Pixabay // Schwoaze

BBM 101

Introduction to Programming I

Lecture #08 – Higher-Order Functions

HACETTEPE UNIVERSITY

Fuat Akal & Erkut Erdem // Fall 2020

Last time... **Tuples, Sets, and Dictionaries**

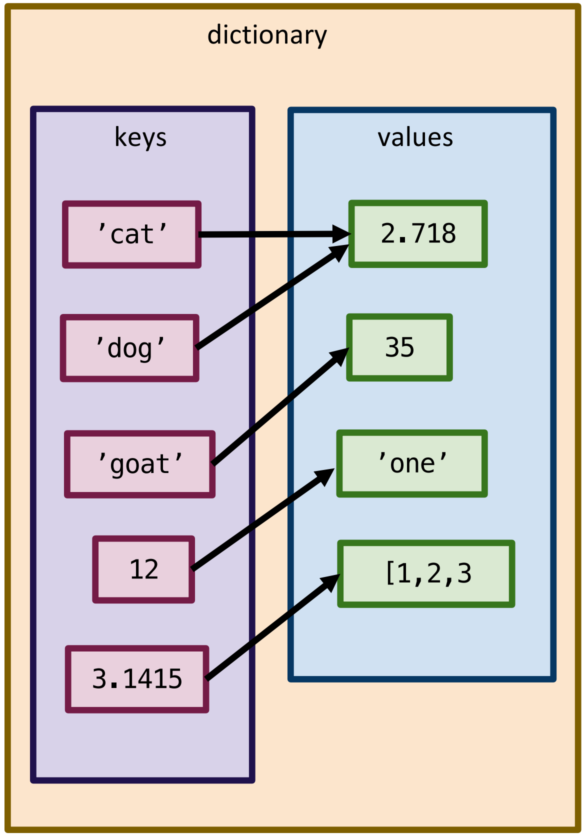
Tuples

```
t1 = ()
t2 = (1, 'two', 3)
print(t1)
print(t2)

>> ()
>> (1, 'two', 3)
```

Sets

```
odd = set([1, 3, 5])
prime = set([2, 3, 5])
empty = set([])
```

Dictionaries



Lecture Overview

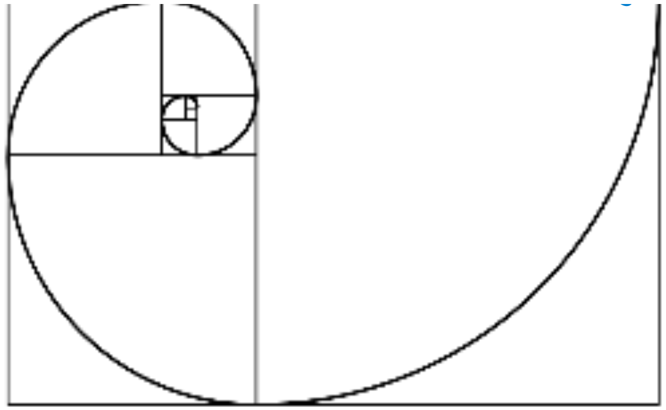
- Iteration Example: The Fibonacci Sequence
- Designing Functions
- Generalization
- Higher-Order Functions
- Functions as Return Values
- Lambda Expressions
- Filter, Map, and Reduce Functions

Disclaimer: Much of the material and slides for this lecture were borrowed from
—John DeNero's Berkeley CS 61A
—Swami Iyer's Umass Boston CS110 class

Lecture Overview

- Iteration Example: The Fibonacci Sequence
- Designing Functions
- Generalization
- Higher-Order Functions
- Functions as Return Values
- Lambda Expressions
- Filter, Map, and Reduce Functions

The Fibonacci Sequence



0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987

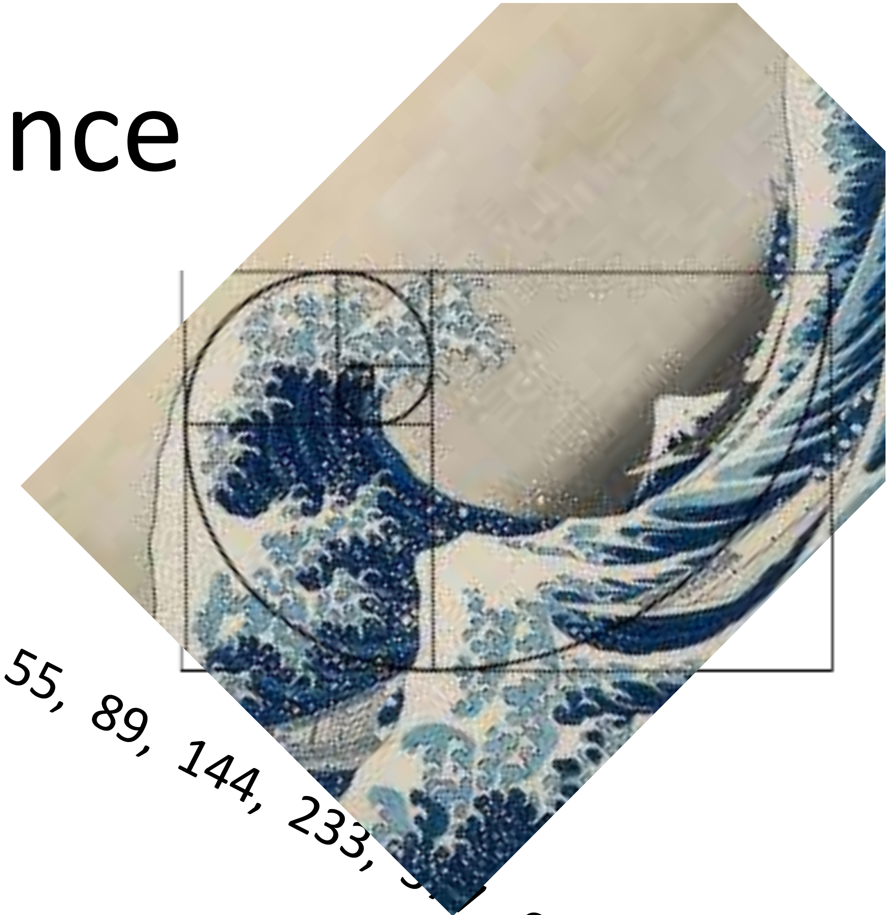
The Fibonacci Seq

0, 1, 1, 2, 3, 5, 8, 13, 21

233, 377, 610, 987

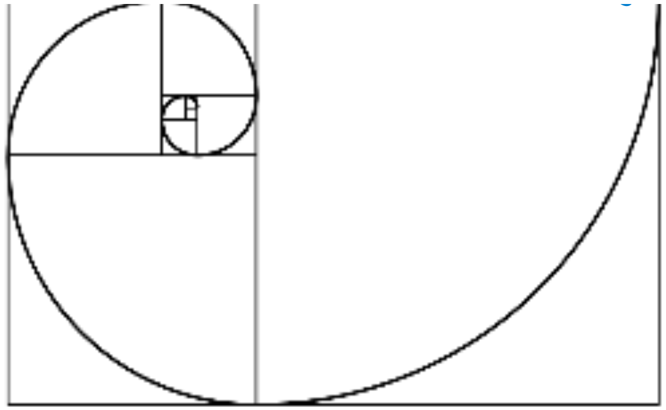
The Fibonacci Sequence

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 3[illegible], 610, 987



The Fibonacci Sequence

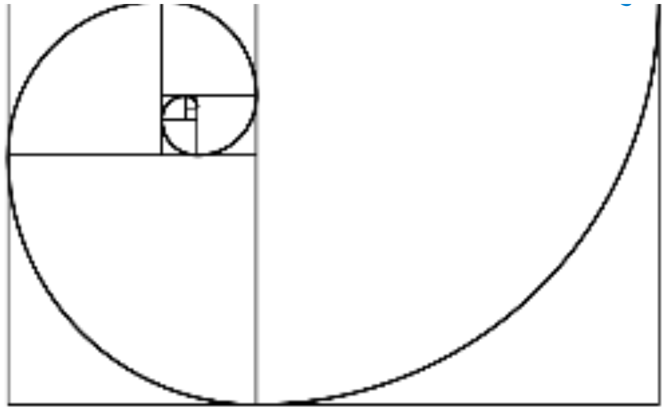
0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987



```python
def fib(n):
    """Compute the nth Fibonacci number, for N >= 1."""
    pred, curr = 0, 1 # 0th and 1st Fibonacci numbers
    k = 1             # curr is the kth Fibonacci number
    while k < n:
        pred, curr = curr, pred + curr
        k=k+1
    return curr
```

The Fibonacci Sequence

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987

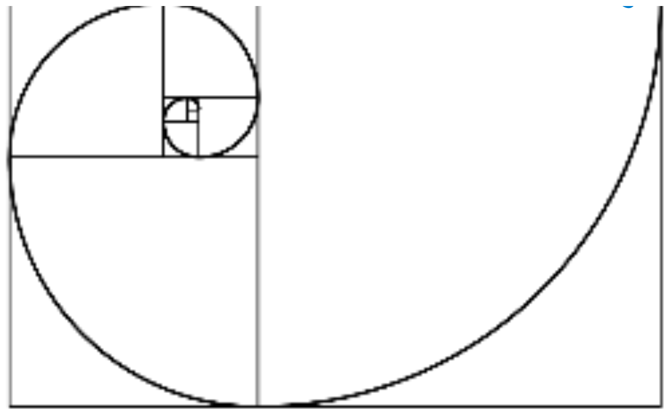


```
def fib(n):
    """Compute the nth Fibonacci number, for N >= 1."""
    pred, curr = 0, 1 # 0th and 1st Fibonacci numbers
    k = 1             # curr is the kth Fibonacci number
    while k < n:
        pred, curr = curr, pred + curr
        k=k+1
    return curr
```

The next Fibonacci number is the sum of the current one and its predecessor

The Fibonacci Sequence

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987

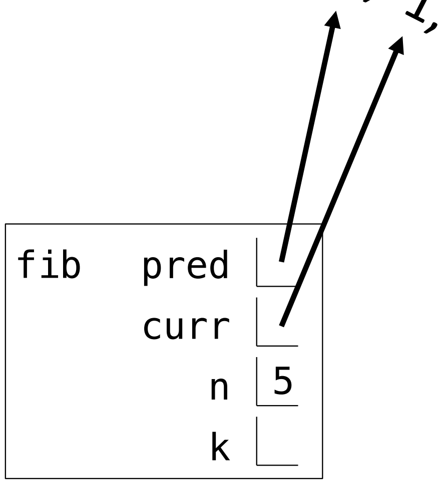


fib		
	pred	
	curr	
	n	5
	k	

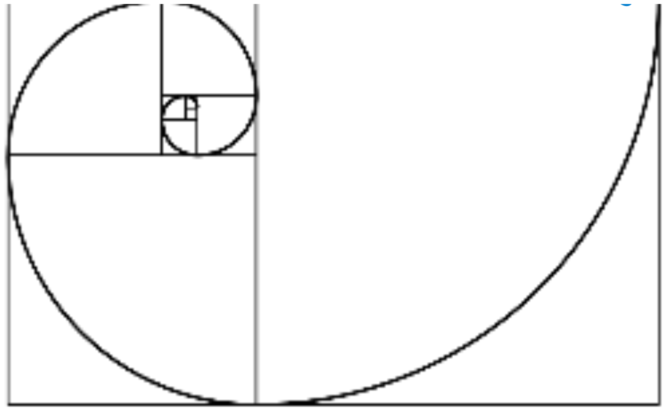
```
def fib(n):
    """Compute the nth Fibonacci number, for N >= 1."""
    pred, curr = 0, 1 # 0th and 1st Fibonacci numbers
    k = 1              # curr is the kth Fibonacci number
    while k < n:
        pred, curr = curr, pred + curr
        k=k+1
    return curr
```

The next Fibonacci number is the sum of the current one and its predecessor

The Fibonacci Sequence



0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987

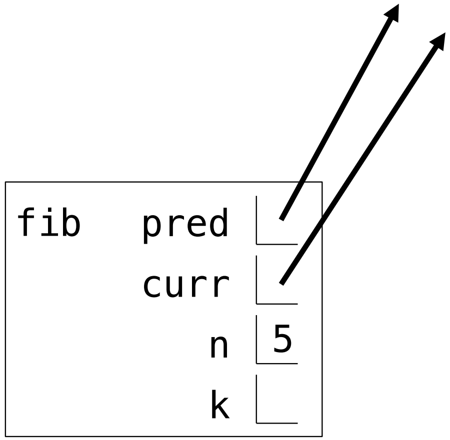


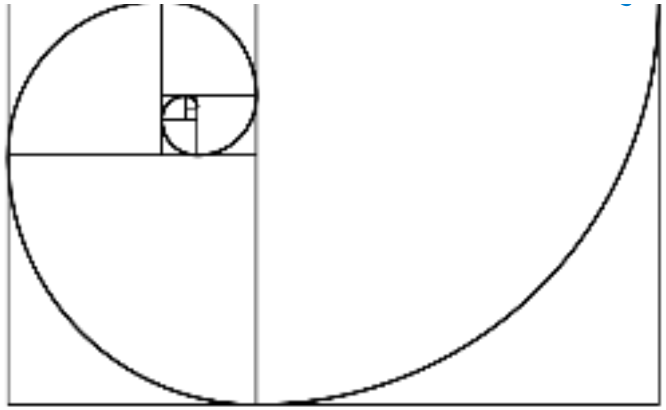
```python
def fib(n):
    """Compute the nth Fibonacci number, for N >= 1."""
    pred, curr = 0, 1 # 0th and 1st Fibonacci numbers
    k = 1              # curr is the kth Fibonacci number
    while k < n:
        pred, curr = curr, pred + curr
        k=k+1
    return curr
```

The next Fibonacci number is the sum of the current one and its predecessor

The Fibonacci Sequence

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987



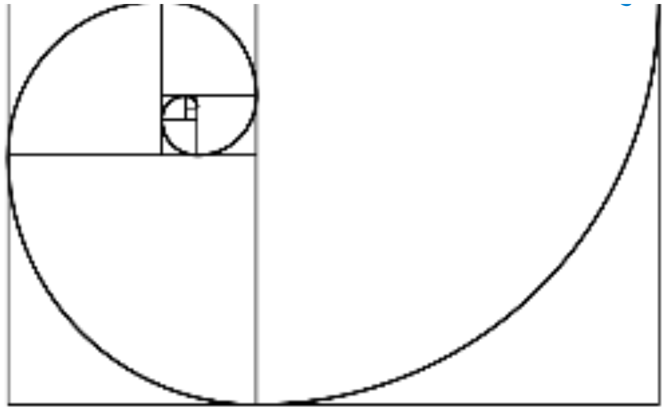


```
def fib(n):
    """Compute the nth Fibonacci number, for N >= 1."""
    pred, curr = 0, 1 # 0th and 1st Fibonacci numbers
    k = 1             # curr is the kth Fibonacci number
    while k < n:
        pred, curr = curr, pred + curr
        k=k+1
    return curr
```

The next Fibonacci number is the sum of the current one and its predecessor

The Fibonacci Sequence

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987



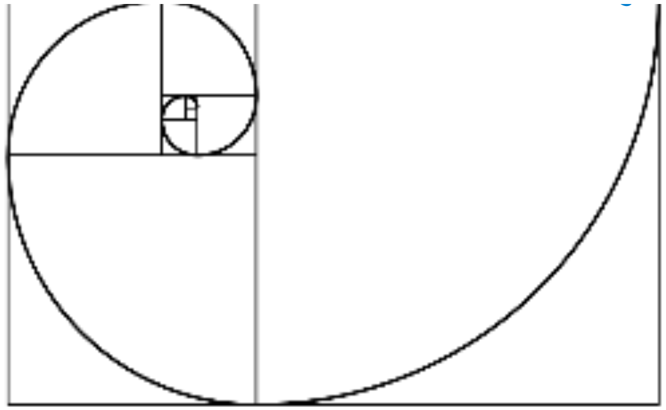
```
fib   pred
      curr
      n     5
      k
```

```
def fib(n):
    """Compute the nth Fibonacci number, for N >= 1."""
    pred, curr = 0, 1 # 0th and 1st Fibonacci numbers
    k = 1             # curr is the kth Fibonacci number
    while k < n:
        pred, curr = curr, pred + curr
        k=k+1
    return curr
```

The next Fibonacci number is the sum of the current one and its predecessor

The Fibonacci Sequence

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987



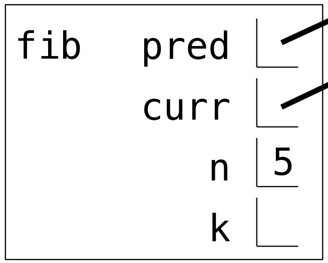
fib		
	pred	
	curr	
	n	5
	k	

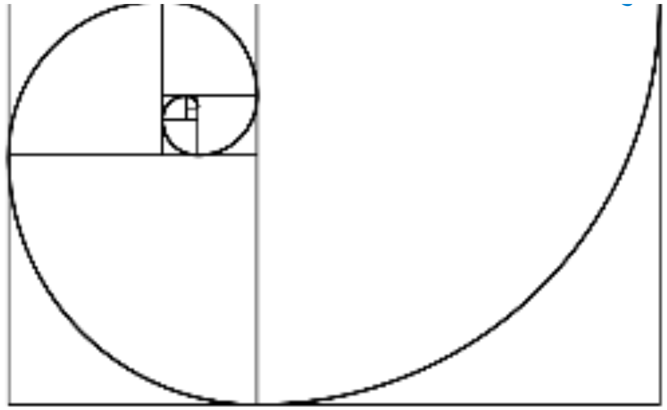
```
def fib(n):
    """Compute the nth Fibonacci number, for N >= 1."""
    pred, curr = 0, 1 # 0th and 1st Fibonacci numbers
    k = 1             # curr is the kth Fibonacci number
    while k < n:
        pred, curr = curr, pred + curr
        k=k+1
    return curr
```

The next Fibonacci number is the sum of the current one and its predecessor

The Fibonacci Sequence

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987





```
def fib(n):
    """Compute the nth Fibonacci number, for N >= 1."""
    pred, curr = 0, 1 # 0th and 1st Fibonacci numbers
    k = 1             # curr is the kth Fibonacci number
    while k < n:
        pred, curr = curr, pred + curr
        k=k+1
    return curr
```

The next Fibonacci number is the sum of the current one and its predecessor

Lecture Overview

- Iteration Example: The Fibonacci Sequence
- Designing Functions
- Generalization
- Higher-Order Functions
- Functions as Return Values
- Lambda Expressions
- Filter, Map, and Reduce Functions

Describing Functions

```
def square(x):
    """Return X * X."""»
```

- A function's ***domain*** is the set of all inputs it might possibly take as arguments.

x is a number

- A function's ***range*** is the set of output values it might possibly return.

square returns a non-negative real number

- A pure function's ***behavior*** is the relationship it creates between input and output.

square returns the square of x

A Guide to Designing Function

- Give each function exactly one job, but make it apply to many related situations

```
>>> round(1.23)
1
```

```
>>> round(1.23,1)
1.2
```

```
>>> round(1.23,0)
1
```

- Don’t repeat yourself (DRY): Implement a process just once, but execute it many times

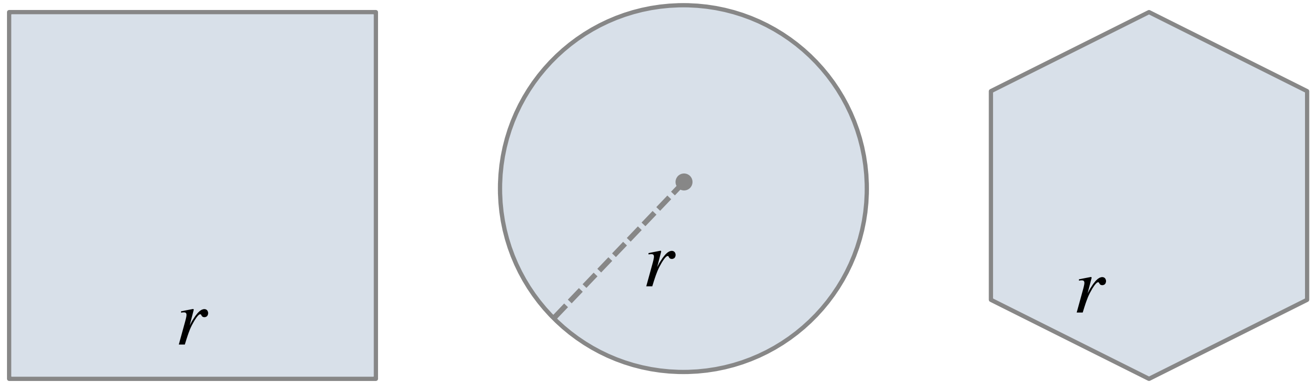
Lecture Overview

- Iteration Example: The Fibonacci Sequence
- Designing Functions
- Generalization
- Higher-Order Functions
- Functions as Return Values
- Lambda Expressions
- Filter, Map, and Reduce Functions

Generalizing Patterns with Arguments

- Regular geometric shapes relate length and area.

