Lecture 27

Generators

Announcements for This Lecture

Prelim 2

- **Thurs, Dec 4 at 7:30**
  - See webpage for rooms
  - Arranged by last name
- **Material up to Nov. 18**
  - Recursion + Loops + Classes
  - No short answer!
- **Make-Ups are Notified**
  - Only accepting emergencies
  - Contact Amy (ahf42)

Assignments

- **A6** is now graded
  - **Mean**: 90 **Median**: 95
  - **Std Dev**: 17
  - **Mean**: 12.8 hr **Median**: 13 hr
  - **Std Dev**: 5.3 hr
  - Several AI cases
- **A7** due **Mon, Dec. 8 (10th)**
  - Extensions are possible
  - Work on it during Thur/Fri

Even More Announcements

- **Final, Dec 13th 2-4:30 pm**
  - Will be in Barton Hall (all of it)
  - Study guide will be posted Thurs
  - We will also send out conflict form at that time
  - *Several* review sessions next week (details later)
- **Submit a course evaluation**
  - Will get an e-mail for this
  - Part of "participation grade" (e.g. polling grade)

Recall: The Range Iterable

range(x)

- Creates an *iterable*
  - Can be used in a for-loop
  - Makes ints (0, 1, ... x-1)
- But it is not a tuple!
  - A **black-box** for numbers
  - Entirely used in for-loop
  - Contents of folder hidden

Example

```
>>> range(3)
range(0,3)
>>> for x in range(3)
...       print(x)
0
1
2
```

Recall: The Range Iterable

range(x)

- Creates an *iterable*
  - Can be u
  - Makes i
- But it is n
  - A **black-**
  - Entirely used in for-loop
  - Contents of folder hidden

Example

```
>>> range(3)
```

```
e(3)
1
2
```

Iterable: Anything that can be used in a for-loop

Iterators: Iterables Outside of For-Loops

- Iterators can *manually* extract elements
  - Get each element with the `next()` function
  - Keep going until you reach the end
  - Ends with a `StopIteration` (Why?)
- Can create iterators with `iter()` function

```
>>> a = iter([1,5,3])
>>> next(a)
1
>>> next(a)
5
```

Must be a **iterable**

Iterators Can Be Used in For-Loops

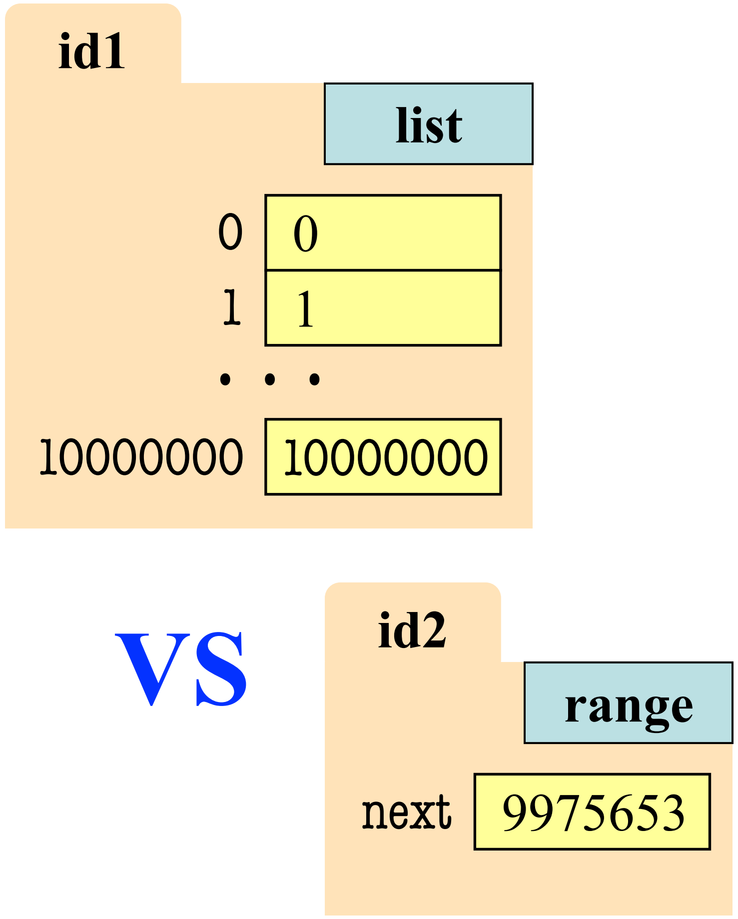
```
>>> a = iter([1,2])
>>> for x in a:
....     print(x)
....
1
2
>>> for x in a:
....     print(x)
....
>>>
```

Technically, iterators are also iterable

But they are one-use only!

Motivation for Iterables

- Large lists are a problem
  - Use a lot of space in heap
  - **Ex:** `list(range(10000001))`
- But do we need all this?
  - for-loop gets just one elt.
  - Only need the *next* value
- This is how `range` works
  - Stores the next value
  - *Generates* this on demand
  - More space efficient



Iterators are Classes

```
class range2iter(object):
    """Iterator class for squares of a range"""
    # Attribute _limit: end of range
    # Attribute _pos: current spot of iterator
    ...
    def __next__(self):
        """Returns the next element"""
        if self._pos >= self._limit:
            raise StopIteration()
        else:
            value = self._pos*self._pos
            self._pos += 1
            return value
```

Iterators are Classes

```
class range2iter(object):
    """Iterator class for squares of a range"""
    # Attribute _limit: end of range
    # Attribute _pos: current s
    ...
    def __next__(self):
        """Returns the next element"""
        if self._pos >= self._limit:
            raise StopIteration()
        else:
            value = self._pos*self._pos
            self._pos += 1
            return value
```

