Discussion 1

6/23

What did we cover?

- How do we **control** what code is executed?
- How **many times** is it executed?
- How do we **access** elements in a list?
- How do we **visualize** code?

What did we cover?

- How do we **control** what code is executed? If statements
- How **many times** is it executed? While loops
- How do we **access** elements in a list? lst[0] lst[1:3]
- How do we **visualize** code? environment diagrams!

Control Structures

If Statements

- Only execute the code that corresponds to the first true conditional
- If none of the conditionals are true, execute the else (if it exists)

What Would Python Do?

```
if True:
    print(“hi”)
elif True:
    print(“61A”)
else:
    print(“rocks!”)
```

```
if True:
    print(“hi”)
if True:
    print(“61A”)
else:
    print(“rocks!”)
```

hint: how does a sequence of **if** conditions behave differently from a sequence of **elif’s after an if?**

What Would Python Do?

```
if True:
    print(“hi”)
elif True:
    print(“61A”)
else:
    print(“rocks!”)
```

```
hi
```

```
if True:
    print(“hi”)
if True:
    print(“61A”)
else:
    print(“rocks!”)
```

```
hi
61A
```

What Would Python Do?

```
if True:
    return “hi”
elif True:
    return “61A”
else:
    return “rocks!”
```

```
if True:
    return “hi”
if True:
    return “61”
else:
    return “rocks!”
```

hint: how does **return** behave differently from **print**?

What Would Python Do?

```
if True:
    return “hi”
elif True:
    return “61A”
else:
    return “rocks!”
```

```
‘hi’
```

```
if True:
    return “hi”
if True:
    return “61”
else:
    return “rocks!”
```

```
‘hi’
```

1.3 #2

```
def handle_overflow(s1, s2):
    “””
    >>> handle_overflow(27, 15)
    No overflow
    >>> handle_overflow(35, 29)
    1 spot left in Section 2
    >>> handle_overflow(20, 32)
    10 spots left in Section 1
    >>> handle_overflow(35, 30)
    No space left in either section
    “””
```

1.3 #2

What **conditions** do we have?

```
def handle_overflow(s1, s2):
    """
    >>> handle_overflow(27, 15)
    No overflow
    >>> handle_overflow(35, 29)
    1 spot left in Section 2
    >>> handle_overflow(20, 32)
    10 spots left in Section 1
    >>> handle_overflow(35, 30)
    No space left in either section
    """
```

doctest

hint: use doctests to figure out how the different arguments affect what the function does

1.3 #2

What **conditions** do we have?

Look at the doctests to determine what conditions produce different results

```
def handle_overflow(s1, s2):
    """

    >>> handle_overflow(27, 15)

    No overflow

    >>> handle_overflow(35, 29)

    1 spot left in Section 2

    >>> handle_overflow(20, 32)

    10 spots left in Section 1

    >>> handle_overflow(35, 30)

    No space left in either section

    """
```

doctest

Both numbers under 30

First number (s1) larger than 30

Second number (s2) larger than 30

Both numbers larger than OR EQUAL TO 30

hint: use doctests to figure out how the different arguments affect what the function does

1.3 #2

What **do we do** for each condition?

(don’t worry about “spot” vs. “spots” yet)

```
def handle_overflow(s1, s2):
    """

    >>> handle_overflow(27, 15)

    No overflow

    >>> handle_overflow(35, 29)

    1 spot left in Section 2

    >>> handle_overflow(20, 32)

    10 spots left in Section 1

    >>> handle_overflow(35, 30)

    No space left in either section

    """
```

doctest

hint: use doctests to figure out what the different actions of the function should be

1.3 #2

What **do we do** for each condition?

(don't worry about "spot" vs. "spots" yet)

```
def handle_overflow(s1, s2):
    """

    >>> handle_overflow(27, 15)

    No overflow

    >>> handle_overflow(35, 29)

    1 spot left in Section 2

    >>> handle_overflow(20, 32)

    10 spots left in Section 1

    >>> handle_overflow(35, 30)

    No space left in either section

    """
```

doctest

Both numbers under 30
—> Print "No overflow"

First number (s1) larger than 30
—> Print "x spots left in Section s2"

Second number (s2) larger than 30
—> Print "x spots left in Section s1"

Both numbers larger than OR EQUAL TO 30
—> Print "No space left in either section"

hint: use doctests to figure out what the different actions of the function should be

1.3 #2

Putting the results of the previous slide into code, we get:

```
def handle_overflow(s1, s2):
    if s1 < 30 and s2 < 30:
        print(“No overflow”)
    elif s1 < 30:
        print(30 – s1, “spots left in Section 2”)
    elif s2 < 30:
        print(30 – s2, “spots left in Section 1”)
    else:
        print(“No space left in either section”)
```

1.3 #2

Now let’s worry about “spot” vs. “spots”

Where in the code should we differentiate between printing “spot” and “spots”?

```
def handle_overflow(s1, s2):
    if s1 < 30 and s2 < 30:
        print(“No overflow”)
    elif s1 < 30:
        print(30 - s1, “spots left in Section 2”)
    elif s2 < 30:
        print(30 - s2, “spots left in Section 1”)
    else:
        print(“No space left in either section”)
```

1.3 #2

Now let’s worry about “spot” vs. “spots”

Where in the code should we differentiate between printing “spot” and “spots”?

```
def handle_overflow(s1, s2):
    if s1 < 30 and s2 < 30:
        print(“No overflow”)
    elif s1 < 30:
        print(30 - s1, “spots left in Section 2”)
    elif s2 < 30:
        print(30 - s2, “spots left in Section 1”)
    else:
        print(“No space left in either section”)
```

So if there is only 1 spot left, we should print “spot”
Otherwise we print “spots”

1.3 #2

```
def handle_overflow(s1, s2):
    if s1 < 30 and s2 < 30:
        print(“No overflow”)
    elif s1 < 30:
        if 30 - s1 == 1:
            print(30 - s1, “spot left in Section 2”)
        else:
            print(30 - s1, “spots left in Section 2”)
    elif s2 < 30:
        if 30 - s2 == 1:
            print(30 - s1, “spot left in Section 1”)
        else:
            print(30 - s1, “spots left in Section 1”)
    else:
        print(“No space left in either section”)
```

1.5 #2

Fill in the is_prime function, which returns True if n is a prime number and False otherwise.

Hint: use the % operator

```
def is_prime(n):
```

1.5 #2

Fill in the is_prime function, which returns True if n is a prime number and False otherwise.

Hint: use the % operator

~~`def is_prime(n):`~~

Wait! Before you even think about writing code, write down what you know!

1.5 #2

Fill in the is_prime function, which returns True if n is a prime number and False otherwise.

Hint: use the % operator

~~`def is_prime(n):`~~

Wait! Before you start writing code, write down what you know!

- What are the arguments?
- What do we want to return?
- What kind of programming constructs that we learned can you use to solve this problem?

hint: before writing code, make sure you understood the problem

1.5 #2

We want to determine **whether or not n is prime**. A number is prime if its only divisors are 1 and itself.

So if dividing n by any number smaller than it produces a **non zero remainder**, then n is definitely prime.

How can we check that all numbers smaller than n will produce a non zero remainder?

How do we return **False** if we get 0 as a remainder somewhere?

How do we return **True** otherwise?

hint: if you can answer all of these questions, you are basically done with the problem

1.5 #2

Formalizing the answers the questions from the previous slide:

```
def is_prime(n):
   if n == 1:
      return False
   k = 2
   while k < n:
      if n % k == 0:
         return True
      k += 1
   return True
```

1.5 #2

Check yourself:

Why do we need the first if statement? What will happen if we start the while loop with k = 1?

Why is it ok for us to just return True after the while loop? In other words: can we ever return True on accident when n is actually prime?

1.6 #1

Implement `fizzbuzz(n)` which prints the numbers from 1 to `n` inclusive. For numbers divisible by 3, print "fizz". For numbers divisible by 5 print "buzz". For numbers divisible by both print "fizzbuzz".

```
def fizzbuzz(n):
```

1.6 #1

Implement `fizzbuzz(n)` which prints the numbers from 1 to `n` inclusive. For numbers divisible by 3, print "fizz". For numbers divisible by 5 print "buzz". For numbers divisible by both print "fizzbuzz".

~~`def fizzbuzz(n):`~~

Wait! Before you start writing code, write down what you know!

- What are the arguments?
- What do we want to return?
- What kind of programming constructs that we learned can you use to solve this problem?

1.6 #1

```
def fizzbuzz(n):
    i = 1
    while i <= n:
```

We need to print something for each number from 1 to n
So we should have a **while** loop!

1.6 #1

```
def fizzbuzz(n):
    i = 1
    while i <= n:
        if i % 3 == 0 and i % 5 == 0:
            print(‘fizzbuzz’)
        elif i % 3 == 0:
            print(‘fizz’)
        elif i % 5 == 0:
            print(‘buzz’)
        else:
```

We need to print something for each number from 1 to n
So we should have a **while** loop!

Use the modulus operator to check if a number is divisible by 3, 5, or both.

Why does the order of the if statements matter here?

1.6 #1

```
def fizzbuzz(n):
    i = 1
    while i <= n:
        if i % 3 == 0 and i % 5 == 0:
            print(‘fizzbuzz’)
        elif i % 3 == 0:
            print(‘fizz’)
        elif i % 5 == 0:
            print(‘buzz’)
        else:
            print(i)
```

We need to print something for each number from 1 to n
So we should have a **while** loop!

Use the modulus operator to check if a number is divisible by 3, 5, or both.

Why does the order of the if statements matter here?

If none of the conditions are met, just print out the number

1.6 #1

```
def fizzbuzz(n):
    i = 1
    while i <= n:
        if i % 3 == 0 and i % 5 == 0:
            print(‘fizzbuzz’)
        elif i % 3 == 0:
            print(‘fizz’)
        elif i % 5 == 0:
            print(‘buzz’)
        else:
            print(i)
        i += 1
```

We need to print something for each number from 1 to n
So we should have a **while** loop!

Use the modulus operator to check if a number is divisible by 3, 5, or both.

Why does the order of the if statements matter here?

If none of the conditions are met, just print out the number

Don’t forget to increment i each time!

Lists and For Statements

2.1 Example

```
>>> pizza = [1, 2, 3, 4]

>>> pizza[1:2]
```

2.1 Example

```
>>> pizza = [1, 2, 3, 4]

>>> pizza[1:2]
```

Think of this as getting the elements of pizza that are from index 1 to index 2, not including index 2 — [1, 2)

```
[2]
```

Note: this returns the list [2], not just the number 2

2.1 Example

```
>>> pizza = [1, 2, 3, 4]

>>> pizza[1:2]
```

Think of this as getting the elements of pizza that are from index 1 to index 2, not including index 2 — [1, 2)

```
[2]
```

Note: this returns the list [2], not just the number 2

```
>>> pizza[1:]
```

2.1 Example

```
>>> pizza = [1, 2, 3, 4]

>>> pizza[1:2]
```

Think of this as getting the elements of pizza that are from index 1 to index 2, not including index 2 — [1, 2)

```
[2]
```

Note: this returns the list [2], not just the number 2

```
>>> pizza[1:]
```

Not specifying the last index means “till the end of the list”

```
[2, 3, 4]
```

2.1 Example

```
>>> pizza = [1, 2, 3, 4]

>>> pizza[1:2]
```

Think of this as getting the elements of pizza that are from index 1 to index 2, not including index 2 — [1, 2)

```
[2]
```

Note: this returns the list [2], not just the number 2

```
>>> pizza[1:]
[2, 3, 4]
```

Not specifying the last index means "till the end of the list"

```
>>> pizza[-2:3]
```

2.1 Example

```
>>> pizza = [1, 2, 3, 4]
```

```
>>> pizza[1:2]
```

Think of this as getting the elements of pizza that are from index 1 to index 2, not including index 2 — [1, 2)

```
[2]
```

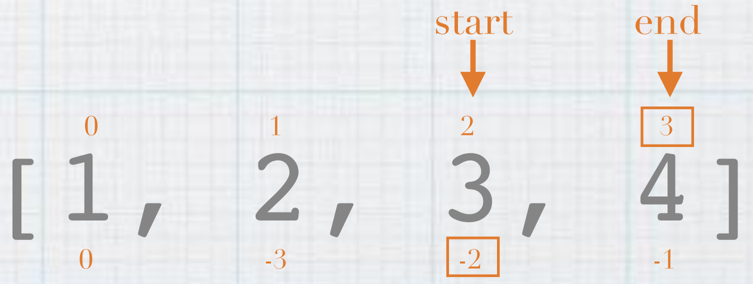
Note: this returns the list [2], not just the number 2

```
>>> pizza[1:]
```

Not specifying the last index means "till the end of the list"

```
[2, 3, 4]
```

```
>>> pizza[-2:3]
[3]
```



Find the start and end indices and return everything between them except for the last element

Environment Diagrams

There are 3 types of things you should be able to draw out

There are 3 types of things you should be able to draw out

bob = 3

ASSIGNMENT

1. Evaluate the RHS
2. Write the name and value in the **current frame**

I like to keep track of the **current frame** up here → CF: G

Global Frame:

There are 3 types of things you should be able to draw out

bob = 3

ASSIGNMENT

1. Evaluate the RHS
2. Write the name and value in the **current frame**

I like to keep track of the **current frame** up here → CF: G

Global Frame:

bob: 3

There are 3 types of things you should be able to draw out

```
bob = 3
```

ASSIGNMENT

1. Evaluate the RHS
2. Write the name and value in the **current frame**

I like to keep track of the **current frame** up here → CF: G

```
def rob(bob):
    a = 2
    return 'mob'
```

DEF STATEMENTS

1. Write the function name in the current frame
2. Point it to the function object which we represent by the function signature and parent

Global Frame:

bob: 3

There are 3 types of things you should be able to draw out

```
bob = 3
```

ASSIGNMENT

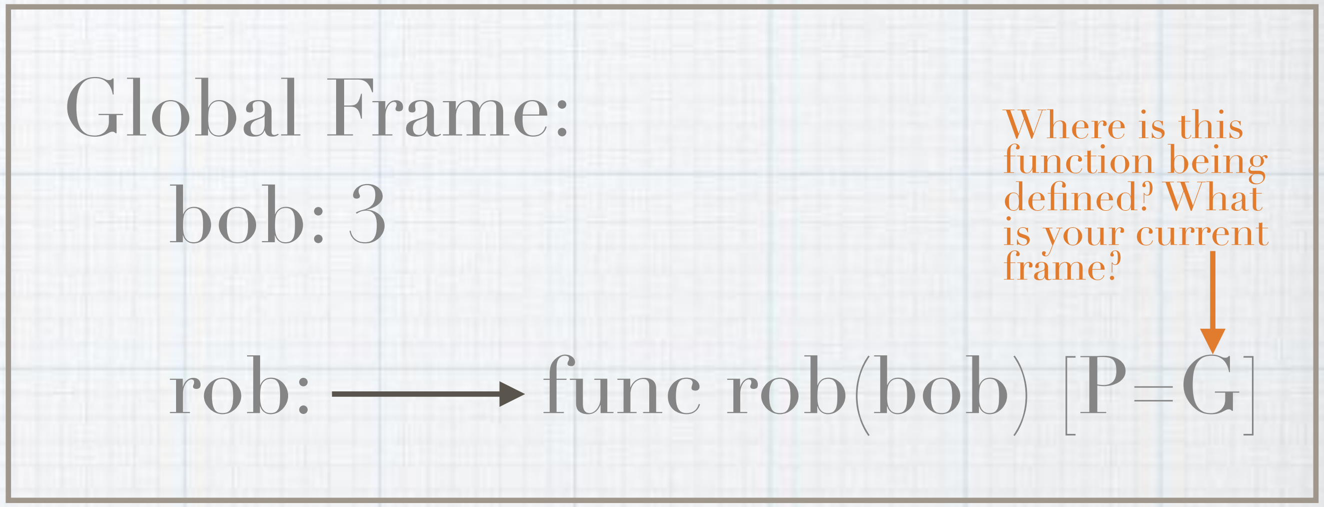
1. Evaluate the RHS
2. Write the name and value in the **current frame**

```
def rob(bob):
    a = 2
    return 'mob'
```

DEF STATEMENTS

1. Write the function name in the current frame
2. Point it to the function object which we represent by the function signature and parent

I like to keep track of the **current frame** up here → CF: G



Global Frame:

bob: 3

rob: → func rob(bob) [P=G]

Where is this function being defined? What is your current frame?

