61A Lecture 30
Data Processing
Data Processing

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
- Distributed computing
Iterators
Iterators

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

```python
iter(iterable): Return an iterator over the elements of an iterable value

next(iterator): Return the next element in an iterator
```

Iterators are always ordered, even if the container that produced them is not

```python
>>> s = [3, 4, 5]
>>> u = iter(s)
>>> next(u)
3
>>> next(t)
3
>>> next(t)
5
>>> next(t)
4
>>> u = iter(s)
>>> next(u)
3
>>> next(u)
4
```
For Statements
The For Statement

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 the <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
>>> counts = [1, 2, 3]
>>> items = iter(counts)
>>> try:
    while True:
        item = next(items)
        print(item)
    except StopIteration:
        pass  # Do nothing
1
2
3
```
A `StopIteration` exception is raised whenever `next` is called on an empty iterator.

```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
```

```text
>>> contains('strength', 'stent')
True
>>> contains('strength', 'rest')
False
>>> contains('strength', 'tenth')
True
```
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 sorted list containing `x` in `iterable` (Demo)
Generators
Generators and Generator Functions

```python
>>> 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.

(Demo)
Iterable User-Defined Classes

The special method \_\_iter\_\_ is called by the built-in \texttt{iter()} \& should return an iterator

\begin{verbatim}
>>> list(Countdown(5))
[5, 4, 3, 2, 1]
>>> for x in Countdown(3):
...     print(x)
3
2
1

class Countdown:
    def \_\_init\_\_(self, start):
        self.start = start
    def \_\_iter\_\_(self):
        v = self.start
        while v > 0:
            yield v
            v -= 1
\end{verbatim}
Generators & Iterators
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

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

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

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

(Demo)