61A Lecture 29

Friday, November 15

Announcements

- Homework 9 due Tuesday 11/19 @ 11:59pm
- Project 4 due Thursday 11/21 @ 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.

Important ideas in **big data processing:**

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

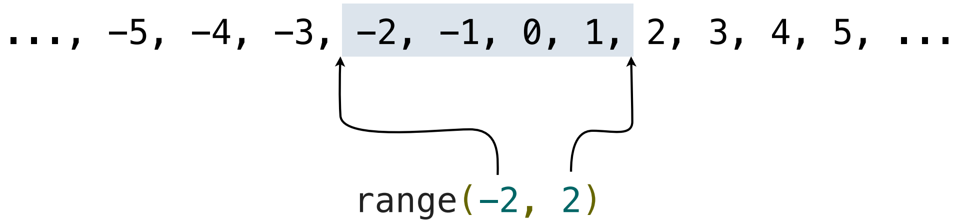
Implicit Sequences

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.



..., -5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5, ...

```
range(-2, 2)
```

(Demo)

Iterators

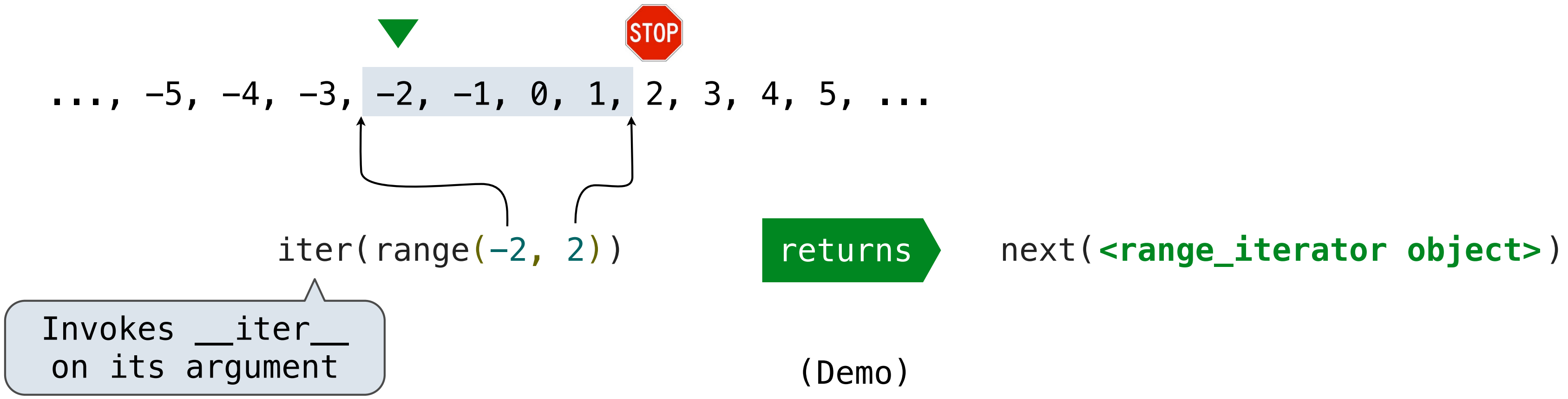
The Iterator Interface

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.



(Demo)

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)

For Statements

The For Statement

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

1. Evaluate the header <expression>, which yields 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
1
2
3
```

Generator Functions

Generators and Generator Functions

A generator is an iterator backed by a generator function.

A generator function is a function that yields values.

When a generator function is called, it returns a generator.

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

>>> for letter in letters_generator('a', 'e'):
...     print(letter)
a
b
c
d
```

(Demo)

Generator Examples

```
fib_generator(): "Yield Fibonacci numbers."

all_pairs(s): "Yield pairs of elements from iterable s."

Letters.__iter__(): "Yield sequential letters."

powerset(t): "Yield all subsets of iterator t."
```