

61A Lecture 30

Wednesday, November 12

Announcements

Announcements

- Homework 8 due Wednesday 11/12 @ 11:59pm

Announcements

- Homework 8 due Wednesday 11/12 @ 11:59pm
- Project 4 due ~~Thursday~~ Friday 11/21 @ 11:59pm

Announcements

- Homework 8 due Wednesday 11/12 @ 11:59pm
- Project 4 due ~~Thursday~~ Friday 11/21 @ 11:59pm
 - Early submission point #1: Questions 1–6 by Friday 11/14 @ 11:59pm

Announcements

- Homework 8 due Wednesday 11/12 @ 11:59pm
- Project 4 due ~~Thursday~~ Friday 11/21 @ 11:59pm
 - Early submission point #1: Questions 1–6 by Friday 11/14 @ 11:59pm
 - Early submission point #2: Questions 1–16 by Tuesday 11/18 @ 11:59pm

Announcements

- Homework 8 due Wednesday 11/12 @ 11:59pm
- Project 4 due ~~Thursday~~ Friday 11/21 @ 11:59pm
 - Early submission point #1: Questions 1–6 by Friday 11/14 @ 11:59pm
 - Early submission point #2: Questions 1–16 by Tuesday 11/18 @ 11:59pm
 - Early submission point #3: Submit by Thursday 11/20 @ 11:59pm

Announcements

- Homework 8 due Wednesday 11/12 @ 11:59pm
- Project 4 due ~~Thursday~~ Friday 11/21 @ 11:59pm
 - Early submission point #1: Questions 1–6 by Friday 11/14 @ 11:59pm
 - Early submission point #2: Questions 1–16 by Tuesday 11/18 @ 11:59pm
 - Early submission point #3: Submit by Thursday 11/20 @ 11:59pm
- Homework 9 combined with Homework 10 (6 pts), due Wednesday 11/26 @ 11:59pm

Announcements

- Homework 8 due Wednesday 11/12 @ 11:59pm
- Project 4 due ~~Thursday~~ Friday 11/21 @ 11:59pm
 - Early submission point #1: Questions 1–6 by Friday 11/14 @ 11:59pm
 - Early submission point #2: Questions 1–16 by Tuesday 11/18 @ 11:59pm
 - Early submission point #3: Submit by Thursday 11/20 @ 11:59pm
- Homework 9 combined with Homework 10 (6 pts), due Wednesday 11/26 @ 11:59pm
- Guest lecture Friday 11/14 by Andrew Huang

Survey Results

Survey Results

Midterm 2 was difficult and discouraging

Survey Results

Midterm 2 was difficult and discouraging

Too many adjacent deadlines

Survey Results

Midterm 2 was difficult and discouraging

Too many adjacent deadlines

Too hard to get help in office hours and homework/project parties

Survey Results

Midterm 2 was difficult and discouraging

Too many adjacent deadlines

Too hard to get help in office hours and homework/project parties

Better guidance on how to solve problems

Survey Results

Midterm 2 was difficult and discouraging

Too many adjacent deadlines

Too hard to get help in office hours and homework/project parties

Better guidance on how to solve problems

Pace, information overload, and extra credit

Data Processing

Processing Sequential Data

Processing Sequential Data

Many data sets can be processed sequentially:

Processing Sequential Data

Many data sets can be processed sequentially:

- The set of all Twitter posts

Processing Sequential Data

Many data sets can be processed sequentially:

- The set of all Twitter posts
- Votes cast in an election

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

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, ...

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.

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.

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.

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**:

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

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

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 and parallel computing

Implicit Sequences

Implicit Sequences

Implicit Sequences

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

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

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

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

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

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, ...`

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)`

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)`




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, ...



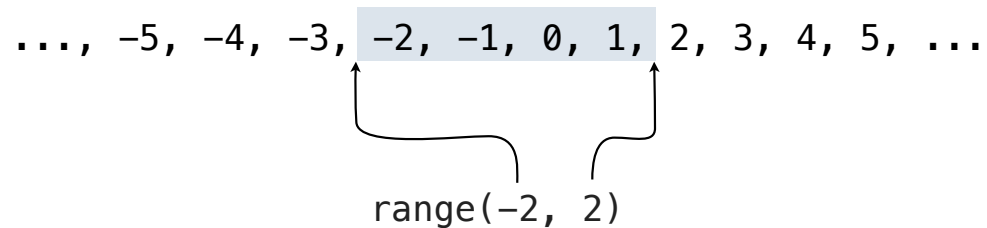
range(-2, 2)