There are 3 types of things you should be able to draw out

```
bob = 3
```

ASSIGNMENT

1. Evaluate the RHS
2. Write the name and value in the **current frame**

```
def rob(bob):
    a = 2
    return 'mob'
```

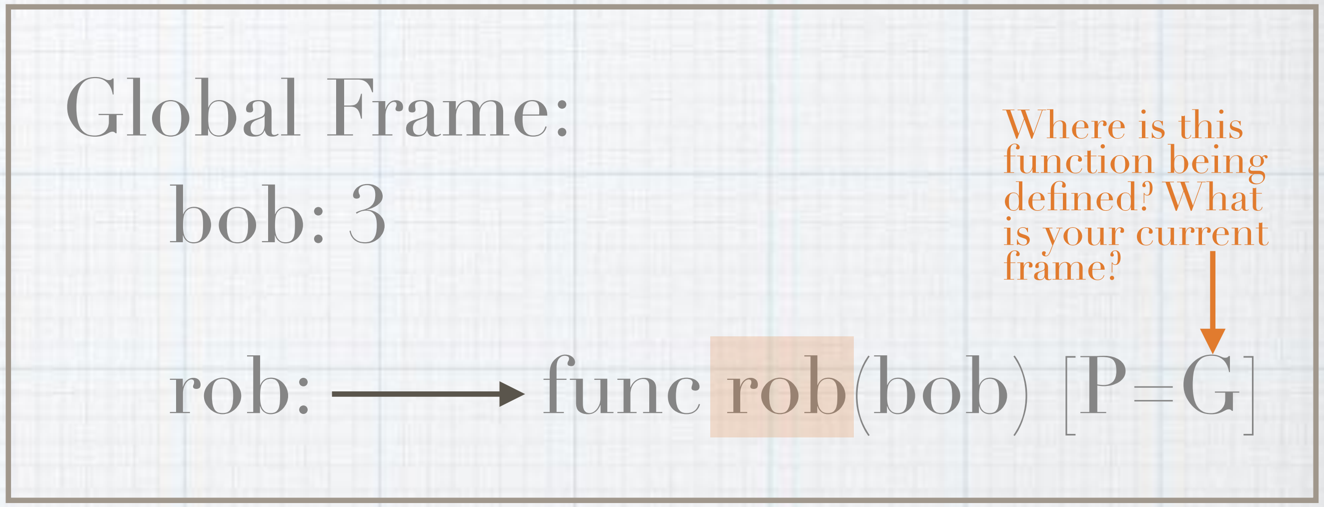
DEF STATEMENTS

1. Write the function name in the current frame
2. Point it to the function object which we represent by the function signature and parent

```
bob = rob
```

What will this **ASSIGNMENT** do?

I like to keep track of the **current frame** up here → CF: G



Global Frame:

bob: 3

rob: → func rob(bob) [P=G]

Where is this function being defined? What is your current frame?

There are 3 types of things you should be able to draw out

```
bob = 3
```

ASSIGNMENT
1. Evaluate the RHS
2. Write the name and value in the **current frame**

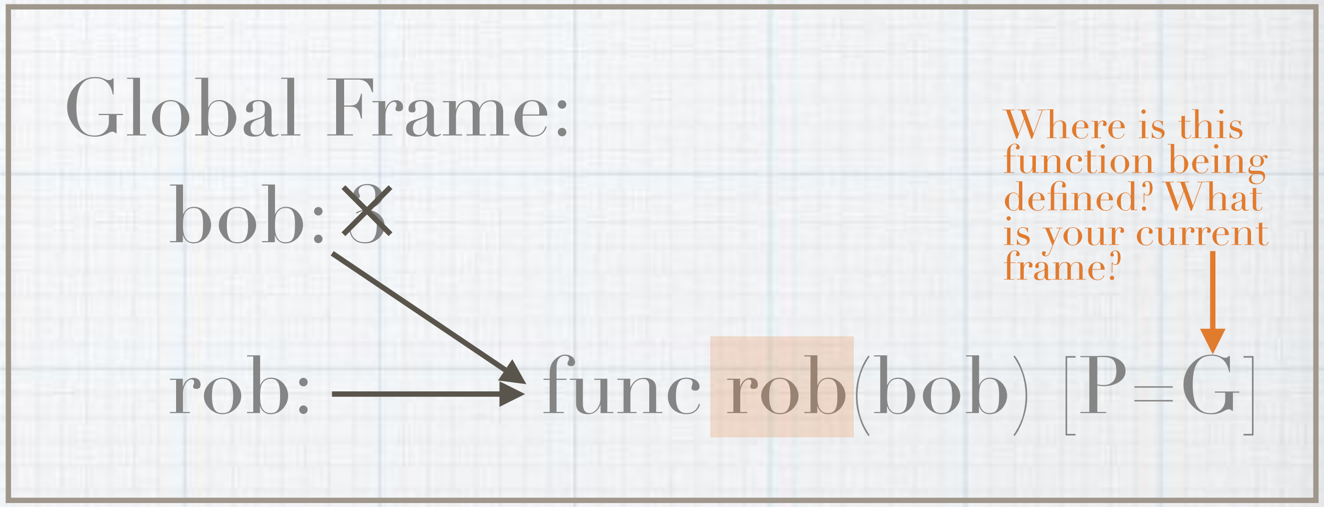
```
def rob(bob):
    a = 2
    return 'mob'
```

DEF STATEMENTS
1. Write the function name in the current frame
2. Point it to the function object which we represent by the function signature + parent

```
bob = rob
```

What will this ASSIGNMENT do?

I like to keep track of the **current frame** up here → CF: G



Global Frame:

bob: ~~3~~

rob:

func rob(bob) [P=G]

Where is this function being defined? What is your current frame?

There are 3 types of things you should be able to draw out

```
bob = 3
```

ASSIGNMENT
1. Evaluate the RHS
2. Write the name and value in the **current frame**

```
def rob(bob):
    a = 2
    return 'mob'
```

DEF STATEMENTS
1. Write the function name in the current frame
2. Point it to the function object which we represent by the function signature + parent

```
bob = rob
```

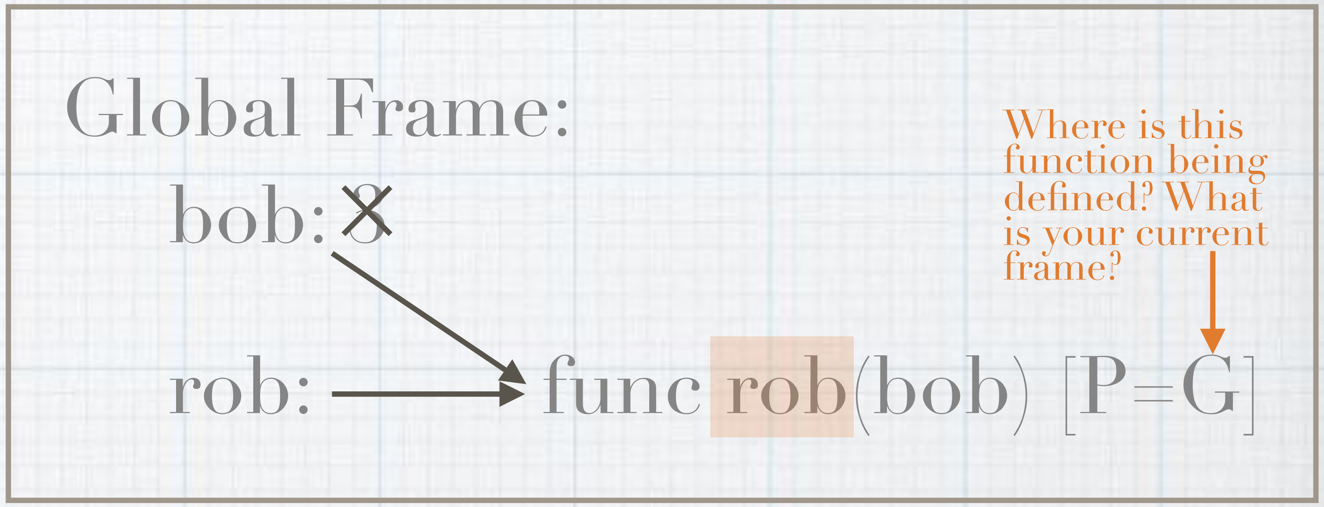
What will this ASSIGNMENT do?

```
bob = bob(bob)
```

bob points to the function rob in the global frame, so we call the rob function

FUNCTION CALLS
1. Evaluate the operator and operand
2. Open a new frame
   Write f#: function name [P = ???]
   (optional; update your current frame in CF:)
   Assign the parameters
3. Execute the body of the function

I like to keep track of the **current frame** up here → CF: G



Where is this function being defined? What is your current frame?

There are 3 types of things you should be able to draw out

```
bob = 3
```

ASSIGNMENT
1. Evaluate the RHS
2. Write the name and value in the **current frame**

```
def rob(bob):
    a = 2
    return 'mob'
```

DEF STATEMENTS
1. Write the function name in the current frame
2. Point it to the function object which we represent by the function signature + parent

```
bob = rob
```

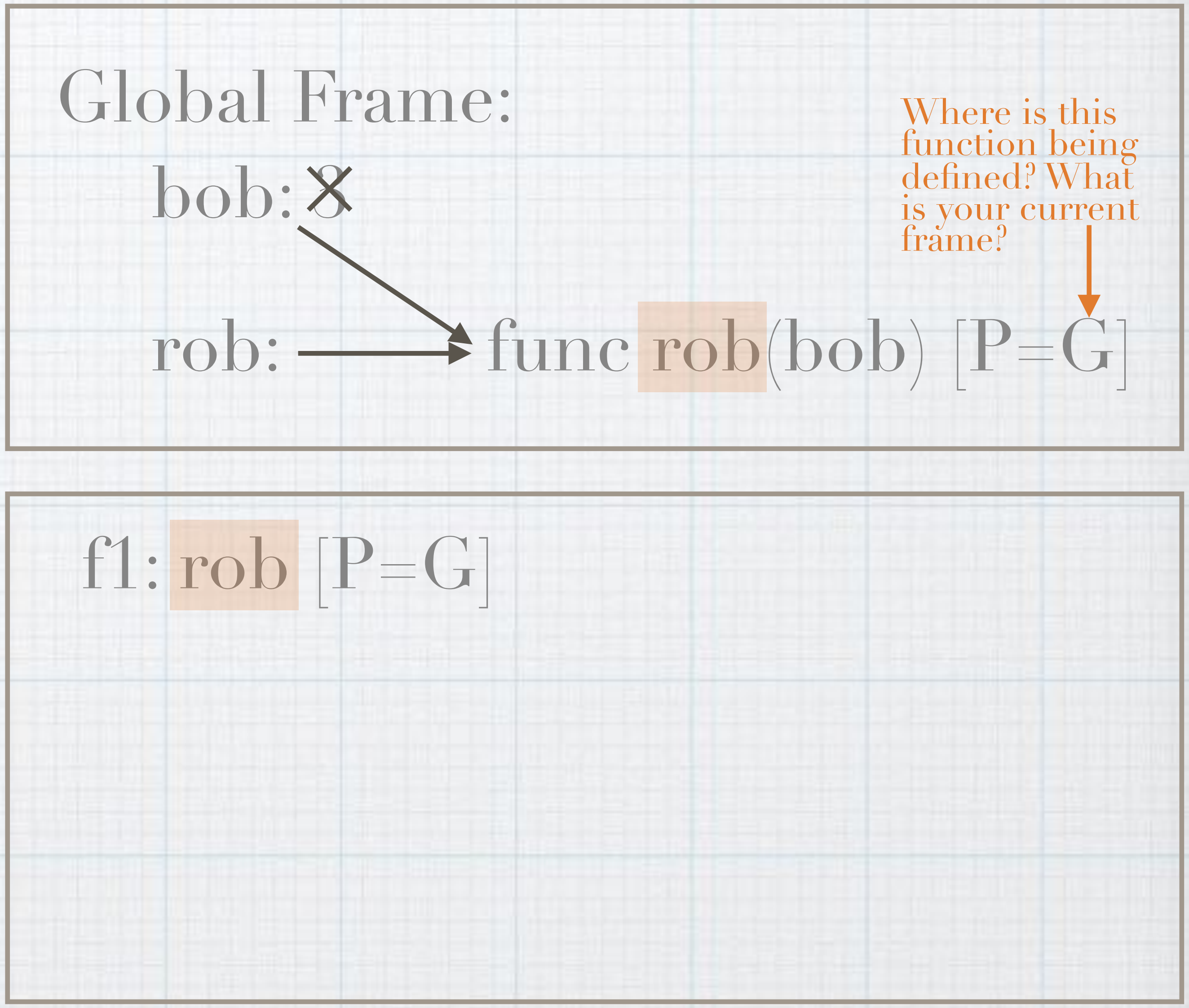
What will this ASSIGNMENT do?

```
bob = bob(bob)
```

bob points to the function rob in the global frame, so we call the rob function

FUNCTION CALLS
1. Evaluate the operator and operand
2. Open a new frame
   Write f#: function name [P = ???]
   (optional; update your current frame in CF:)
   Assign the parameters
3. Execute the body of the function

I like to keep track of the **current frame** up here → CF: G, $\boxed{\text{f1}}$


Global Frame:
bob: ~~3~~
rob:
func rob(bob) [P=G]
Where is this function being defined? What is your current frame?
f1: rob [P=G]

There are 3 types of things you should be able to draw out

```
bob = 3
```

ASSIGNMENT
1. Evaluate the RHS
2. Write the name and value in the **current frame**

```
def rob(bob):
    a = 2
    return 'mob'
```

DEF STATEMENTS
1. Write the function name in the current frame
2. Point it to the function object which we represent by the function signature + parent

```
bob = rob
```

