Review 7

Coroutines

Coroutines on the Exam

- We may ask you to **code a coroutine**
  - This is *potentially* the easier question
  - Need to know how to read specification
  - Could be animation or something else
- Way may ask you a **call frames question**
  - This is not *that* hard, actually
  - Behaves like normal function 90% of time
  - The hardest part is the **first step**

Reworded Lab Problem

```
def animate(ball,dx,speed):
    """Coroutine to animate a ball

    Each call to send provides a time offset dt (float >= 0). Provided
    the sum of dt values is < speed, this moves the ball dx*dt/speed
    horizontally.  If the sum is > speed, it snaps the ball to dx.

    Parameter ball: The ball object to move
    Precondition: ball is an instance of GImage

    Parameter dx: The amount to move the ball
    Precondition: dx is a number (int or float)

    Paramter speed: The number of seconds to animate the ball
    Precondition: speed is a number > 0"""
    pass
```

Reworded Lab Problem

```
def animate(ball,dx,speed):
    """Coroutine to animate a ball

    Each call to send provides a time offset dt (float >= 0). Provided
    the sum of dt values is < speed, this moves the ball dx*dt/speed
    horizontally. If the sum of dt values is >= speed, it snaps the ball to dx.

    Parameter ball: The ball object to move
    Precondition: ball is an instance of GImage

    Parameter dx: The amount to move the ball
    Precondition: dx is a number (int or float)

    Paramter speed: The number of seconds to animate the ball
    Precondition: speed is a number > 0"""
    pass
```

Reference to send

Reworded Lab Problem

```
def animate(ball,dx,speed):
    """Coroutine to animate a ball"""
    # Compute number of steps per second (dx/speed)
    # Create aggregator to track total time

    # Until we reach maximum time (speed)
        # Get the number of seconds passed
        # Add it to the total number of seconds
        # Use time to compute the correct position
        # Update the ball position

    # Snap the ball into place
```

Reworded Lab Problem

```
def animate(ball,dx,speed):
    """Coroutine to animate a ball"""
    # Compute number of steps per second (dx/speed)
    # Create aggregator to track total time

    # Until we reach maximum time (speed)
        # Get the number of seconds passed
        # Add it to the total number of seconds
        # Use time to compute the correct position
        # Update the ball position

    # Snap the ball into place
```

Requires a yield expr

Why??

Reworded Lab Problem

```
def animate(ball,dx,speed):
    """Coroutine to animate a ball"""
    step = dx/speed
    total = 0
    final = ball.x+dx

    while total < speed:
        dt = (yield)
        total = total + dt
        move = dt*step
        ball.x = ball.x + move

    ball.x = final
```

This is doable on an exam

Writing a Coroutine with Both Ways

```
def chunkify(input):
    """Coroutine to break a list into chunks.

    Each call to send is the number of elements to chunk. At each call,
    it yields a new list of the size of the number of elements in send.

    If the size (which is an int) sent is <= 0, it yields the empty list

    Parameter input: The data to process
    Precondition: input is a list"""
    pass
```

Writing a Coroutine with Both Ways

```
def chunkify(input):
    """Coroutine to b                    hunks.

    Each call to send is the number of elements to chunk. At each call,
    it yields a new list of the size of the number of elements in send.

    If the size (wh                      nt is <= 0, it yields the empty list

    Parameter input: The data to process
    Precondition: input is a list"""
    pass
```

Reference to send

What it outputs

Writing a Coroutine with Both Ways

```
def chunkify(input):
    """Coroutine to break a list into chunks.

    Each call to send is the number of elements to chunk. At each call,
    it yields a new list of the size of the number of elements in send.

    If the size (which is an int) sent is <= 0, it yields the empty list

    Parameter input: The data to proces
    Precondition: input is a list"""
    pass
```

Kind of like a **precondition**

Writing a Coroutine with Both Ways

```
def chunkify(input):
    """Coroutine to break a list into chunks."""
    # Get size of first chunk
    # Create an accumulator for chunks (output lists)

    # for each item in input:
        # check if number of items needed is 0
            # output the accumulator have so far
            # get the size of the next chunk
            # reset the accumulator to empty list
        # add item to accumulator
```

Writing a Coroutine with Both Ways

```
def chunkify(input):
    """Coroutine to break a list ...
    # Get size of first chunk
    # Create an accumulator for chunks

    # for each item in input:
        # check if number of items needed is 0
            # output the accumulator have so far
            # get the size of the next chunk
            # reset the accumulator to empty list
        # add item to accumulator
```

Initial input

These are done by same yield expr

Writing a Coroutine with Both Ways

```
def chunkify(input):
    """Coroutine to br                    ks."""
    size = (yield)
    result = []

    for item in input:
        if size == 0:
            size = (yield result)
            result = []
        result.append(item)
        size = size -1
```

Initial input

Output result
Input the size

Writing a Coroutine with Both Ways

```
def chunkify(input):
    """Coroutine to break a list into chunks."""
    size = (yield)
    result = []

    for item in input:
        if size == 0:
            size = (yield result)
            result = []
        result.append(item)
        size = size -1
```

Are we done?

Writing a Coroutine with Both Ways

```
def chunkify(input):
    """Coroutine to break a list into chunks."""
    size = (yield)
    result = []

    for item in input:
        if size == 0:
            size = (yield result)
            result = []
        result.append(item)
        size = size -1

    yield result
```

Need one last output

Writing a Coroutine with Both Ways

```
def chunkify(input):
    """Coroutine to break a list into chunks."""
    size = (yield)
    result = []

    for item in input:
        if size == 0:
            size = (yield result)
            result = []
        result.append(item)
        size = size -1

    yield result
```

Are we done?

Writing a Coroutine with Both Ways

```
def chunkify(input):
    """Coroutine to break a list into chunks."""
    size = (yield)
    result = []

    for item in input:
        if size == 0:
            size = (yield result)
            result = []
        result.append(item)
        size = size -1

    yield result
```

What if size is 0?

Writing a Coroutine with Both Ways

```
def chunkify(input):
    """Coroutine to break a list into chunks."""
    size = (yield)
    result = []

    for item in input:
        while size == 0:
            size = (yield result)
            result = []
        result.append(item)
        size = size -1

    yield result
```

Simple change

Writing a Coroutine with Both Ways

```
def chunkify(input):
    """Coroutine to break a list into chunks."""
    size = (yield)
    result = []

    for item in input:
        while size == 0:
            size = (yield result)
            result = []
        result.append(item)
        size = size -1

    yield result
```

Initial input

Input/output

Final output

Pseudocode makes this much easier

Coroutines and Call Frames

- Recall a generator has **three** steps
  - Initial creation of coroutine (like constructor)
  - Initial call to `next` to get started
  - Subsequent call to `send` to keep going
- Cannot ask you a question about first!
- The second is just like a generator!
  - Stops at the first yield expression/statement
- Only the third one is actually new
  - And only challenge is input vs. output

Coroutine with Input Only

Coroutine

```
    def addit(nums):
        """Send added to next item
        Pre: nums is a list of ints"""
15      pos = 0
16      while pos < len(nums):
17          add = (yield)
18          nums[pos]=nums[pos]+add
19          pos = pos + 1
```

Code to Execute

```
23  a = [1,2,3]
24  b = addit(a)
25  next(b)
26  c = b.send(2)
```

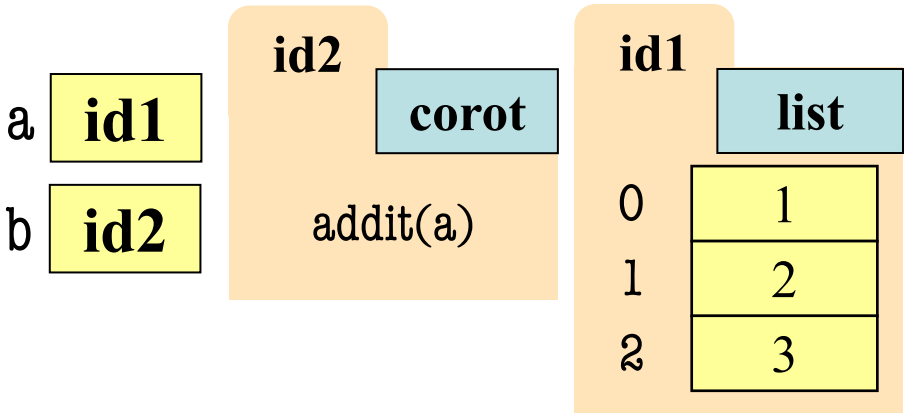
Coroutine with Input Only

Coroutine

```
    def addit(nums):
        """Send added to next item
        Pre: nums is a list of ints"""
15      pos = 0
16      while pos < len(nums):
17          add = (yield)
18          nums[pos]=nums[pos]+add
19          pos = pos + 1
```

Given at Line 24

```
23  a = [1,2,3]
24  b = addit(a)
25  next(b)
26  c = b.send(2)
```



Coroutine with Input Only

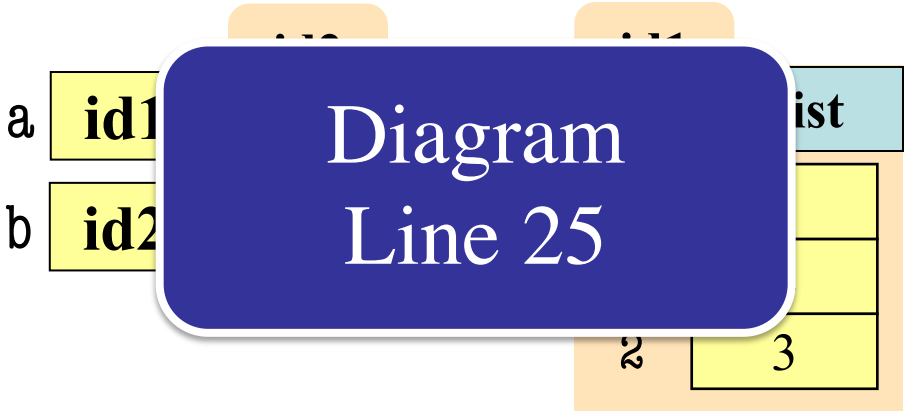
Coroutine

```
    def addit(nums):
        """Send added to next item
        Pre: nums is a list of ints"""
15      pos = 0
16      while pos < len(nums):
17          add = (yield)
18          nums[pos]=nums[pos]+add
19          pos = pos + 1
```

Given at Line 24

```
23  a = [1,2,3]
24  b = addit(a)
25  next(b)
26  c = b.send(2)
```

Just like a generator

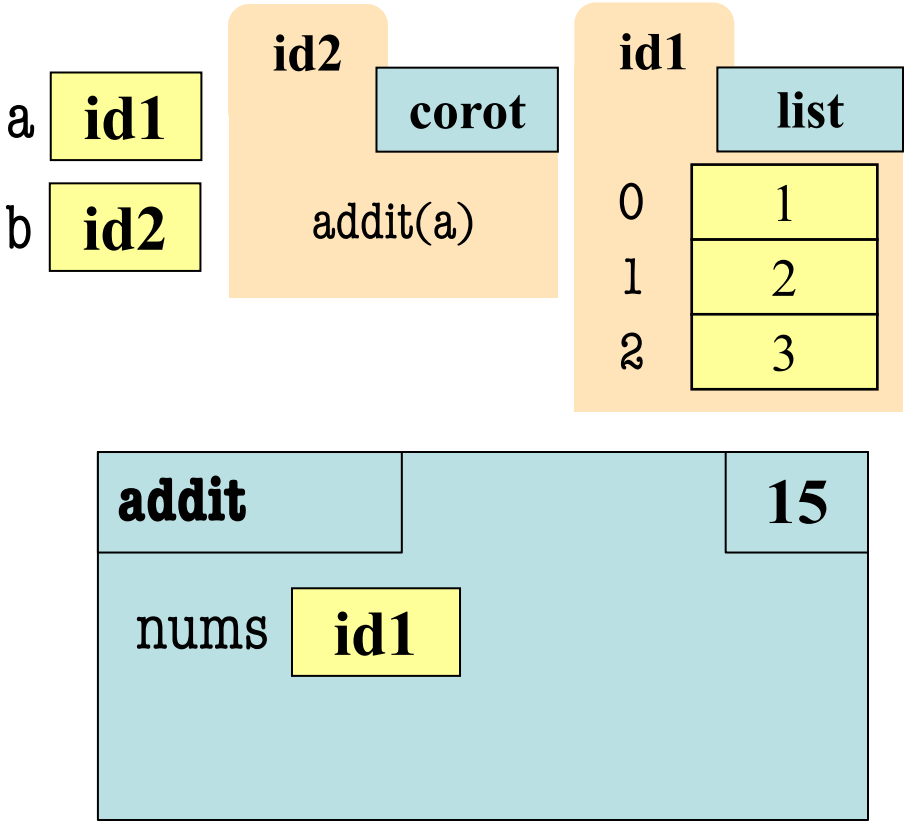


Coroutine with Input Only

Coroutine

```
    def addit(nums):
        """Send added to next item
        Pre: nums is a list of ints"""
15      pos = 0
16      while pos < len(nums):
17          add = (yield)
18          nums[pos]=nums[pos]+add
19          pos = pos + 1
```

Initial Diagram



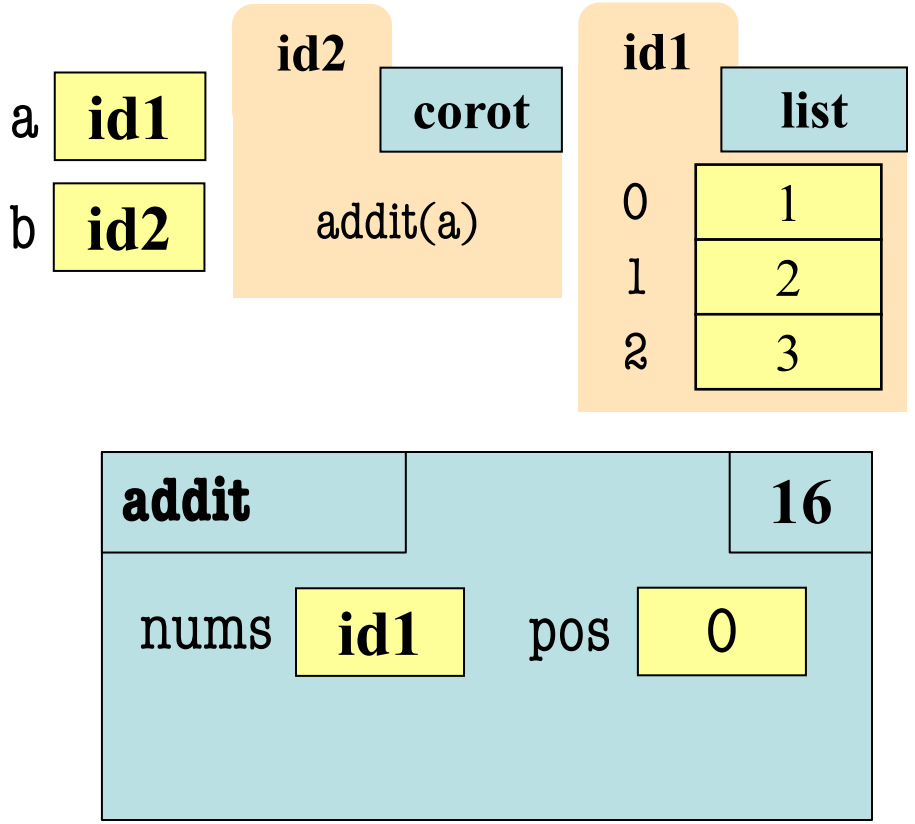
Coroutine with Input Only

Coroutine

```
    def addit(nums):
        """Send ad
        Pre: nums
15      pos = 0
16      while pos < len(nums):
17          add = (yield)
18          nums[pos]=nums[pos]+add
19          pos = pos + 1
```

