61A Lecture 21

Monday, October 17

Space Consumption

Space Consumption

Which environment frames do we need to keep during evaluation?

Space Consumption

Which environment frames do we need to keep during evaluation?

Each step of evaluation has a set of **active** environments.

Space Consumption

Which environment frames do we need to keep during evaluation?

Each step of evaluation has a set of **active** environments.

Values and frames referenced by active environments are kept.

Space Consumption

Which environment frames do we need to keep during evaluation?

Each step of evaluation has a set of **active** environments.

Values and frames referenced by active environments are kept.

Memory used for other values and frames can be reclaimed.

Space Consumption

Which environment frames do we need to keep during evaluation?

Each step of evaluation has a set of **active** environments.

Values and frames referenced by active environments are kept.

Memory used for other values and frames can be reclaimed.

Active environments:

Space Consumption

Which environment frames do we need to keep during evaluation?

Each step of evaluation has a set of **active** environments.

Values and frames referenced by active environments are kept.

Memory used for other values and frames can be reclaimed.

Active environments:

- The environment for the current expression being evaluated

Space Consumption

Which environment frames do we need to keep during evaluation?

Each step of evaluation has a set of **active** environments.

Values and frames referenced by active environments are kept.

Memory used for other values and frames can be reclaimed.

Active environments:

- The environment for the current expression being evaluated
- Environments for calls that depend upon the value of the current expression

Space Consumption

Which environment frames do we need to keep during evaluation?

Each step of evaluation has a set of **active** environments.

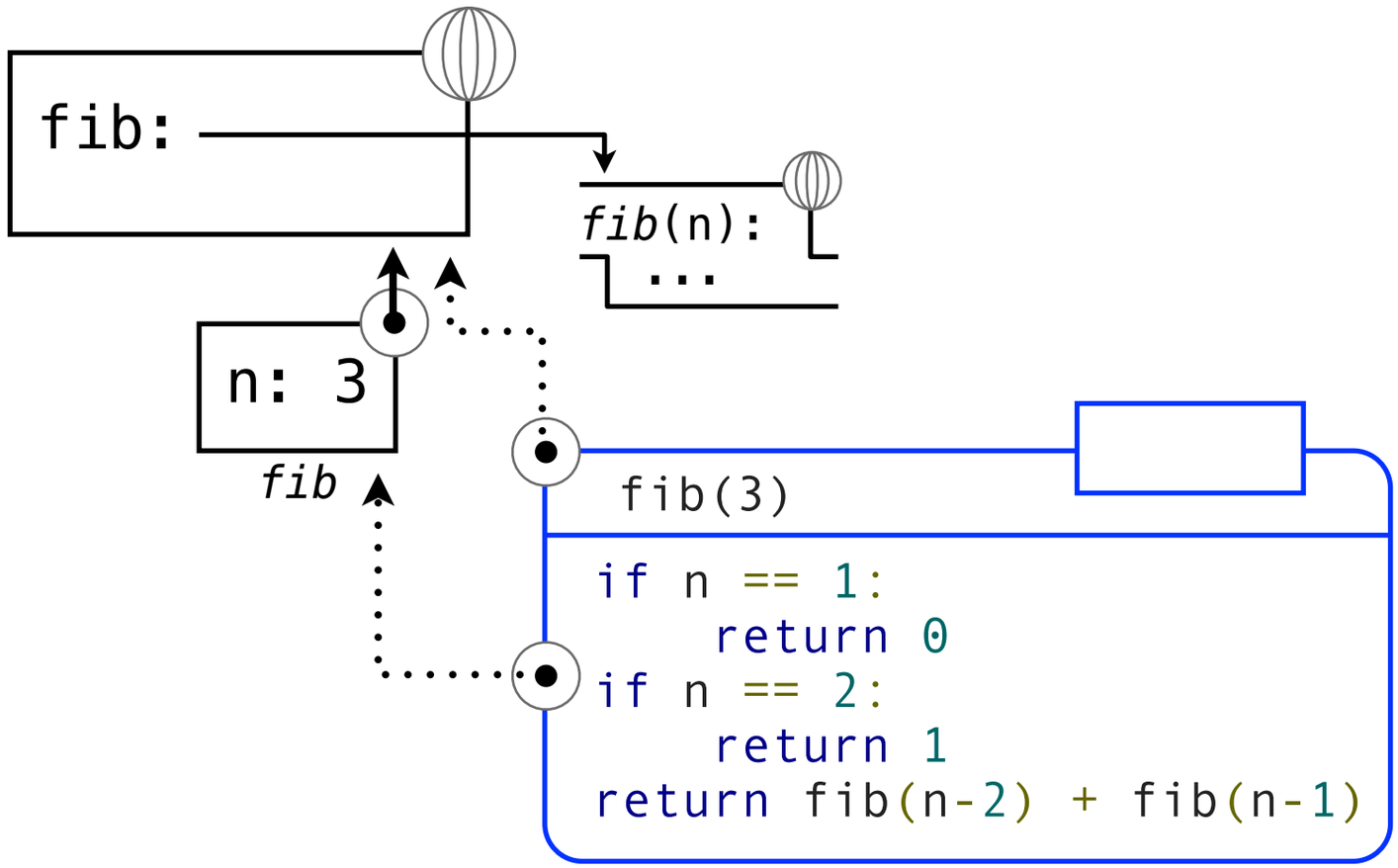
Values and frames referenced by active environments are kept.

Memory used for other values and frames can be reclaimed.

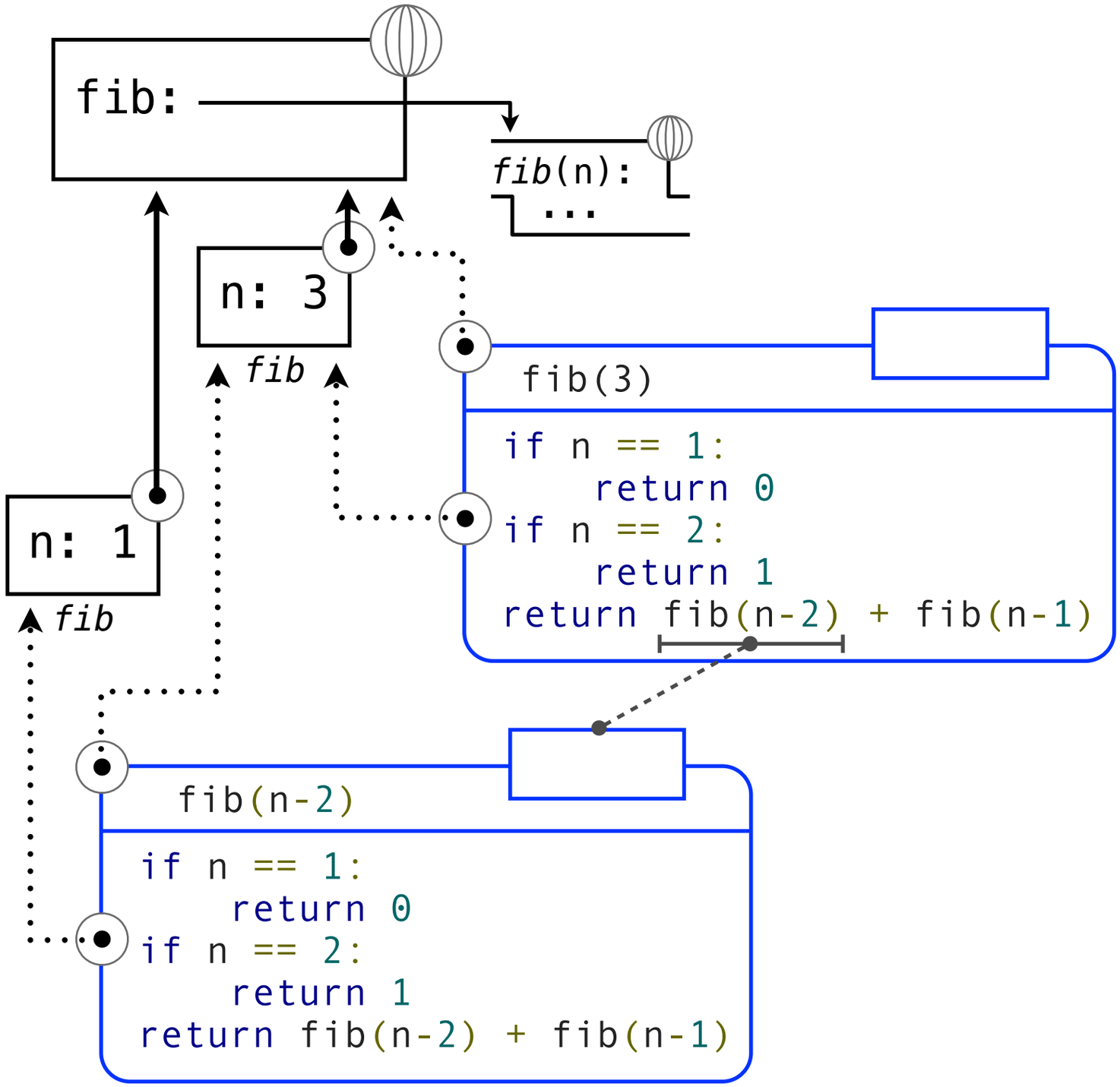
Active environments:

- The environment for the current expression being evaluated
- Environments for calls that depend upon the value of the current expression
- Environments associated with functions referenced by active environments