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

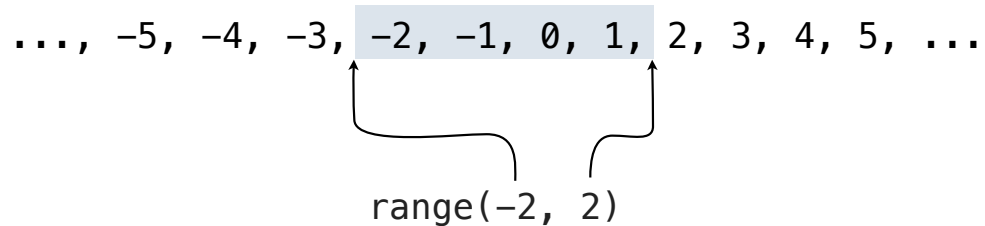


Implicit Sequences

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

The Iterator Interface

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

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 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.

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.

The Iterator Interface

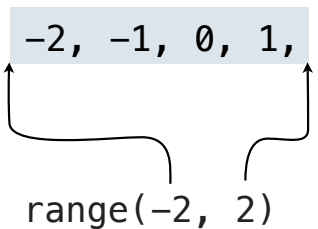
An iterator is an object that can provide the next element of a sequence.

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.

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



range(-2, 2)

The Iterator Interface

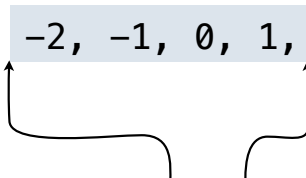
An iterator is an object that can provide the next element of a sequence.

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.

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



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

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.

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

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

Invokes `__iter__`
on its argument

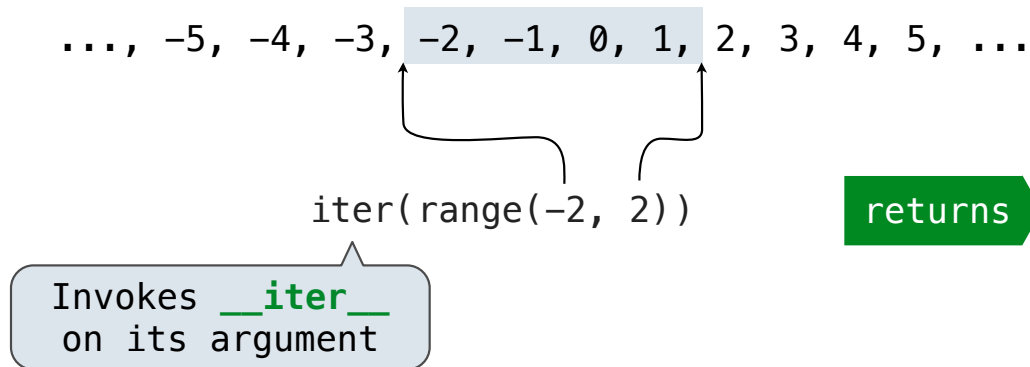
The Iterator Interface

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.



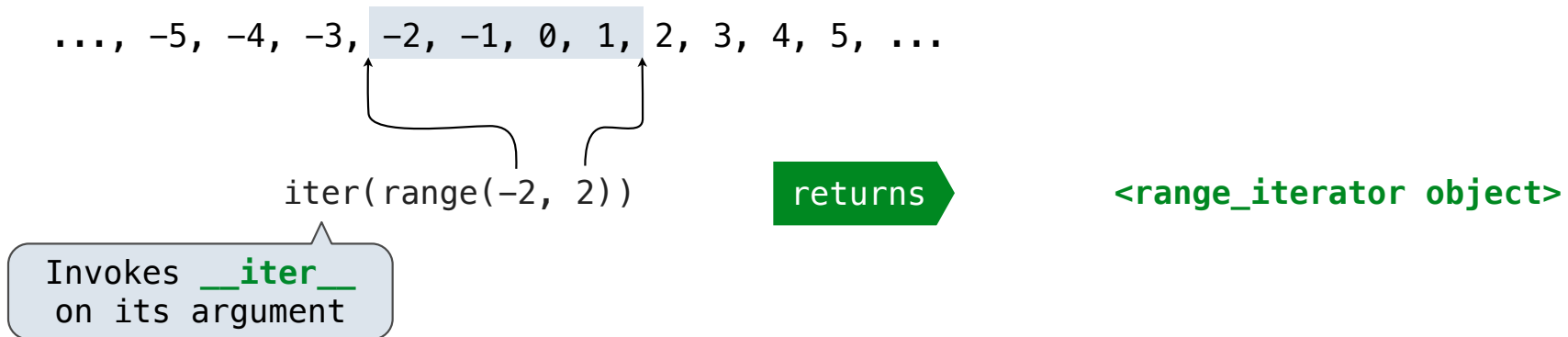
The Iterator Interface

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.