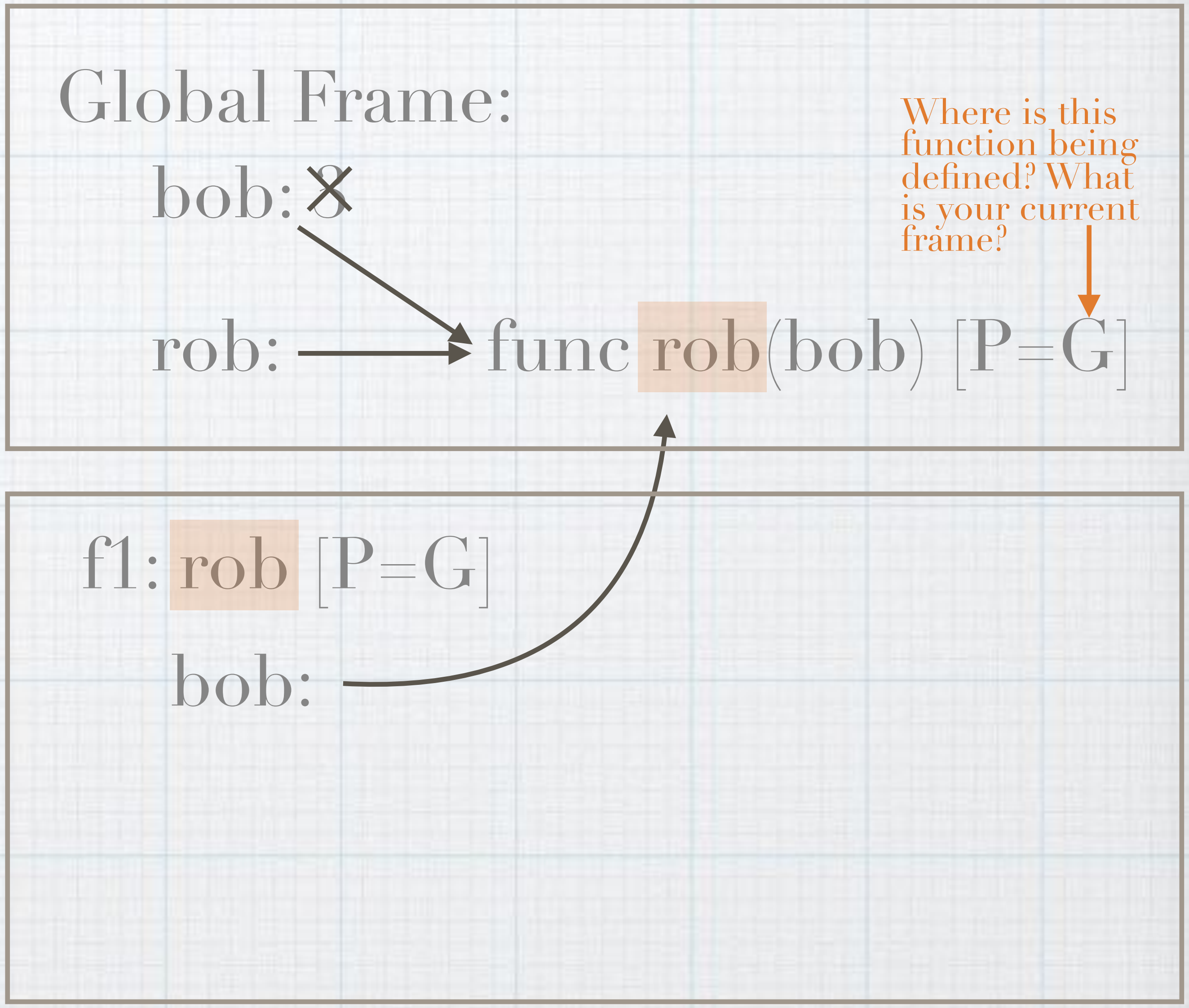
What will this ASSIGNMENT do?

```
bob = bob(bob)
```

bob points to the function rob in the global frame, so we call the rob function

FUNCTION CALLS
1. Evaluate the operator and operand
2. Open a new frame
   Write f#: function name [P = ???]
   (optional; update your current frame in CF:)
   Assign the parameters
3. Execute the body of the function

I like to keep track of the **current frame** up here → CF: G, f1



Global Frame:

bob: ~~3~~

rob: → func rob(bob) [P=G]

Where is this function being defined? What is your current frame?

f1: rob [P=G]

bob:

There are 3 types of things you should be able to draw out

I like to keep track of the **current frame** up here → CF: G, ~~f1~~

cross out a frame when you **return**

```
bob = 3
```

ASSIGNMENT

1. Evaluate the RHS
2. Write the name and value in the **current frame**

```
def rob(bob):
    a = 2
    return 'mob'
```

DEF STATEMENTS

1. Write the function name in the current frame
2. Point it to the function object which we represent by the function signature + parent

```
bob = rob
```

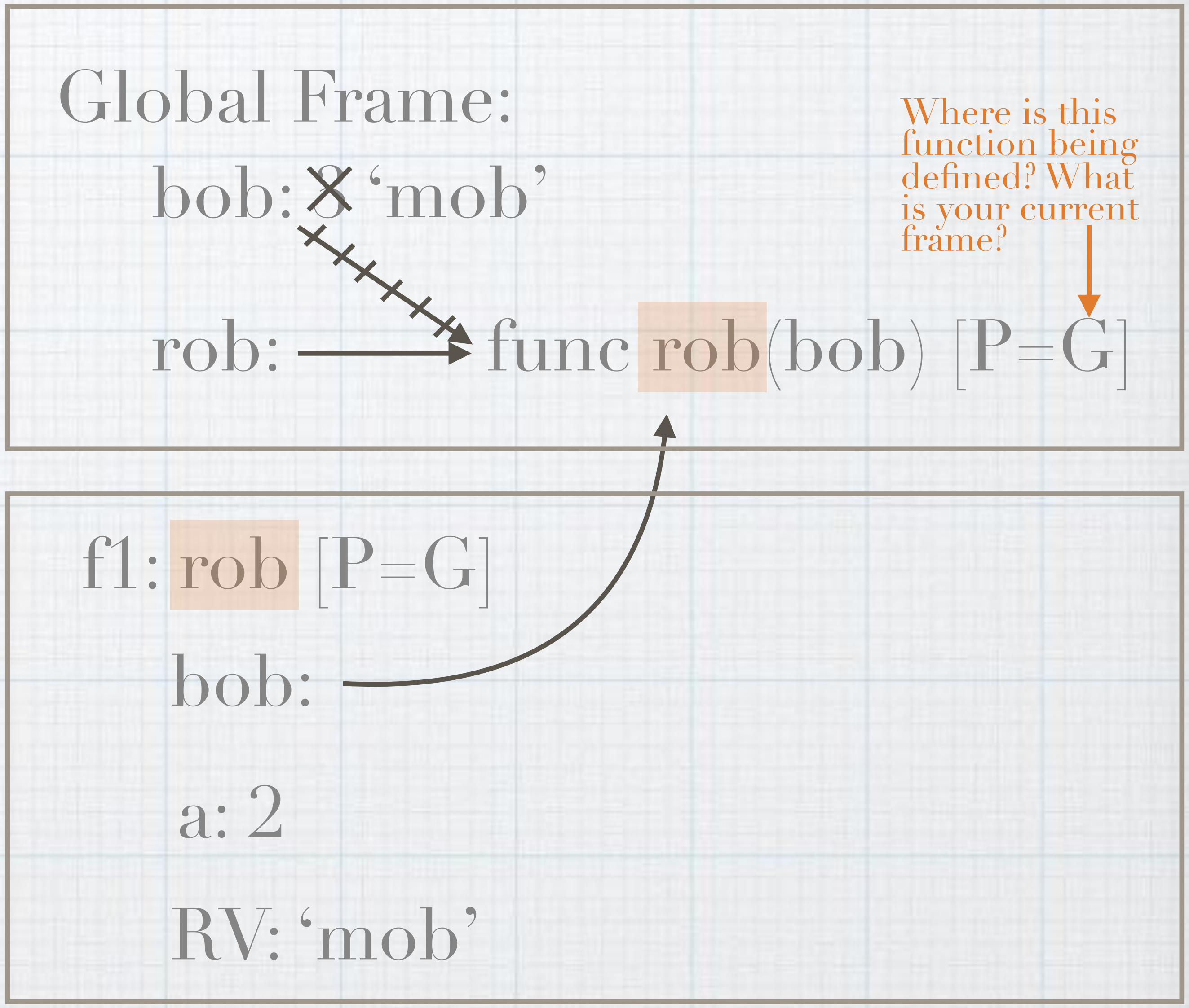
What will this ASSIGNMENT do?

```
bob = bob(bob)
```

bob points to the function rob in the global frame, so we call the rob function

FUNCTION CALLS

1. Evaluate the operator and operand
2. Open a new frame
   Write f#: function name [P = ???]
   (optional; update your current frame in CF:)
   Assign the parameters
3. Execute the body of the function



Global Frame:

bob: ~~3~~ 'mob'

rob: → func rob(bob) [P=G]

Where is this function being defined? What is your current frame?

f1: rob [P=G]

bob:

a: 2

RV: 'mob'

Diagram Rules

ASSIGNMENT

1. Evaluate the RHS
2. Write the name and value in the **current frame**

DEF STATEMENTS

1. Write the function name in the current frame
2. Point it to the function object which we represent by the function signature + parent

FUNCTION CALLS

1. Evaluate the operator and operand
2. Open a new frame

   Write f#: function name [P = ???]

   (optional; update your current frame in CF:)

   Assign the parameters
3. Execute the body of the function

3.1 #1

```
a = 1
def b(b):
    return a + b
a = b(a)
a = b(a)
```

tip: take it a line at a time

3.1 #1

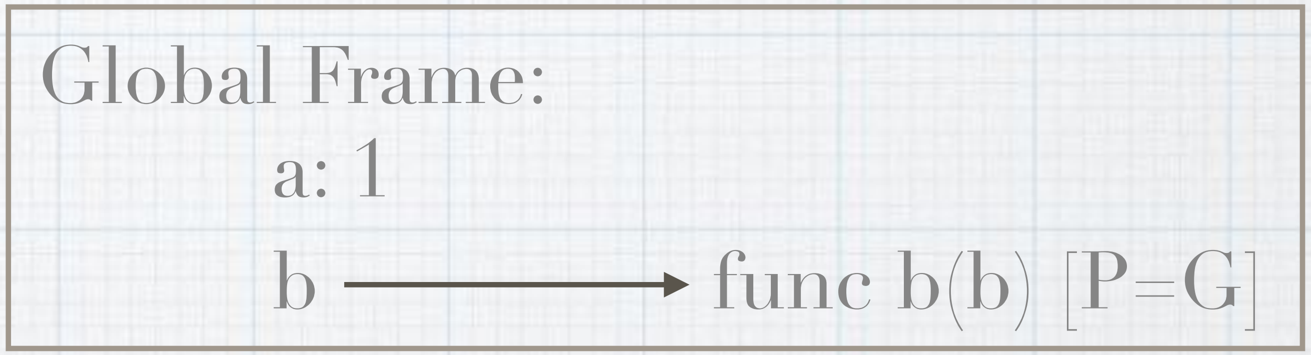
Just executed the first two lines

```
a = 1
def b(b):
        return a + b
a = b(a)
a = b(a)
```

This is an assignment. To find the value of the RHS we need to do a function call.

Before opening a new frame, make sure you know what the values of the operator and operands are (here a is 1 since that is it’s value in the **global frame**)

CF: G



Global Frame:

a: 1

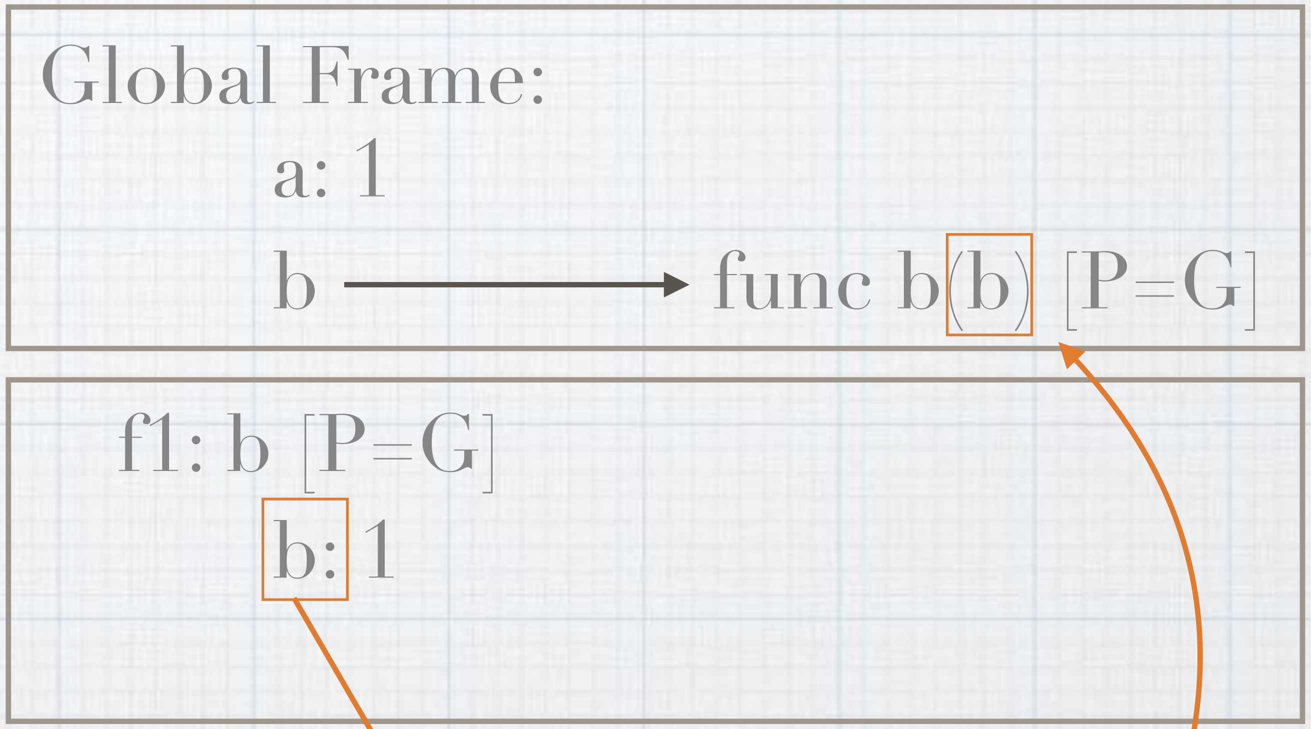
b → func b(b) [P=G]

tip: take it a line at a time

3.1 #1

```
➜ a = 1
➜ def b(b):
        return a + b
➜ a = b(a)
  a = b(a)
```

CF: G, f1



Global Frame:

a: 1

b → func b(b) [P=G]

f1: b [P=G]

b: 1

To evaluate the body of the function, we need to do a + b. Since there is no a defined in f1 (the **current frame**) we must look for a in it's parent

Note: The parameter is always just copied from the function signature up here. Even though we pass in a, we do not write a as the name of the parameter.

