Many data sets can be processed sequentially:
• The set of all Twitter posts
• Votes cast in an election
• Sensor readings of an airplane
• The positive integers: 1, 2, 3, ...

However, the sequence interface we used before does not always apply
• A sequence has a finite, known length
• A sequence allows element selection for any element

Some important ideas in big data processing:
• Implicit representations of streams of sequential data
• Declarative programming languages to manipulate and transform data

Iterators
A container can provide an iterator that provides access to its elements in some order

```python
>>> s = [3, 4, 5]
>>> a = iter(s)
>>> b = iter(s)
>>> next(a)
3
>>> next(b)
3
```

Keys and values are iterated over in an arbitrary order which is non-random, varies across Python implementations, and depends on the dictionary's history of insertions and deletions. If keys, values and item views are iterated over with no intervening modifications to the dictionary, the order of items will directly correspond.

```python
>>> d = {'one': 1, 'two': 2, 'three': 3}
>>> k = iter(d)
>>> next(k)
'one'
```

For Statements
```
for <name> in <expression>:
   <suite>
```
1. Evaluate the header `expression`, which must evaluate to an iterable object
2. For each element in that sequence, in order:
   A. Bind `<name>` to that element in the first frame of the current environment
   B. Execute `<suite>`

When executing a `for` statement, `iter` returns an iterator and `next` provides each item:

```python
>>> counts = [1, 2, 3]
>>> for item in counts:
    print(item)
1
2
3
>>> try:
    while True:
        item = next(items)
        print(item)
except StopIteration:
    pass  # Do nothing
1
2
3
```
Processing Iterators

A `StopIteration` exception is raised whenever `next` is called on an empty iterator

```python
>>> contains('strength', 'stent')
True
>>> contains('strength', 'rest')
False
>>> contains('strength', 'tenth')
True
```

```python
def contains(a, b):
    ai = iter(a)
    for x in b:
        try:
            while next(ai) != x:
                pass  # do nothing
        except StopIteration:
            return False
    return True
```

Built-In Iterator Functions

Many built-in Python sequence operations return iterators that compute results lazily:

- `map(func, iterable)`
- `filter(func, iterable)`
- `zip(first_iter, second_iter)`
- `reversed(sequence)`

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

- `list(iterable)`
- `tuple(iterable)`
- `sorted(iterable)`

(Demo)

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)

Iterables & Generators

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

```python
def a_then_b(a, b):
    yield from a
    yield from b
```

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

Built-in Functions for Iteration

The special method `__iter__` is called by the built-in `iter()` & should return an iterator

```python
class Countdown:
    def __init__(self, start):
        self.start = start
    def __iter__(self):
        v = self.start
        while v > 0:
            yield v
            v -= 1
```

```python
def countdown(start):
    if start > 0:
        yield start
        for x in countdown(start - 1):
            yield x
```

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

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

```
>>> list(x.then_b([1, 2, 1]), [1, 2, 3])
[1, 2, 1, 1, 2, 3]
```

Generators & Iterators

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)

(End of document)