Shape:

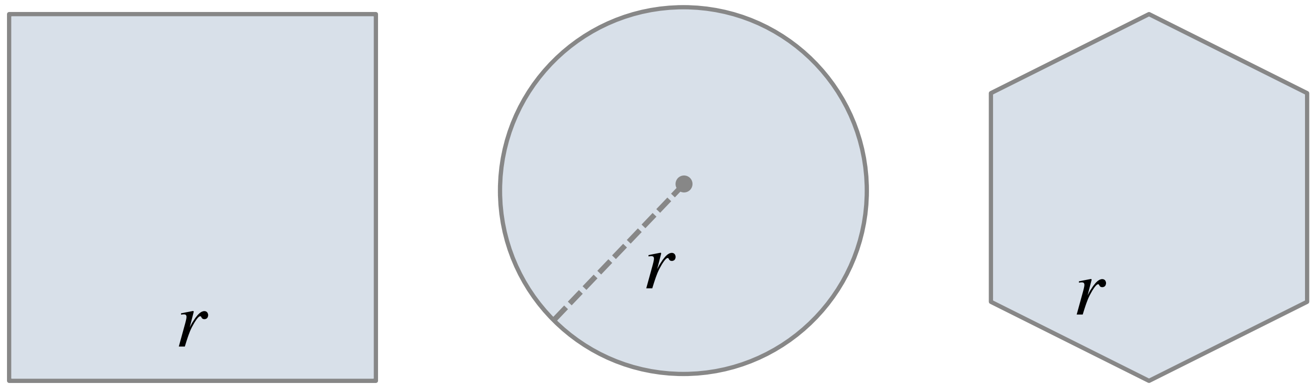


Area: r^2 $\pi \cdot r^2$ $\frac{3\sqrt{3}}{2} \cdot r^2$

Generalizing Patterns with Arguments

- Regular geometric shapes relate length and area.

Shape:

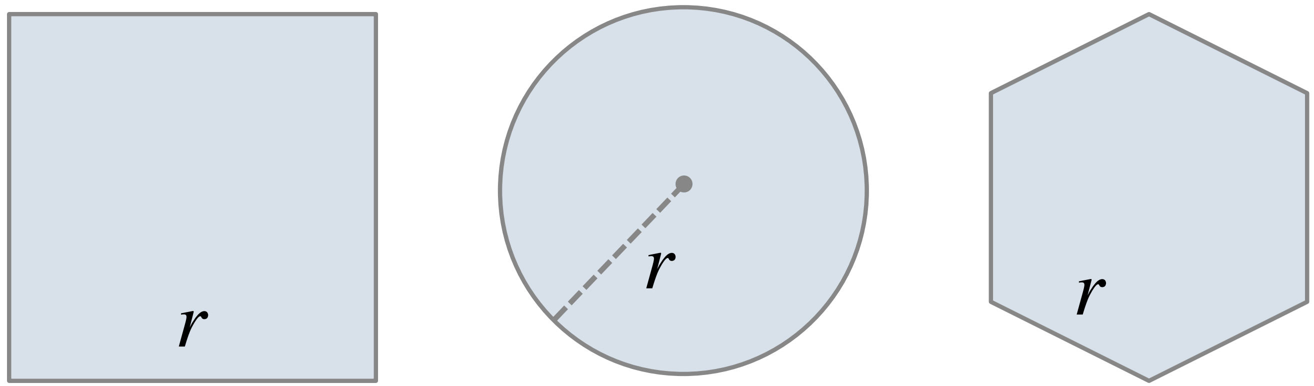


Area: $1 \cdot r^2$ $\quad$ $\pi \cdot r^2$ $\quad$ $\frac{3\sqrt{3}}{2} \cdot r^2$

Generalizing Patterns with Arguments

- Regular geometric shapes relate length and area.

Shape:



Area: $\boxed{1} \cdot r^2$ $\boxed{\pi} \cdot r^2$ $\boxed{\frac{3\sqrt{3}}{2}} \cdot r^2$

Finding common structure allows for shared implementation!

Generalizing Patterns with Arguments

Solution 1

```
from math import pi, sqrt

def area_square(r):
    """Return the area of a
    square with side length R."""
    return r * r

def area_circle(r):
    """Return the area of a
    circle with radius R."""
    return r * r * pi

def area_hexagon(r):
    """Return the area of a
    regular hexagon with side
    length R."""
    return r * r * 3 * sqrt(3)/2
```

Solution 2

```
def area(r, shape_constant):
    """Return the area of a shape
    from length measurement R."""
    return r * r * shape_constant

def area_square(r):
    return area(r, 1)

def area_circle(r):
    return area(r, pi)

def area_hexagon(r):
    return area(r, 3 * sqrt(3)/2)
```

Finding common structure allows for shared implementation!

Lecture Overview

- Iteration Example: The Fibonacci Sequence
- Designing Functions
- Generalization
- Higher-Order Functions
- Functions as Return Values
- Lambda Expressions
- Filter, Map, and Reduce Functions

Generalizing Over Computational Processes

The common structure among functions may be a computational process, rather than a number.

$$\sum_{k=1}^{5} k = 1 + 2 + 3 + 4 + 5 \qquad = 15$$

$$\sum_{k=1}^{5} k^3 = 1^3 + 2^3 + 3^3 + 4^3 + 5^3 \qquad = 225$$

$$\sum_{k=1}^{5} \frac{8}{(4k-3)\cdot(4k-1)} = \frac{8}{3} + \frac{8}{35} + \frac{8}{99} + \frac{8}{195} + \frac{8}{323} \qquad = 3.04$$

Generalizing Over Computational Processes

The common structure among functions may be a computational process, rather than a number.

$$\sum_{k=1}^{5} \boxed{k} = 1 + 2 + 3 + 4 + 5 \qquad = 15$$

$$\sum_{k=1}^{5} \boxed{k^3} = 1^3 + 2^3 + 3^3 + 4^3 + 5^3 \qquad = 225$$

$$\sum_{k=1}^{5} \boxed{\frac{8}{(4k-3)\cdot(4k-1)}} = \frac{8}{3} + \frac{8}{35} + \frac{8}{99} + \frac{8}{195} + \frac{8}{323} \qquad = 3.04$$

Summation Example

```
def sum_naturals(n):
    """Sum the first N
    natural numbers.

    >>> sum_naturals(5)
    15
    """
    total, k = 0, 1
    while k <= n:
        total = total + k
        k = k + 1
    return total
```

```
def sum_cubes(n):
    """Sum the first N cubes of
    natural numbers.

    >>> sum_cubes(5)
    225
    """
    total, k = 0, 1
    while k <= n:
        total = total + pow(k,3)
        k= k + 1
    return total
```

Summation Example

```
def summation(n, term):
    """Sum the first N terms
    of a sequence.

    >>> summation(5, cube)
    225
    """
    total, k = 0, 1
    while k <= n:
        total = total + term(k)
        k = k + 1
    return total
```

```
def identity(k):
    return k


def cube(k):
    return pow(k, 3)


from operator import mul

def pi_term(k):
    return 8/mul(k*4-3,k*4-1)

>>> summation(1000000, pi_term)
```

Summation Example

```
def cube(k):
    return pow(k, 3)

def summation(n, term):
    """Sum the first n terms of a sequence.
    >>> summation(5, cube)
    225
    """
    total, k = 0, 1
    while k <= n:
        total, k = total + term(k), k + 1
    return total
```

Summation Example

```
def cube(k):
    return pow(k, 3)
```

Function of a single argument (not called "term")

```
def summation(n, term):
    """Sum the first n terms of a sequence.
    >>> summation(5, cube)
    225
    """
    total, k = 0, 1
    while k <= n:
        total, k = total + term(k), k + 1
    return total
```

A formal parameter that will be bound to a function

The cube function is passed as an argument value

0+1+8+27+64+125

The function bound to term gets called here

Lecture Overview

- Iteration Example: The Fibonacci Sequence
- Designing Functions
- Generalization
- Higher-Order Functions
- Functions as Return Values
- Lambda Expressions
- Filter, Map, and Reduce Functions

Locally Defined Functions

Functions defined within other function bodies are bound to names in a local frame

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

    >>> add_three = make_adder(3)
    >>> add_three(4) 7
    """
    def adder(k):
        return k + n

    return adder
```

Locally Defined Functions

Functions defined within other function bodies are bound to names in a local frame

A function that returns a function

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

    >>> add_three = make_adder(3)
    >>> add_three(4) 7
    """
    def adder(k):
        return k + n

    return adder
```

The name add_three is bound to a function

A def statement within another def statement

Can refer to names in the enclosing function

Call Expressions as Operator Expressions

```
make_adder(1)    (         2         )
```

Call Expressions as Operator Expressions

$$\underbrace{\texttt{make\_adder(1)}}^{\textit{Operator}} \quad (\qquad \texttt{2} \qquad)$$

Call Expressions as Operator Expressions

$$\underbrace{\texttt{make\_adder(1)}}_{\textit{Operator}} \quad (\quad \underbrace{\texttt{2}}_{\textit{Operand}} \quad)$$

Call Expressions as Operator Expressions

An expression that
evaluates to a function

Operator *Operand*

```
make_adder(1)    (        2        )
```

Call Expressions as Operator Expressions

An expression that evaluates to a function

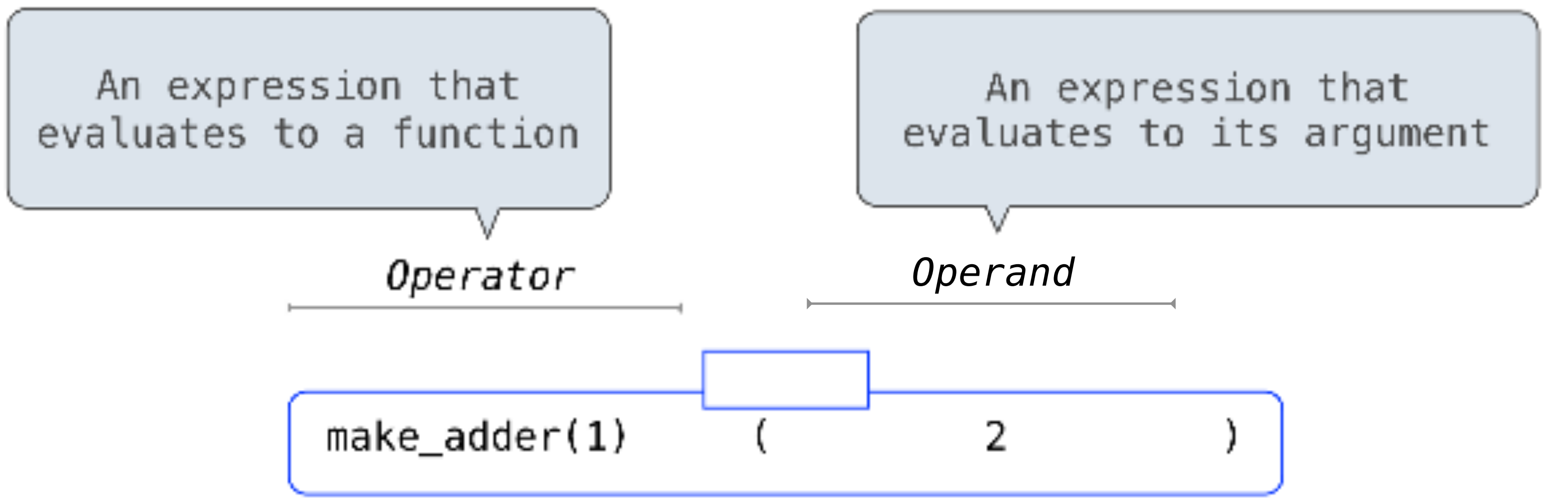
An expression that evaluates to its argument

Operator

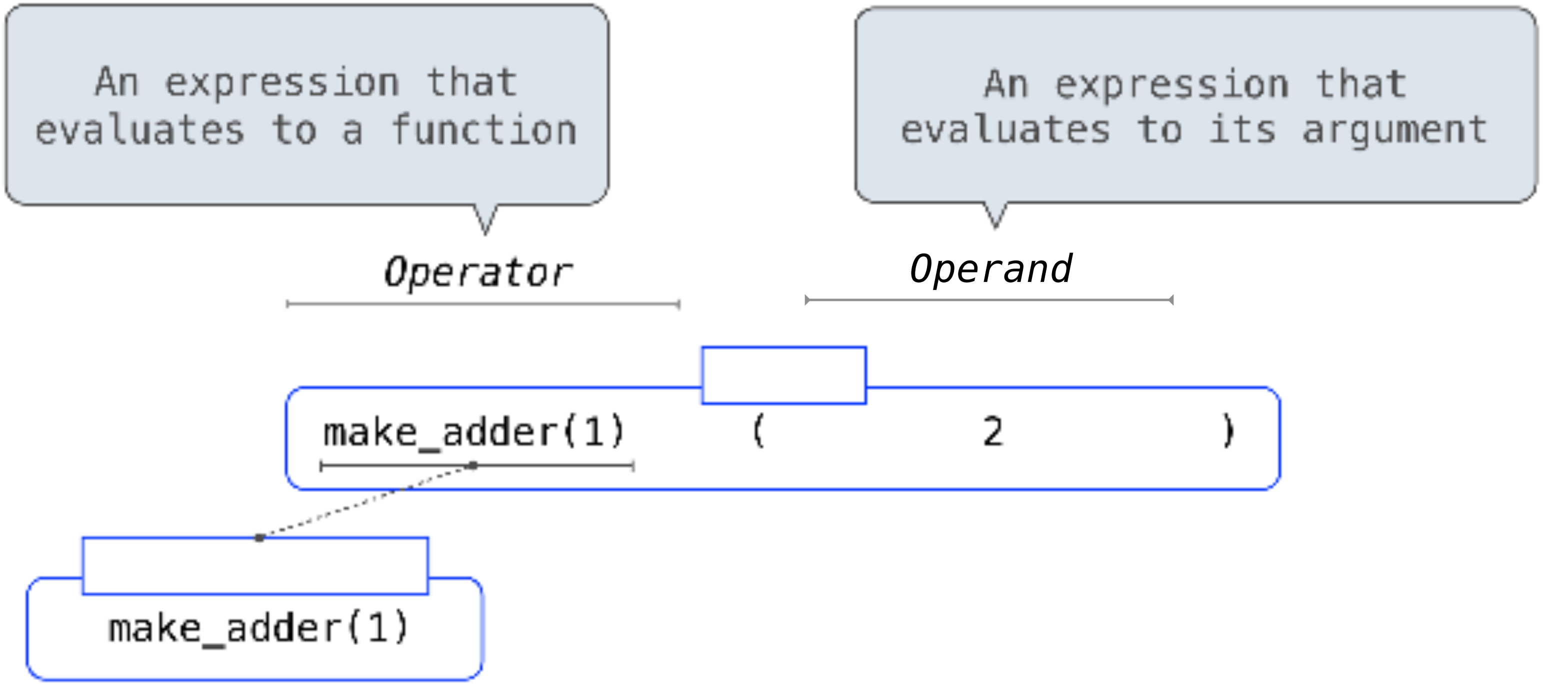
Operand

```
make_adder(1)     (        2         )
```

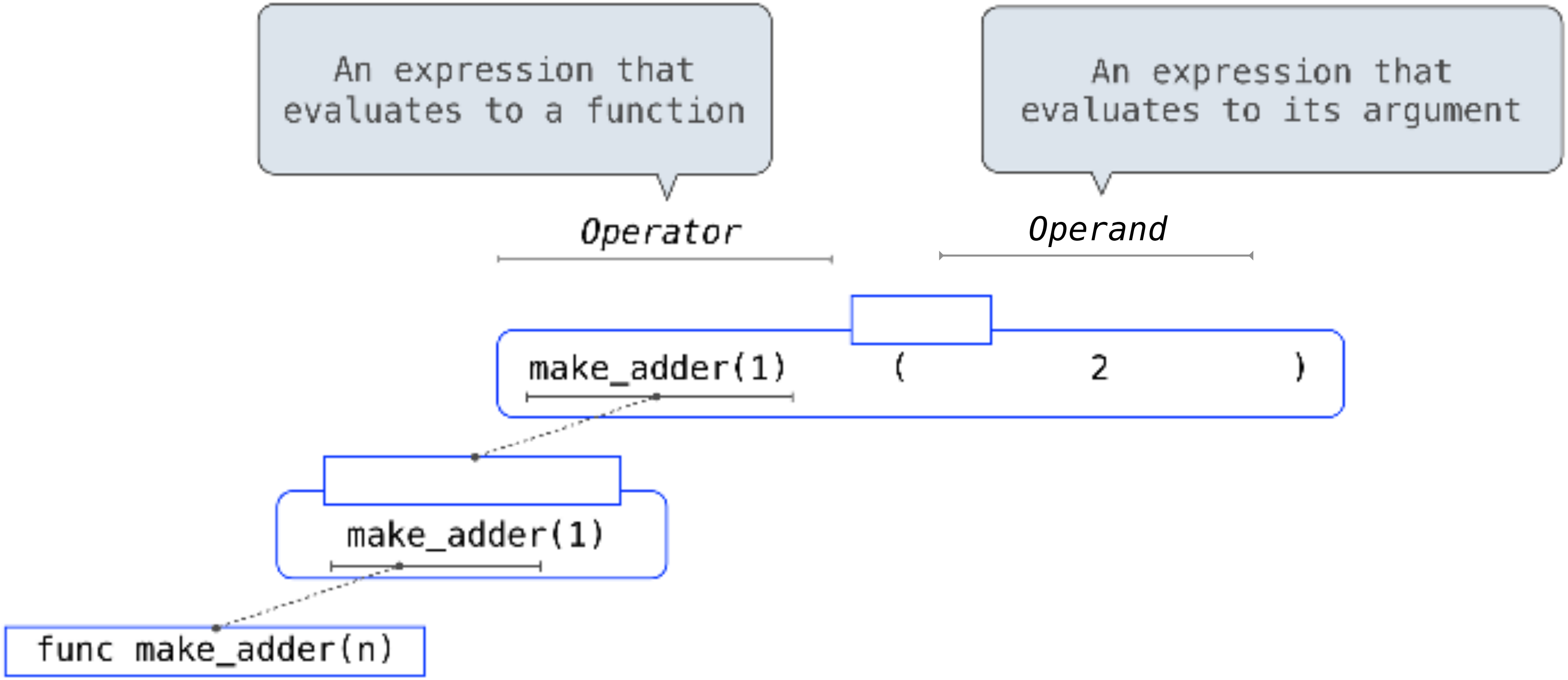
Call Expressions as Operator Expressions



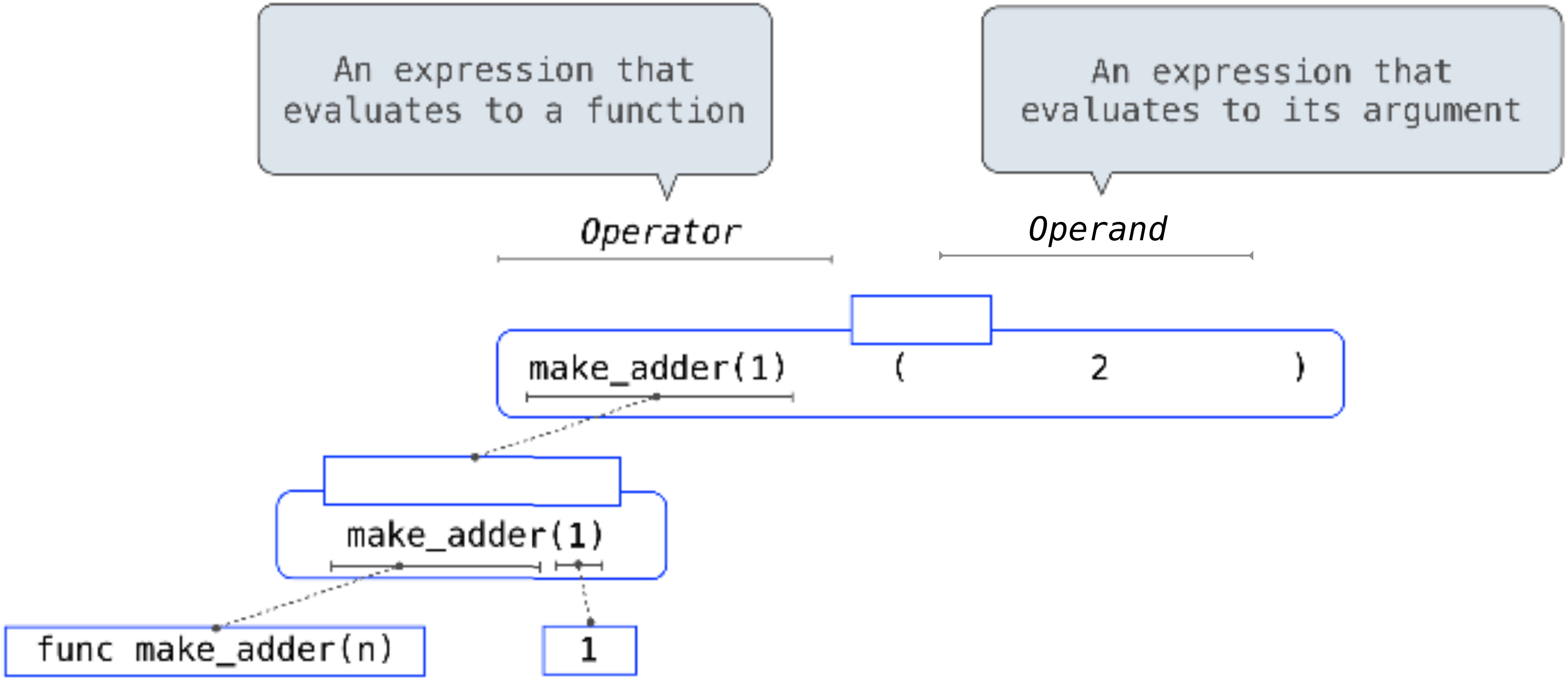
Call Expressions as Operator Expressions