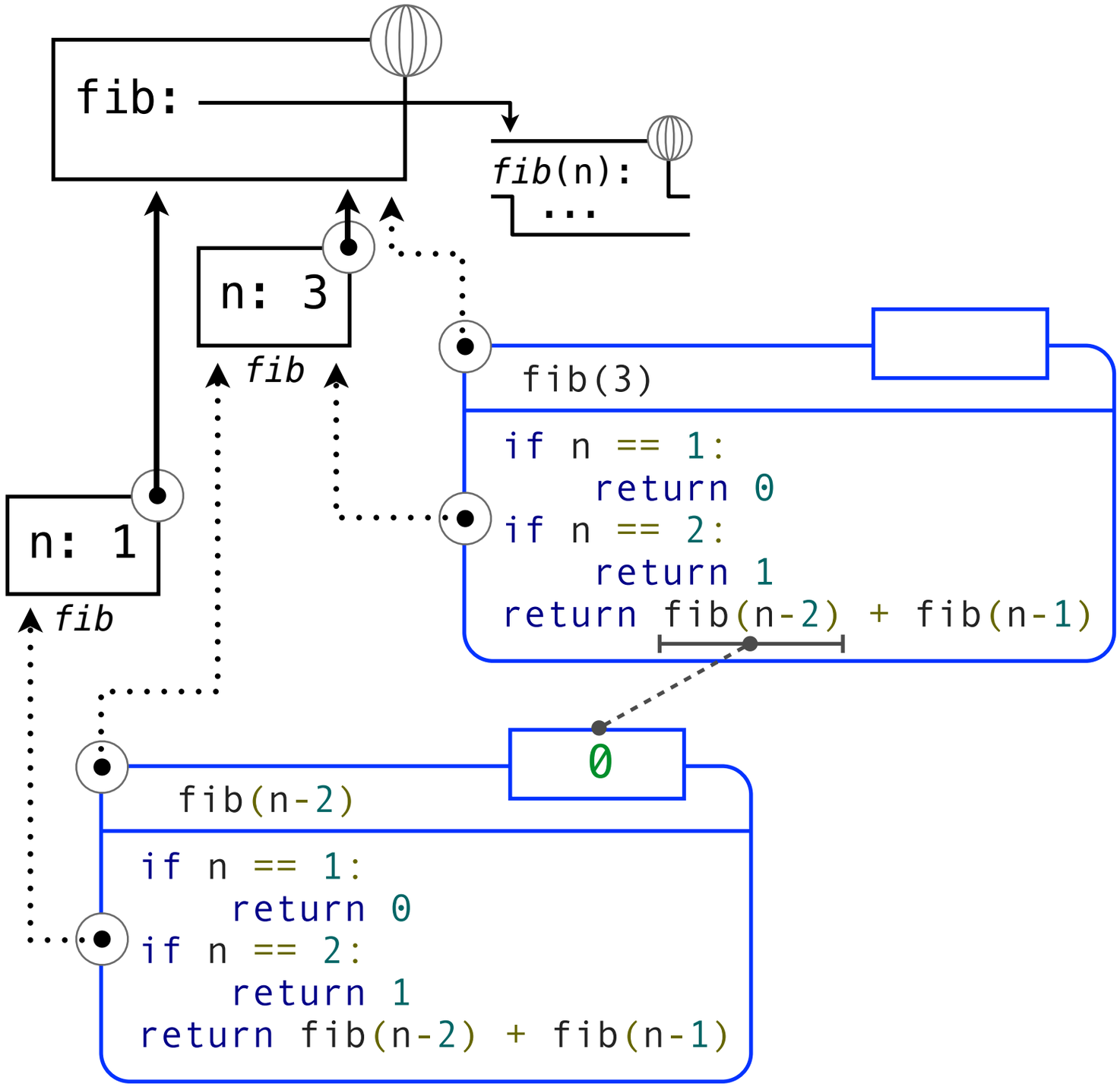
Fibonacci Environment Diagram



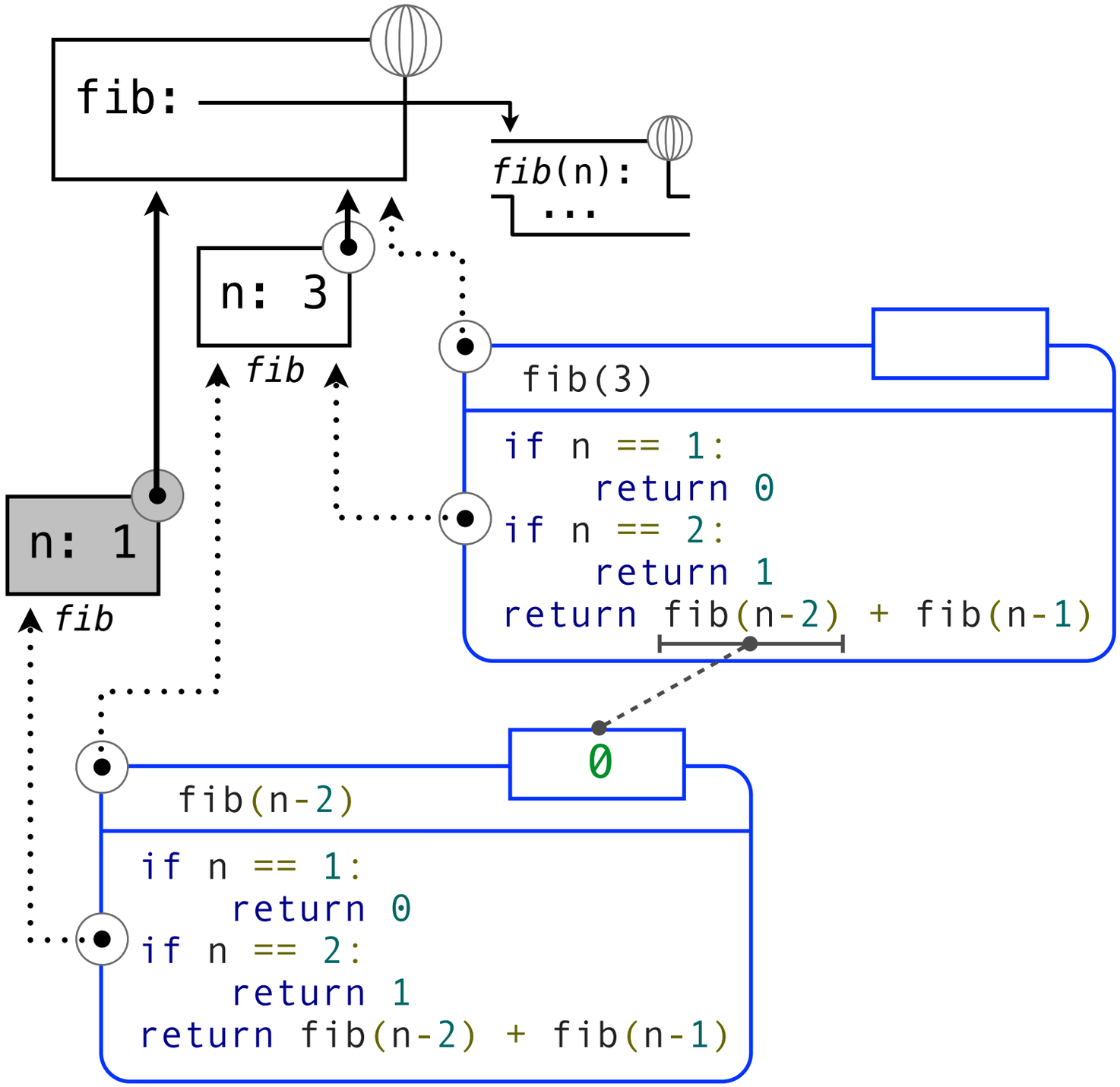
Fibonacci Environment Diagram



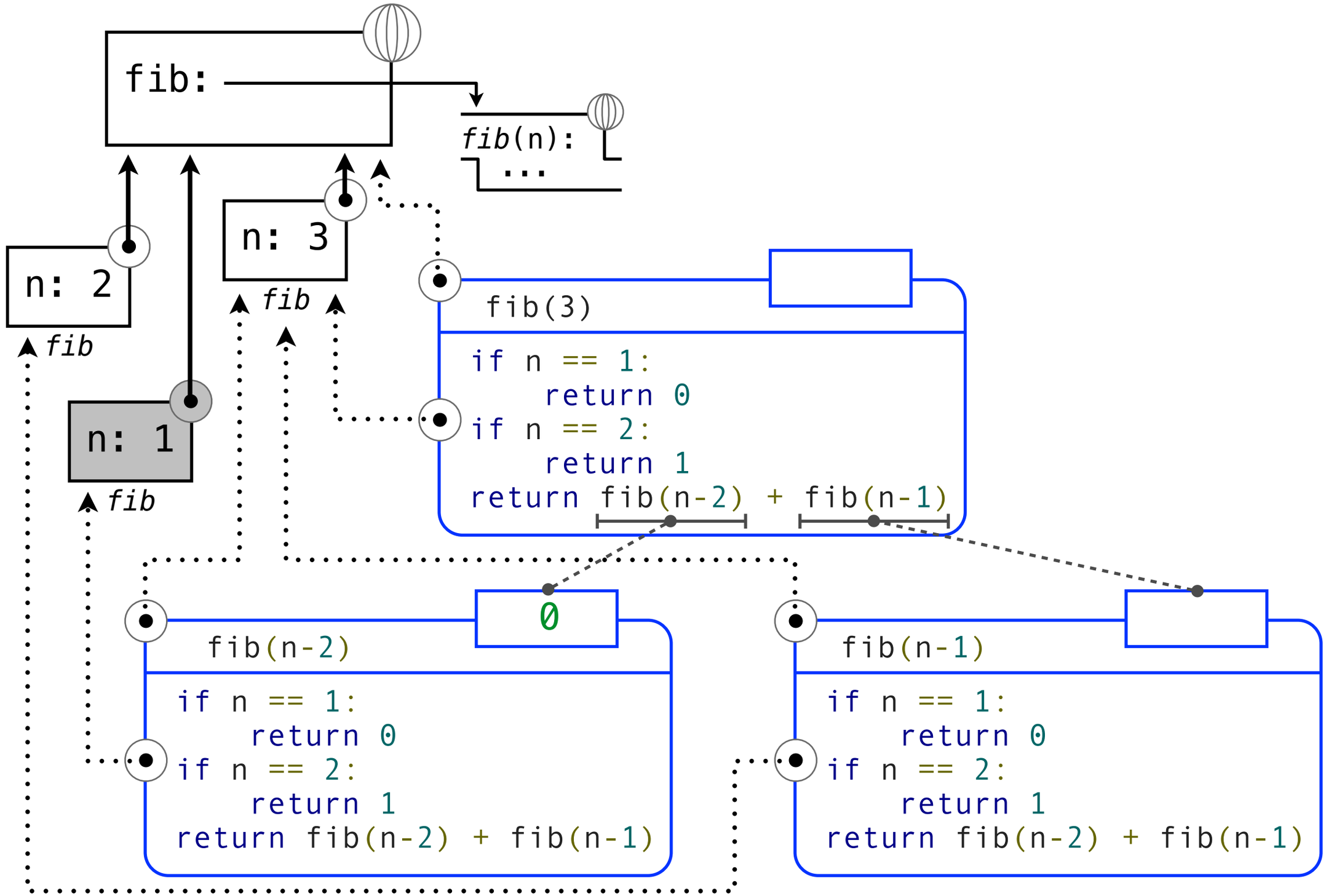
Fibonacci Environment Diagram



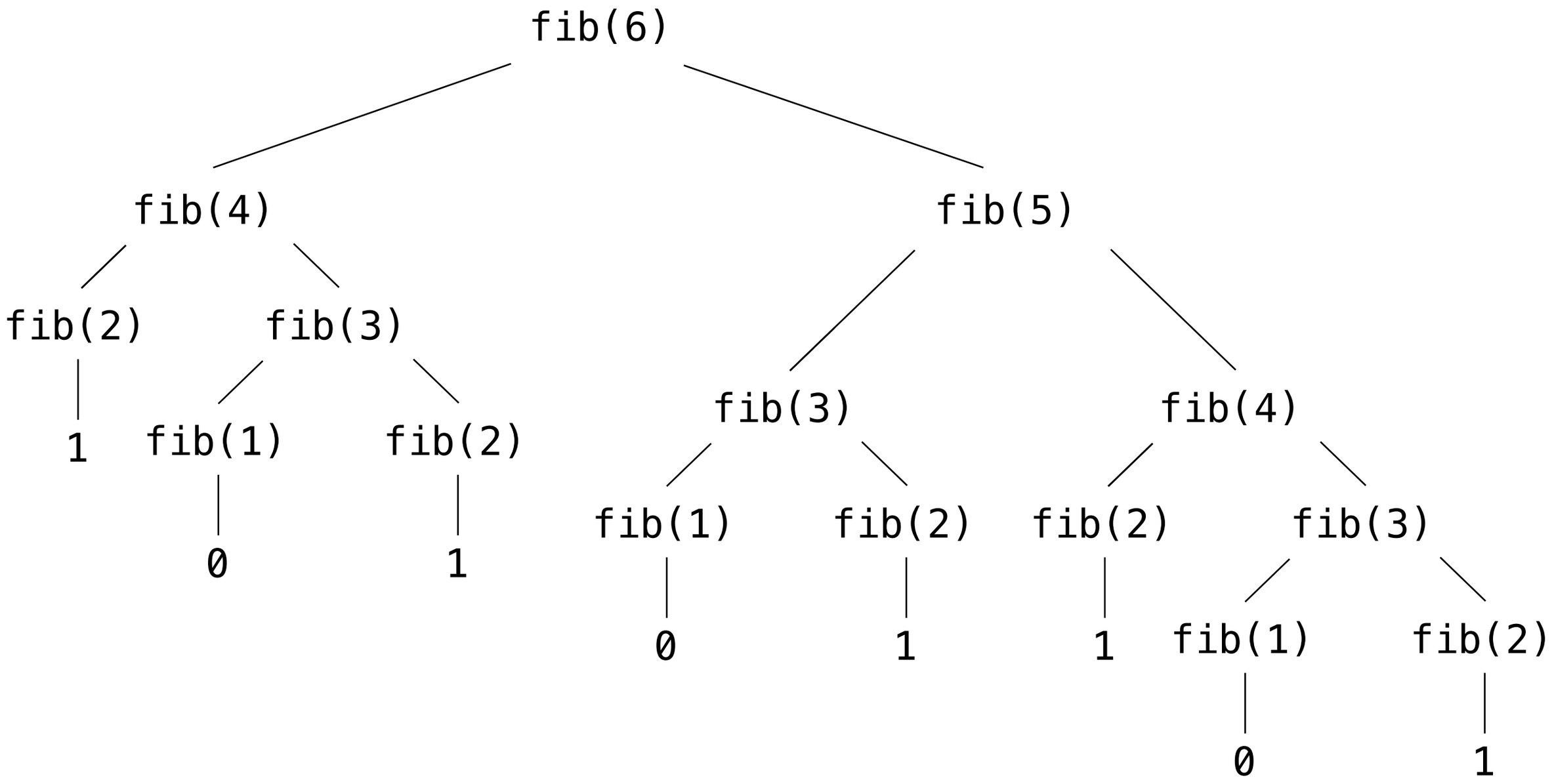
Fibonacci Environment Diagram



