

# Iterators

---

# Announcements

# Iterators

## 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

---

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)

```
>>> 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

<code>map(func, iterable):</code>	Iterate over <code>func(x)</code> for <code>x</code> in <code>iterable</code>
<code>filter(func, iterable):</code>	Iterate over <code>x</code> in <code>iterable</code> if <code>func(x)</code>
<code>zip(first_iter, second_iter):</code>	Iterate over co-indexed <code>(x, y)</code> pairs
<code>reversed(sequence):</code>	Iterate over <code>x</code> in a sequence in reverse order

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

<code>list(iterable):</code>	Create a list containing all <code>x</code> in <code>iterable</code>
<code>tuple(iterable):</code>	Create a tuple containing all <code>x</code> in <code>iterable</code>
<code>sorted(iterable):</code>	Create a sorted list containing <code>x</code> in <code>iterable</code>

(Demo)

Zip

## The Zip Function

---

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

```
>>> 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.

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

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

```
>>> 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.

```
>>> palindrome([3, 1, 4, 1, 3])           >>> palindrome('seveneves')
True                                       True
>>> palindrome([3, 1, 4, 1, 5])           >>> palindrome('seven eves')
False                                     False
```

## 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.

- 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.

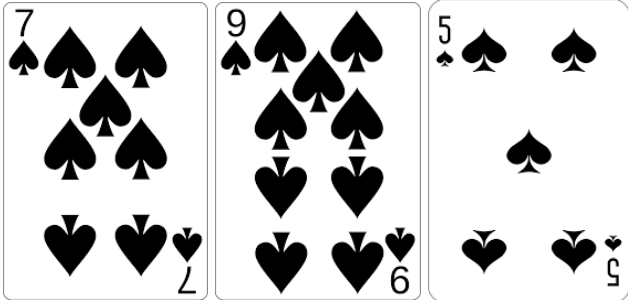
An iterator bundles together a sequence and a position within that sequence as one object.

- 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:



(Demo)

Dealer:

