Iterators
Announcements
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

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

\textbf{next}(\texttt{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

- **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)
```
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

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

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

```
>>> 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
```
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)
```
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
globals()['s'] = [3, 4, 5]
globals()['t'] = iter(s)
globals()['u'] = iter(s)
>>> next(t)
3
>>> next(t)
4
>>> next(u)
3
>>> next(t)
5
```
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
>>> 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

(Demo)
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)
>>> v = iter(d.values())
>>> 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)
>>> v = iter(d.values())
>>> 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
```
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'

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

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

>>> 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'

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

>>> 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

\begin{itemize}
\item \textbf{map(func, iterable):} Iterate over \texttt{func(x)} for \texttt{x} in \texttt{iterable}
\item \textbf{filter(func, iterable):} Iterate over \texttt{x} in \texttt{iterable} if \texttt{func(x)}
\end{itemize}
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)
Zip
The Zip Function
The Zip Function

The built-in `zip` function returns an iterator over co-indexed tuples.
The Zip Function

The built-in `zip` function returns an iterator over co-indexed tuples.

```python
>>> list(zip([1, 2], [3, 4]))
[(1, 3), (2, 4)]
```
The built-in `zip` function returns an iterator over co-indexed tuples.

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

If one iterable is longer than the other, `zip` only iterates over matches and skips extras.
**The Zip Function**

The built-in `zip` function returns an iterator over co-indexed tuples.

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

If one iterable is longer than the other, `zip` only iterates over matches and skips extras.

```python
>>> list(zip([1, 2], [3, 4, 5]))
[(1, 3), (2, 4)]
```
The Zip Function

The built-in `zip` function returns an iterator over co-indexed tuples.

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

If one iterable is longer than the other, `zip` only iterates over matches and skips extras.

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

More than two iterables can be passed to `zip`. 
The Zip Function

The built-in `zip` function returns an iterator over co-indexed tuples.

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

If one iterable is longer than the other, `zip` only iterates over matches and skips extras.

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

More than two iterables can be passed to `zip`.

```python
>>> list(zip([1, 2], [3, 4, 5], [6, 7]))
[(1, 3, 6), (2, 4, 7)]
```
The Zip Function

The built-in `zip` function returns an iterator over co-indexed tuples.

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

If one iterable is longer than the other, `zip` only iterates over matches and skips extras.

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

More than two iterables can be passed to `zip`.

```python
>>> list(zip([1, 2], [3, 4, 5], [6, 7]))
[(1, 3, 6), (2, 4, 7)]
```

Implement `palindrome`, which returns whether s is the same forward and backward.
The Zip Function

The built-in `zip` function returns an iterator over co-indexed tuples.

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

If one iterable is longer than the other, `zip` only iterates over matches and skips extras.

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

More than two iterables can be passed to `zip`.

```python
>>> list(zip([1, 2], [3, 4, 5], [6, 7]))
[(1, 3, 6), (2, 4, 7)]
```

Implement `palindrome`, which returns whether `s` is the same forward and backward.

```python
>>> palindrome([3, 1, 4, 1, 3])
True
>>> palindrome([3, 1, 4, 1, 5])
False
```
The Zip Function

The built-in `zip` function returns an iterator over co-indexed tuples.

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

If one iterable is longer than the other, `zip` only iterates over matches and skips extras.

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

More than two iterables can be passed to `zip`.

```python
>>> list(zip([1, 2], [3, 4, 5], [6, 7]))
[(1, 3, 6), (2, 4, 7)]
```

Implement `palindrome`, which returns whether `s` is the same forward and backward.

```python
>>> palindrome([3, 1, 4, 1, 3])
True
>>> palindrome([3, 1, 4, 1, 5])
False
>>> palindrome('seveneves')
True
>>> palindrome('seven eves')
False
```
Using Iterators
Reasons for Using Iterators
Reasons for Using Iterators

Code that processes an iterator (via `next`) or iterable (via `for` or `iter`) makes few assumptions about the data itself.
**Reasons for Using Iterators**

Code that processes an iterator (via `next`) or iterable (via `for` or `iter`) makes few assumptions about the data itself.

- Changing the data representation from a `list` to a `tuple`, `map object`, or `dict_keys` doesn't require rewriting code.
Reasons for Using Iterators

Code that processes an iterator (via `next`) or iterable (via `for` or `iter`) makes few assumptions about the data itself.

- Changing the data representation from a `list` to a `tuple`, `map object`, or `dict_keys` doesn't require rewriting code.

- Others are more likely to be able to use your code on their data.
Reasons for Using Iterators

Code that processes an iterator (via `next`) or iterable (via `for` or `iter`) makes few assumptions about the data itself.

- Changing the data representation from a `list` to a `tuple`, `map object`, or `dict_keys` doesn't require rewriting code.

- Others are more likely to be able to use your code on their data.

An iterator bundles together a sequence and a position within that sequence as one object.
Reasons for Using Iterators

Code that processes an iterator (via `next`) or iterable (via `for` or `iter`) makes few assumptions about the data itself.

- Changing the data representation from a `list` to a `tuple`, `map object`, or `dict_keys` doesn't require rewriting code.
- Others are more likely to be able to use your code on their data.

An iterator bundles together a sequence and a position within that sequence as one object.

- Passing that object to another function always retains the position.
Reasons for Using Iterators

Code that processes an iterator (via `next`) or iterable (via `for` or `iter`) makes few assumptions about the data itself.

- Changing the data representation from a `list` to a `tuple`, `map object`, or `dict_keys` doesn't require rewriting code.
- Others are more likely to be able to use your code on their data.

An iterator bundles together a sequence and a position within that sequence as one object.

- Passing that object to another function always retains the position.
- Useful for ensuring that each element of a sequence is processed only once.
Reasons for Using Iterators

Code that processes an iterator (via `next`) or iterable (via `for` or `iter`) makes few assumptions about the data itself.

- Changing the data representation from a `list` to a `tuple`, `map object`, or `dict_keys` doesn't require rewriting code.
- Others are more likely to be able to use your code on their data.

An iterator bundles together a sequence and a position within that sequence as one object.

- Passing that object to another function always retains the position.
- Useful for ensuring that each element of a sequence is processed only once.
- Limits the operations that can be performed on the sequence to only requesting `next`. 
Example: Casino Blackjack

Player:

Dealer:
Example: Casino Blackjack

Player:

Dealer:
Example: Casino Blackjack

**Player:**

![Player's cards](image)

**Dealer:**

![Dealer's cards](image)
Example: Casino Blackjack

Player:

Dealer:
Example: Casino Blackjack

**Player:**

```
7
\spades

2
\hearts
```

**Dealer:**

```
J
\spades

\clubs
\clubs
```


Example: Casino Blackjack

**Player:**

- 7 of Spades
- 9 of Spades
- 5 of Spades
- 7 of Hearts
- 6 of Hearts

**Dealer:**

- Jack of Spades
- Ace of Spades
Example: Casino Blackjack

Player:

Dealer:
Example: Casino Blackjack

Player:

Dealer:
Example: Casino Blackjack

Player:

Dealer:
Example: Casino Blackjack

Player:

Dealer:

(Demo)