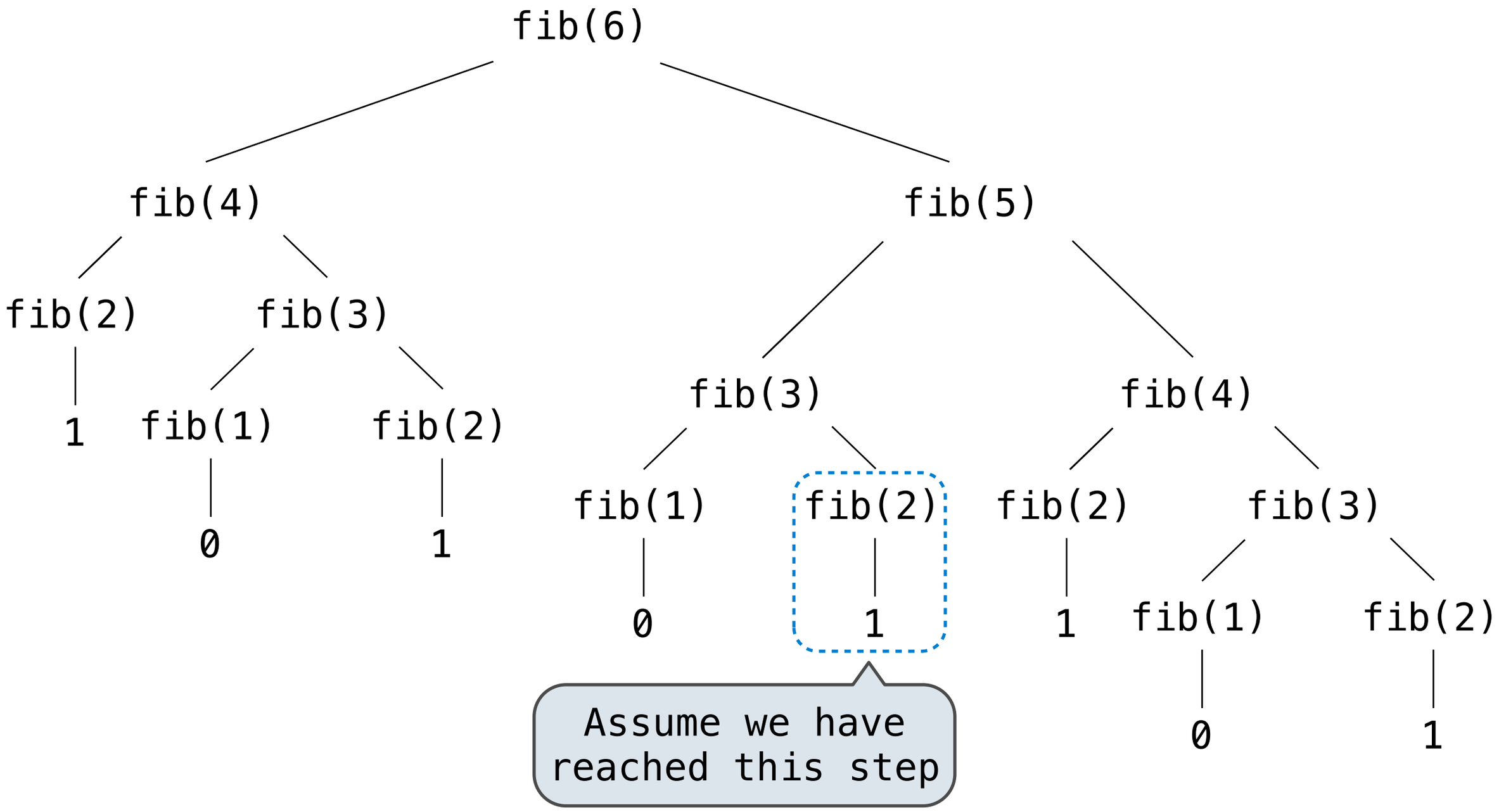
Fibonacci Environment Diagram



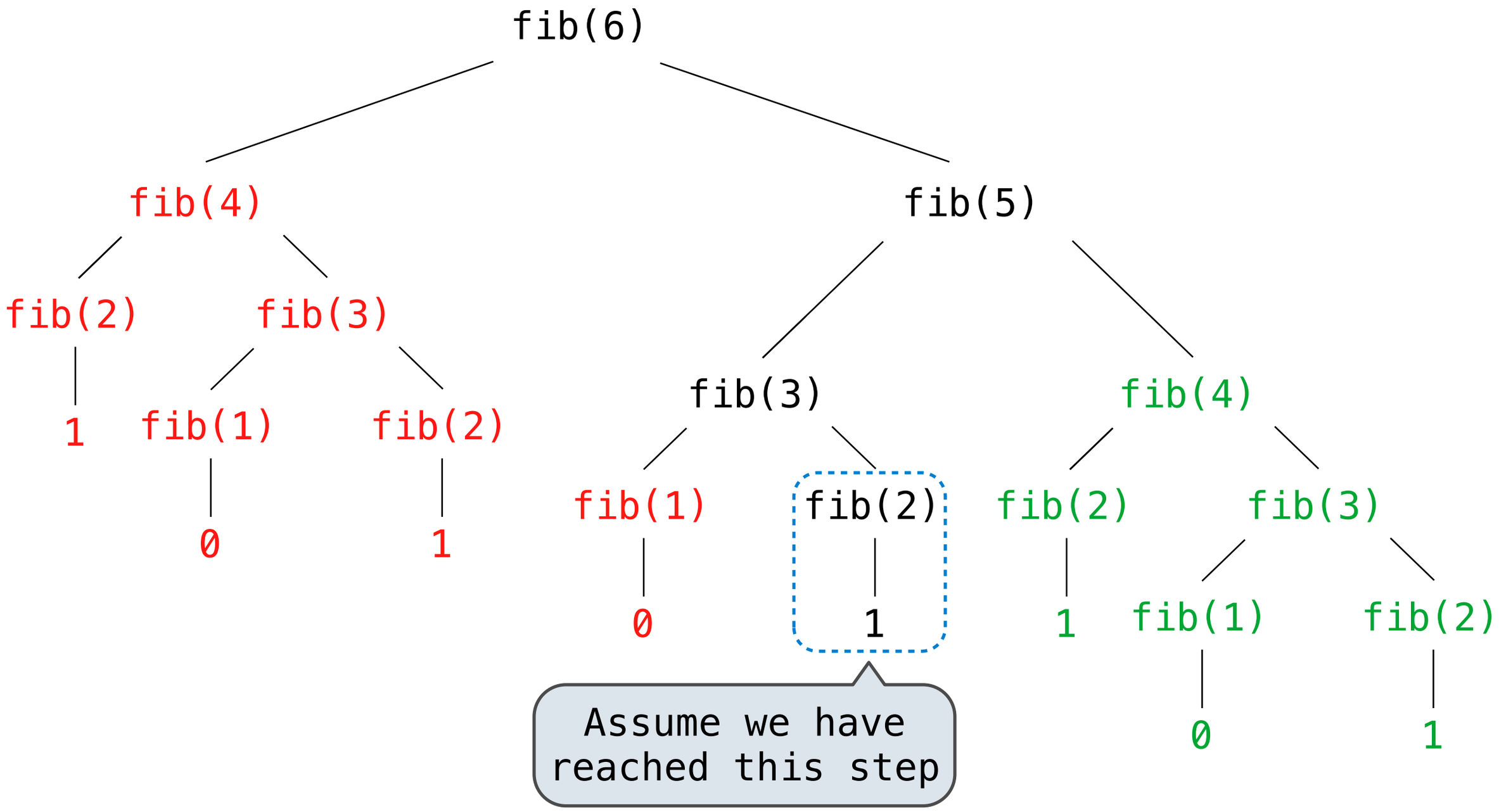
Fibonacci Memory Consumption



Fibonacci Memory Consumption

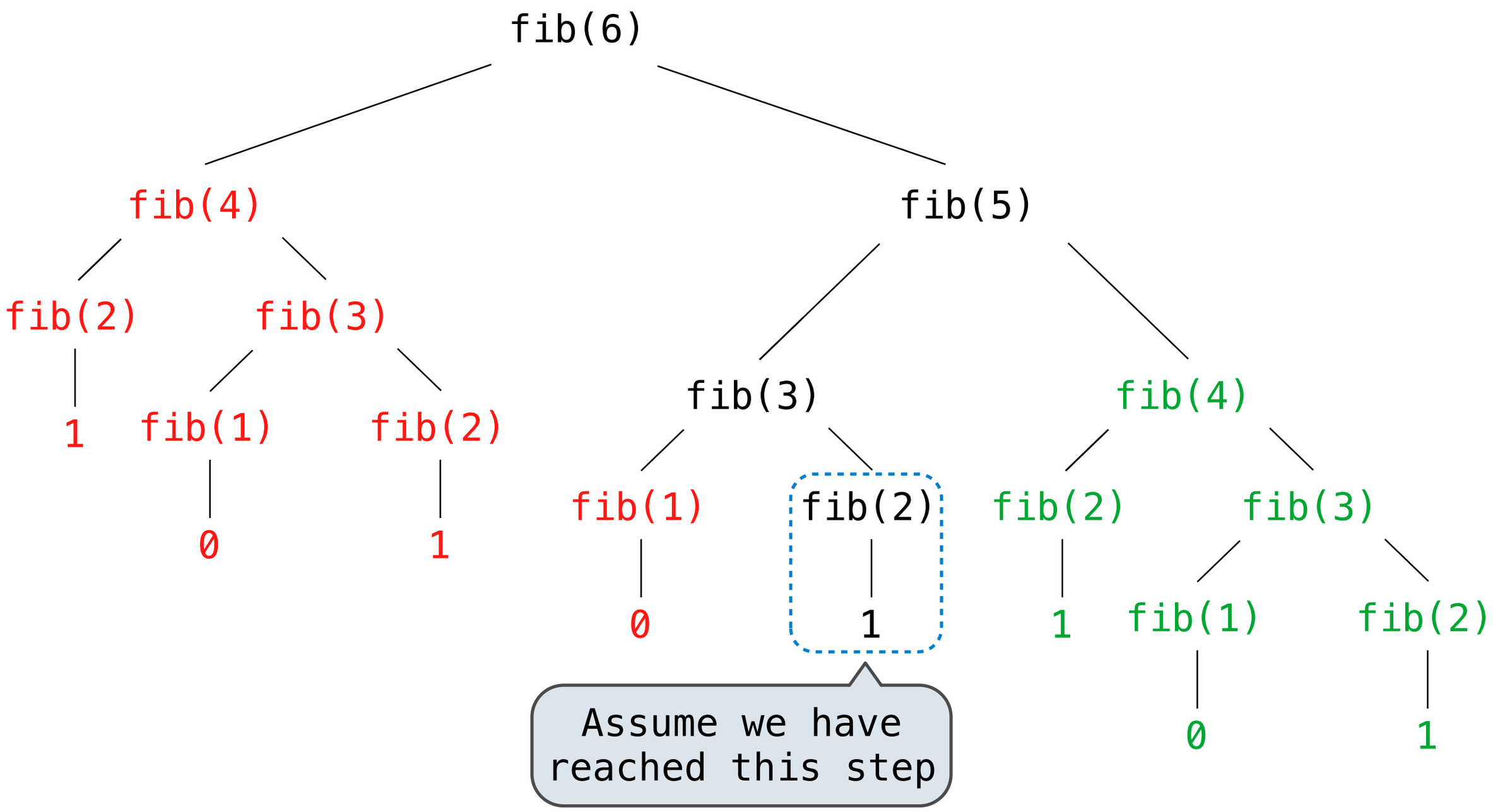


Fibonacci Memory Consumption



Fibonacci Memory Consumption

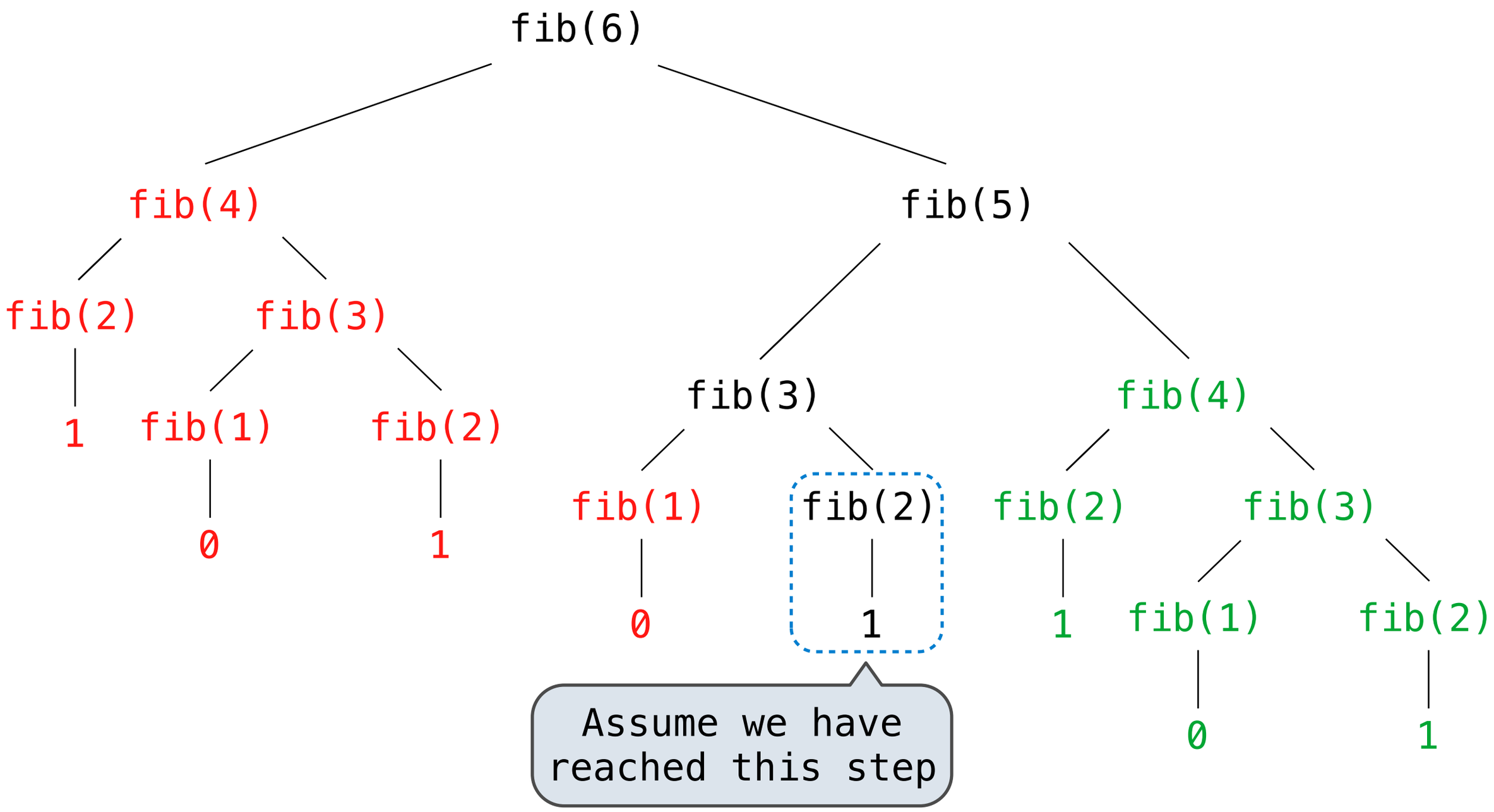