Defines the next() fcn

Iterators are Classes

```
class range2iter(object):
    """Iterator class for squares of a range"""
    # Attribute _limit: end of range
    # Attribute _pos: current spot of iterator
    ...
    def __next__(self):
        """Returns the next element"""
        if self._pos >= self._limit:
            raise StopIteration()
        else:
            value = self._pos*self._pos
            self._pos += 1
            return value
```

How far to go

How far we are

Raise error when gone too far

Iterators are Classes

```
class range2iter(object):
    """Iterator class for squares of a range"""
    # Attribute _limit: end of range
    # Attribute _pos: current spot of iterator
    ...
    def __next__(self):
        """Returns the next element"""
        if self._pos >= self._limit:
            raise StopIteration()
        else:
            value = self._pos*self._pos
            self._pos += 1
            return value
```

Update "loop" **after** doing computation

Essentially a loop variable

Iterables are Also Classes

```
class range2(object):
    """Iterable class for squares of a range"""

    def __init__(self,n):
        """Initializes a squares iterable"""
        self._limit = n

    def __iter__(self):
        """Returns a new iterator"""
        return range2iter(self._limit)
```

Defines the iter() function

Returns an iterator

Iterables are Also Classes

```
class range2(object):
    """Iterable class for squares of a range"""

    def __init__(self,n):
        """Initializes a squares iter
        self._limit = n

    def __iter__(self):
        """Returns a new iterator"""
        return range2iter(self._limit)
```

Iterables are objects that generate **iterators** on demand

Iterators are Hard to Write!

- Has the same problem as GUI applications
  - We have a hidden loop
  - All loop variables are now attributes
  - Similar to inter-frame/intra-frame reasoning
- Would be easier if loop were **not** hidden
  - **Idea:** Write this as a function definition
  - Function makes loop/loop variables visible
- But iterators "return" multiple values
  - So how would this work?

The Wrong Way

```
def range2iter(n):
    """
    Iterator for the squares of numbers 0 to n-1

    Precondition: n is an int >= 0
    """
    for x in range(n):
        return x*x
```

Stops at the first value

The **yield** Statement

- **Format**: yield <*expression*>
  - Used to produce a value
  - But it **does not stop** the "function"
  - Useful for making iterators
- **But**: These are not normal functions
  - Presence of a yield makes a **generator**
  - Function that returns an iterator

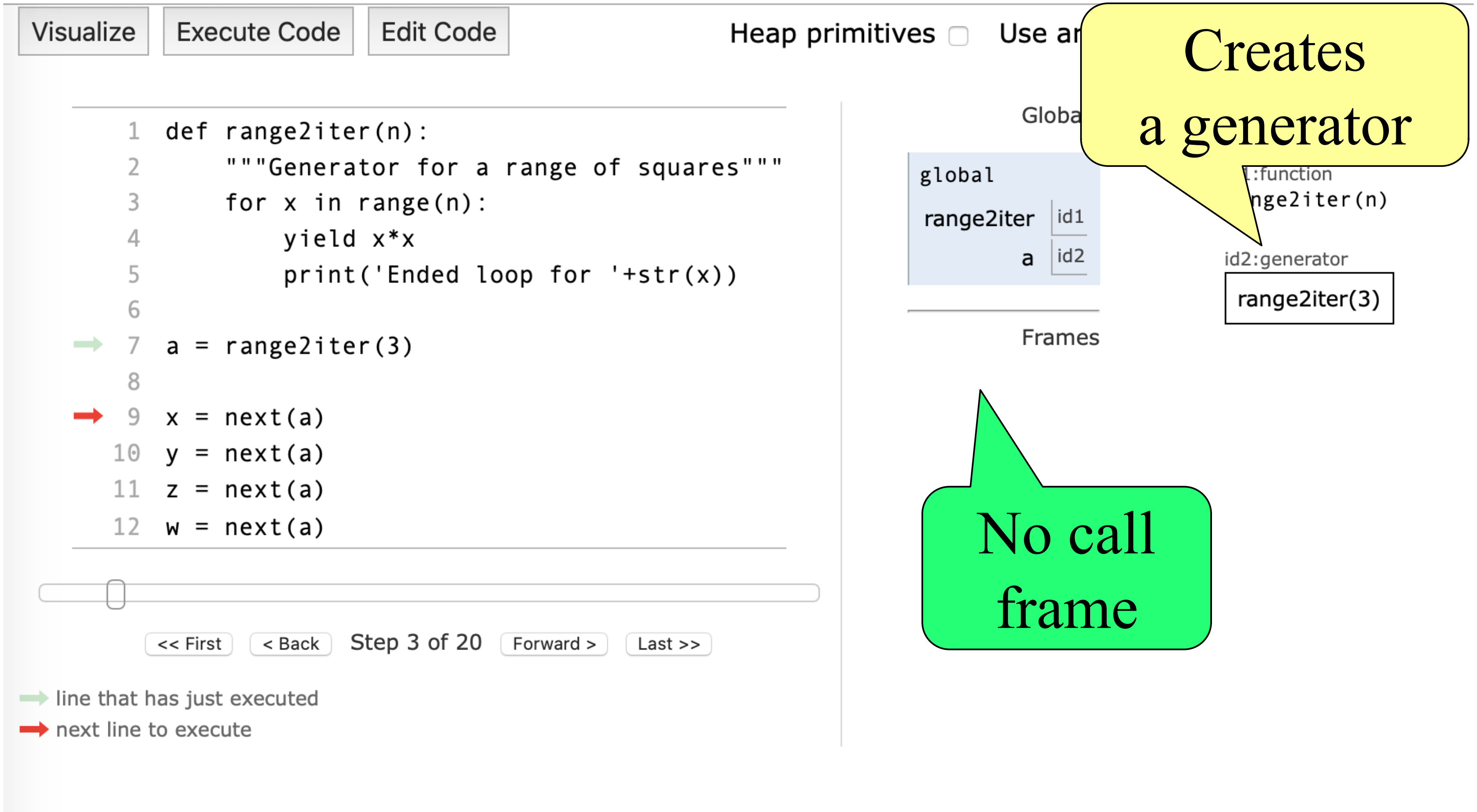
The Generator approach

```python
def range2iter(n):
    """
    Generator for the squares
    of numbers 0 to n-1

    Precon: n is an int >= 0
    """
    for x in range(n):
        yield x*x
```

```
>>> a = range2iter(3)
>>> a
<generator
>>> next(a)
0
>>> next(a)
1
>>> next(a)
4
```