Will cheat and not show frame

Diagram Step 2

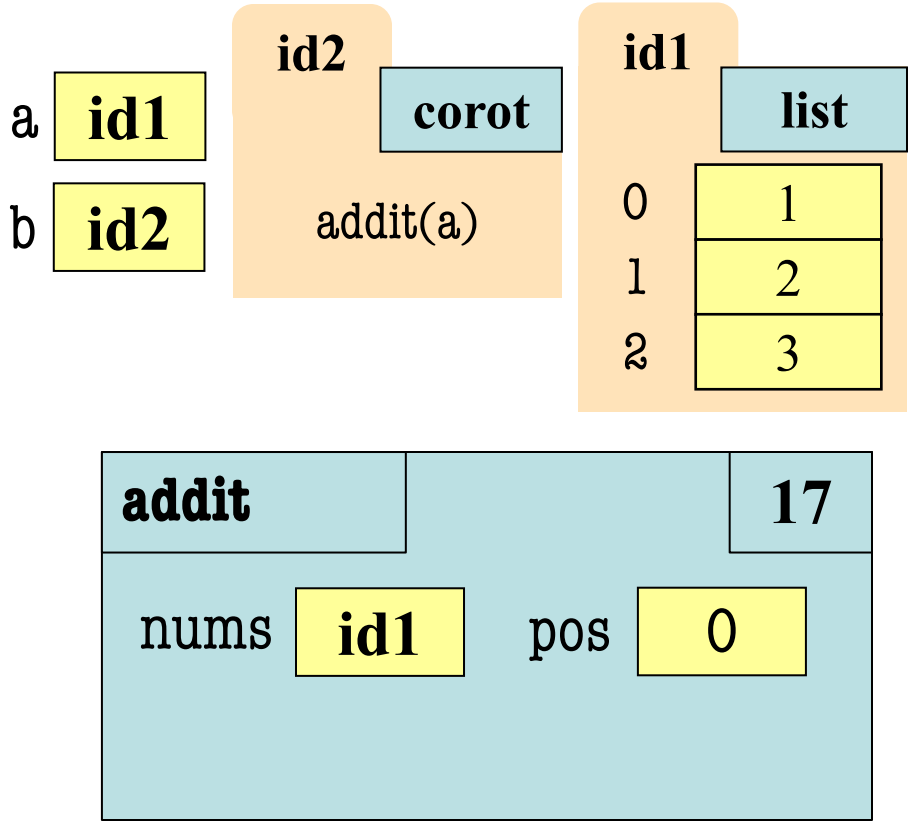


Coroutine with Input Only

Coroutine

```
    def addit(nums):
        """Send added to next item
        Pre: nums is a list of ints"""
15      pos = 0
16      while pos < len(nums):
17          add = (yield)
18          nums[pos]=nums[pos]+add
19          pos = pos + 1
```

Diagram Step 3

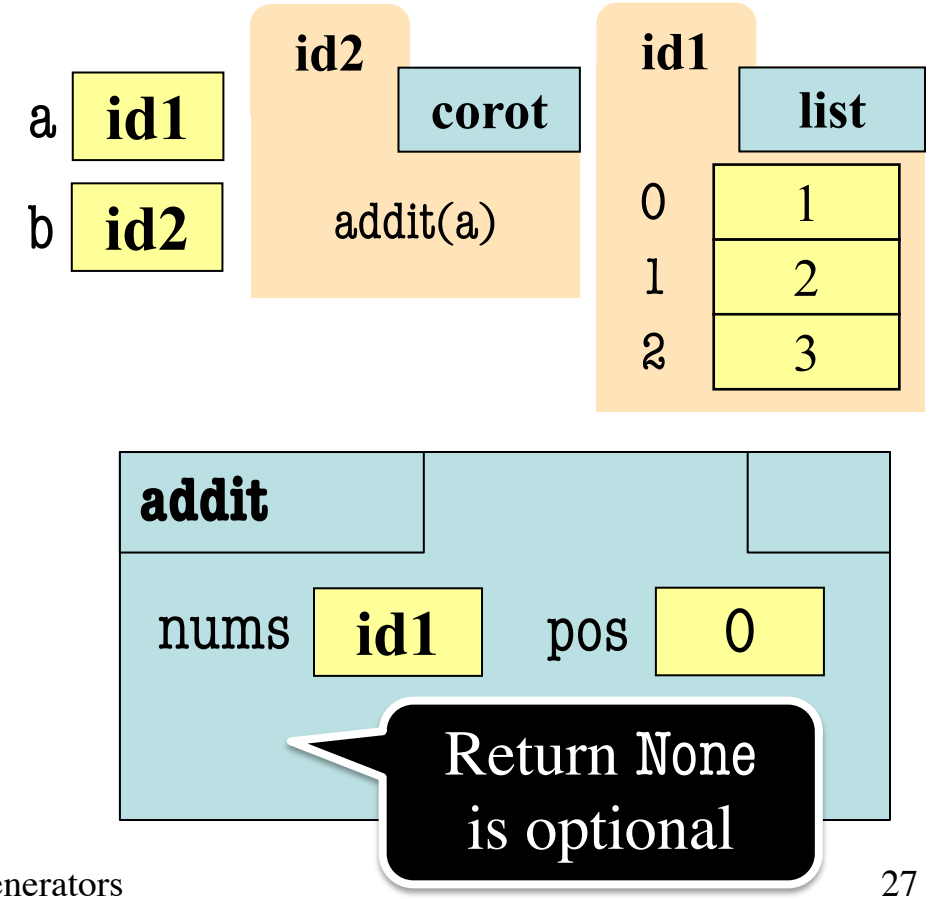


Coroutine with Input Only

Coroutine

```
    def addit(nums):
        """Send added to next item
        Pre: nums is a list of ints"""
15      pos = 0
16      while pos < len(nums):
17          add = (yield)
18          nums[pos]=nums[pos]+add
19          pos = pos + 1
```

Diagram Step 4

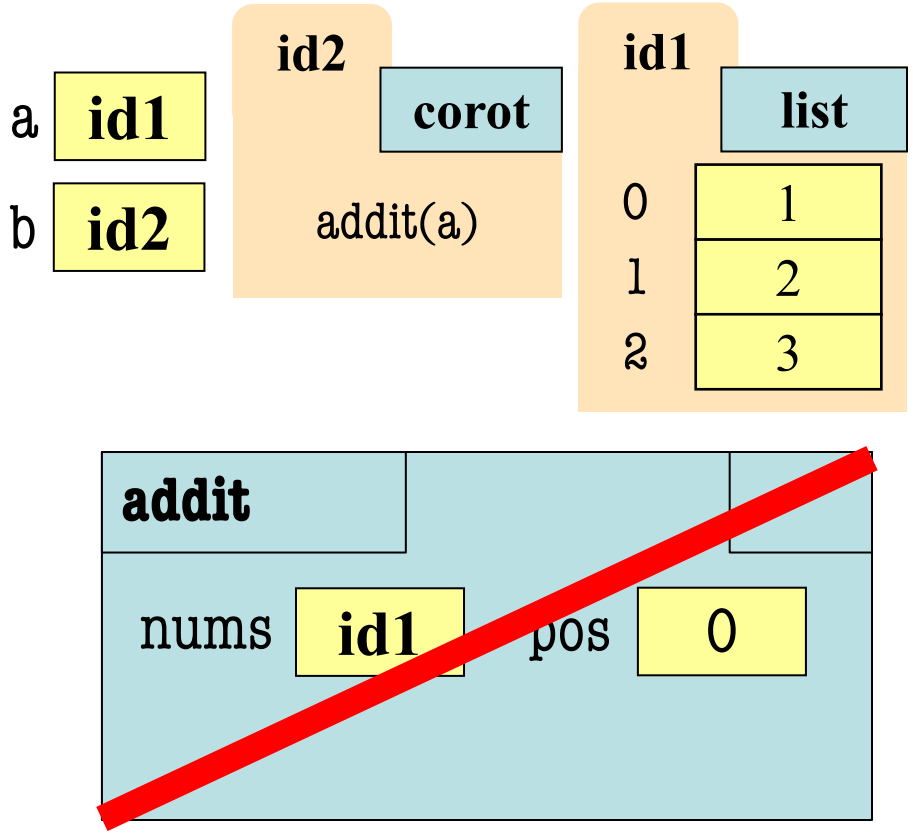


Coroutine with Input Only

Coroutine

```
    def addit(nums):
        """Send added to next item
        Pre: nums is a list of ints"""
15      pos = 0
16      while pos < len(nums):
17          add = (yield)
18          nums[pos]=nums[pos]+add
19          pos = pos + 1
```

Erase the Frame



Coroutine with Input Only

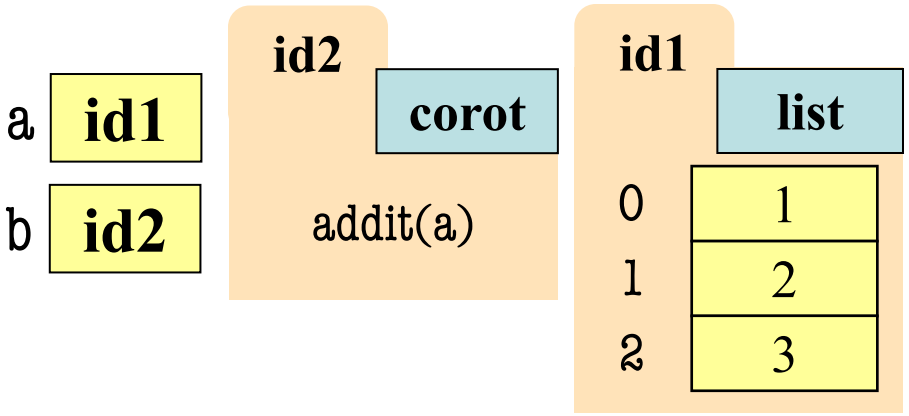
Coroutine

```
    def addit(nums):
        """Send added to next item
        Pre: nums is a list of ints"""
15      pos = 0
16      while pos < len(nums):
17          add = (yield)
18          nums[pos]=nums[pos]+add
19          pos = pos + 1
```

Given at Line 25

```
23  a = [1,2,3]
24  b = addit(a)
25  next(b)
26  c = b.send(2)
```

Not much has changed



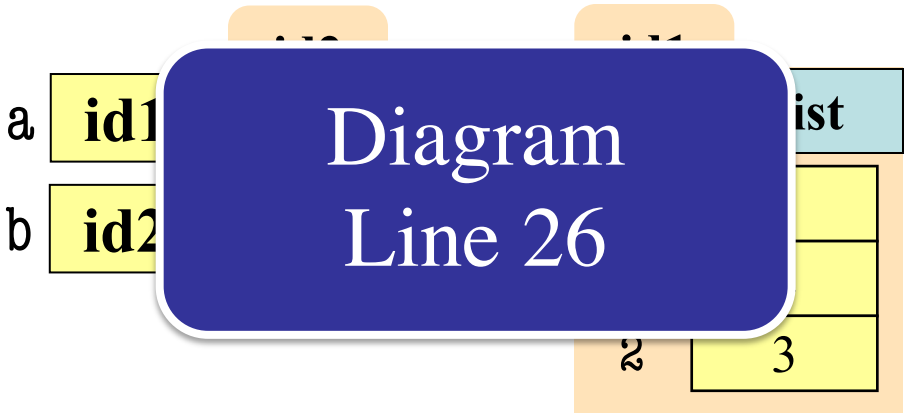
Coroutine with Input Only

Coroutine

```
    def addit(nums):
        """Send added to next item
        Pre: nums is a list of ints"""
15      pos = 0
16      while pos < len(nums):
17          add = (yield)
18          nums[pos]=nums[pos]+add
19          pos = pos + 1
```

Given at Line 24

```
23  a = [1,2,3]
24  b = addit(a)
25  next(b)
26  c = b.send(2)
```



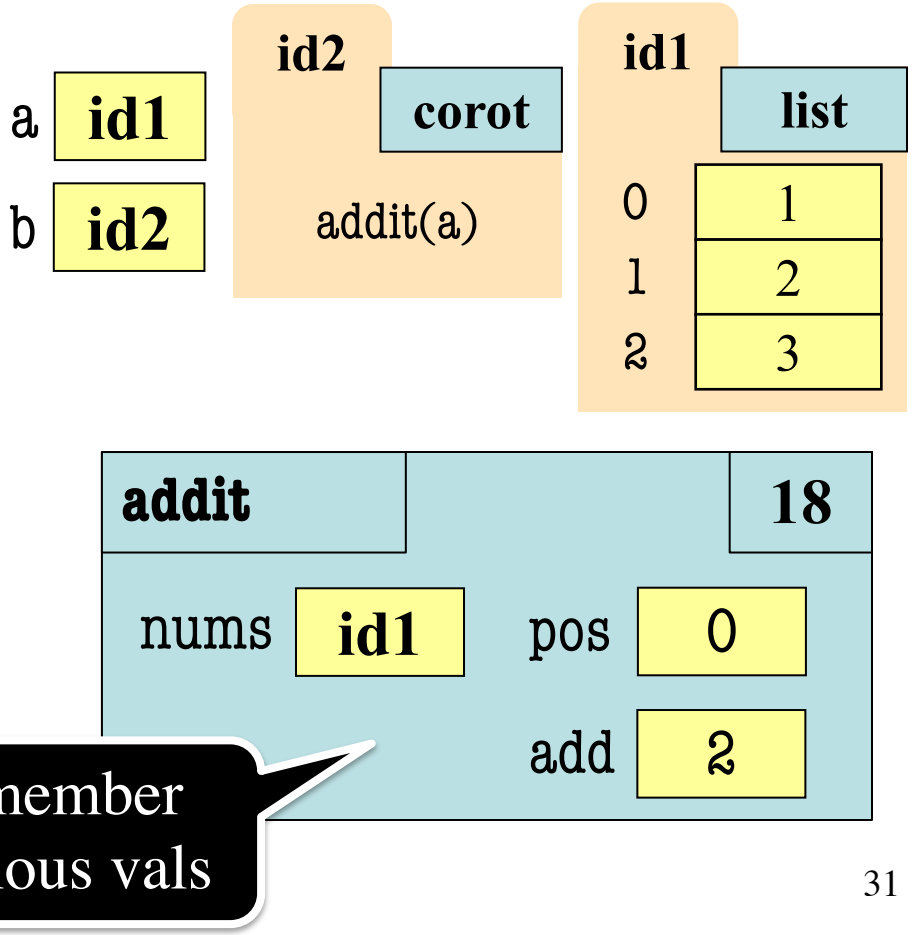
Coroutine with Input Only

Coroutine

```
def addit(nums):
    """Send added to next item
    Pre: nums is a list of ints"""
15  pos = 0
16  while pos < len(nums):
17      add = (yield)
18      nums[pos]=nums[pos]+add
19
```