Has an active environment



Fibonacci Memory Consumption

Has an active environment

Can be reclaimed

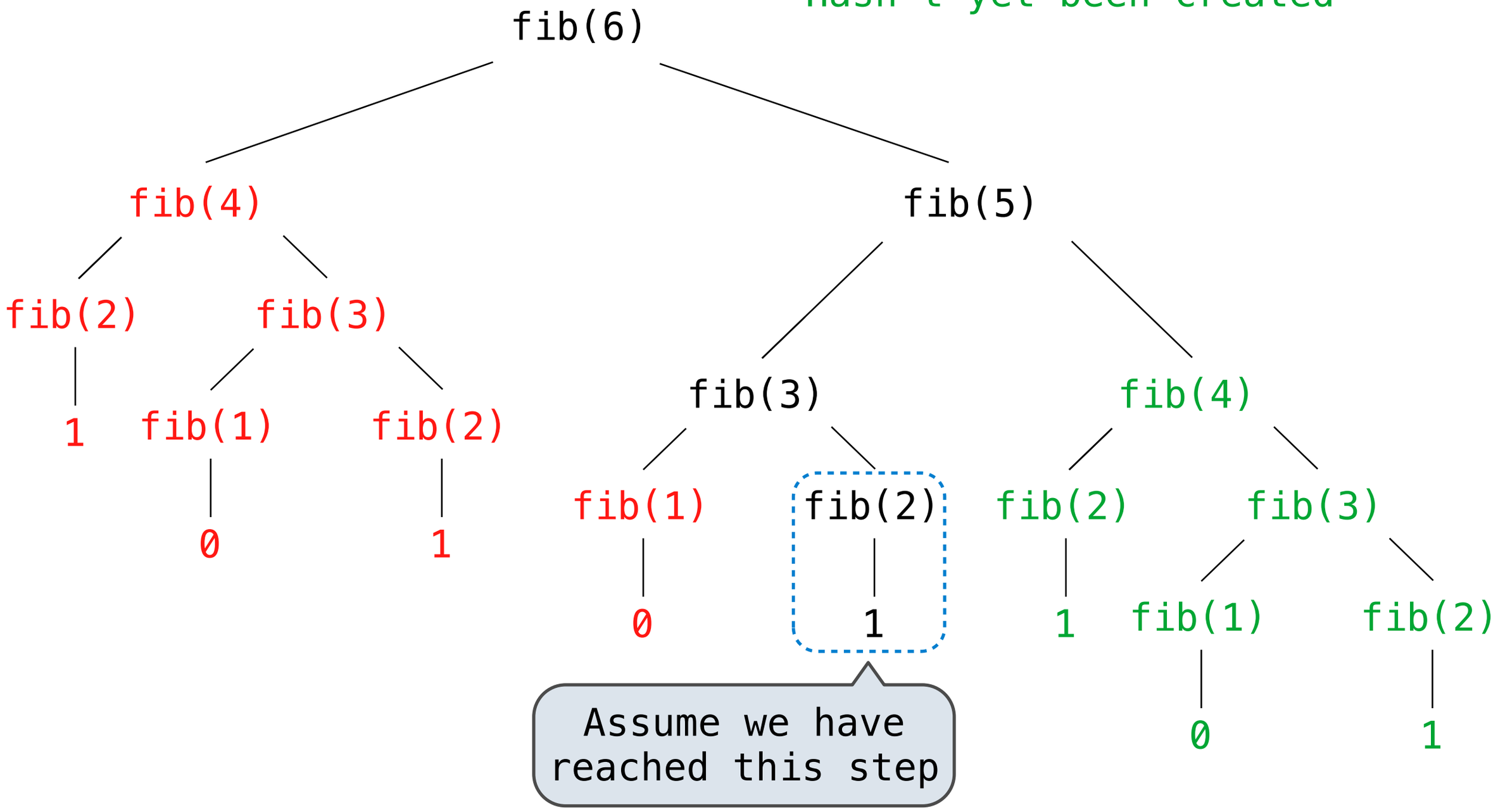


Fibonacci Memory Consumption

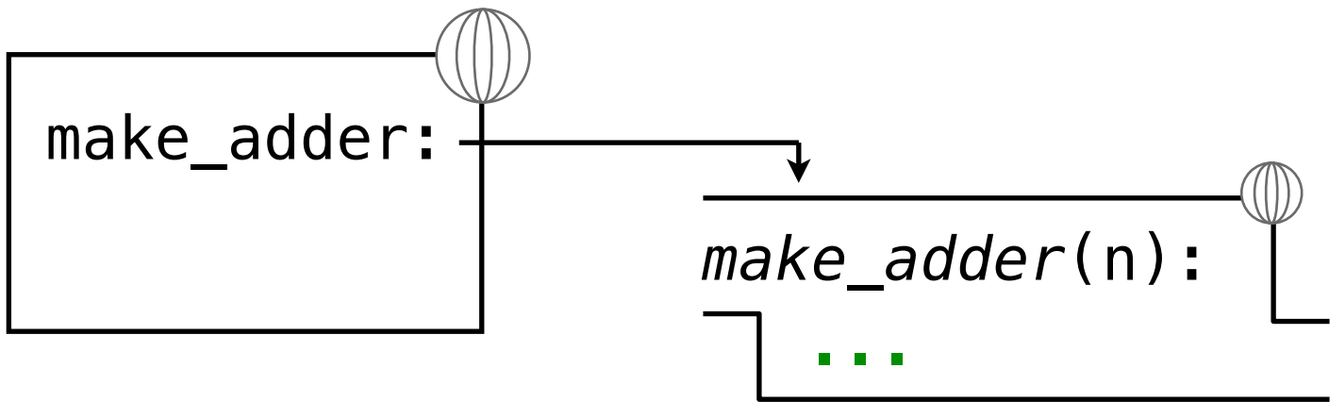
Has an active environment

Can be reclaimed

Hasn't yet been created

