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
```

Iterators are Classes

```python
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
```

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)
```

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 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

```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
```

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
```

```python
>>> a = range2iter(3)
<generator object>
>>> next(a)
0
>>> next(a)
1
>>> next(a)
4
```
What Happens on a Function Call?

1. No call frame

next() Initiates a Function Call

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
```

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

input ➔ evens ➔ average ➔ add_one ➔ output