Announcements
Data Processing
Data Processing
Data Processing

Many data sets can be processed sequentially:
Data Processing

Many data sets can be processed sequentially:
• The set of all Twitter posts
Data Processing

Many data sets can be processed sequentially:
- The set of all Twitter posts
- Votes cast in an election
Data Processing

Many data sets can be processed sequentially:
- The set of all Twitter posts
- Votes cast in an election
- Sensor readings of an airplane
Data Processing

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, ...
Data Processing

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.
Data Processing

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
Data Processing

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
Data Processing

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:
Data Processing

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
Data Processing

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
Data Processing

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
Iterators
Iterators
Iterators

A container can provide an iterator that provides access to its elements in some order
Iterators

A container can provide an iterator that provides access to its elements in some order

- \texttt{iter(iterable)}: Return an iterator over the elements of an iterable value
- \texttt{next(iterator)}: Return the next element in an iterator
Iterators

A container can provide an iterator that provides access to its elements in some order

```python
>>> s = [3, 4, 5]
```

- `iter(iterable)`: Return an iterator over the elements of an iterable value

- `next(iterator)`: Return the next element in an iterator
Iterators

A container can provide an iterator that provides access to its elements in some order

iter(iterable): Return an iterator over the elements of an iterable value

next(iterator): Return the next element in an iterator

>>> s = [3, 4, 5]
>>> t = iter(s)
A container can provide an iterator that provides access to its elements in some order

```python
>>> s = [3, 4, 5]
>>> t = iter(s)
```

**`iter(iterable)`:** Return an iterator over the elements of an iterable value

**`next(iterator)`:** Return the next element in an iterator
Iterators

A container can provide an iterator that provides access to its elements in some order

**iter**(iterable): Return an iterator over the elements of an iterable value

**next**(iterator): Return the next element in an iterator

```python
>>> s = [3, 4, 5]
>>> t = iter(s)
>>> next(t)
3
```
Iterators

A container can provide an iterator that provides access to its elements in some order

iter(iterable): Return an iterator over the elements of an iterable value

next(iterator): Return the next element in an iterator

```python
>>> s = [3, 4, 5]
>>> t = iter(s)
>>> next(t)
3
```
Iterators

A container can provide an iterator that provides access to its elements in some order.

- `iter(iterable)`: Return an iterator over the elements of an iterable value.
- `next(iterator)`: Return the next element in an iterator.

```python
>>> s = [3, 4, 5]
>>> t = iter(s)
>>> next(t)
3
>>> next(t)
4
```
Iterators

A container can provide an iterator that provides access to its elements in some order

```
iter(iterable): Return an iterator over the elements of an iterable value

next(iterator): Return the next element in an iterator
```

```python
>>> s = [3, 4, 5]
>>> u = iter(s)
>>> t = iter(s)
>>> next(t)
3
>>> next(t)
4
```
Iterators

A container can provide an iterator that provides access to its elements in some order

`iter(iterable)`: Return an iterator over the elements of an iterable value

`next(iterator)`: Return the next element in an iterator

```
>>> s = [3, 4, 5]
>>> t = iter(s)
>>> next(t)  # 3
3
>>> next(t)  # 4
4
>>> u = iter(s)
>>> next(u)  # 3
```
Iterators

A container can provide an iterator that provides access to its elements in some order

```python
iter(iterable): Return an iterator over the elements of an iterable value
next(iterator): Return the next element in an iterator
```

```python
>>> s = [3, 4, 5]
>>> t = iter(s)
>>> next(t)
3
>>> next(t)
4
>>> u = iter(s)
>>> next(u)
3
>>> next(u)
5
```
Iterators

A container can provide an iterator that provides access to its elements in some order

\textbf{iter}(iterable): Return an iterator over the elements of an iterable value

\textbf{next}(iterator): Return the next element in an iterator

\begin{verbatim}
>>> s = [3, 4, 5]
>>> t = iter(s)
>>> next(t)
3
>>> next(t)
5
>>> next(t)
4
>>> u = iter(s)
>>> next(u)
3
>>> next(u)
5
>>> next(u)
4
\end{verbatim}
Iterators