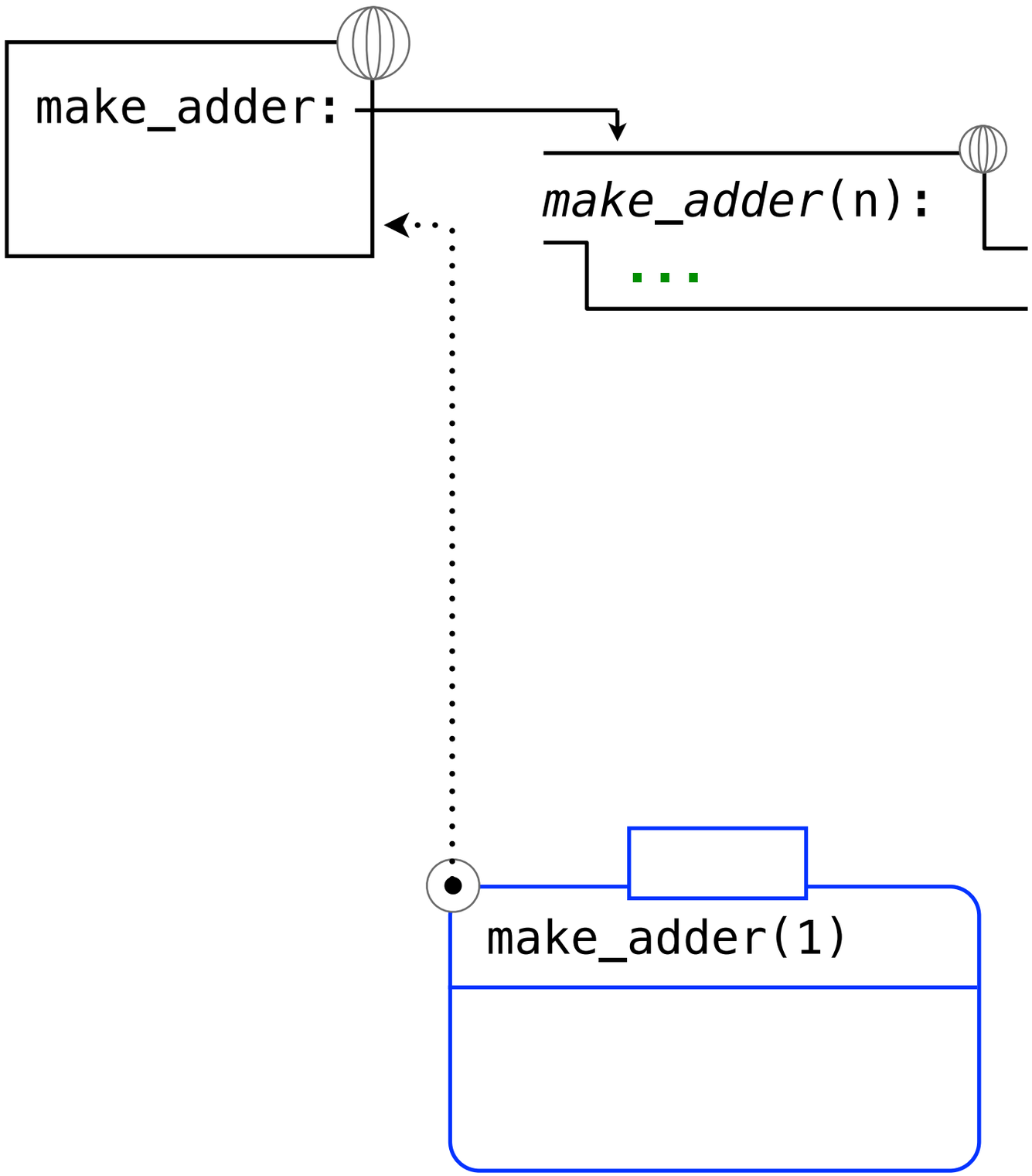


Active Environments for Returned Functions



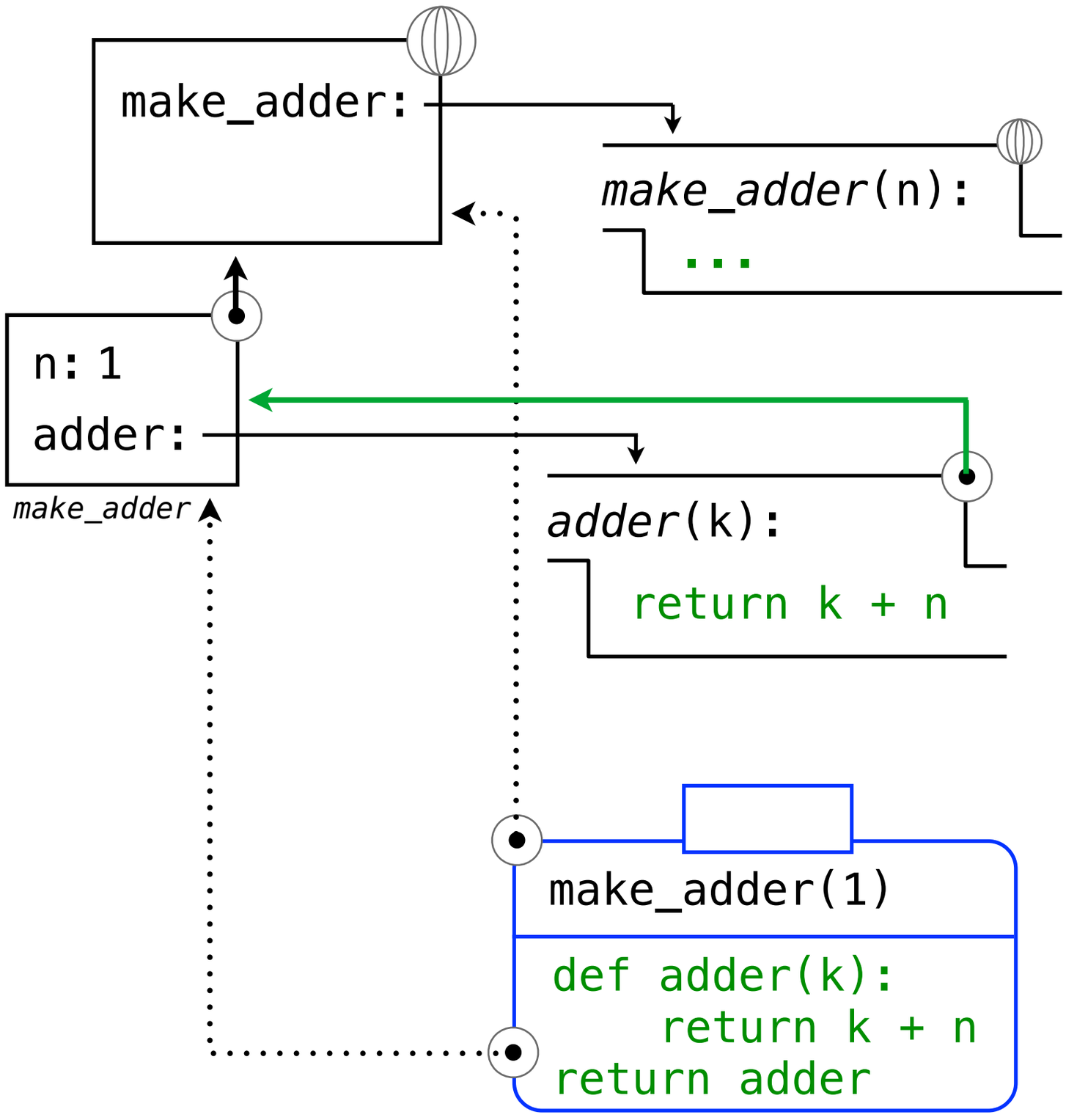

```
def make_adder(n):
    def adder(k):
        return k + n
    return adder
add1 = make_adder(1)
```

Active Environments for Returned Functions



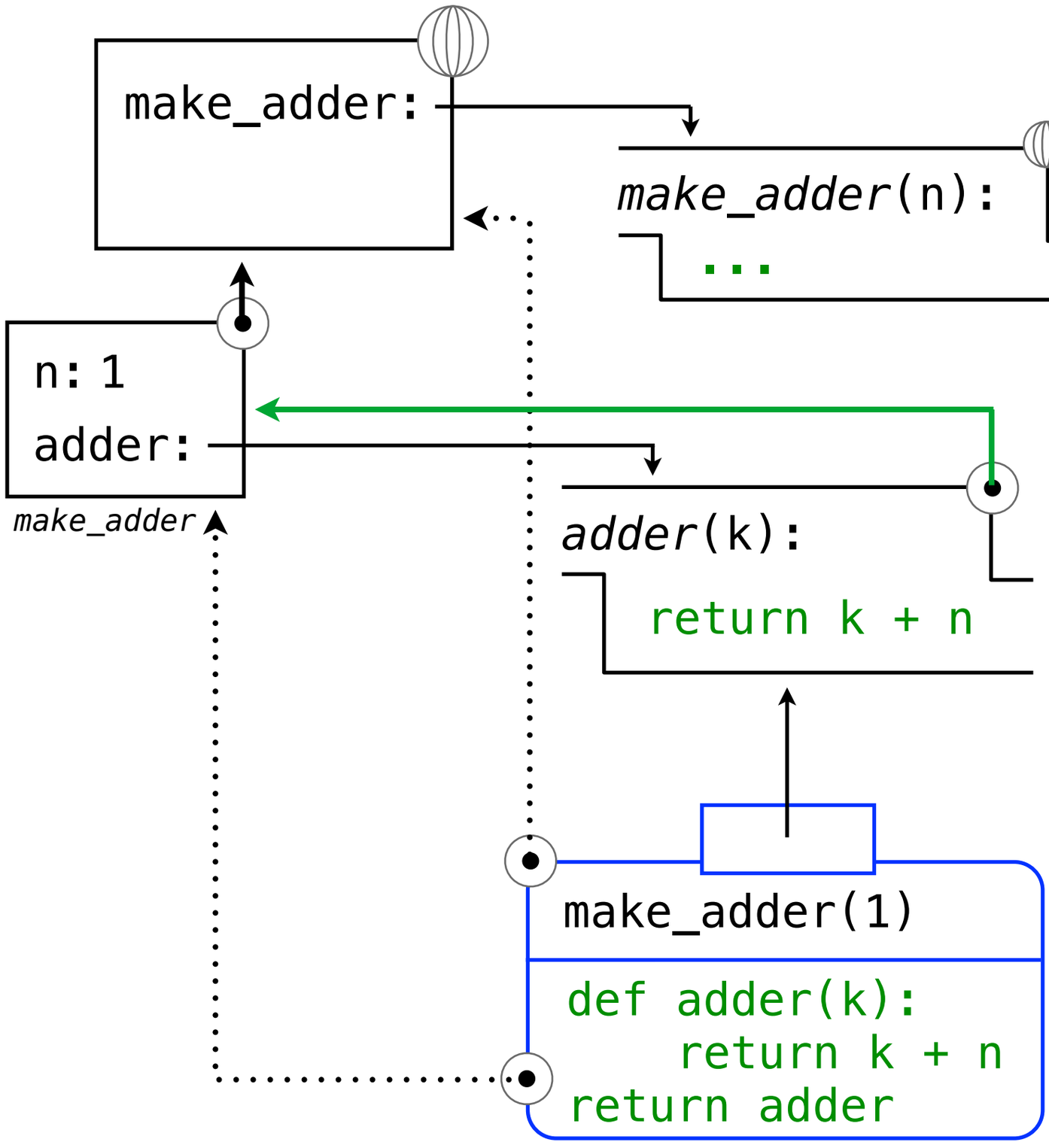

