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

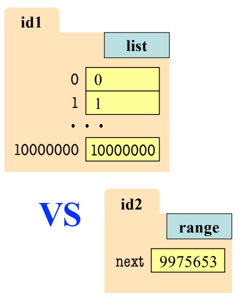
```
>>> a = iter([1,5,3])
>>> next(a)
1
>>> next(a)
5
```

Must be a **iterable**

1

Motivation for Iterables

- Large lists are a problem
  - Use a lot of space in heap
  - **Ex:** list(range(10000000))
- 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



2

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

3

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

4

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?

5

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

6

The Generator approach

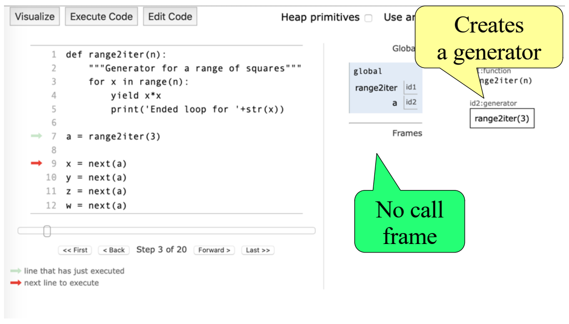
```
def range2iter(n):
    """
    Generator for the squares
    of numbers 0 to n-1

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

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

Essentially a constructor

7

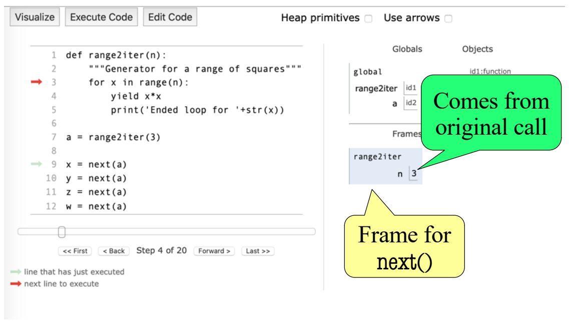
What Happens on a Function Call?



```
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)
```

8

next() Initiates a Function Call



```
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)
```

9

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

10

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
```

11

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

12