A container can provide an iterator that provides access to its elements in some order

```
iter(iterable): Return an iterator over the elements of an iterable value
next(iterator): Return the next element in an iterator
```

Iterators are always ordered, even if the container that produced them is not

```
>>> s = [3, 4, 5]
>>> t = iter(s)
>>> next(t)
3
>>> u = iter(s)
>>> next(u)
3
>>> next(t)
5
>>> next(u)
4
```
Iterators

A container can provide an iterator that provides access to its elements in some order

```
iter(iterable): Return an iterator over the elements of an iterable value

next(iterator): Return the next element in an iterator
```

Iterators are always ordered, even if the container that produced them is not

```
>>> s = [3, 4, 5]
>>> t = iter(s)
>>> next(t) 3
>>> next(t) 4
>>> u = iter(s)
>>> next(u) 3
>>> next(u) 4

>>> d = {'one': 1, 'two': 2, 'three': 3}
```
Iterators

A container can provide an iterator that provides access to its elements in some order

iter(iterable): Return an iterator over the elements of an iterable value

next(iterator): Return the next element in an iterator

Iterators are always ordered, even if the container that produced them is not

```python
>>> s = [3, 4, 5]
>>> t = iter(s)
>>> next(t)
3
>>> next(t)
4
>>> u = iter(s)
>>> next(u)
3
>>> next(t)
5
>>> next(u)
4

>>> d = {'one': 1, 'two': 2, 'three': 3}
>>> k = iter(d)
```
Iterators

A container can provide an iterator that provides access to its elements in some order

iter(iterable): Return an iterator over the elements of an iterable value

next(iterator): Return the next element in an iterator

Iterators are always ordered, even if the container that produced them is not

>>> s = [3, 4, 5]
>>> t = iter(s)
>>> next(t) 3
>>> next(t) 4
>>> u = iter(s)
>>> next(u) 3
>>> next(u) 4

>>> d = {'one': 1, 'two': 2, 'three': 3}
>>> k = iter(d)
>>> next(k) 'one'
Iterators

A container can provide an iterator that provides access to its elements in some order

\[
\text{iter}(\text{iterable}): \text{Return an iterator over the elements of an iterable value}
\]

\[
\text{next}(\text{iterator}): \text{Return the next element in an iterator}
\]

Iterators are always ordered, even if the container that produced them is not

```python
def main():
    s = [3, 4, 5]
    t = iter(s)
    print(next(t))  # 3
    print(next(t))  # 4
    u = iter(s)
    print(next(u))  # 3
    print(next(u))  # 4

    d = {'one': 1, 'two': 2, 'three': 3}
    k = iter(d)
    print(next(k))  # 'one'
    print(next(k))  # 'three'
```

```
Iterators

A container can provide an iterator that provides access to its elements in some order

\[
\text{iter(Iterable): Return an iterator over the elements of an iterable value}
\]

\[
\text{next(Iterator): Return the next element in an iterator}
\]

Iterators are always ordered, even if the container that produced them is not

```python
>>> s = [3, 4, 5]
>>> u = iter(s)
>>> t = iter(s)
>>> next(u)
3
>>> next(t)
3
>>> next(t)
5
>>> next(u)
4
```