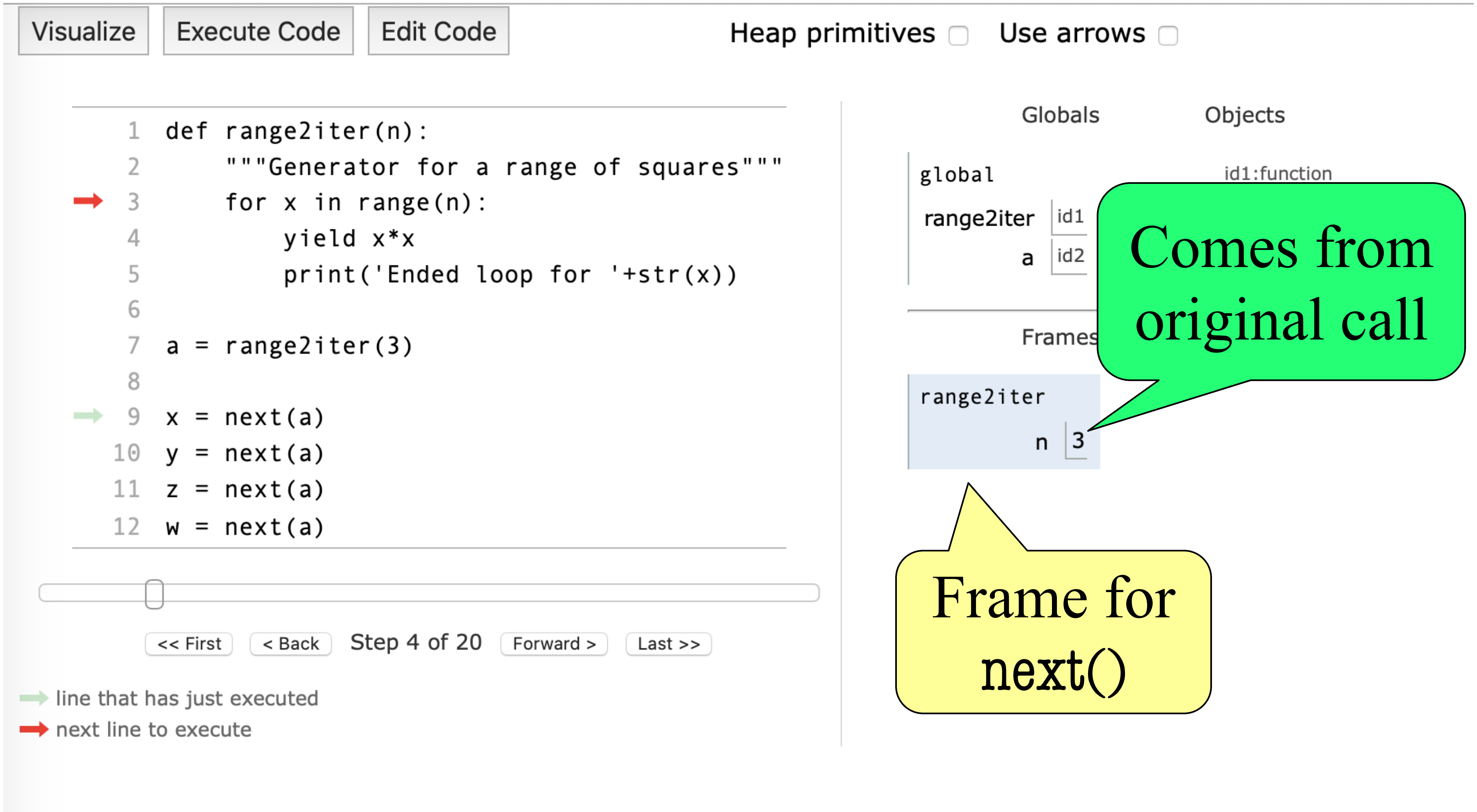
Essentially a constructor

What Happens on a Function Call?