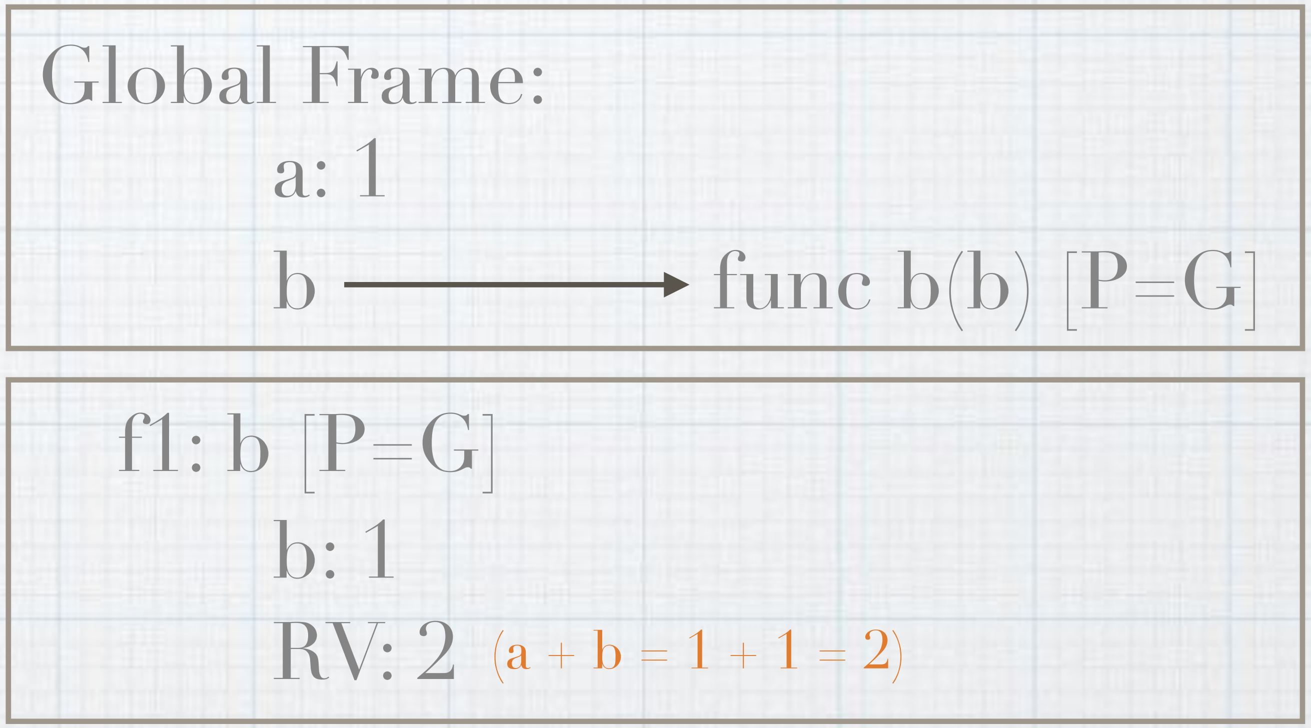
tip: take it a line at a time

3.1 #1

```
a = 1
def b(b):
      return a + b
a = b(a)
a = b(a)
```

To evaluate the body of the function, we need to do a + b. Since there is no a defined in f1 (the **current frame**) we must look for a in it's parent

CF: G, ~~f1~~



Global Frame:

a: 1

b → func b(b) [P=G]

f1: b [P=G]

b: 1

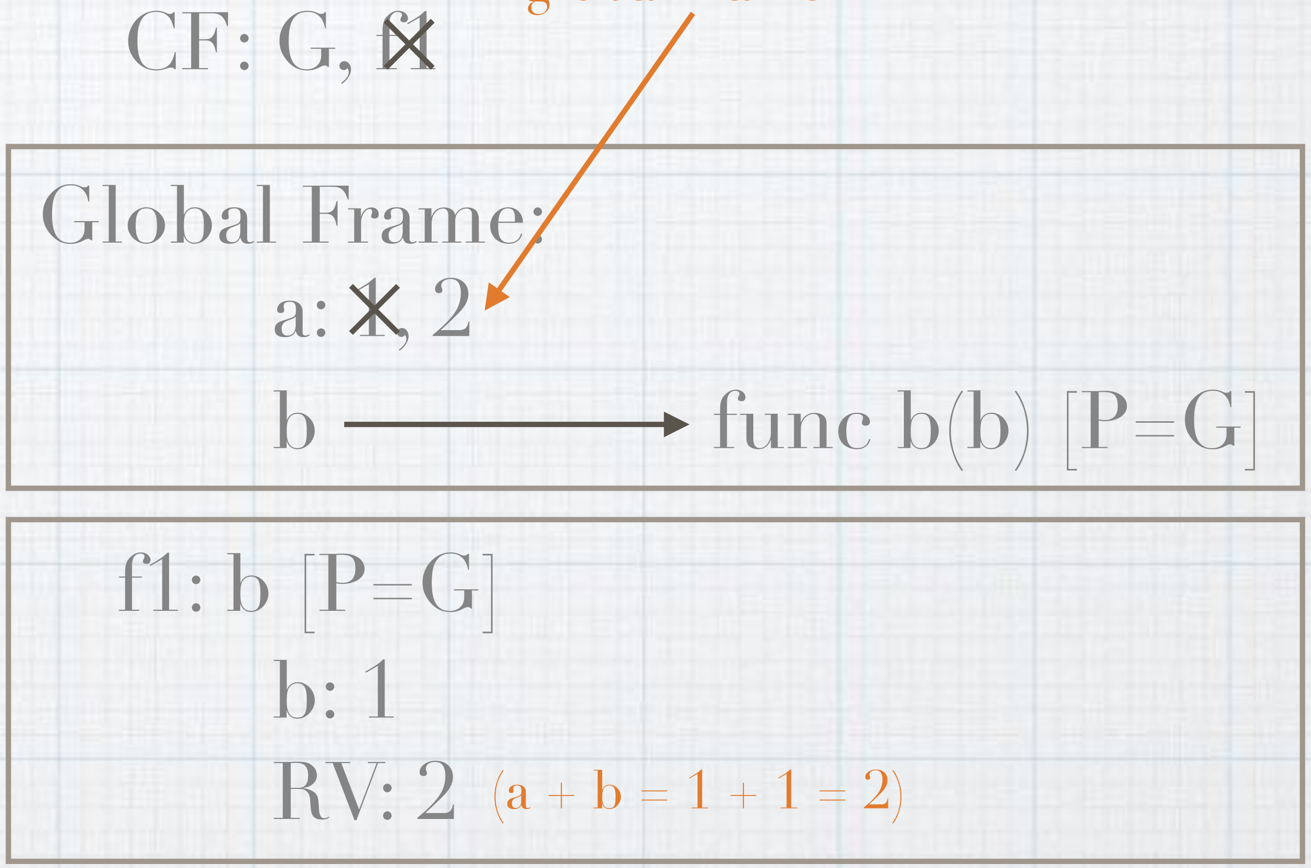
RV: 2 (a + b = 1 + 1 = 2)

tip: take it a line at a time

3.1 #1

Now we are finally ready to do the assignment. We know that b(a) evaluates to 2 (since this is the return value of f1) and we can reassign a to be 2 in the global frame

```
→ a = 1
→ def b(b):
      return a + b
→ a = b(a)
  a = b(a)
```



CF: G, ~~f1~~

Global Frame:

a: ~~1~~, 2

b → func b(b) [P=G]

f1: b [P=G]

b: 1

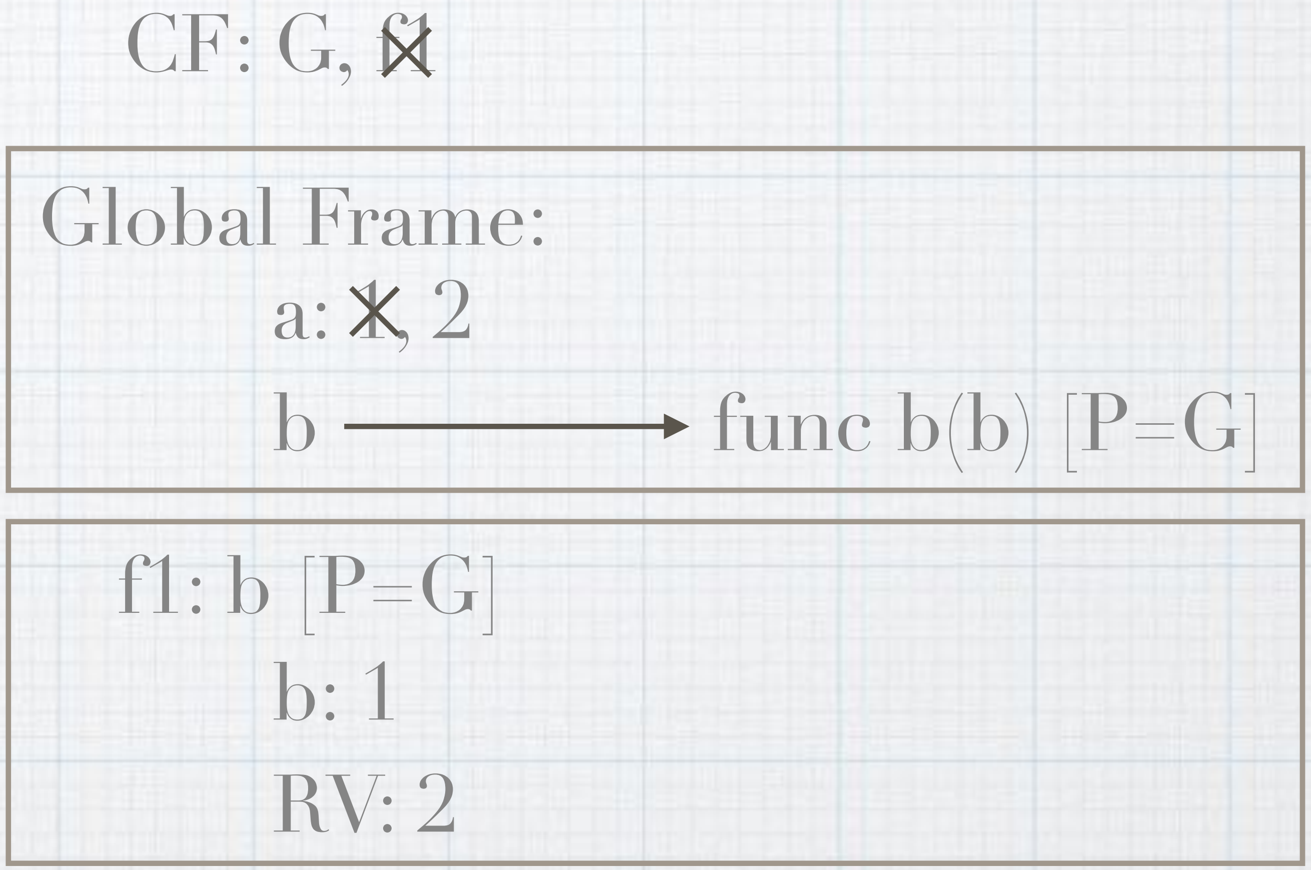
RV: 2 (a + b = 1 + 1 = 2)

tip: take it a line at a time

3.1 #1

```
a = 1
def b(b):
    return a + b
a = b(a)
a = b(a)
```

Another assignment and function call



CF: G, ~~f1~~

Global Frame:

a: ~~1~~, 2

b → func b(b) [P=G]

f1: b [P=G]

b: 1

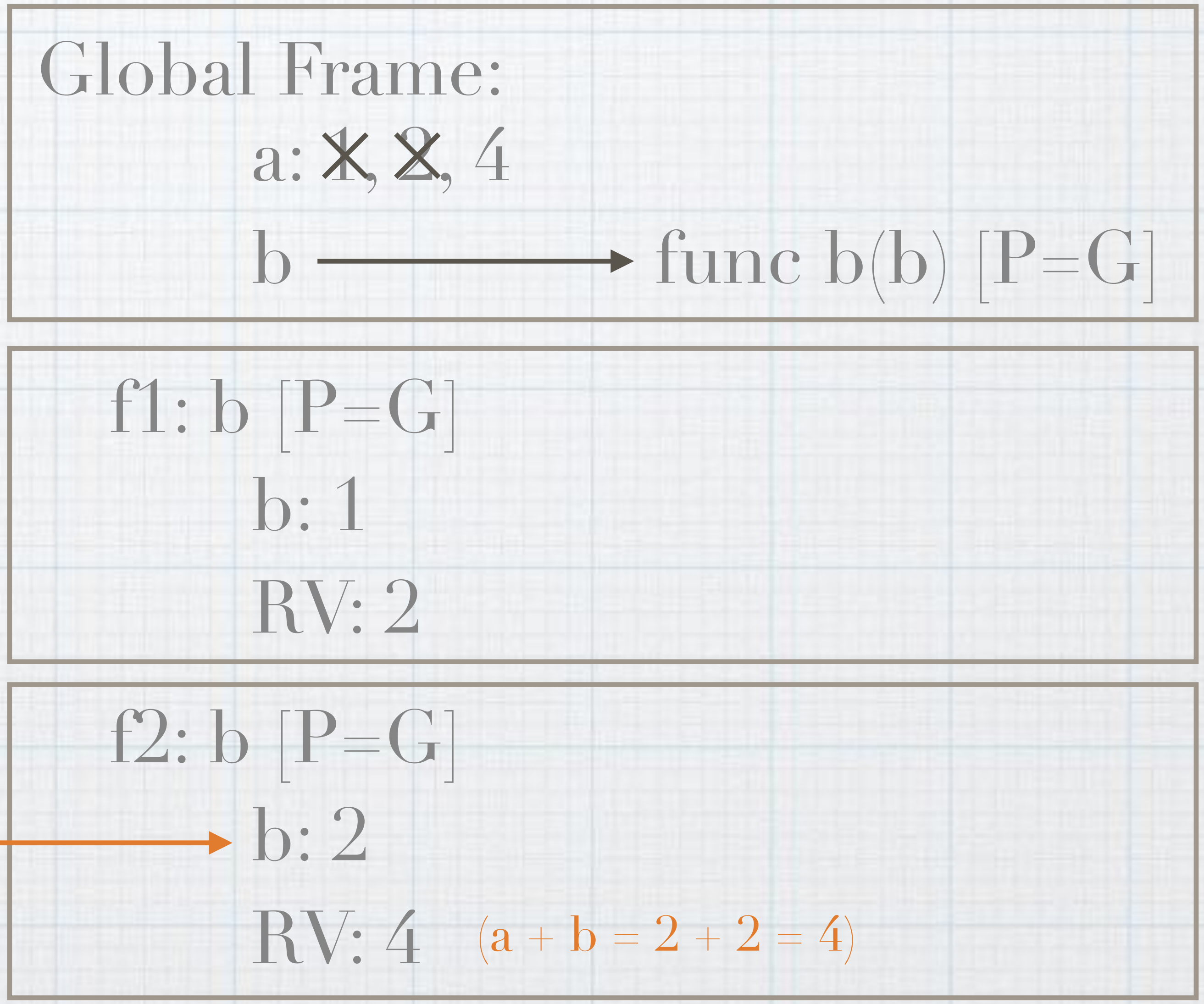
RV: 2

tip: take it a line at a time

3.1 #1

```
→ a = 1
→ def b(b):
        return a + b
→ a = b(a)
→ a = b(a)
```

CF: G, ~~f1~~, ~~f2~~



Global Frame:

a: ~~1~~, ~~2~~, 4

b → func b(b) [P=G]

f1: b [P=G]