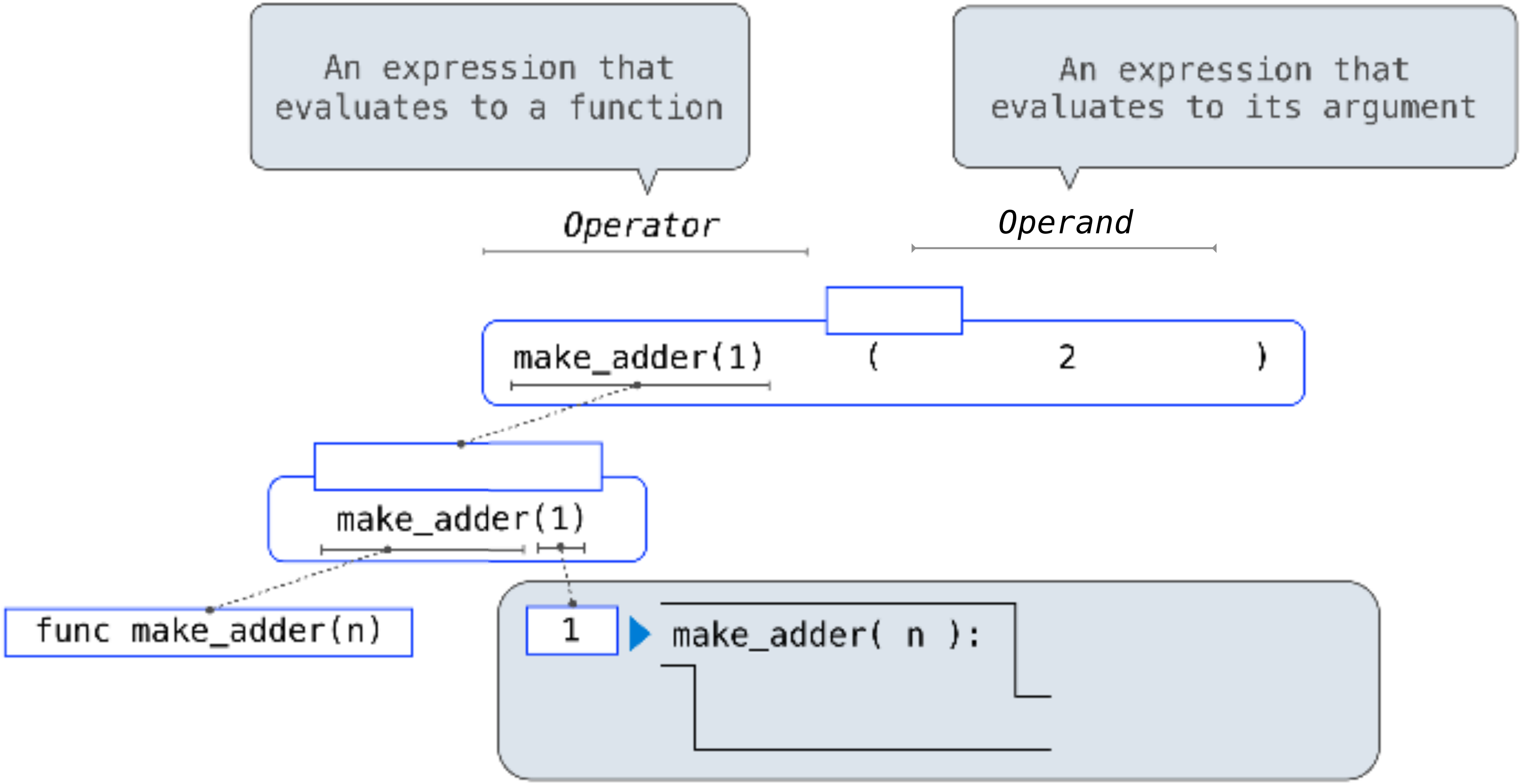
Call Expressions as Operator Expressions



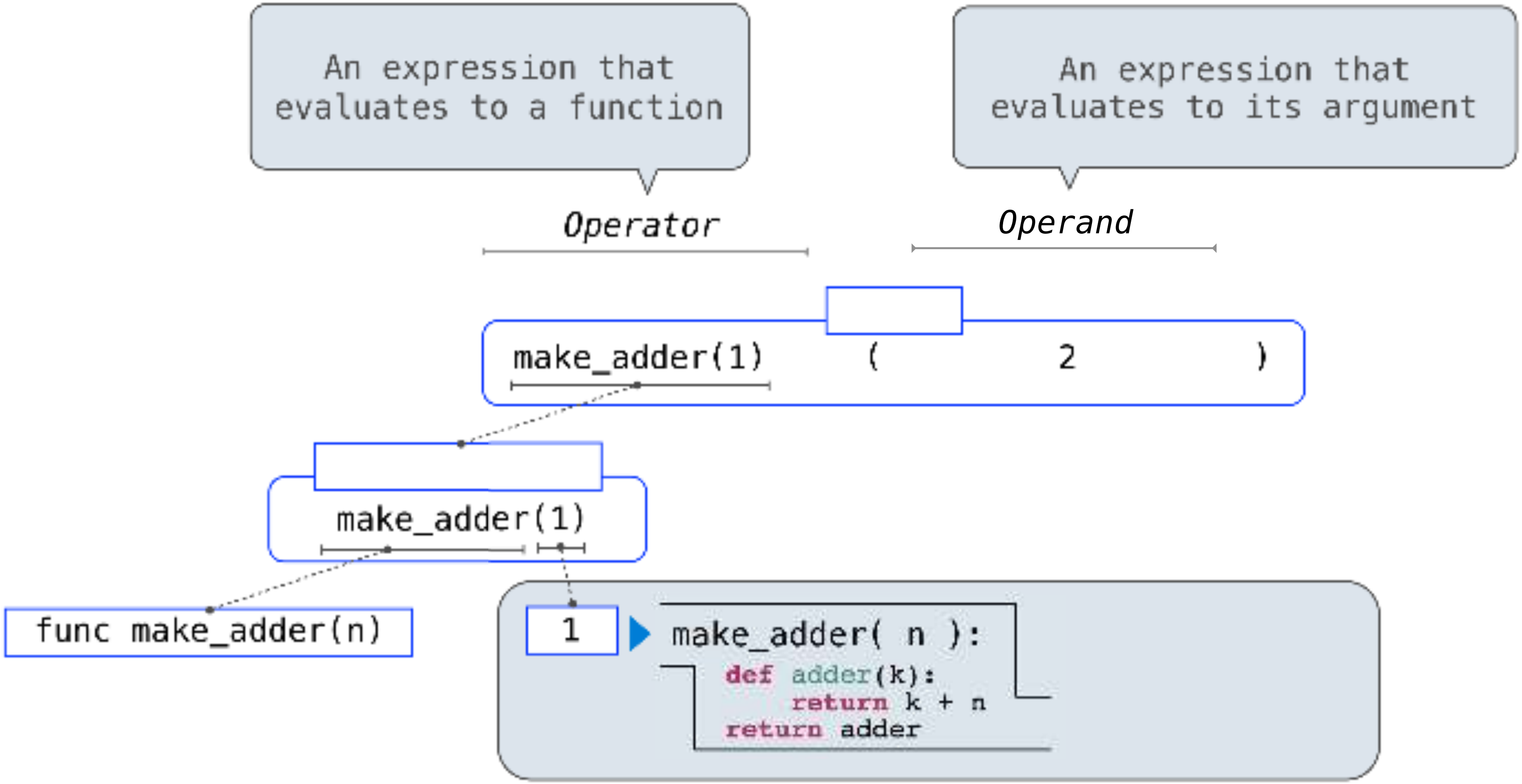
Call Expressions as Operator Expressions



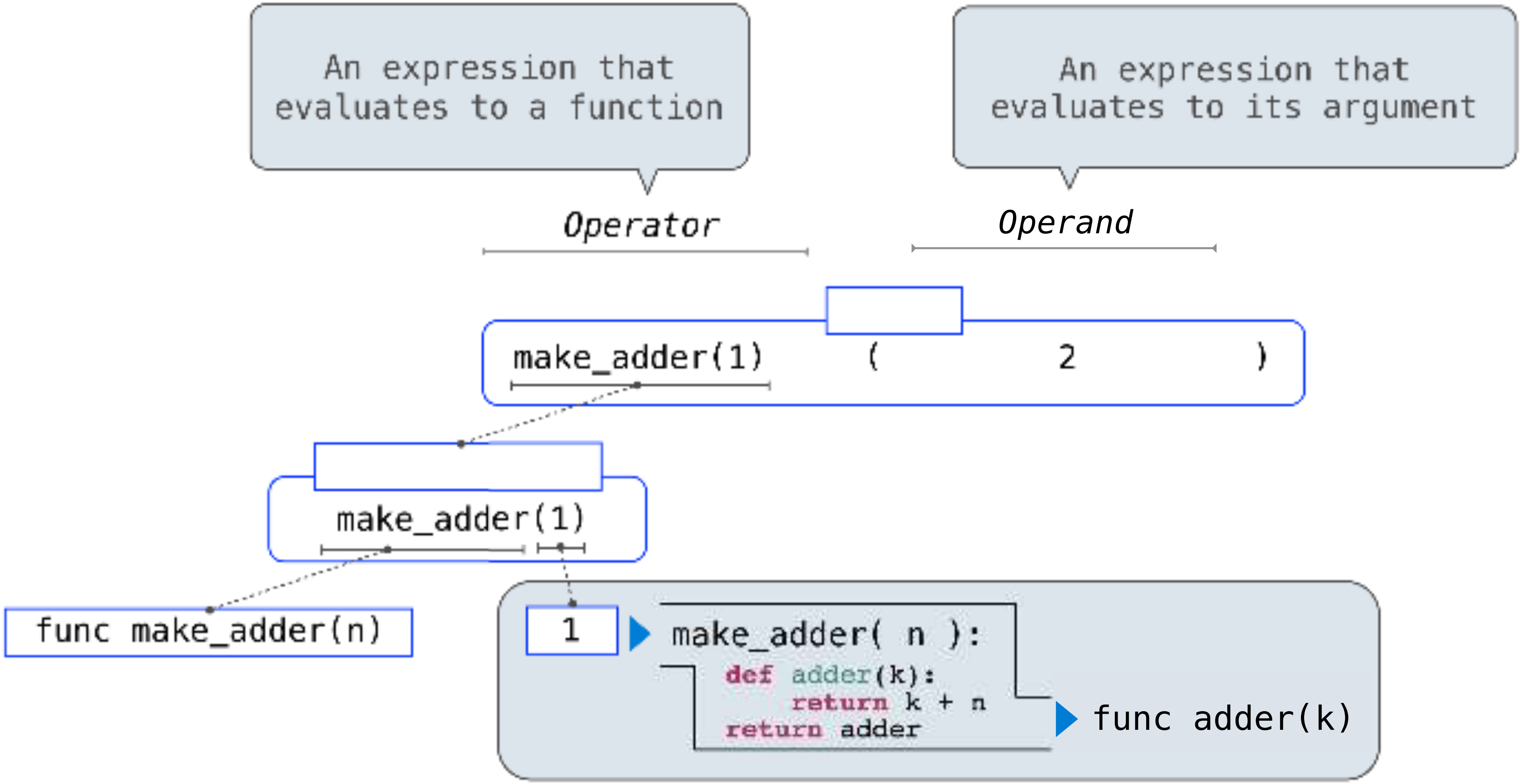
Call Expressions as Operator Expressions



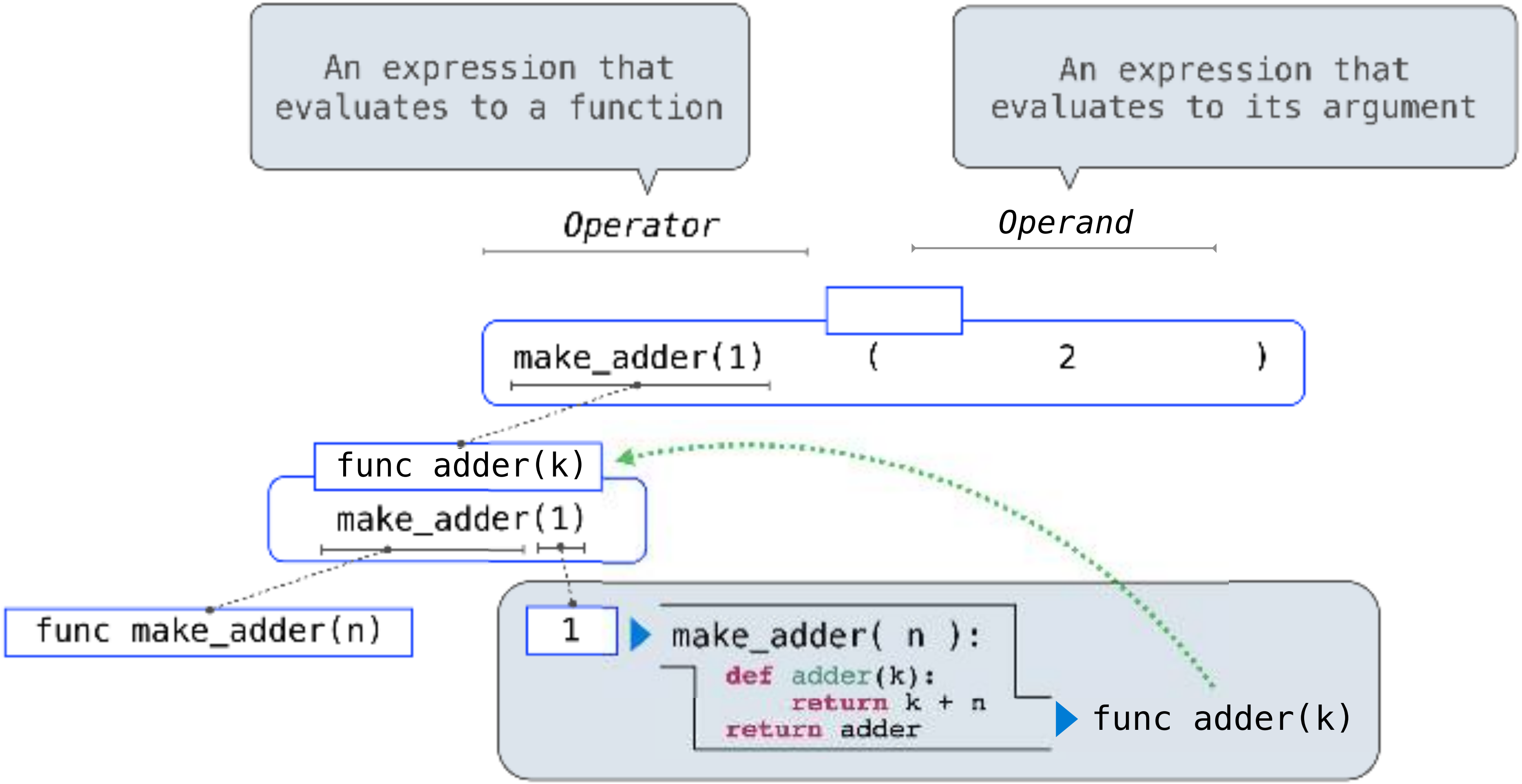
Call Expressions as Operator Expressions



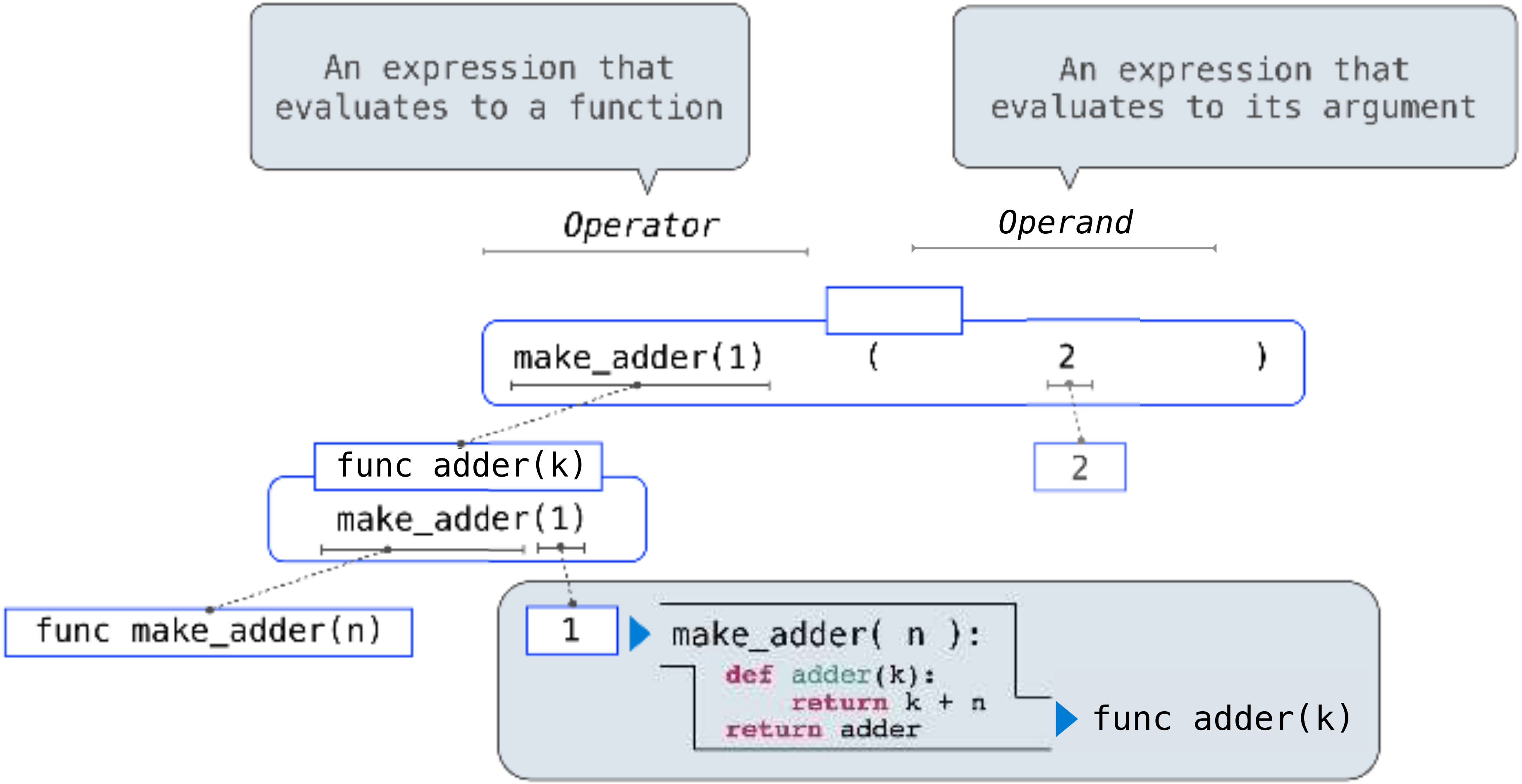
Call Expressions as Operator Expressions



