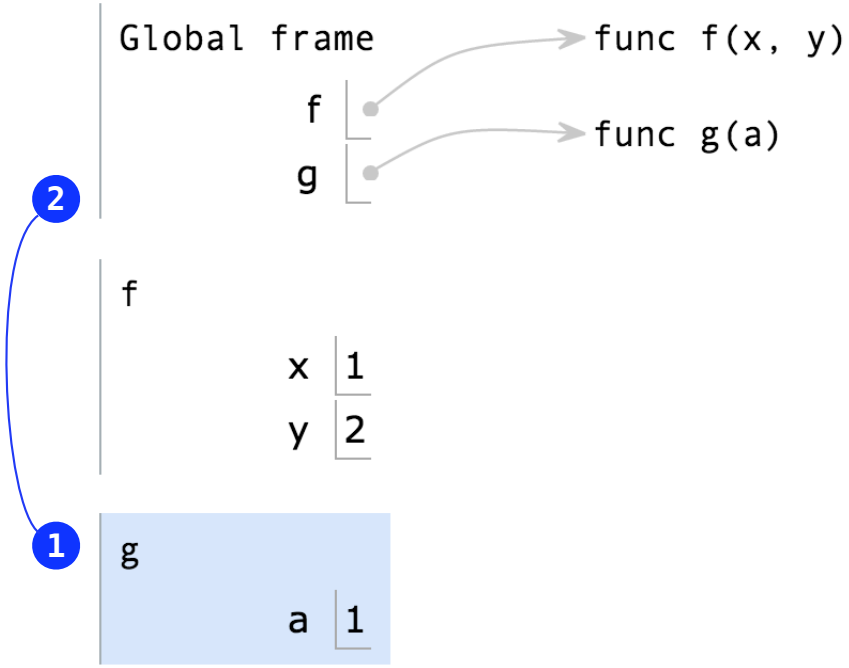
```
1  def f(x, y):
2      return g(x)
3
4  def g(a):
5      return a + y
6
7  result = f(1, 2)
```



Example: http://goo.gl/b6WvUc

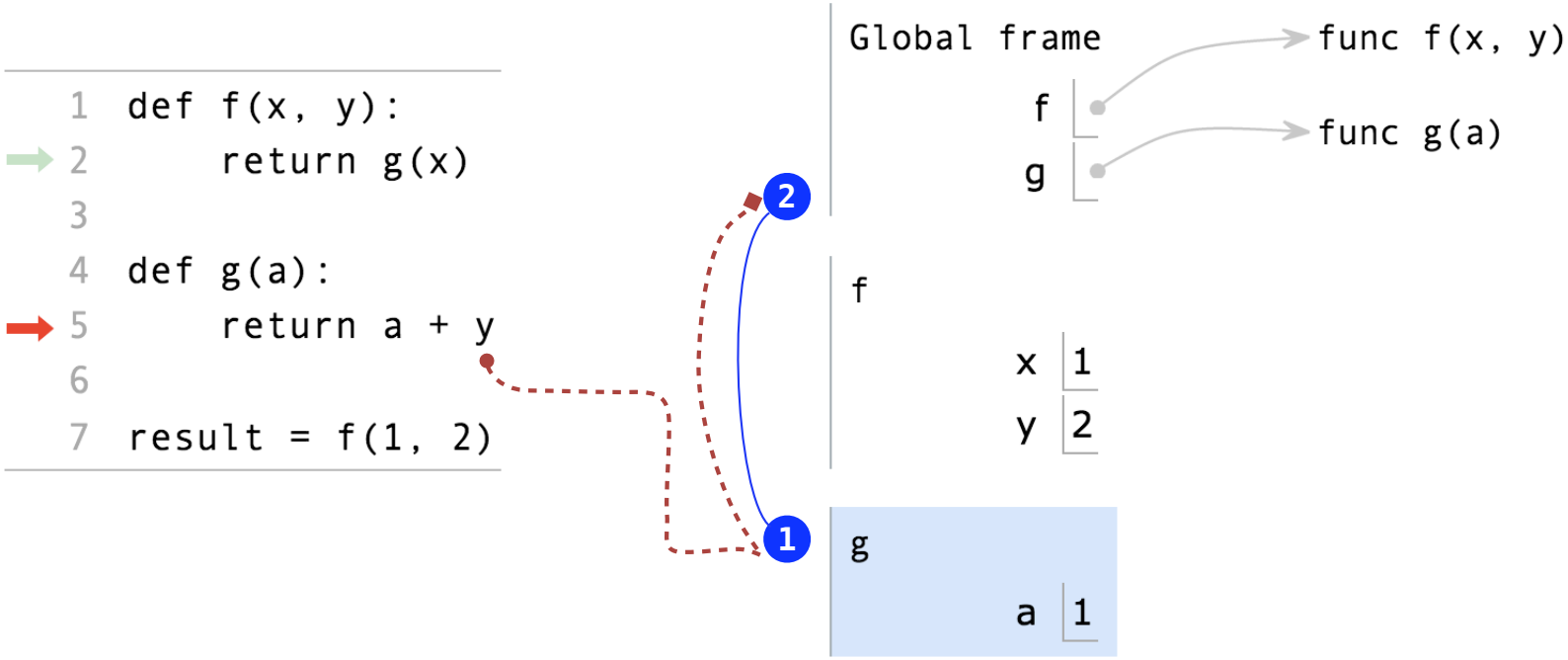
Local Names are *not* Visible to Other (Non-Nested) Functions