```
def make_adder(n):
    def adder(k):
        return k + n
    return adder
add1 = make_adder(1)
```

Active Environments for Returned Functions



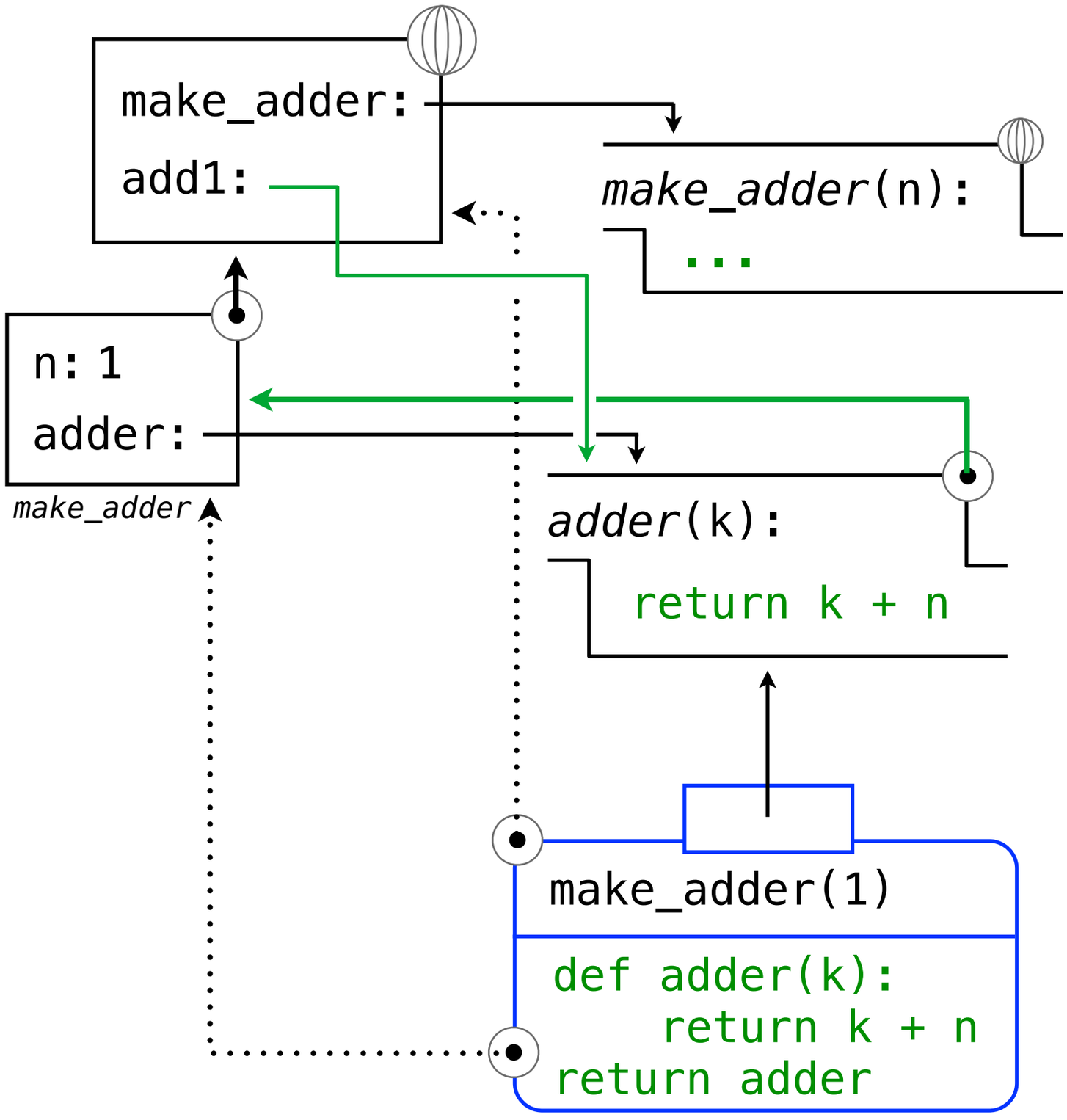

```
def make_adder(n):
    def adder(k):
        return k + n
    return adder
add1 = make_adder(1)
```

