Module 28

Generators

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!

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

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 **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
  - Pr
  - Fu

More on this distinction in a bit

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
```

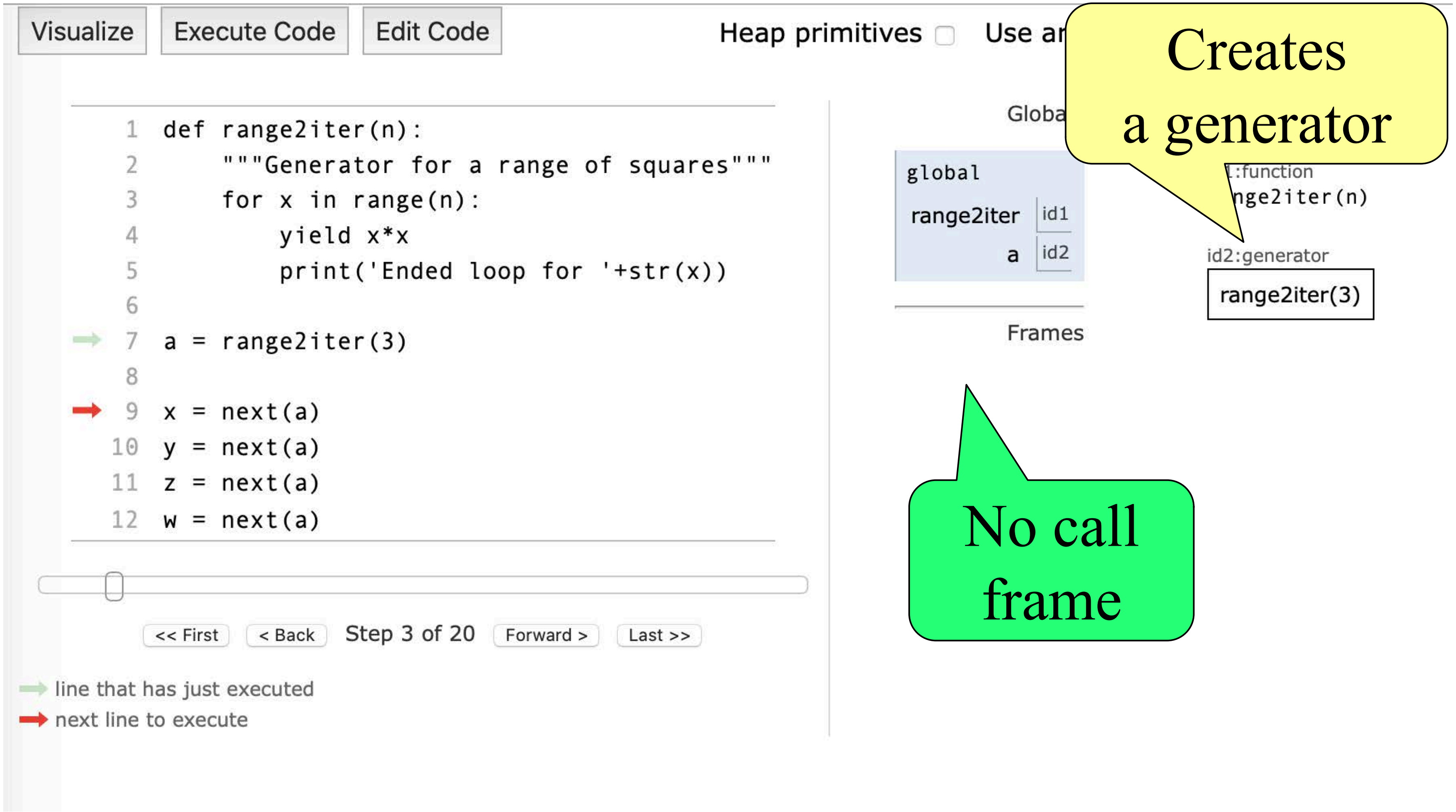
The Generator approach

```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?



```
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)
```