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



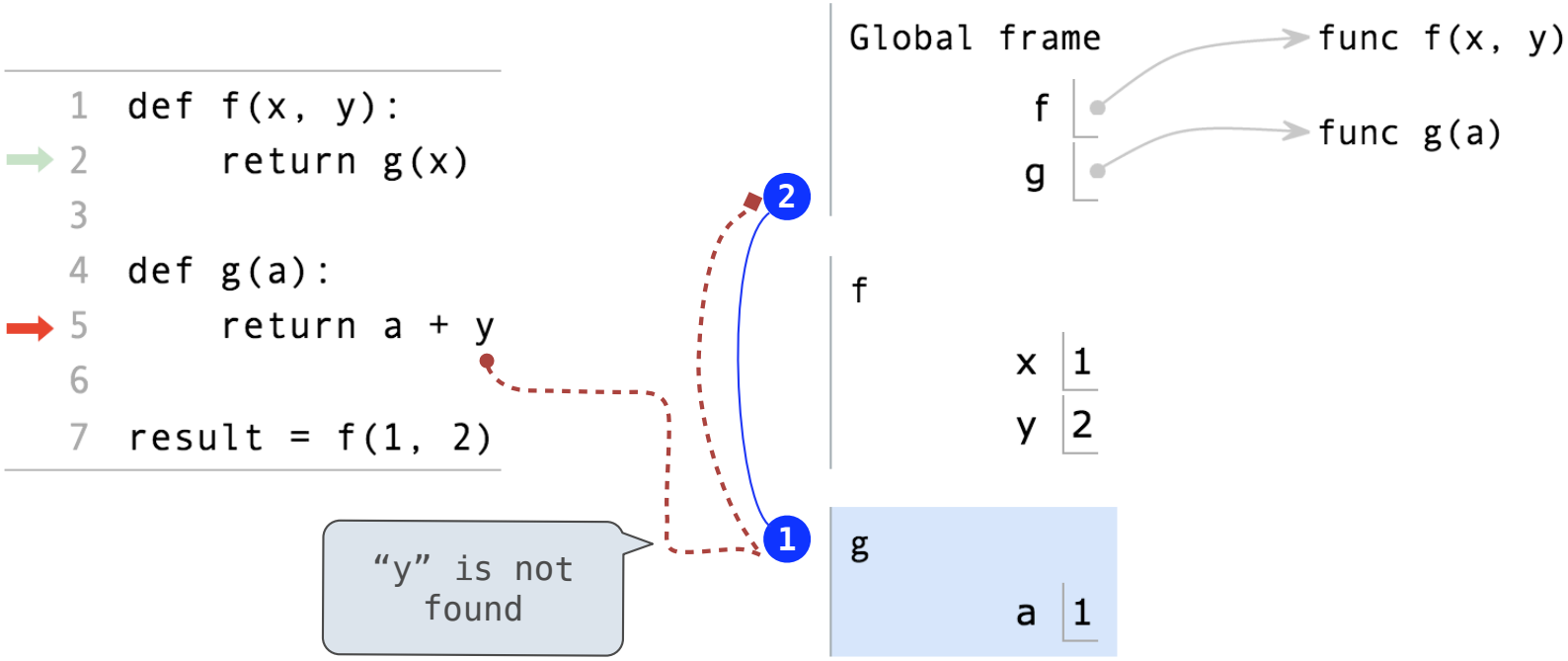
Example: http://goo.gl/b6WvUc

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



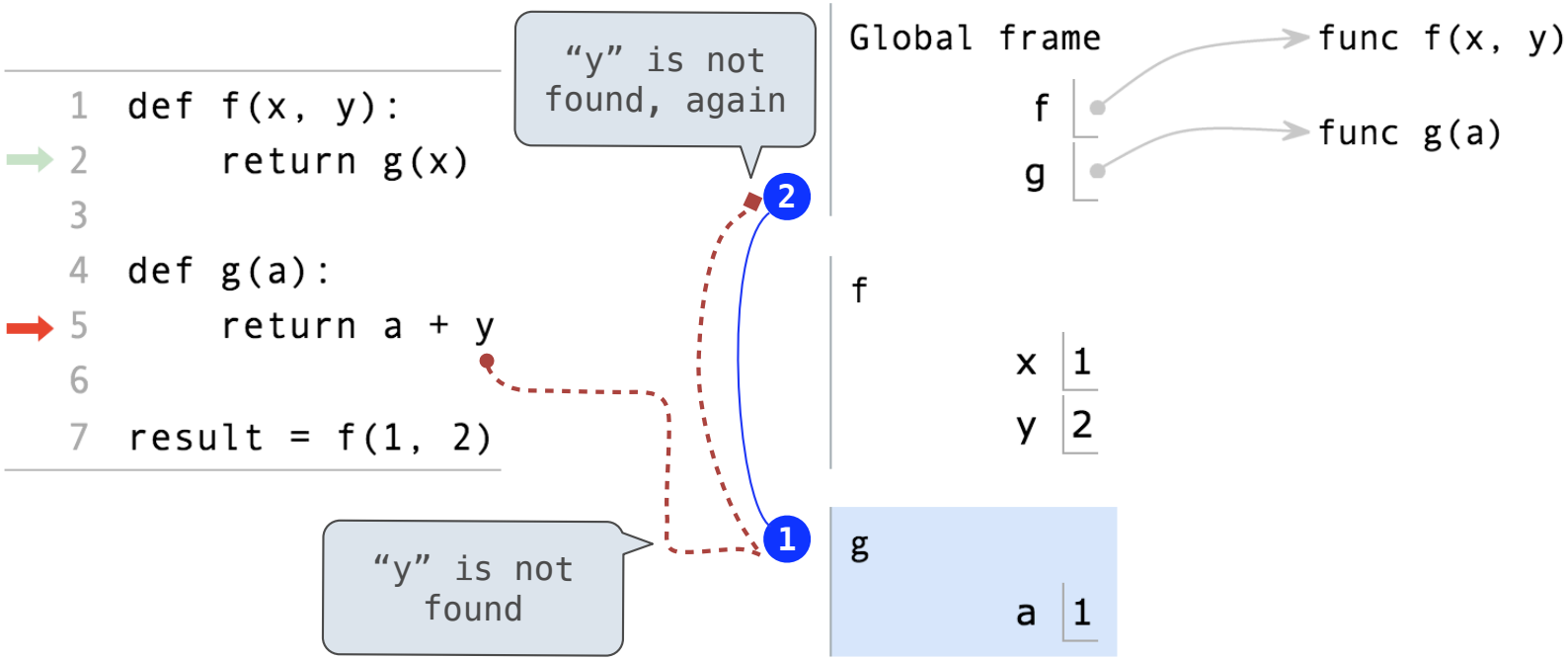
Example: http://goo.gl/b6WvUc