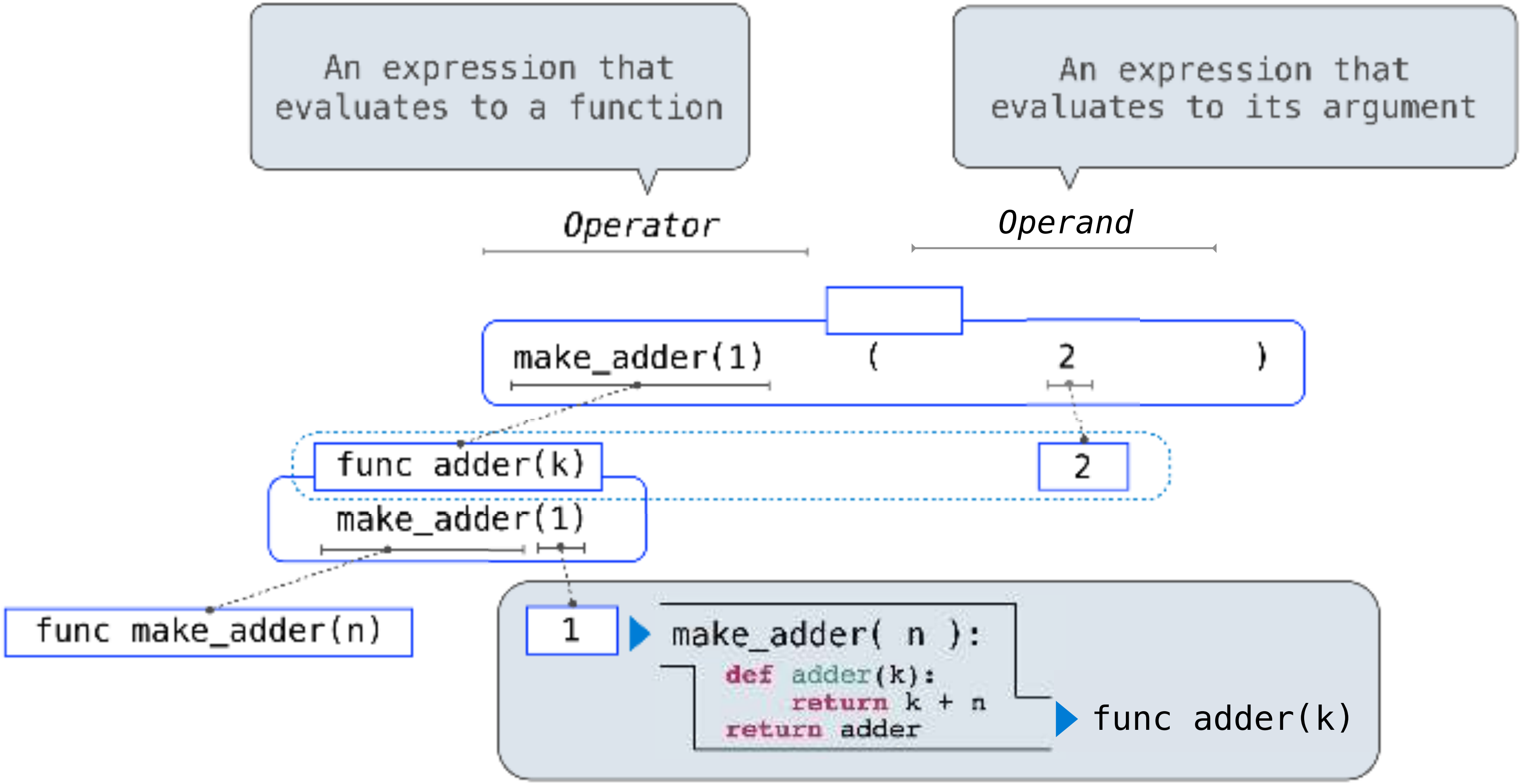
Call Expressions as Operator Expressions



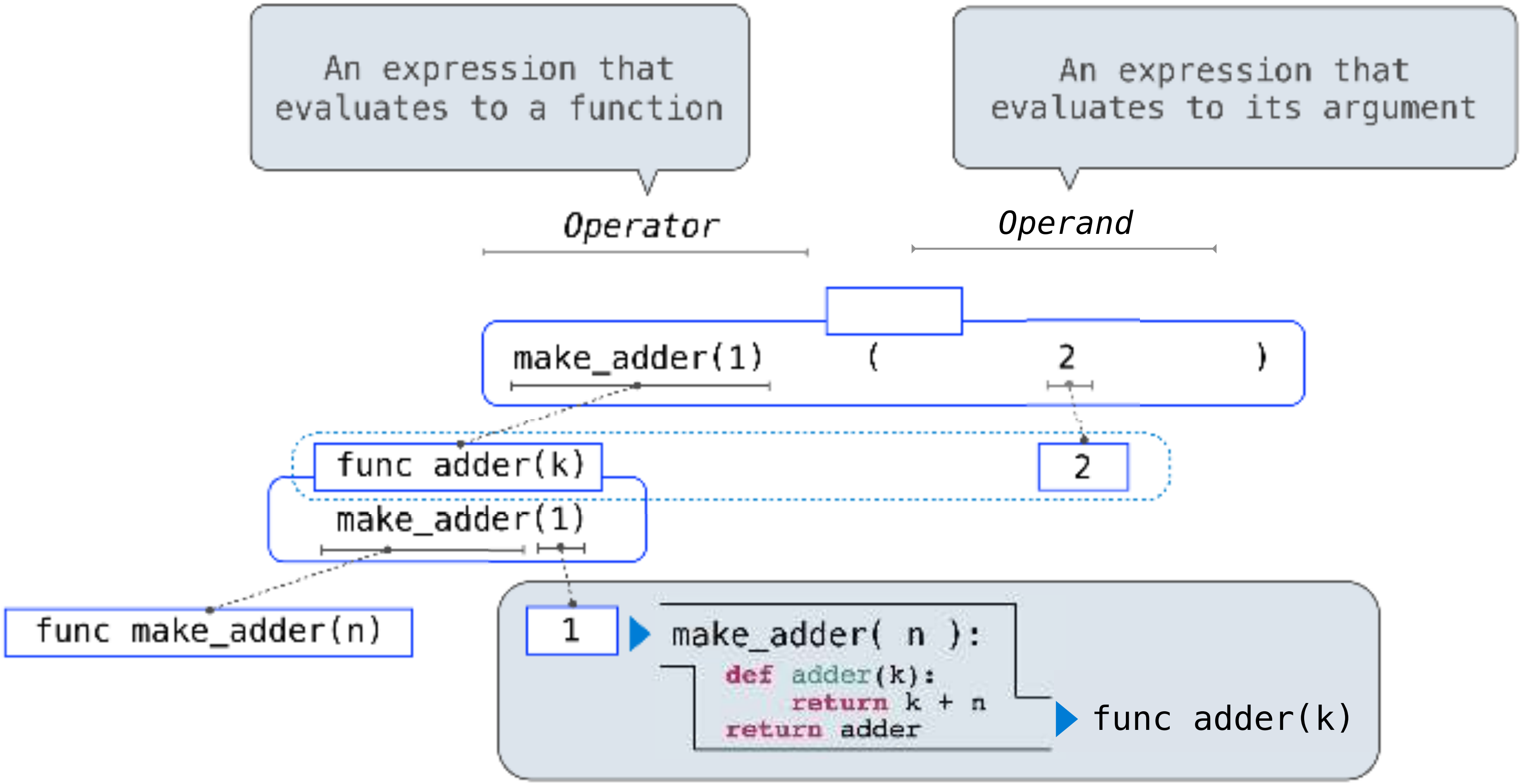
Call Expressions as Operator Expressions



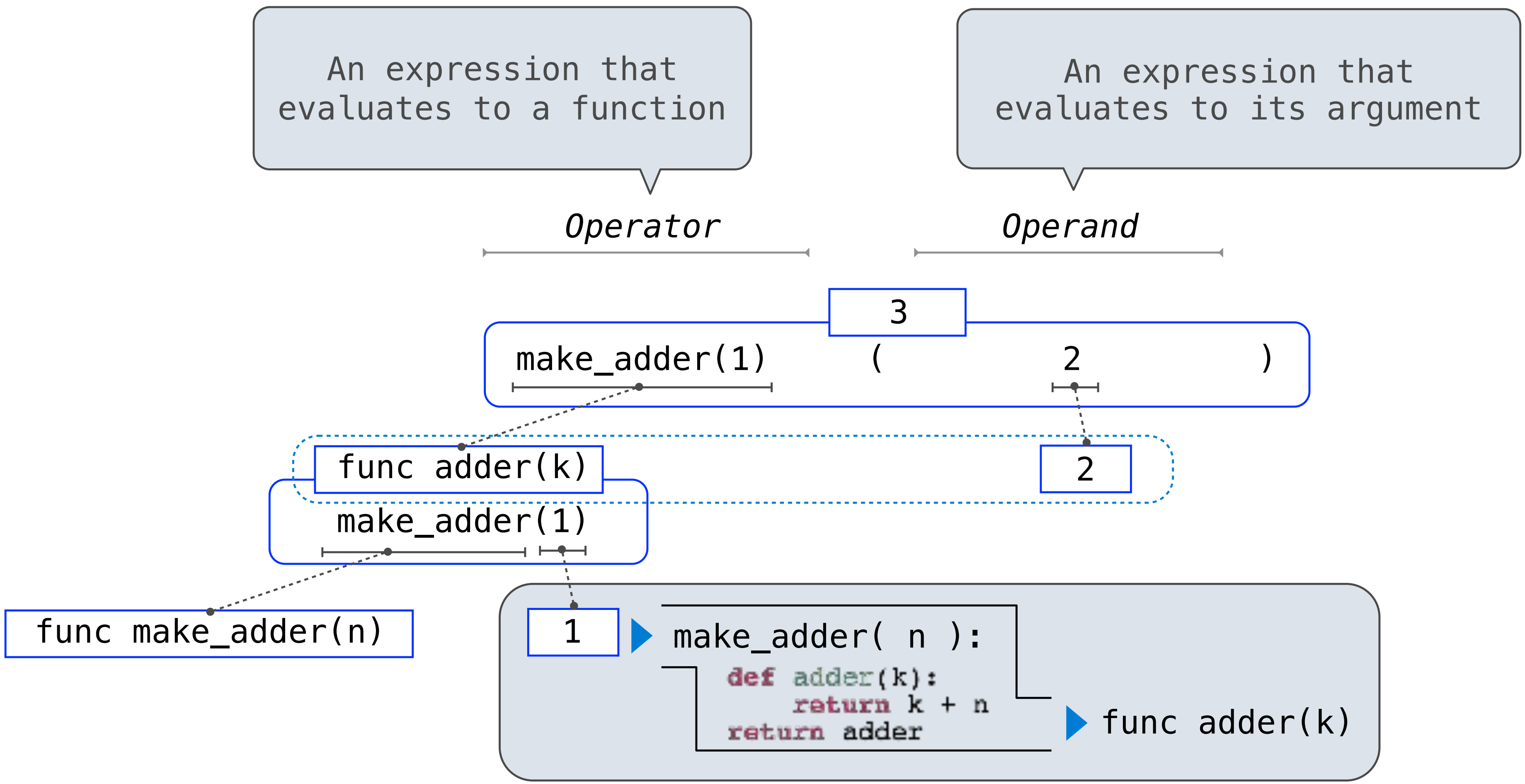
Call Expressions as Operator Expressions



Call Expressions as Operator Expressions



Call Expressions as Operator Expressions



Lecture Overview

- Iteration Example: The Fibonacci Sequence
- Designing Functions
- Generalization
- Higher-Order Functions
- Functions as Return Values
- Lambda Expressions
- Filter, Map, and Reduce Functions

Lambda Expressions

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

An expression: this one evaluates to a number

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

Lambda Expressions

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

An expression: this one evaluates to a number

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

An expression: this one evaluates to a number

```
>>> 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
```

An expression: this one evaluates to a number

```
>>> 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

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

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

Lambda expressions in Python cannot contain statements at all!

Lambda Expressions vs. Def Statements

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

VS

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

Lambda Expressions vs. 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 bind that function to the name square.
- Only the def statement gives the function an intrinsic name, which shows up in environment diagrams but doesn't affect execution (unless the function is printed).

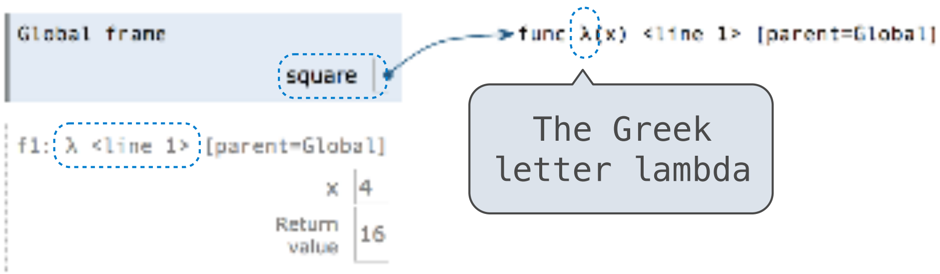
Lambda Expressions vs. Def Statements

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

VS

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

- Both create a function with the same domain, range, and behavior.
- Both bind that function to the name square.
- Only the def statement gives the function an intrinsic name, which shows up in environment diagrams but doesn't affect execution (unless the function is printed).



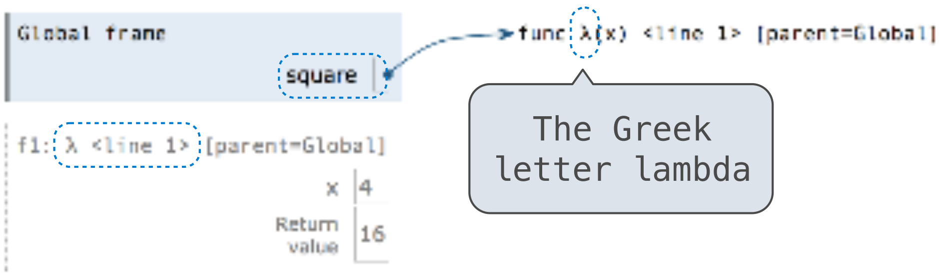
Lambda Expressions vs. Def Statements

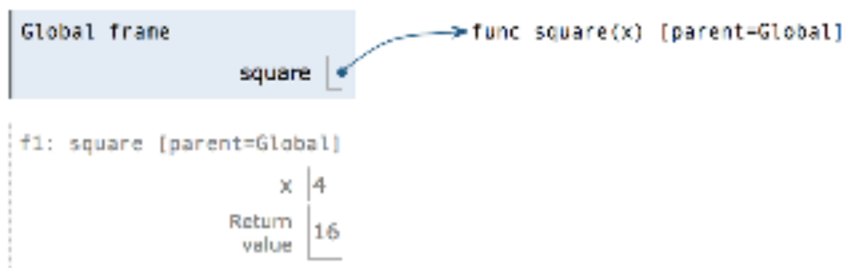
```
square = lambda x: x * x
```

VS

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

- Both create a function with the same domain, range, and behavior.
- Both bind that function to the name square.
- Only the def statement gives the function an intrinsic name, which shows up in environment diagrams but doesn't affect execution (unless the function is printed).





Function Currying

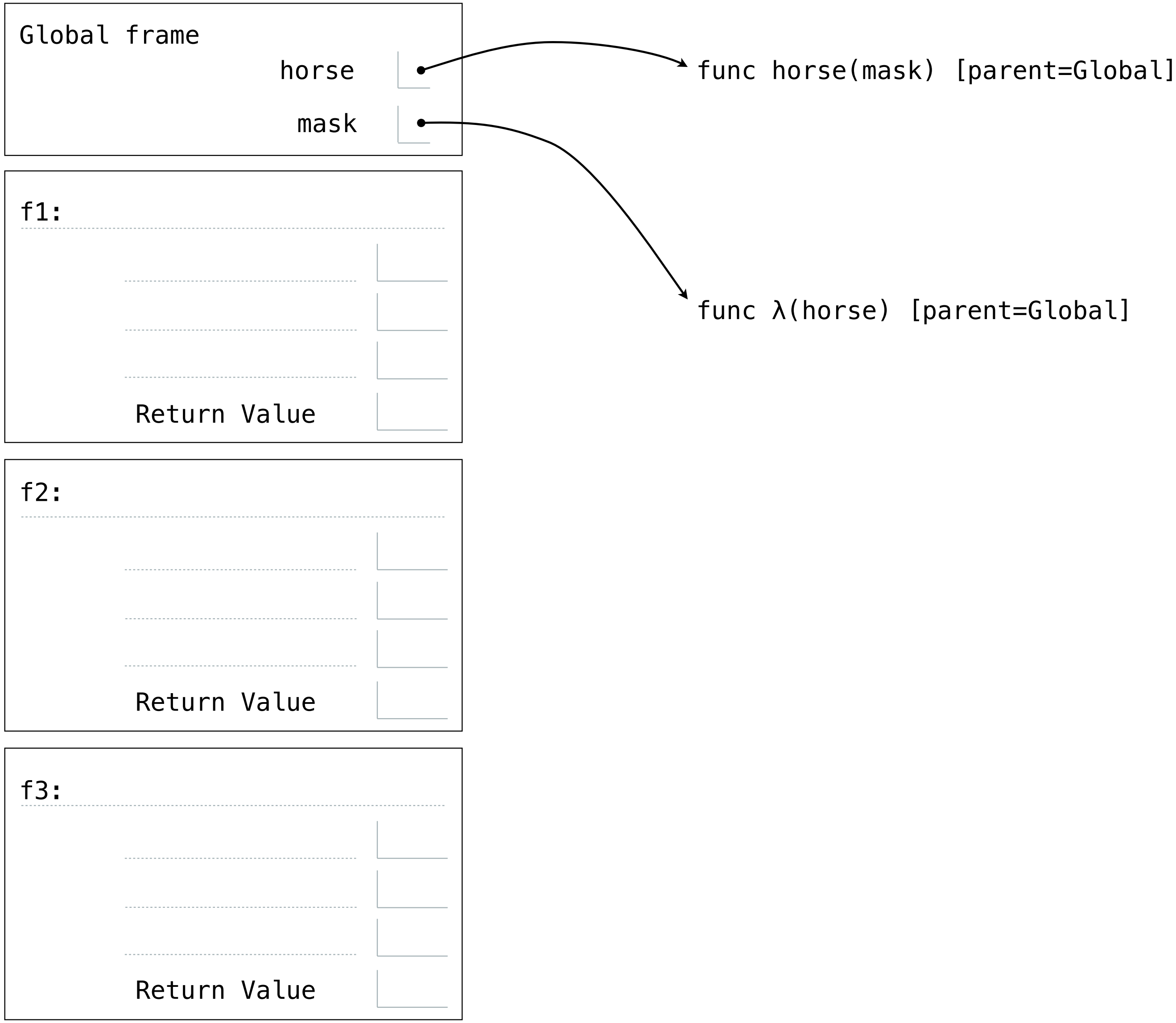
```
def make_adder(n):
    return lambda k: n + k
```

