

Higher Order Functions

Thursday, 6/25

Announcements

- Fill out [OK Issues](#) survey
- Tutoring
 - [Signups available today](#)
 - Starts this Saturday
 - Tutoring sessions next week (up to Friday) will review week 1 material
- [Homework 1](#) due Friday, June 26
- [Homework 2](#) due Monday, June 29
- [Project 1: Hog](#) will be released this afternoon
 - Due next Thursday, July 2
 - Work with at most 1 partner
 - Must declare partner on ok!
 - Start early!

Higher Order Functions

1. Review: functions as arguments
2. Nested functions
3. Functions as return values
4. Lambda functions

Review: Functions as Arguments

```
def summation(term, n):  
    i, total = 1, 0  
    while i <= n:  
        total += term(i)  
        i += 1  
    return total
```

1-argument
function

```
>>> def double(x):  
...     return 2*x  
...  
>>> summation(double, 3)  
12
```

Higher Order Functions

1. Review: functions as arguments
2. Nested functions
3. Functions as return values
4. Lambda functions

Nested functions

Functions can be defined inside functions.

This is useful for helper functions.

$$f(a, b, c) = \frac{ab + \frac{b}{a} + a^b}{ac + \frac{c}{a} + a^c}$$

```
def f(a, b, c):  
    numer_ab = a*b + b/a + a**b  
    denom_ac = a*c + c/a + a**c  
    return numer_ab / denom_ac
```

numer_ab &
denom_ac
are very
similar!

Nested functions

Helper functions can simplify code!

$$f(a, b, c) = \frac{ab + \frac{b}{a} + a^b}{ac + \frac{c}{a} + a^c}$$

How?

```
def f(a, b, c):  
    def g(d):  
        return a*d + d/a + a**d  
    return g(b) / g(c)
```

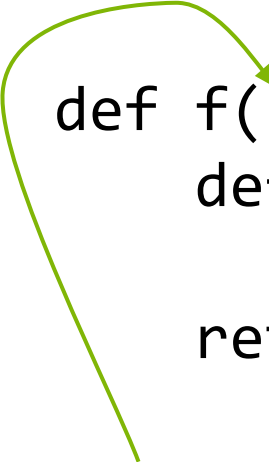
f(2, 3, 4)

g is defined inside the body of f!

Nested functions

Helper functions can simplify code!

$$f(a, b, c) = \frac{ab + \frac{b}{a} + a^b}{ac + \frac{c}{a} + a^c}$$



```
def f(2, b, c):  
    def g(d):  
        return a*d + d/a + a**d  
    return g(b) / g(c)
```

```
f(2, 3, 4)
```


Nested functions

Helper functions can simplify code!

$$f(a, b, c) = \frac{ab + \frac{b}{a} + a^b}{ac + \frac{c}{a} + a^c}$$

```
def f(2, b, c):  
    def g(d):  
        return 2*d + d/2 + 2**d  
    return g(b) / g(c)  
  
f(2, 3, 4)
```

Nested functions

Helper functions can simplify code!

$$f(a, b, c) = \frac{ab + \frac{b}{a} + a^b}{ac + \frac{c}{a} + a^c}$$

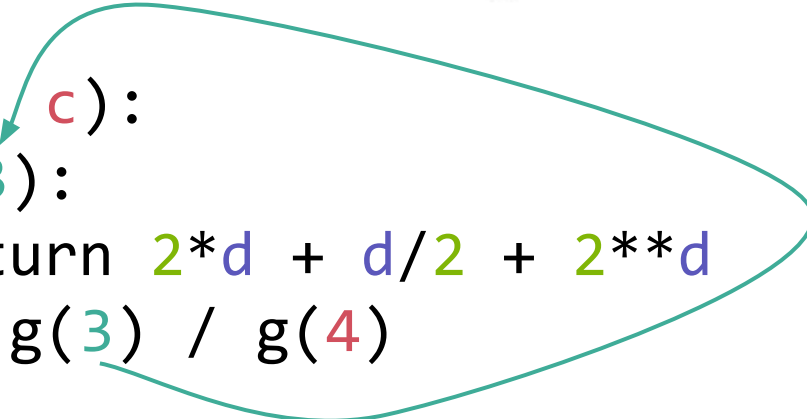
```
def f(2, 3, c):  
    def g(d):  
        return 2*d + d/2 + 2**d  
    return g(3) / g(4)  
  
f(2, 3, 4)
```

Nested functions

Helper functions can simplify code!

$$f(a, b, c) = \frac{ab + \frac{b}{a} + a^b}{ac + \frac{c}{a} + a^c}$$

```
def f(2, 3, c):  
    def g(3):  
        return 2*d + d/2 + 2**d  
    return g(3) / g(4)
```