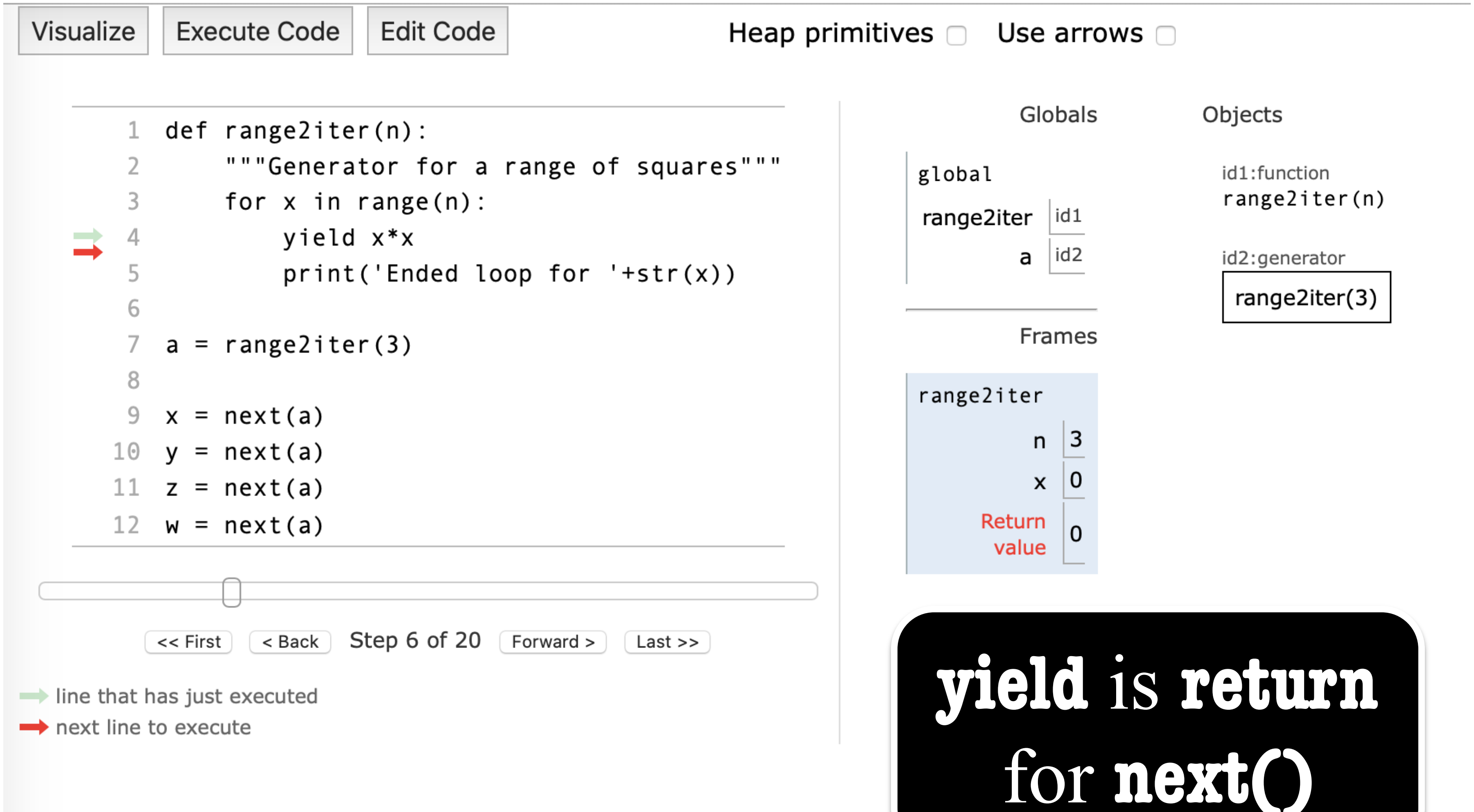



```python
def range2iter(n):
    """Generator for a range of squares"""
    for x in range(n):
        yield x*x
        print('Ended loop for '+str(x))

a = range2iter(3)

x = next(a)
y = next(a)
z = next(a)
w = next(a)
```

next() Initiates a Function Call



Call Finishes at the yield



yield is **return** for **next()**

Later Calls Resume After the **yield**

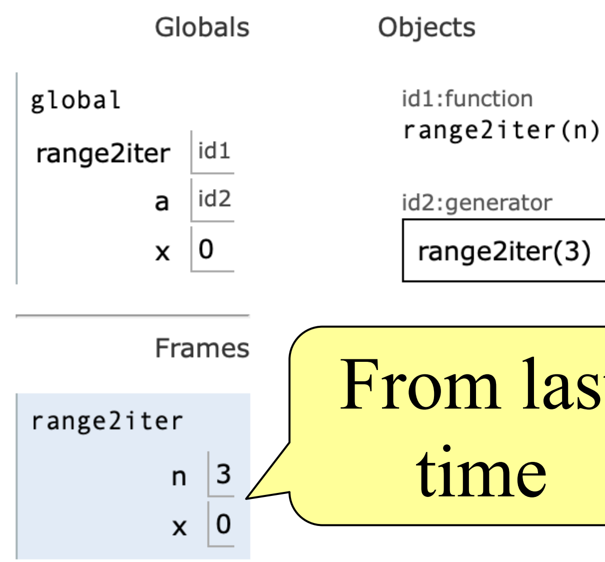
Visualize | Execute Code | Edit Code — Heap primitives ☐ Use arrows ☐

```python
def range2iter(n):
    """Generator for a range of squares"""
    for x in range(n):
        yield x*x
        print('Ended loop for '+str(x))

a = range2iter(3)

x = next(a)
y = next(a)
z = next(a)
w = next(a)
```

<< First | < Back | Step 8 of 20 | Forward > | Last >>

line that has just executed

next line to execute



From last time

Next call returns to where it left off

Exception is Made Automatically

Visualize | Execute Code | Edit Code | Heap primitives ☐ | Use arrows ☐

```
def range2iter(n):
    """Generator for a range of squares"""
    for x in range(n):
        yield x*x
        print('Ended loop for '+str(x))

a = range2iter(3)

x = next(a)
y = next(a)
z = next(a)
w = next(a)
```

<< First | < Back | Program terminated | Forward > | Last >>

StopIteration:

Globals | Objects

global	
range2iter	id1
x	0
y	1
z	4

id1:function
range2iter(n)

Frames

Exception when generator is done

Return Statements Make Exceptions

Visualize | Execute Code | Edit Code | Heap primitives | Use arrows

```
def range2iter(n):
    """Generator for a range of squares"""
    for x in range(n):
        yield x*x
        print('Ended loop for '+str(x))
    return x # The final x

a = range2iter(3)

x = next(a)
y = next(a)
z = next(a)
w = next(a)
```

<< First | < Back | Program terminated | Forward > | Last >>

StopIteration: 2

Return Value

Globals | Objects

global	
range2iter	id1
x	0
y	1
z	4

id1:function
range2iter(n)

