

Lecture 24

Generators

Announcements for This Lecture

Assignments

- A6 due **TONIGHT**
- A7 due **December 7th**
 - Can do all but LAST step
 - Get ship moving by Tues.
 - Will cover last step Tues.

Optional Videos

- **Videos 28.1-28.7** for today
- **Lesson 29.1-29.6** next time

Next Week

- There is no lab next week
 - But Tuesday hours are open
 - Open to **EVERYONE**
 - Go for help on lab or A7
- But lecture is **important**
 - Return to topic of invariants
 - Setting us up for sorting
- Try to finish lab 24 first
 - Frees remaining time for A7

Recall: The Range Iterable

range(x)

Example

- 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

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

Recall: The Range Iterable

range(x)

Example

- Creates an *iterable*

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

- Can be used in a for-loop
- Makes it possible to iterate over a range of numbers

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

- But it is not a **black-box**

- A **black-box** is a folder
- Entirely used in for-loop
- Contents of folder hidden

```
1
```

```
2
```

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

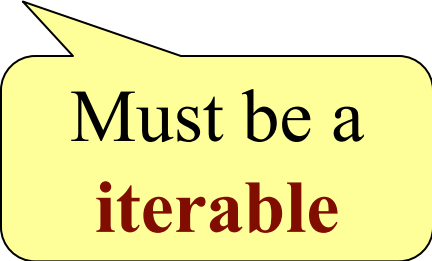
```
>>> a = iter([1,5,3])
```

```
>>> 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!

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 square  
    ...  
    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 iterable

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

```
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 object
```

```
>>> next(a)
```

```
0
```

```
>>> next(a)
```

```
1
```

```
>>> next(a)
```

```
4
```

Essentially
a constructor

What Happens on a Function Call?

Visualize Execute Code Edit Code Heap primitives Use an

```
1 def range2iter(n):
2     """Generator for a range of squares"""
3     for x in range(n):
4         yield x*x
5         print('Ended loop for '+str(x))
6
7 → a = range2iter(3)
8
9 → x = next(a)
10 y = next(a)
11 z = next(a)
12 w = next(a)
```

Global

global	
range2iter	id1
a	id2

Frames

id2:generator
range2iter(3)

Creates a generator

No call frame

<< First < Back Step 3 of 20 Forward > Last >>

→ line that has just executed
→ next line to execute

next() Initiates a Function Call

Visualize

Execute Code

Edit Code

Heap primitives Use arrows

```
1 def range2iter(n):
2     """Generator for a range of squares"""
3     for x in range(n):
4         yield x*x
5         print('Ended loop for '+str(x))
6
7 a = range2iter(3)
8
9 x = next(a)
10 y = next(a)
11 z = next(a)
12 w = next(a)
```



<< First

< Back

Step 4 of 20

Forward >

Last >>

→ line that has just executed

→ next line to execute

Globals

Objects

global

range2iter

id1

a

id2

id1:function

Comes from original call

Frames

range2iter

n

3

Frame for next()

Call Finishes at the yield

Visualize

Execute Code

Edit Code

Heap primitives Use arrows

```
1 def range2iter(n):
2     """Generator for a range of squares"""
3     for x in range(n):
4         yield x*x
5         print('Ended loop for '+str(x))
6
7 a = range2iter(3)
8
9 x = next(a)
10 y = next(a)
11 z = next(a)
12 w = next(a)
```



<< First

< Back

Step 6 of 20

Forward >

Last >>

→ line that has just executed

→ next line to execute

Globals

```
global
range2iter | id1
a          | id2
```

Objects

```
id1:function
range2iter(n)

id2:generator
range2iter(3)
```

Frames

```
range2iter
n | 3
x | 0
Return value | 0
```

**yield is return
for next()**

Later Calls Resume After the yield

Visualize

Execute Code

Edit Code

Heap primitives Use arrows

```
1 def range2iter(n):
2     """Generator for a range of squares"""
3     for x in range(n):
4         yield x*x
5         print('Ended loop for '+str(x))
6
7 a = range2iter(3)
8
9 x = next(a)
10 y = next(a)
11 z = next(a)
12 w = next(a)
```



<< First

< Back

Step 8 of 20

Forward >

Last >>

→ line that has just executed

→ next line to execute

Globals

global	
range2iter	id1
a	id2
x	0

Objects

id1: function
range2iter(n)
id2: generator
range2iter(3)

Frames

range2iter	
n	3
x	0

From last time

Next call returns to where it left off

Exception is Made Automatically

Visualize

Execute Code

Edit Code

Heap primitives Use arrows

```
1 def range2iter(n):
2     """Generator for a range of squares"""
3     for x in range(n):
4         yield x*x
5         print('Ended loop for '+str(x))
6
7 a = range2iter(3)
8
9 x = next(a)
10 y = next(a)
11 z = next(a)
12 w = next(a)
```

Globals

Objects

global	
range2iter	id1
x	0
y	1
z	4

id1:function
range2iter(n)

Frames

<< First

< Back

Program terminated

Forward >

Last >>

StopIteration:

Exception when
generator is done

Return Statements Make Exceptions

Visualize

Execute Code

Edit Code

Heap primitives Use arrows