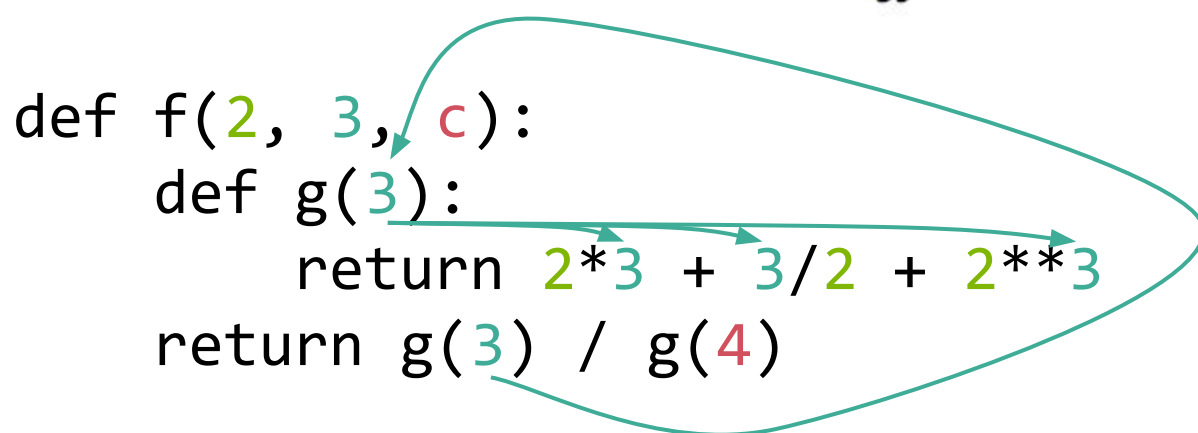
```
f(2, 3, 4)
```

Nested functions

Helper functions can simplify code!

$$f(a, b, c) = \frac{ab + \frac{b}{a} + a^b}{ac + \frac{c}{a} + a^c}$$

```
def f(2, 3, c):  
    def g(3):  
        return 2*3 + 3/2 + 2**3  
    return g(3) / g(4)
```



```
f(2, 3, 4)
```

Nested functions

Helper functions can simplify code!

$$f(a, b, c) = \frac{ab + \frac{b}{a} + a^b}{ac + \frac{c}{a} + a^c}$$

```
def f(2, 3, c):  
    def g(3):  
        return 2*3 + 3/2 + 2**3  
    return 15.5 / g(4)
```

```
f(2, 3, 4)
```

Nested functions

Helper functions can simplify code!

$$f(a, b, c) = \frac{ab + \frac{b}{a} + a^b}{ac + \frac{c}{a} + a^c}$$

```
def f(2, 3, 4):  
    def g(4):  
        return 2*4 + 4/2 + 2**4  
    return 15.5 / g(4)
```

f(2, 3, 4)

The diagram illustrates the execution of the nested function `g` within `f`. Red arrows show the following argument passing:

- The value `4` from the third parameter of `f` is passed to the parameter of `g`.
- The value `2` from the first parameter of `f` is passed to the first operand of the multiplication in `g`'s return statement.
- The value `4` from the parameter of `g` is passed to the numerator of the division in `g`'s return statement.
- The value `2` from the first parameter of `f` is passed to the base of the exponentiation in `g`'s return statement.
- The value `4` from the parameter of `g` is passed to the argument of `g` in the return statement of `f`.

Nested functions

Helper functions can simplify code!

$$f(a, b, c) = \frac{ab + \frac{b}{a} + a^b}{ac + \frac{c}{a} + a^c}$$

```
def f(2, 3, 4):  
    def g(4):  
        return 2*4 + 4/2 + 2**4  
    return 15.5 / 26.0
```

```
f(2, 3, 4)
```

Nested functions

Helper functions can simplify code!

$$f(a, b, c) = \frac{ab + \frac{b}{a} + a^b}{ac + \frac{c}{a} + a^c}$$

```
def f(2, 3, 4):  
    def g(4):  
        return 2*4 + 4/2 + 2**4  
    return 15.5 / 26.0
```

1.6774193548387097

Higher Order Functions

1. Review: functions as arguments
2. Nested functions
3. Functions as return values
4. Lambda functions

Adders

Functions that add **some number**
to **another number**

```
def add_1(x):  
    return x + 1
```

```
def add_2(x):  
    return x + 2
```

```
def add_3(x):  
    return x + 3
```

...

```
def add_128(x):  
    return x + 128
```

(Demo)

DRY

make_adder

function that
returns a function

```
def make_adder(n):  
    """Return func that takes one arg k and returns k + n
```

```
>>> add_three = make_adder(3)
```

```
>>> add_three(4)
```

```
7
```

```
"""
```

nested function

```
def adder(k):
```

```
    return k + n
```

```
return adder
```

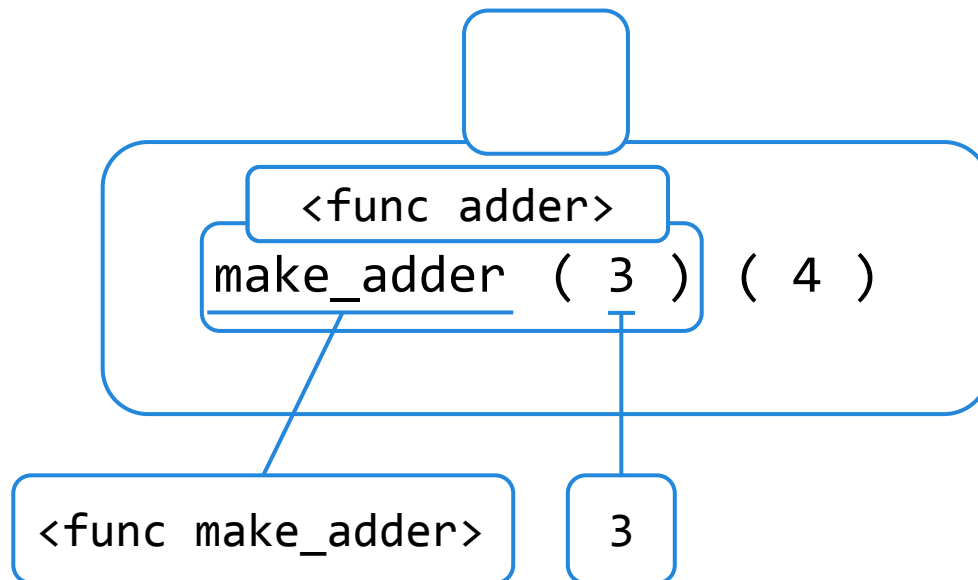
can refer to names in
enclosing functions

name add_three is
bound to a function

make_adder

```
def make_adder(n):  
    def adder(k):  
        return k + n  
    return adder
```

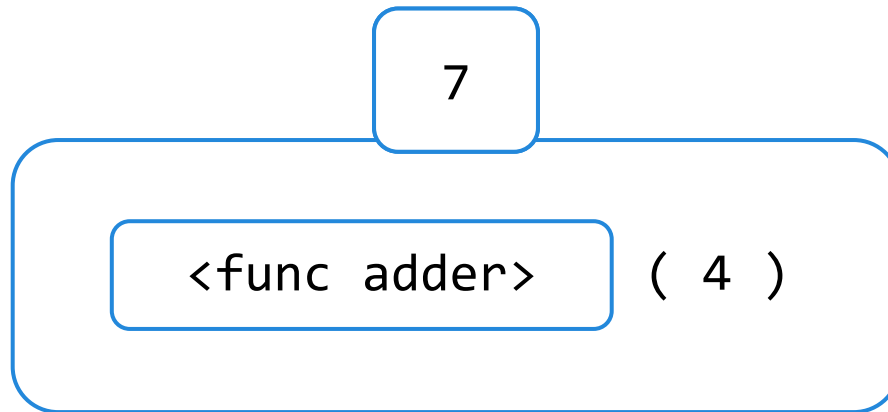
For compound call expressions, read left to right



make_adder

```
def make_adder(n):  
    def adder(k):  
        return k + n  
    return adder
```

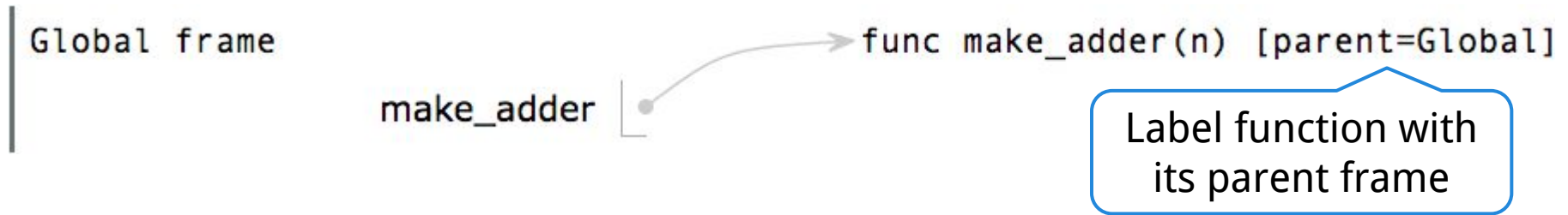
For compound call expressions, read left to right



Environment diagram: make_adder

```
▶ def make_adder(n):  
    def adder(k):  
        return k + n  
    return adder
```

make_adder(3)(4)

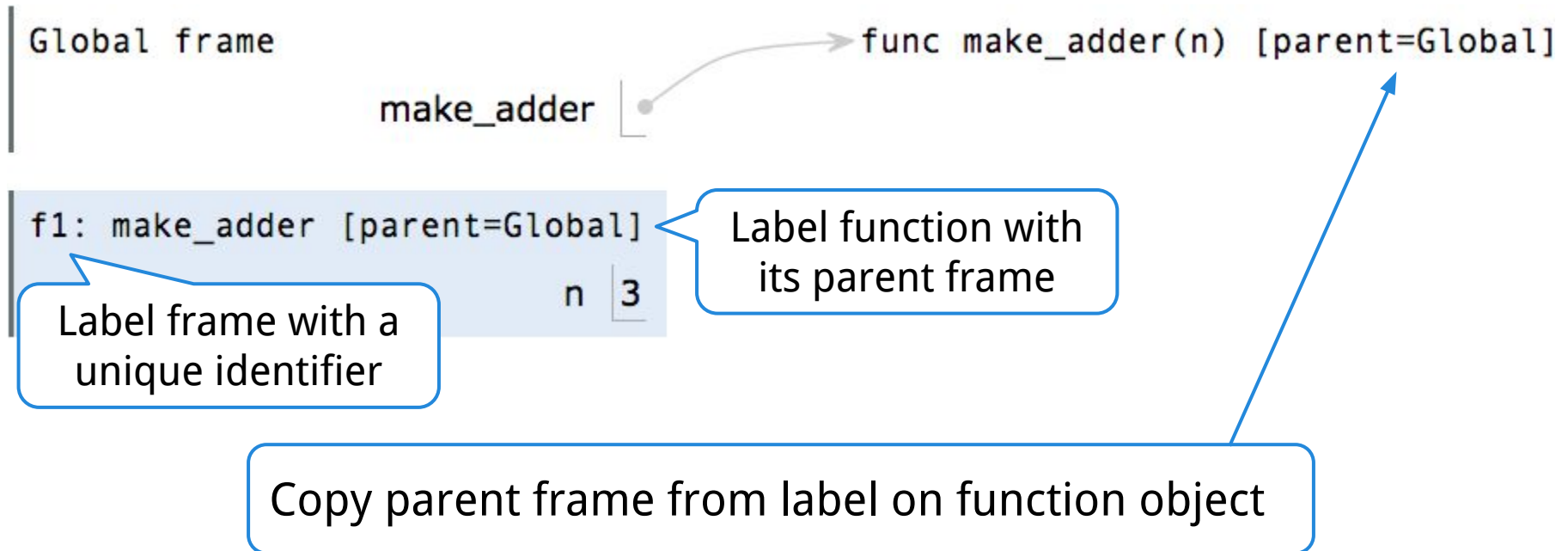


Parent frame: frame in which the function is defined

Environment diagram: make_adder

```
def make_adder(n):  
    def adder(k):  
        return k + n  
    return adder
```

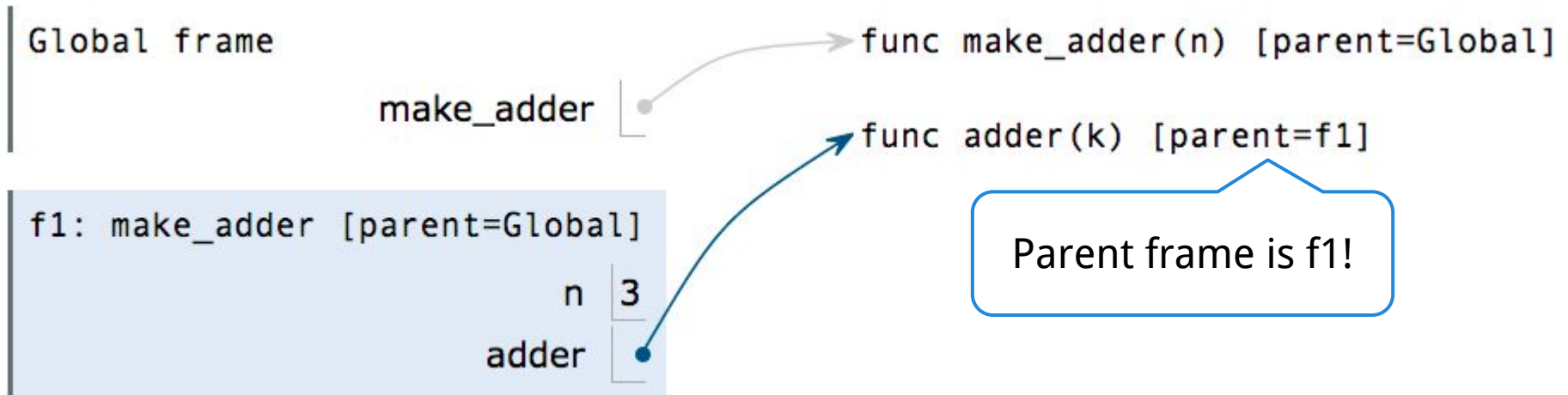
► make_adder(3)(4)