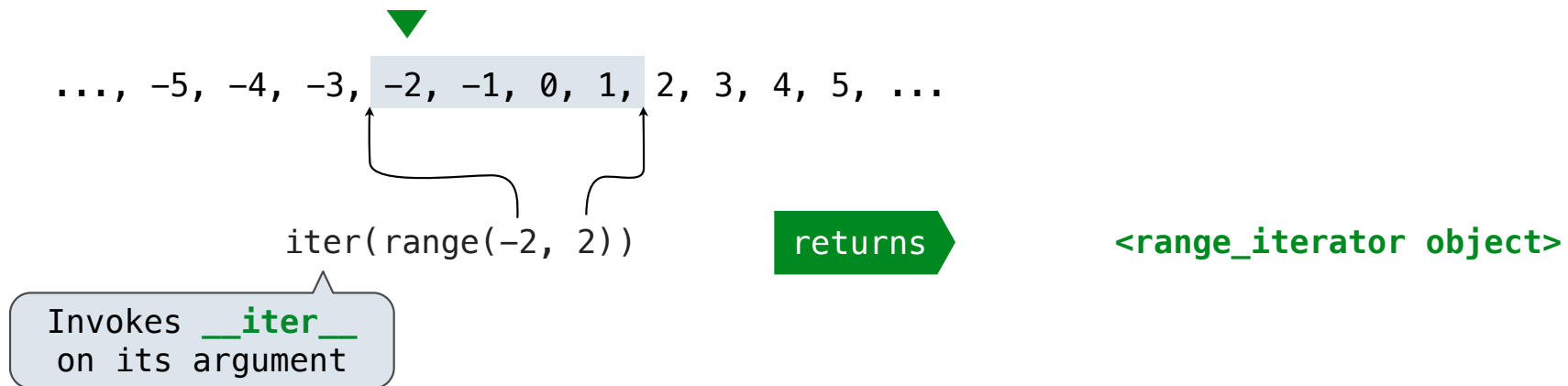
The Iterator Interface

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.



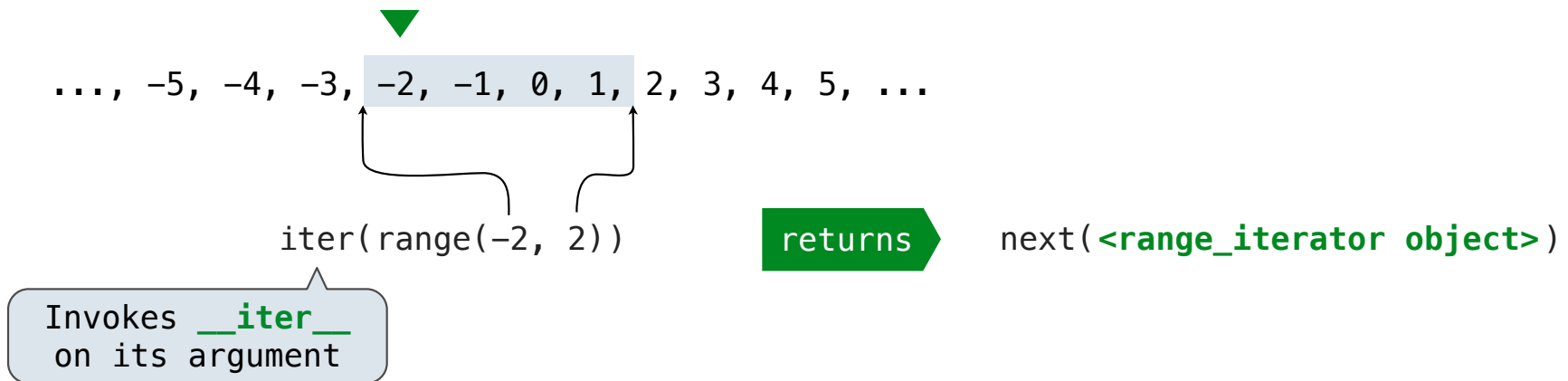
The Iterator Interface

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.