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



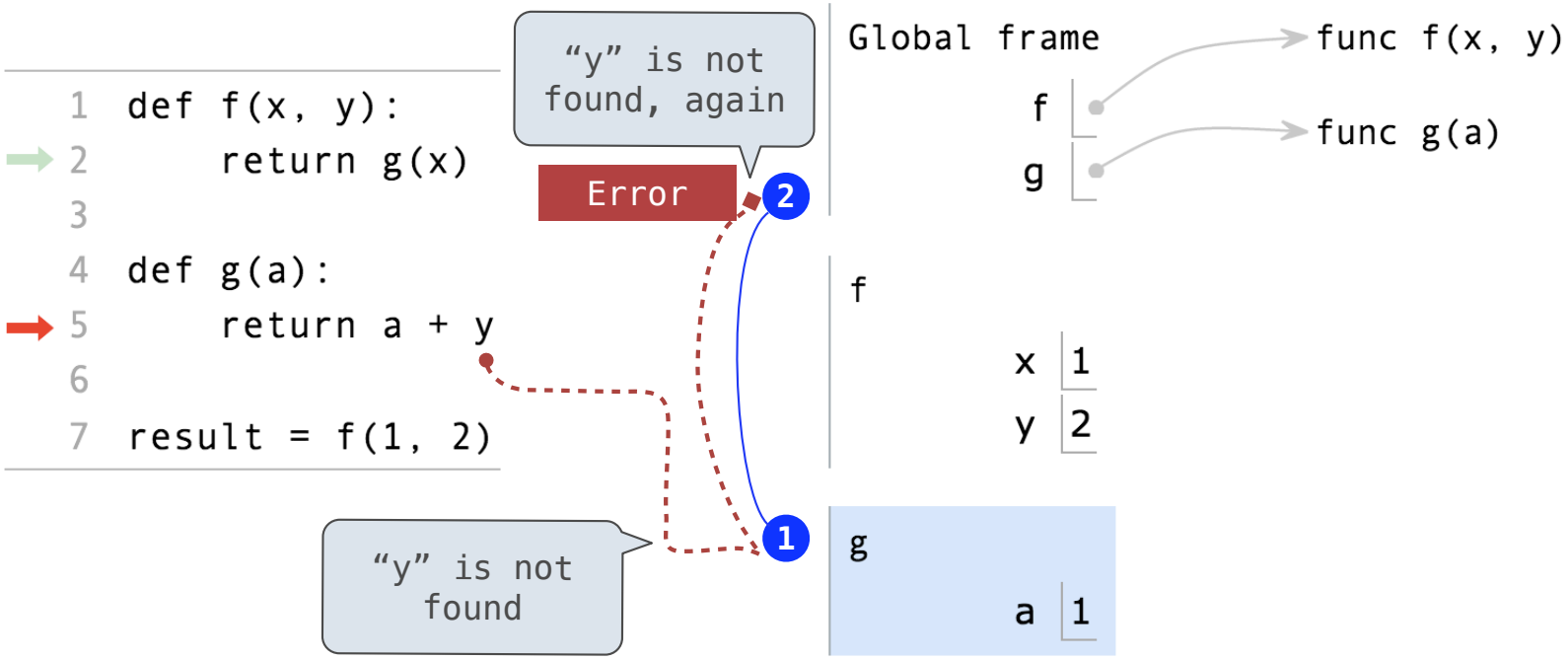
Example: http://goo.gl/b6WvUc

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



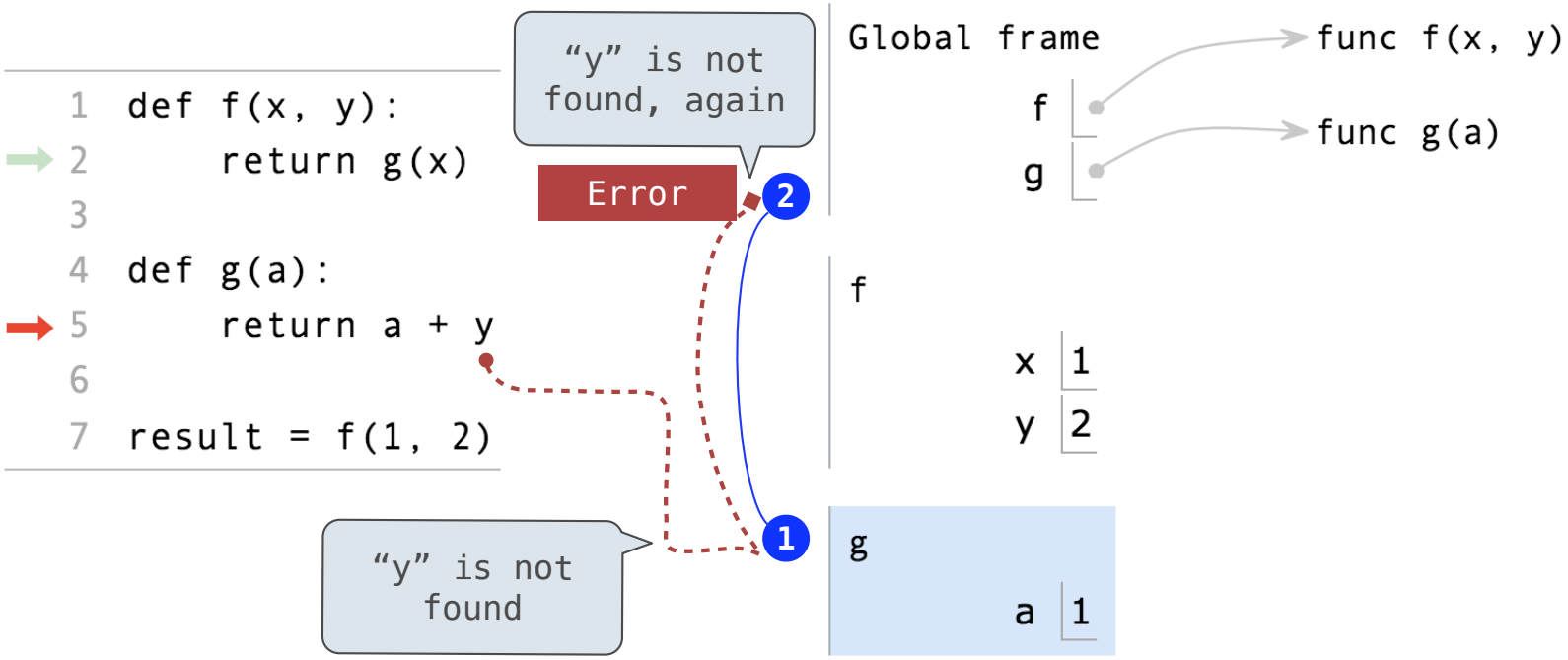
Example: http://goo.gl/b6WvUc

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