Input goes to add

Initial Diagram

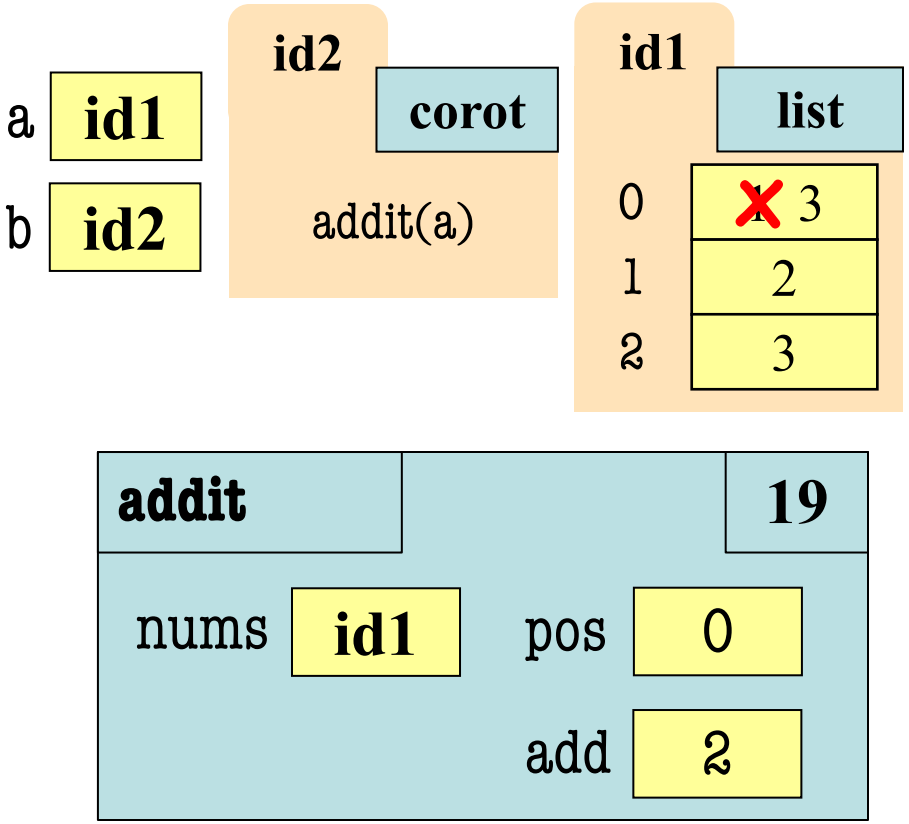


Coroutine with Input Only

Coroutine

```
    def addit(nums):
        """Send added to next item
        Pre: nums is a list of ints"""
15      pos = 0
16      while pos < len(nums):
17          add = (yield)
18          nums[pos]=nums[pos]+add
19          pos = pos + 1
```

Diagram Step 2

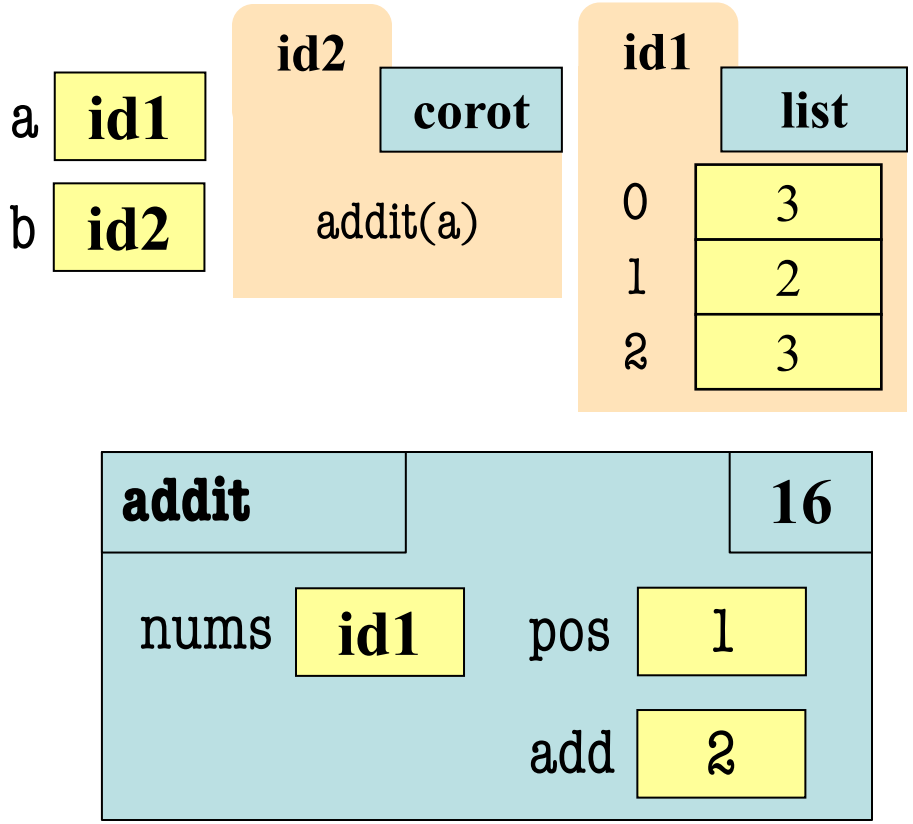


Coroutine with Input Only

Coroutine

```
    def addit(nums):
        """Send added to next item
        Pre: nums is a list of ints"""
15      pos = 0
16      while pos < len(nums):
17          add = (yield)
18          nums[pos]=nums[pos]+add
19          pos = pos + 1
```

Diagram Step 3

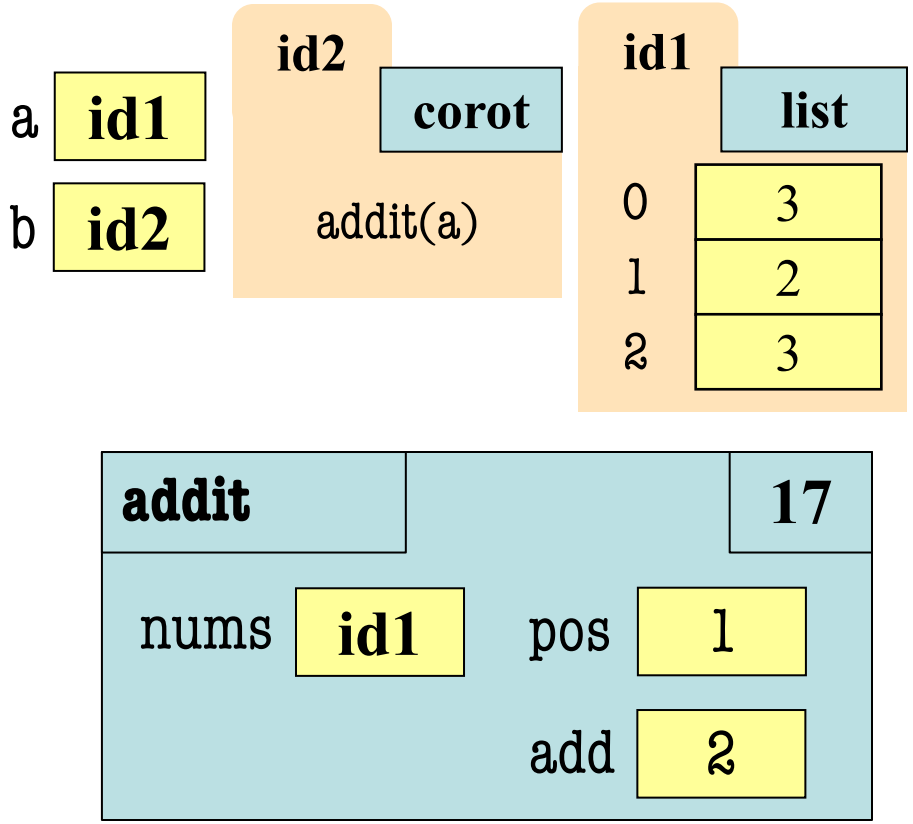


Coroutine with Input Only

Coroutine

```
    def addit(nums):
        """Send added to next item
        Pre: nums is a list of ints"""
15      pos = 0
16      while pos < len(nums):
17          add = (yield)
18          nums[pos]=nums[pos]+add
19          pos = pos + 1
```

Diagram Step 4

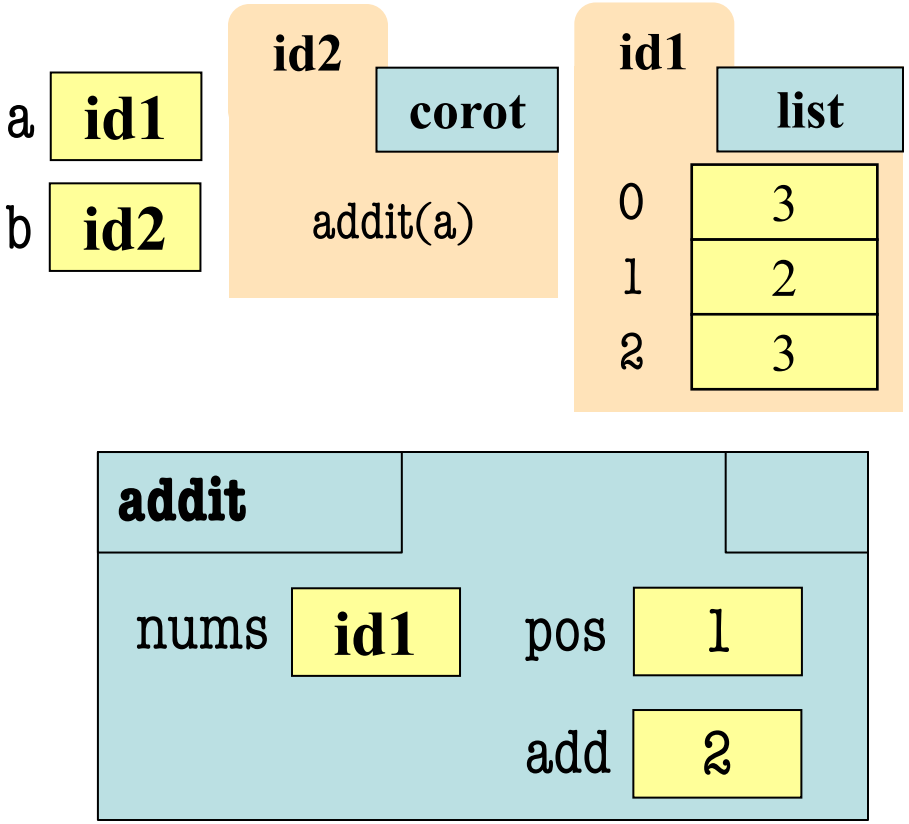


Coroutine with Input Only

Coroutine

```
    def addit(nums):
        """Send added to next item
        Pre: nums is a list of ints"""
15      pos = 0
16      while pos < len(nums):
17          add = (yield)
18          nums[pos]=nums[pos]+add
19          pos = pos + 1
```

Diagram Step 5

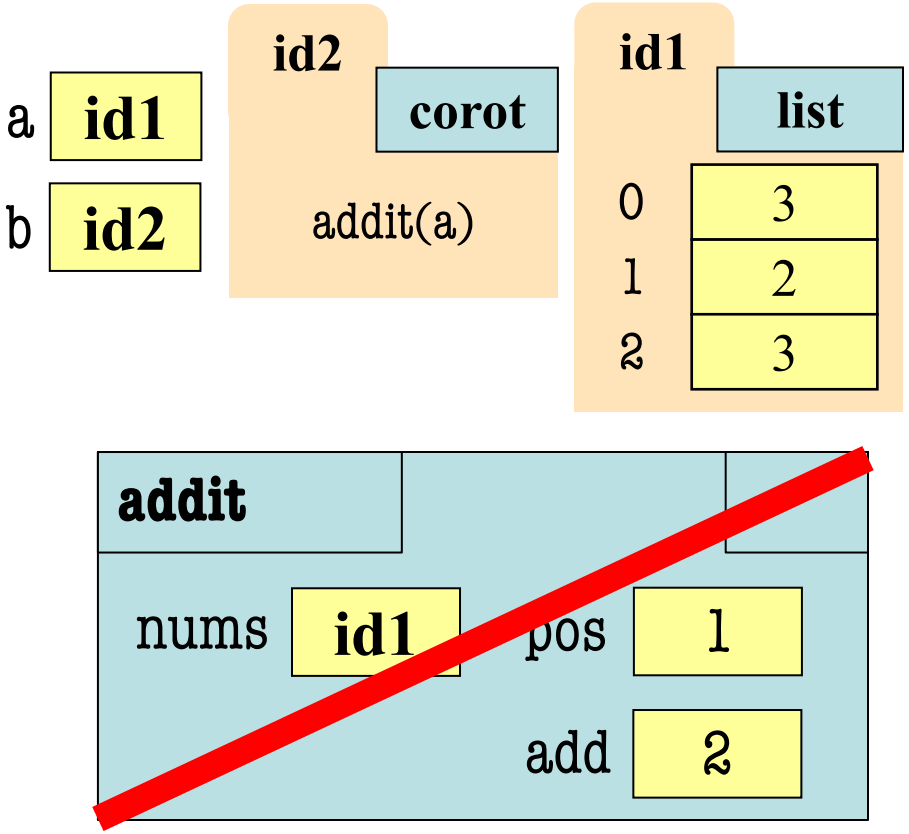


Coroutine with Input Only

Coroutine

```
    def addit(nums):
        """Send added to next item
        Pre: nums is a list of ints"""
15      pos = 0
16      while pos < len(nums):
17          add = (yield)
18          nums[pos]=nums[pos]+add
19          pos = pos + 1
```