b: 1

RV: 2

f2: b [P=G]

b: 2

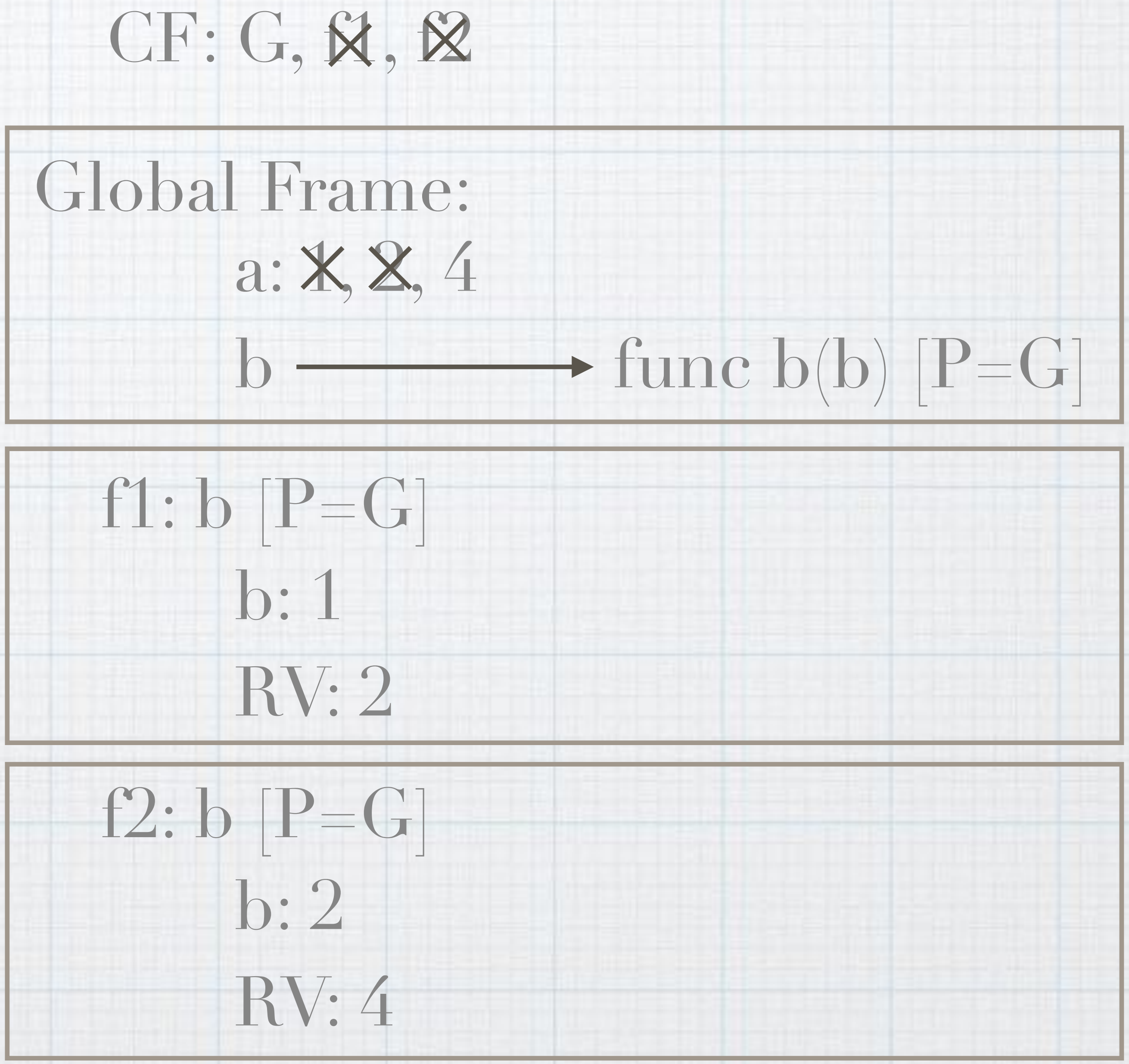
RV: 4 (a + b = 2 + 2 = 4)

Notice that b is 2 here now, since the global a has changed

tip: take it a line at a time

3.1 #1

```
a = 1
def b(b):
    return a + b
a = b(a)
a = b(a)
```



CF: G, ~~f1~~, ~~f2~~

Global Frame:

a: ~~1~~, ~~2~~, 4

b → func b(b) [P=G]

f1: b [P=G]

b: 1

RV: 2

f2: b [P=G]

b: 2

RV: 4

Make sure that every frame has a return value!

tip: take it a line at a time

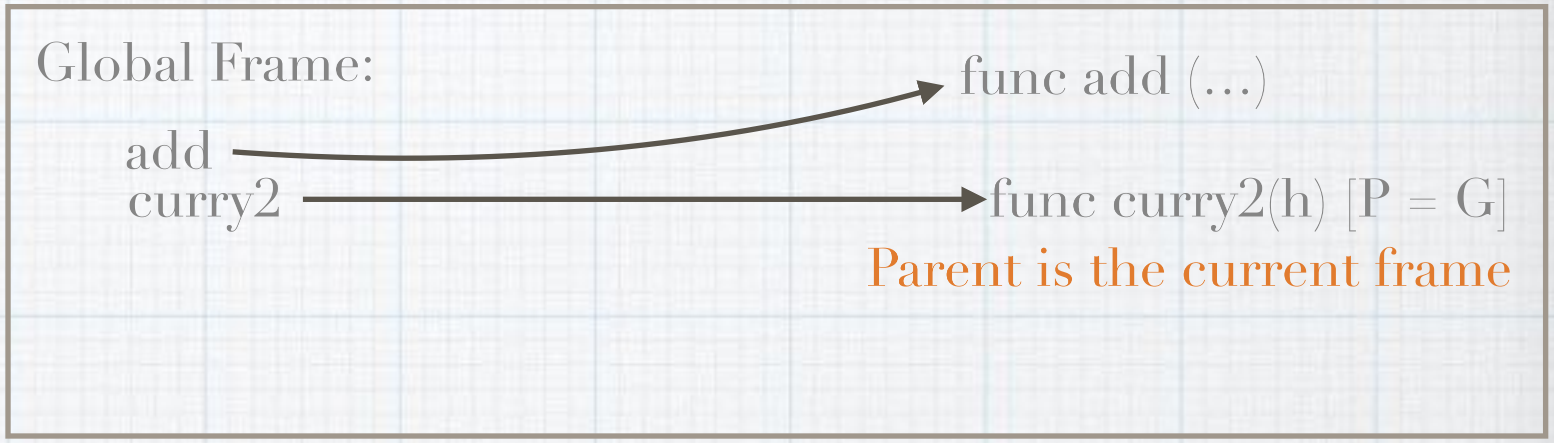
3.1 #2

```
def curry2(h):
    def f(x):
        def g(y):
            return h(x, y)
        return g
    return f
make_adder = curry2(add)
add_three = make_adder(3)
five = add_three(2)
```

tip: when you start doing a function call, remember where you were before

3.1 #2

CF: G



Global Frame:

add → func add (...)

curry2 → func curry2(h) [P = G]

Parent is the current frame

I like to draw a line here so that I don't accidentally start evaluating the body of the def right away

```
def curry2(h):

    def f(x):

        def g(y):

            return h(x, y)

        return g

    return f

make_adder = curry2(add)

add_three = make_adder(3)

five = add_three(2)
```

tip: when you start doing a function call, remember where you were before

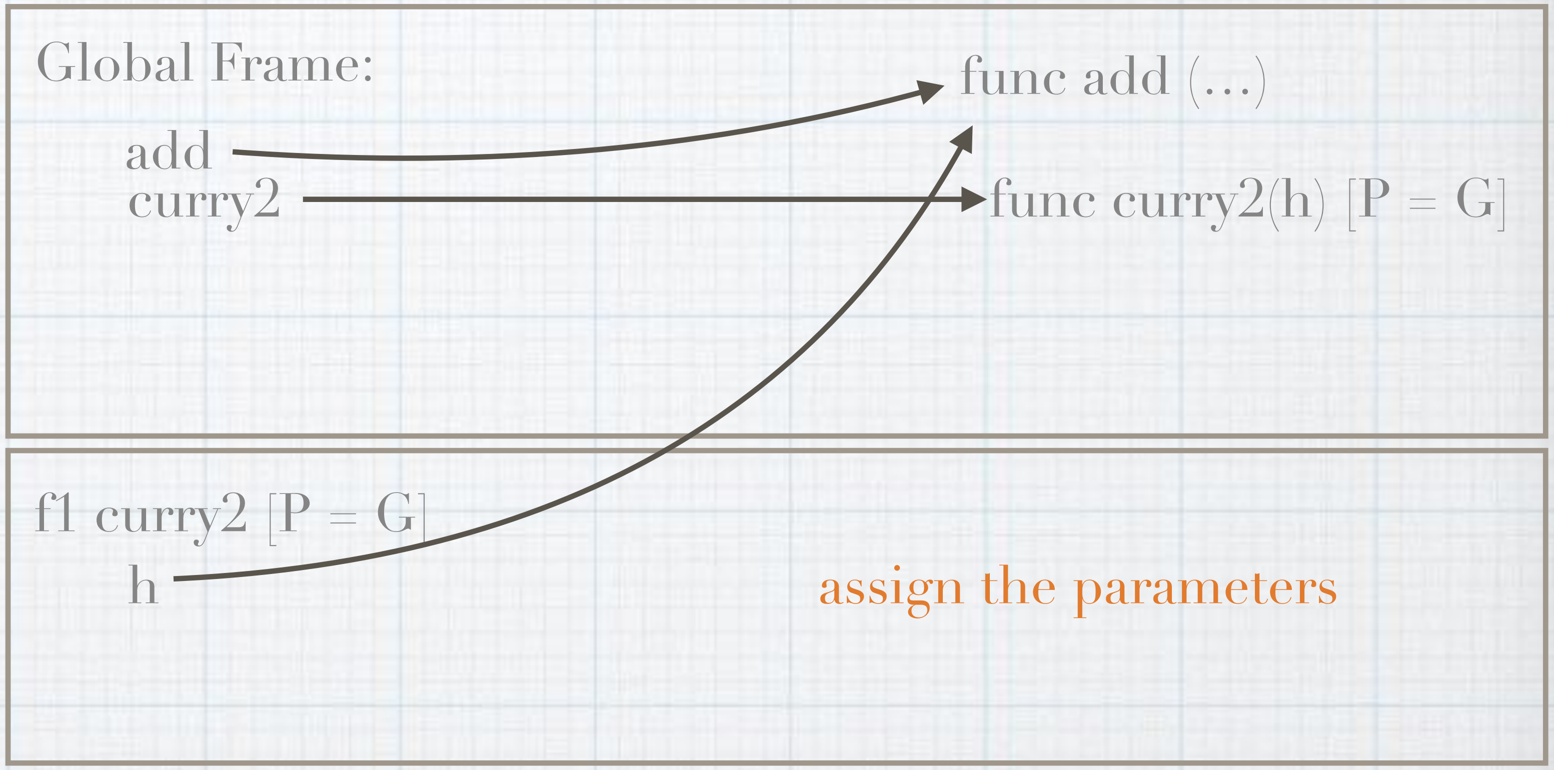
3.1 #2

CF: G, f1

I like to draw a line here so that I don't accidentally start evaluating the body of the def right away

```
def curry2(h):
    def f(x):
        def g(y):
            return h(x, y)
        return g
    return f
make_adder = curry2(add)
add_three = make_adder(3)
five = add_three(2)
```

function call



assign the parameters

Recall function calls:

1. Evaluate operator and operands,
2. Create a new frame
3. Assign the parameters in the new frame

tip: when you start doing a function call, remember where you were before

3.1 #2

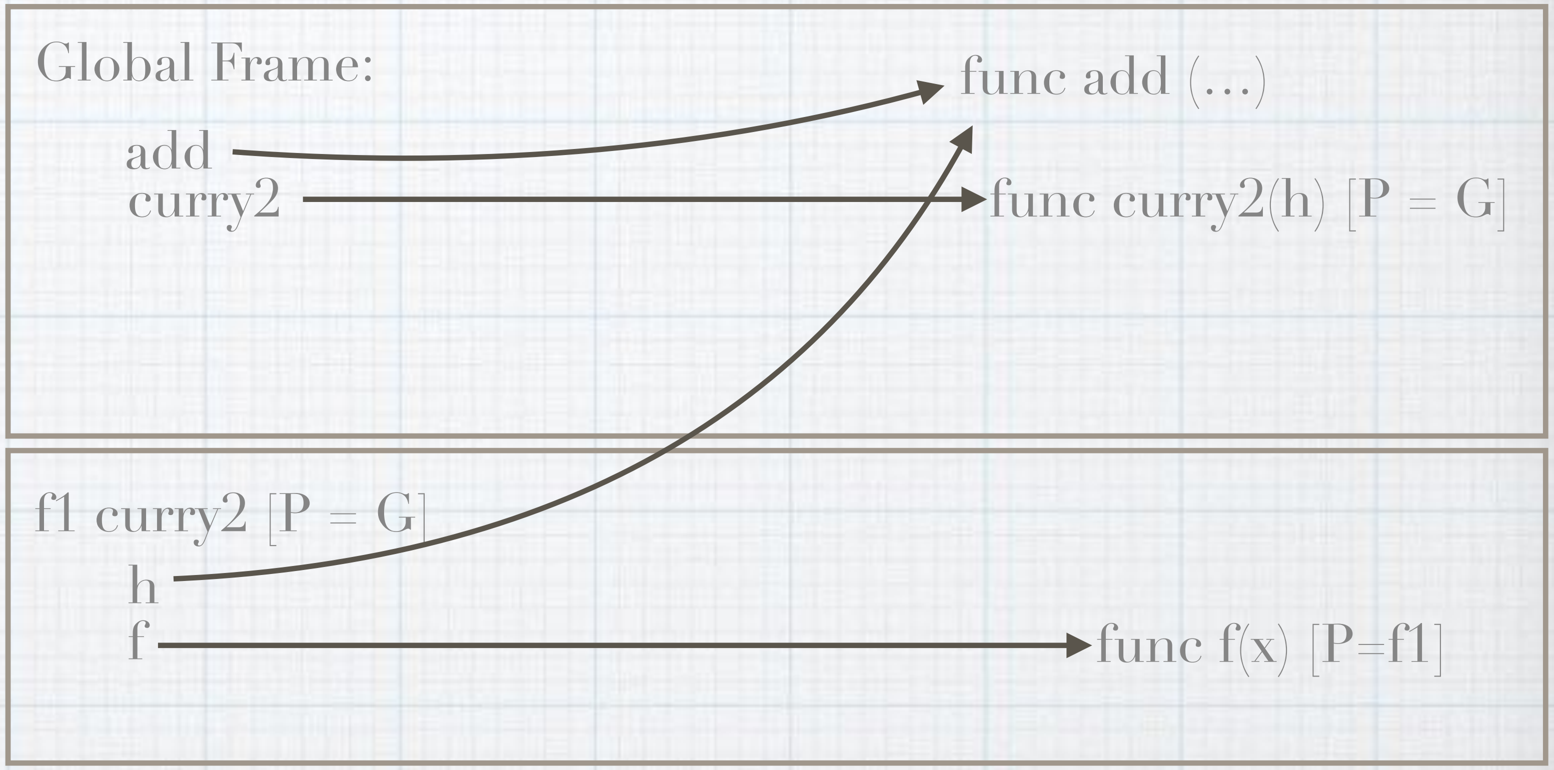
CF: G, f1

I like to draw a line
here so that I don't
accidentally start
evaluating the body
of the def right
away

```
def curry2(h):
    def f(x):
        def g(y):
            return h(x, y)
        return g
    return f
make_adder = curry2(add)
add_three = make_adder(3)
five = add_three(2)
```