Environment diagram: make_adder

```
def make_adder(n):  
    ▶ def adder(k):  
        return k + n  
    return adder
```

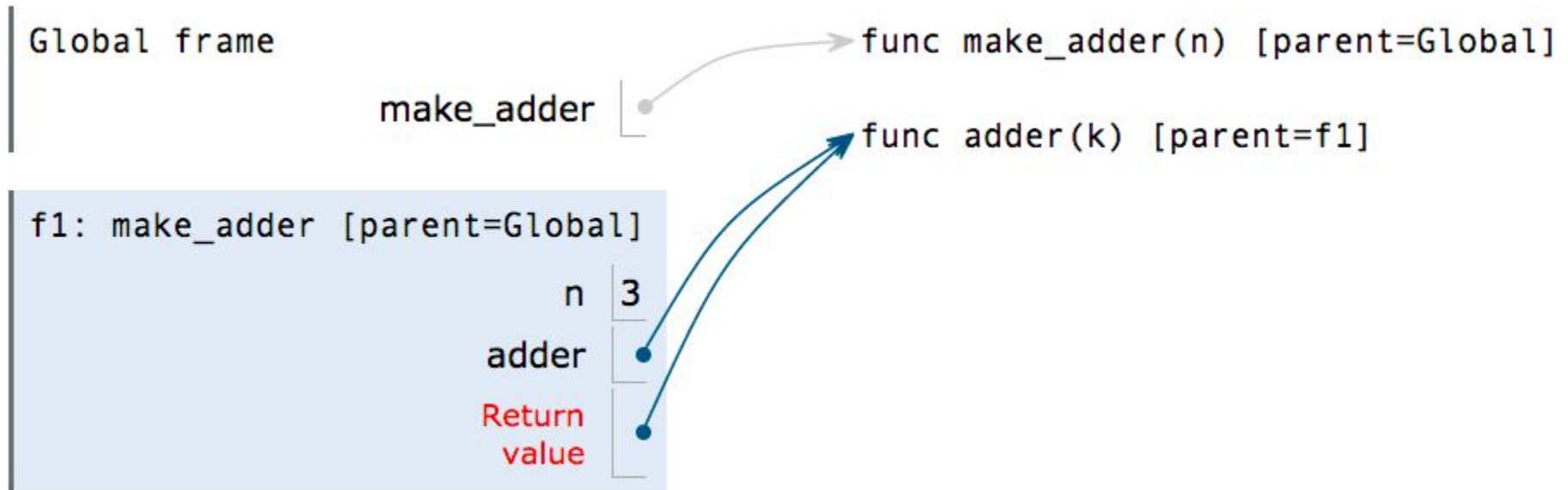
make_adder(3)(4)



Environment diagram: make_adder

```
def make_adder(n):  
    def adder(k):  
        return k + n  
    ► return adder
```