Example: http://goo.gl/b6WvUc

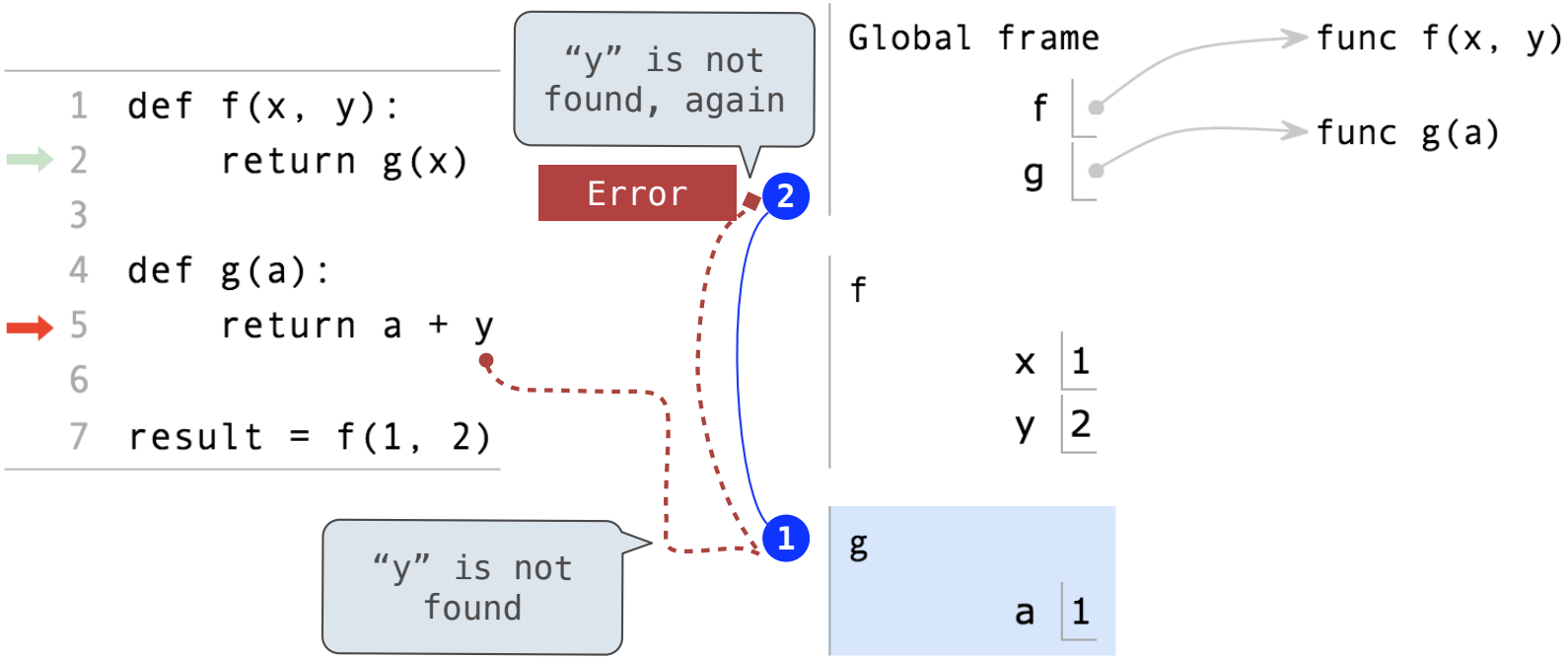
Local Names are *not* Visible to Other (Non-Nested) Functions



- An environment is a sequence of frames.

Example: http://goo.gl/b6WvUc

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.