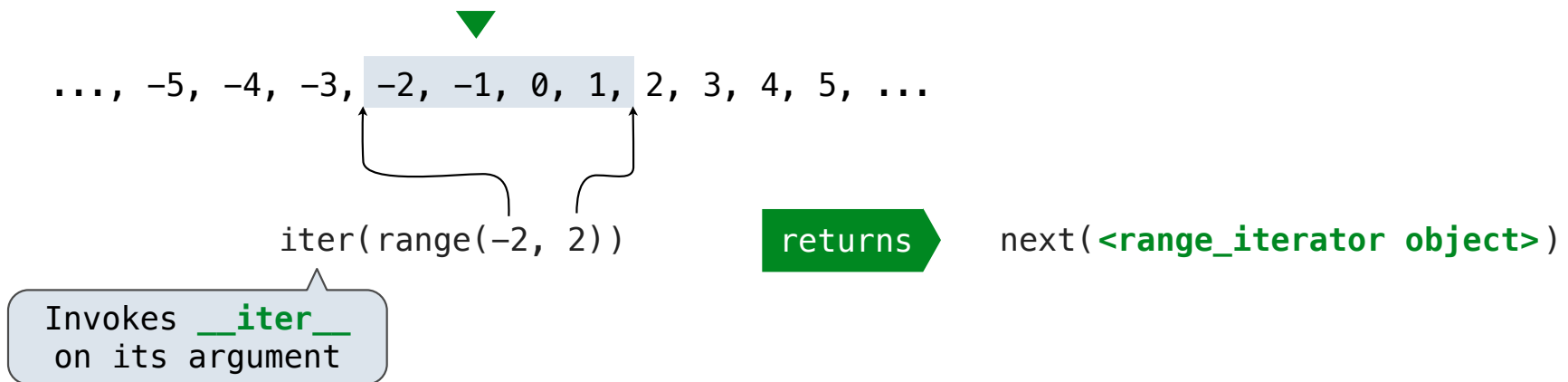
The Iterator Interface

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.



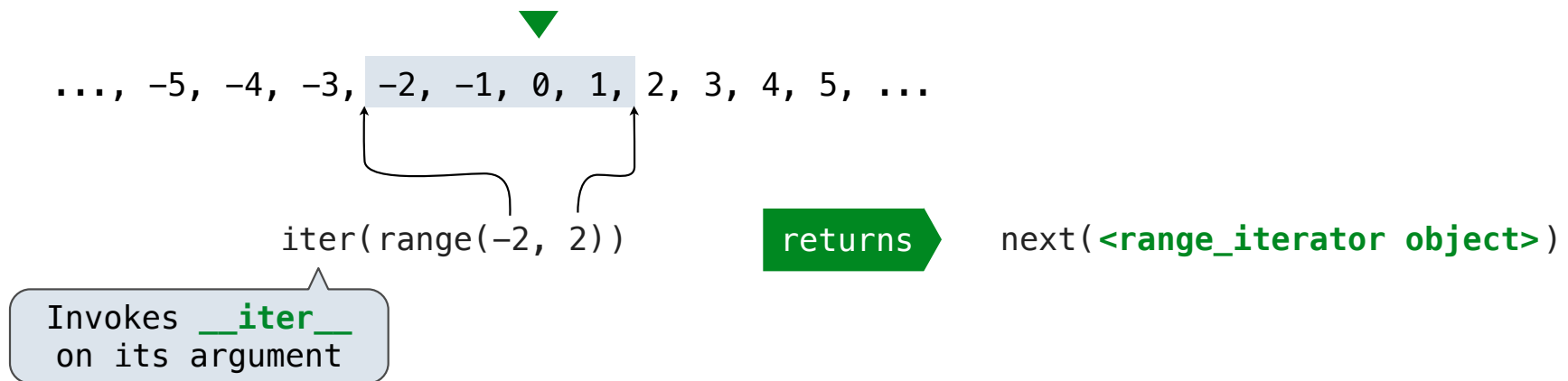
The Iterator Interface

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.



