61A Lecture 30
 Wednesday, November 12

Announcements

- Homework 8 due Wednesday 11/12 @ 11:59pm
- Project 4 due Thursday 11/21 @ 11:59pm
- Early submission point #1: Questions 1-6 by Friday 11/14 @ 11:59pm
- Early submission point #2: Questions 1-16 by Tuesday 11/18 @ 11:59pm
- Early submission point #3: Submit by Thursday 11/20 @ 11:59pm
- Homework 9 combined with Homework 10 (6 pts), due Wednesday 11/26 @ 11:59pm
- Guest lecture Friday 11/14 by Andrew Huang

Survey Results

- Midterm 2 was difficult and discouraging
- Too many adjacent deadlines
- Too hard to get help in office hours and homework/project parties
- Better guidance on how to solve problems
- Pace, information overload, and extra credit

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 and parallel 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

\[ ..., -5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5, ... \]
\[ \text{range(-2, 2)} \]

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, ...
```

```
invokes __iter__
```

```
on its argument
```

```
returns next(range_iterator object)
```

Iterables and Iterators

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

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

```
LetterIter is an iterator: LetterIter('a', 'e') ▼
LetterIter('a', 'e') ▼
```

```
Letters is iterable: Letters('a', 'e') 'a' 'b' 'c' 'd'
```

Built-in Iterators

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

```
lis(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

```
for <name> in <expression>:
  <suite>
```

1. Evaluate the header `<expression>`, which yields 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:

```
>>> counts = [1, 2, 3]
>>> for item in counts:
...     print(item)
1
2
3
```

```
for item in counts:
...     item = item.__next__()
```

```
>>> try:
...     while True:
...         item = items.__next__()
...         print(item)
...     except StopIteration:
...         pass  # Do nothing
```

Generator Functions
Generators and Generator Functions

A generator is an iterator, created by a generator function

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

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

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

```python
def letters_generator(next_letter, end):
    while next_letter < end:
        yield next_letter
        next_letter = chr(ord(next_letter)+1)

>>> for letter in letters_generator('a', 'e'):
    print(letter)
    a
    b
    c
d
(Demo)
```