Step 3 of 20

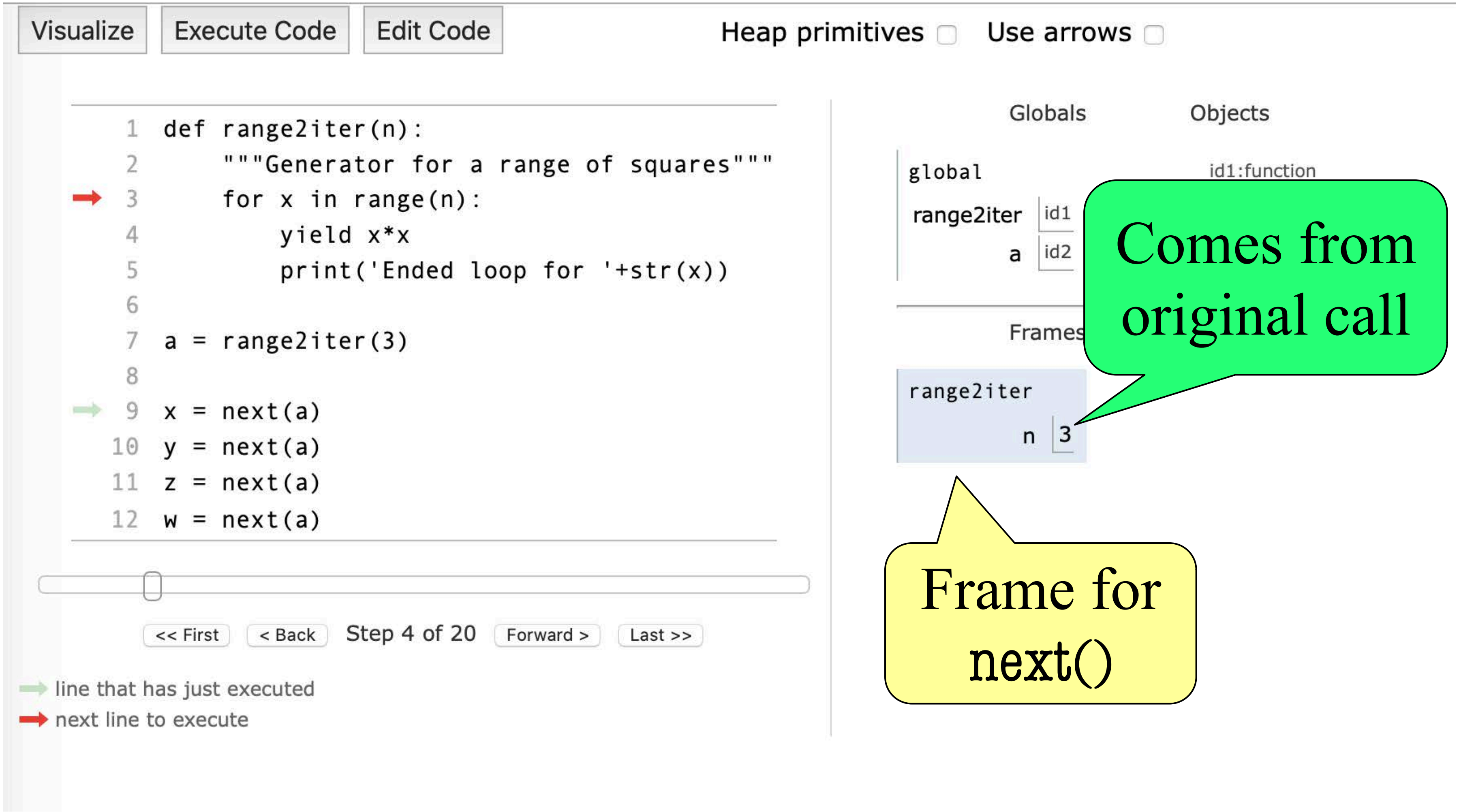
line that has just executed

next line to execute

Creates a generator

No call frame

next() Initiates a Function Call

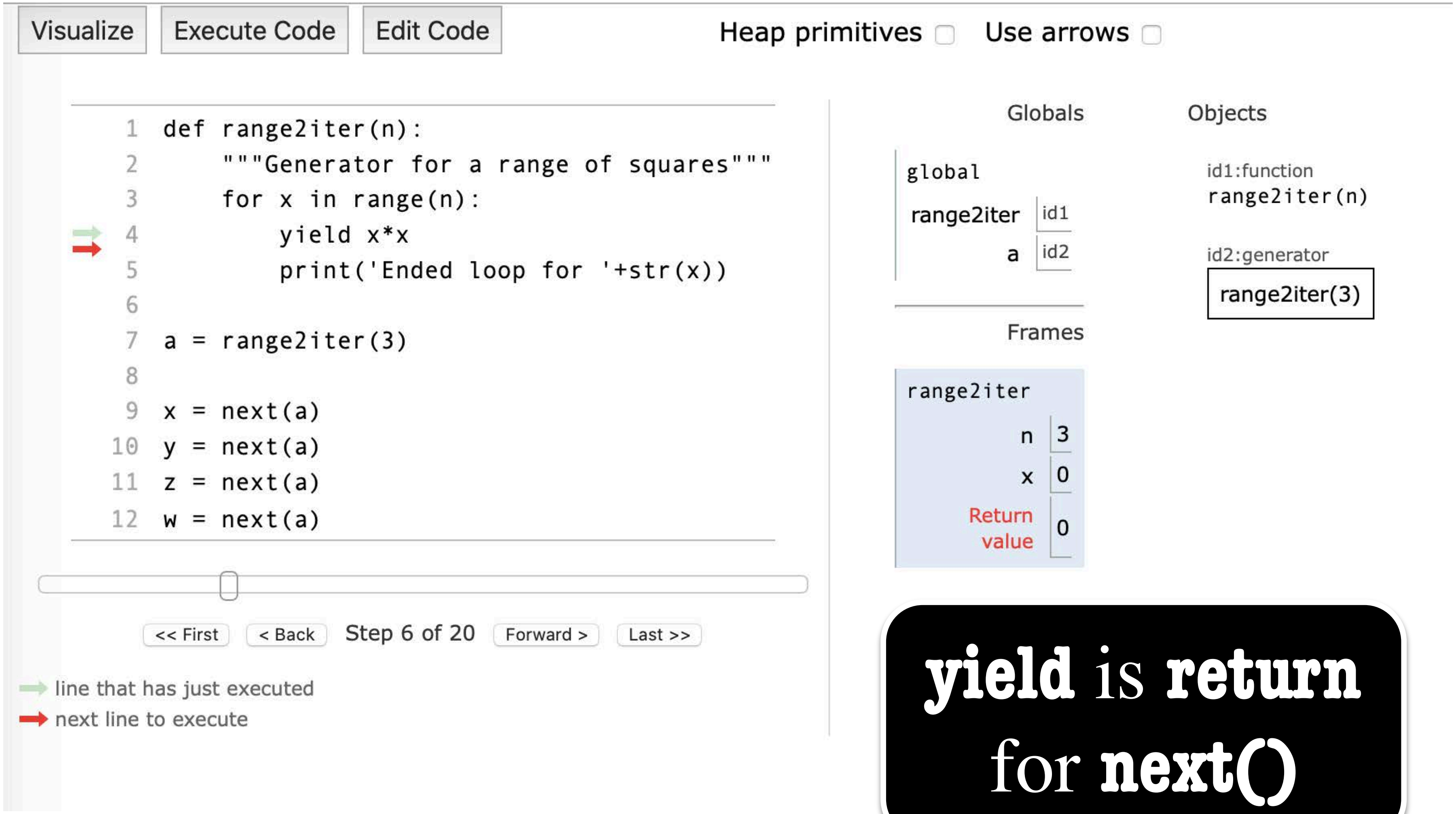


```python
def range2iter(n):
    """Generator for a range of squares"""
    for x in range(n):
        yield x*x
        print('Ended loop for '+str(x))

a = range2iter(3)

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

Comes from original call

Frame for next()

Call Finishes at the **yield**



```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)
```

Globals

global
range2iter: id1
a: id2

Frames

range2iter
n: 3
x: 0
Return value: 0

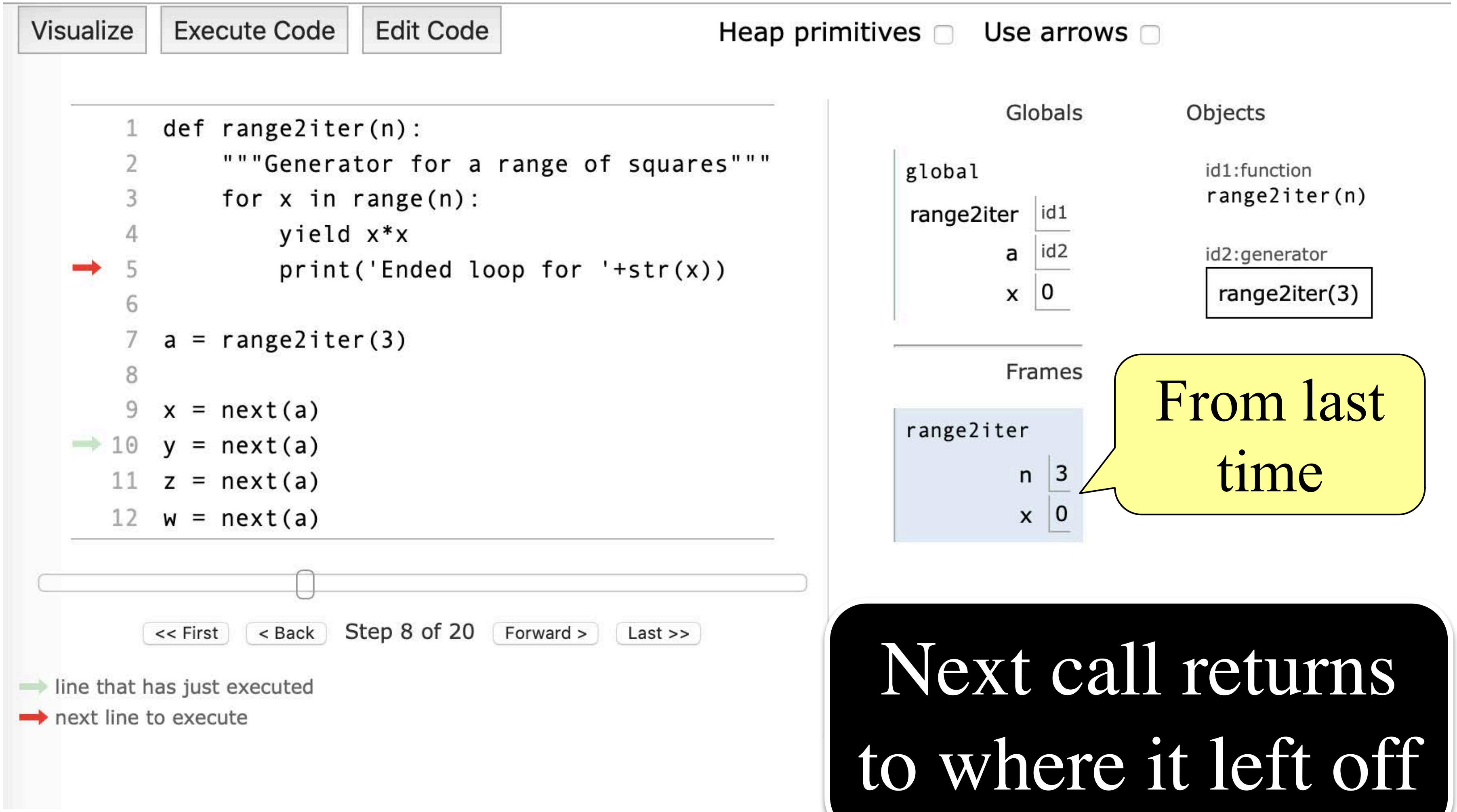
Objects

id1:function
range2iter(n)

id2:generator
range2iter(3)

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

Later Calls Resume After the **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)
```

Globals

global
range2iter id1
a id2
x 0

Objects

id1:function
range2iter(n)

id2:generator
range2iter(3)

