Last week
Last week

Distributed computing
Last week

Distributed computing
  • Client–server
Last week

Distributed computing

- Client-server
- Peer-to-peer
Last week

Distributed computing

- Client-server
- Peer-to-peer
- Message passing
Last week

Distributed computing

- Client-server
- Peer-to-peer
- Message passing
- Modularity
Last week

Distributed computing
- Client-server
- Peer-to-peer
- Message passing
- Modularity
- Interfaces
Last week

Distributed computing
- Client–server
- Peer–to–peer
- Message passing
- Modularity
- Interfaces

Parallel computing
Last week

Distributed computing
- Client–server
- Peer-to-peer
- Message passing
- Modularity
- Interfaces

Parallel computing
- Threads
Last week

Distributed computing
- Client–server
- Peer-to-peer
- Message passing
- Modularity
- Interfaces

Parallel computing
- Threads
- Shared memory
Last week

Distributed computing
  ▪ Client–server
  ▪ Peer–to–peer
  ▪ Message passing
  ▪ Modularity
  ▪ Interfaces

Parallel computing
  ▪ Threads
  ▪ Shared memory
  ▪ Problems: Synchronization and stale data
Last week

Distributed computing
- Client-server
- Peer-to-peer
- Message passing
- Modularity
- Interfaces

Parallel computing
- Threads
- Shared memory
- Problems: Synchronization and stale data
- Solutions: Locks, semaphores (and conditions)
Last week

Distributed computing
- Client-server
- Peer-to-peer
- Message passing
- Modularity
- Interfaces

Parallel computing
- Threads
- Shared memory
- Problems: Synchronization and stale data
- Solutions: Locks, semaphores (and conditions)
- Deadlock
Sequential data
Sequential data

Some of the most interesting real-world problems in computer science center around sequential data.
Sequential data

Some of the most interesting real-world problems in computer science center around sequential data.

DNA sequences
Sequential data

Some of the most interesting real-world problems in computer science center around sequential data.

DNA sequences

Web and cell-phone traffic streams
Sequential data

Some of the most interesting real-world problems in computer science center around sequential data.

DNA sequences

Web and cell-phone traffic streams

The social data stream
Sequential data

Some of the most interesting real–world problems in computer science center around sequential data.

DNA sequences

Web and cell–phone traffic streams

The social data stream

Series of measurements from instruments on a robot
Sequential data

Some of the most interesting real-world problems in computer science center around sequential data.

DNA sequences

Web and cell-phone traffic streams

The social data stream

Series of measurements from instruments on a robot

Stock prices, weather patterns
So far: the sequence abstraction
So far: the sequence abstraction

Sequences have
So far: the sequence abstraction

Sequences have
- Length
So far: the sequence abstraction

Sequences have

- Length
- Element selection
So far: the sequence abstraction

Sequences have

- Length
- Element selection
- In python
So far: the sequence abstraction

Sequences have

- Length
- Element selection
- In python
  - Membership testing
So far: the sequence abstraction

Sequences have

- Length
- Element selection
- In python
  - Membership testing
  - Slicing
So far: the sequence abstraction

Sequences have

- Length
- Element selection
- In python
  - Membership testing
  - Slicing

Data structures that support the sequence abstraction
So far: the sequence abstraction

Sequences have
  ▪ Length
  ▪ Element selection
  ▪ In python
    • Membership testing
    • Slicing

Data structures that support the sequence abstraction
  ▪ Nested tuples
So far: the sequence abstraction

Sequences have

- Length
- Element selection
- In python
  - Membership testing
  - Slicing

Data structures that support the sequence abstraction

- Nested tuples
- Tuples
So far: the sequence abstraction

Sequences have

- Length
- Element selection
- In python
  - Membership testing
  - Slicing

Data structures that support the sequence abstraction

- Nested tuples
- Tuples
- Strings
So far: the sequence abstraction

Sequences have

- Length
- Element selection
- In python
  - Membership testing
  - Slicing

Data structures that support the sequence abstraction

- Nested tuples
- Tuples
- Strings
- Lists (mutable)
Problems with sequences
Problems with sequences

