Data Processing

Processing Sequential Data

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

Implicit Sequences

An implicit sequence is a representation of sequential data that does not explicitly store each element

Example: The built-in `range` class represents consecutive integers
- The range is represented by two values: start and end
- The length and elements are computed on demand
- Constant space for arbitrarily long sequences

```
..., -5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5, ...
```

Iterators

The Iterator Interface

An iterator is an object that can provide the next element of a sequence

The `_next_` method of an iterator returns the next element

The built-in `next` function invokes the `_next_` method on its argument

If there is no next element, then the `_next_` method of an iterator should raise a `StopIteration` exception

```
..., -5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5, ...

iter(range(-2, 2))  # invokes __iter__ on its argument
next(<range_iterator object>)  # returns
```

Implicit Sequences

An implicit sequence is a representation of sequential data that does not explicitly store each element

Example: The built-in `range` class represents consecutive integers
- The range is represented by two values: start and end
- The length and elements are computed on demand
- Constant space for arbitrarily long sequences

```
..., -5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5, ...
```
Iterables and Iterators

**Iterator:** Mutable object that tracks a position in a sequence, advancing on `__next__`

**Iterable:** Represents a sequence and returns a new iterator on `__iter__`

<table>
<thead>
<tr>
<th>LetterIter is an iterator:</th>
<th>LetterIter('a', 'e') ▼</th>
</tr>
</thead>
<tbody>
<tr>
<td>Letters is iterable:</td>
<td>Letters(['a', 'e'])</td>
</tr>
</tbody>
</table>

10 (Demo)

Iterators from Built-in Functions

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 results, place the resulting elements in a sequence

- `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`

For Statements

```python
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)
...   try:
...     while True:
...       item = item.__next__()  # Do nothing
...   except StopIteration:
...     pass
1
2
3
```

Generators and Generator Functions

A generator function is a function that yields values instead of returning them

A normal function returns once; a generator function yields multiple times

A generator is an iterator, created by a generator function

When a generator function is called, it returns a generator that iterates over yields

```python
>> def letter_generator(next_letter, end):
...   while next_letter < end:
...     yield next_letter
...     next_letter = chr(ord(next_letter)+1)
>> x = letter_generator('a', 'e')
>> next(x)
'a'
>> x
'B'
```

(Demo)