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(u)
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).

```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'
```

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.

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

For Statements

Built-In Iterator Functions

(Demo)
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)

The Zip Function

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

- `list(zip([1, 2], [3, 4]))` evaluates to `[[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]))` evaluates to `[[1, 3], [2, 4]]`

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

- `list(zip([1, 2], [3, 4, 5], [6, 7]))` evaluates to `[[1, 3, 6], [2, 4, 7]]`

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

- `palindrome([3, 1, 4, 1, 3])` evaluates to `True`
- `palindrome([3, 1, 4, 1, 3])` evaluates to `True`
- `palindrome('seven eves')` evaluates to `False`
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

![Card images for Player and Dealer]