Iterators
Announcements
Iterators
Iterators
Iterators

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

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

`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 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]
```
Iterators

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

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

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

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

A container can provide an iterator that provides access to its elements in 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)
```
Iterators

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

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

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

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

A container can provide an iterator that provides access to its elements in 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
>>> u = iter(s)
>>> next(u)
3
```
Iterators

A container can provide an iterator that provides access to its elements in 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
>>> u = iter(s)
>>> next(u)
3
>>> next(t)
5
```
Iterators

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

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

\textbf{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(t)
5
>>> next(u)
4
```
Iterators

A container can provide an iterator that provides access to its elements in 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
>>> next(t)
4
>>> u = iter(s)
>>> next(u)
3
>>> next(t)
5
>>> next(u)
4
```
Dictionary Iteration
Views of a Dictionary
Views of a Dictionary

An *iterable* value is any value that can be passed to `iter` to produce an iterator.

An *iterator* is returned from `iter` and can be passed to `next`; all iterators are mutable.
Views of a Dictionary

An *iterable* value is any value that can be passed to `iter` to produce an iterator.

An *iterator* is returned from `iter` and can be passed to `next`; all iterators are mutable.

A dictionary, its keys, its values, and its items are all iterable values.
Views of a Dictionary

An *iterable* value is any value that can be passed to `iter` to produce an iterator.

An *iterator* is returned from `iter` and can be passed to `next`; all iterators are mutable.

A dictionary, its keys, its values, and its items are all iterable values.

- The order of items in a dictionary is the order in which they were added (Python 3.6+).
Views of a Dictionary

An *iterable* value is any value that can be passed to `iter` to produce an iterator.

An *iterator* is returned from `iter` and can be passed to `next`; all iterators are mutable.

A dictionary, its keys, its values, and its items are all iterable values:
- The order of items in a dictionary is the order in which they were added (Python 3.6+)
- Historically, items appeared in an arbitrary order (Python 3.5 and earlier)
Views of a Dictionary

An iterable value is any value that can be passed to `iter` to produce an iterator.

An iterator is returned from `iter` and can be passed to `next`; all iterators are mutable.

A dictionary, its keys, its values, and its items are all iterable values:

- The order of items in a dictionary is the order in which they were added (Python 3.6+).
- Historically, items appeared in an arbitrary order (Python 3.5 and earlier).

```python
>>> d = {'one': 1, 'two': 2, 'three': 3}
```
Views of a Dictionary

An iterable value is any value that can be passed to `iter` to produce an iterator.

An iterator is returned from `iter` and can be passed to `next`; all iterators are mutable.

A dictionary, its keys, its values, and its items are all iterable values:

- The order of items in a dictionary is the order in which they were added (Python 3.6+).
- Historically, items appeared in an arbitrary order (Python 3.5 and earlier).

```python
>>> d = {'one': 1, 'two': 2, 'three': 3}
>>> d['zero'] = 0
```
Views of a Dictionary

An *iterable* value is any value that can be passed to `iter` to produce an iterator.

An *iterator* is returned from `iter` and can be passed to `next`; all iterators are mutable.

A dictionary, its keys, its values, and its items are all iterable values:

- The order of items in a dictionary is the order in which they were added (Python 3.6+).
- Historically, items appeared in an arbitrary order (Python 3.5 and earlier).

```python
>>> d = {'one': 1, 'two': 2, 'three': 3}
>>> d['zero'] = 0
>>> k = iter(d.keys())  # or iter(d)
>>> next(k)
```
Views of a Dictionary

An *iterable* value is any value that can be passed to `iter` to produce an iterator.

An *iterator* is returned from `iter` and can be passed to `next`; all iterators are mutable.

A dictionary, its keys, its values, and its items are all iterable values:

- The order of items in a dictionary is the order in which they were added (Python 3.6+).
- Historically, items appeared in an arbitrary order (Python 3.5 and earlier).

```python
>>> d = {'one': 1, 'two': 2, 'three': 3}
>>> d['zero'] = 0
>>> k = iter(d.keys())  # or iter(d)
>>> next(k)
'one'
>>> next(k)
```
Views of a Dictionary

An *iterable* value is any value that can be passed to *iter* to produce an iterator.