Inside curry2 we define a new function, f.
What is it's parent?

function call



Global Frame:

add → func add (...)

curry2 → func curry2(h) [P = G]

f1 curry2 [P = G]

h

f → func f(x) [P=f1]

Recall function calls:

1. Evaluate operator and operands,
2. Create a new frame
3. Assign the parameters in the new frame

tip: when you start doing a function call, remember where you were before

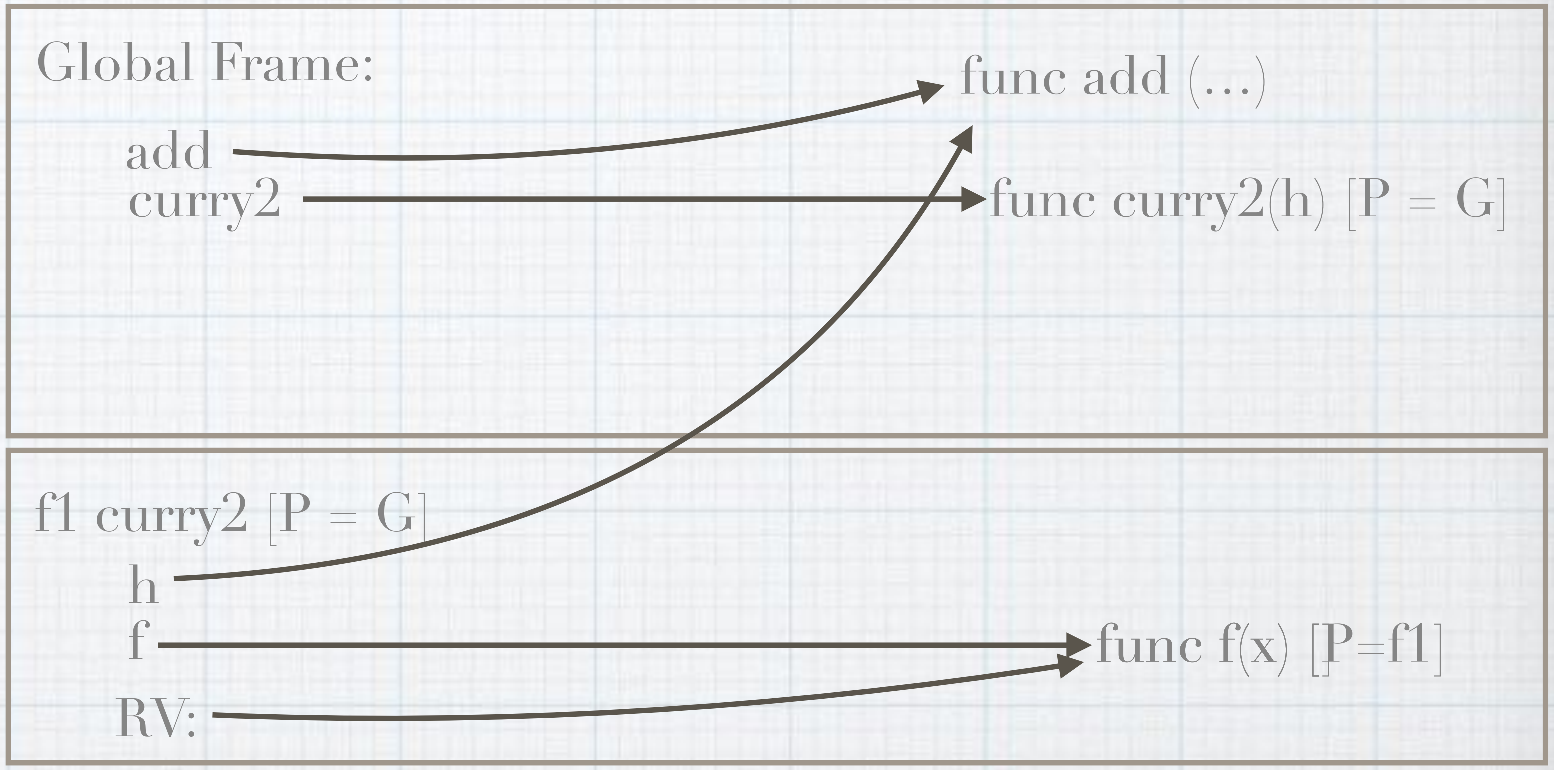
3.1 #2

CF: G, ~~f1~~

I like to draw a line here so that I don't accidentally start evaluating the body of the def right away

```
def curry2(h):
    def f(x):
        def g(y):
            return h(x, y)
        return g
    return f
make_adder = curry2(add)
add_three = make_adder(3)
five = add_three(2)
```

function call


Global Frame:

add

curry2

func add (...)

func curry2(h) [P = G]

f1 curry2 [P = G]

h

f

RV:

func f(x) [P=f1]

now we return the function we just defined

tip: when you start doing a function call, remember where you were before

3.1 #2

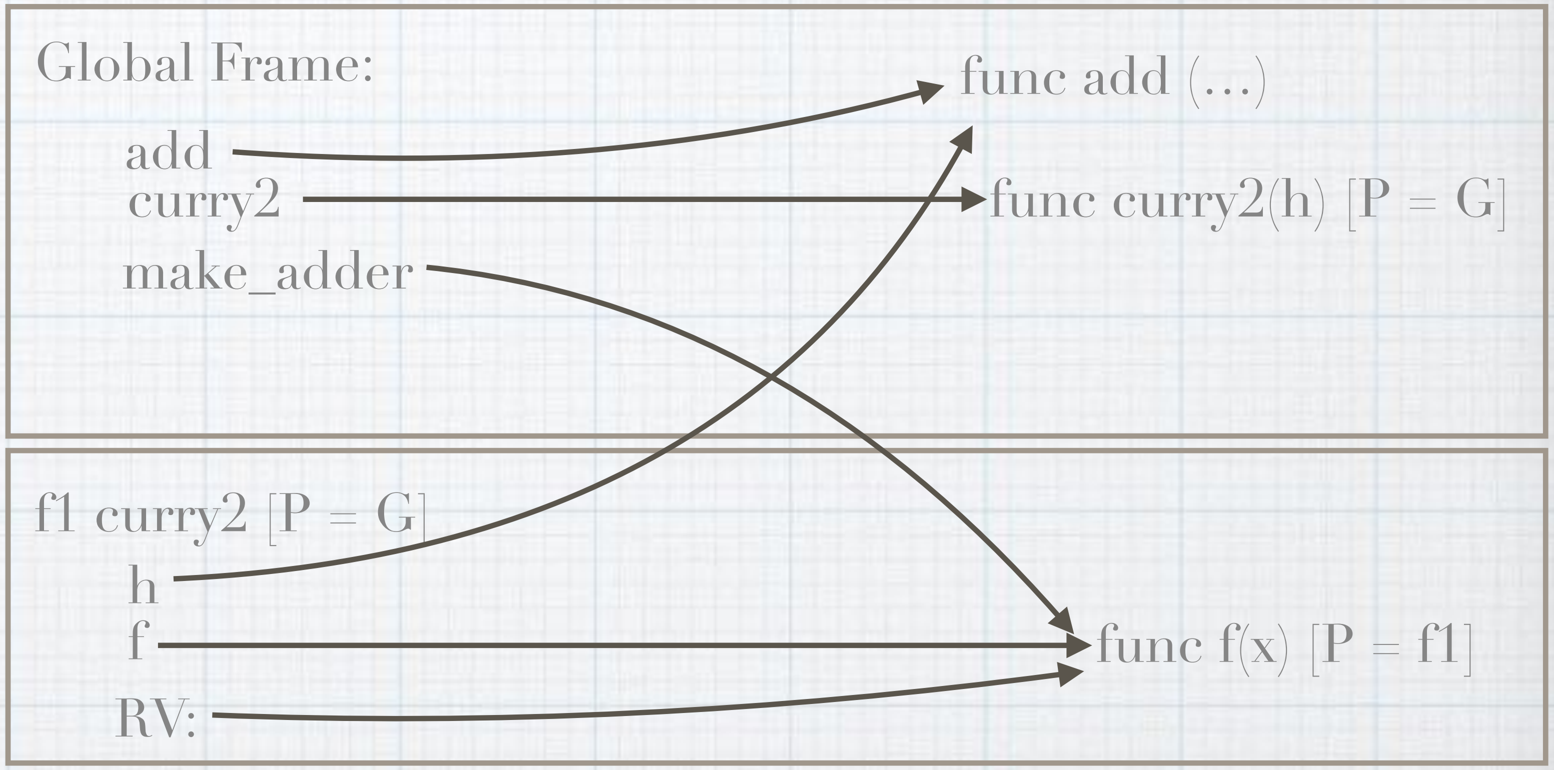
CF: G, ~~f1~~

I like to draw a line here so that I don't accidentally start evaluating the body of the def right away

```
def curry2(h):
    def f(x):
        def g(y):
            return h(x, y)
        return g
    return f
make_adder = curry2(add)
add_three = make_adder(3)
five = add_three(2)
```

assignment

function call



Global Frame:

add → func add (...)

curry2 → func curry2(h) [P = G]

make_adder → func f(x) [P = f1]

f1 curry2 [P = G]

h → func add (...)

f → func f(x) [P = f1]

RV: → func f(x) [P = f1]

finally assign the value that curry2 returned to make_adder

tip: when you start doing a function call, remember where you were before

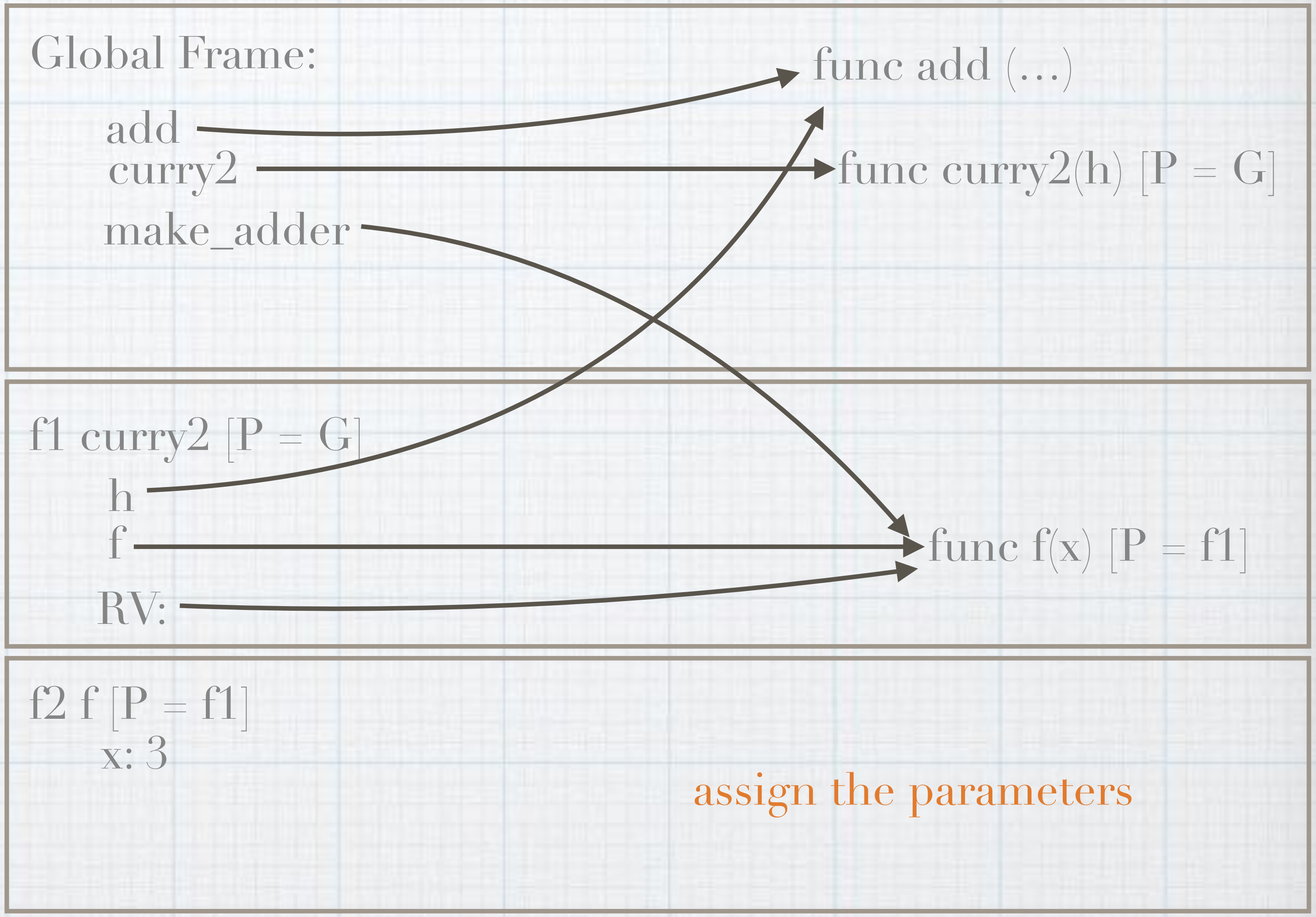
3.1 #2

CF: G, ~~f1~~, f2

I like to draw a line here so that I don't accidentally start evaluating the body of the def right away

```
def curry2(h):

    def f(x):

        def g(y):

            return h(x, y)

        return g

    return f

make_adder = curry2(add)

add_three = make_adder(3)

five = add_three(2)
```

function call



Global Frame:

add → func add (…)

curry2 → func curry2(h) [P = G]

make_adder → func f(x) [P = f1]

f1 curry2 [P = G]

h → func add (…)

f → func f(x) [P = f1]