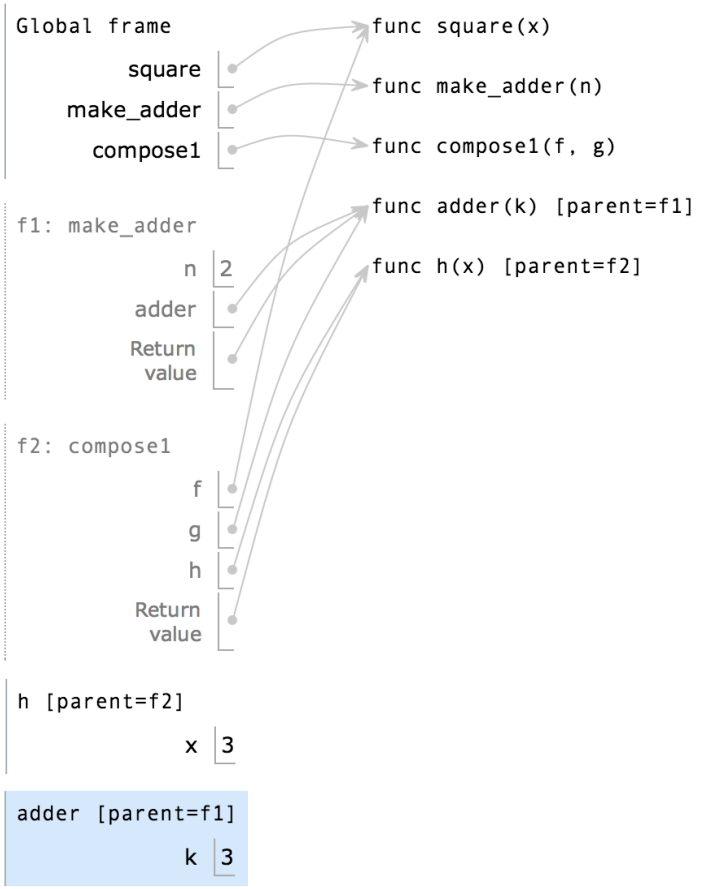
Example: http://goo.gl/b6WvUc

Function Composition

(Demo)

The Environment Diagram for Function Composition

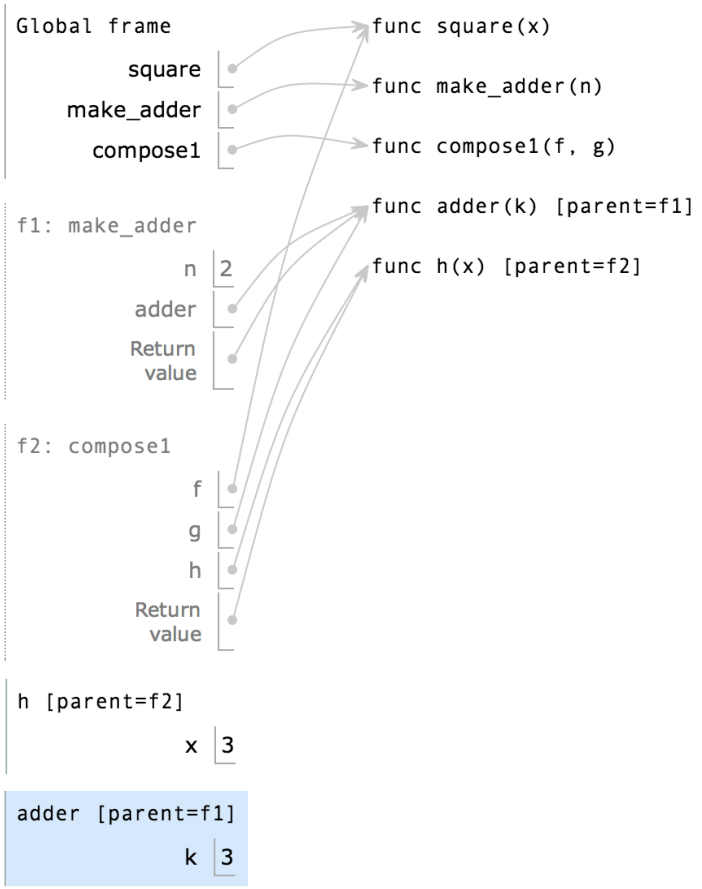
```
1  def square(x):
2      return x * x
3
4  def make_adder(n):
5      def adder(k):
6          return k + n
7      return adder
8
9  def compose1(f, g):
10     def h(x):
11         return f(g(x))
12     return h
13
14 compose1(square, make_adder(2))(3)
```



Example:

The Environment Diagram for Function Composition

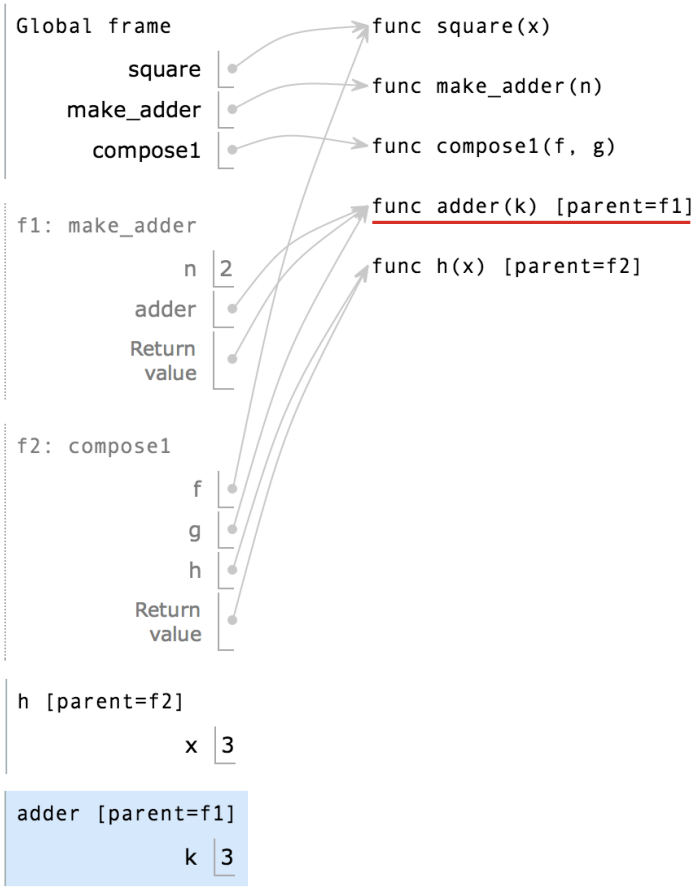
```
1  def square(x):
2      return x * x
3
4  def make_adder(n):
5      def adder(k):
6          return k + n
7      return adder
8
9  def compose1(f, g):
10     def h(x):
11         return f(g(x))
12     return h
13
14 compose1(square, make_adder(2))(3)
```



Example:

The Environment Diagram for Function Composition

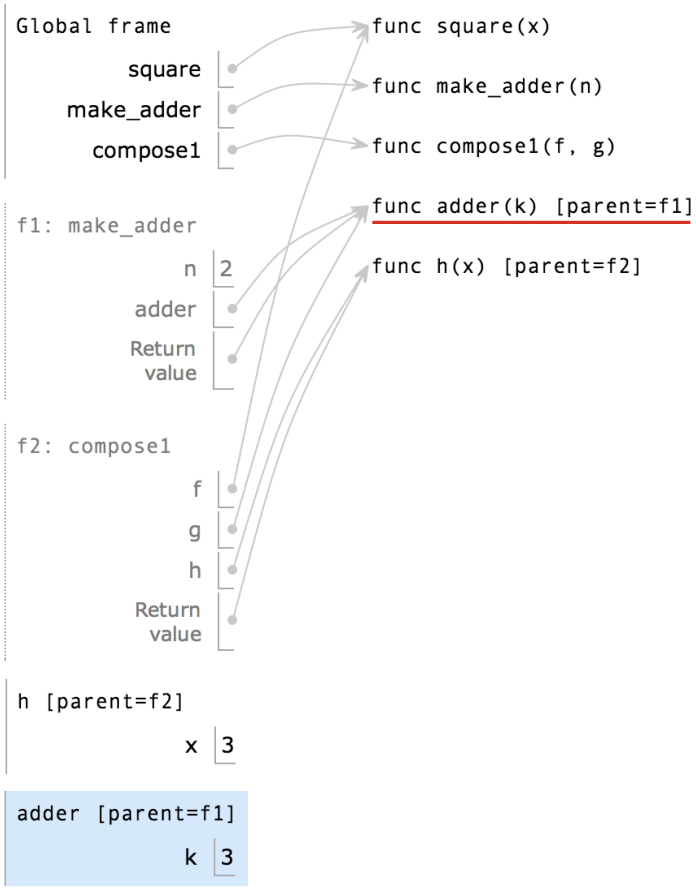
```
1  def square(x):
2      return x * x
3
4  def make_adder(n):
5      def adder(k):
6          return k + n
7      return adder
8
9  def compose1(f, g):
10     def h(x):
11         return f(g(x))
12     return h
13
14 compose1(square, make_adder(2))(3)
```



Example:

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)
```

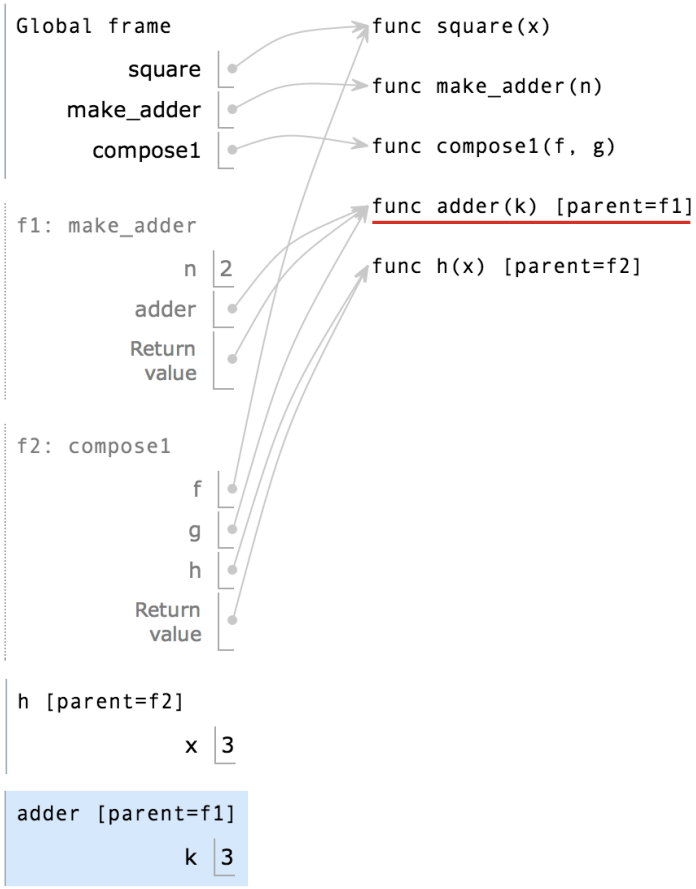


Example:

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

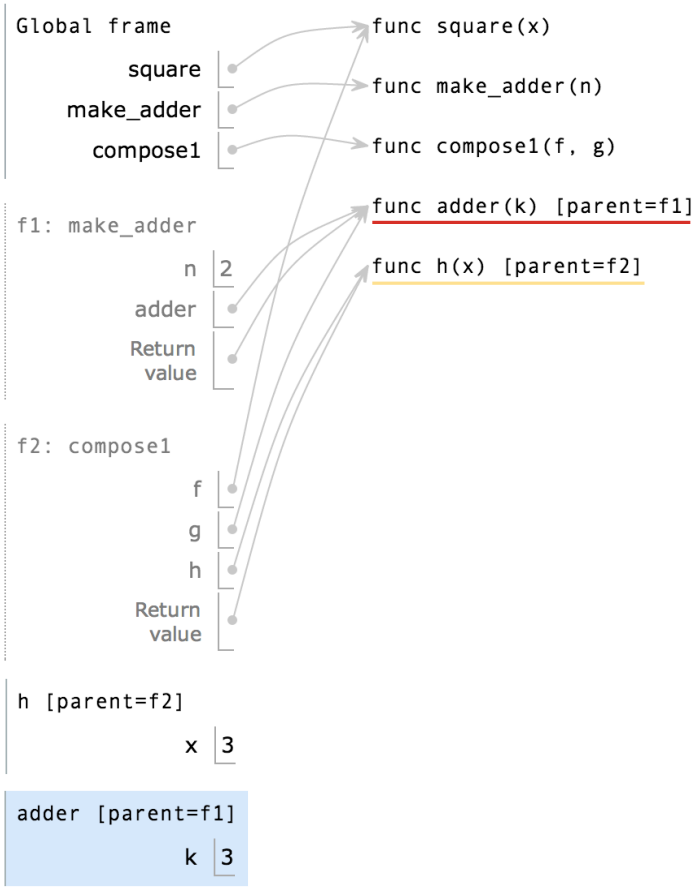


Example:

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

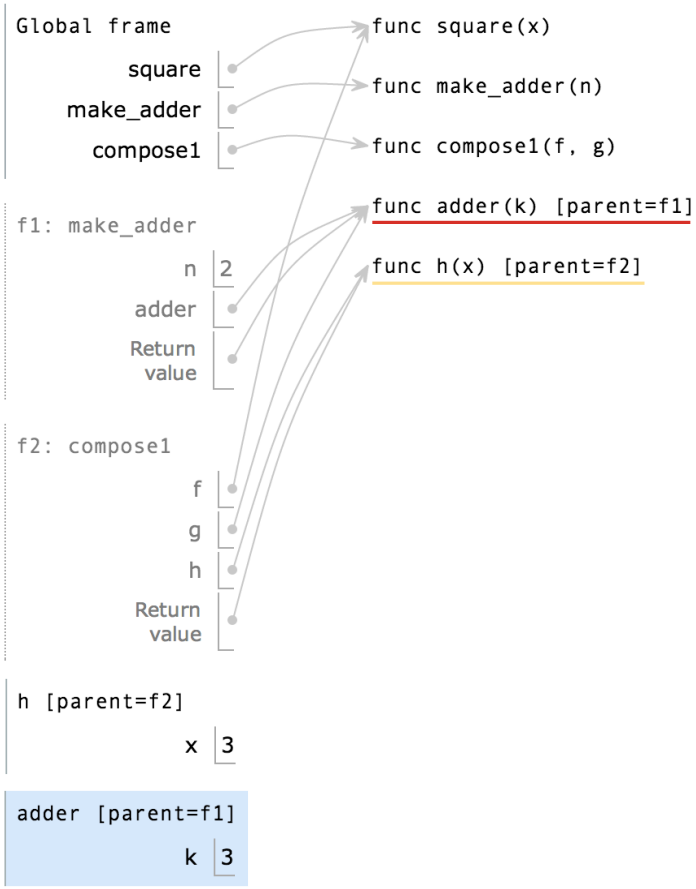


Example:

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

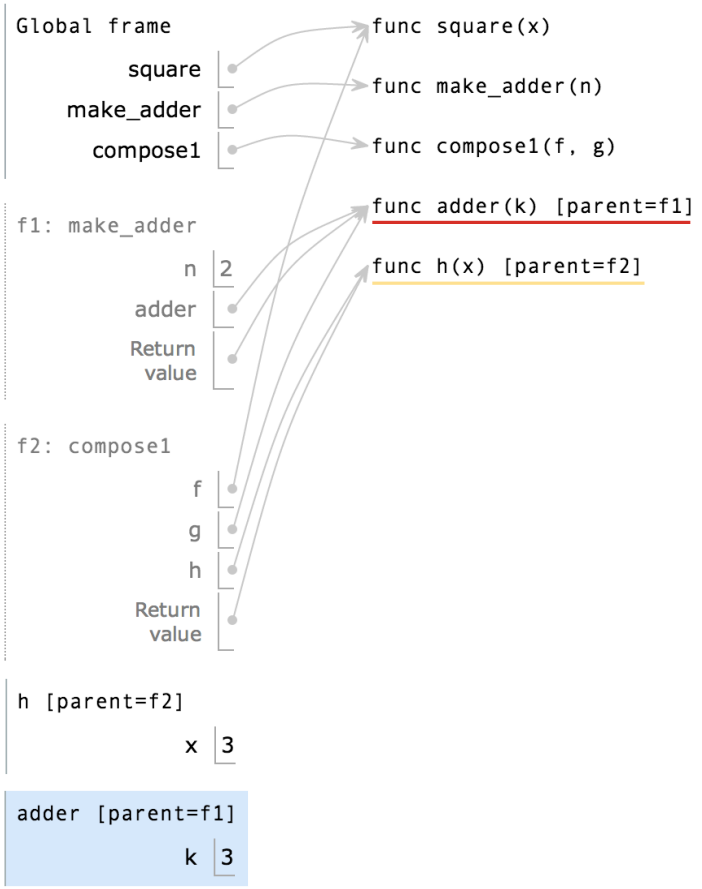