```python
>>> d = {\'one\': 1, \'two\': 2, \'three\': 3}
>>> k = iter(d)
>>> next(k)
\'one\'
>>> next(k)
\'three\'
>>> next(k)
\'two\'
```
Iterators

A container can provide an iterator that provides access to its elements in some order

\[
\begin{align*}
\text{iter}(\text{iterable}): & \text{ Return an iterator over the elements of an iterable value} \\
\text{next}(\text{iterator}): & \text{ Return the next element in an iterator}
\end{align*}
\]

Iterators are always ordered, even if the container that produced them is not

\[
\begin{align*}
>>> s &= [3, 4, 5] \\
>>> u &= \text{iter}(s) \\
>>> t &= \text{iter}(s) \\
>>> v &= \text{next}(u) \\
>>> \text{next}(u) &= 3 \\
>>> \text{next}(v) &= 3 \\
>>> \text{next}(t) &= 4 \\
>>> \text{next}(u) &= 4 \\
\end{align*}
\]

Keys and values are iterated over in an arbitrary order which is non-random, varies across Python implementations, and depends on the dictionary’s history of insertions and deletions. If keys, values and items views are iterated over with no intervening modifications to the dictionary, the order of items will directly correspond.

https://docs.python.org/3/library/stdtypes.html#dictionary-view-objects
Iterators

A container can provide an iterator that provides access to its elements in some order

\[
\begin{align*}
\text{iter(iterable)}: \text{Return an iterator over the elements of an iterable value} \\
\text{next(iterator)}: \text{Return the next element in an iterator}
\end{align*}
\]

Iterators are always ordered, even if the container that produced them is not

```python
>>> s = [3, 4, 5]
>>> t = iter(s)
>>> next(t)
3
>>> u = iter(s)
>>> next(t)
3
>>> next(u)
4
```

Keys and values are iterated over in an arbitrary order which is non-random, varies across Python implementations, and depends on the dictionary’s history of insertions and deletions. If keys, values and items views are iterated over with no intervening modifications to the dictionary, the order of items will directly correspond.

https://docs.python.org/3/library/stdtypes.html#dictionary-view-objects
## Iterators

A container can provide an iterator that provides access to its elements in some order.

- **`iter(iterable)`**: Return an iterator over the elements of an iterable value.
- **`next(iterator)`**: Return the next element in an iterator.

Iterators are always ordered, even if the container that produced them is not.

```python
>>> s = [3, 4, 5]
>>> u = iter(s)
>>> t = iter(s)
>>> next(u)
3
>>> next(t)
3
>>> next(t)
5
>>> next(u)
4
```

```python
>>> d = {'one': 1, 'two': 2, 'three': 3}
>>> k = iter(d)
>>> v = iter(d.values())
>>> next(k)
'one'
>>> next(v)
1
>>> next(k)
'three'
>>> next(k)
'two'
```

Keys and values are iterated over in an arbitrary order which is non-random, varies across Python implementations, and depends on the dictionary’s history of insertions and deletions. If keys, values and items views are iterated over with no intervening modifications to the dictionary, the order of items will directly correspond.

[https://docs.python.org/3/library/stdtypes.html#dictionary-view-objects](https://docs.python.org/3/library/stdtypes.html#dictionary-view-objects)
Iterators

A container can provide an iterator that provides access to its elements in some order.

```
iter(iterable): Return an iterator over the elements of an iterable value

next(iterator): Return the next element in an iterator
```

Iterators are always ordered, even if the container that produced them is not.

```
>>> s = [3, 4, 5]  >>> u = iter(s)
>>> t = iter(s)  >>> next(u)
>>> next(t)  3
>>> next(t)  3
>>> next(t)  4
```

Keys and values are iterated over in an arbitrary order which is non-random, varies across Python implementations, and depends on the dictionary’s history of insertions and deletions. If keys, values and items views are iterated over with no intervening modifications to the dictionary, the order of items will directly correspond.

```
>>> d = {'one': 1, 'two': 2, 'three': 3}
>>> k = iter(d)  >>> v = iter(d.values())
>>> next(k)  >>> next(v)
'one'  1
>>> next(k)  >>> next(v)
'three'  3
>>> next(k)  'two'
```

https://docs.python.org/3/library/stdtypes.html#dictionary-view-objects
Iterators

A container can provide an iterator that provides access to its elements in some order

\[
\text{iter(iterable): Return an iterator over the elements of an iterable value}
\]

\[
\text{next(iterator): Return the next element in an iterator}
\]

Iterators are always ordered, even if the container that produced them is not

\[
\begin{align*}
\text{s} &= [3, 4, 5] \\
\text{u} &= \text{iter(s)} \\
\text{t} &= \text{iter(s)} \\
\text{v} &= \text{iter(d.values())}
\end{align*}
\]

Keys and values are iterated over in an arbitrary order which is non-random, varies across Python implementations, and depends on the dictionary’s history of insertions and deletions. If keys, values and items views are iterated over with no intervening modifications to the dictionary, the order of items will directly correspond.

https://docs.python.org/3/library/stdtypes.html#dictionary-view-objects
Iterators

A container can provide an iterator that provides access to its elements in some order