```
>>> make_adder(2)(3)
5
>>> add(2, 3)
5
```

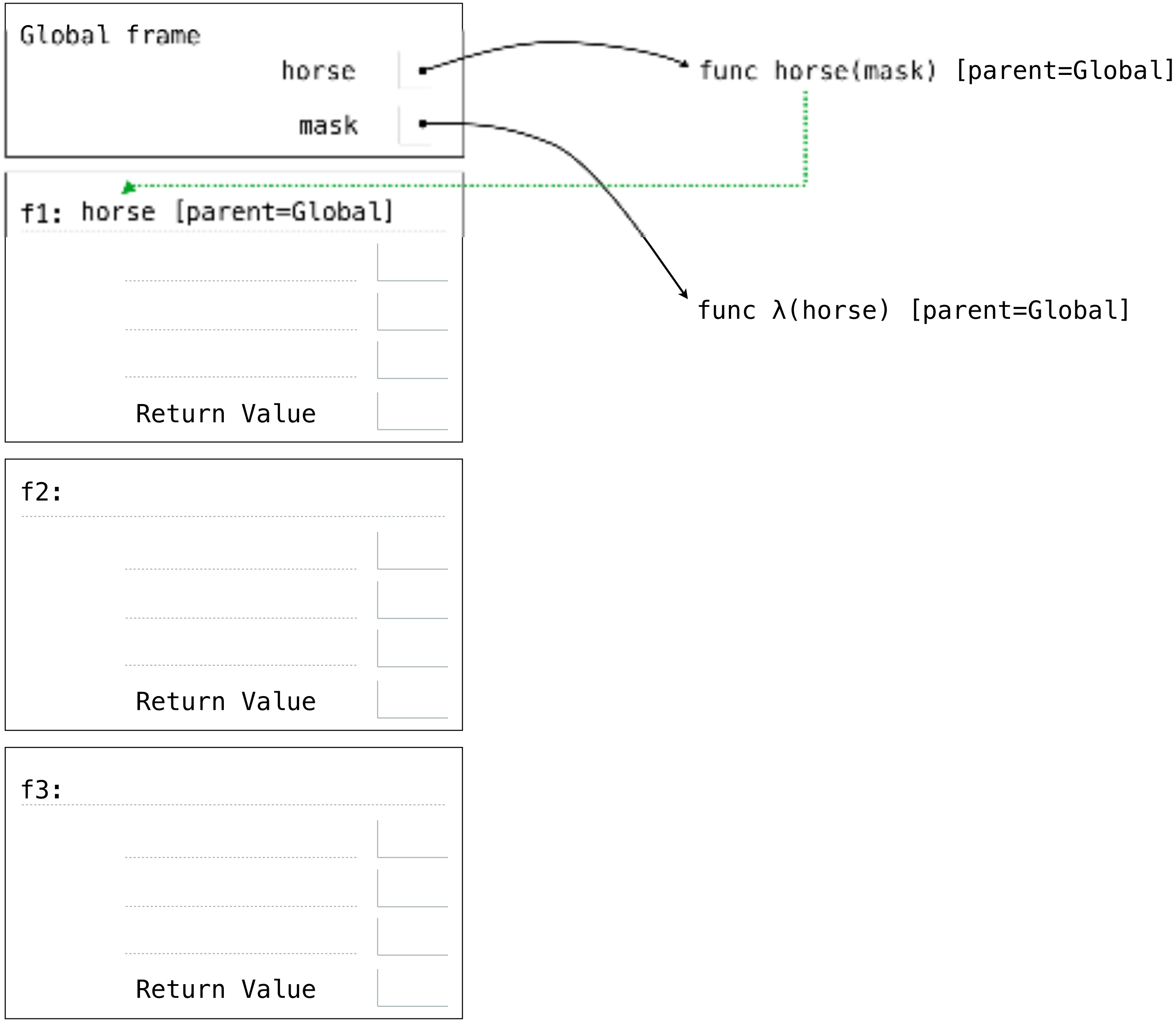
There's a general relationship between these functions

- **Curry:** Transform a multi-argument function into a single-argument, higher-order function

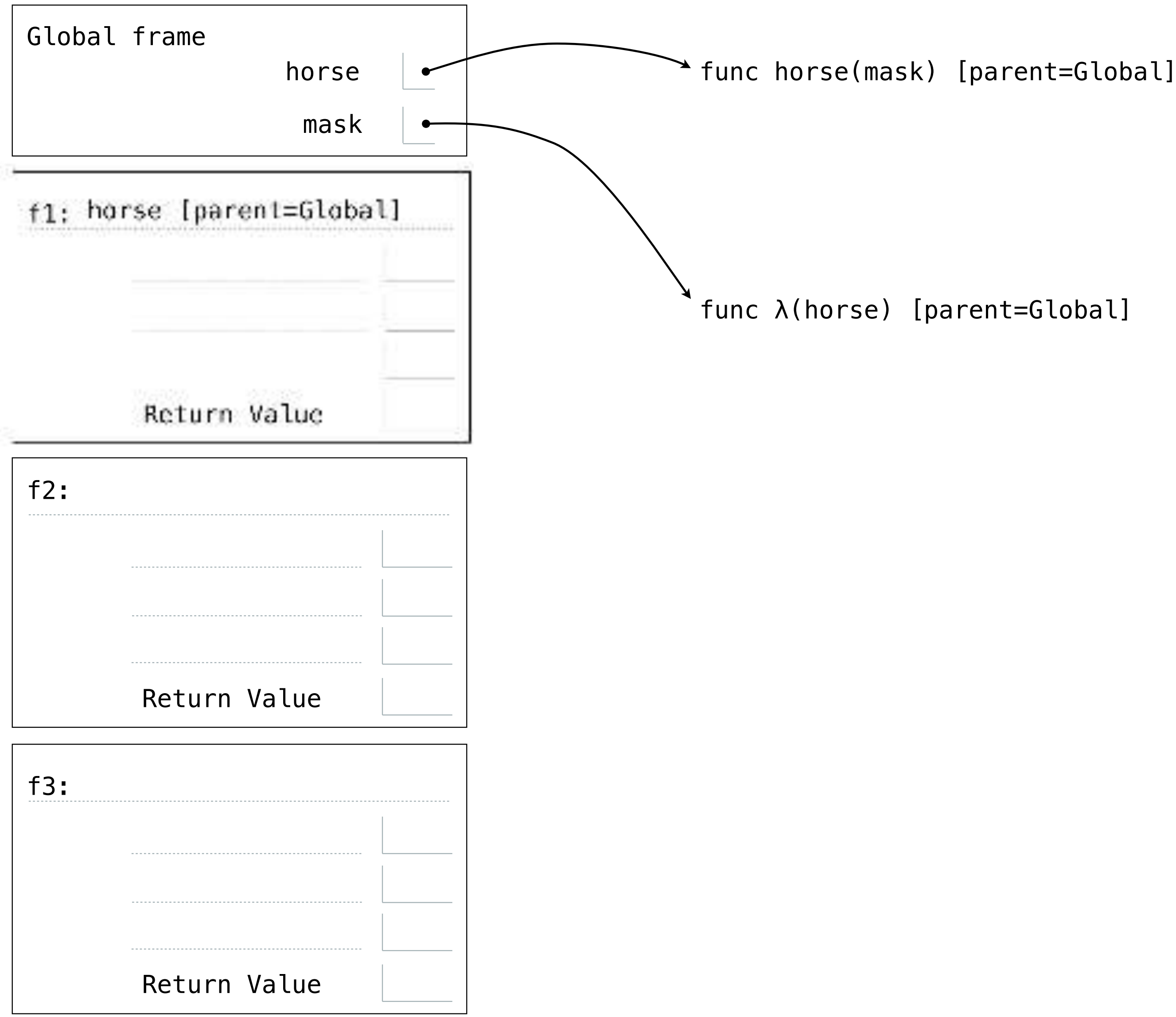
```
def horse(mask):
    horse = mask
    def mask(horse):
        return horse
    return horse(mask)

mask = lambda horse: horse(2)

horse(mask)
```



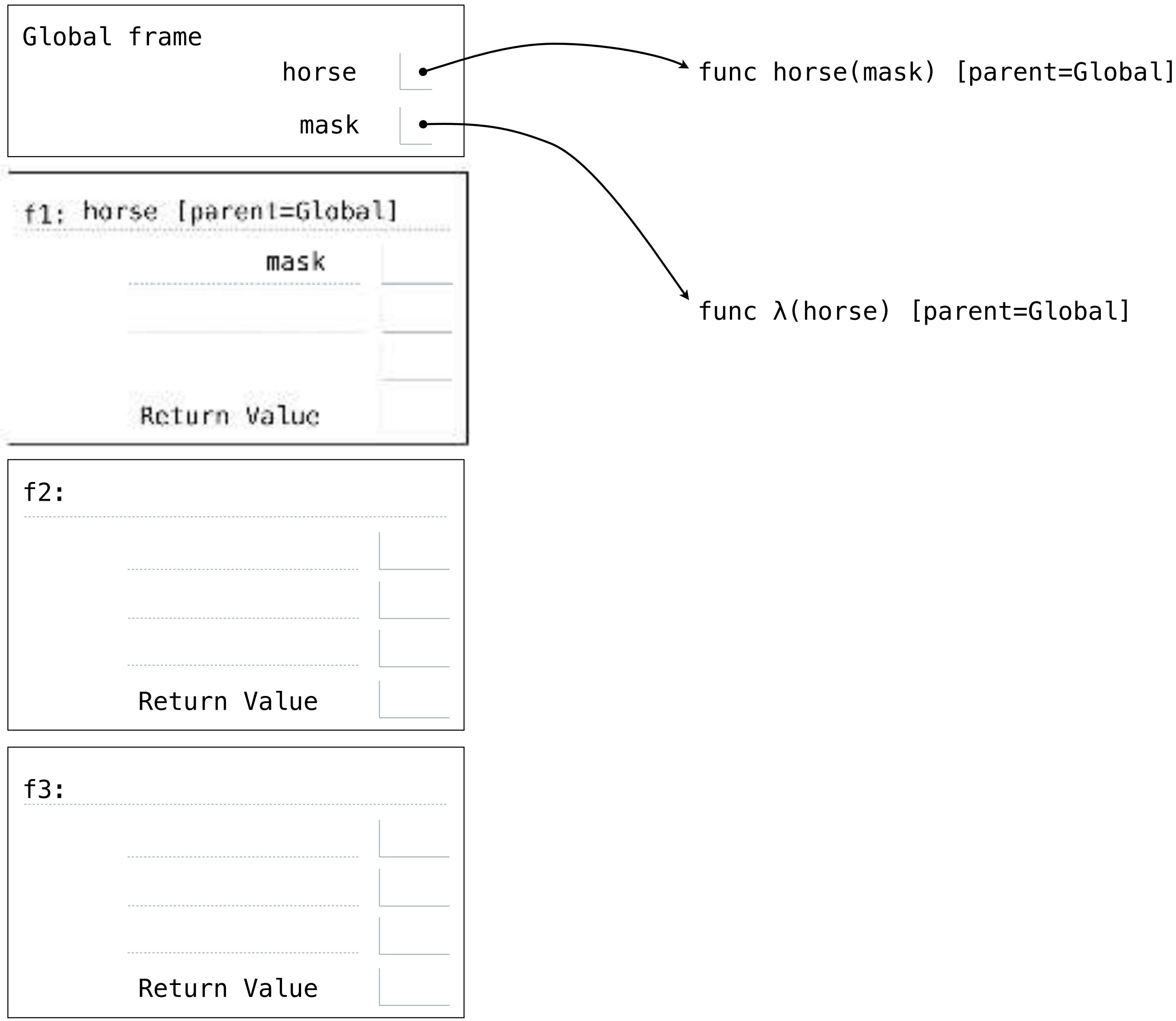
```
def horse(mask):
    horse = mask
    def mask(horse):
        return horse
    return horse(mask)

mask = lambda horse: horse(2)

horse(mask)
```



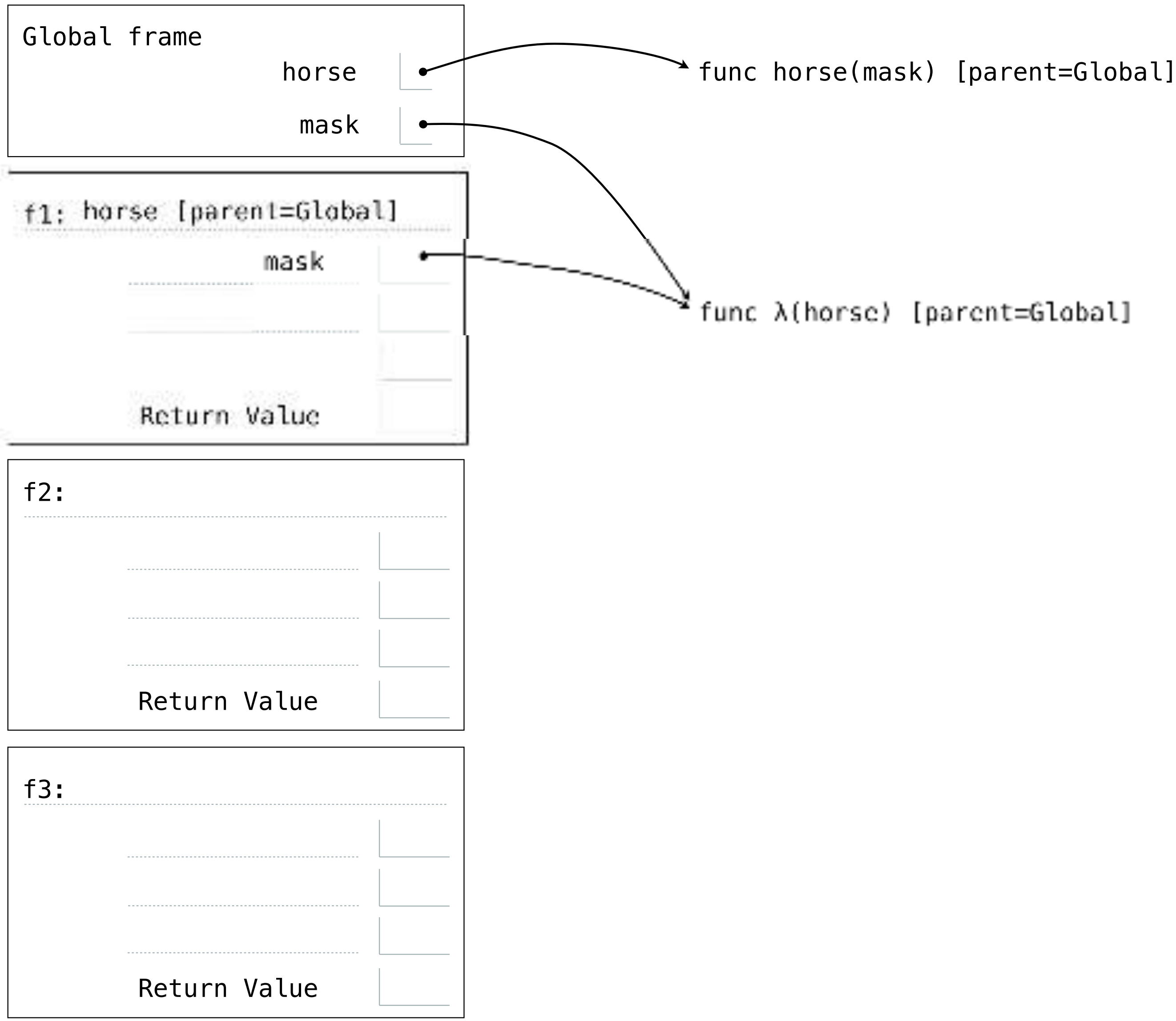
```
def horse(mask):
    horse = mask
    def mask(horse):
        return horse
    return horse(mask)

mask = lambda horse: horse(2)

horse(mask)
```



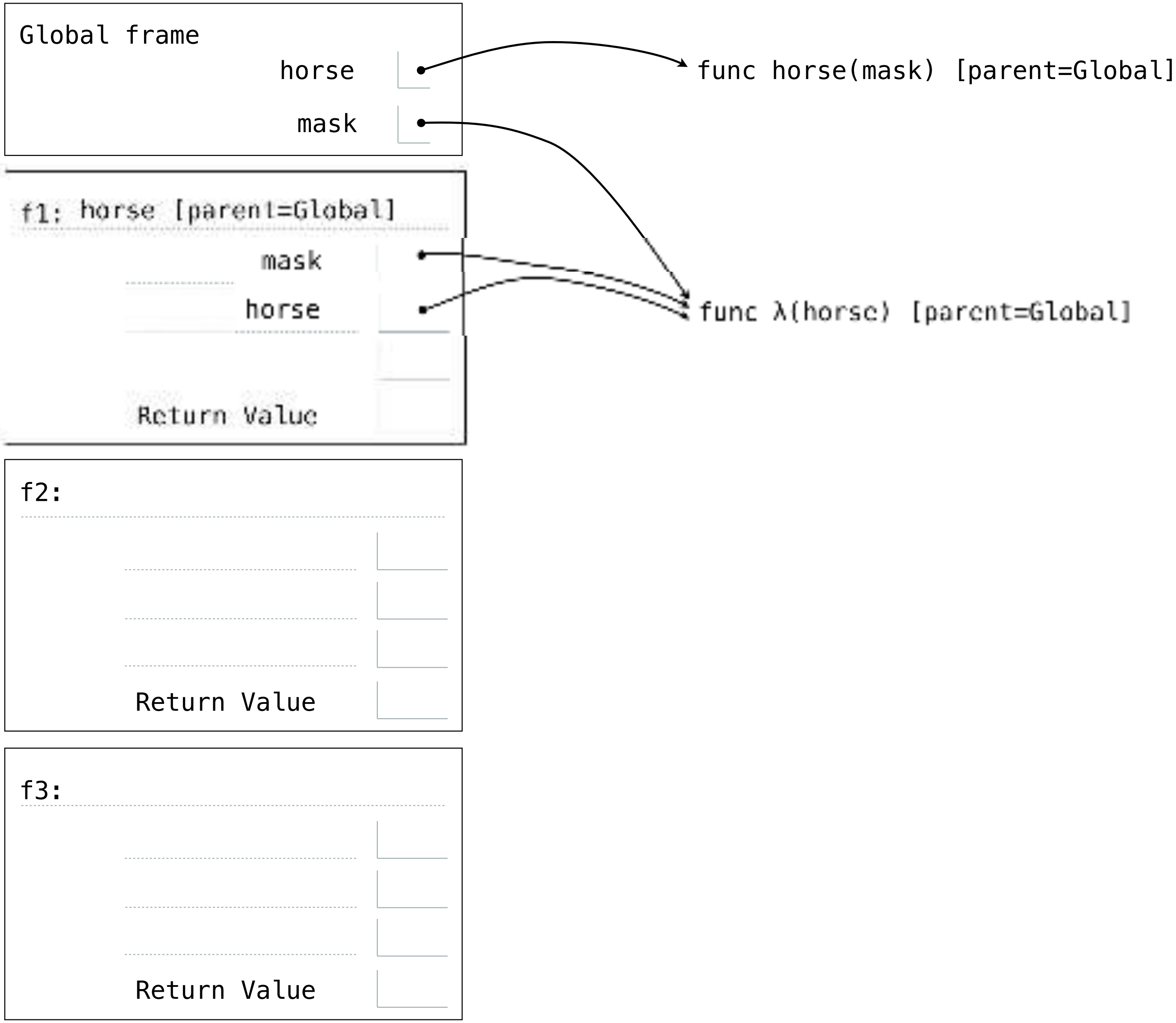
```
def horse(mask):
    horse = mask
    def mask(horse):
        return horse
    return horse(mask)

mask = lambda horse: horse(2)

horse(mask)
```



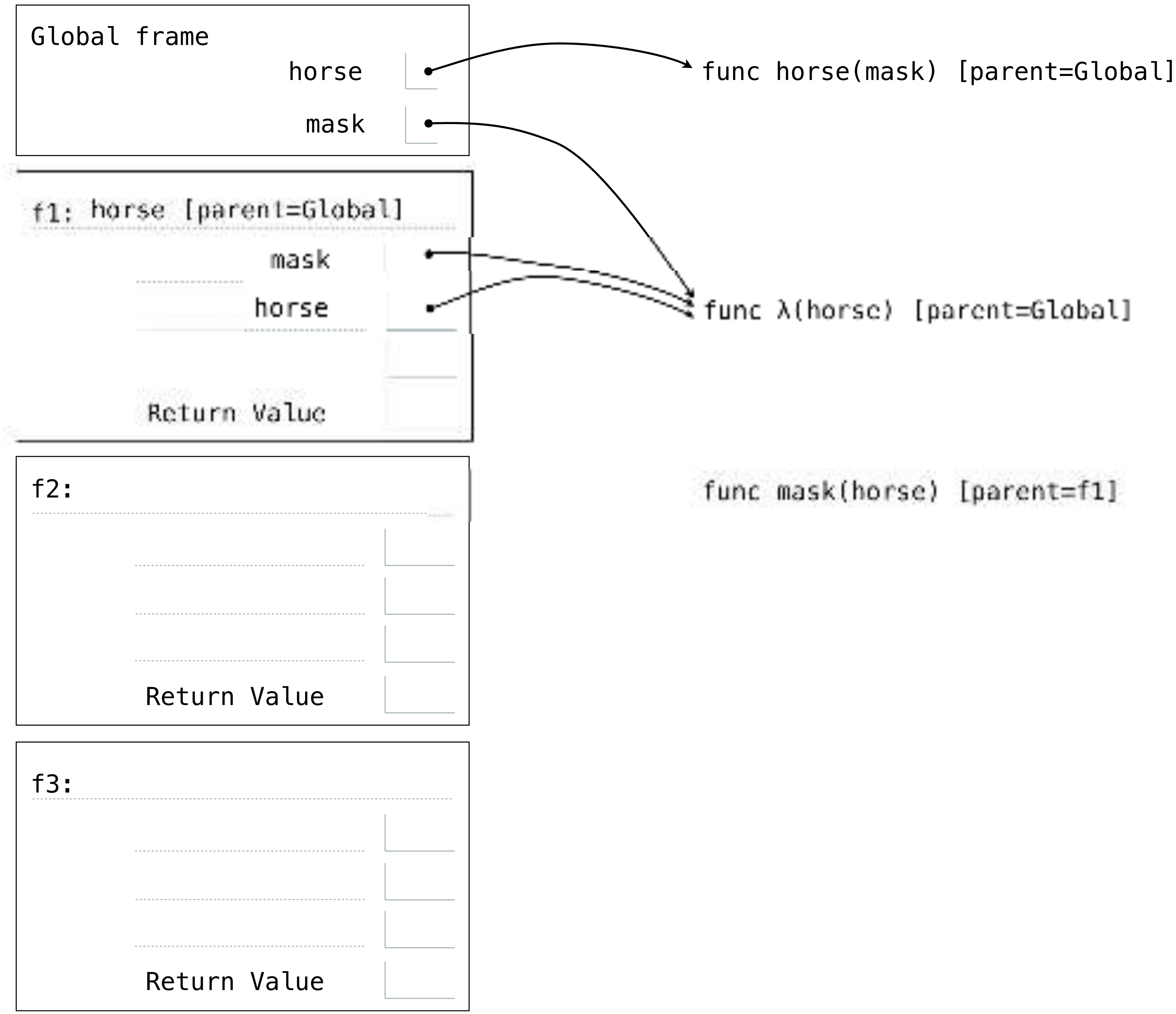
```
def horse(mask):
    horse = mask
    def mask(horse):
        return horse
    return horse(mask)

