Lecture 27

Generators
Announcements for This Lecture

Assignments

- A6 is now graded
  - Mean: 88.2 Median: 92
  - Std Dev: 13.4
  - Mean: 17.5 hr Median: 15 hr
  - Std Dev: 9 hr
- A7 due **December 7th**
  - Should be moving asteroids
  - Extensions via lab instructor
  - Can work in Lab Thu/Fri

Finishing Up

- Submit a course evaluation
  - Will get an e-mail for this
  - Part of the “participation grade” (e.g. polling grade)
- Final, Dec 13\(^{th}\) 2-4:30 pm
  - Study guide is posted
- Conflict with Final Exam?
  - e.g. > 2 finals in 24 hours
  - Submit conflicts to CMS

11/29/22
Recall: The Range Iterable

<table>
<thead>
<tr>
<th>range(x)</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Creates an <em>iterable</em></td>
<td>&gt;&gt;&gt; range(3)</td>
</tr>
<tr>
<td>▪ Can be used in a for-loop</td>
<td>range(0,3)</td>
</tr>
<tr>
<td>▪ Makes ints (0, 1, ... x-1)</td>
<td>&gt;&gt;&gt; for x in range(3)</td>
</tr>
<tr>
<td>• But it is not a tuple!</td>
<td>... print(x)</td>
</tr>
<tr>
<td>▪ A <em>black-box</em> for numbers</td>
<td>0</td>
</tr>
<tr>
<td>▪ Entirely used in for-loop</td>
<td>1</td>
</tr>
<tr>
<td>▪ Contents of folder hidden</td>
<td>2</td>
</tr>
</tbody>
</table>
Recall: The Range Iterable

<table>
<thead>
<tr>
<th>range(x)</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Creates an <em>iterable</em></td>
<td><code>&gt;&gt;&gt; range(3)</code></td>
</tr>
<tr>
<td>- Can be used in a for-loop</td>
<td></td>
</tr>
<tr>
<td>- Makes int sequence <code>(0, 1, ..., x - 1)</code></td>
<td></td>
</tr>
<tr>
<td>• But it is not a tuple!</td>
<td></td>
</tr>
<tr>
<td>- A black-box for numbers</td>
<td></td>
</tr>
<tr>
<td>- Entirely used in for-loop</td>
<td></td>
</tr>
<tr>
<td>- Contents of folder hidden</td>
<td>1</td>
</tr>
</tbody>
</table>

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

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

Must be a *iterable*
Iterators Can Be Used in For-Loops

```python
>>> 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 spot
    ...
    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
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

```python
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)
```

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

```
1 def range2iter(n):
2     """Generator for a range of squares"""
3     for x in range(n):
4         yield 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)
```

Creates a generator

No call frame
next() Initiates 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)
```

Frame for next()

Comes from original call
Call Finishes at the \texttt{yield}

```python
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)
```

\texttt{yield} is \texttt{return} for \texttt{next()}
Later Calls Resume After the \textit{yield}

```
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)
```

From last time

Next call returns to where it left off
Exception is Made Automatically

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)

Exception when generator is done
Return Statements Make Exceptions

```
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)

Exception when generator is done
StopIteration: 2
Return Value
```
Activity: Call Frame Time

Function Definitions

```python
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

```python```

>>> 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
- sum
- rnginv

B: 
- harmonic
- sum
- rnginv

C: 
- harmonic
- sum

D: 
- harmonic
- sum

11/29/22
Generators
Which One is Closest to Your Answer?

A:  
<table>
<thead>
<tr>
<th>harmonic</th>
<th>n</th>
<th>2</th>
<th>34</th>
</tr>
</thead>
<tbody>
<tr>
<td>sum</td>
<td>0</td>
<td>g</td>
<td>id3</td>
</tr>
<tr>
<td>rnginv</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

B:  
<table>
<thead>
<tr>
<th>harmonic</th>
<th>n</th>
<th>2</th>
<th>34</th>
</tr>
</thead>
<tbody>
<tr>
<td>sum</td>
<td>0</td>
<td>g</td>
<td>id3</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

C:  
<table>
<thead>
<tr>
<th>harmonic</th>
<th>n</th>
<th>2</th>
<th>34</th>
</tr>
</thead>
<tbody>
<tr>
<td>sum</td>
<td>0</td>
<td>g</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

E:  
\( \sqrt{\frac{\text{ツ}}{\text{ツ}}} \)

D:  
<table>
<thead>
<tr>
<th>harmonic</th>
<th>n</th>
<th>2</th>
<th>34</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>rnginv</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

11/29/22  Generators  25
Activity: Call Frame Time

Function Definitions

```python
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

```python
>>> 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  20

C:

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

D:

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

11/29/22  Generators
Activity: Call Frame Time

Function Definitions

```python
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

```python
>>> 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 Definitions

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

>>> x = harmonic(2)

D:

34  x  1  RETURN  1
33  g  id3  
32  sum  0  
20    yield 1/x
19  for x in range(1,n):
def harmonic(n):  #Harmonic sum
36    return x
35        sum = sum+x
34         for x in g:
33             g = rnginv(n)
32               sum = 0
19 def rnginv(n):  #Inverse range
20    for x in range(1,n):
19       yield 1/x

11/29/22
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
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
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
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
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
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

Allows multiple assignments per line
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

```python
>>> 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]
# Start with any iterable
>>> b = add_one(average(evens(a)))
# Apply right 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 loop is an iterator as well
  ▪ All loop variables are attributes
  ▪ Generators are a way to simplify all this

Unfortunately out of scope this year