Example:

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

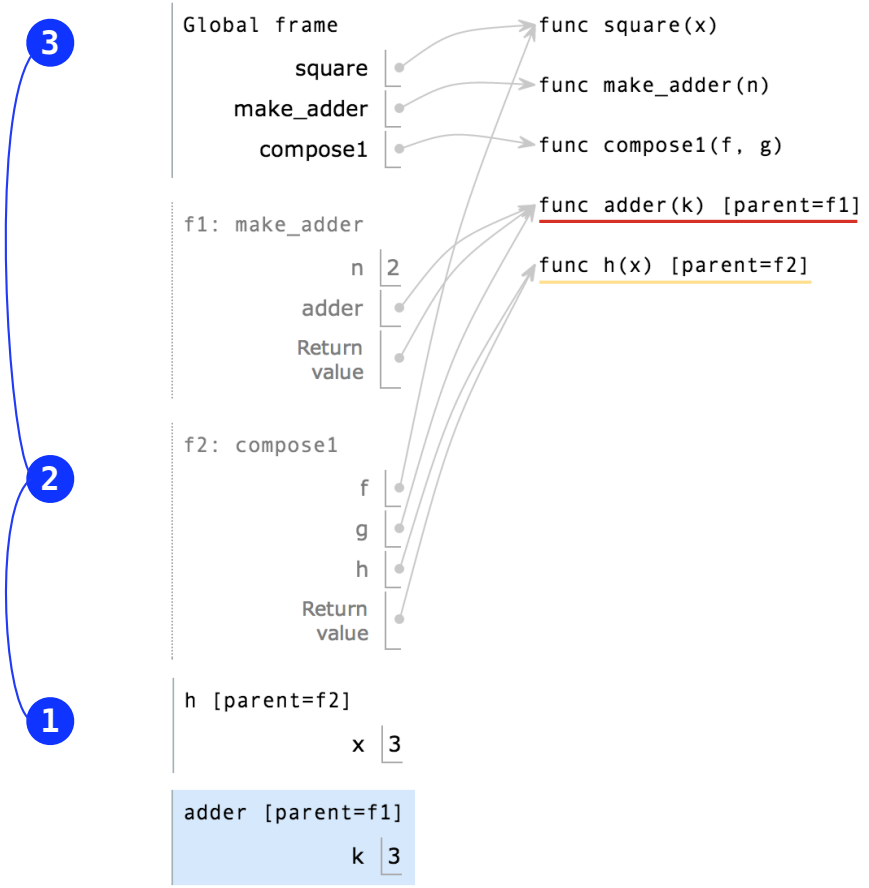


Example:

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

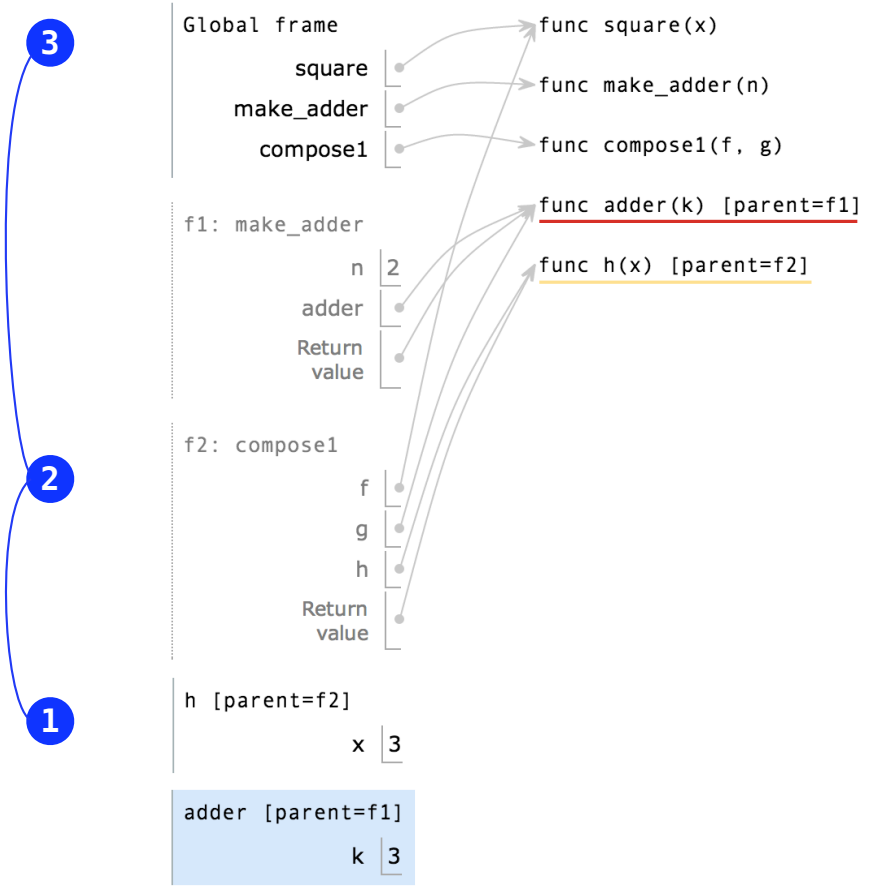


Example:

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

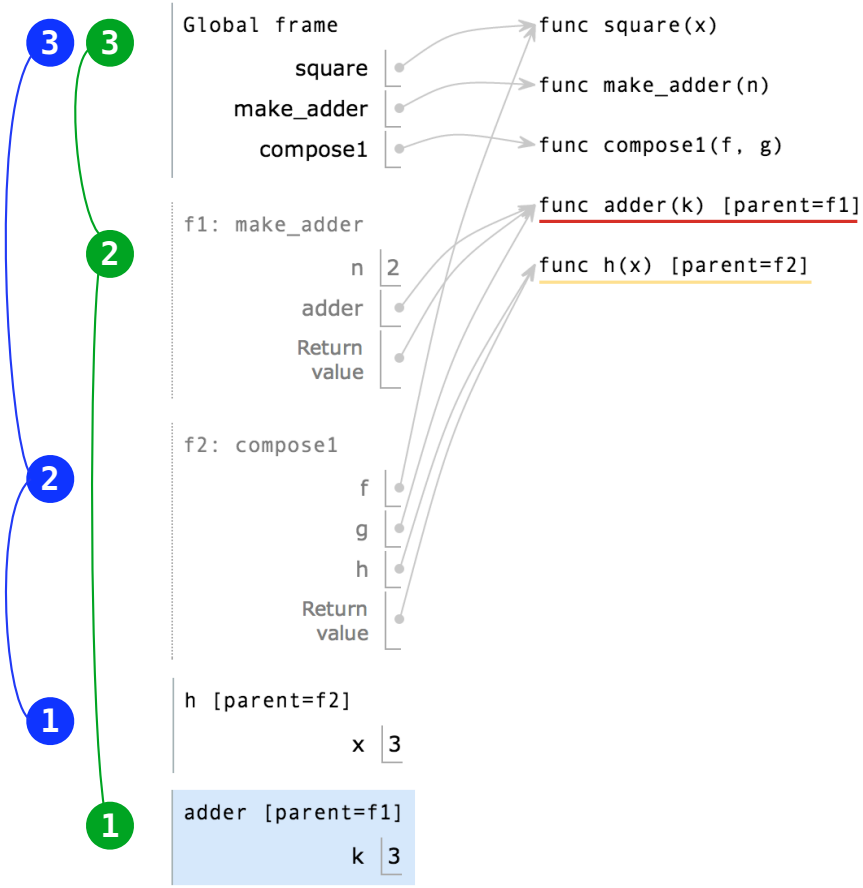


Example:

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



Example:

The Game of Hog

(Demo)