Frames

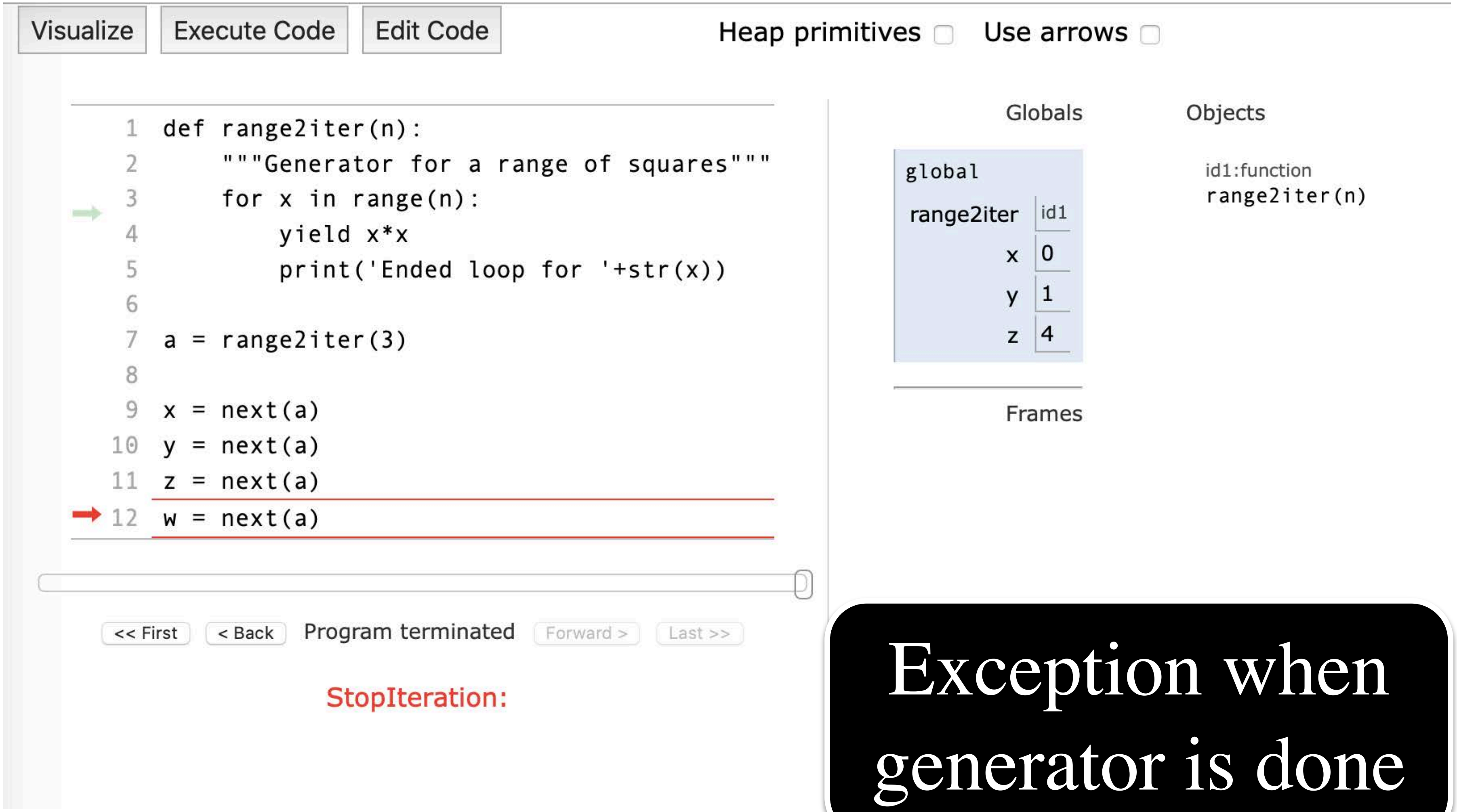
range2iter
n 3
x 0

→ line that has just executed
→ next line to execute

From last time

Next call returns to where it left off

Exception is Made Automatically



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)
```

Program terminated

StopIteration:

Globals

global	
range2iter	id1
x	0
y	1
z	4

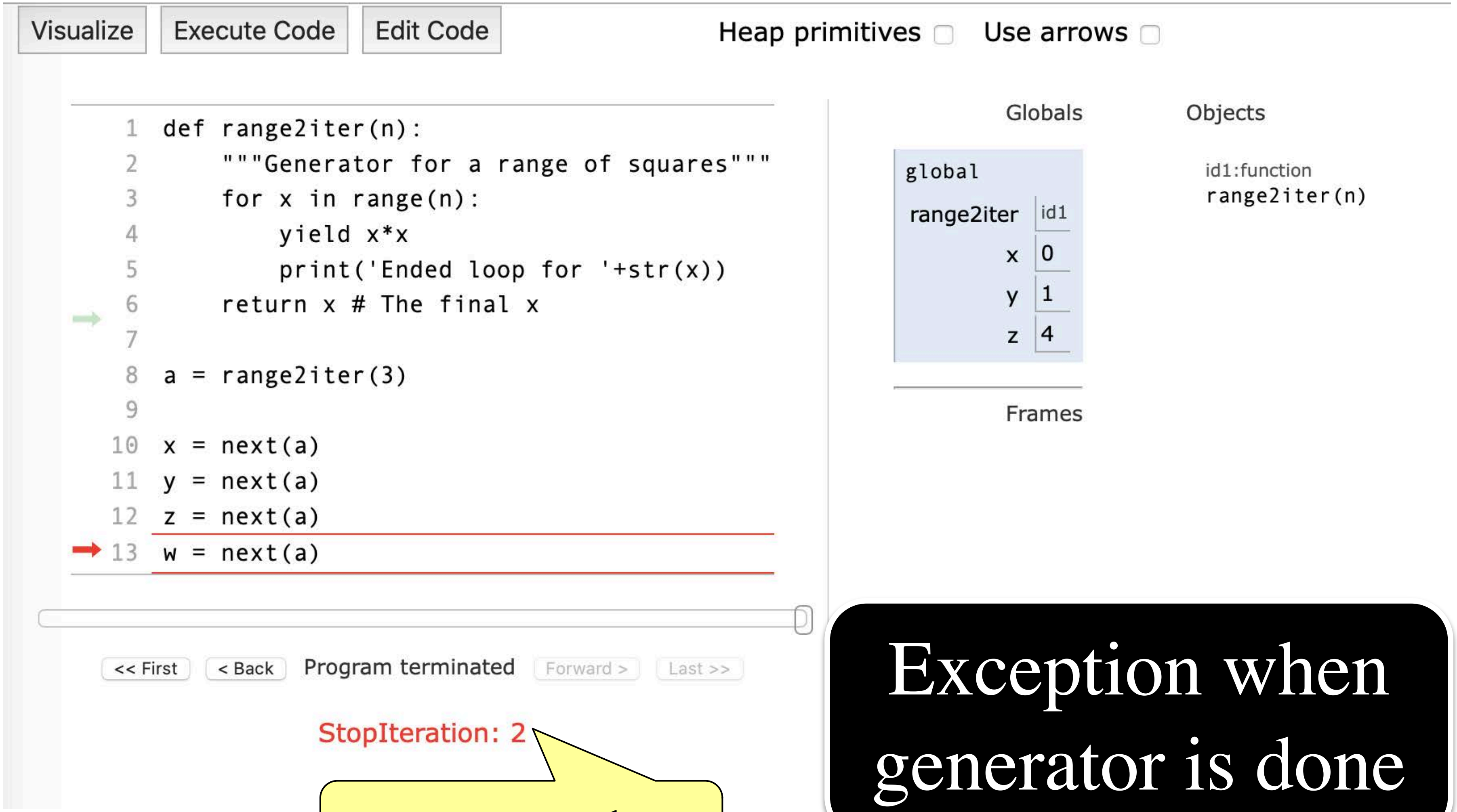
Frames

Objects

id1:function
range2iter(n)

Exception when generator is done

Return Statements Make Exceptions



