**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
>>> a = [1, 2, 3]
>>> x = iter(a)
>>> x
<iterobject>
>>> next(x)  # x = iter(a)
1
>>> x = iter(a)
>>> next(x)  # x = iter(a)
2
>>> x = iter(a)
>>> next(x)  # x = iter(a)
3
```

**Dictionary Iteration**

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.

- 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}
>>> 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)
```

**For Statements**

**Built-In Iterator Functions**
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 non-sorted list containing `x` in `iterable`

Generators

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)

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

>>> t = plus_minus(3)

>>> next(t)
3

>>> next(t)
-3

>>> next(t)

>>> generator object plus_minus ...>
```

Generators and Generator Functions

```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]

>>> 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

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

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

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

Generators & Iterators

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)