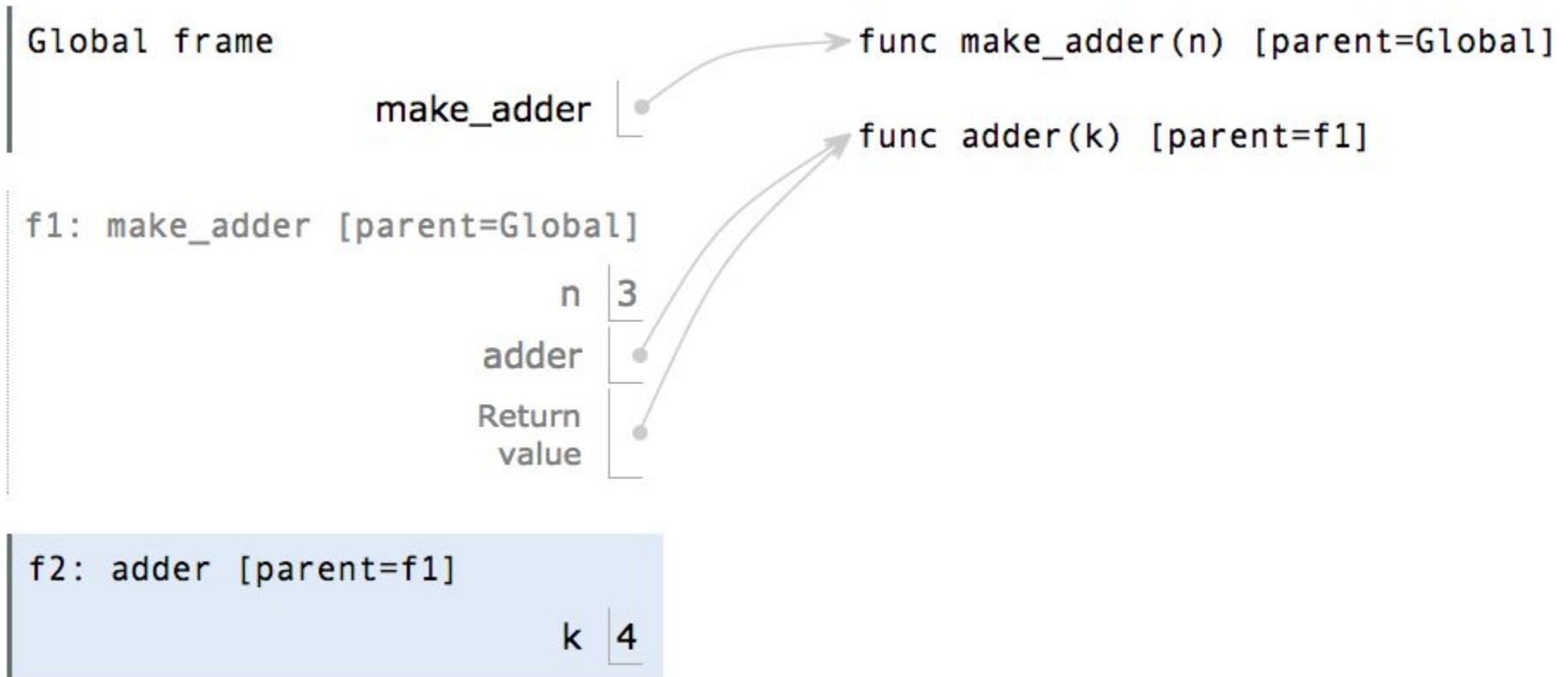
make_adder(3)(4)



Environment diagram: make_adder

```
def make_adder(n):  
    def adder(k):  
        return k + n  
    return adder
```

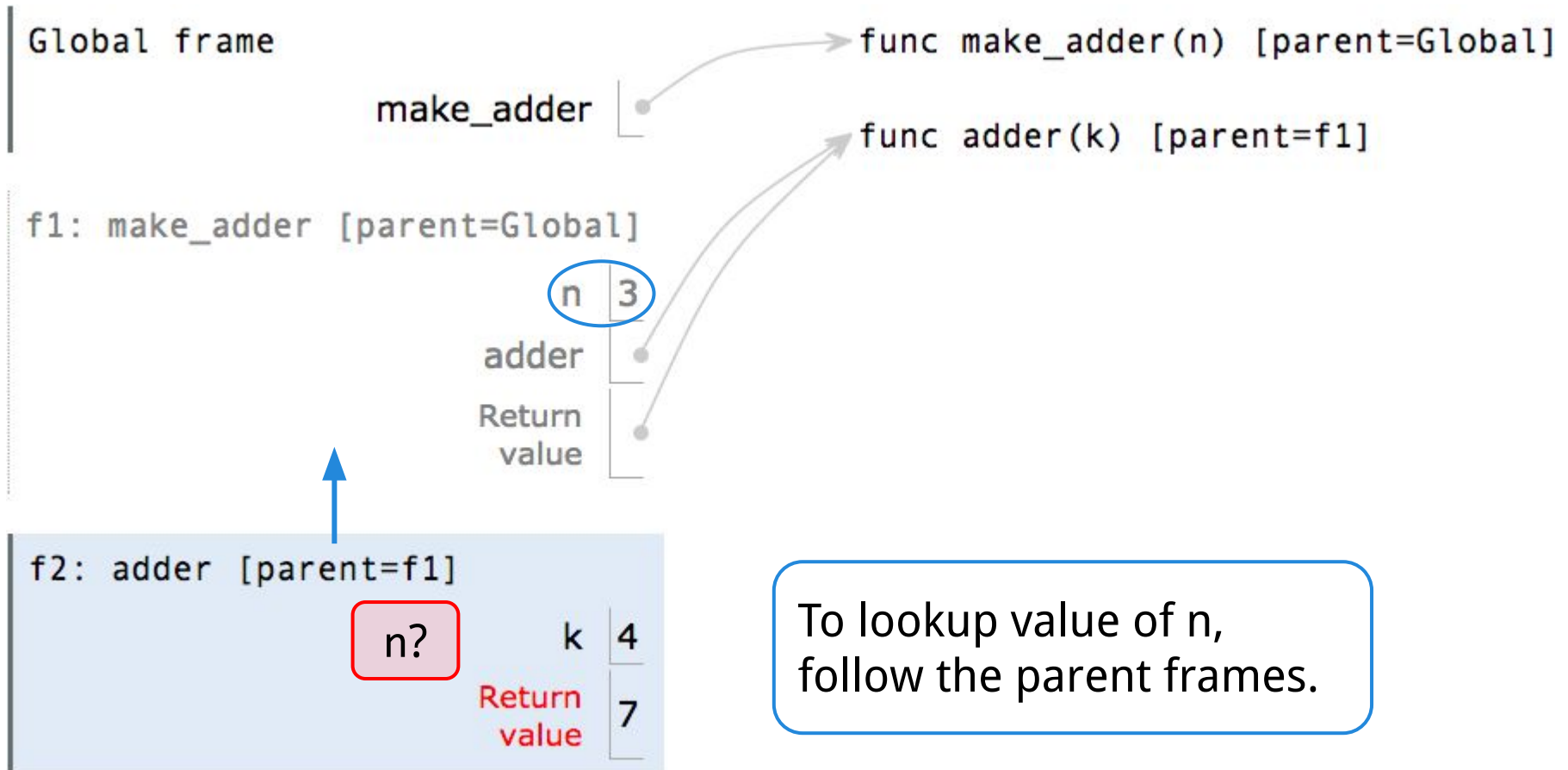
► make_adder(3)(4)