mask = lambda horse: horse(2)

horse(mask)
```



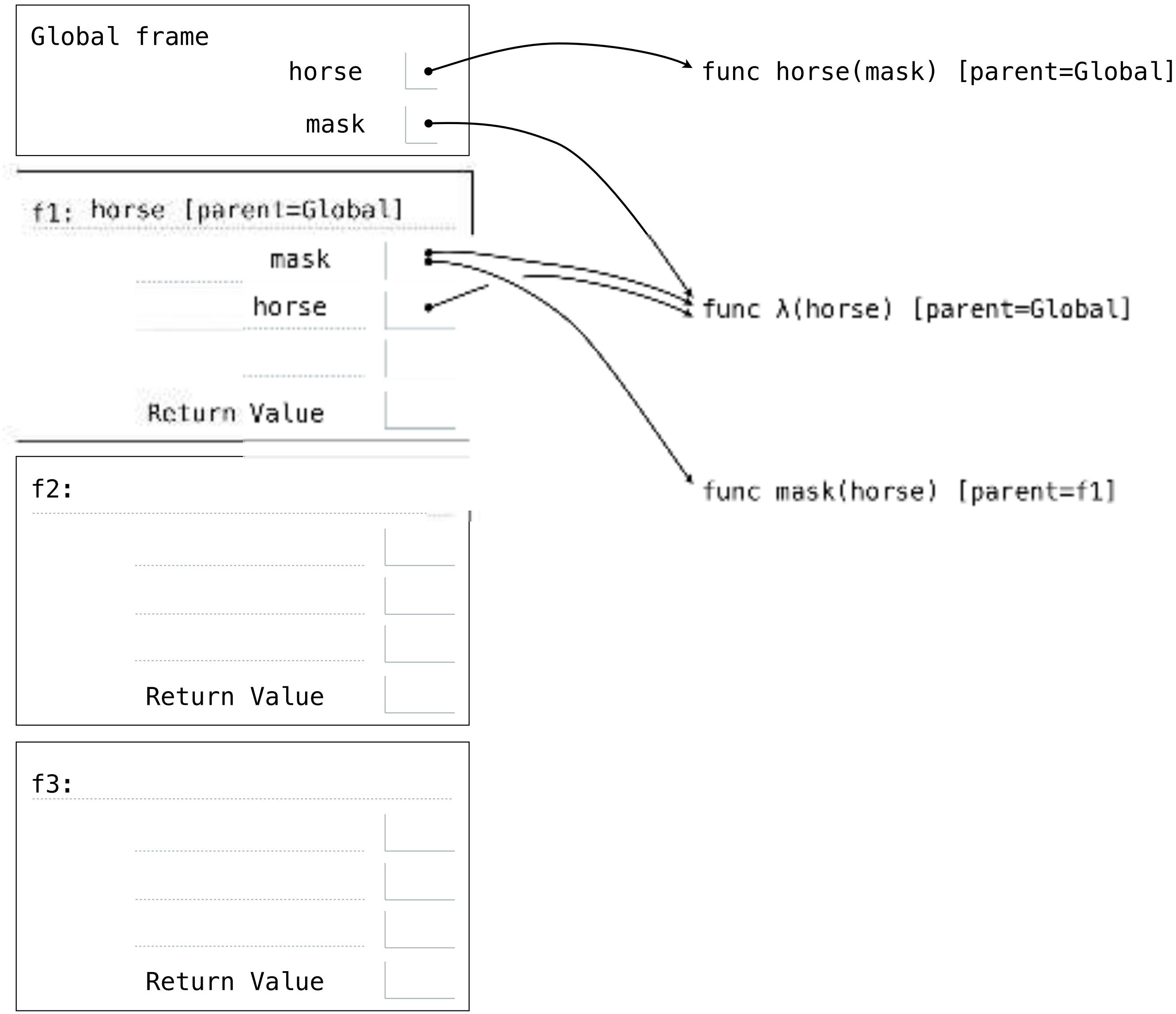
```
def horse(mask):
    horse = mask
    def mask(horse):
        return horse
    return horse(mask)

mask = lambda horse: horse(2)

horse(mask)
```



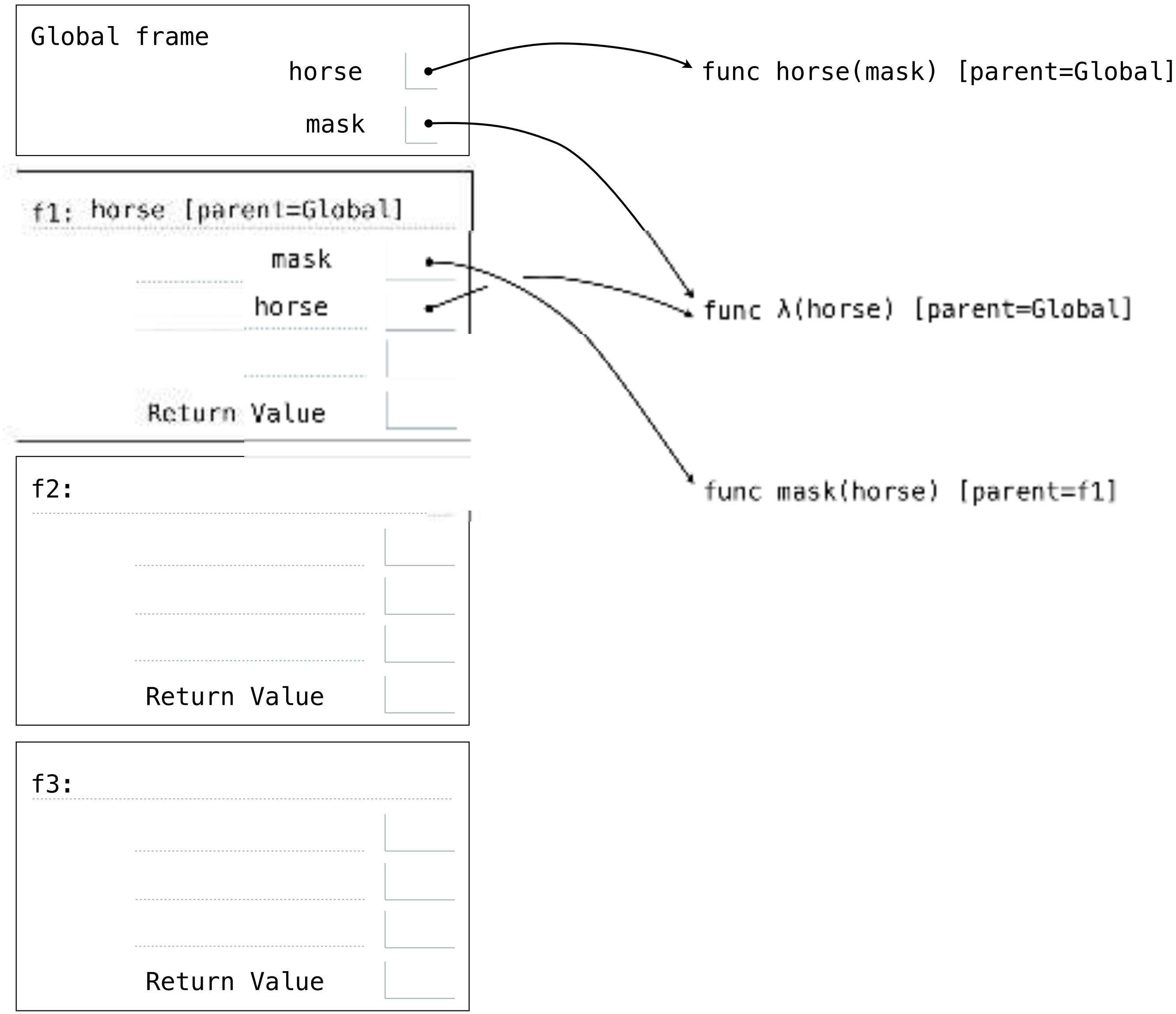
```
def horse(mask):
    horse = mask
    def mask(horse):
        return horse
    return horse(mask)

mask = lambda horse: horse(2)

horse(mask)
```



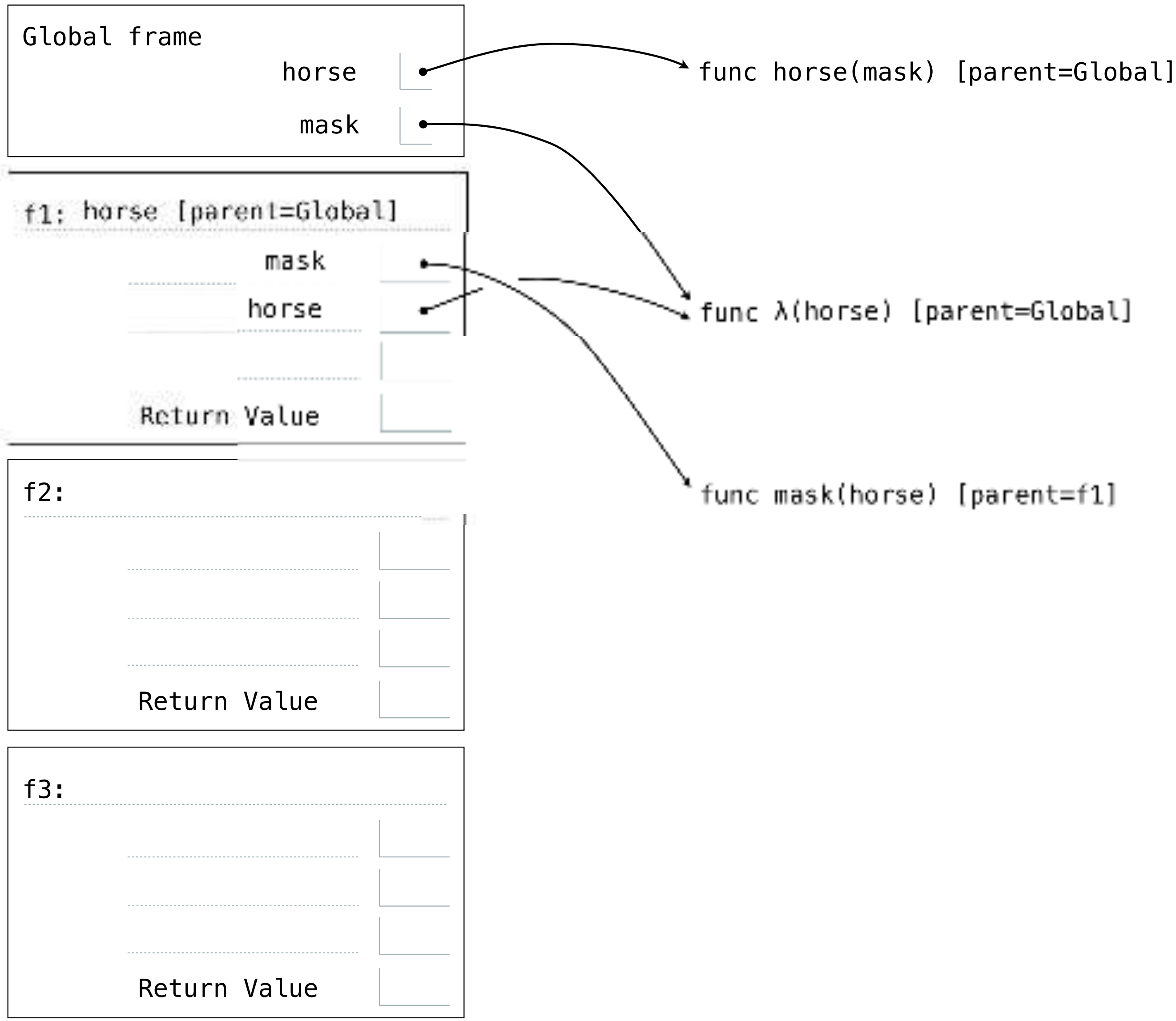
```
def horse(mask):
    horse = mask
    def mask(horse):
        return horse
    return horse(mask)

mask = lambda horse: horse(2)

horse(mask)
```



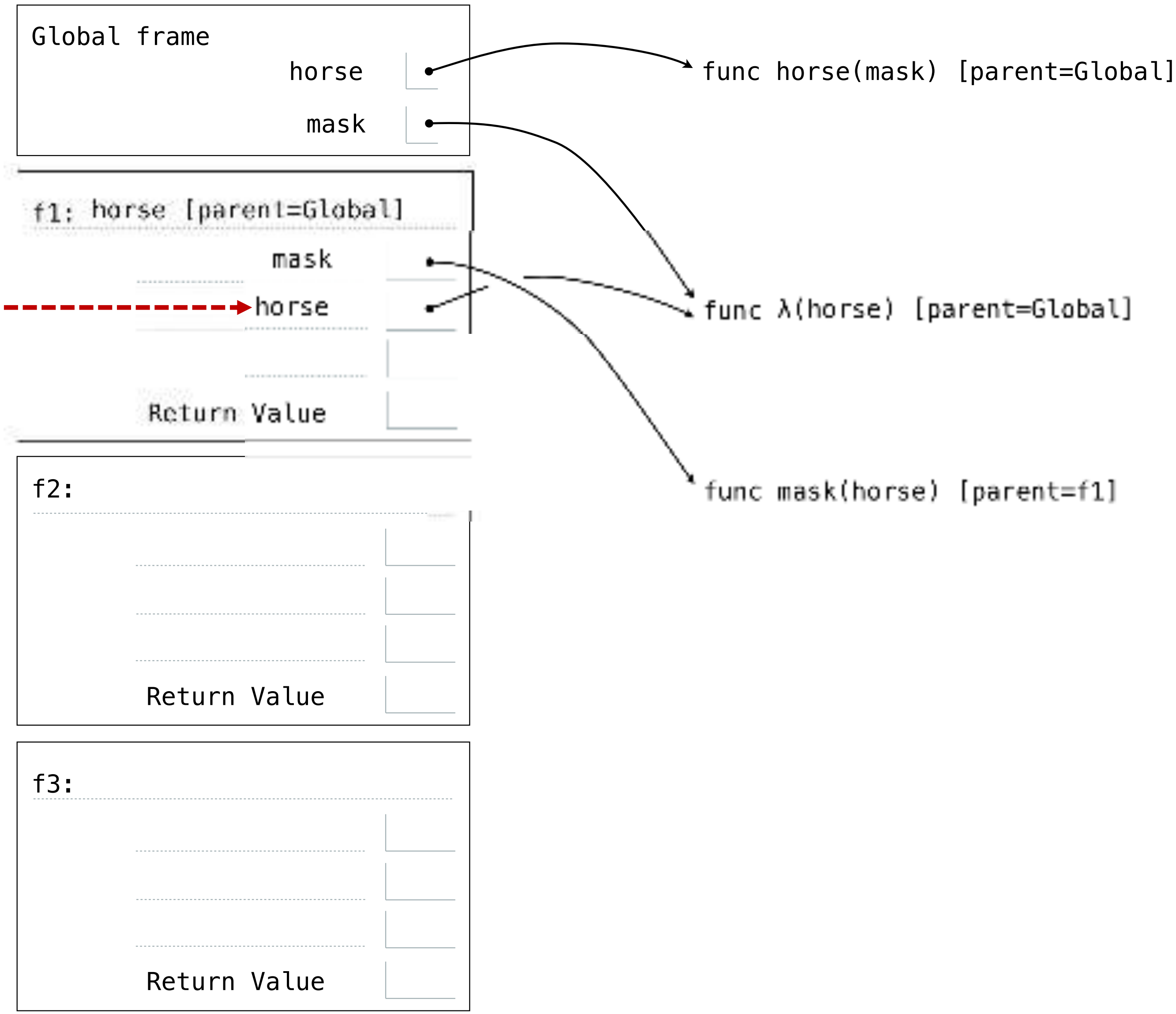
```
def horse(mask):
    horse = mask
    def mask(horse):
        return horse
    return horse(mask)

mask = lambda horse: horse(2)

horse(mask)
```



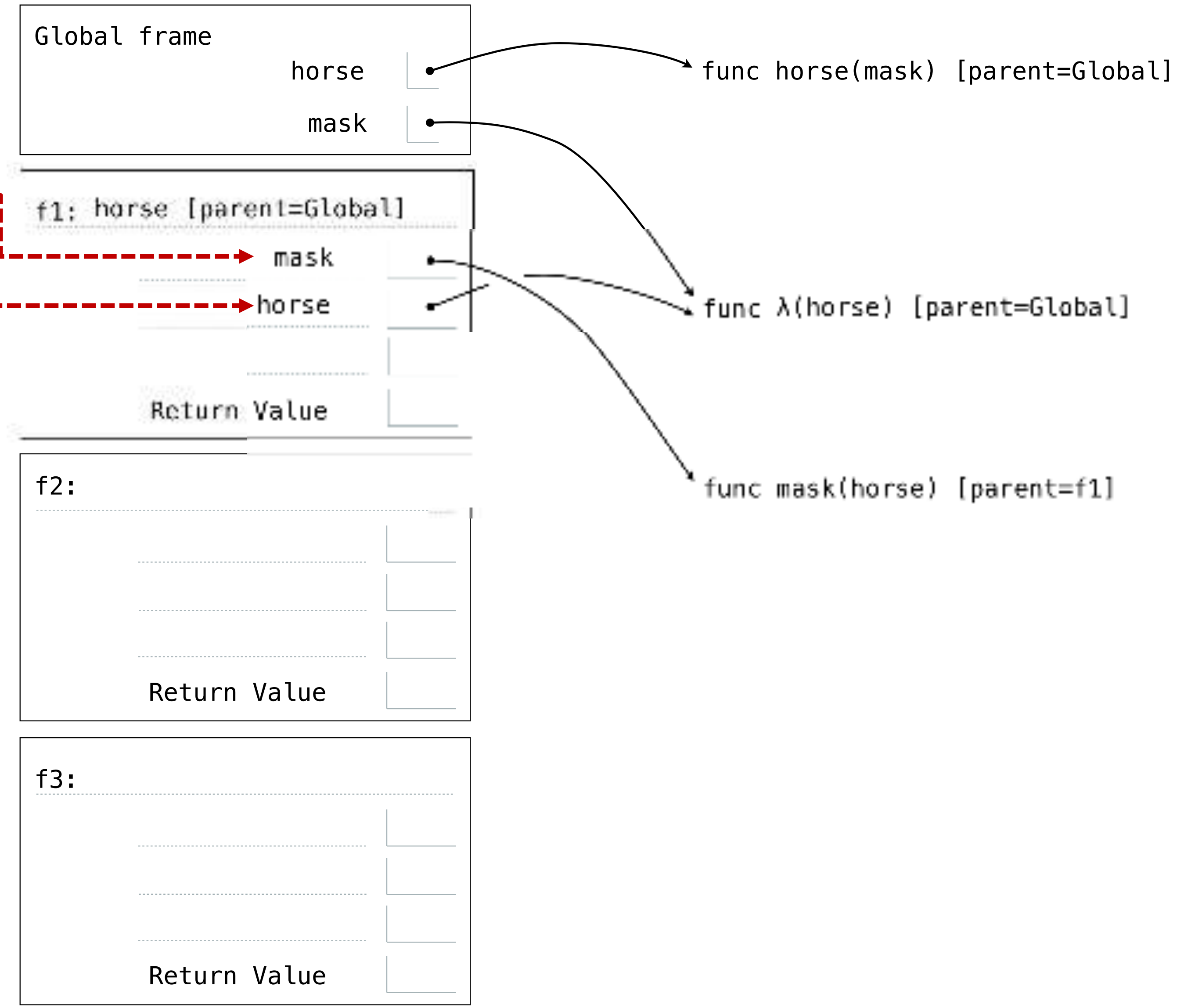
```
def horse(mask):
    horse = mask
    def mask(horse):
        return horse
    return horse(mask)

mask = lambda horse: horse(2)