Frames

Exception when generator is done

Activity: Call Frame Time

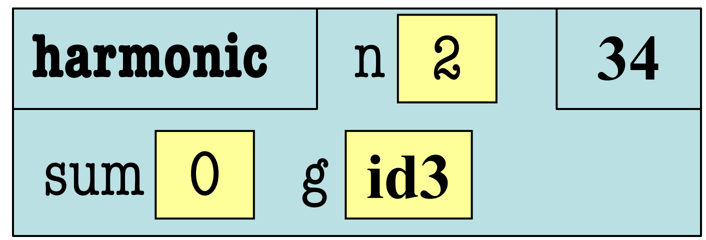
Function Defintions

```
    def rnginv(n):        #Inverse range
19      for x in range(1,n):
20          yield 1/x

    def harmonic(n):      #Harmonic sum
32      sum = 0
33      g = rnginv(n)
34      for x in g:
35          sum = sum+x
36      return x
```

Function Call

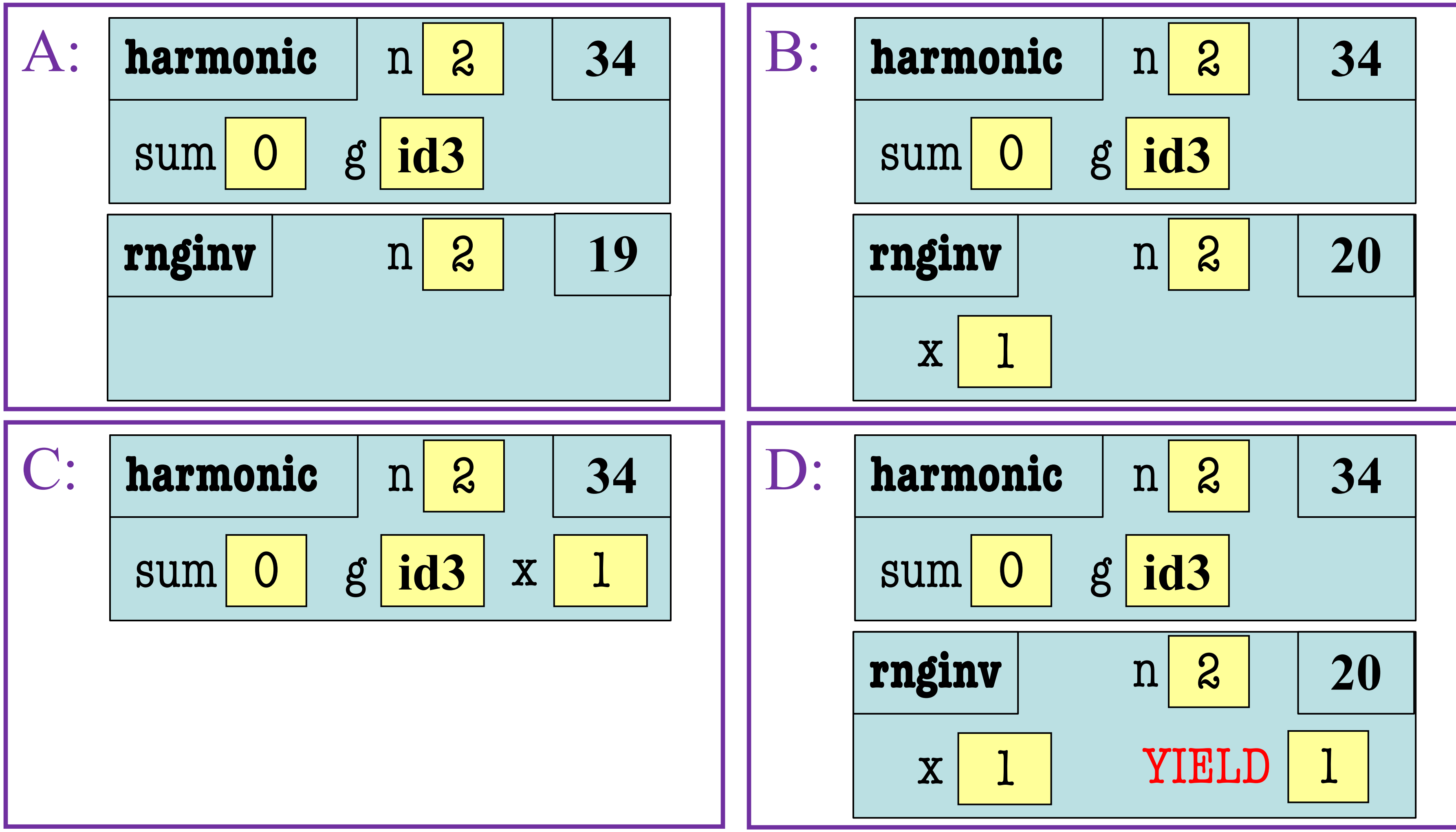
```
>>> x = harmonic(2)
```

Assume we are here:

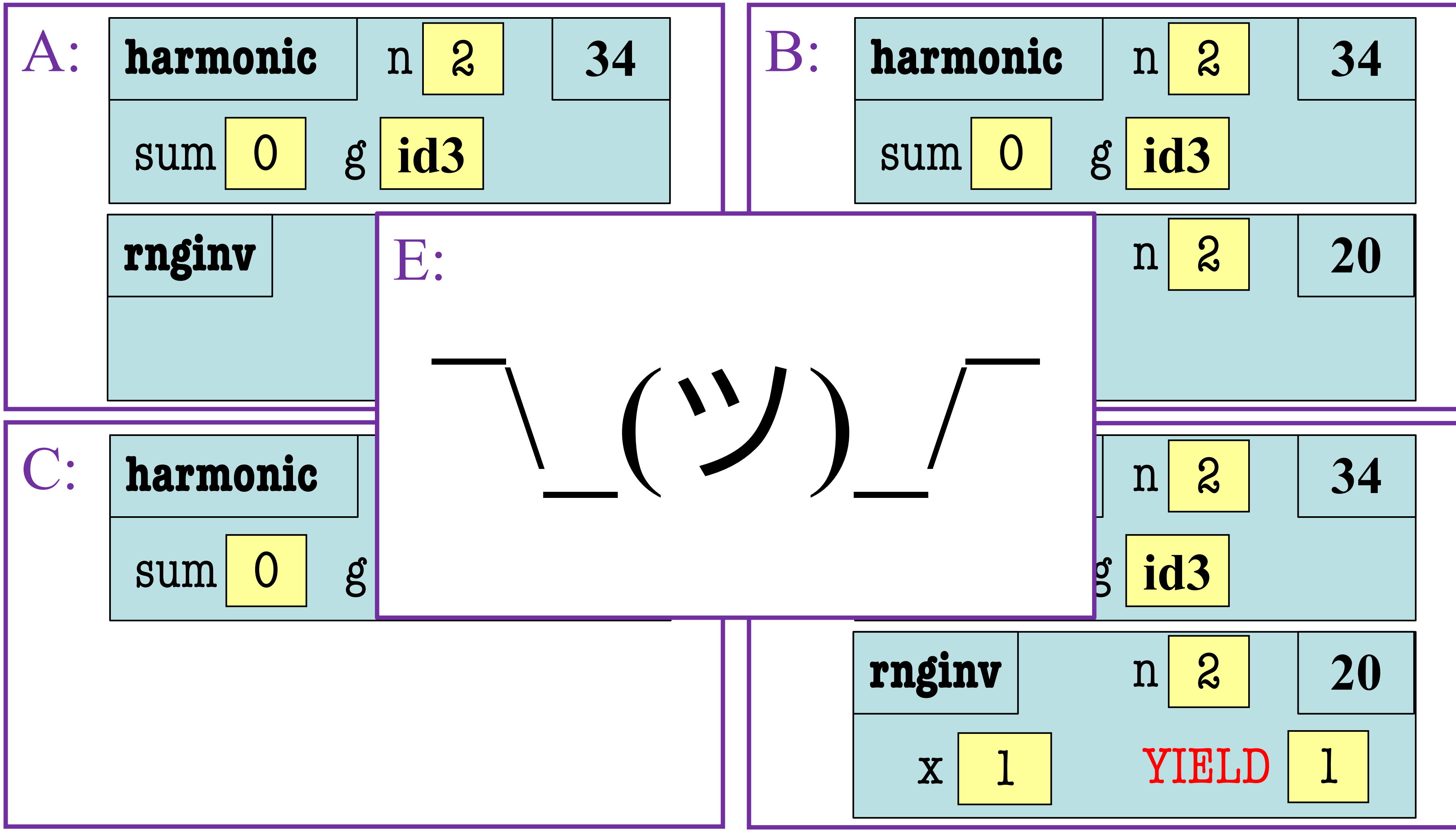


Ignoring the heap, what is the **next step**?

Which One is Closest to Your Answer?



Which One is Closest to Your Answer?



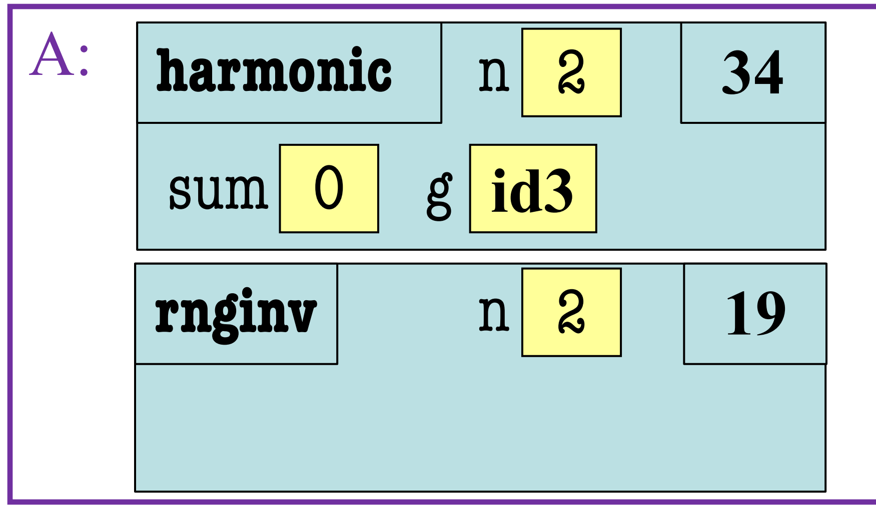
Activity: Call Frame Time

Function Defintions

```
   def rnginv(n):        #Inverse range
19   for x in range(1,n):
20     yield 1/x

   def harmonic(n):     #Harmonic sum
32   sum = 0
33   g = rnginv(n)
34   for x in g:
35     sum = sum+x
36   return x
```