Environment diagram: make_adder

```
def make_adder(n):  
    def adder(k):  
        ► return k + n  
    return adder
```

make_adder(3)(4)



To lookup value of `n`, follow the parent frames.

Higher Order Functions

1. Review: functions as arguments
2. Nested functions
3. Functions as return values
4. Lambda functions

make_adder

```
def make_adder(n):  
    def adder(k):  
        return k + n  
    return adder
```

Why bother naming adder if we only use the name once?

Lambda functions

```
def make_adder(n):  
    return lambda k: n + k
```

The `lambda` expression creates an **anonymous function**!

```
lambda <parameters>: <return expression>
```

Translating between named and anonymous functions

No "return"!

The diagram illustrates the translation of a lambda function into a named function definition. It features two lines of code with colored annotations and arrows. The top line is a lambda function: `square = lambda x: x * x`. The bottom line is a named function definition: `def square(x):` followed by `return x * x` on the next line. A blue arrow points from the `square` variable in the lambda expression to the `square` function name in the definition. A green arrow points from the parameter `x` in the lambda expression to the parameter `x` in the function definition. A red arrow points from the first `x` in the lambda's body (`x * x`) to the first `x` in the function's body (`return x * x`).

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

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

Translating between named and anonymous functions

The diagram illustrates the translation of a lambda function assignment into a named function definition. It consists of two lines of code with three colored arrows pointing from the lambda version to the def version:

```
make_adder = lambda n: lambda k: n + k
```

→

```
def make_adder(n):  
    return lambda k: n + k
```

The arrows indicate the following mappings:

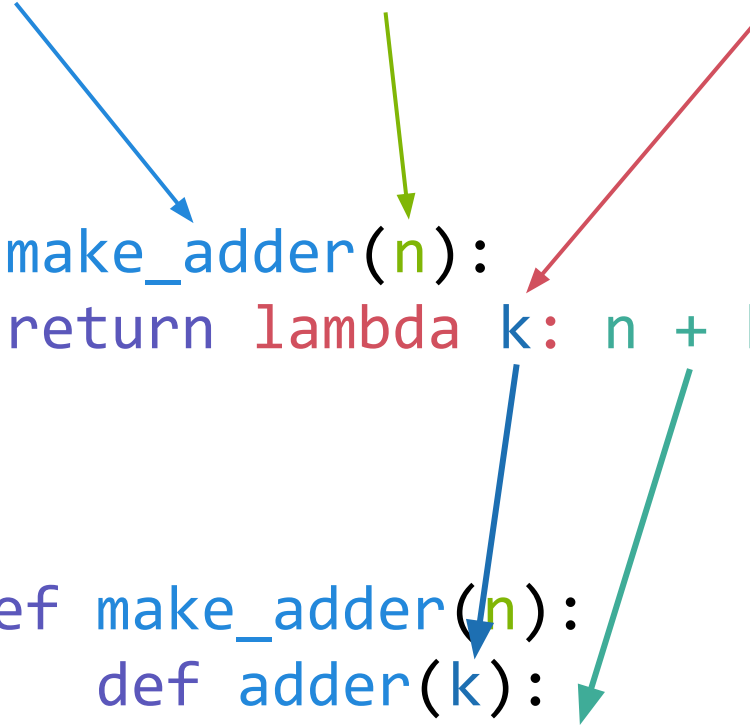
- A blue arrow points from `make_adder` in the lambda line to `make_adder` in the `def` line.
- A green arrow points from the parameter `n` in the lambda line to the parameter `n` in the `def` line.
- A red arrow points from the lambda body `lambda k: n + k` in the lambda line to the return value `lambda k: n + k` in the `def` line.