Memory
Problems with sequences

Memory

- Each item must be explicitly represented
Problems with sequences

Memory

- Each item must be explicitly represented
- Even if all can be generated by a common formula or function
Problems with sequences

Memory

- Each item must be explicitly represented
- Even if all can be generated by a common formula or function

Up-front computation
Problems with sequences

Memory

- Each item must be explicitly represented
- Even if all can be generated by a common formula or function

Up-front computation

- Have to compute all items up-front
Problems with sequences

Memory

- Each item must be explicitly represented
- Even if all can be generated by a common formula or function

Up-front computation

- Have to compute all items up-front
- Even if using them one by one
Problems with sequences

Memory

- Each item must be explicitly represented
- Even if all can be generated by a common formula or function

Up-front computation

- Have to compute all items up-front
- Even if using them one by one

Can’t be infinite
Problems with sequences

Memory
- Each item must be explicitly represented
- Even if all can be generated by a common formula or function

Up-front computation
- Have to compute all items up-front
- Even if using them one by one

Can’t be infinite
- Why care about “infinite” sequences?
Problems with sequences

Memory
- Each item must be explicitly represented
- Even if all can be generated by a common formula or function

Up-front computation
- Have to compute all items up-front
- Even if using them one by one

Can’t be infinite
- Why care about “infinite” sequences?
  - They’re everywhere!
Problems with sequences

Memory
- Each item must be explicitly represented
- Even if all can be generated by a common formula or function

Up-front computation
- Have to compute all items up-front
- Even if using them one by one

Can’t be infinite
- Why care about “infinite” sequences?
  • They’re everywhere!
  • Internet and cell phone traffic
Problems with sequences

Memory
- Each item must be explicitly represented
- Even if all can be generated by a common formula or function

Up-front computation
- Have to compute all items up-front
- Even if using them one by one

Can’t be infinite
- Why care about “infinite” sequences?
  - They’re everywhere!
  - Internet and cell phone traffic
  - Instrument measurement feeds, real-time data
Problems with sequences

Memory

- Each item must be explicitly represented
- Even if all can be generated by a common formula or function

Up-front computation

- Have to compute all items up-front
- Even if using them one by one

Can’t be infinite

- Why care about “infinite” sequences?
  - They’re everywhere!
  - Internet and cell phone traffic
  - Instrument measurement feeds, real-time data
  - Mathematical sequences
Finding prime numbers
Finding prime numbers

Sieve of Erastothenes
Finding prime numbers

Sieve of Eratosthenes

- Find prime numbers by walking down integers
Finding prime numbers

Sieve of Eratosthenes

- Find prime numbers by walking down integers
- For each integer, eliminate all multiples of that integer
Finding prime numbers

Sieve of Eratosthenes

- Find prime numbers by walking down integers
- For each integer, eliminate all multiples of that integer
- Left with indivisible numbers
Finding prime numbers

Sieve of Erastothenes

- Find prime numbers by walking down integers
- For each integer, eliminate all multiples of that integer
- Left with indivisible numbers

2 3 4 5 6 7 8 9 10 11 12 13
Finding prime numbers

Sieve of Erastothenes

- Find prime numbers by walking down integers
- For each integer, eliminate all multiples of that integer
- Left with indivisible numbers

2 3 4 5 6 7 8 9 10 11 12 13

2 3 4 5 6 7 8 9 10 11 12 13
Finding prime numbers

Sieve of Erastothenes

- Find prime numbers by walking down integers
- For each integer, eliminate all multiples of that integer
- Left with indivisible numbers