An *iterator* is returned from *iter* and can be passed to *next*; all iterators are mutable.

A dictionary, its keys, its values, and its items are all iterable values.

- The order of items in a dictionary is the order in which they were added (Python 3.6+).
- Historically, items appeared in an arbitrary order (Python 3.5 and earlier).

```python
>>> d = {'one': 1, 'two': 2, 'three': 3}
>>> d['zero'] = 0
>>> k = iter(d.keys())  # or iter(d)
>>> next(k)
'one'
>>> next(k)
'two'
>>> next(k)
'three'
>>> next(k)
'zero'
```
Views of a Dictionary

An *iterable* value is any value that can be passed to `iter` to produce an iterator.

An *iterator* is returned from `iter` and can be passed to `next`; all iterators are mutable.

A dictionary, its keys, its values, and its items are all iterable values:

- The order of items in a dictionary is the order in which they were added (Python 3.6+).
- Historically, items appeared in an arbitrary order (Python 3.5 and earlier).

```python
>>> d = {'one': 1, 'two': 2, 'three': 3}
>>> d['zero'] = 0
>>> k = iter(d.keys())  # or iter(d)
>>> next(k)
'one'
>>> next(k)
'two'
>>> next(k)
'three'
>>> next(k)
'zero'
>>> v = iter(d.values())
>>> next(v)
1
>>> next(v)
2
>>> next(v)
3
>>> next(v)
0
```
Views of a Dictionary

An iterable value is any value that can be passed to `iter` to produce an iterator.

An iterator is returned from `iter` and can be passed to `next`; all iterators are mutable.

A dictionary, its keys, its values, and its items are all iterable values.

- The order of items in a dictionary is the order in which they were added (Python 3.6+).
- Historically, items appeared in an arbitrary order (Python 3.5 and earlier).

```python
>>> d = {'one': 1, 'two': 2, 'three': 3}
>>> d['zero'] = 0
>>> k = iter(d.keys())  # or iter(d)
>>> next(k)
'one'
>>> next(k)
'two'
>>> next(k)
'three'
>>> next(k)
'zero'
```
Views of a Dictionary

An iterable value is any value that can be passed to `iter` to produce an iterator

An iterator is returned from `iter` and can be passed to `next`; all iterators are mutable

A dictionary, its keys, its values, and its items are all iterable values

- The order of items in a dictionary is the order in which they were added (Python 3.6+)
- Historically, items appeared in an arbitrary order (Python 3.5 and earlier)

```python
>>> d = {'one': 1, 'two': 2, 'three': 3}
>>> d['zero'] = 0
>>> k = iter(d.keys())  # or iter(d)
>>> next(k)
'two'
>>> next(k)
'three'
>>> next(k)
'zero'
```
Views of a Dictionary

An *iterable* value is any value that can be passed to `iter` to produce an iterator.

An *iterator* is returned from `iter` and can be passed to `next`; all iterators are mutable.

A dictionary, its keys, its values, and its items are all iterable values.

- The order of items in a dictionary is the order in which they were added (Python 3.6+).
- Historically, items appeared in an arbitrary order (Python 3.5 and earlier).

```python
>>> d = {'one': 1, 'two': 2, 'three': 3}
>>> d['zero'] = 0
>>> k = iter(d.keys())  # or iter(d)
>>> next(k)
    'one'
>>> next(k)
    'two'
>>> next(k)
    'three'
>>> next(k)
    'zero'
```
Views of a Dictionary

An *iterable* value is any value that can be passed to `iter` to produce an iterator.

An *iterator* is returned from `iter` and can be passed to `next`; all iterators are mutable.

A dictionary, its keys, its values, and its items are all iterable values:

- The order of items in a dictionary is the order in which they were added (Python 3.6+).
- Historically, items appeared in an arbitrary order (Python 3.5 and earlier).

```python
>>> d = {'one': 1, 'two': 2, 'three': 3}
>>> d['zero'] = 0
>>> k = iter(d.keys())  # or iter(d)
>>> next(k) 'one'
>>> next(k) 'two'
>>> next(k) 'three'
>>> next(k) 'zero'
```

```python
>>> v = iter(d.values())
>>> next(v) 1
>>> next(v) 2
>>> next(v) 3
>>> next(v) 0
```
Views of a Dictionary

An *iterable* value is any value that can be passed to `iter` to produce an iterator.