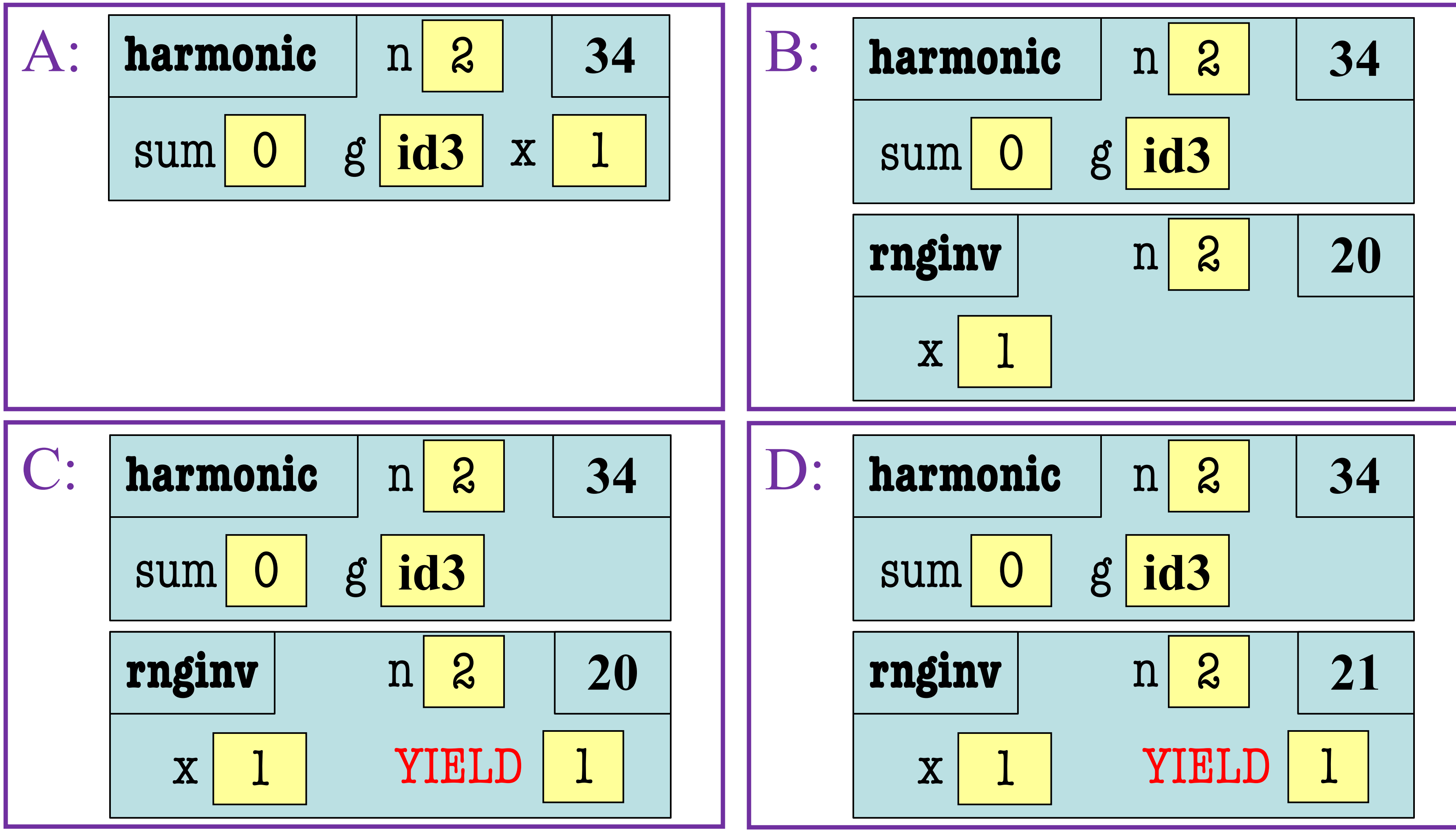
Function Call

```
>>> x = harmonic(2)
```



What is the **next step**?

Which One is Closest to Your Answer?



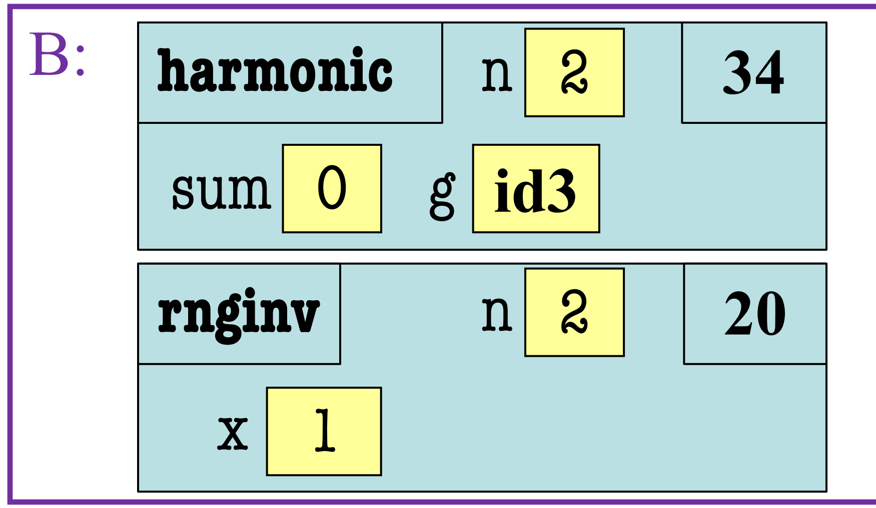
Activity: Call Frame Time

Function Defintions

```
    def rnginv(n):          #Inverse range
19      for x in range(1,n):
20          yield 1/x

    def harmonic(n):        #Harmonic sum
32      sum = 0
33      g = rnginv(n)
34      for x in g:
35          sum = sum+x
36      return x
```

Function Call

```
>>> x = harmonic(2)
```



What is the **next step**?

Which One is Closest to Your Answer?