2 3 4 5 6 7 8 9 10 11 12 13

2 3 4 5 6 7 8 9 10 11 12 13

2 3 4 5 6 7 8 9 10 11 12 13
Finding prime numbers

Sieve of Erastothenes

- Find prime numbers by walking down integers
- For each integer, eliminate all multiples of that integer
- Left with indivisible numbers

2 3 4 5 6 7 8 9 10 11 12 13
2 3 4 5 6 7 8 9 10 11 12 13
2 3 4 5 6 7 8 9 10 11 12 13
2 3 4 5 6 7 8 9 10 11 12 13

Monday, November 21, 2011
Finding prime numbers

Sieve of Eratosthenes

- Find prime numbers by walking down integers
- For each integer, eliminate all multiples of that integer
- Left with indivisible numbers

2 3 4 5 6 7 8 9 10 11 12 13

2 3 4 5 6 7 8 9 10 11 12 13

2 3 4 5 6 7 8 9 10 11 12 13

2 3 4 5 6 7 8 9 10 11 12 13
Finding prime numbers

Sieve of Erastothenes

- Find prime numbers by walking down integers
- For each integer, eliminate all multiples of that integer
- Left with indivisible numbers

2 3 4 5 6 7 8 9 10 11 12 13

2 3 4 5 6 7 8 9 10 11 12 13

2 3 4 5 6 7 8 9 10 11 12 13

2 3 4 5 6 7 8 9 10 11 12 13

2 3 4 5 6 7 8 9 10 11 12 13

2 3 4 5 6 7 8 9 10 11 12 13

2 3 4 5 6 7 8 9 10 11 12 13
Finding prime numbers

Sieve of Erastothenes

- Find prime numbers by walking down integers
- For each integer, eliminate all multiples of that integer
- Left with indivisible numbers

- 2 3 4 5 6 7 8 9 10 11 12 13
- 2 3 4 5 6 7 8 9 10 11 12 13
- 2 3 4 5 6 7 8 9 10 11 12 13
- 2 3 4 5 6 7 8 9 10 11 12 13
- 2 3 4 5 6 7 8 9 10 11 12 13
- 2 3 4 5 6 7 8 9 10 11 12 13
- 2 3 4 5 6 7 8 9 10 11 12 13
- 2 3 4 5 6 7 8 9 10 11 12 13
- 2 3 4 5 6 7 8 9 10 11 12 13
Finding prime numbers

Sieve of Erastothenes

- Find prime numbers by walking down integers
- For each integer, eliminate all multiples of that integer
- Left with indivisible numbers

2 3 4 5 6 7 8 9 10 11 12 13
2 3 4 5 6 7 8 9 10 11 12 13
2 3 4 5 6 7 8 9 10 11 12 13
2 3 4 5 6 7 8 9 10 11 12 13
2 3 4 5 6 7 8 9 10 11 12 13
2 3 4 5 6 7 8 9 10 11 12 13
2 3 4 5 6 7 8 9 10 11 12 13
2 3 4 5 6 7 8 9 10 11 12 13
2 3 4 5 6 7 8 9 10 11 12 13
Finding prime numbers

Sieve of Erastothenes

- Find prime numbers by walking down integers
- For each integer, eliminate all multiples of that integer
- Left with indivisible numbers

2 3 4 5 6 7 8 9 10 11 12 13

2 3 4 5 6 7 8 9 10 11 12 13

2 3 4 5 6 7 8 9 10 11 12 13

2 3 4 5 6 7 8 9 10 11 12 13

2 3 4 5 6 7 8 9 10 11 12 13

2 3 4 5 6 7 8 9 10 11 12 13

2 3 4 5 6 7 8 9 10 11 12 13

2 3 4 5 6 7 8 9 10 11 12 13

2 3 4 5 6 7 8 9 10 11 12 13
Finding prime numbers

Sieve of Erastothenes

- Find prime numbers by walking down integers
- For each integer, eliminate all multiples of that integer
- Left with indivisible numbers