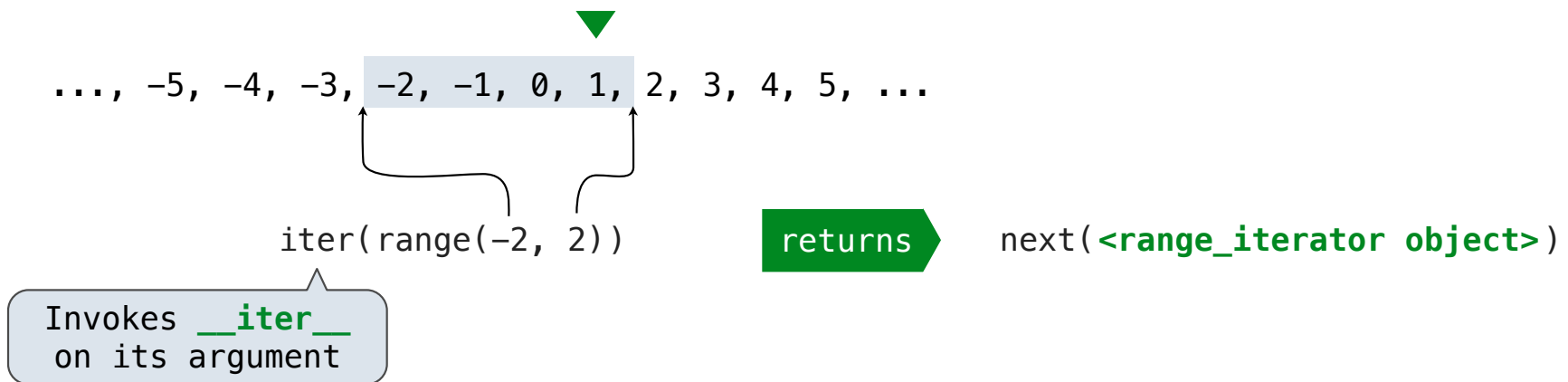
The Iterator Interface

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.



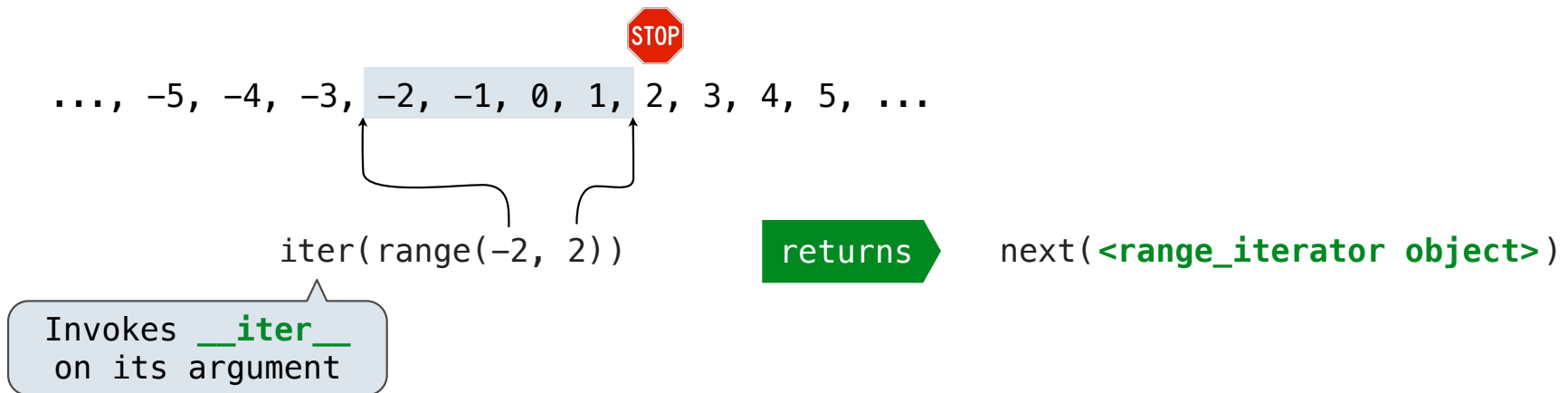
The Iterator Interface

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.