```python
iter(iterable): Return an iterator over the elements of an iterable value
```

```python
next(iterator): Return the next element in an iterator
```

Iterators are always ordered, even if the container that produced them is not

```python
>>> s = [3, 4, 5]
>>> u = iter(s)
>>> t = iter(s)
>>> next(u)
3
>>> next(t)
3
>>> next(u)
4
>>> next(t)
5
```

```python
d = {'one': 1, 'two': 2, 'three': 3}
>>> k = iter(d)
>>> v = iter(d.values())
>>> next(k)
'one'
1
>>> next(k)
'three'
3
>>> next(k)
'two'
2
```

Keys and values are iterated over in an arbitrary order which is non-random, varies across Python implementations, and depends on the dictionary’s history of insertions and deletions. If keys, values and items views are iterated over with no intervening modifications to the dictionary, the order of items will directly correspond.

(Demo)

https://docs.python.org/3/library/stdtypes.html#dictionary-view-objects
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 must evaluate to an iterable object
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:
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.
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

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 = iter(counts)
>>> try:
    while True:
        item = next(items)
        print(item)
except StopIteration:
    pass  # Do nothing
1
2
3
```
Processing Iterators
Processing Iterators

A `StopIteration` exception is raised whenever `next` is called on an empty iterator.
Processing Iterators

A `StopIteration` exception is raised whenever `next` is called on an empty iterator

```python
>>> contains('strength', 'stent')
True
```
A `StopIteration` exception is raised whenever `next` is called on an empty iterator.

```python
>>> contains('strength', 'stent')
True
>>> contains('strength', 'rest')
False
```
Processing Iterators

A **StopIteration** exception is raised whenever `next` is called on an empty iterator.

```python
>>> contains('strength', 'stent')
True
>>> contains('strength', 'rest')
False
>>> contains('strength', 'tenth')
True
```
Processing Iterators

A `StopIteration` exception is raised whenever `next` is called on an empty iterator.

```python
def contains(a, b):
    # Your implementation here
```

```python
>>> contains('strength', 'stent')
True
>>> contains('strength', 'rest')
False
>>> contains('strength', 'tenth')
True
```
A `StopIteration` exception is raised whenever `next` is called on an empty iterator.

```python
def contains(a, b):
    ai = iter(a)
```

```python
>>> contains('strength', 'stent')
True
>>> contains('strength', 'rest')
False
>>> contains('strength', 'tenth')
True
```
A `StopIteration` exception is raised whenever `next` is called on an empty iterator.

```python
def contains(a, b):
    ai = iter(a)
```

```python
>>> contains('strength', 'stent')
True
>>> contains('strength', 'rest')
False
>>> contains('strength', 'tenth')
True
```
Processing Iterators

A **StopIteration** exception is raised whenever `next` is called on an empty iterator.

```python
>>> contains('strength', 'stent')
True
>>> contains('strength', 'rest')
False
>>> contains('strength', 'tenth')
True
```

```python
def contains(a, b):
    ai = iter(a)
    for x in b:
        if x in ai:
            return True
    return False
```
A `StopIteration` exception is raised whenever `next` is called on an empty iterator.

```python
def contains(a, b):
    ai = iter(a)
    for x in b:
        ...
```

```text
>>> contains('strength', 'stent')
True
>>> contains('strength', 'rest')
False
>>> contains('strength', 'tenth')
True
```
Processing Iterators

