

61A Lecture 29

Friday, April 10

Announcements

- Homework 8 due Wednesday 4/15 @ 11:59pm (small)
- Project 4 due Thursday 4/23 @ 11:59pm (BIG!)
 - Early point #1: Questions 1–12 submitted (correctly) by Friday 4/17 @ 11:59pm
 - Early point #2: All questions (including Extra Credit) by Wednesday 4/22 @ 11:59pm

Data Processing

Processing Sequential Data

Many data sets can be processed sequentially:

- The set of all Twitter posts
- Votes cast in an election
- Sensor readings of an airplane
- The positive integers: 1, 2, 3, ...

However, the **sequence interface** we used before does not always apply

- A sequence has a finite, known length
- A sequence allows element selection for any element

Some important ideas in **big data processing**:

- Implicit representations of streams of sequential data
- Declarative programming languages to manipulate and transform data
- Distributed computing

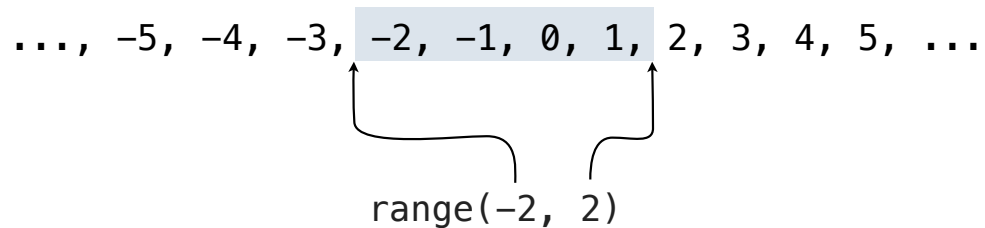
Implicit Sequences

Implicit Sequences

An implicit sequence is a representation of sequential data that does not explicitly store each element

Example: The built-in `range` class represents consecutive integers

- The range is represented by two values: start and end
- The length and elements are computed on demand
- Constant space for arbitrarily long sequences



(Demo)

Iterators

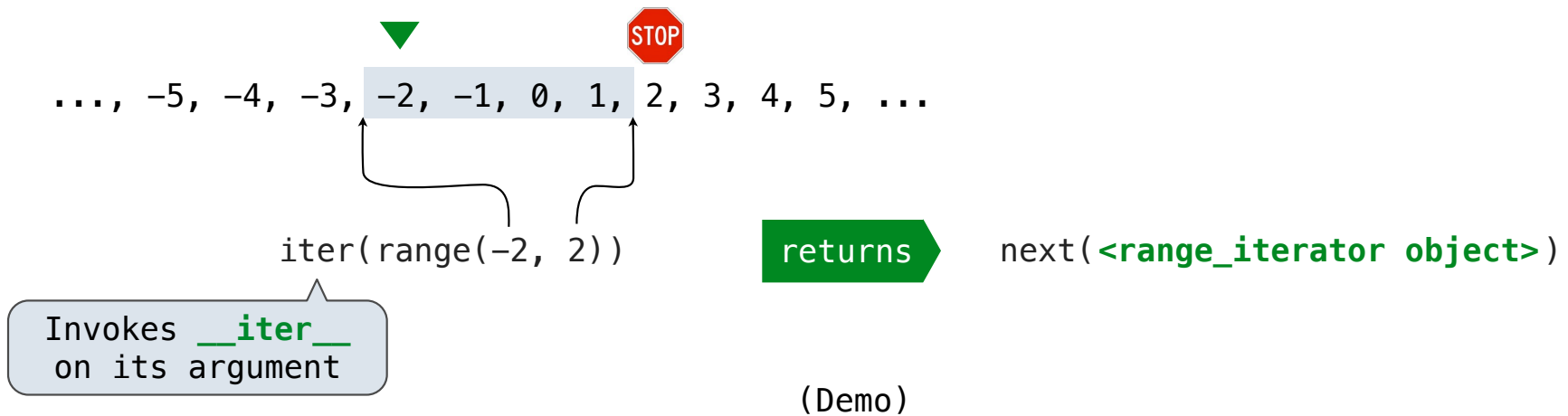
The Iterator Interface

An iterator is an object that can provide the next element of a sequence

The `__next__` method of an iterator returns the next element

The built-in `next` function invokes the `__next__` method on its argument

If there is no next element, then the `__next__` method of an iterator should raise a `StopIteration` exception



Iterable Objects

Iterables and Iterators

Iterator: Mutable object that tracks a position in a sequence, advancing on `__next__`

Iterable: Represents a sequence and returns a new iterator on `__iter__`

LetterIter is an iterator:

LetterIter('a', 'e')



LetterIter('a', 'e')



Letters is iterable:

Letters('a', 'e')

'a'

'b'

'c'

'd'

(Demo)

Built-in Iterators

Iterators from Built-in Functions

Many built-in Python sequence operations return iterators that compute results lazily

<code>map(func, iterable):</code>	Iterate over <code>func(x)</code> for <code>x</code> in <code>iterable</code>
<code>filter(func, iterable):</code>	Iterate over <code>x</code> in <code>iterable</code> if <code>func(x)</code>
<code>zip(first_iter, second_iter):</code>	Iterate over co-indexed <code>(x, y)</code> pairs
<code>reversed(sequence):</code>	Iterate over <code>x</code> in a sequence in reverse order

To view the results, place the resulting elements in a sequence

<code>list(iterable):</code>	Create a list containing all <code>x</code> in <code>iterable</code>
<code>tuple(iterable):</code>	Create a tuple containing all <code>x</code> in <code>iterable</code>
<code>sorted(iterable):</code>	Create a sorted list containing <code>x</code> in <code>iterable</code>

(Demo)

For Statements

The For Statement

```
for <name> in <expression>:  
    <suite>
```

1. Evaluate the header `<expression>`, which must evaluate to an iterable object
2. For each element in that sequence, in order:
 - A. Bind `<name>` to that element in the first frame of the current environment
 - B. Execute the `<suite>`

When executing a `for` statement, `__iter__` returns an iterator and `__next__` provides each item:

```
>>> counts = [1, 2, 3]  
>>> for item in counts:  
    print(item)
```

```
1  
2  
3
```

```
>>> counts = [1, 2, 3]  
>>> items = counts.__iter__()  
>>> try:  
    while True:  
        item = items.__next__()  
        print(item)  
except StopIteration:  
    pass # Do nothing
```

```
1  
2  
3
```

Generator Functions

Generators and Generator Functions

A generator function is a function that yields values instead of returning them

A normal function returns once; a generator function yields multiple times

A generator is an iterator, created by a *generator function*

When a generator function is called, it returns a generator that iterates over yields

```
>>> def letter_generator(next_letter, end):
    while next_letter < end:
        yield next_letter
        next_letter = chr(ord(next_letter)+1)
```

```
>>> s = letter_generator('a', 'z')
>>> next(s)
'a'
>>> next(s)
'b'
```

(Demo)