Erase the Frame



Coroutine with Input and Ouput

Coroutine

```
    def skipit(nums):
        """Send tells positions to skip
        Pre: nums is a list of ints"""
15      pos = 0
16      while pos < len(nums):
17          skip = (yield nums[pos])
18          pos = pos+skip
19      yield nums[-1]
```

Code to Execute

```
22  a = [9,7,4,2]
23  b = skipit(a)
24  c = next(b)
25  d = b.send(2)
26  e = b.send(3)
```

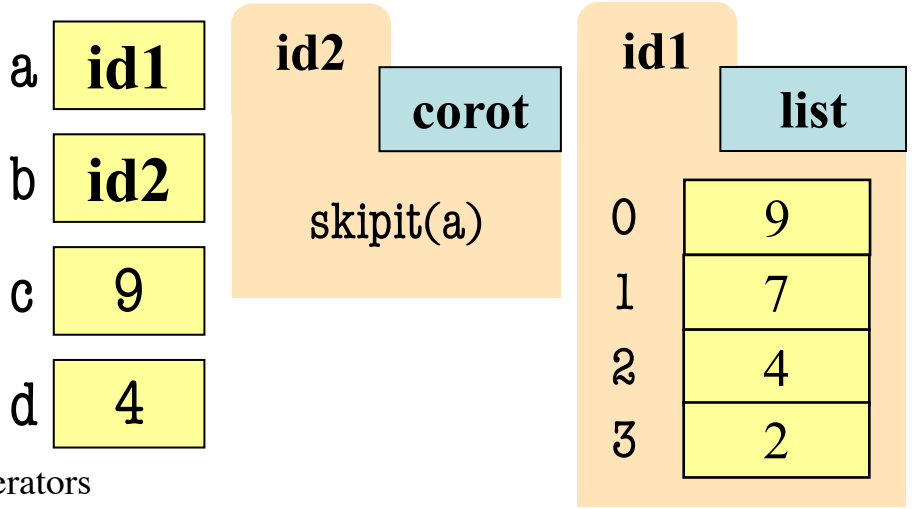
Coroutine with Input and Ouput

Coroutine

```
    def skipit(nums):
        """Send tells positions to skip
        Pre: nums is a list of ints"""
15      pos = 0
16      while pos < len(nums):
17          skip = (yield nums[pos])
18          pos = pos+skip
19      yield nums[-1]
```

Given at Line 25

```
22  a = [9,7,4,2]
23  b = skipit(a)
24  c = next(b)
25  d = b.send(2)
26  e = b.send(3)
```



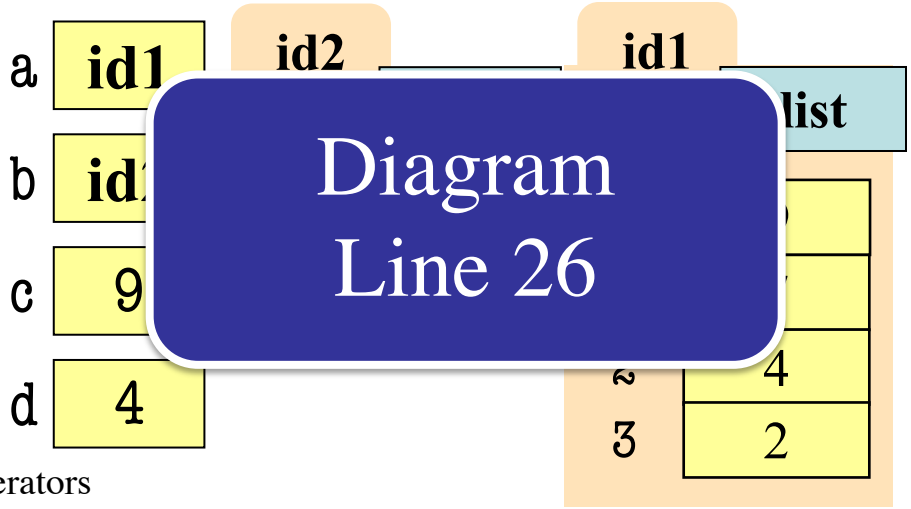
Coroutine with Input and Ouput

Coroutine

```
    def skipit(nums):
        """Send tells positions to skip
        Pre: nums is a list of ints"""
15      pos = 0
16      while pos < len(nums):
17          skip = (yield nums[pos])
18          pos = pos+skip
19      yield nums[-1]
```

Given at Line 25

```
22  a = [9,7,4,2]
23  b = skipit(a)
24  c = next(b)
25  d = b.send(2)
26  e = b.send(3)
```

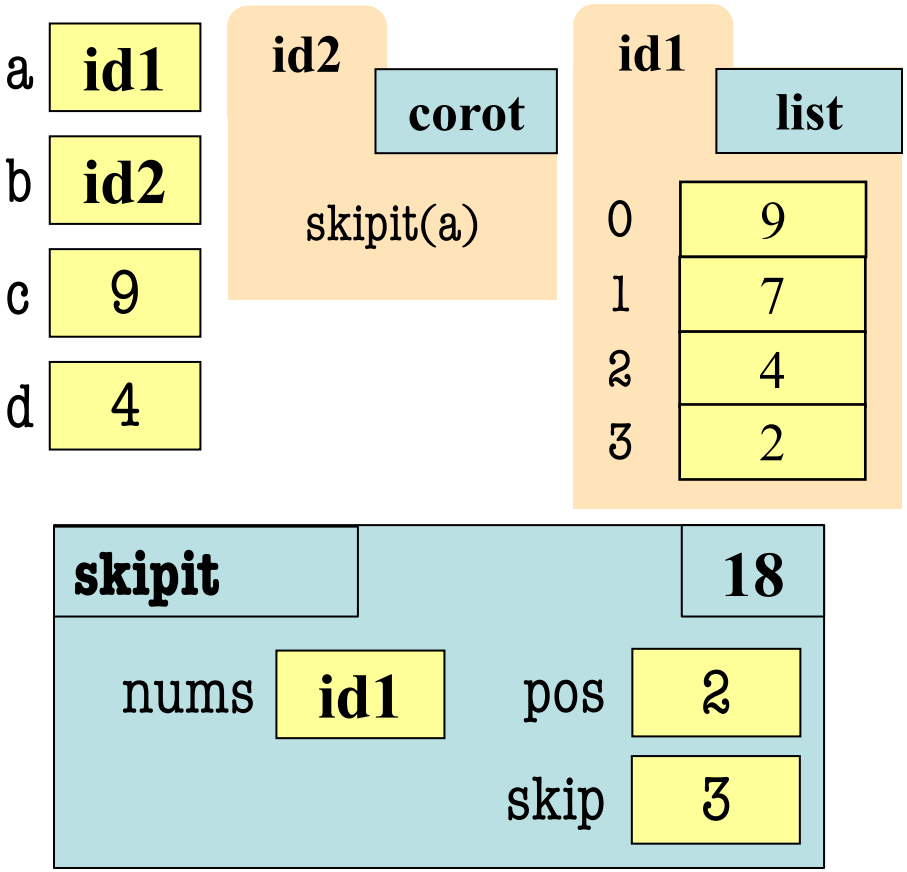


Coroutine with Input and Ouput

Coroutine

```
    def skipit(nums):
        """Send tells positions to skip
        Pre: nums is a list of ints"""
15      pos = 0
16      while pos < len(nums):
17          skip = (yield nums[pos])
18          pos = pos+skip
19      yield nums[-1]
```

Initial Diagram

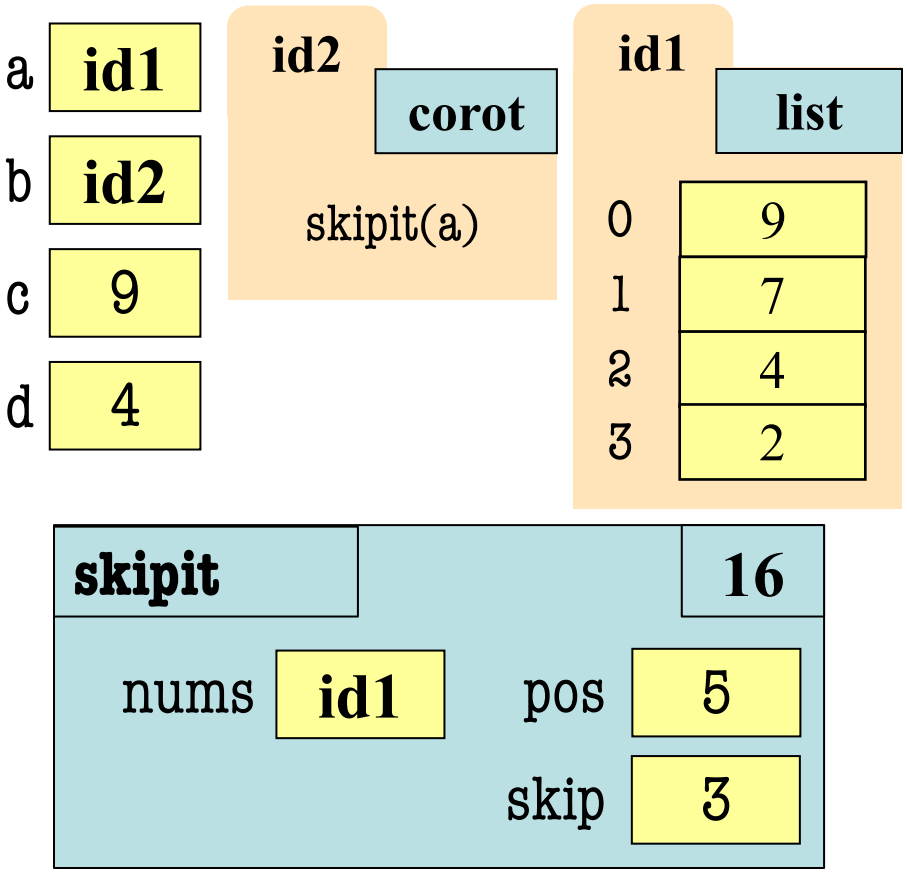


Coroutine with Input and Ouput

Coroutine

```
    def skipit(nums):
        """Send tells positions to skip
        Pre: nums is a list of ints"""
15      pos = 0
16      while pos < len(nums):
17          skip = (yield nums[pos])
18          pos = pos+skip
19      yield nums[-1]
```

Diagram Step 2

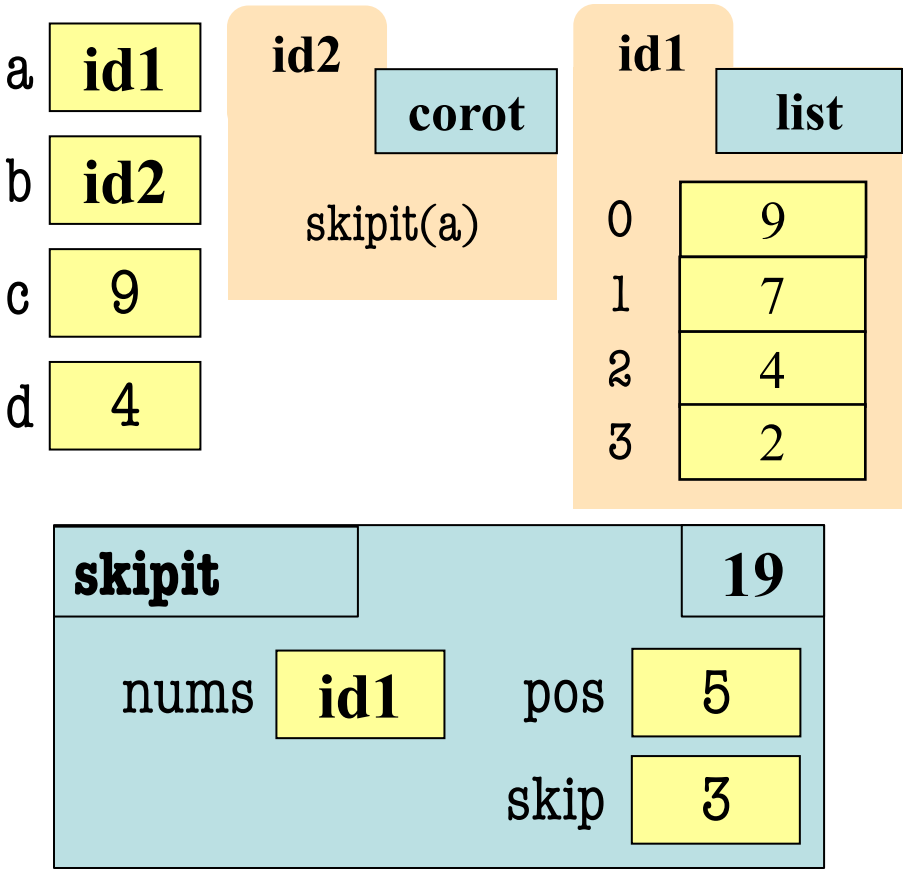


Coroutine with Input and Ouput

Coroutine

```
    def skipit(nums):
        """Send tells positions to skip
        Pre: nums is a list of ints"""
15      pos = 0
16      while pos < len(nums):
17          skip = (yield nums[pos])
18          pos = pos+skip
19      yield nums[-1]
```