Active Environments for Returned Functions



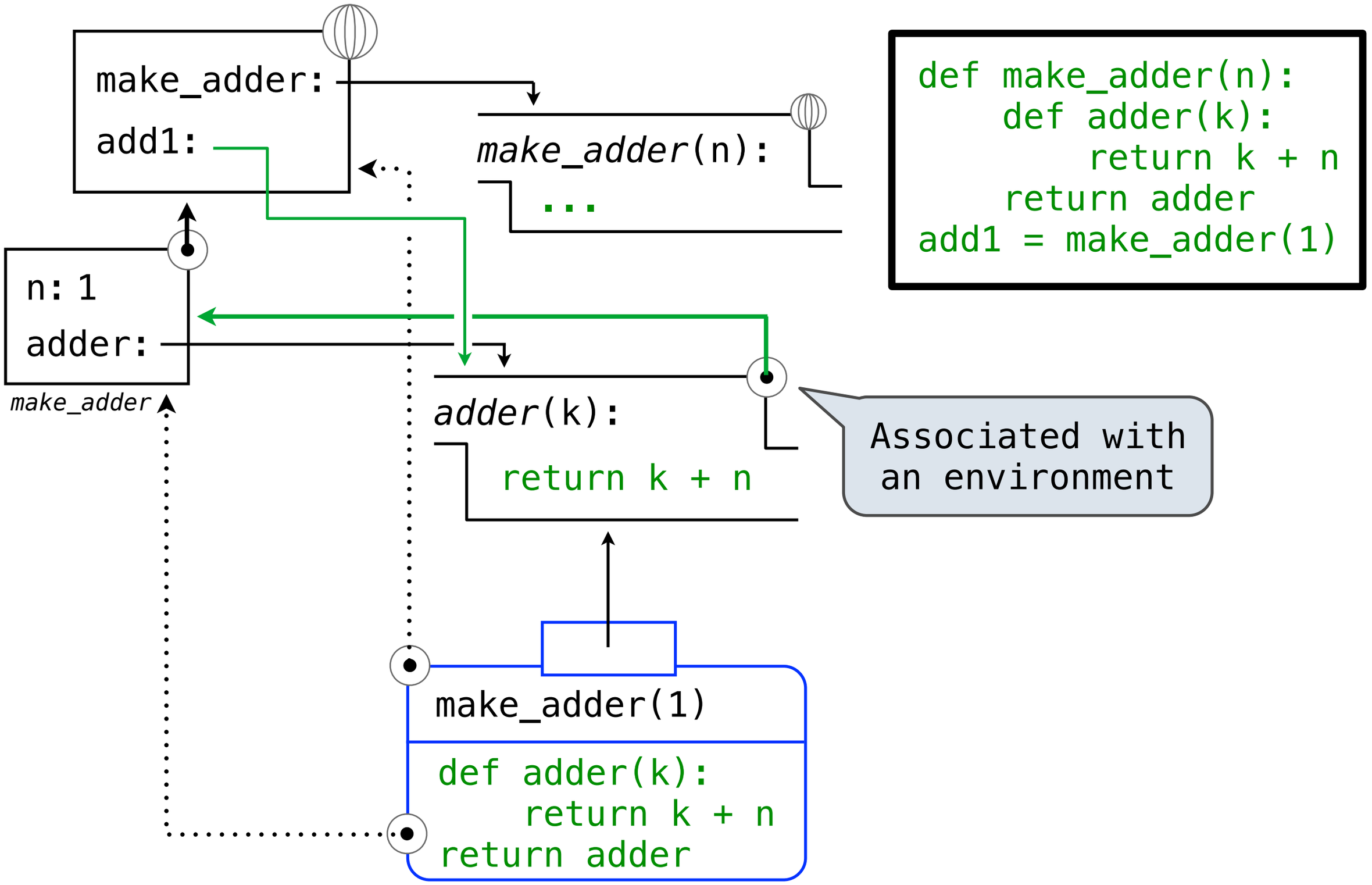

```
def make_adder(n):
    def adder(k):
        return k + n
    return adder
add1 = make_adder(1)
```

Active Environments for Returned Functions



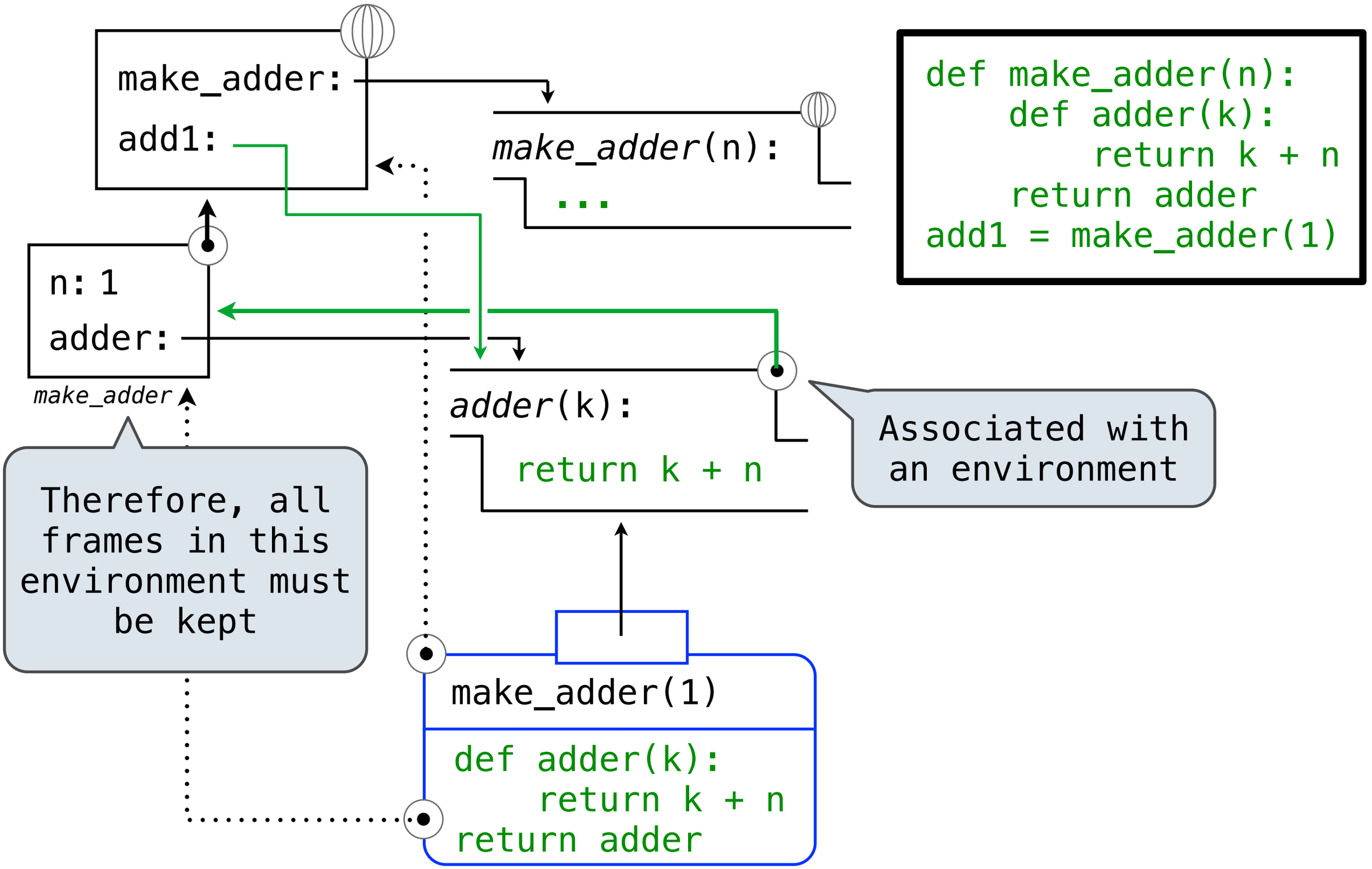

```
def make_adder(n):
    def adder(k):
        return k + n
    return adder
add1 = make_adder(1)
```

Active Environments for Returned Functions




```
def make_adder(n):
    def adder(k):
        return k + n
    return adder
add1 = make_adder(1)
```

Active Environments for Returned Functions



Order of Growth

Order of Growth

A method for bounding the resources used by a function as the "size" of a problem increases

Order of Growth

A method for bounding the resources used by a function as the "size" of a problem increases

n: size of the problem

Order of Growth

A method for bounding the resources used by a function as the "size" of a problem increases

n: size of the problem

***R*(*n*)**: Measurement of some resource used (time or space)

Order of Growth

A method for bounding the resources used by a function as the "size" of a problem increases

n: size of the problem

***R*(*n*)**: Measurement of some resource used (time or space)

$$R(n) = \Theta(f(n))$$

Order of Growth

A method for bounding the resources used by a function as the "size" of a problem increases

n: size of the problem

***R*(*n*)**: Measurement of some resource used (time or space)

$$R(n) = \Theta(f(n))$$

means that there are constants k_1 and k_2 such that

Order of Growth

A method for bounding the resources used by a function as the "size" of a problem increases

n: size of the problem

***R*(*n*)**: Measurement of some resource used (time or space)

$$R(n) = \Theta(f(n))$$

means that there are constants k_1 and k_2 such that

$$k_1 \cdot f(n) \leq R(n) \leq k_2 \cdot f(n)$$

Order of Growth

A method for bounding the resources used by a function as the "size" of a problem increases

n: size of the problem

***R*(*n*)**: Measurement of some resource used (time or space)

$$R(n) = \Theta(f(n))$$

means that there are constants k_1 and k_2 such that

$$k_1 \cdot f(n) \leq R(n) \leq k_2 \cdot f(n)$$

for sufficiently large values of ***n*.**

Iteration vs Memoized Tree Recursion

Iterative and memoized implementations are not the same.