RV: → func f(x) [P = f1]

f2 f [P = f1]

x: 3

assign the parameters

tip: when you start doing a function call, remember where you were before

3.1 #2

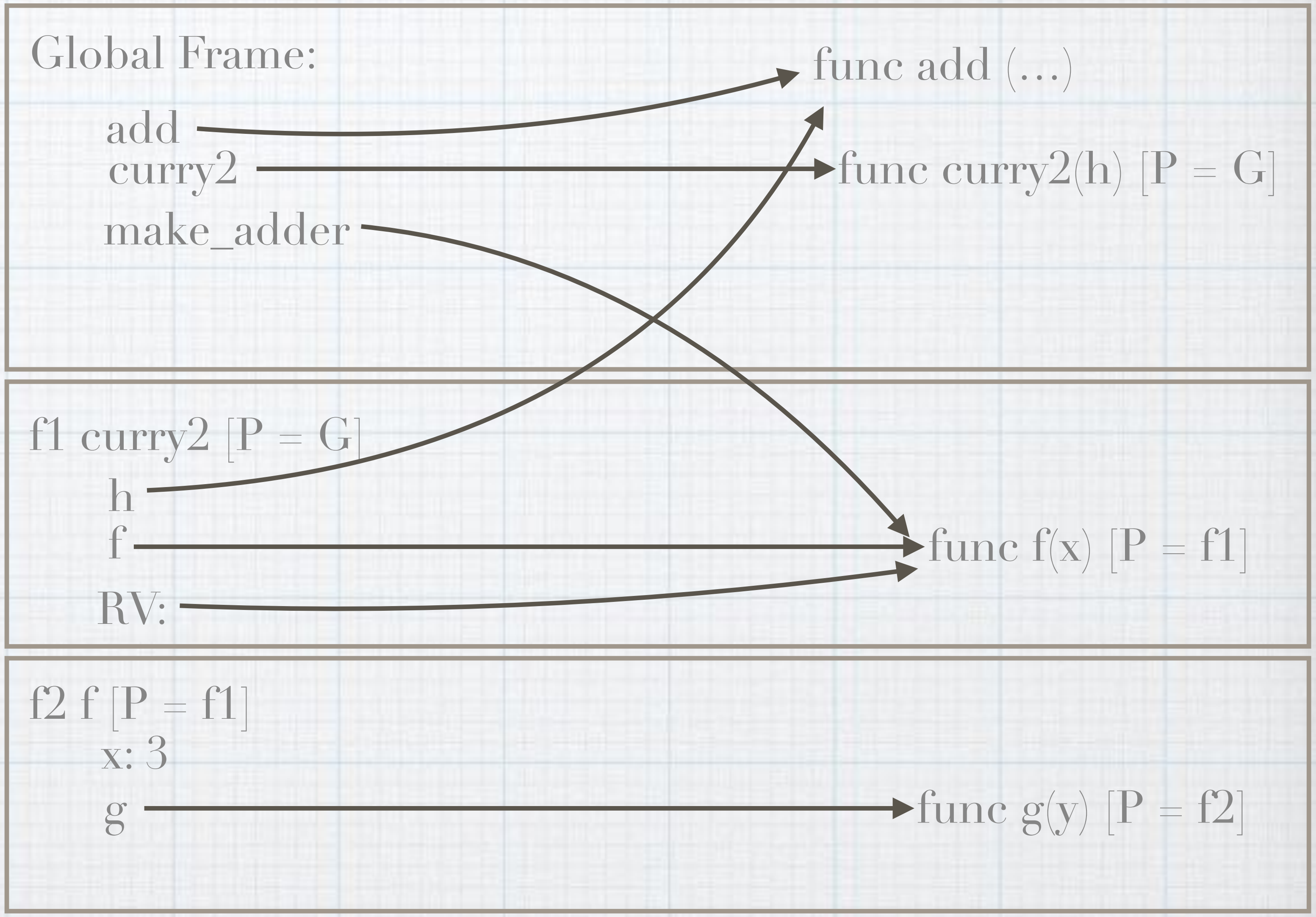
CF: G, ~~f1~~, f2

I like to draw a line here so that I don't accidentally start evaluating the body of the def right away

```
def curry2(h):
    def f(x):
        def g(y):
            return h(x, y)
        return g
    return f
make_adder = curry2(add)
add_three = make_adder(3)
five = add_three(2)
```

Inside f we define a new function, g. What is it's parent?

function call



Global Frame:

add → func add (...)

curry2 → func curry2(h) [P = G]

make_adder → func f(x) [P = f1]

f1 curry2 [P = G]

h → func add (...)

f → func f(x) [P = f1]

RV: → func f(x) [P = f1]

f2 f [P = f1]

x: 3

g → func g(y) [P = f2]

tip: when you start doing a function call, remember where you were before

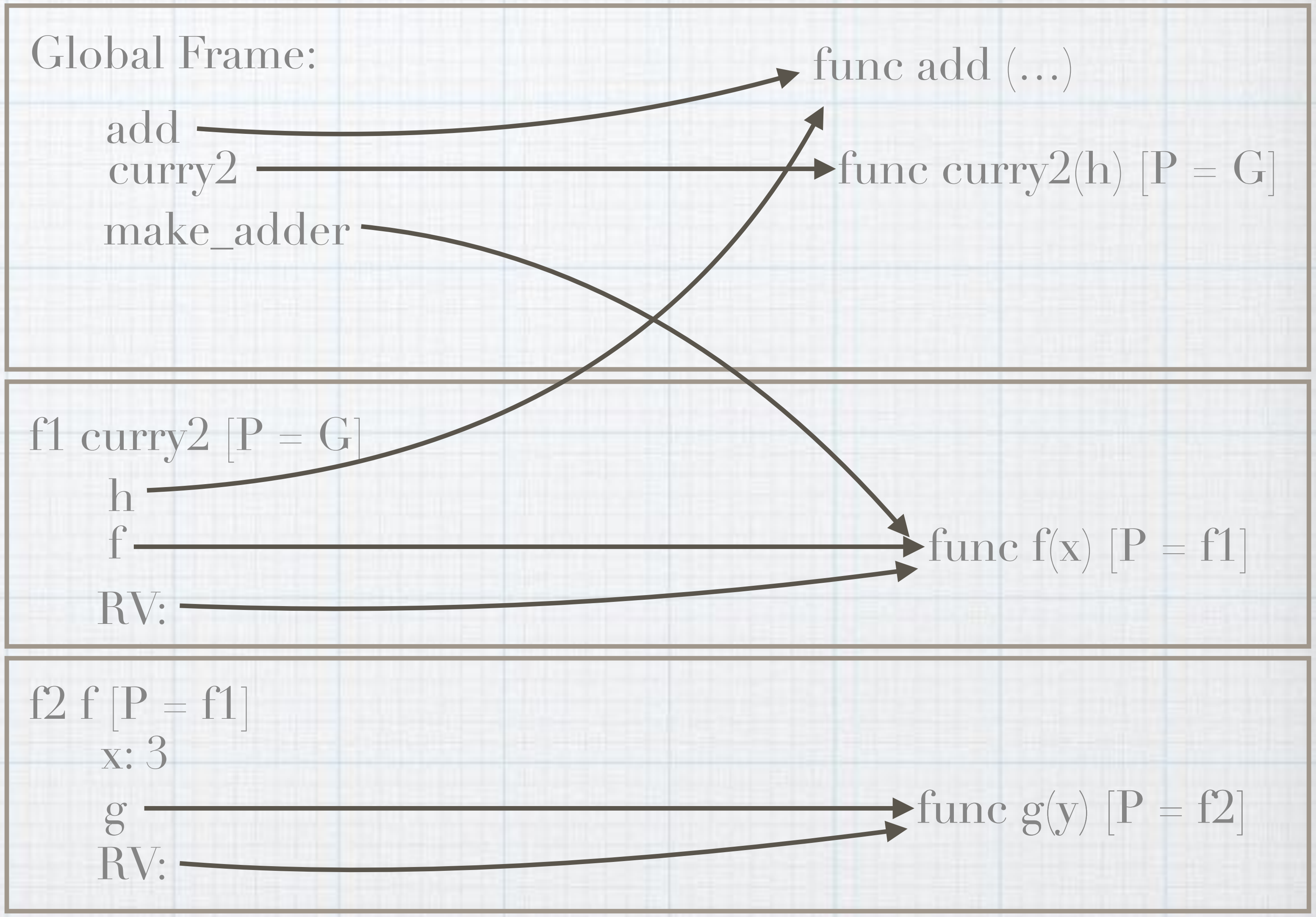
3.1 #2

CF: G, ~~f1~~, ~~f2~~

I like to draw a line here so that I don't start accidentally evaluating the body of the def right away

```
def curry2(h):
    def f(x):
        def g(y):
            return h(x, y)
        return g
    return f
make_adder = curry2(add)
add_three = make_adder(3)
five = add_three(2)
```

function call



now we return the function we just defined

tip: when you start doing a function call, remember where you were before

3.1 #2

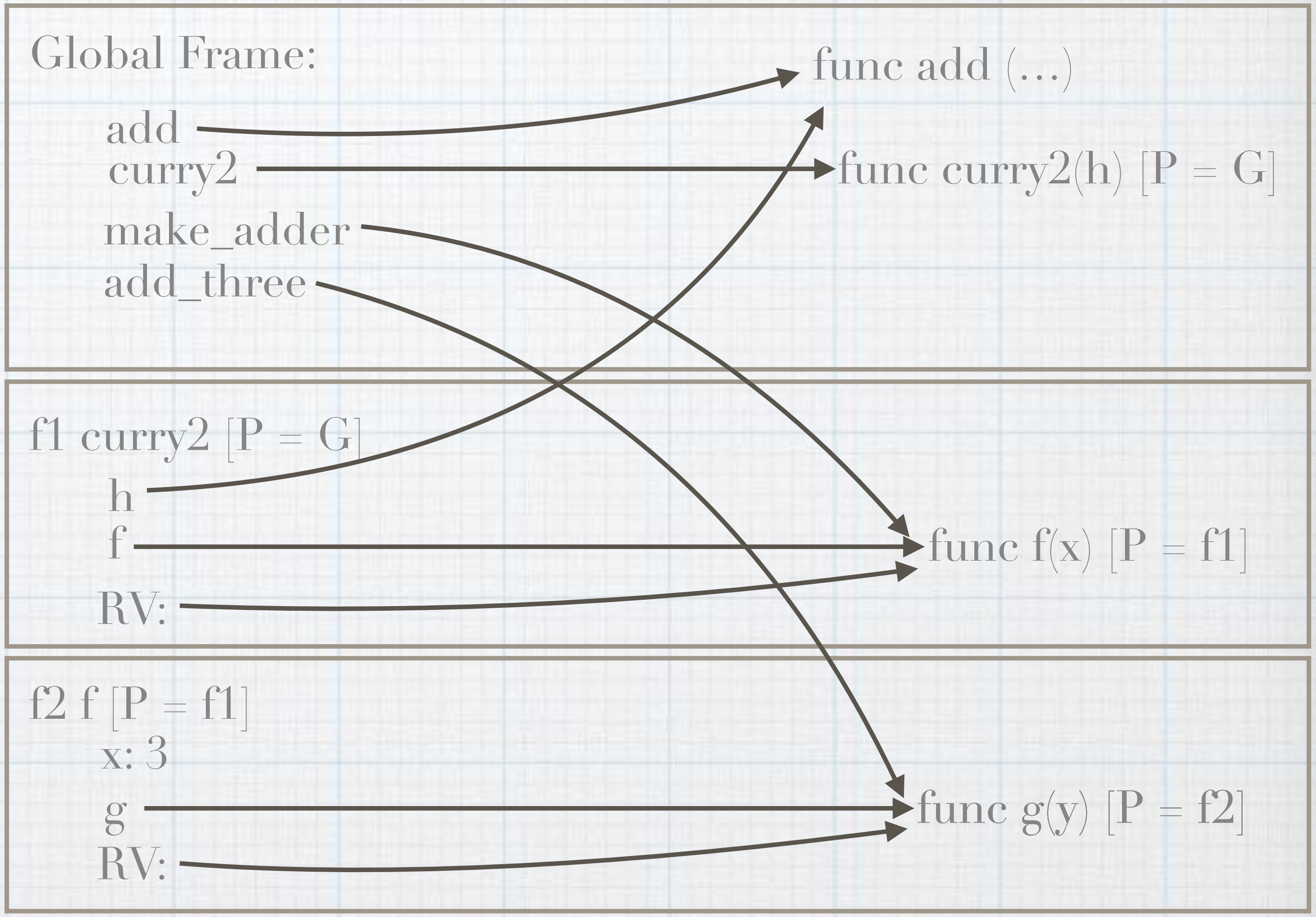
CF: G, ~~f1~~, ~~f2~~

I like to draw a line here so that I don't accidentally start evaluating the body of the def right away

```
def curry2(h):

    def f(x):

        def g(y):

            return h(x, y)

        return g

    return f

make_adder = curry2(add)

add_three = make_adder(3)

five = add_three(2)
```

assignment

function call



Global Frame:

add

curry2

make_adder

add_three

func add (...)

func curry2(h) [P = G]

f1 curry2 [P = G]

h

f

RV:

func f(x) [P = f1]

f2 f [P = f1]

x: 3

g

RV:

func g(y) [P = f2]

finally, assign the return value to add_three

tip: when you start doing a function call, remember where you were before

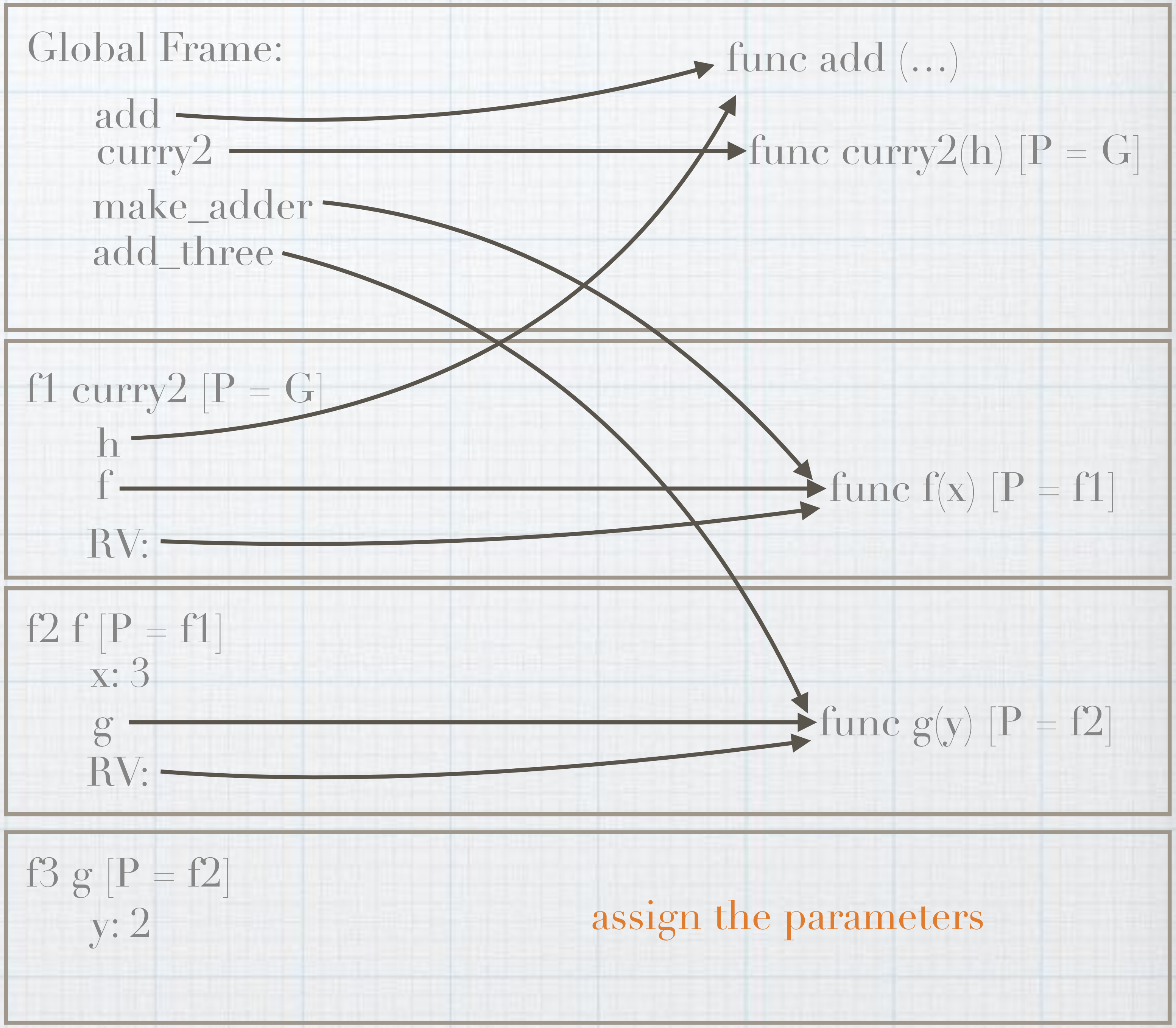
3.1 #2

CF: G, ~~f1~~

I like to draw a line here so that I don't accidentally start evaluating the body of the def right away

```
def curry2(h):
    def f(x):
        def g(y):
            return h(x, y)
        return g
    return f
make_adder = curry2(add)
add_three = make_adder(3)
five = add_three(2)
```