Diagram Step 3

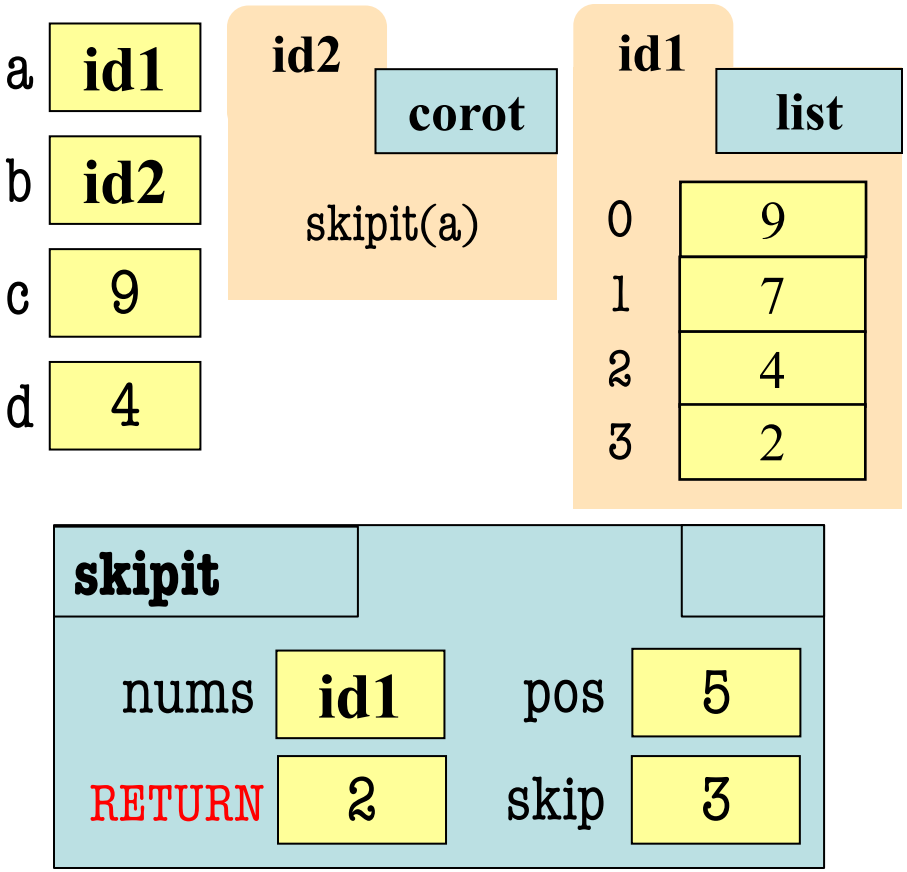


Coroutine with Input and Ouput

Coroutine

```
    def skipit(nums):
        """Send tells positions to skip
        Pre: nums is a list of ints"""
15      pos = 0
16      while pos < len(nums):
17          skip = (yield nums[pos])
18          pos = pos+skip
19      yield nums[-1]
```

Diagram Step 4

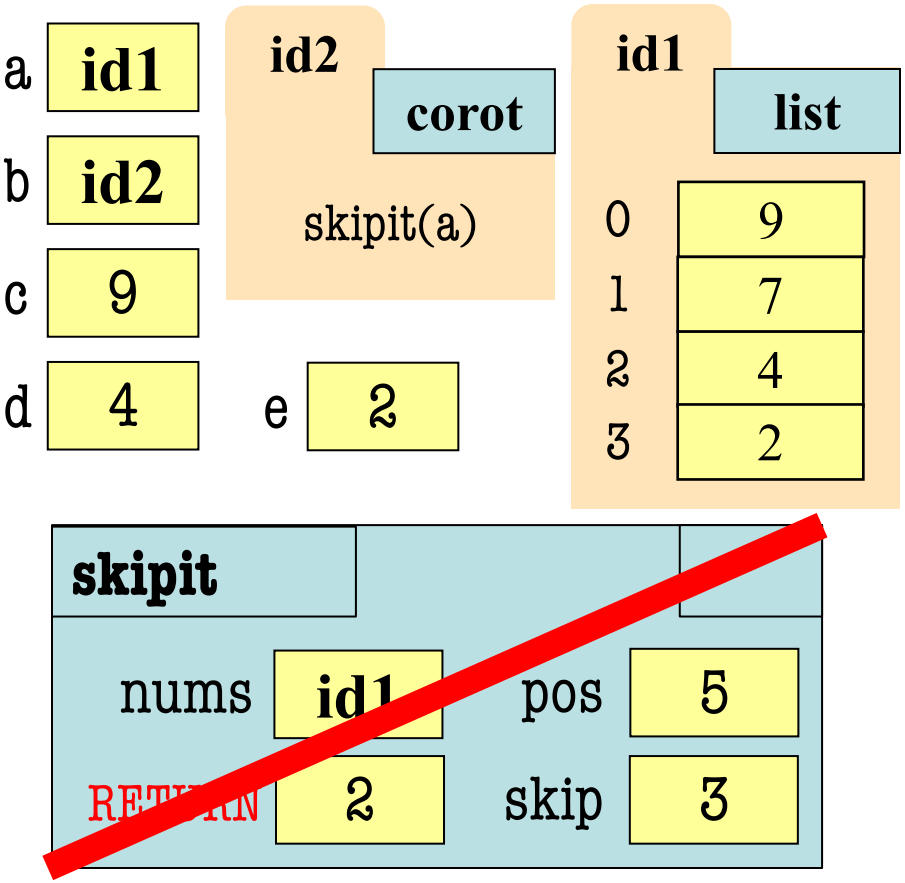


Coroutine with Input and Ouput

Coroutine

```
    def skipit(nums):
        """Send tells positions to skip
        Pre: nums is a list of ints"""
15      pos = 0
16      while pos < len(nums):
17          skip = (yield nums[pos])
18          pos = pos+skip
19      yield nums[-1]
```

Erase the Frame



Coroutines and Functions

Parent Function

```
    def sumfold(lst):
        """Returns list of sums"""
32      sum = []
33      g = pushsum(len(lst))
34      next(g)
35      for x in lst:
36          a = g.send(x)
37          sum.append(a)
38      return sum
```

Coroutine

```
    def pushsum(n):
        """Yields sum of all sent"""
16      sum = 0
17      for x in range(n):
18          val = (yield sum)
19          sum = sum+val
20      yield sum
```

Coroutines and Functions

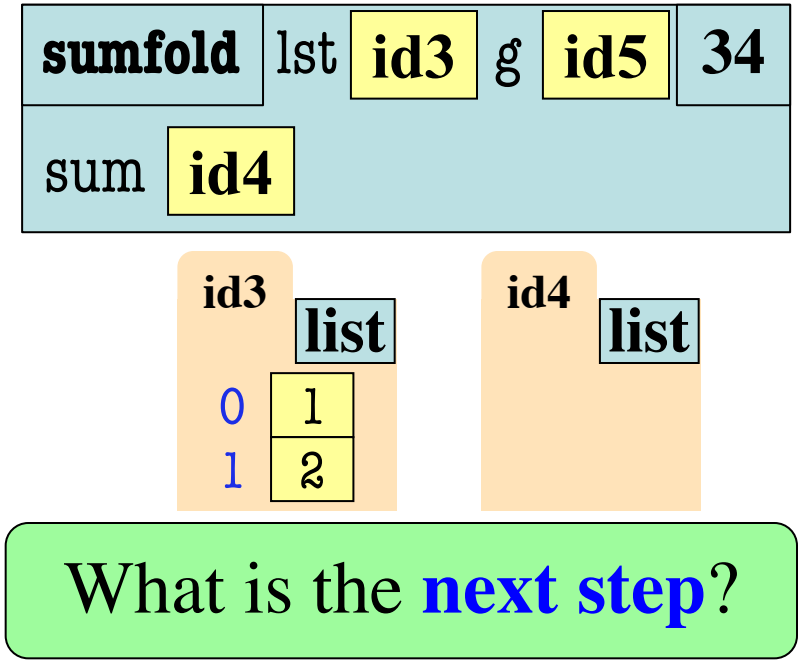
Parent Function

```
def sumfold(lst):
    """Returns list of sums"""
32  sum = []
33  g = pushsum(len(lst))
34  next(g)
35  for x in lst:
36      a = g.send(x)
37      sum.append(a)
38  return sum
```

Function Call

```
>>> x = sumfold([1,2])
```

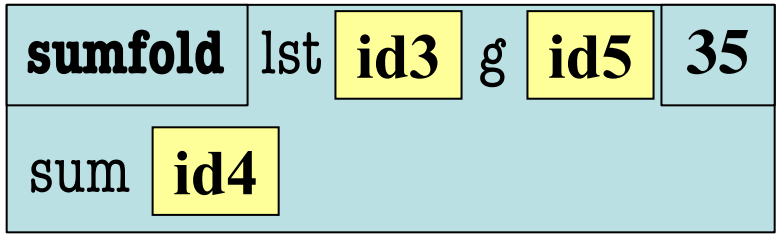
Assume we are here:



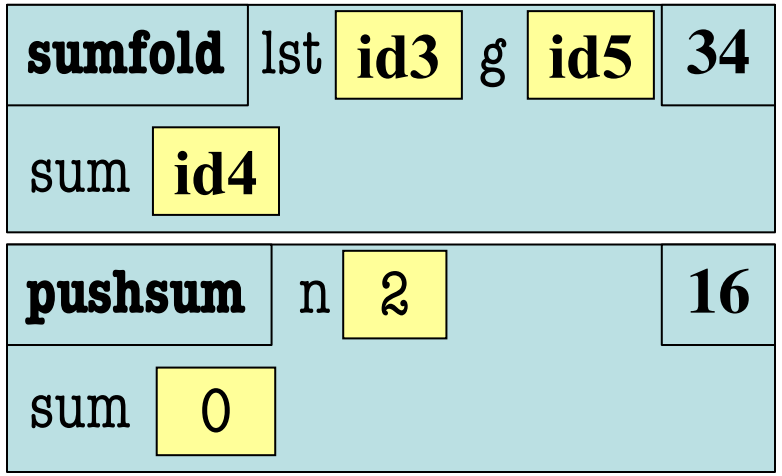
What is the **next step**?

Which One is Closest to Your Answer?

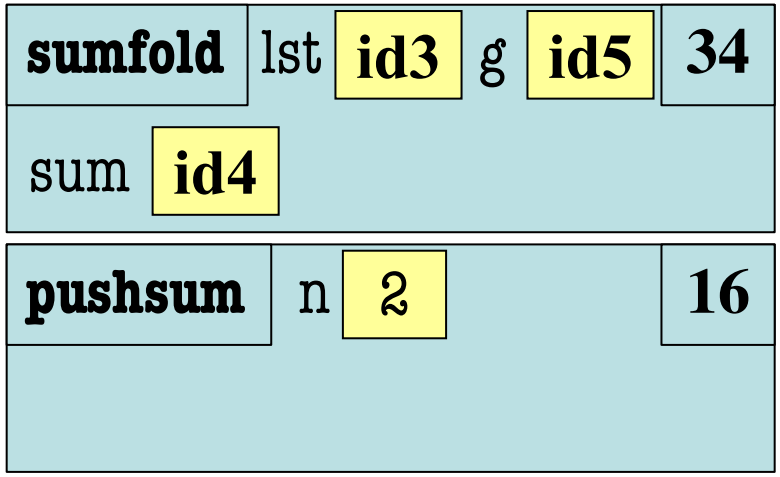
A:



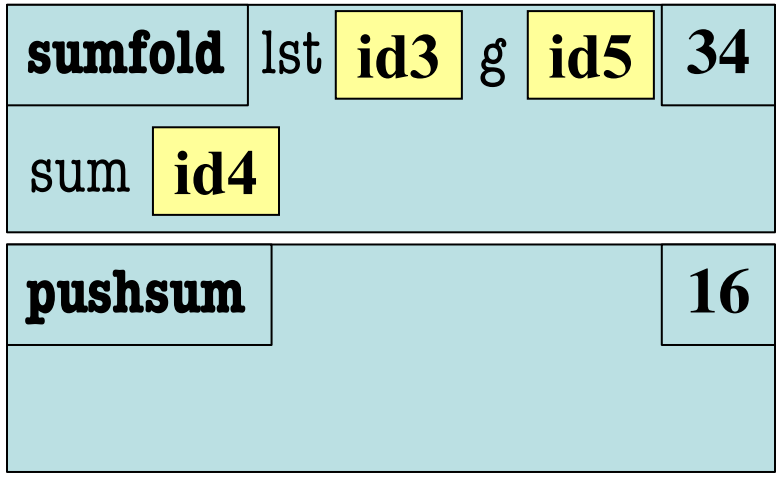
B:



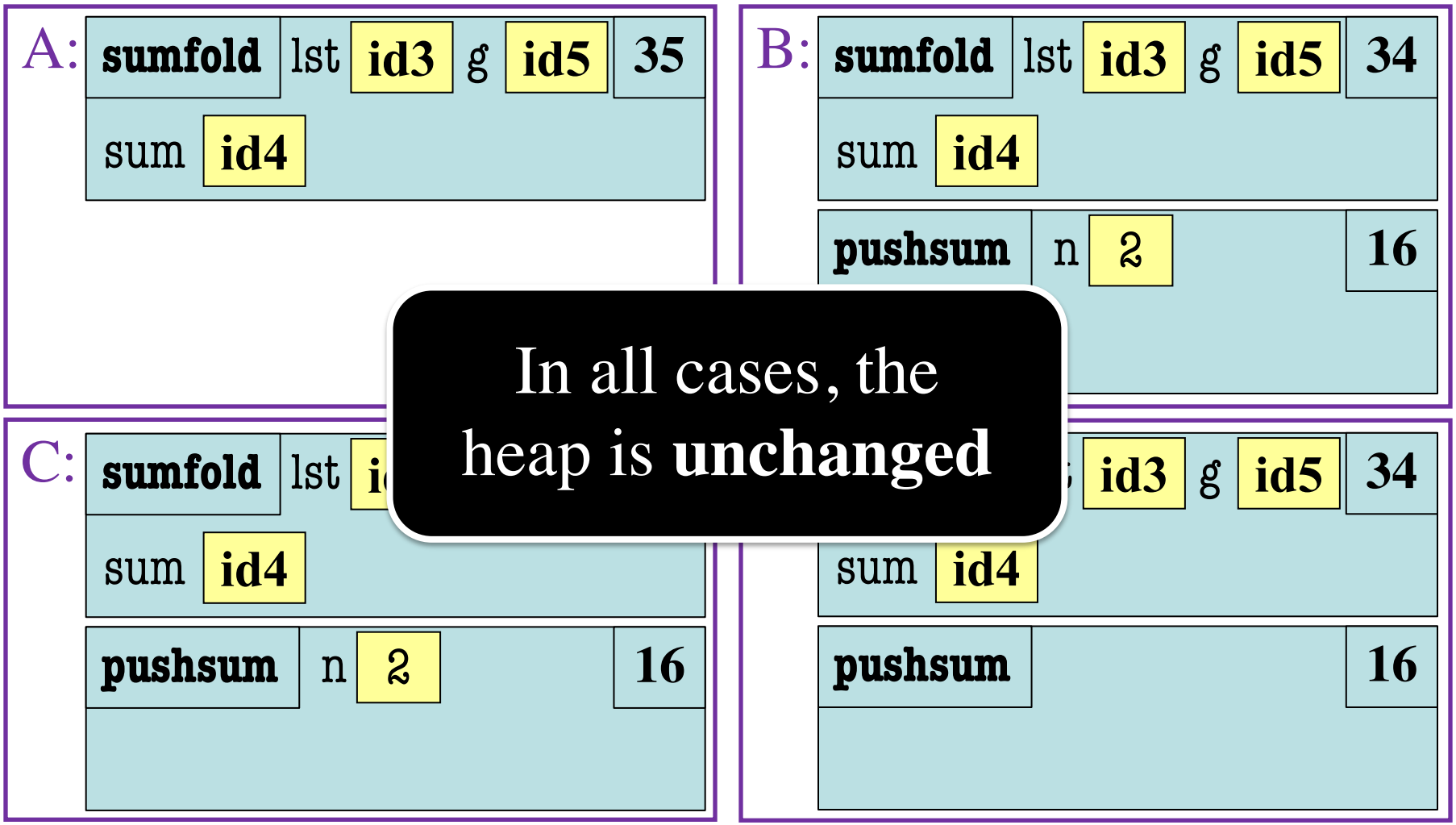
C:



D:



Which One is Closest to Your Answer?