A StopIteration exception is raised whenever next is called on an empty iterator

```python
>>> contains('strength', 'stent')
True
>>> contains('strength', 'rest')
False
>>> contains('strength', 'tenth')
True
```

```python
def contains(a, b):
    ai = iter(a)
    for x in b:
        while next(ai) != x:
            pass  # do nothing
```

A `StopIteration` exception is raised whenever `next` is called on an empty iterator.

```python
def contains(a, b):
    ai = iter(a)
    for x in b:
        while next(ai) != x:
            pass  # do nothing

>>> contains('strength', 'stent')
True
>>> contains('strength', 'rest')
False
>>> contains('strength', 'tenth')
True
```
A `StopIteration` exception is raised whenever `next` is called on an empty iterator.

```python
def contains(a, b):
    ai = iter(a)
    for x in b:
        while next(ai) != x:
            pass  # do nothing
```
Processing Iterators

A **StopIteration** exception is raised whenever `next` is called on an empty iterator.

```python
def contains(a, b):
    ai = iter(a)
    for x in b:
        while next(ai) != x:
            pass  # do nothing
```

```python
>>> contains('strength', 'stent')
True
>>> contains('strength', 'rest')
False
>>> contains('strength', 'tenth')
True
```
A `StopIteration` exception is raised whenever `next` is called on an empty iterator.

```python
>>> contains('strength', 'stent')
True
>>> contains('strength', 'rest')
False
>>> contains('strength', 'tenth')
True
```

```python
def contains(a, b):
    ai = iter(a)
    for x in b:
        while next(ai) != x:
            pass  # do nothing
```
Processing Iterators

A **StopIteration** exception is raised whenever **next** is called on an empty iterator.

```python
def contains(a, b):
    ai = iter(a)
    for x in b:
        while next(ai) != x:
            pass  # do nothing

>>> contains('strength', 'stent')
True
>>> contains('strength', 'rest')
False
>>> contains('strength', 'tenth')
True
```
A **StopIteration** exception is raised whenever `next` is called on an empty iterator.

```python
def contains(a, b):
    ai = iter(a)
    for x in b:
        while next(ai) != x:
            pass  # do nothing
```

```python
>>> contains('strength', 'stent')
True
>>> contains('strength', 'rest')
False
>>> contains('strength', 'tenth')
True
```
Processing Iterators

A `StopIteration` exception is raised whenever `next` is called on an empty iterator

```python
def contains(a, b):
    ai = iter(a)
    for x in b:
        while next(ai) != x:
            pass  # do nothing

>>> contains('strength', 'stent')
True
>>> contains('strength', 'rest')
False
>>> contains('strength', 'tenth')
True
```
Processing Iterators

A `StopIteration` exception is raised whenever `next` is called on an empty iterator.

```python
>>> contains('strength', 'stent')
True
>>> contains('strength', 'rest')
False
>>> contains('strength', 'tenth')
True
```

```python
def contains(a, b):
    ai = iter(a)
    for x in b:
        while next(ai) != x:
            pass # do nothing
```
Processing Iterators

A `StopIteration` exception is raised whenever `next` is called on an empty iterator.

```python
def contains(a, b):
    ai = iter(a)
    for x in b:
        while next(ai) != x:
            pass  # do nothing
    return True

>>> contains('strength', 'stent')
True
>>> contains('strength', 'rest')
False
>>> contains('strength', 'tenth')
True
```
A `StopIteration` exception is raised whenever `next` is called on an empty iterator.

```python
def contains(a, b):
    ai = iter(a)
    for x in b:
        try:
            while next(ai) != x:
                pass  # do nothing
        except StopIteration:
            return False
    return True
```

```python
>>> contains('strength', 'stent')
True
>>> contains('strength', 'rest')
False
>>> contains('strength', 'tenth')
True
```
Built-In Iterator Functions
Built-in Functions for Iteration

Many built-in Python sequence operations return iterators that compute results lazily.
Built-in Functions for Iteration

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

\texttt{map(func, iterable): \hspace{1cm} Iterate over func(x) for x in iterable}
Built-in Functions for Iteration