```python
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)
```

global

range2iter	id1
x	0
y	1
z	4

id1:function
range2iter(n)

```
StopIteration: 2
```

Return Value

Exception when generator is done

Iterator Parameters

- The initial call is essentially a constructor
  - Creates a generator object
  - Parameters used to initialize the object
- **Pattern:** Use an iterable parameter
  - Iterator loops over this iterable
  - Iterator transforms contents of the iterable
  - Iterator yields the transformed data
- Generators often replace **accumulator pattern**

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 lst:
        sum = sum+x
        count = count+1
        yield sum/count
```

Advanced Data Processing

- Previous lesson saw functions as variables
  - Seemed like a weird but useless trick
- It is very useful in **large data processing**
  - Start with a function on a single piece of data
  - Have a large set (list/tuple) of this data
  - Want to apply function to every data in set
- We can process this data with a *for-loop*
  - But write a new for-loop for each function?

Example: map()

```
def map(f,data)
    """Returns a copy of data, f applied to each entry

    Precond: f is a function taking exactly one argument
    Precond: data iterable, each elt satisfying precond of f"""
    accum = []
    for item in data:
        accum.append( f(item) )
    return accum
```

Apply function f to each item

Example: map()

```
def plus1(x)
    """Returns x+1"""
    return x+1


def negate(x):
    """Returns –x"""
    return -x
```

```
>>> a = [1,2,3]
>>> b = map(plus1, a)
>>> b
[2,3,4]
>>> c = map(negate, a)
>>> c
[-1,-2,-3]
```

The Generator Version

```
def map(f,data)
    """Generates f applied to each element

    Precond: f is a function taking exactly one argument
    Precond: data iterable, each elt satisfying precond of f"""


    for item in data:
        yield f(item)
```

Apply function f to each item

Example: filter()

```
def filter(f,data)
    """Returns a copy of data, removing anything f is False on

    Precond: f is a boolean function taking exactly one argument
    Precond: data iterable, each elt satisfying precond of f"""
    accum = []
    for item in data:
        if f(item):
            accum.append( item)
    return accum
```

Only add if f(item) is True

Example: filter()

```
def iseven(x)
    """Rets True if x even"""
    return x % 2 == 0


def ispos(x):
    """Rets True if x > 0"""
    return x > 0
```

```
>>> a = [-2,1,4]
>>> b = filter(iseven, a)
>>> b
[-2,4]
>>> c = filter(ispos, a)
>>> c
[1,4]
```

The Generator Version

```
def filter(f,data)
    """”Generates all elements of data where f is True

    Precond: f is a boolean function taking exactly one argument
    Precond: data iterable, each elt satisfying precond of f"""

    for item in data:
        if f(item):
            yield accum
```

Only add if f(item) is True

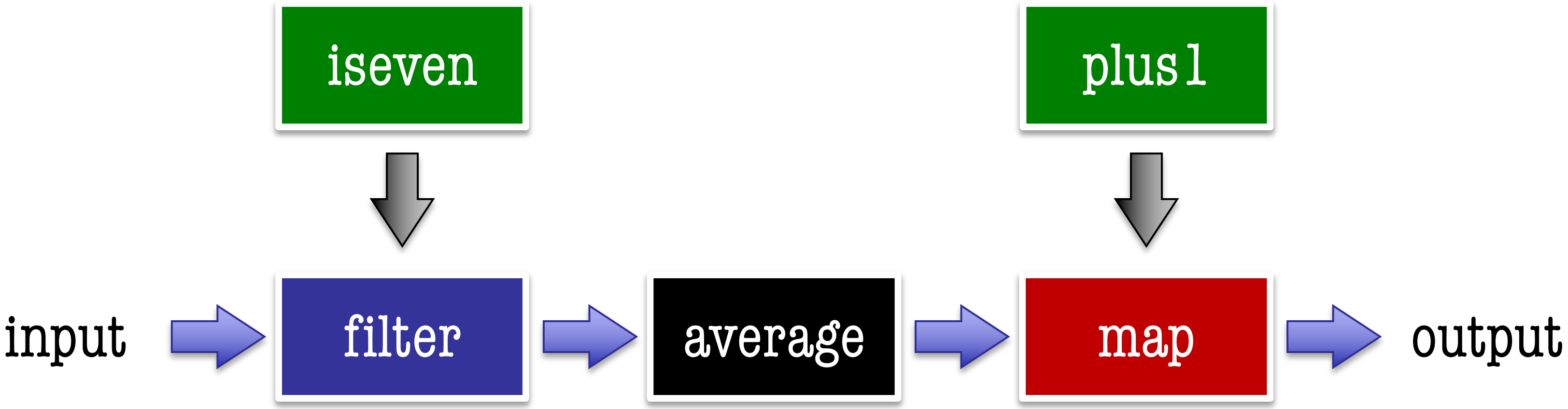
These Are Famously Powerful

- Functions `map` and `filter` are very powerful tools
  - Focus of study in advanced language courses
  - Form the basis of data processing infrastructure
- They are building blocks to combine together
  - The generators take iterables/iterators as input
  - And the output is a iterator itself
  - So you can chain these generators together
- **Benefit:** Python needs *much* less memory
  - Only looks at one element at a time

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]
```

Chaining with Map and Filter



iseven

plus1

input

filter

average

map

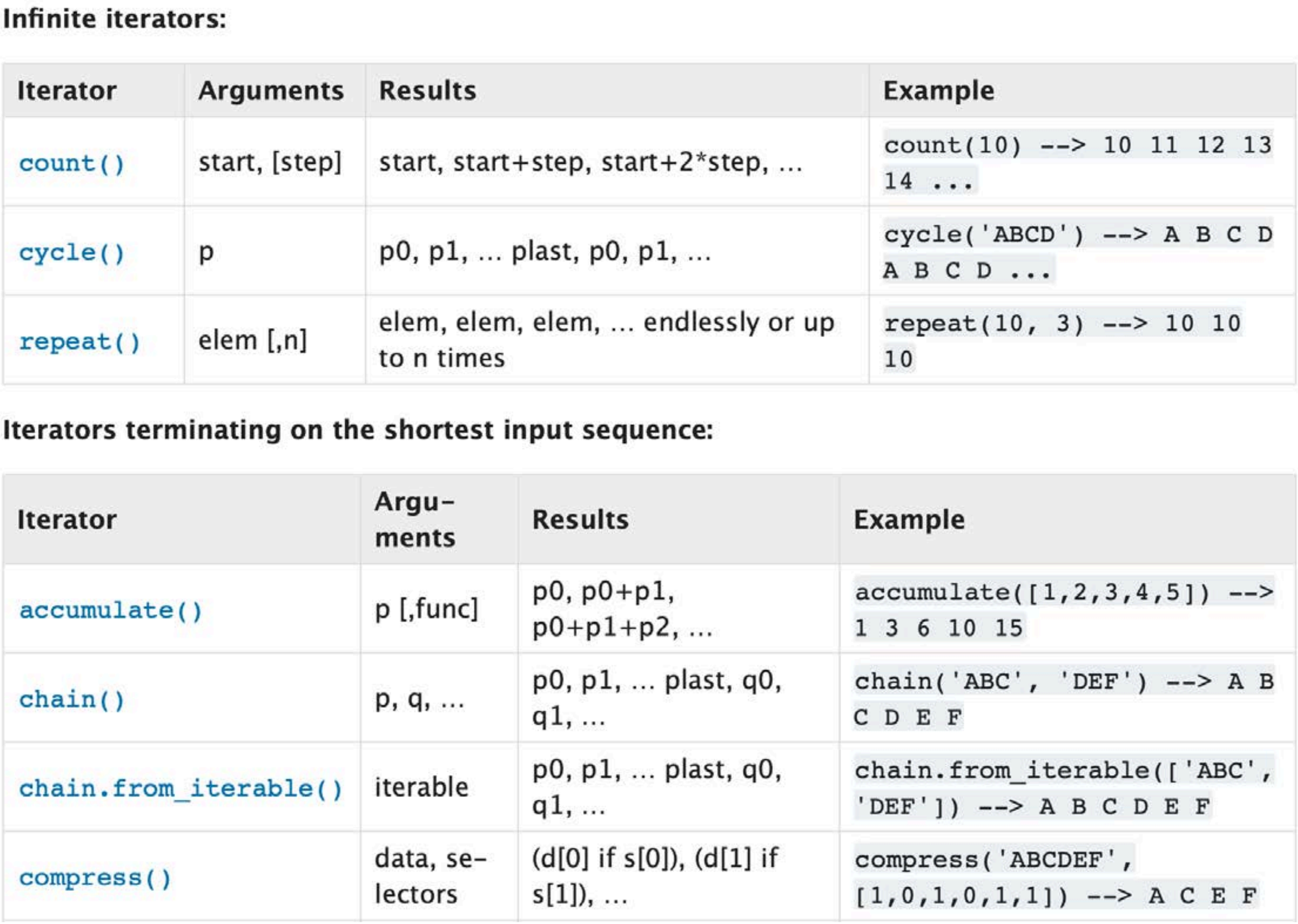
output