```
1 def range2iter(n):
2     """Generator for a range of squares"""
3     for x in range(n):
4         yield x*x
5         print('Ended loop for '+str(x))
6     return x # The final x
7
8 a = range2iter(3)
9
10 x = next(a)
11 y = next(a)
12 z = next(a)
13 w = next(a)
```

Globals

Objects

global	
range2iter	id1
x	0
y	1
z	4

id1: function
range2iter(n)

Frames

<< First

< Back

Program terminated

Forward >

Last >>

StopIteration: 2

Return Value

Exception when
generator is done

Activity: Call Frame Time

Function Definitions

```
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:

harmonic	n	2	34
sum	0	g	id3

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

Which One is Closest to Your Answer?

A:

harmonic	n	2	34
sum	0	g	id3
rnginv	n	2	19

B:

harmonic	n	2	34
sum	0	g	id3
rnginv	n	2	20
x	1		

C:

harmonic	n	2	34		
sum	0	g	id3	x	1

D:

harmonic	n	2	34
sum	0	g	id3
rnginv	n	2	20
x	1	YIELD	1

Which One is Closest to Your Answer?

<p>A:</p> <table border="1"><tr><td>harmonic</td><td>n</td><td>2</td><td>34</td></tr><tr><td>sum</td><td>0</td><td>g</td><td>id3</td></tr></table>	harmonic	n	2	34	sum	0	g	id3	<p>B:</p> <table border="1"><tr><td>harmonic</td><td>n</td><td>2</td><td>34</td></tr><tr><td>sum</td><td>0</td><td>g</td><td>id3</td></tr></table>	harmonic	n	2	34	sum	0	g	id3
harmonic	n	2	34														
sum	0	g	id3														
harmonic	n	2	34														
sum	0	g	id3														
<p>rnginv</p>	<p>E:</p> <p>~_ (ツ) _ /</p>	<p>n</p>	2	20													
<p>C:</p> <table border="1"><tr><td>harmonic</td></tr><tr><td>sum</td><td>0</td><td>g</td></tr></table>		harmonic	sum	0	g	<p>n</p>	2	34	id3								
harmonic																	
sum	0	g															
	<table border="1"><tr><td>rnginv</td><td>n</td><td>2</td><td>20</td></tr><tr><td>x</td><td>1</td><td>YIELD</td><td>1</td></tr></table>	rnginv	n	2	20	x	1	YIELD	1								
rnginv	n	2	20														
x	1	YIELD	1														

Activity: Call Frame Time

Function Definitions

```
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)
```

A:

harmonic	n	2	34
sum	0	g	id3
rnginv	n	2	19

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
rnginv		n	2
			20
x	1		

C:

harmonic	n	2	34
sum	0	g	id3
rnginv		n	2
			20
x	1	YIELD	1

D:

harmonic	n	2	34
sum	0	g	id3
rnginv		n	2
			21
x	1	YIELD	1

Activity: Call Frame Time

Function Definitions

```
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

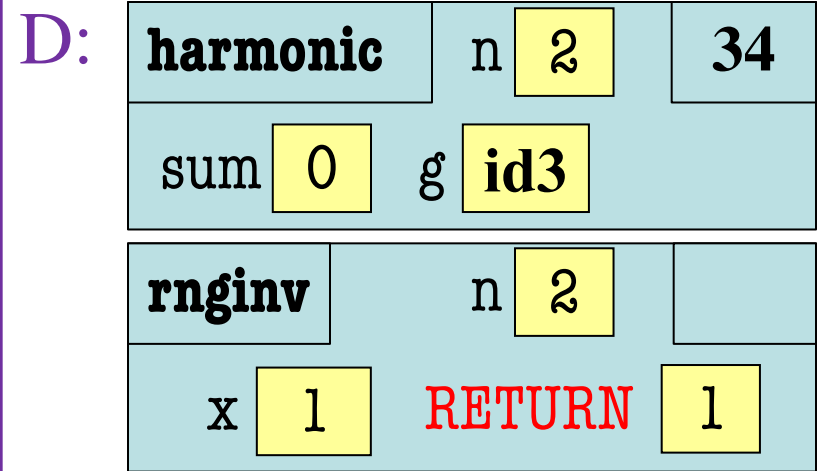
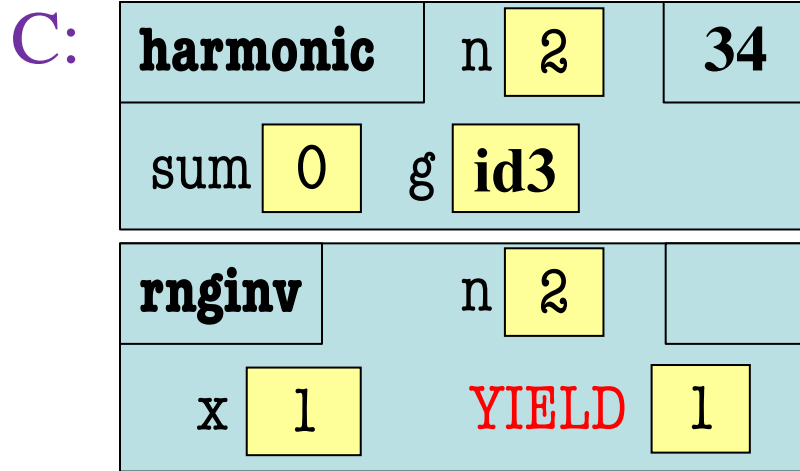
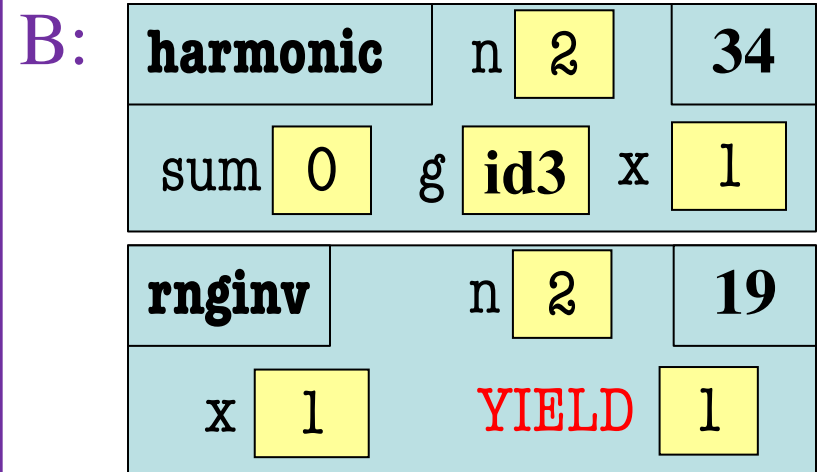
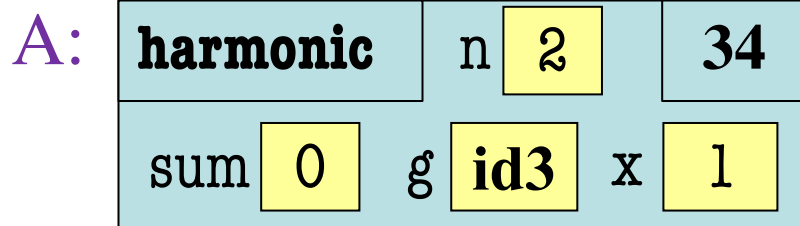
```
>>> x = harmonic(2)
```

B:

harmonic	n	2	34
sum	0	g	id3
rnginv	n	2	20
x	1		

What is the **next step**?

Which One is Closest to Your Answer?



Activity: Call Frame Time

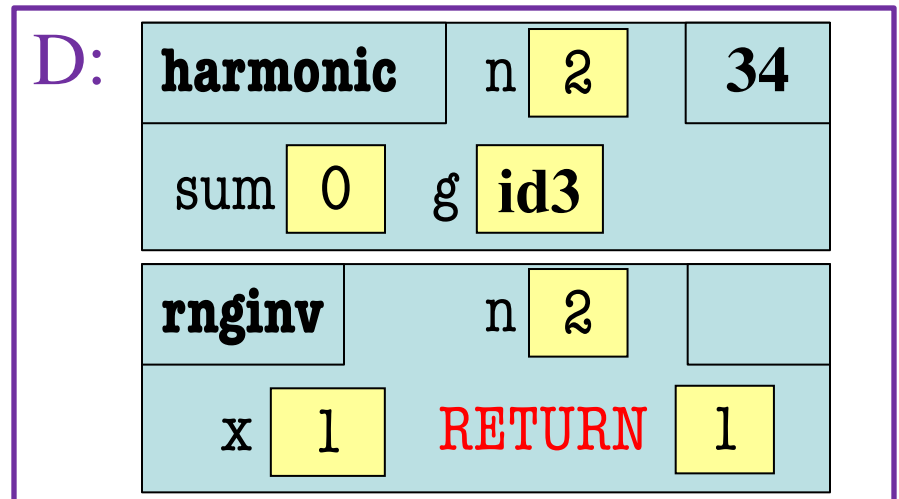
Function Definitions

```
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)
```



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 accumulator**

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 lst:
```

```
        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, 3, 4, 5]
>>> b = add_one(x) for x in a
>>> c = list(b) # Convert to list/tuple
>>> c
[3.0, 4.0]
```

Natural way to process any iterable data streams (right to left)

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 for **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 for **A7!**
 - Remember that GUIs are like iterator classes
 - Game
 - All local **Topic of next lecture** **utes**
 - Generators are a way to **simplify** an **ans**