An *iterator* is returned from `iter` and can be passed to `next`; all iterators are mutable.

A dictionary, its keys, its values, and its items are all iterable values:

- The order of items in a dictionary is the order in which they were added (Python 3.6+).
- Historically, items appeared in an arbitrary order (Python 3.5 and earlier).

```python
>>> d = {'one': 1, 'two': 2, 'three': 3}
>>> d['zero'] = 0
>>> k = iter(d.keys())  # or iter(d)
>>> next(k)
'one'
>>> next(k)
'two'
>>> next(k)
'three'
>>> next(k)
'zero'
```

```python
>>> v = iter(d.values())
>>> next(v)
1
>>> next(v)
2
>>> next(v)
3
>>> next(v)
0
>>> i = iter(d.items())
>>> next(i)
('one', 1)
```
Views of a Dictionary

An iterable value is any value that can be passed to `iter` to produce an iterator.

An iterator is returned from `iter` and can be passed to `next`; all iterators are mutable.

A dictionary, its keys, its values, and its items are all iterable values.

- The order of items in a dictionary is the order in which they were added (Python 3.6+).
- Historically, items appeared in an arbitrary order (Python 3.5 and earlier).

```python
>>> d = {'one': 1, 'two': 2, 'three': 3}
>>> d['zero'] = 0
>>> k = iter(d.keys())  # or iter(d)
>>> next(k)
'one'
>>> next(k)
'two'
>>> next(k)
'three'
>>> next(k)
'zero'
```

```python
>>> v = iter(d.values())
>>> next(v)
1
>>> next(v)
2
>>> next(v)
3
>>> next(v)
0
```

```python
>>> i = iter(d.items())
>>> next(i)
('one', 1)
>>> next(i)
('two', 2)
```
Views of a Dictionary

An iterable value is any value that can be passed to iter to produce an iterator.

An iterator is returned from iter and can be passed to next; all iterators are mutable.

A dictionary, its keys, its values, and its items are all iterable values.

- The order of items in a dictionary is the order in which they were added (Python 3.6+).
- Historically, items appeared in an arbitrary order (Python 3.5 and earlier).

```python
>>> d = {'one': 1, 'two': 2, 'three': 3}
>>> d['zero'] = 0
>>> k = iter(d.keys())  # or iter(d)
>>> next(k)
'one'
>>> next(k)
'two'
>>> next(k)
'three'
>>> next(k)
'zero'
```

```python
>>> v = iter(d.values())
>>> next(v)
1
>>> next(v)
2
>>> next(v)
3
>>> next(v)
0
```

```python
>>> i = iter(d.items())
>>> next(i)
('one', 1)
>>> next(i)
('two', 2)
>>> next(i)
('three', 3)
>>> next(i)
('zero', 0)
```
Views of a Dictionary

An iterable value is any value that can be passed to `iter` to produce an iterator.

An iterator is returned from `iter` and can be passed to `next`; all iterators are mutable.

A dictionary, its keys, its values, and its items are all iterable values:

- The order of items in a dictionary is the order in which they were added (Python 3.6+);
- Historically, items appeared in an arbitrary order (Python 3.5 and earlier).

```python
>>> d = {'one': 1, 'two': 2, 'three': 3}
>>> d['zero'] = 0
>>> k = iter(d.keys())  # or iter(d)
>>> next(k)
'one'
>>> next(k)
'two'
>>> next(k)
'three'
>>> next(k)
'zero'
```

```python
>>> v = iter(d.values())
>>> next(v)
1
>>> next(v)
2
>>> next(v)
3
>>> next(v)
0
```

```python
>>> i = iter(d.items())
>>> next(i)
('one', 1)
>>> next(i)
('two', 2)
>>> next(i)
('three', 3)
>>> next(i)
('zero', 0)
```

(Demo)
For Statements

(Demo)
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

```python
map(func, iterable): 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

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

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

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

>>> 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

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

```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.

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

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

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

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

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

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

```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
    yield from b
    for x in b:
        yield x
```

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

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

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]

demo = [3, 4, 5, 6]

demo

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