A:

harmonic	n	2	34
sum 0	g id3	x 1	

B:

harmonic	n	2	34
sum 0	g id3	x 1	

rnginv	n	2	19
x 1	YIELD 1		

C:

harmonic	n	2	34
sum 0	g id3		

rnginv	n	2	
x 1	YIELD 1		

D:

harmonic	n	2	34
sum 0	g id3		

rnginv	n	2	
x 1	RETURN 1		

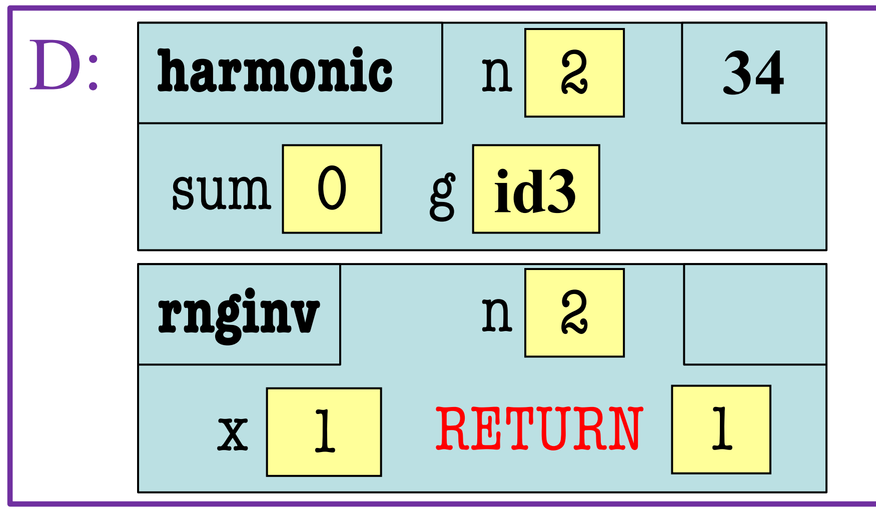
Activity: Call Frame Time

Function Defintions

```
def rnginv(n):        #Inverse range
    for x in range(1,n):
        yield 1/x

def harmonic(n):    #Harmonic sum
    sum = 0
    g = rnginv(n)
    for x in g:
        sum = sum+x
    return x
```

Function Call

```
>>> x = harmonic(2)
```



Generators Are Easy

- They replace the **accumulator pattern**
  - Function input is an iterable (string, list, tuple)
  - Function output typically a transformed copy
  - **Old way:** Accumulate a new list or tuple
  - **New way:** Yield one element at a time
- New way makes an **iterator** (not **iterable**)
  - So can only be used once!
  - But easily turned into a list or tuple

Accumulators: The Old Way

```
def add_one(lst):
    """Returns copy with 1 added to every element

    Precond: lst is a list of all numbers"""
    copy = []  # accumulator
    for x in lst:
        x = x +1
        copy.append(x)
    return copy
```

Generators: The New Way

```
def add_one(input)
    """Generates 1 added to each element of input

    Precond: input is a iterable of all numbers"""


    for x in input :
        yield x +1
```

Much Simpler!

yield eliminates the accumlator

Accumulators: The Old Way

```
def evens(lst):
    """Returns a copy with even elements only

    Precond: lst is a list of all numbers"""
    copy = []  # accumulator
    for x in lst:
        if x % 2 == 0:
            copy.append(x)
    return copy
```

Generators: The New Way

```
def evens(input):
    """Generates only the even elements of input

    Precond: input is a iterable of all numbers"""


    for x in input:
        if x % 2 == 0:
            yield x
```

Accumulators: The Old Way

```
def average(lst):
    """Returns a running average of lst (elt n is average of lst[0:n])

    Ex: average([1, 3, 5, 7]) returns [1.0, 2.0, 3.0, 4.0]

    Precond: lst is a list of all numbers"""
    result = []                # actual accumulator
    sum = 0; count = 0    # accumulator "helpers"
    for x in lst:
        sum = sum+x; count = count+1
        result.append(sum/count)
    return result
```

Accumulators: The Old Way

```
def average(lst):
    """Returns a running average of lst (elt n is average of lst[0:n])

    Ex: average([1, 3, 5, 7]) returns [1.0, 2.0, 3.0, 4.0]

    Precond: lst is a list of all numbers"""
    result = []              #
    sum = 0; count = 0       #
    for x in lst:
        sum = sum+x; count = count+1
        result.append(sum/count)
    return result
```

Allows multiple assignments per line

Generators: The New Way

```
def average(input):
    """Generates a running average of input

    Ex: input 1, 3, 5, 7 yields 1.0, 2.0, 3.0, 4.0

    Precond: input is a iterable of all numbers"""
    sum = 0      # accumulator “helper”
    count = 0    # accumulator “helper”
    for x in input:
        sum = sum+x
        count = count+1
        yield sum/count
```

Chaining Generators

- Generators can be chained together
  - Take an iterator/iterable as input
  - Produce an iterator as output
  - Output of one generator = input of another
- Powerful programming technique

Simple Chaining


```
>>> a = [1, 2, 3, 4]                           # Start w/ any iterable
>>> b = add_one(average(evens(a)))             # Apply right to left
>>> c = list(b)                                # Convert to list/tuple
>>> c
[3.0, 4.0]
```

Simple Chaining


```
>>> a = [1, 2                                  any iterable
>>> b = add_                                   ht to left
>>> c = list(b)                     # Convert to list/tuple
>>> c
[3.0, 4.0]
```

Natural way to process data **streams**

Why Do We Care?

- Stream programming is an advanced topic
  - Involves chaining together many generators
  - Will see this again if go on to 3110
- But we have an application in **A7**!
  - Remember that GUIs are like iterator classes
  - Game app runs with an "invisible" loop
  - All **loop variables** implemented as **attributes**
  - Generators are a way to **simplify** all this

Why Do We Care?

- Stream programming is an advanced topic
  - Involves chaining together many generators
  - Will see this again if go on to 3110
- But we have an application in **A7**!
  - Remember that GUIs are like iterator classes
  - Game
  - All **loc** **utes**
  - Generators are a way to **simplify** all this

Unfortunately out of scope