2 3 4 5 6 7 8 9 10 11 12 13
2 3 4 5 6 7 8 9 10 11 12 13
2 3 4 5 6 7 8 9 10 11 12 13
2 3 4 5 6 7 8 9 10 11 12 13
2 3 4 5 6 7 8 9 10 11 12 13
2 3 4 5 6 7 8 9 10 11 12 13
2 3 4 5 6 7 8 9 10 11 12 13
2 3 4 5 6 7 8 9 10 11 12 13
2 3 4 5 6 7 8 9 10 11 12 13
2 3 4 5 6 7 8 9 10 11 12 13
2 3 4 5 6 7 8 9 10 11 12 13
2 3 4 5 6 7 8 9 10 11 12 13
2 3 4 5 6 7 8 9 10 11 12 13
Finding prime numbers

Sieve of Erastothenes

- Find prime numbers by walking down integers
- For each integer, eliminate all multiples of that integer
- Left with indivisible numbers

2 3 4 5 6 7 8 9 10 11 12 13
2 3 4 5 6 7 8 9 10 11 12 13
2 3 4 5 6 7 8 9 10 11 12 13
2 3 4 5 6 7 8 9 10 11 12 13
2 3 4 5 6 7 8 9 10 11 12 13
2 3 4 5 6 7 8 9 10 11 12 13
2 3 4 5 6 7 8 9 10 11 12 13
2 3 4 5 6 7 8 9 10 11 12 13
2 3 4 5 6 7 8 9 10 11 12 13
2 3 4 5 6 7 8 9 10 11 12 13
2 3 4 5 6 7 8 9 10 11 12 13
2 3 4 5 6 7 8 9 10 11 12 13
2 3 4 5 6 7 8 9 10 11 12 13
2 3 4 5 6 7 8 9 10 11 12 13
2 3 4 5 6 7 8 9 10 11 12 13
2 3 4 5 6 7 8 9 10 11 12 13
2 3 4 5 6 7 8 9 10 11 12 13
2 3 4 5 6 7 8 9 10 11 12 13
Finding prime numbers

Sieve of Eratosthenes

- Find prime numbers by walking down integers
- For each integer, eliminate all multiples of that integer
- Left with indivisible numbers

2 3 4 5 6 7 8 9 10 11 12 13
2 3 4 5 6 7 8 9 10 11 12 13
2 3 4 5 6 7 8 9 10 11 12 13
2 3 4 5 6 7 8 9 10 11 12 13
2 3 4 5 6 7 8 9 10 11 12 13
2 3 4 5 6 7 8 9 10 11 12 13
2 3 4 5 6 7 8 9 10 11 12 13
2 3 4 5 6 7 8 9 10 11 12 13
2 3 4 5 6 7 8 9 10 11 12 13
2 3 4 5 6 7 8 9 10 11 12 13
2 3 4 5 6 7 8 9 10 11 12 13
2 3 4 5 6 7 8 9 10 11 12 13
2 3 4 5 6 7 8 9 10 11 12 13
2 3 4 5 6 7 8 9 10 11 12 13
Working example: finding prime numbers
Working example: finding prime numbers

def primes_sieve(limit):

Working example: finding prime numbers

def primes_sieve(limit):
    # mark all numbers as prime at first
def primes_sieve(limit):
    # mark all numbers as prime at first
    prime = [True] * (limit+1)
Working example: finding prime numbers

def primes_sieve(limit):
    # mark all numbers as prime at first
    prime = [True] * (limit+1)
    primes = []

def primes_sieve(limit):
    # mark all numbers as prime at first
    prime = [True] * (limit+1)
    primes = []
    # eliminate multiples of previous numbers
def primes_sieve(limit):
    # mark all numbers as prime at first
    prime = [True] * (limit+1)
    primes = []
    # eliminate multiples of previous numbers
    for i in range(2, limit+1):
Working example: finding prime numbers