Translating between named and anonymous functions

```
make_adder = lambda n: lambda k: n + k
```

```
def make_adder(n):  
    return lambda k: n + k
```

```
def make_adder(n):  
    def adder(k):  
        return n + k  
    return adder
```



Lambda functions

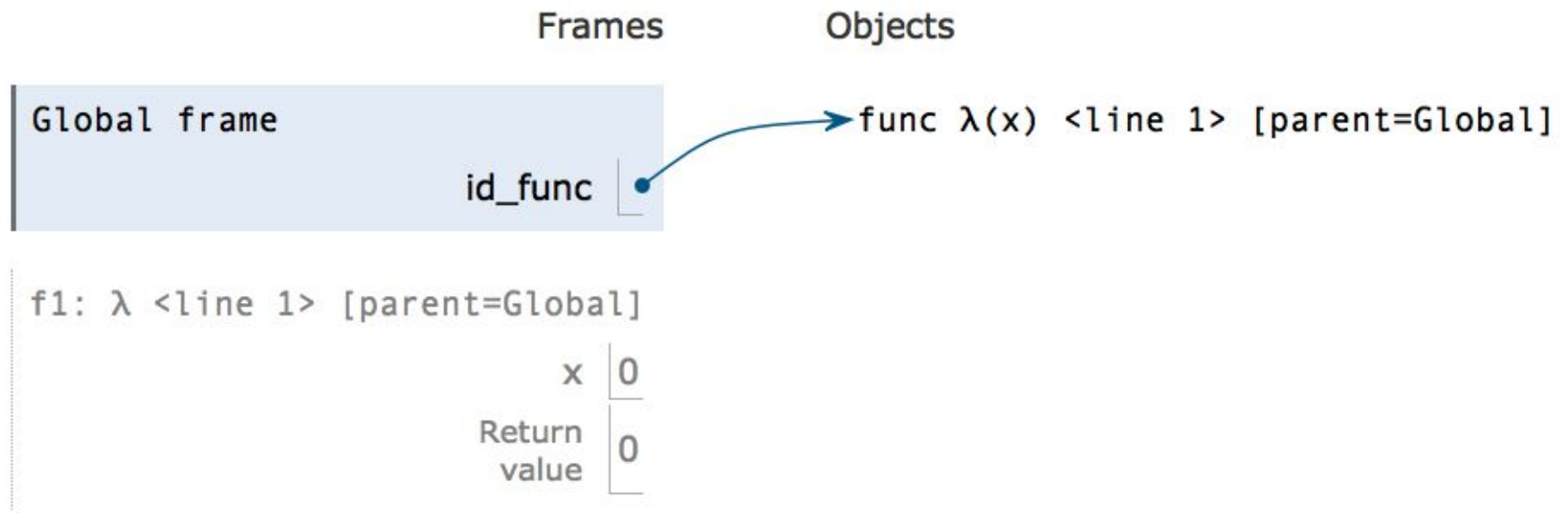
Lambdas are expressions! You can use them anywhere you can use any other expression.

A lambda's body can only be a *single* expression. This is the return expression.

Use sparingly! Better to have clear names.

Environment diagram: lambdas

```
id_func = lambda x: x  
id_func(0)
```



Environment diagram: make_adder

```
def make_adder(n):  
    return lambda k: n + k
```

```
add_3 = make_adder(3)  
add_3(4)
```

[\(Interactive environment diagram\)](#)

Lambda functions: practice

```
>>> def summation(term, n):  
...     i, total = 1, 0  
...     while i <= n:  
...         total += term(i)  
...         i += 1  
...     return total  
...  
>>> summation(lambda x: x**2, 3)  
14
```

Lambda functions: practice

```
def repeat(f, x):  
    while f(x) != x:  
        x = f(x)  
    return x  
  
def g(y):  
    return (y + 5) // 3  
  
result = repeat(g, 5)
```

Hog demo!