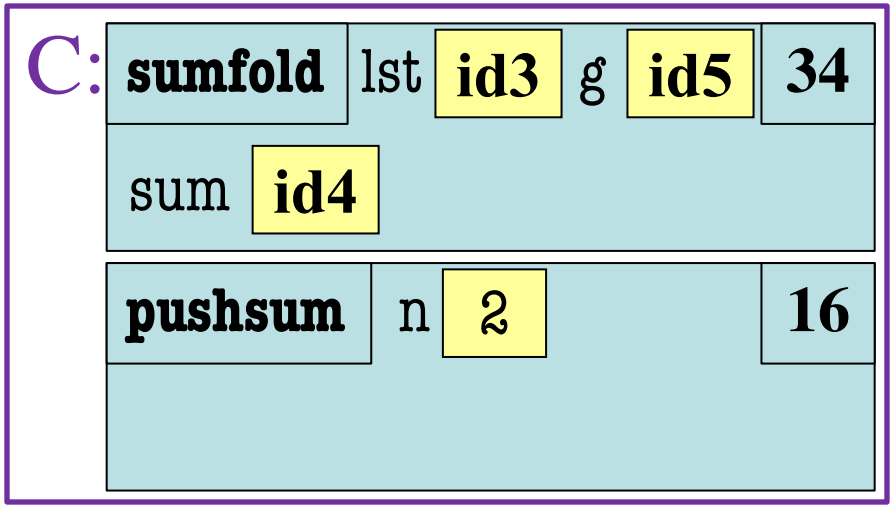
Coroutines and Functions

Coroutine

```
def pushsum(n):
    """Yields sum of all sent"""
16  sum = 0
17  for x in range(n):
18      val = (yield sum)
19      sum = sum+val
20  yield sum
```

Function Call

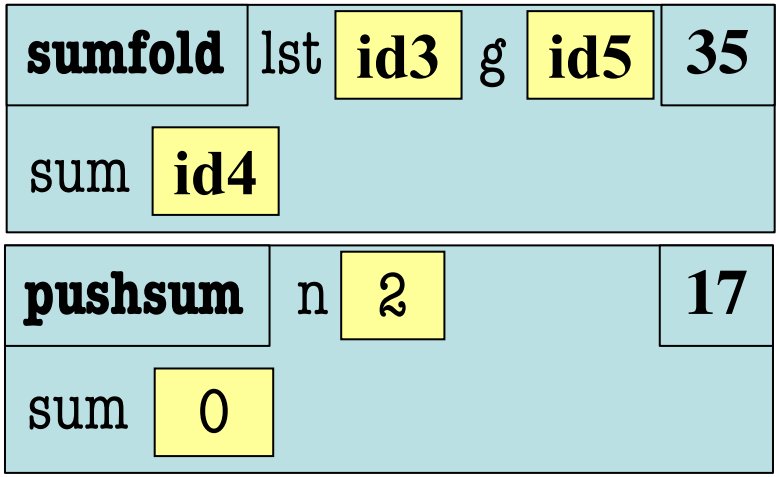
```
>>> x = sumfold([1,2])
```



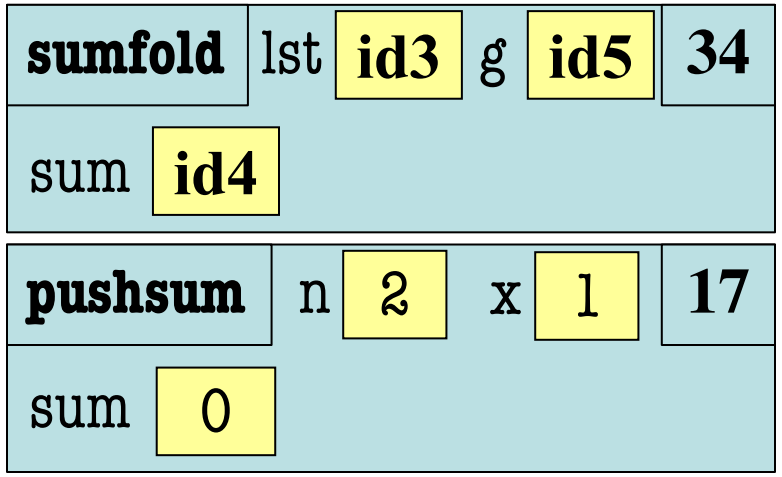
What is the **next step**?

Which One is Closest to Your Answer?

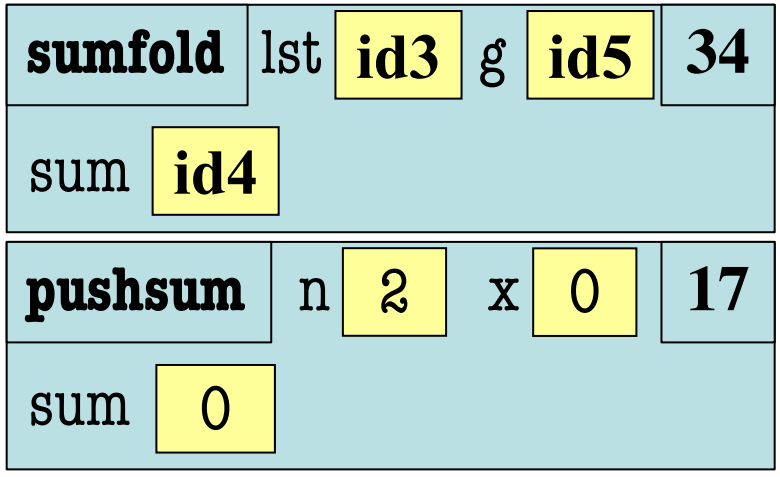
A:



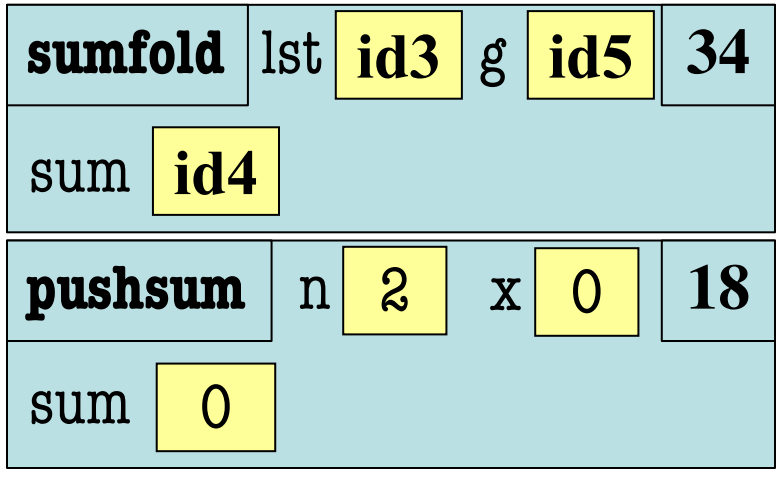
B:



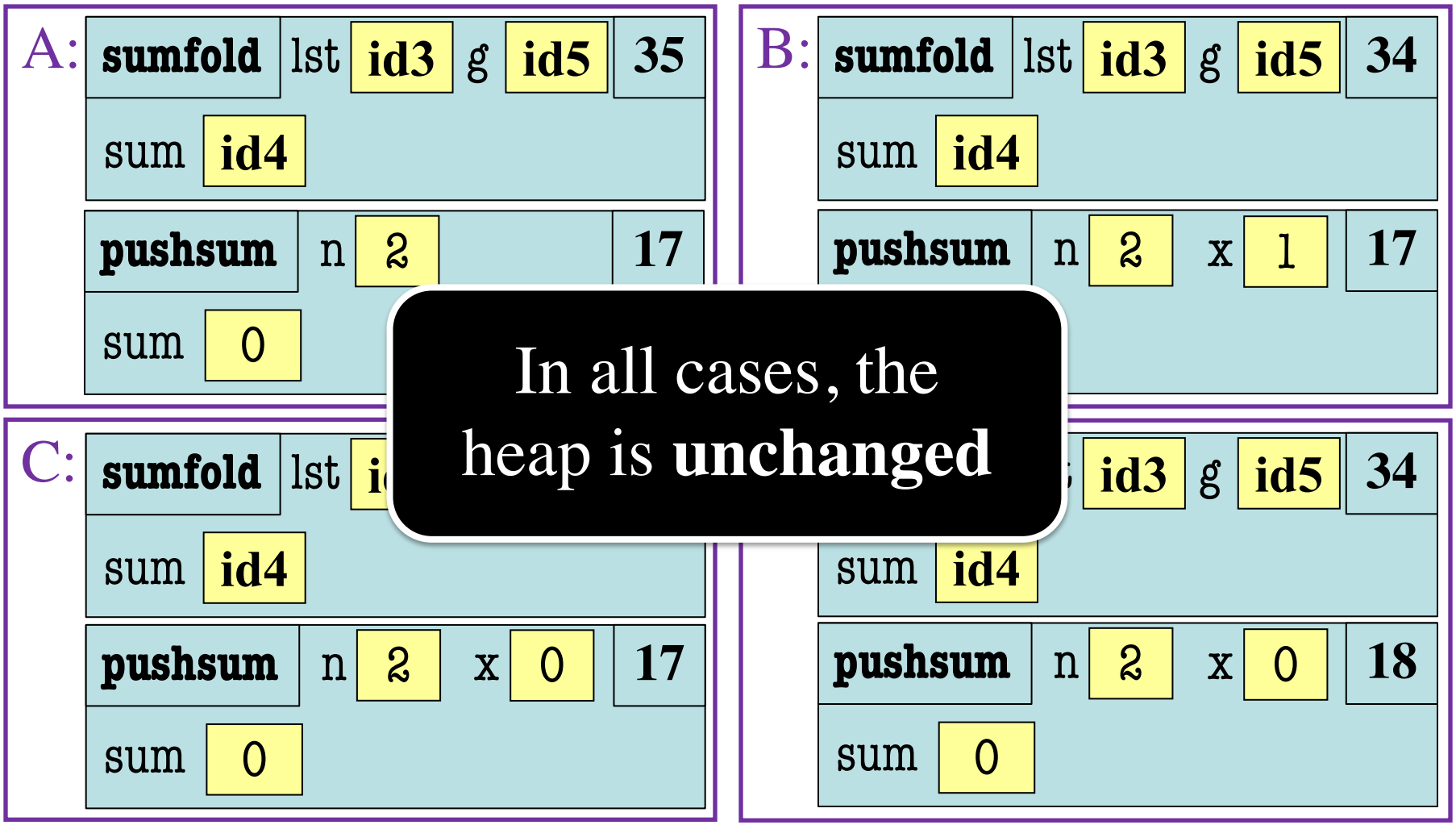
C:



D:



Which One is Closest to Your Answer?



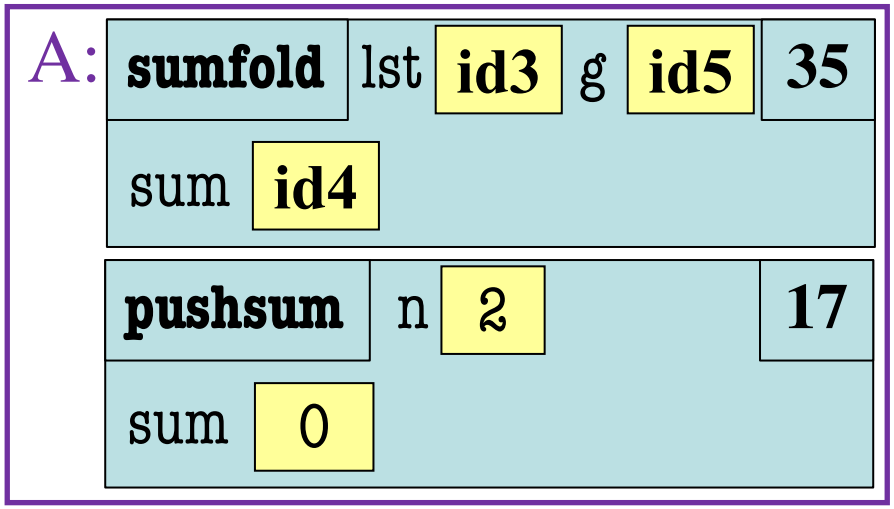
Coroutines and Functions

Coroutine

```
def pushsum(n):
    """Yields sum of all sent"""
16  sum = 0
17  for x in range(n):
18      val = (yield sum)
19      sum = sum+val
20  yield sum
```

Function Call

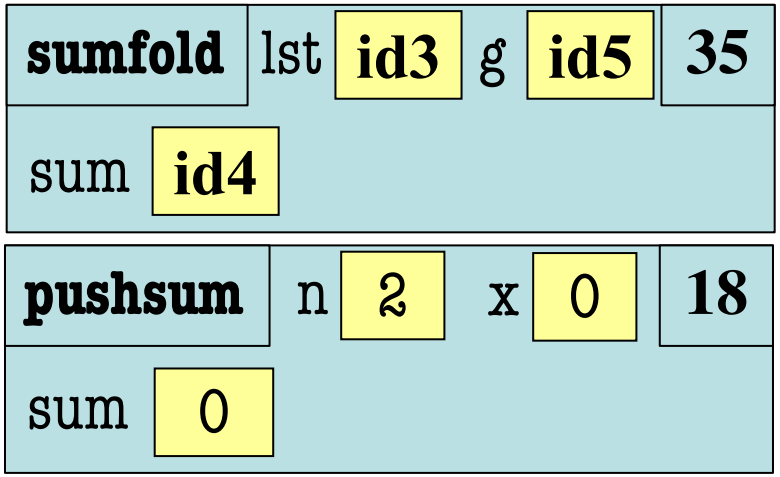
```
>>> x = sumfold([1,2])
```



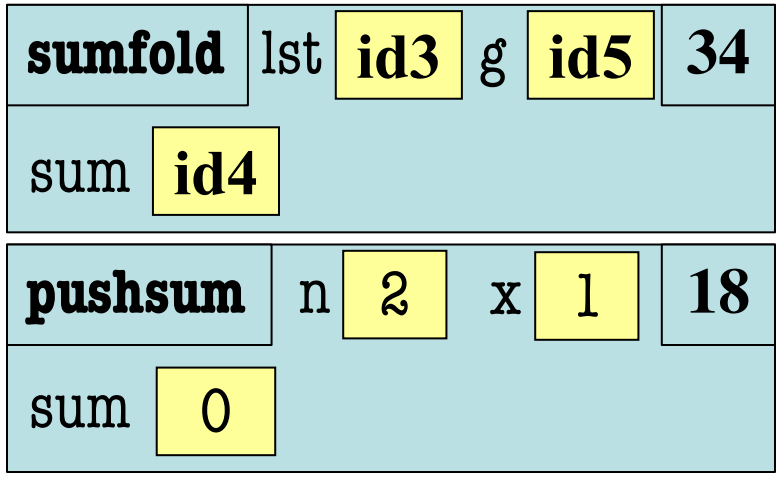
What is the **next step**?