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

\[
\text{map}(\text{func}, \text{iterable}): \quad \text{Iterate over } \text{func}(x) \text{ for } x \text{ in } \text{iterable}
\]

\[
\text{filter}(\text{func}, \text{iterable}): \quad \text{Iterate over } x \text{ in } \text{iterable if } \text{func}(x)
\]
Built-in Functions for Iteration

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
Built-in Functions for Iteration

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.
Built-in Functions for Iteration

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 contents of an iterator, place the resulting elements into a container
Built-in Functions for Iteration

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 contents of an iterator, place the resulting elements into a container:

- `list(iterable)`: Create a list containing all `x` in `iterable`
Built-in Functions for Iteration

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 contents of an iterator, place the resulting elements into a container

- `list(iterable)`: Create a list containing all `x` in `iterable`
- `tuple(iterable)`: Create a tuple containing all `x` in `iterable`
Built-in Functions for Iteration

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 contents of an iterator, place the resulting elements into a container:

- `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`
Built-in Functions for Iteration

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 contents of an iterator, place the resulting elements into a container:

- `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)
Generators
Generators and Generator Functions
Generators and Generator Functions

```python
>>> def plus_minus(x):
...     yield x
...     yield -x
```
Generators and Generator Functions

```python
>>> def plus_minus(x):
...     yield x
...     yield -x

>>> t = plus_minus(3)
```
Generators and Generator Functions

```python
>>> def plus_minus(x):
...     yield x
...     yield -x

>>> t = plus_minus(3)
>>> next(t)
3
```
Generators and Generator Functions

```python
>>> def plus_minus(x):
...     yield x
...     yield -x

>>> t = plus_minus(3)
>>> next(t)
3
>>> next(t)
-3
```
Generators and Generator Functions

```python
>>> def plus_minus(x):
...     yield x
...     yield -x

>>> t = plus_minus(3)
>>> next(t)
3
>>> next(t)
-3
>>> t
<generator object plus_minus ...>
```
Generators and Generator Functions

>>> def plus_minus(x):
...     yield x
...     yield -x

>>> t = plus_minus(3)
>>> next(t)
3
>>> next(t)
-3
>>> t
<generator object plus_minus ...>

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* can *yield* multiple times.

```python
>>> def plus_minus(x):
...     yield x
...     yield -x

>>> t = plus_minus(3)
>>> next(t)
3
>>> next(t)
-3
>>> t
<generator object plus_minus ...>
```
A **generator function** is a function that **yields** values instead of **returning** them. A normal function **returns** once; a **generator function** can **yield** multiple times. A **generator** is an iterator created automatically by calling a **generator function**.
Generators and Generator Functions

>>> def plus_minus(x):
...     yield x
...     yield -x

>>> t = plus_minus(3)
>>> next(t)
3
>>> next(t)
-3
>>> t
<generator object plus_minus ...>

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

A normal function *returns* once; a **generator function** can *yield* multiple times.

A **generator** is an iterator created automatically by calling a **generator function**.

When a **generator function** is called, it returns a **generator** that iterates over its yields.
Generators and Generator Functions

```python
>>> def plus_minus(x):
...    yield x
...    yield -x

>>> t = plus_minus(3)
>>> next(t)
3
>>> next(t)
-3
>>> t
<generator object plus_minus ...>
```

A **generator function** is a function that **yields** values instead of **returning** them. A normal function **returns** once; a **generator function** can **yield** multiple times. A **generator** is an iterator created automatically by calling a **generator function**. When a **generator function** is called, it returns a **generator** that iterates over its yields.