function call



Global Frame:
add
curry2
make_adder
add_three

func add (...)
func curry2(h) [P = G]

f1 curry2 [P = G]
h
f
RV:

func f(x) [P = f1]

f2 f [P = f1]
x: 3
g
RV:

func g(y) [P = f2]

f3 g [P = f2]
y: 2

assign the parameters

tip: when you start doing a function call, remember where you were before

3.1 #2

CF: G, ~~f1~~

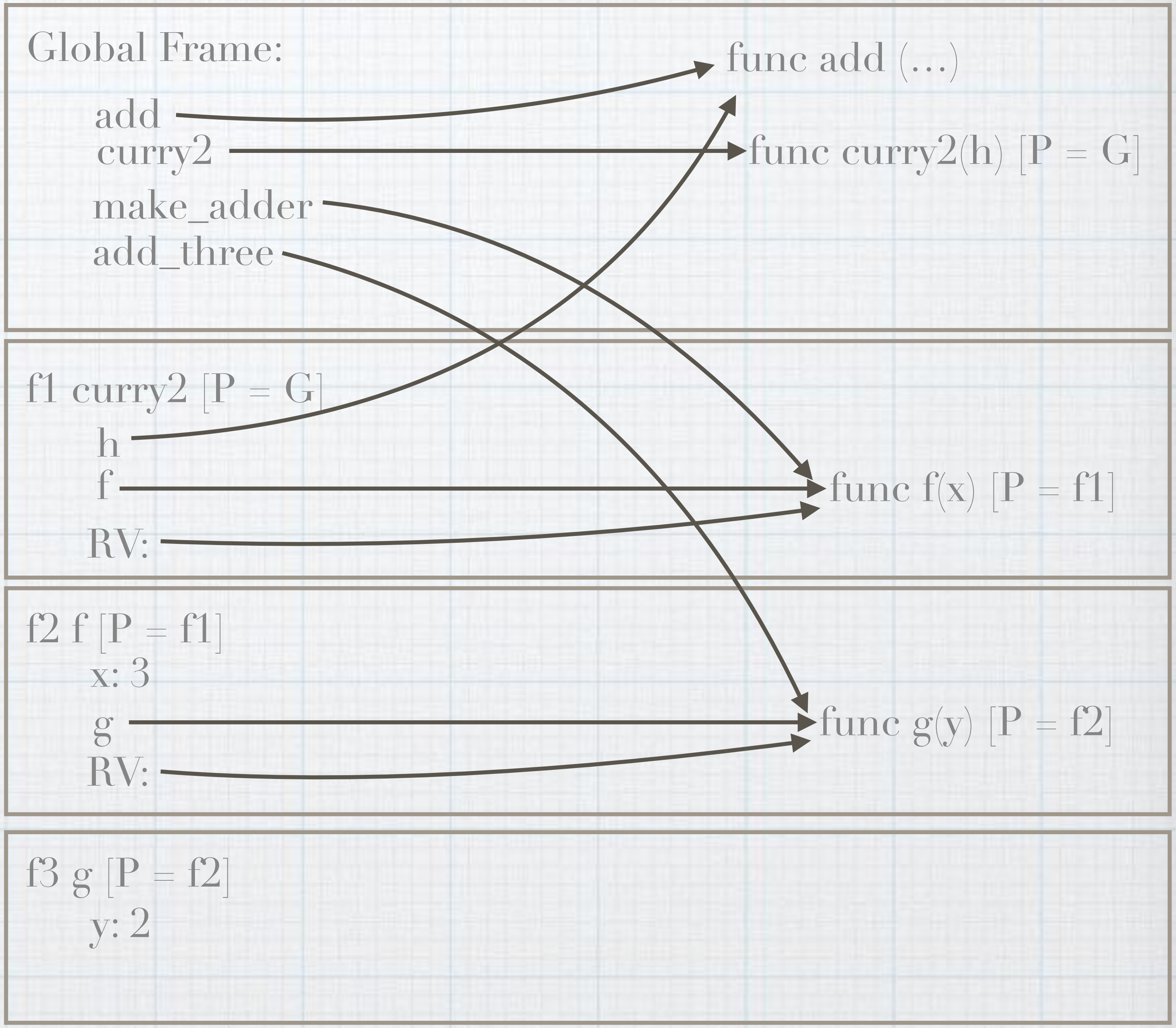
I like to draw a line here so that I don't accidentally start evaluating the body of the def right away

```
def curry2(h):
    def f(x):
        def g(y):
            return h(x, y)
        return g
    return f

make_adder = curry2(add)

add_three = make_adder(3)

five = add_three(2)
```

Here we call h(x, y) but we do not draw a new frame for it. Why?

What are x and y?

function call



Global Frame:

add
curry2
make_adder
add_three

func add (...)

func curry2(h) [P = G]

f1 curry2 [P = G]

h
f
RV:

func f(x) [P = f1]

f2 f [P = f1]

x: 3
g
RV:

func g(y) [P = f2]

f3 g [P = f2]

y: 2

tip: when you start doing a function call, remember where you were before

3.1 #2

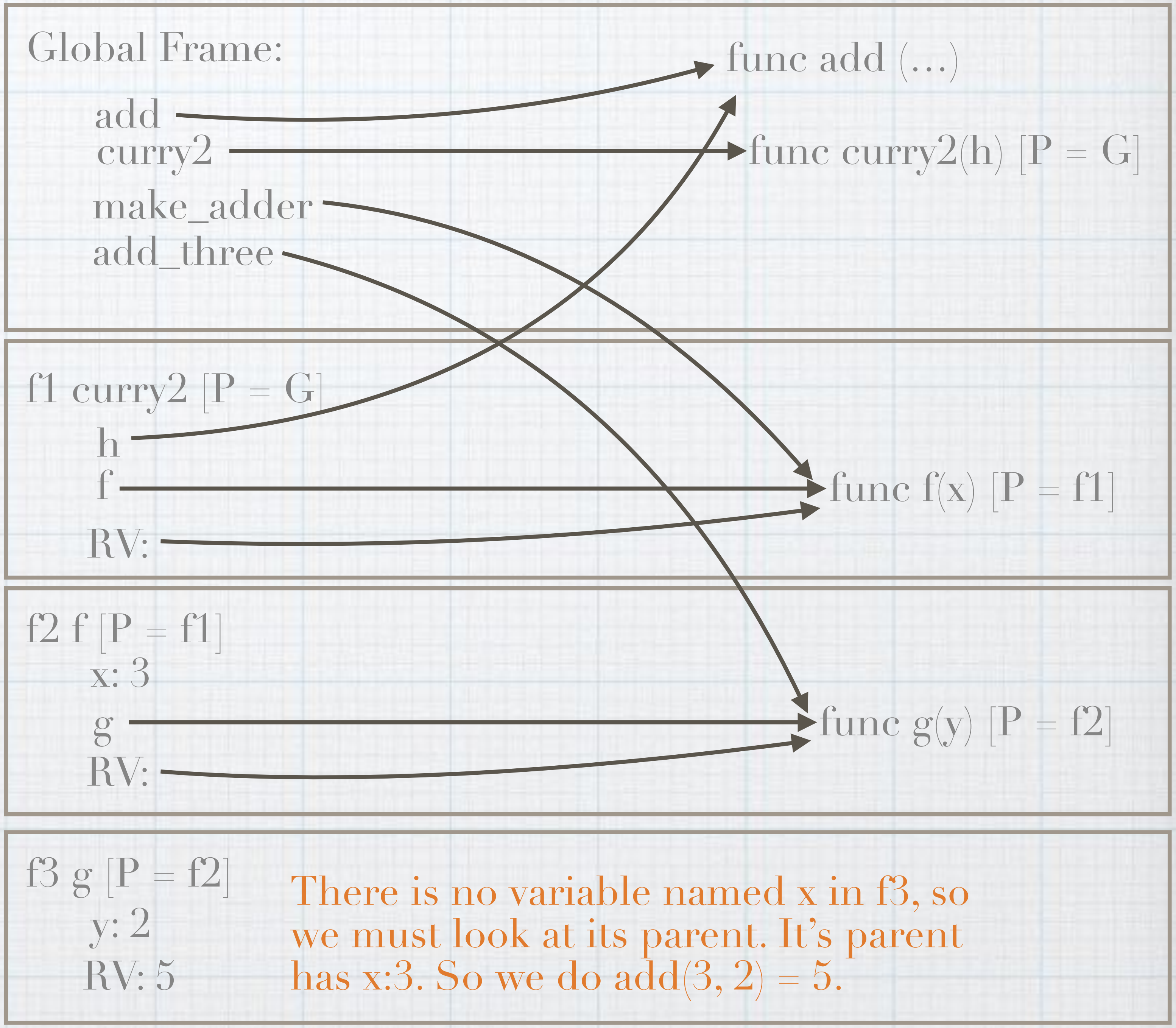
CF: G, ~~f1~~

I like to draw a line here so that I don't accidentally start evaluating the body of the def right away

```
def curry2(h):
    def f(x):
        def g(y):
            return h(x, y)
        return g
    return f
make_adder = curry2(add)
add_three = make_adder(3)
five = add_three(2)
```

return the result of add(3, 2)

function call



Global Frame:
add
curry2
make_adder
add_three

func add (...)
func curry2(h) [P = G]

f1 curry2 [P = G]
h
f
RV:

func f(x) [P = f1]

f2 f [P = f1]
x: 3
g
RV:

func g(y) [P = f2]

f3 g [P = f2]
y: 2
RV: 5

There is no variable named x in f3, so we must look at its parent. It's parent has x:3. So we do add(3, 2) = 5.

tip: when you start doing a function call, remember where you were before

3.1 #2

CF: G, ~~f1~~

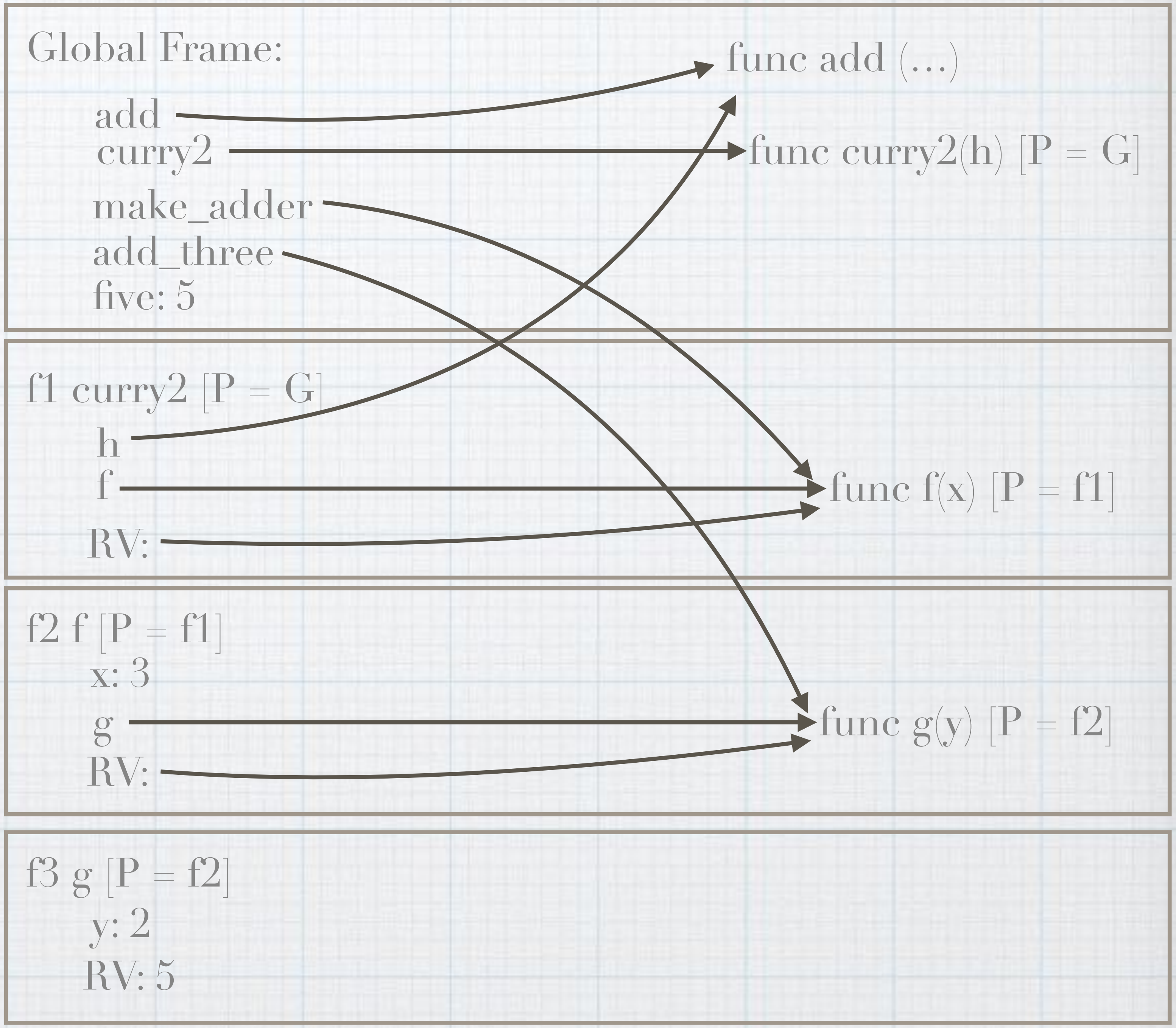
I like to draw a line here so that I don't accidentally start evaluating the body of the def right away

```
def curry2(h):

    def f(x):

        def g(y):

            return h(x, y)

        return g

    return f

make_adder = curry2(add)

add_three = make_adder(3)

five = add_three(2)
```

return the result of add(3, 2)

function call

assignment



Global Frame:
add
curry2
make_adder
add_three
five: 5

func add (...)

func curry2(h) [P = G]

f1 curry2 [P = G]
h
f
RV:

func f(x) [P = f1]

f2 f [P = f1]
x: 3
g
RV:

func g(y) [P = f2]

f3 g [P = f2]
y: 2
RV: 5

tip: when you start doing a function call, mark where you were before so that you know which line to go back to