Our Sequence Abstraction

Recall our previous sequence interface:
• A sequence has a finite, known length
• A sequence allows element selection for any element

In most cases, satisfying the sequence interface requires storing the entire sequence in a computer’s memory

Problems?
• Infinite sequences- primes, positive integers
• Really large sequences- all Twitter posts, votes in a presidential election

The Sequence of Primes

Think about the sequence of prime numbers:
• What’s the first one?
• The next one?
• The next one?
• How about the next two?
• How about the 105th prime?
  • Our sequence abstraction would give an instant answer

Implicit Sequences

• We compute each of the elements on demand.
• Don’t explicitly store each element
• Called an implicit sequence.

A Python Example

Example: The range class represents a regular sequence of integers
• The range is represented by three values: start, end, and step
• The length and elements are computed on demand
• Constant space for arbitrarily long sequences

\[
\text{length} = \max \left( \left\lfloor \frac{\text{end} - \text{start}}{\text{step}} \right\rfloor, 0 \right)
\]

\[
\text{elem}(k) = \text{start} + k \cdot \text{step} \quad \text{(for } k \in [0, \text{length}])
\]
A Range Class

class Range(object):
    def __init__(self, start, end=None, step=1):
        if end is None:
            start, end = 0, start
        self.start = start
        self.end = end
        self.step = step
    def __len__(self):
        return max(0, ceil((self.end - self.start) / self.step))
    def __getitem__(self, k):
        if k < 0:
            k = len(self) + k
        if k < 0 or k >= len(self):
            raise IndexError('index out of range')
        return self.start + k * self.step

The Iterator Interface

An iterator is an object that can provide the next element of a (possibly implicit) sequence.
The iterator interface has two methods:
* __iter__(self) returns an equivalent iterator.
* __next__(self) returns the next element in the sequence
* Next prime, etc.
* If no next, raises StopIteration exception.

Rangelter

class Rangelter(object):
    def __init__(self, start, end, step):
        self.current = start
        self.end = end
        self.step = step
        self.sign = 1 if step > 0 else -1
    def __next__(self):
        if self.current * self.sign >= self.end * self.sign:
            raise StopIteration
        result = self.current
        self.current += self.step
        return result
    def __iter__(self):
        return self

Fibonacci

class FibIter(object):
    def __init__(self):
        self.prev = -1
        self.current = 1
    def __next__(self):
        self.prev, self.current = (self.current,
                                   self.prev + self.current)
        return self.current
    def __iter__(self):
        return self

The For Statement

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

Generators and Generator Functions

Generators:
* An iterator backed by a function, called a generator function.

Generator Functions:
* A function that returns a generator.
* Can tell by looking for the yield keyword.
* Another example of a continuation
Fibonacci Generator

A generator function that lazily computes the Fibonacci sequence:

```python
def fib_generator():
    yield 0
    prev, current = 0, 1
    while True:
        yield current
        prev, current = current, prev + current
```

A generator expression is like a list comprehension, but it produces a lazy generator rather than a list:

```
double_fibs = (fib * 2 for fib in fib_generator())
```

Generator Semantics

```python
def fib_generator():
    yield 0
    prev, current = 0, 1
    while True:
        yield current
        prev, current = current, prev + current
```

Calling a generator function returns an iterator that stores a frame for the function, its body, and the current location in the body.

Calling `next` on the iterator resumes execution of the body at the current location, until a `yield` is reached.

The yielded value is returned by `next`, and execution of the body is halted until the next call to `next`.

When execution reaches the end of the body, a `StopIteration` is raised.

Map and Filter

```python
def map_gen(fn, iterable):
    iterator = iter(iterable)
    while True:
        yield fn(next(iterator))

def filter_gen(fn, iterable):
    iterator = iter(iterable)
    while True:
        item = next(iterator)
        if fn(item):
            yield item
```

Bitstring Generator

```python
from itertools import product

def bitstrings():
    """Generate bitstrings in order of increasing size."
    size = 0
    while True:
        tuples = product(('0', '1'), repeat=size)
        for elem in tuples:
            yield ''.join(elem)
        size += 1
```

```python
from itertools import product
def bitstrings():
    """Generate bitstrings in order of increasing size."
    size = 0
    while True:
        tuples = product(('0', '1'), repeat=size)
        for elem in tuples:
            yield ''.join(elem)
        size += 1
```