```
>>> a = [1, 2, 3, 4]                # Start w/ any iterable
>>> b = average(filter(iseven,a))   # Apply first funcs
>>> b = map(plus1,b)                # Add map to chain
>>> c = list(b)                     # Convert to list/tuple
```

Python Encourages This Approach

- This is a natural way to process data
  - Don't write complex programs
  - Just download functions and string together
  - Will see this again if go on to 3110
- Python has a lot of these tools for you
  - Generators map and filter are **built-in!**
  - Other tools in the `itertools` module
- Worth exploring on your own

Module itertools



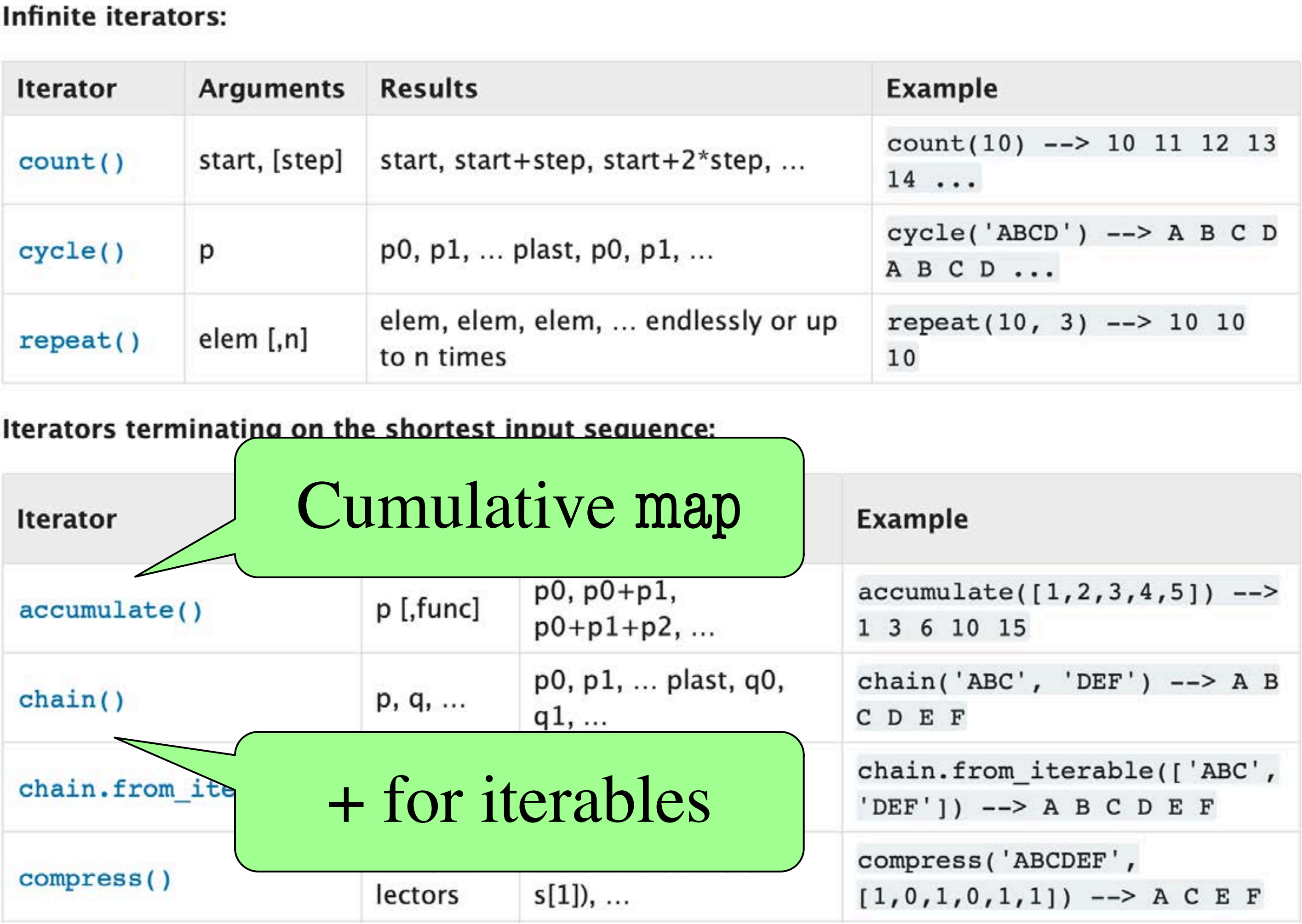
Infinite iterators:

Iterator	Arguments	Results	Example
`count()`	start, [step]	start, start+step, start+2*step, …	`count(10) --> 10 11 12 13 14 ...`
`cycle()`	p	p0, p1, … plast, p0, p1, …	`cycle('ABCD') --> A B C D A B C D ...`
`repeat()`	elem [,n]	elem, elem, elem, … endlessly or up to n times	`repeat(10, 3) --> 10 10 10`

Iterators terminating on the shortest input sequence:

Iterator	Arguments	Results	Example
`accumulate()`	p [,func]	p0, p0+p1, p0+p1+p2, …	`accumulate([1,2,3,4,5]) --> 1 3 6 10 15`
`chain()`	p, q, …	p0, p1, … plast, q0, q1, …	`chain('ABC', 'DEF') --> A B C D E F`
`chain.from_iterable()`	iterable	p0, p1, … plast, q0, q1, …	`chain.from_iterable(['ABC', 'DEF']) --> A B C D E F`
`compress()`	data, selectors	(d[0] if s[0]), (d[1] if s[1]), …	`compress('ABCDEF', [1,0,1,0,1,1]) --> A C E F`

Module itertools



Infinite iterators:

Iterator	Arguments	Results	Example
`count()`	start, [step]	start, start+step, start+2*step, …	`count(10) --> 10 11 12 13 14 ...`
`cycle()`	p	p0, p1, … plast, p0, p1, …	`cycle('ABCD') --> A B C D A B C D ...`
`repeat()`	elem [,n]	elem, elem, elem, … endlessly or up to n times	`repeat(10, 3) --> 10 10 10`

Iterators terminating on the shortest input sequence:

Iterator			Example
`accumulate()`	p [,func]	p0, p0+p1, p0+p1+p2, …	`accumulate([1,2,3,4,5]) --> 1 3 6 10 15`
`chain()`	p, q, …	p0, p1, … plast, q0, q1, …	`chain('ABC', 'DEF') --> A B C D E F`
`chain.from_ite`			`chain.from_iterable(['ABC', 'DEF']) --> A B C D E F`
`compress()`	lectors	s[1]), …	`compress('ABCDEF', [1,0,1,0,1,1]) --> A C E F`

Cumulative map

+ for iterables

Final Step of Chaining

- The last step of a chain is to convert “back”
    - Data less useful as a generator
    - Would like a list/tuple; easier to manipulate
    - Called **materializing** the computation
- Are there alternatives to list/tuple function?
    - What if we could add code at materialization?
    - We can, but only for lists (not tuples)
    - Called **list comprehension**

List Comprehension

- Basic Format:

  `[ <expression> for <var> in <iterable> ]`

  - Looks like a backwards for-loop
  - That because this is an expression
- Similar to conditional expressions:

  `<expression> if <boolean-exp> else <expression>`
- Example: `[x for x in iterable]`
  - This is the same as `list(iterable)`

Only Works for Lists

Contents of parens is a **generator expression**

```
>>> ( x for x in lst )  # Not a tuple
<generator object <genexpr>>
```

Traditional For-Loops

```
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
```

List Comprehension

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

    Precond: lst is a list of all numbers"""
    return [x+1 for x in lst]
```

For-Loops with Conditionals

```
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
```

List Comprehension

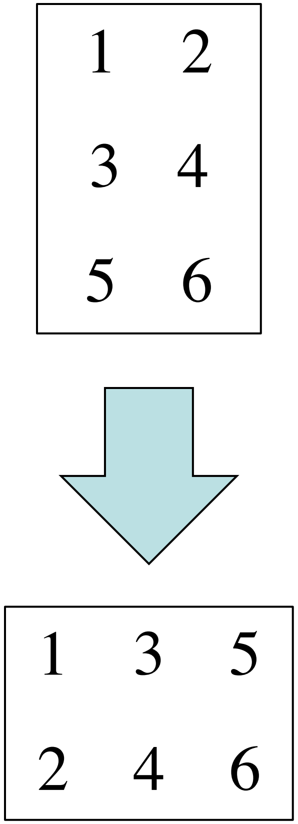
```
def evens(lst):
    """Returns a copy with even elements only

    Precond: lst is a list of all numbers"""
    return [ x for x in lst if x % 2 == 0]


    # THIS IS VERY DIFFERENT
    # return [ (x if x % 2 == 0 else None) for x in lst]
```