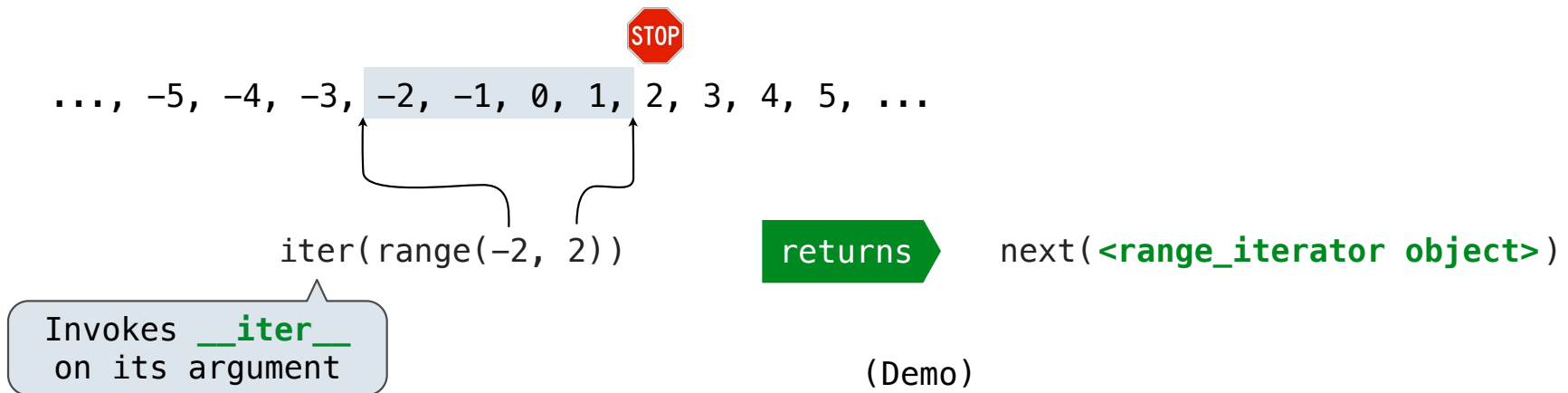
The Iterator Interface

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

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__`

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:

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:

`Letters` is iterable:

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:

Letters is iterable:

`Letters('a', 'e')`

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:

Letters is iterable:

Letters('a', 'e') 'a' 'b' 'c' 'd'

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')`

Letters is iterable: `Letters('a', 'e')` 'a' 'b' 'c' 'd'

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') ▼

Letters is iterable: Letters('a', 'e') 'a' 'b' 'c' 'd'

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'

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'

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'

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'

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'

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'

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

Iterators from Built-in Functions

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

`map(func, iterable):` Iterate over `func(x)` for `x` in `iterable`

Iterators from Built-in Functions

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

`map(func, iterable):` Iterate over `func(x)` for `x` in `iterable`

`filter(func, iterable):` Iterate over `x` in `iterable` if `func(x)`

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

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

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

Iterators from Built-in Functions

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

`map(func, iterable):` Iterate over `func(x)` for `x` in `iterable`

`filter(func, iterable):` Iterate over `x` in `iterable` if `func(x)`

`zip(first_iter, second_iter):` Iterate over co-indexed `(x, y)` pairs

`reversed(sequence):` Iterate over `x` in a sequence in reverse order

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

`list(iterable):` Create a list containing all `x` in `iterable`

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>

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>

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

The For Statement

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

The For Statement

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

1. Evaluate the header `<expression>`, which yields an iterable object

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:

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

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>`

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>`

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:

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
```

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 # Do nothing
```

```
1  
2  
3
```

Generator Functions

Generators and Generator Functions

Generators and Generator Functions

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

Generators and Generator Functions

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

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

Generators and Generator Functions

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

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

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

Generators and Generator Functions

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

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

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

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

Generators and Generator Functions

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

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

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

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

```
>>> def letters_generator(next_letter, end):
```



Generators and Generator Functions

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

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

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

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

```
>>> def letters_generator(next_letter, end):  
    while next_letter < end:
```



Generators and Generator Functions

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

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

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

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

```
>>> def letters_generator(next_letter, end):  
    while next_letter < end:  
        yield next_letter
```



Generators and Generator Functions

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

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

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

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

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



Generators and Generator Functions

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

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

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

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

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



Generators and Generator Functions

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

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

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

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

```
>>> 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'):
```

Generators and Generator Functions

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

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

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

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

```
>>> 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', 'z'):
    print(letter)
```

Generators and Generator Functions

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

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

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

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

```
>>> 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', 'z'):
    print(letter)
```

a

Generators and Generator Functions

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

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

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

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

```
>>> 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', 'c'):  
    print(letter)
```

```
a  
b
```

Generators and Generator Functions

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

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

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

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

```
>>> 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', 'd'):
    print(letter)
```

```
a
b
c
```

Generators and Generator Functions

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

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

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

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

```
>>> 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', 'd'):  
    print(letter)
```

```
a  
b  
c  
d
```

Generators and Generator Functions

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

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

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

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

```
>>> 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', 'd'):  
    print(letter)
```

```
a  
b  
c  
d
```

(Demo)