Which One is Closest to Your Answer?

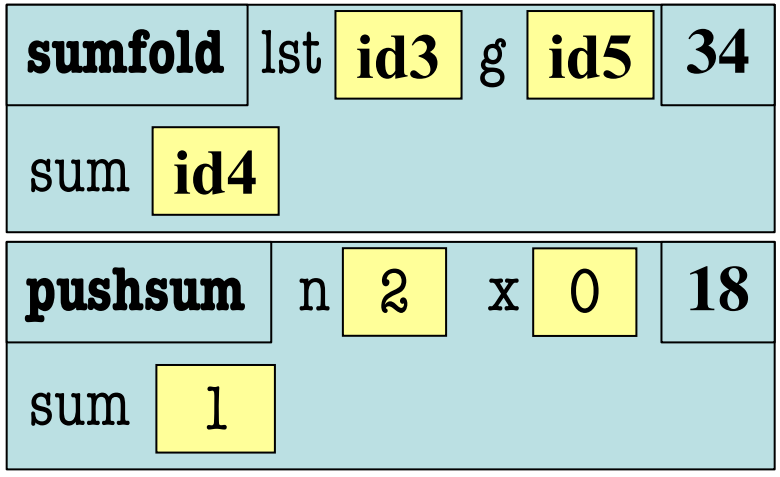
A:



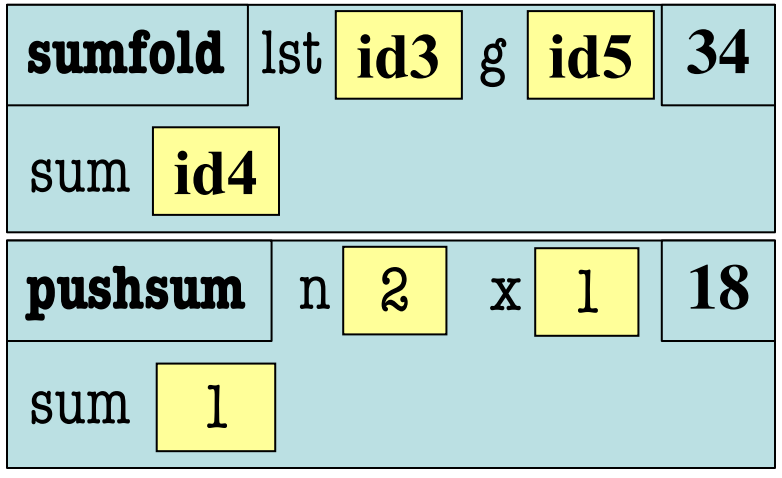
B:



C:



D:



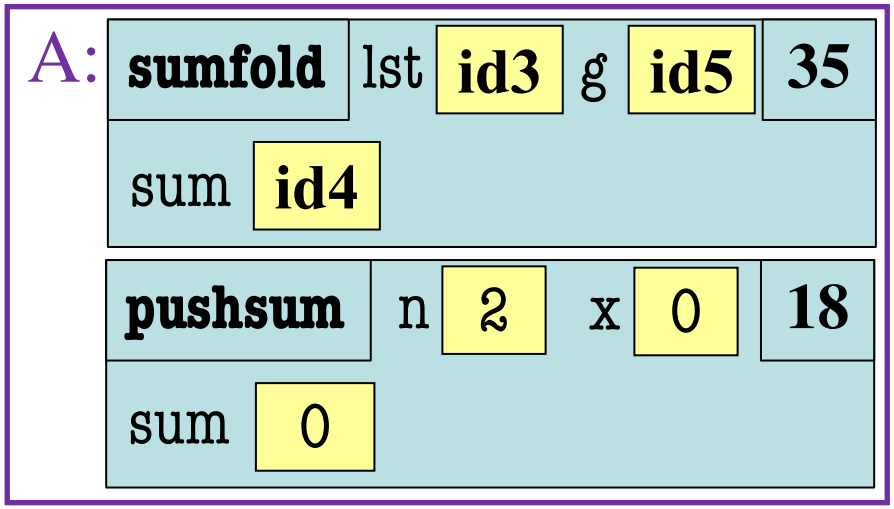
Coroutines and Functions

Coroutine

```
def pushsum(n):
    """Yields sum of all sent"""
16  sum = 0
17  for x in range(n):
18      val = (yield sum)
19      sum = sum+val
20  yield sum
```

Function Call

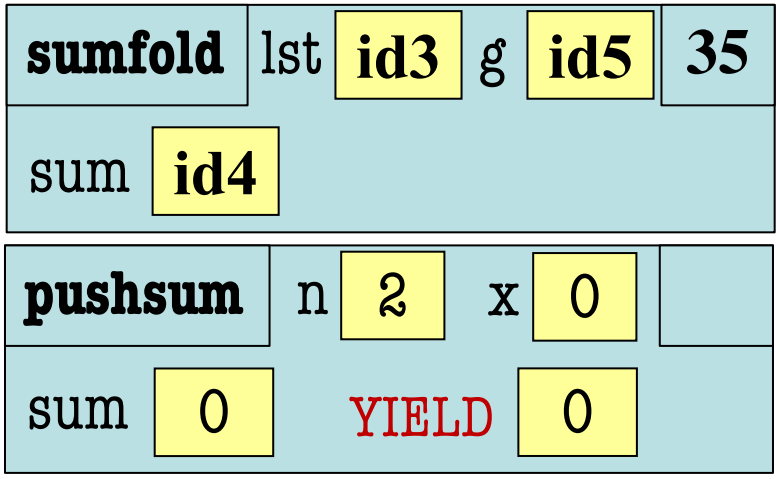
```
>>> x = sumfold([1,2])
```



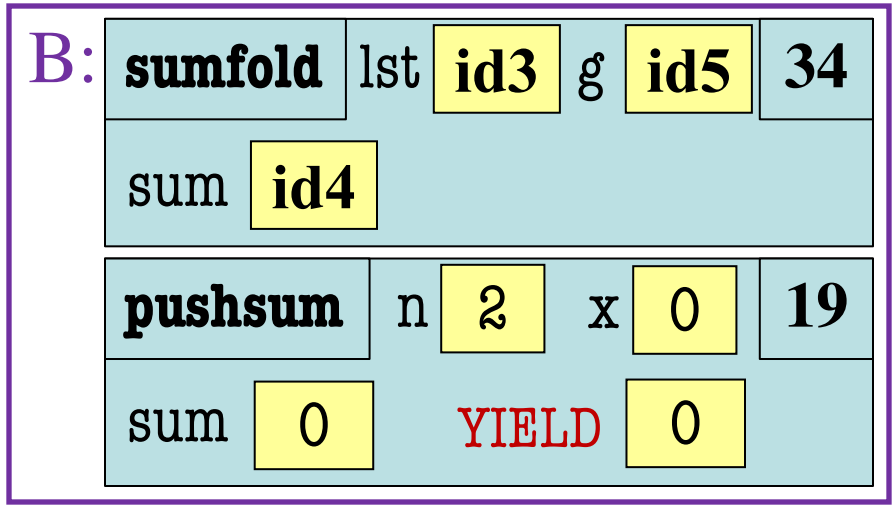
What is the **next step**?

Which One is Closest to Your Answer?

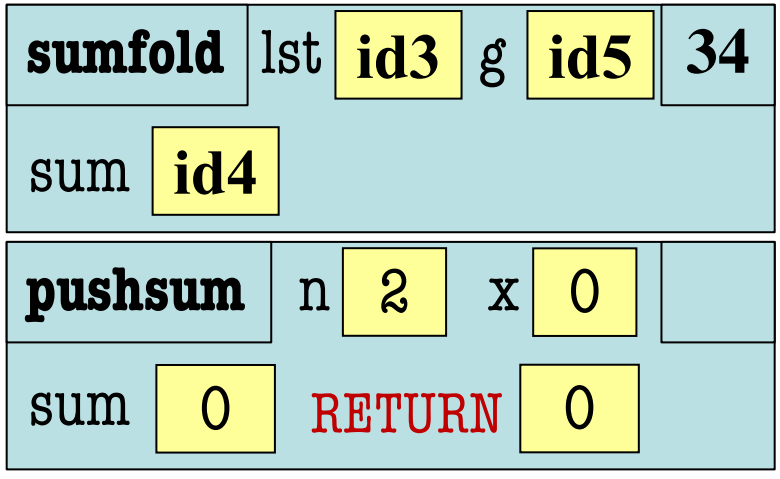
A:



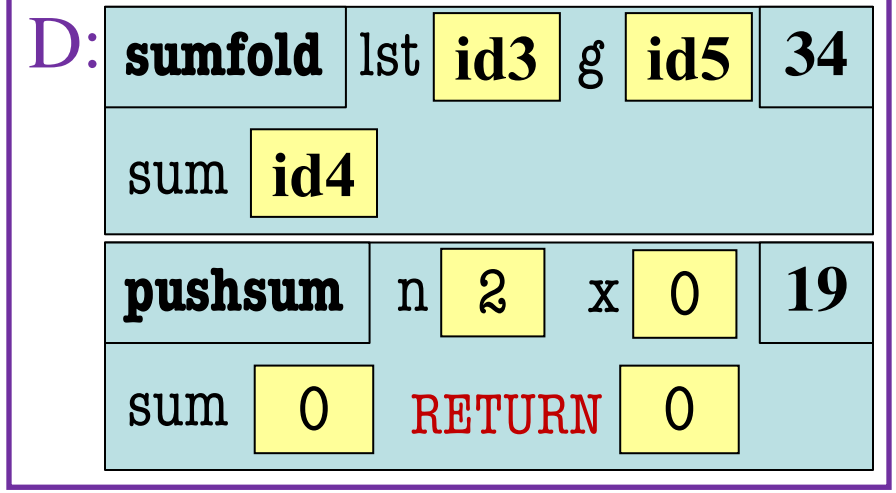
B:



C:



D:



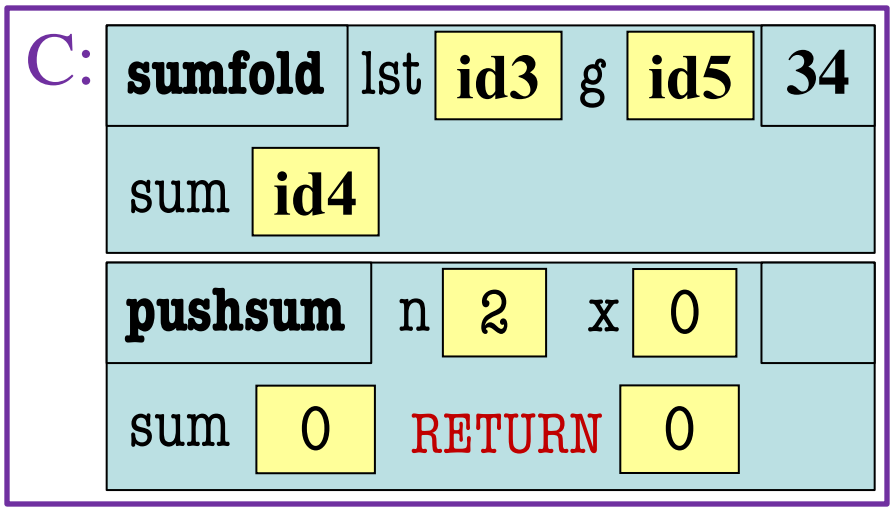
Coroutines and Functions

Parent Function

```
    def sumfold(lst):
        """Returns list of sums"""
32      sum = []
33      g = pushsum(len(lst))
34      next(g)
35      for x in lst:
36          a = g.send(x)
37          sum.append(a)
38      return sum
```

Function Call

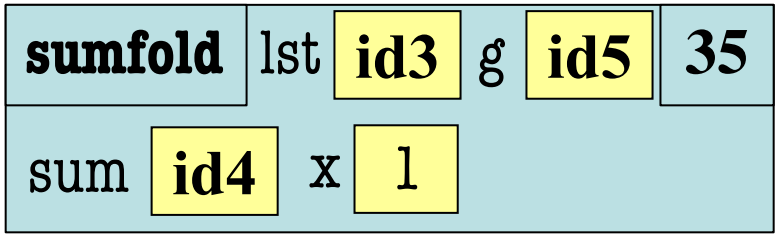
```
>>> x = sumfold([1,2])
```



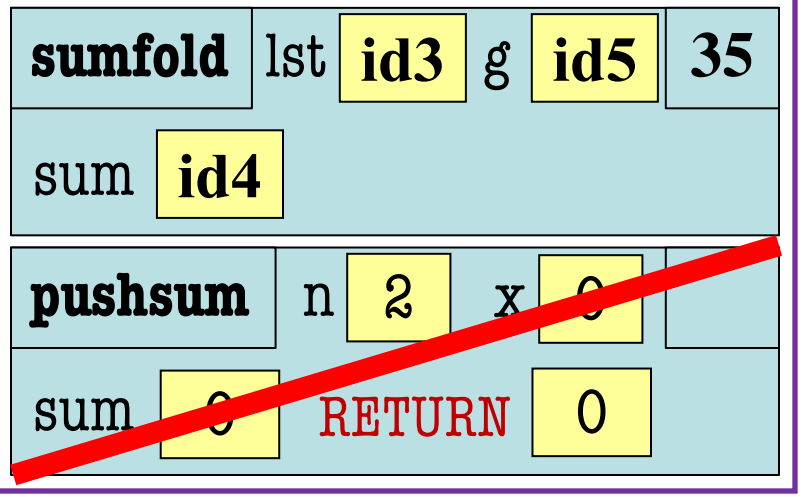
What is the **next step**?

Which One is Closest to Your Answer?