(Demo)
Iterable User-Defined Classes
Iterable User-Defined Classes

The special method `__iter__` is called by the built-in `iter()` & should return an iterator.
Iterable User-Defined Classes

The special method `__iter__` is called by the built-in `iter()` & should return an iterator

```python
>>> list(Countdown(5))
[5, 4, 3, 2, 1]
```
Iterable User-Defined Classes

The special method `__iter__` is called by the built-in `iter()` & should return an iterator

```python
>>> list(Countdown(5))
[5, 4, 3, 2, 1]
>>> for x in Countdown(3):
...    print(x)
3
2
1
```
Iterable User-Defined Classes

The special method \_\_iter\_\_ is called by the built-in \_\_iter\_\_ function and should return an iterator.

```python
>>> list(Countdown(5))
[5, 4, 3, 2, 1]
>>> for x in Countdown(3):
...    print(x)
3
2
1
```

class Countdown:
def \_\_init\_\_(self, start):
    self.start = start
Iterable User-Defined Classes

The special method `__iter__` is called by the built-in `iter()` & should return an iterator.

```python
class Countdown:
    def __init__(self, start):
        self.start = start
    def __iter__(self):
        list(Countdown(5))
        [5, 4, 3, 2, 1]
        >>> for x in Countdown(3):
        ...     print(x)
        3
        2
        1
```
Iterable User-Defined Classes

The special method `__iter__` is called by the built-in `iter()` & should return an iterator.

```python
class Countdown:
    def __init__(self, start):
        self.start = start

    def __iter__(self):
        v = self.start
        while v > 0:
            yield v
            v -= 1

countdown = Countdown(5)
for x in countdown:
    print(x)
```

```python
countdown = Countdown(5)
list(countdown)
```

```python
countdown = Countdown(3)
for x in countdown:
    print(x)
```

```python
>>> list(Countdown(5))
[5, 4, 3, 2, 1]
>>> for x in Countdown(3):
    print(x)
3
2
1
```
Generators & Iterators
Generators can Yield from Iterators
Generators can Yield from Iterators

A `yield from` statement yields all values from an iterator or iterable (Python 3.3)
Generators can Yield from Iterators

A `yield from` statement yields all values from an iterator or iterable (Python 3.3)

```python
>>> list(a_then_b([[3, 4], [5, 6]]))
[3, 4, 5, 6]
```
Generators can Yield from Iterators

A `yield from` statement yields all values from an iterator or iterable (Python 3.3)

```python
>>> list(a_then_b([3, 4], [5, 6]))
[3, 4, 5, 6]

def a_then_b(a, b):
    for x in a:
        yield x
    for x in b:
        yield x
```
Generators can Yield from Iterators

A `yield from` statement yields all values from an iterator or iterable (Python 3.3)

```python
>>> list(a_then_b([3, 4], [5, 6]))
[3, 4, 5, 6]

def a_then_b(a, b):
    for x in a:
        yield x
    for x in b:
        yield x
```

```python
def a_then_b(a, b):
    yield from a
    yield from b
```
Generators can Yield from Iterators

A `yield from` statement yields all values from an iterator or iterable (Python 3.3)

```python
>>> list(a_then_b([3, 4], [5, 6]))
[3, 4, 5, 6]

def a_then_b(a, b):
    for x in a:
        yield x
    for x in b:
        yield x

>>> list(countdown(5))
[5, 4, 3, 2, 1]
```
Generators can Yield from Iterators

A `yield from` statement yields all values from an iterator or iterable (Python 3.3)

```python
def a_then_b(a, b):
    for x in a:
        yield x
    for x in b:
        yield x
def countdown(k):
    if k > 0:
        yield k
    yield from countdown(k-1)

>>> list(a_then_b([3, 4], [5, 6]))
[3, 4, 5, 6]
```

```python
def a_then_b(a, b):
    yield from a
    yield from b
```

```python
def countdown(k):
    yield from countdown(k-1)
```

```python
>>> list(countdown(5))
[5, 4, 3, 2, 1]
```
Generators can Yield from Iterators

A `yield from` statement yields all values from an iterator or iterable (Python 3.3)

```python
>>> list(a_then_b([3, 4], [5, 6]))
[3, 4, 5, 6]

def a_then_b(a, b):
    for x in a:
        yield x
    for x in b:
        yield x

def countdown(k):
    if k > 0:
        yield k
        yield from countdown(k-1)

>>> list(countdown(5))
[5, 4, 3, 2, 1]
```

(Demo)