	Time	Space

```
def fib_iter(n):
    prev, curr = 1, 0
    for _ in range(n-1):
        prev, curr = curr, prev + curr
    return curr
```

```
@memo
def fib(n):
    if n == 1:
        return 0
    if n == 2:
        return 1
    return fib(n-2) + fib(n-1)
```

Iteration vs Memoized Tree Recursion

Iterative and memoized implementations are not the same.

```
def fib_iter(n):
    prev, curr = 1, 0
    for _ in range(n-1):
        prev, curr = curr, prev + curr
    return curr
```

```
@memo
def fib(n):
    if n == 1:
        return 0
    if n == 2:
        return 1
    return fib(n-2) + fib(n-1)
```

Time	Space
$\Theta(n)$	

Iteration vs Memoized Tree Recursion

Iterative and memoized implementations are not the same.

```
def fib_iter(n):
    prev, curr = 1, 0
    for _ in range(n-1):
        prev, curr = curr, prev + curr
    return curr
```

```
@memo
def fib(n):
    if n == 1:
        return 0
    if n == 2:
        return 1
    return fib(n-2) + fib(n-1)
```

Time	Space
$\Theta(n)$	$\Theta(1)$

Iteration vs Memoized Tree Recursion

Iterative and memoized implementations are not the same.

```
def fib_iter(n):
    prev, curr = 1, 0
    for _ in range(n-1):
        prev, curr = curr, prev + curr
    return curr
```

```
@memo
def fib(n):
    if n == 1:
        return 0
    if n == 2:
        return 1
    return fib(n-2) + fib(n-1)
```

Time	Space
$\Theta(n)$	$\Theta(1)$
$\Theta(n)$	

Iteration vs Memoized Tree Recursion

Iterative and memoized implementations are not the same.

```
def fib_iter(n):
    prev, curr = 1, 0
    for _ in range(n-1):
        prev, curr = curr, prev + curr
    return curr
```

```
@memo
def fib(n):
    if n == 1:
        return 0
    if n == 2:
        return 1
    return fib(n-2) + fib(n-1)
```

	Time	Space
	$\Theta(n)$	$\Theta(1)$
	$\Theta(n)$	$\Theta(n)$

Comparing orders of growth

Comparing orders of growth

$$\Theta(b^n)$$

Comparing orders of growth

$\Theta(b^n)$ Exponential growth! Recursive fib takes

Comparing orders of growth

$\Theta(b^n)$ Exponential growth! Recursive fib takes $\Theta(\phi^n)$ steps, where $\phi = \frac{1+\sqrt{5}}{2} \approx 1.61828$

Comparing orders of growth

$\Theta(b^n)$ Exponential growth! Recursive fib takes

$\Theta(\phi^n)$ steps, where $\phi = \dfrac{1+\sqrt{5}}{2} \approx 1.61828$

Incrementing the problem scales R(n) by a factor.

Comparing orders of growth

$\Theta(b^n)$ Exponential growth! Recursive fib takes $\Theta(\phi^n)$ steps, where $\phi = \frac{1+\sqrt{5}}{2} \approx 1.61828$

Incrementing the problem scales R(n) by a factor.

$\Theta(n)$

Comparing orders of growth

$\Theta(b^n)$ Exponential growth! Recursive fib takes $\Theta(\phi^n)$ steps, where $\phi = \frac{1+\sqrt{5}}{2} \approx 1.61828$

Incrementing the problem scales R(n) by a factor.

$\Theta(n)$ Linear growth. Resources scale with the problem.

Comparing orders of growth

$\Theta(b^n)$ Exponential growth! Recursive fib takes $\Theta(\phi^n)$ steps, where $\phi = \frac{1+\sqrt{5}}{2} \approx 1.61828$

Incrementing the problem scales R(n) by a factor.

$\Theta(n)$ Linear growth. Resources scale with the problem.

$\Theta(\log n)$

Comparing orders of growth

$\Theta(b^n)$ Exponential growth! Recursive fib takes $\Theta(\phi^n)$ steps, where $\phi = \frac{1+\sqrt{5}}{2} \approx 1.61828$

Incrementing the problem scales R(n) by a factor.

$\Theta(n)$ Linear growth. Resources scale with the problem.

$\Theta(\log n)$ Logarithmic growth. These functions scale well.

Comparing orders of growth

$\Theta(b^n)$ Exponential growth! Recursive fib takes $\Theta(\phi^n)$ steps, where $\phi = \frac{1+\sqrt{5}}{2} \approx 1.61828$

Incrementing the problem scales R(n) by a factor.

$\Theta(n)$ Linear growth. Resources scale with the problem.

$\Theta(\log n)$ Logarithmic growth. These functions scale well.

Doubling the problem increments resources needed.

Comparing orders of growth

$\Theta(b^n)$ Exponential growth! Recursive fib takes $\Theta(\phi^n)$ steps, where $\phi = \frac{1+\sqrt{5}}{2} \approx 1.61828$

Incrementing the problem scales R(n) by a factor.

$\Theta(n)$ Linear growth. Resources scale with the problem.

$\Theta(\log n)$ Logarithmic growth. These functions scale well.

Doubling the problem increments resources needed.

$\Theta(1)$

Comparing orders of growth

$\Theta(b^n)$ Exponential growth! Recursive fib takes $\Theta(\phi^n)$ steps, where $\phi = \frac{1+\sqrt{5}}{2} \approx 1.61828$

Incrementing the problem scales R(n) by a factor.

$\Theta(n)$ Linear growth. Resources scale with the problem.

$\Theta(\log n)$ Logarithmic growth. These functions scale well.

Doubling the problem increments resources needed.

$\Theta(1)$ Constant. The problem size doesn't matter.

Exponentiation

Exponentiation

Goal: one more multiplication lets us double the problem size.

Exponentiation

Goal: one more multiplication lets us double the problem size.

```
def exp(b, n):
    if n == 0:
        return 1
    return b * exp(b, n-1)
```

Exponentiation

Goal: one more multiplication lets us double the problem size.

```
def exp(b, n):
    if n == 0:
        return 1
    return b * exp(b, n-1)
```

$$b^n = \begin{cases} 1 & \text{if } n = 0 \\ b \cdot b^{n-1} & \text{otherwise} \end{cases}$$

Exponentiation

Goal: one more multiplication lets us double the problem size.

```
def exp(b, n):
    if n == 0:
        return 1
    return b * exp(b, n-1)
```

$$b^n = \begin{cases} 1 & \text{if } n = 0 \\ b \cdot b^{n-1} & \text{otherwise} \end{cases}$$

$$b^n = \begin{cases} 1 & \text{if } n = 0 \\ (b^{\frac{1}{2}n})^2 & \text{if } n \text{ is even} \\ b \cdot b^{n-1} & \text{if } n \text{ is odd} \end{cases}$$

Exponentiation

Goal: one more multiplication lets us double the problem size.

```
def exp(b, n):
    if n == 0:
        return 1
    return b * exp(b, n-1)
```

$$b^n = \begin{cases} 1 & \text{if } n = 0 \\ b \cdot b^{n-1} & \text{otherwise} \end{cases}$$

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

$$b^n = \begin{cases} 1 & \text{if } n = 0 \\ (b^{\frac{1}{2}n})^2 & \text{if } n \text{ is even} \\ b \cdot b^{n-1} & \text{if } n \text{ is odd} \end{cases}$$

Exponentiation

Goal: one more multiplication lets us double the problem size.

```
def exp(b, n):
    if n == 0:
        return 1
    return b * exp(b, n-1)
```

$$b^n = \begin{cases} 1 & \text{if } n = 0 \\ b \cdot b^{n-1} & \text{otherwise} \end{cases}$$

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

def fast_exp(b, n):
```

$$b^n = \begin{cases} 1 & \text{if } n = 0 \\ (b^{\frac{1}{2}n})^2 & \text{if } n \text{ is even} \\ b \cdot b^{n-1} & \text{if } n \text{ is odd} \end{cases}$$

Exponentiation

Goal: one more multiplication lets us double the problem size.

```
def exp(b, n):
    if n == 0:
        return 1
    return b * exp(b, n-1)
```

$$b^n = \begin{cases} 1 & \text{if } n = 0 \\ b \cdot b^{n-1} & \text{otherwise} \end{cases}$$

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

def fast_exp(b, n):
    if n == 0:
        return 1
```

$$b^n = \begin{cases} 1 & \text{if } n = 0 \\ (b^{\frac{1}{2}n})^2 & \text{if } n \text{ is even} \\ b \cdot b^{n-1} & \text{if } n \text{ is odd} \end{cases}$$

Exponentiation

Goal: one more multiplication lets us double the problem size.

```
def exp(b, n):
    if n == 0:
        return 1
    return b * exp(b, n-1)
```

$$b^n = \begin{cases} 1 & \text{if } n = 0 \\ b \cdot b^{n-1} & \text{otherwise} \end{cases}$$

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

def fast_exp(b, n):
    if n == 0:
        return 1
    if n % 2 == 0:
        return square(fast_exp(b, n//2))
```

$$b^n = \begin{cases} 1 & \text{if } n = 0 \\ (b^{\frac{1}{2}n})^2 & \text{if } n \text{ is even} \\ b \cdot b^{n-1} & \text{if } n \text{ is odd} \end{cases}$$

Exponentiation

Goal: one more multiplication lets us double the problem size.

```
def exp(b, n):
    if n == 0:
        return 1
    return b * exp(b, n-1)
```

$$b^n = \begin{cases} 1 & \text{if } n = 0 \\ b \cdot b^{n-1} & \text{otherwise} \end{cases}$$

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

def fast_exp(b, n):
    if n == 0:
        return 1
    if n % 2 == 0:
        return square(fast_exp(b, n//2))
    else:
        return b * fast_exp(b, n-1)
```

$$b^n = \begin{cases} 1 & \text{if } n = 0 \\ (b^{\frac{1}{2}n})^2 & \text{if } n \text{ is even} \\ b \cdot b^{n-1} & \text{if } n \text{ is odd} \end{cases}$$

Exponentiation

Goal: one more multiplication lets us double the problem size.

	Time	Space

```
def exp(b, n):
    if n == 0:
        return 1
    return b * exp(b, n-1)
```

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

def fast_exp(b, n):
    if n == 0:
        return 1
    if n % 2 == 0:
        return square(fast_exp(b, n//2))
    else:
        return b * fast_exp(b, n-1)
```

Exponentiation

Goal: one more multiplication lets us double the problem size.

	Time	**Space**
	$\Theta(n)$	$\Theta(n)$

```
def exp(b, n):
    if n == 0:
        return 1
    return b * exp(b, n-1)
```

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

def fast_exp(b, n):
    if n == 0:
        return 1
    if n % 2 == 0:
        return square(fast_exp(b, n//2))
    else:
        return b * fast_exp(b, n-1)
```

Exponentiation

Goal: one more multiplication lets us double the problem size.

```
def exp(b, n):
    if n == 0:
        return 1
    return b * exp(b, n-1)
```

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

def fast_exp(b, n):
    if n == 0:
        return 1
    if n % 2 == 0:
        return square(fast_exp(b, n//2))
    else:
        return b * fast_exp(b, n-1)
```

	Time	**Space**
	$\Theta(n)$	$\Theta(n)$
	$\Theta(\log n)$	$\Theta(\log n)$