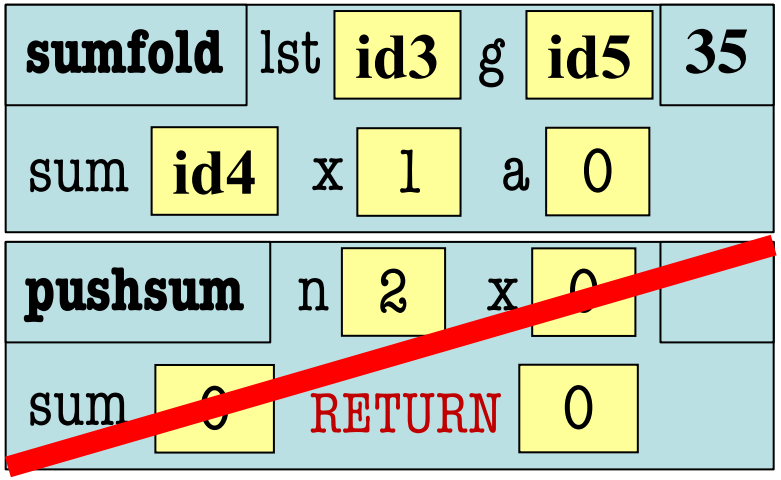
A:



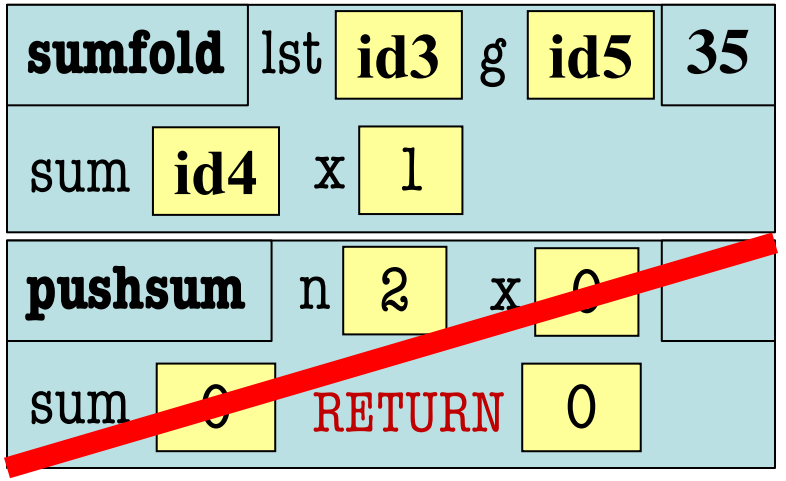
B:



C:



D:



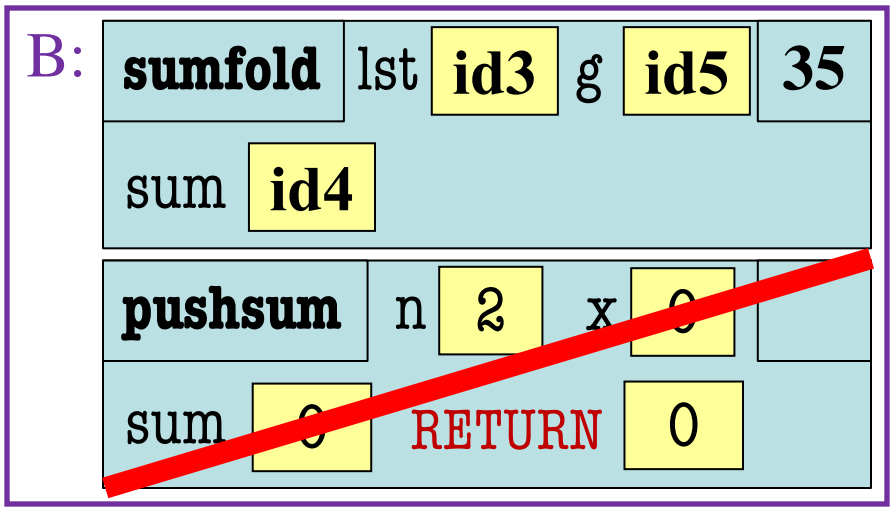
Coroutines and Functions

Parent Function

```
    def sumfold(lst):
        """Returns list of sums"""
32      sum = []
33      g = pushsum(len(lst))
34      next(g)
35      for x in lst:
36          a = g.send(x)
37          sum.append(a)
38      return sum
```

Function Call

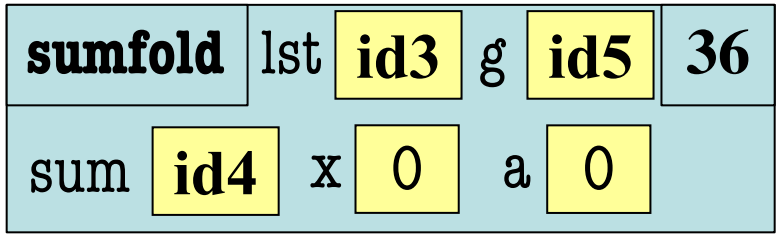
```
>>> x = sumfold([1,2])
```



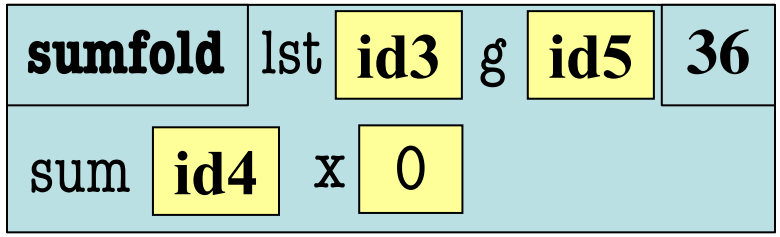
What is the **next step**?

Which One is Closest to Your Answer?

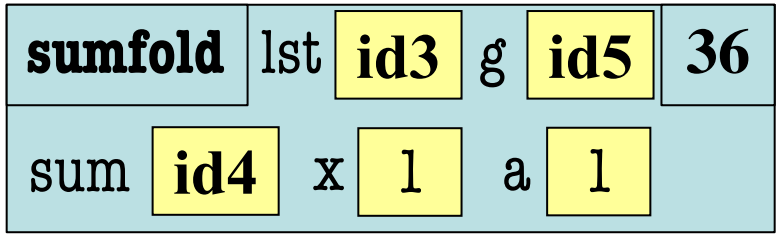
A:



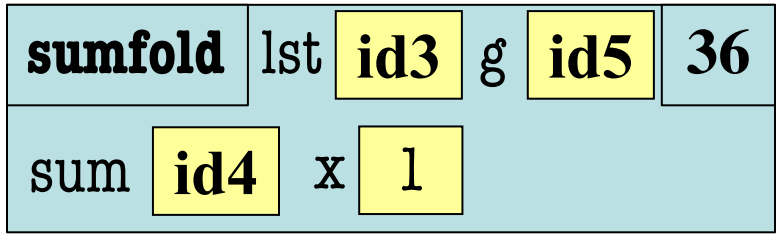
B:



C:



D:



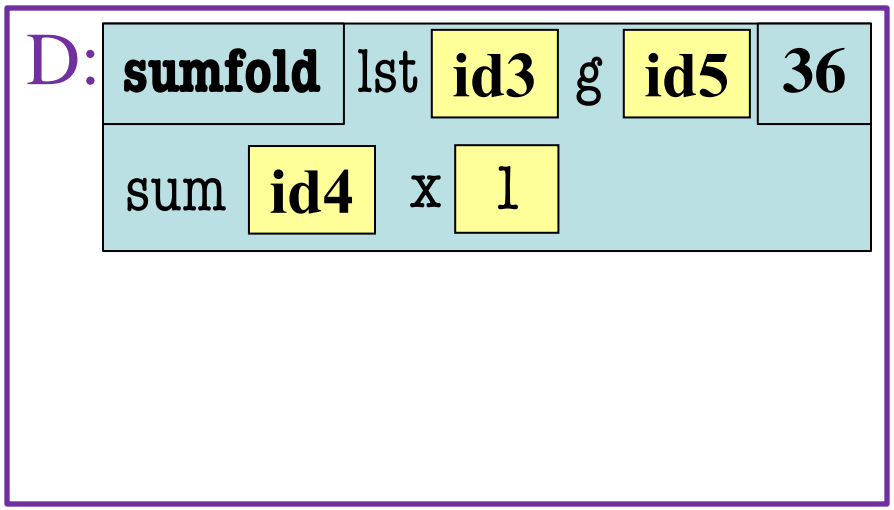
Coroutines and Functions

Parent Function

```
    def sumfold(lst):
        """Returns list of sums"""
32      sum = []
33      g = pushsum(len(lst))
34      next(g)
35      for x in lst:
36          a = g.send(x)
37          sum.append(a)
38      return sum
```

Function Call

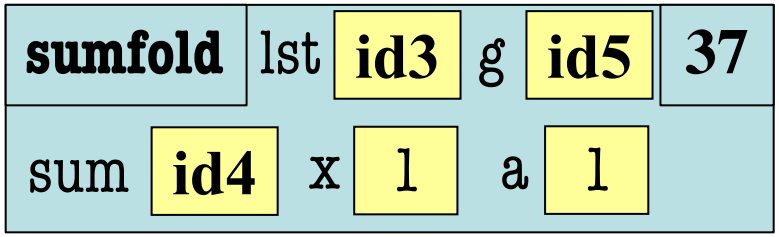
```
>>> x = sumfold([1,2])
```



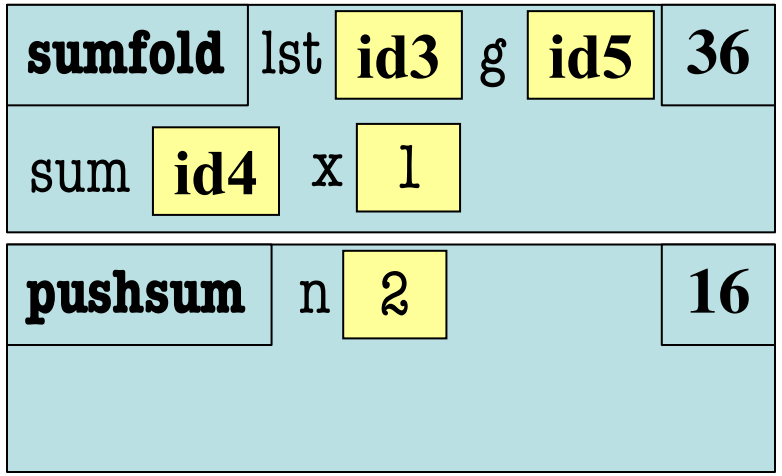
What is the **next step**?

Which One is Closest to Your Answer?

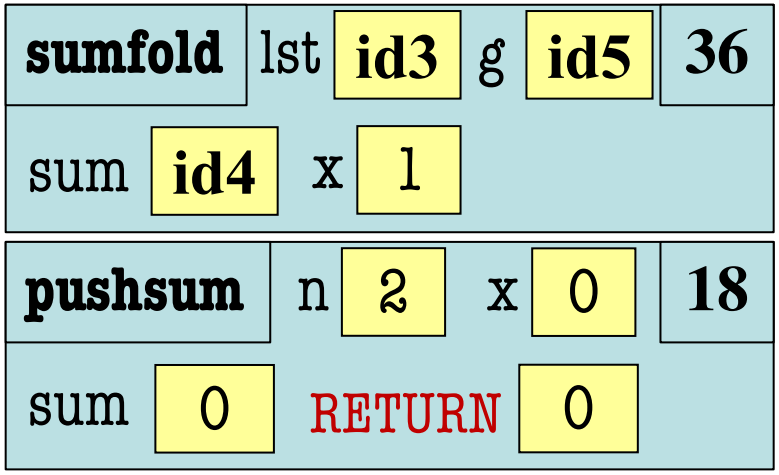
A:



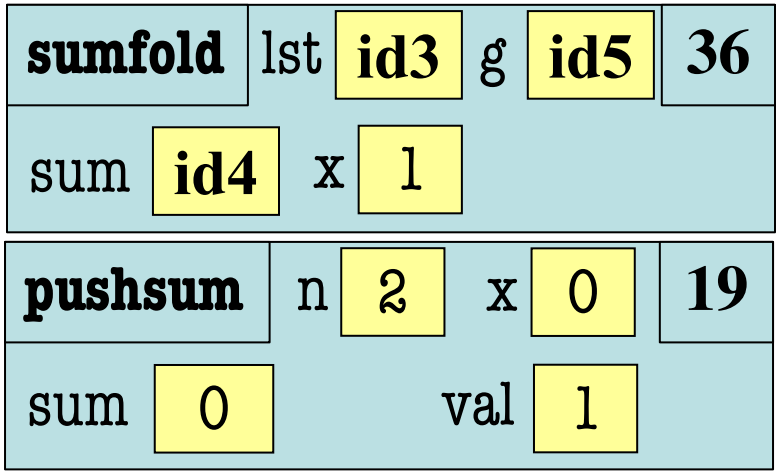
B:



C:



D:



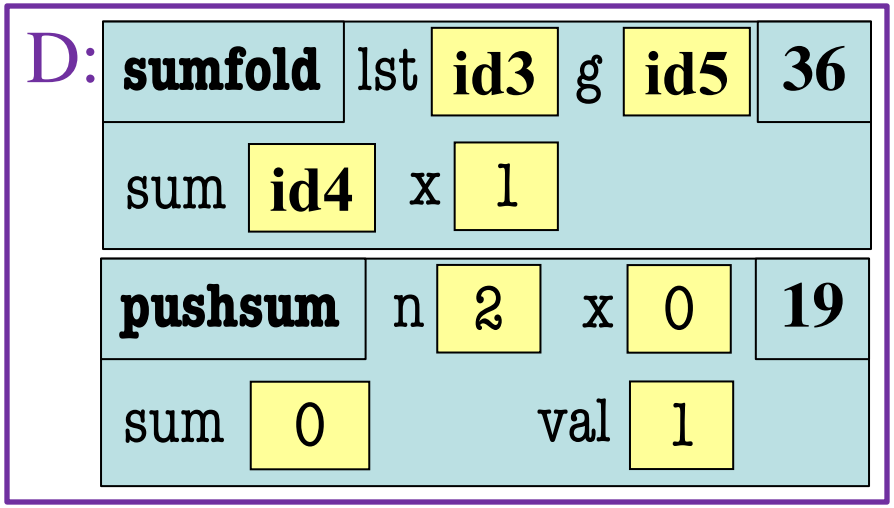
Coroutines and Functions

Coroutine

```
def pushsum(n):
    """Yields sum of all sent"""
16  sum = 0
17  for x in range(n):
18      val = (yield sum)
19      sum = sum+val
20  yield sum
```

Function Call

```
>>> x = sumfold([1,2])
```



Try the rest on your own

Questions?