

# Lecture 17. Iterators

## Iterators

### Iterators



A container can provide an iterator that provides access to its elements in some 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]
>>> u = iter(s)
>>> t = iter(s)
>>> next(t)
3
>>> next(t)
5
4
>>> next(u)
4
```

## Dictionary Iterators

### 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)
```



better to see the Iterators as pointers, so record the position and moves, so changing the size of the list/dic is not allowed after the iterators is created, since the iterator will not be allowed to access anymore

```
>>> ri = iter(r)
>>> ri
<range_iterator object at 0x10d247a20>
>>> next(ri)
3
>>> for i in ri:
...     print(i)
...
4
5
>>> 
```



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

(Demo)

```
~/lec$ python3 -i ex.py
>>> m = map(double, range(3, 7))
>>> f = lambda y: y >= 10
>>> t = filter(f, m)
>>> next(t)
** 3 => 6 **
** 4 => 8 **
** 5 => 10 **
10
>>> next(t)
** 6 => 12 **
12
>>> list(t)
[]
>>> list(filter(f, map(double, range(3, 7))))
** 3 => 6 **
** 4 => 8 **
** 5 => 10 **
** 6 => 12 **
[10, 12]
>>>
```

  

```
def double(x):
    print('**', x, '=>', 2*x, '**')
    return 2*x
```

"ex.py" 3L, 66C written

```
>>> t = [1, 2, 3, 2, 1]
>>> t
[1, 2, 3, 2, 1]
>>> reversed(t)
<list_reverseiterator object at 0x101b7ad30>
>>> reversed(t) == t
False
>>> list(reversed(t))
[1, 2, 3, 2, 1]
>>> list(reversed(t)) == t
True
```

```
def double(x):
    print('**', x, '=>', 2*x, '**')
    return 2*x
```

## Generators

## 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**

```
>>> list(evens(1, 10))
[2, 4, 6, 8]
>>> t = evens(2, 10)
>>> next(t)
2
>>> next(t)
4
>>> 
```

```
def evens(start, end):
    even = start + (start % 2)
    while even < end:
        yield even
        even += 2
~
```



**yield** is kinda like a record point here, which return when executed, and will run at the end of the **yield** sentence next time it was called.

## Generators & Iterators

## Generators can Yield from Iterators



A `yield from` statement yields all values from an iterator or iterable (Python 3.3)

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

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

```
~/lec$ python3 -i ex.py
>>> prefixes('both')
<generator object prefixes at 0x102379f30>
>>> list(prefixes('both'))
['b', 'bo', 'bot', 'both']
>>> ^D
~/lec$ python3 -i ex.py
>>> substrings('tops')
<generator object substrings at 0x101379f30>
>>> list(substrings('tops'))
['t', 'to', 'top', 'tops', 'o', 'op', 'ops', 'p', 'ps', 's']
>>> █
```

```
def countdown(k):
    if k > 0:
        yield k
        yield from countdown(k-1)
    else:
        yield 'Blast off'

def prefixes(s):
    if s:
        yield from prefixes(s[:-1])
        yield s

def substrings(s):
    if s:
        yield from prefixes(s)
        yield from substrings(s[1:])
```