horse(mask)
```



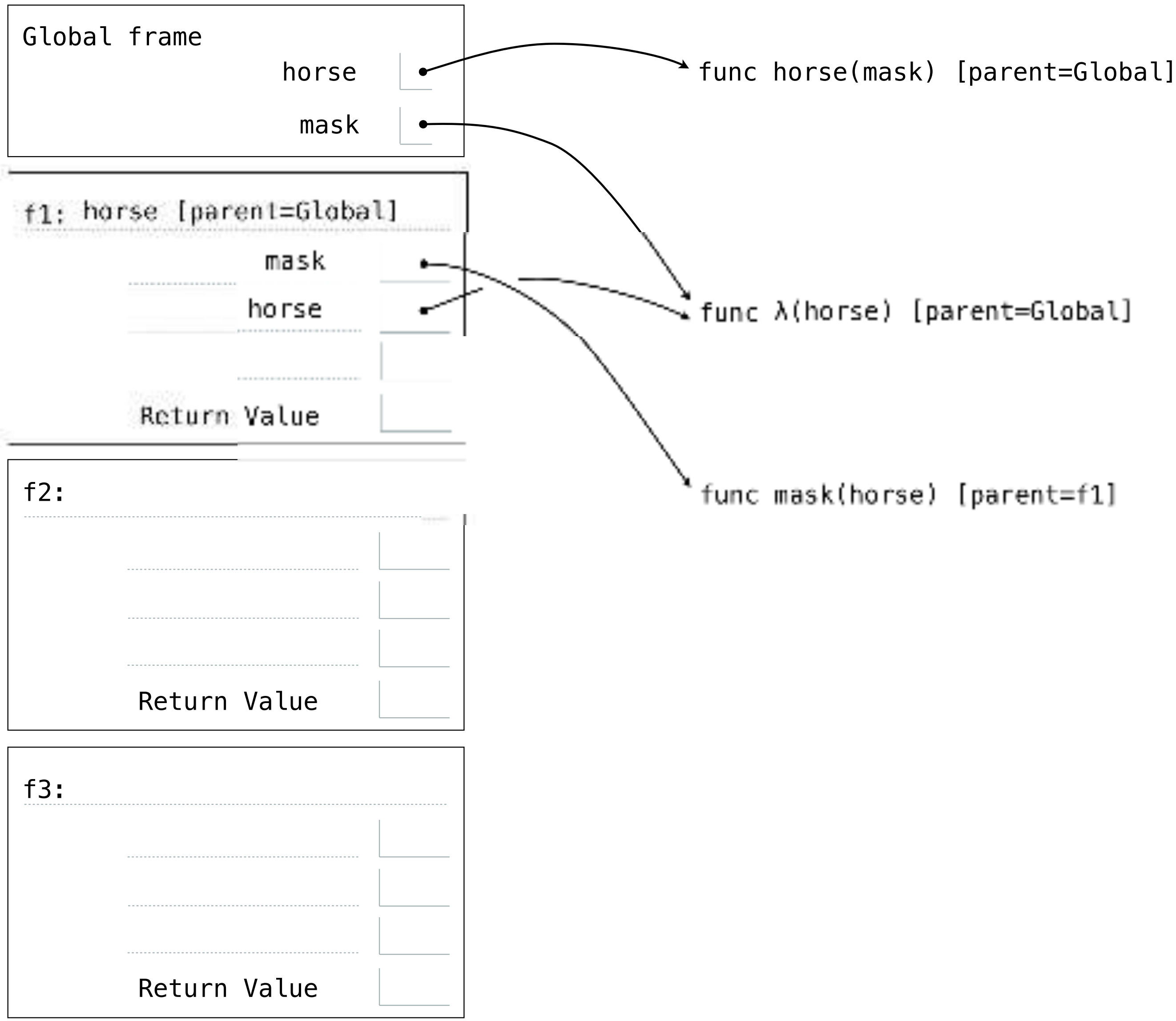
```
def horse(mask):
    horse = mask
    def mask(horse):
        return horse
    return horse(mask)

mask = lambda horse: horse(2)

horse(mask)
```



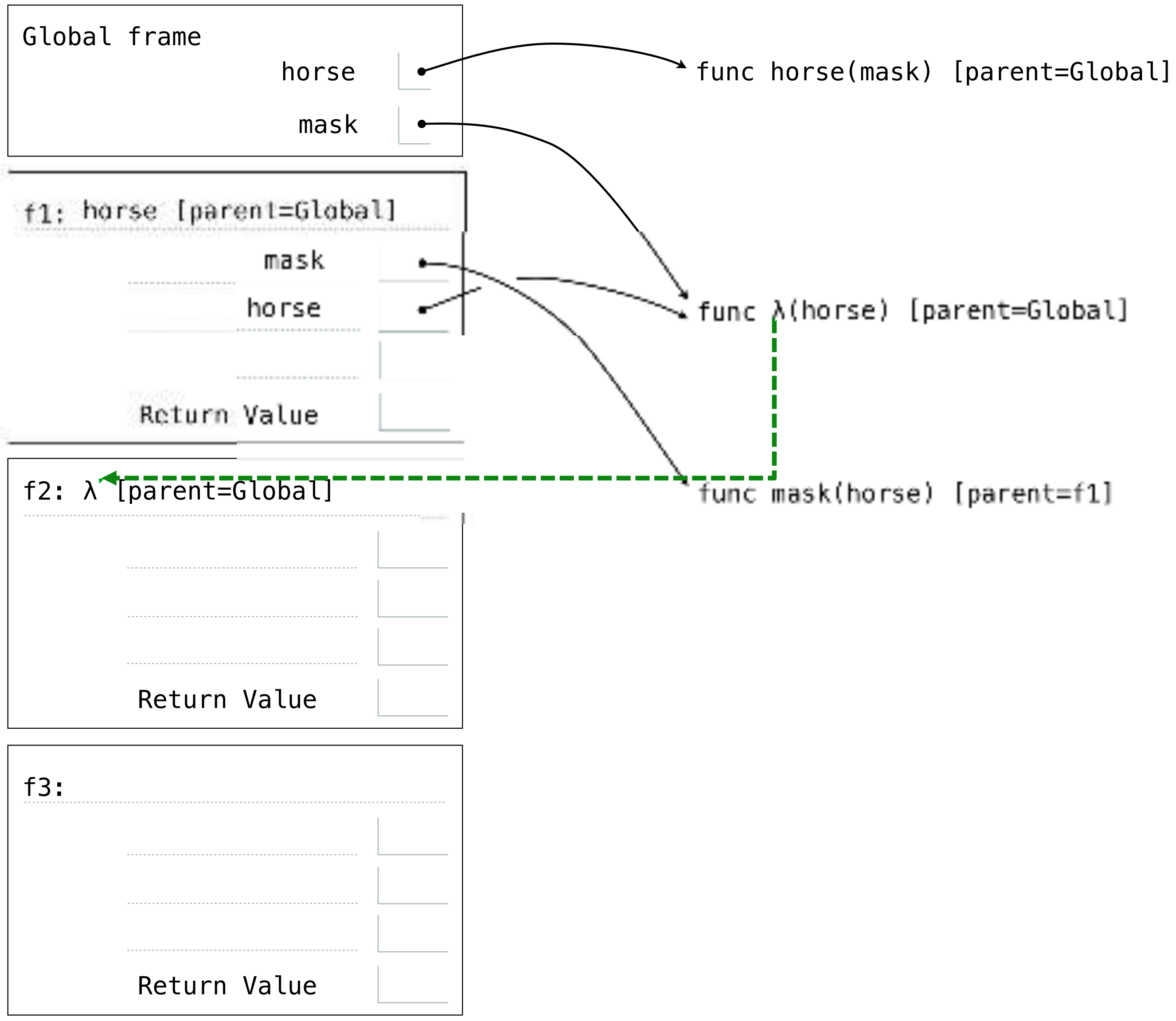
```
def horse(mask):
    horse = mask
    def mask(horse):
        return horse
    return horse(mask)

mask = lambda horse: horse(2)

horse(mask)
```



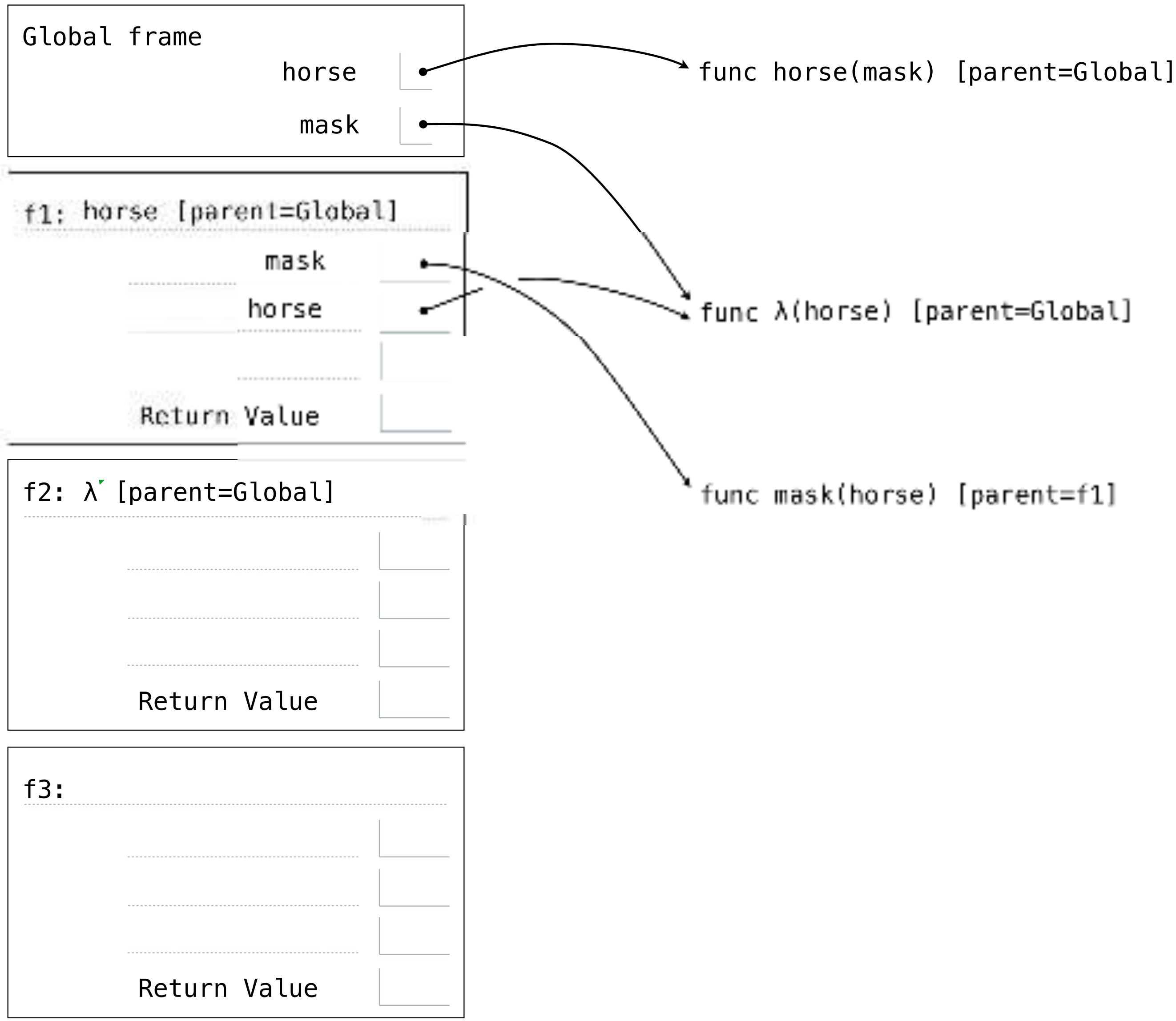
```
def horse(mask):
    horse = mask
    def mask(horse):
        return horse
    return horse(mask)

mask = lambda horse: horse(2)

horse(mask)
```



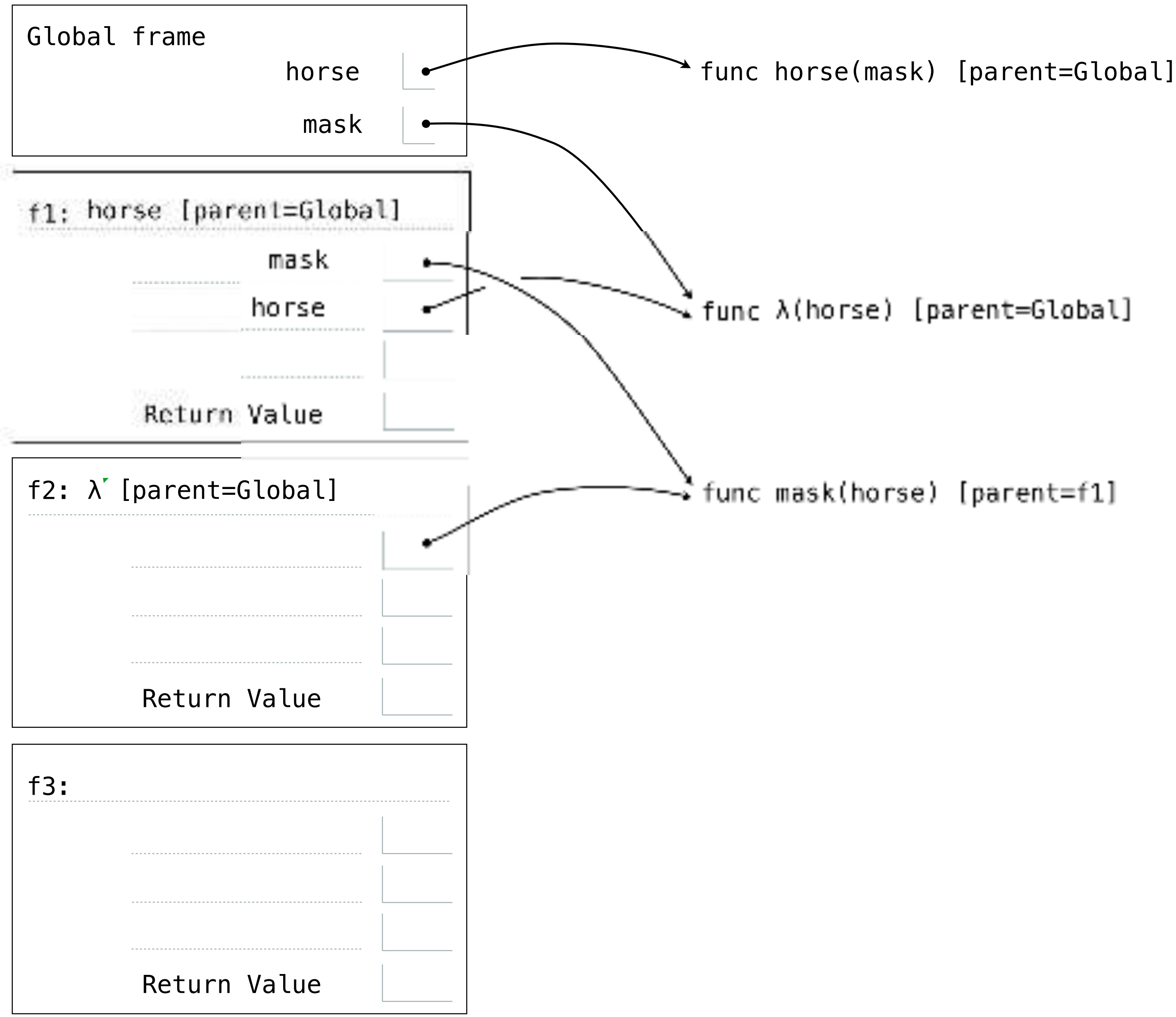
```
def horse(mask):
    horse = mask
    def mask(horse):
        return horse
    return horse(mask)

mask = lambda horse: horse(2)

horse(mask)
```



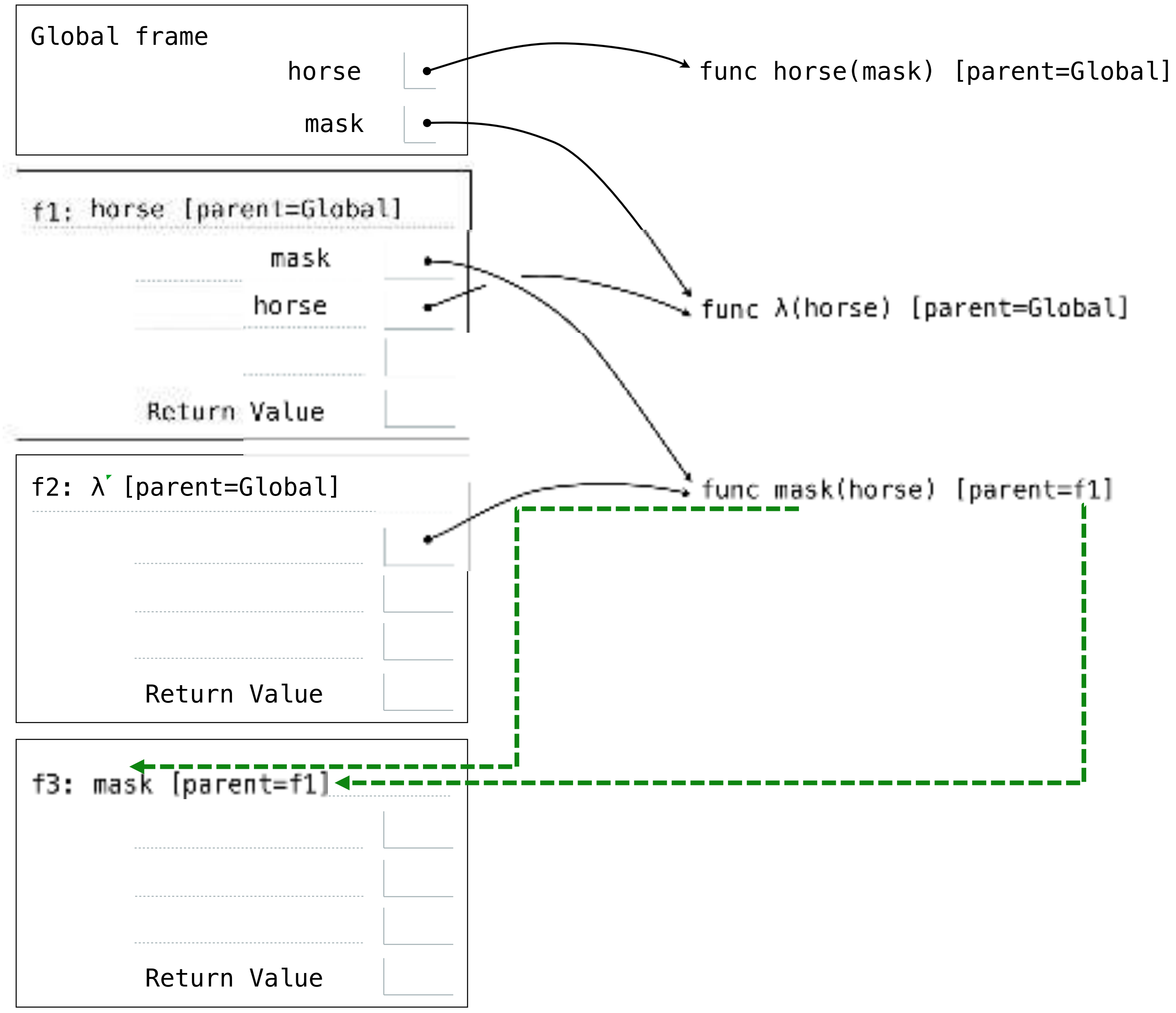
```
def horse(mask):
    horse = mask
    def mask(horse):
        return horse
    return horse(mask)

mask = lambda horse: horse(2)

horse(mask)
```



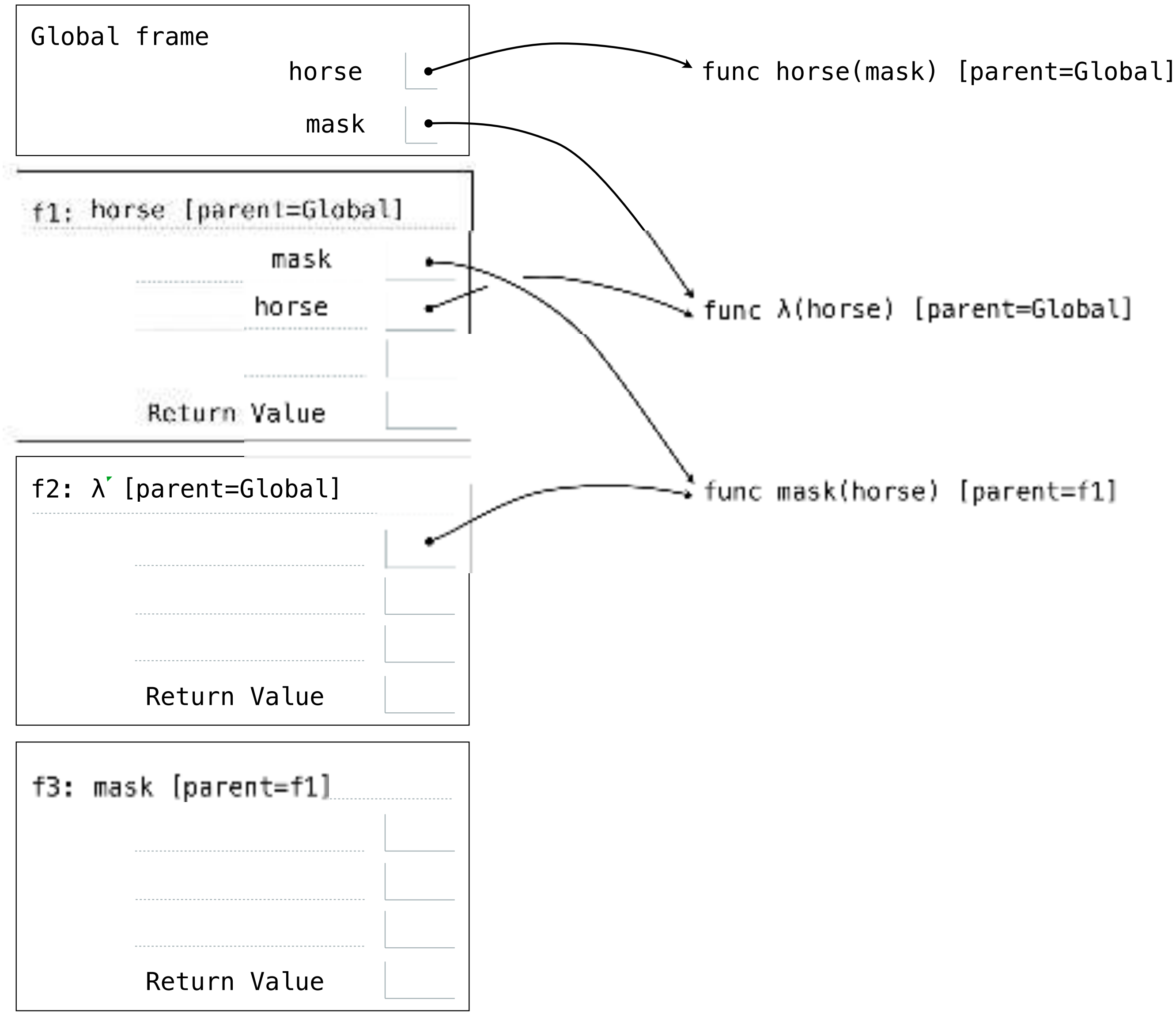
```
def horse(mask):
    horse = mask
    def mask(horse):
        return horse
    return horse(mask)

mask = lambda horse: horse(2)

horse(mask)
```



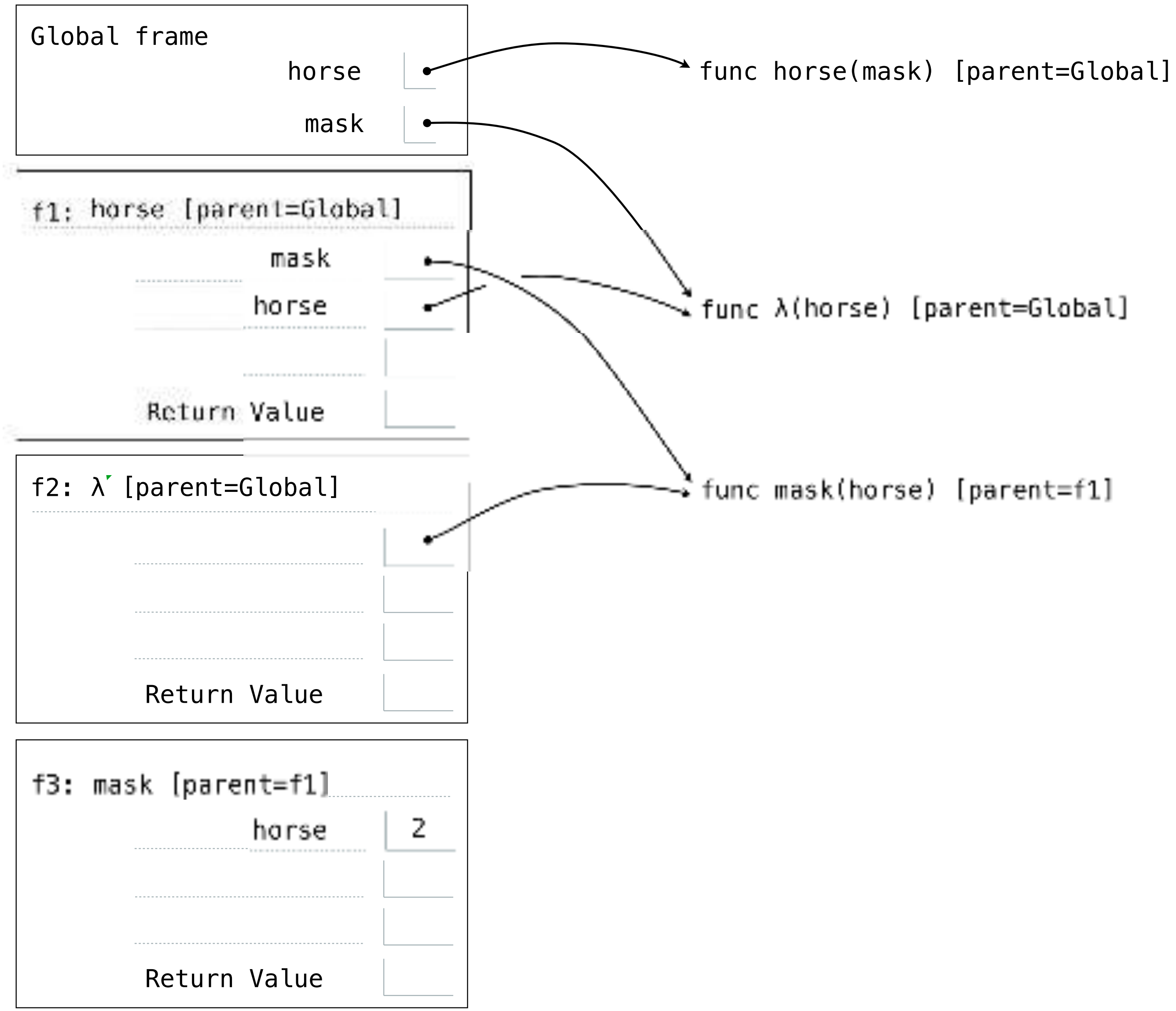
```
def horse(mask):
    horse = mask
    def mask(horse):
        return horse
    return horse(mask)

mask = lambda horse: horse(2)

horse(mask)
```



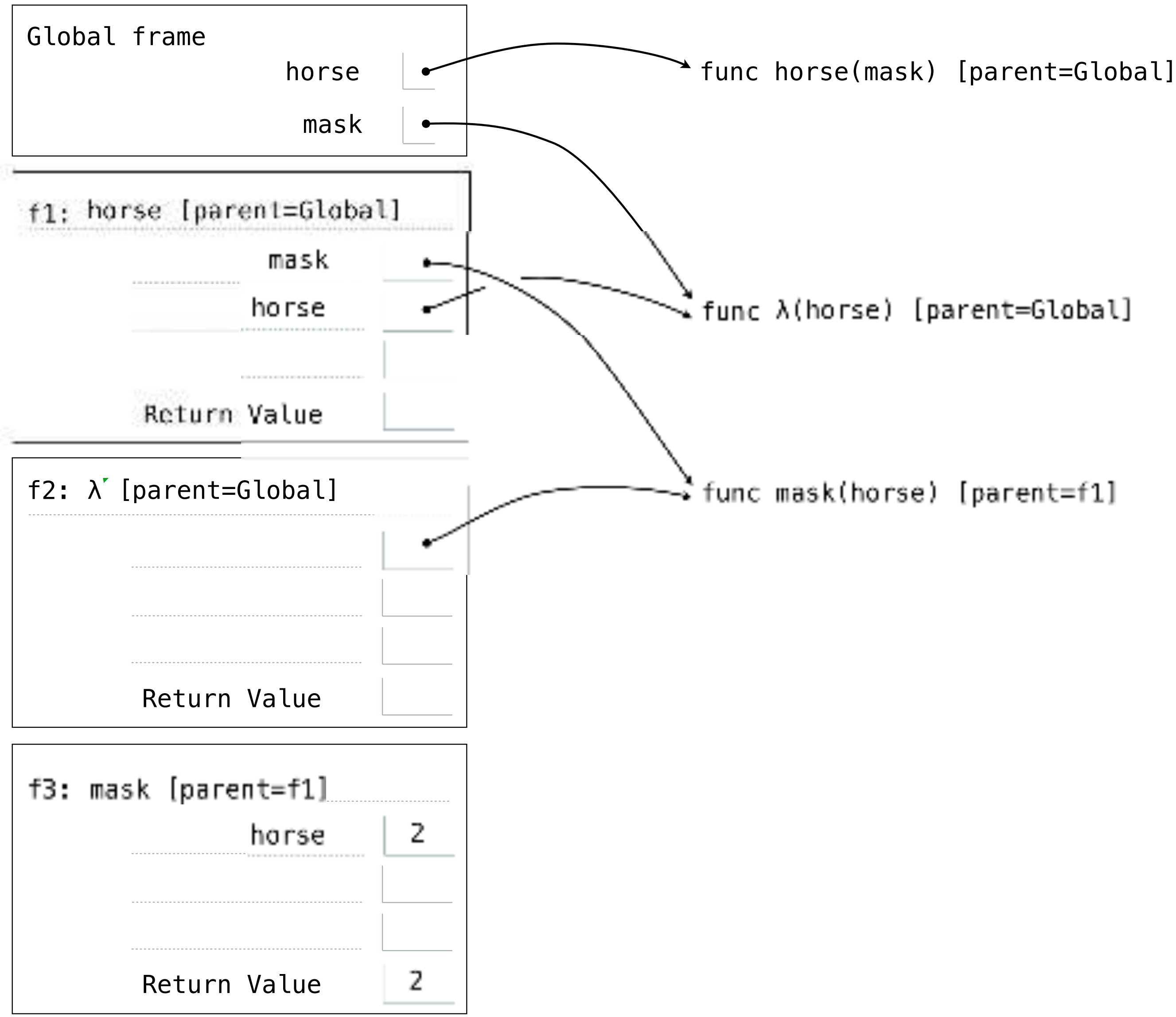
```
def horse(mask):
    horse = mask
    def mask(horse):
        return horse
    return horse(mask)

mask = lambda horse: horse(2)

horse(mask)
```



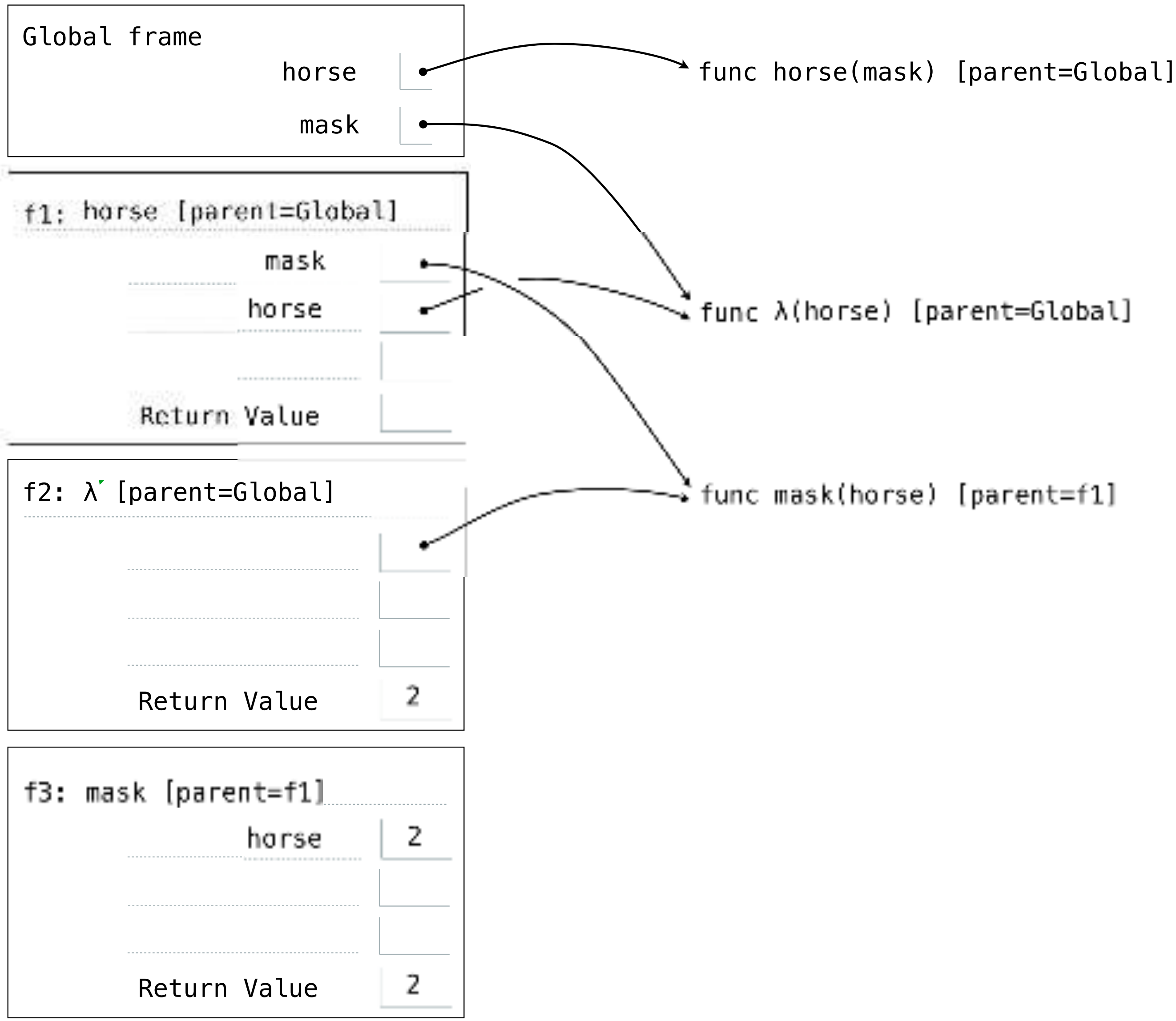
```
def horse(mask):
    horse = mask
    def mask(horse):
        return horse
    return horse(mask)

mask = lambda horse: horse(2)

horse(mask)
```



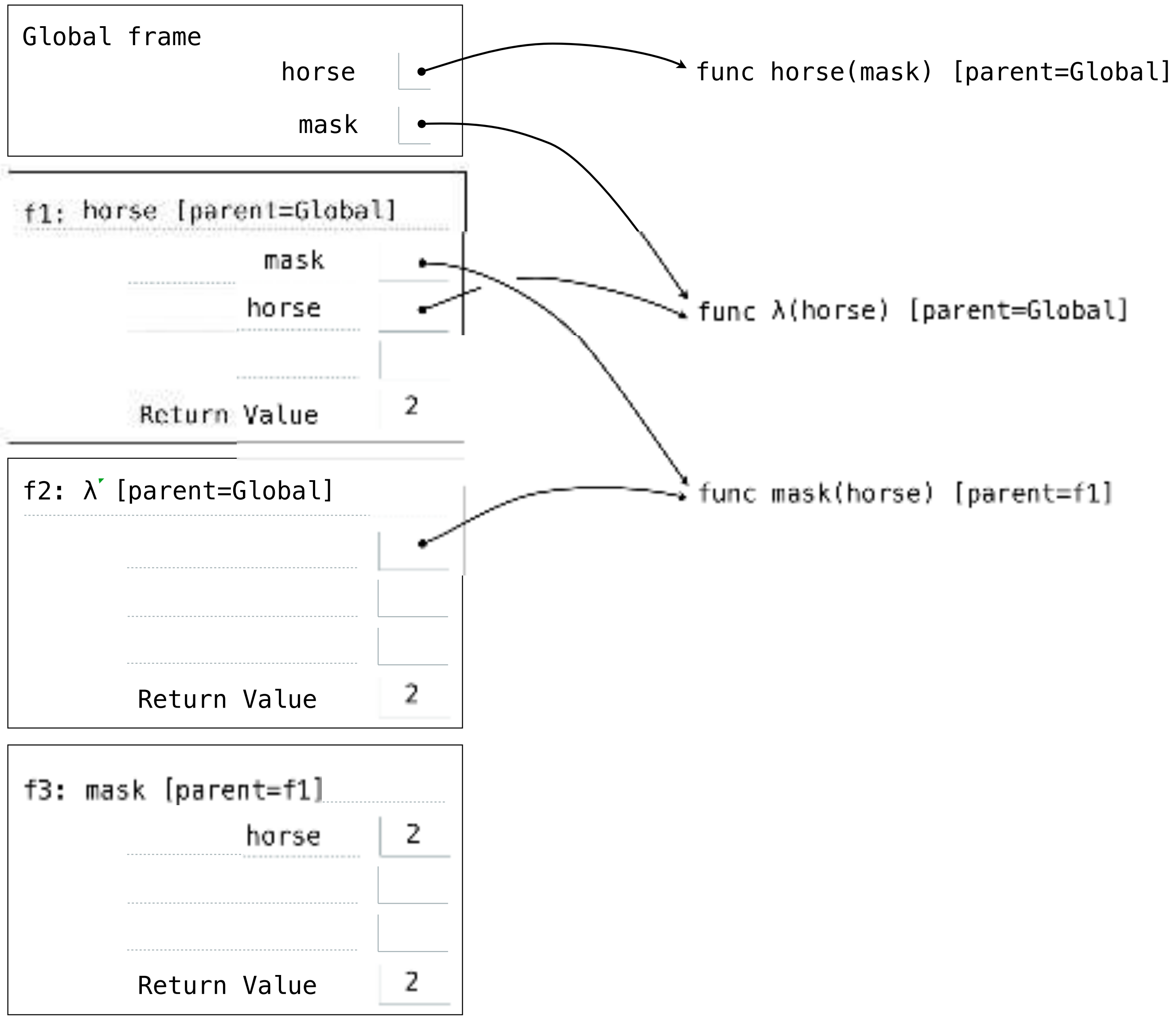
```
def horse(mask):
    horse = mask
    def mask(horse):
        return horse
    return horse(mask)

mask = lambda horse: horse(2)

horse(mask)
```



```
def horse(mask):
    horse = mask
    def mask(horse):
        return horse
    return horse(mask)

mask = lambda horse: horse(2)

horse(mask)
```



```
def horse(mask):
    horse = mask
    def mask(horse):
        return horse
    return horse(mask)

mask = lambda horse: horse(2)

horse(mask)
```



Lecture Overview

- Iteration Example: The Fibonacci Sequence
- Designing Functions
- Generalization
- Higher-Order Functions
- Functions as Return Values
- Lambda Expressions
- Filter, Map, and Reduce Functions

Filter Functions

- The built-in function `filter(f, seq)` returns those items of the sequence `seq` for which `f(item)` is `True`

```
>>> primes = filter(is_prime, range(11))
>>> primes
[2, 3, 5, 7]
```

- A lambda function is a “disposable” function that we can define just when we need it and then immediately throw it away after we are done using it

```
>>> odds = filter(lambda x : x % 2 != 0, range(11))
>>> odds
[1, 3, 5, 7, 9]
```

Map and Reduce Functions

- The built-in function `map(f, seq)` returns a list of the results of applying the function `f` to the items of the sequence `seq`

```
squares = map(lambda x : x ** 2, range(11))
>>> squares
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

- The function `functools.reduce(f, seq)` applies a function `f` of two arguments cumulatively to the items of a sequence `seq`, from left to right, so as to reduce the sequence to a single value

```
>>> total = functools.reduce(lambda x, y: x+y, range(11))
>>> total
55
```

Example: RSA Cryptosystem

- The RSA cryptosystem is the most widely-used public key cryptography algorithm in the world, which can be used to encrypt a message without the need to exchange a secret key separately
- A message x is encrypted using the function $f(x) = x^e \bmod n$, where $n = pq$ for two different large primes p and q chosen at random, and e is a random prime number less than $m = (p-1)(q-1)$ such that e does not divide m
- The maximum number that can be encrypted is $n-1$
- Together, the values e and n are called the public key
- A message y is decrypted using the function $g(y) = y^d \; mod \; n$, where $1 \leq d < m$ is the multiplicative inverse of e mod m, i.e., $ed \bmod m = 1$
- The value d is called the private key

Example: RSA Cryptosystem

```
import random
import stdio

def is_prime(N):
    if N < 2:
        return False
    i=2
    while i <= N // i:
        if N % i == 0:
            return False
        i += 1
    return True

def primes(N):
    return filter(is_prime, range(N))

def inverse(e, m):
    return filter(lambda d: e * d % m == 1, range(1, m))[0]

def make_encoder_decoder(N):
    p, q = random.sample(primes(N), 2)
    n = p * q
    m = (p - 1) * (q - 1)
    stdio.writef('Maximum number that can be encrypted is %d\n', n - 1)
    e = random.choice(primes(m))
    while m % e == 0:
        e = random.choice(primes(m))
    d = inverse(e, m)
    return [lambda x: (x ** e) % n, lambda y: (y ** d) % n]
```

Example: RSA Cryptosystem

```
>>> import cryptography
>>> encoder, decoder = cryptography.make_encoder_decoder(100)
Maximum number that can be encrypted is 2536
>>> encoder(42)
2235L
>>> decoder(2235)
42L
>>> decoder(encoder(1729))
1729L
```

Next time… **File IO**

File I/O

- `open(fn, 'w’)`
- `open(fn, 'r’)`
- `open(fn, 'a')`
- `fn.close()`
- `fn.read()`
- `fn.readline()`
- `fn.readlines()`
- `fn.write(s)`
- `fn.writelines(S)`