Announcements
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 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.

\[ \ldots, -5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5, \ldots \]

\[ \text{range}(-2, 2) \]
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.

\[
\ldots, -5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5, \ldots
\]

\text{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.

\[ \ldots, -5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5, \ldots \]

`range(-2, 2)` (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 \texttt{\_\_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 \texttt{\_\_next\_} method of an iterator returns the next element

The built-in \texttt{next} function invokes the \texttt{\_\_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 \texttt{\_\_next\_} method of an iterator returns the next element

The built-in \texttt{next} function invokes the \texttt{\_\_next\_} method on its argument

If there is no next element, then the \texttt{\_\_next\_} method of an iterator should raise a \texttt{StopIteration} exception

\[ \ldots, -5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5, \ldots \]
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.

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

```python
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.

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

```
invokes __iter__ on its argument
```

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

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))  returns  <range_iterator object>
```
The Iterator Interface

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

The \texttt{\_\_next\_\_} method of an iterator returns the next element

The built-in \texttt{next} function invokes the \texttt{\_\_next\_\_} method on its argument

If there is no next element, then the \texttt{\_\_next\_\_} method of an iterator should raise a \texttt{StopIteration} exception

\begin{verbatim}
..., -5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5, ...
\end{verbatim}

\begin{verbatim}
iter(range(-2, 2)) returns <range_iterator object>
\end{verbatim}

Invokes \texttt{\_\_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.

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

```python
iter(range(-2, 2))  # returns <range_iterator object>
next(<range_iterator object>)
```

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.

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

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

`next(<range_iterator object>)`

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

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

```
iter(range(-2, 2))    returns    next(<range_iterator object>)
```

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 \texttt{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 \texttt{StopIteration} exception

\texttt{...}, \texttt{-5}, \texttt{-4}, \texttt{-3}, \texttt{-2}, \texttt{-1}, \texttt{0}, \texttt{1}, \texttt{2}, \texttt{3}, \texttt{4}, \texttt{5}, ... \texttt{STOP}

\texttt{iter(range(-2, 2))} \quad \texttt{return} \quad \texttt{next(<range_iterator object>)}

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.

(Demo)
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__`

```python
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')`

**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:  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

```python
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

```python
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:

- `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
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.
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
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:

- `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`
- `tuple(iterable)`: Create a tuple containing all `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 \( \text{func}(x) \) for \( x \) in iterable
- **filter(func, iterable):** Iterate over \( x \) in iterable if \( \text{func}(x) \)
- **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
- **tuple(iterable):** Create a tuple containing all \( x \) in iterable
- **sorted(iterable):** Create a sorted list containing \( x \) 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)`
- `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`
- `tuple(iterable)`: Create a tuple containing all `x` in `iterable`
- `sorted(iterable)`: Create a sorted list containing `x` in `iterable`

(Demo)
For Statements
The For Statement

for <name> in <expression>:
    <suite>
The For Statement

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

1. Evaluate the header `<expression>`, which must evaluate to an iterable object
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:
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
The For Statement

```python
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>`
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>
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:
The For Statement

```python
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:

```python
>>> 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 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:

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

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

```python
```
Generator Functions
Generators and Generator Functions
Generators and Generator Functions

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

```python
>>> def letter_generator(next_letter, end):
```
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.

```python
>>> def letter_generator(next_letter, end):
    while next_letter < end:
```
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.

```python
>>> def letter_generator(next_letter, end):
    while next_letter < end:
        yield next_letter
```
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.

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

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

```python
>>> 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')
```
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.

```python
>>> 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)
```
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

```python
>>> 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'
```
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.

```python
>>> 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)
16
```
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.

```python
>>> 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'
```
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.

```python
>>> 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)