Comprehension Filter

Conditional Expression

Nested For-Loops

```
def transpose(table):
    """Returns: copy of table with rows and columns swapped

    Precondition: table is a (non-ragged) 2d List"""
    numcols  = len(table[0])   # All rows have same no. cols
    result = []                # Result (new table) accumulator
    for m in range(numcols):
        newrow = []                # Single row accumulator
        for row in table:
            newrow.append(row[m])  # Create a new row list
        result.append(newrow)      # Add result to table
    return result
```

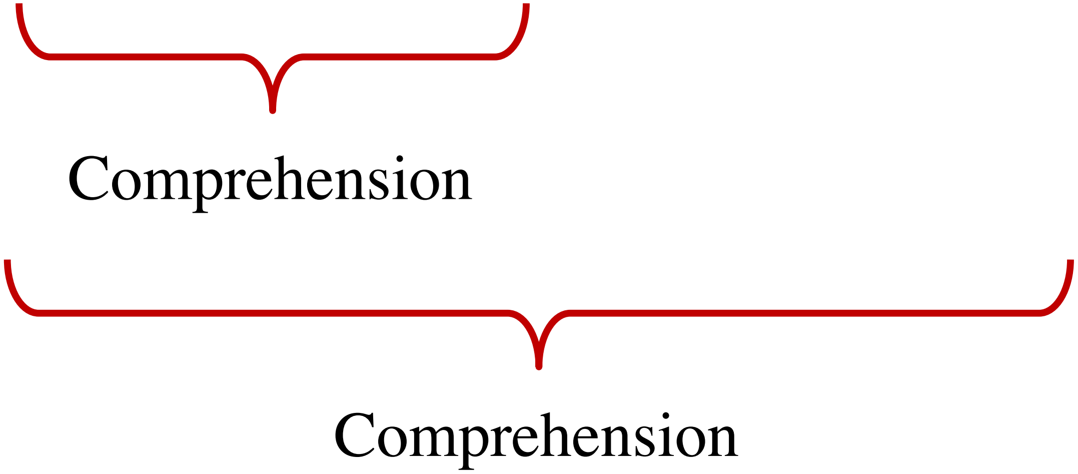


1	2
3	4
5	6

1	3	5
2	4	6

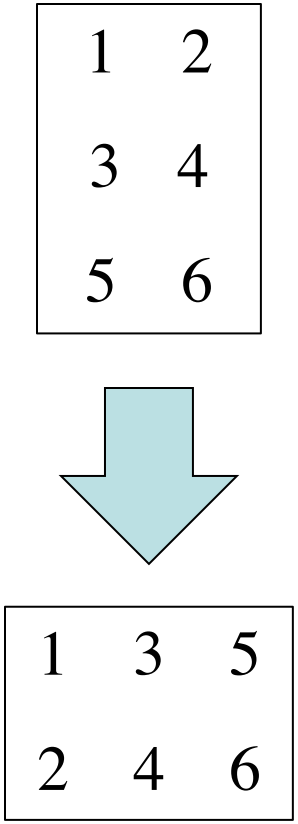
Nested For-Loops

```
def transpose(table):
    """Returns: copy of table with rows and columns swapped
    
    Precondition: table is a (non-ragged) 2d List"""
    numcols  = len(table[0])   # All rows have same no. cols
    return [[row[i] for row in table] for i in range(numcols)]
```



Comprehension

Comprehension



1	2
3	4
5	6

1	3	5
2	4	6

Recall: Dictionaries are Iterable

- Start with a dictionary `d = {'a':1, 'b':2}`
- **Key Iterator:** `d.keys()`

```
>>> list(d.keys())
['a','b']
```

- **Value Iterator:** `d.values()`

```
>>> list(d.values())
[1,2]
```

- **Pair Iterator:** `d.items()`

```
>>> list(d.items())
[('a',1),('b',2)]
```

Dictionary Comprehension

- Basic Format:

  `{ <exp1>:<exp2> for <var> in <iterable> }`

  - `<exp1>` is the key
  - `<exp2>` is the value
  - Pairs together form the dictionary
- Otherwise, just like list comprehension
  - Can filter it (with an if at then end)
  - Can nest it with other comprehension

Traditional For-Loops

```
def halve_grades(grades):
    """Returns a copy cutting all exam grades in half.

    Precondition: grades has netids as keys, ints as values"""
    result = { }
    for k in grades:
        result[k] = grades[k]//2
    return result
```

Traditional For-Loops

```
def halve_grades(grades):
    """Returns a copy cutting all exam grades in half.

    Precondition: grades has netids as keys, ints as values"""
    return { k:grades[k]//2 for k in grades }
```

Traditional For-Loops

```
def extra_credit(grades,students,bonus):
    """Returns a copy of grades with extra credit assigned

    Precond: grades has netids as keys, ints as values.
    netids is a list of valid (string) netids, bonus an int"""
    result = { }
    for k in grades:
        if k in students:
            result[k] = grades[k]+bonus
        else:
            result[k] = grades[k]
    return result
```

Traditional For-Loops

```
def extra_credit(grades,students,bonus):
    """Returns a copy of grades with extra credit assigned

    Precond: grades has netids as keys, ints as values.
    netids is a list of valid (string) netids, bonus an int"""
    return { k:(grades[k]+bonus if k in students else grades[k])
                    for k in grades }
```

Final Words on Comprehension

Advantages	Disadvantages
Code very compact/concise	Harder to read/understand
Python can optimize heavily (no wasteful accumulators)	Much harder to debug (more stuff on one line)

Use this technique sparingly