def primes_sieve(limit):
    # mark all numbers as prime at first
    prime = [True] * (limit+1)
    primes = []
    # eliminate multiples of previous numbers
    for i in range(2, limit+1):
        if prime[i]:
            # ...
Working example: finding prime numbers

def primes_sieve(limit):
    # mark all numbers as prime at first
    prime = [True] * (limit+1)
    primes = []
    # eliminate multiples of previous numbers
    for i in range(2, limit+1):
        if prime[i]:
            primes.append(i)
Working example: finding prime numbers

def primes_sieve(limit):
    # mark all numbers as prime at first
    prime = [True] * (limit+1)
    primes = []
    # eliminate multiples of previous numbers
    for i in range(2, limit+1):
        if prime[i]:
            primes.append(i)
            multiple = i*i
def primes_sieve(limit):
    # mark all numbers as prime at first
    prime = [True] * (limit+1)
    primes = []
    # eliminate multiples of previous numbers
    for i in range(2, limit+1):
        if prime[i]:
            primes.append(i)
            multiple = i*i
            while multiple <= limit :
                prime[multiple] = False
                multiple += i
Working example: finding prime numbers

def primes_sieve(limit):
    # mark all numbers as prime at first
    prime = [True] * (limit+1)
    primes = []
    # eliminate multiples of previous numbers
    for i in range(2, limit+1):
        if prime[i]:
            primes.append(i)
            multiple = i*i
            while multiple <= limit :
                prime[multiple] = False
def primes_sieve(limit):
    # mark all numbers as prime at first
    prime = [True] * (limit+1)
    primes = []
    # eliminate multiples of previous numbers
    for i in range(2, limit+1):
        if prime[i]:
            primes.append(i)
            multiple = i*i
            while multiple <= limit:
                prime[multiple] = False
                multiple += i
Working example: finding prime numbers

def primes_sieve(limit):
    # mark all numbers as prime at first
    prime = [True] * (limit+1)
    primes = []
    # eliminate multiples of previous numbers
    for i in range(2, limit+1):
        if prime[i]:
            primes.append(i)
            multiple = i*i
            while multiple <= limit :
                prime[multiple] = False
                multiple += i
    return primes
def primes_sieve(limit):
    # mark all numbers as prime at first
    prime = [True] * (limit+1)
    primes = []
    # eliminate multiples of previous numbers
    for i in range(2, limit+1):
        if prime[i]:
            primes.append(i)
            multiple = i*i
            while multiple <= limit:
                prime[multiple] = False
                multiple += i
    return primes
def primes_sieve(limit):
    # mark all numbers as prime at first
    prime = [True] * (limit+1)
    primes = []
    # eliminate multiples of previous numbers
    for i in range(2, limit+1):
        if prime[i]:
            primes.append(i)
            multiple = i*i
            while multiple <= limit :
                prime[multiple] = False
                multiple += i
    return primes
def primes_sieve(limit):
    # mark all numbers as prime at first
    prime = [True] * (limit+1)
    primes = []
    # eliminate multiples of previous numbers
    for i in range(2, limit+1):
        if prime[i]:
            primes.append(i)
            multiple = i*i
            while multiple <= limit :
                prime[multiple] = False
                multiple += i

    return primes

primes_sieve(1000000000) anyone?
def primes_sieve(limit):
    # mark all numbers as prime at first
    prime = [True] * (limit+1)
    primes = []
    # eliminate multiples of previous numbers
    for i in range(2, limit+1):
        if prime[i]:
            primes.append(i)
            multiple = i*i
            while multiple <= limit :
                prime[multiple] = False
                multiple += i
    return primes

primes_sieve(1000000000) anyone?
  1 billion
def primes_sieve(limit):
    # mark all numbers as prime at first
    prime = [True] * (limit+1)
    primes = []
    # eliminate multiples of previous numbers
    for i in range(2, limit+1):
        if prime[i]:
            primes.append(i)
            multiple = i*i
            while multiple <= limit :
                prime[multiple] = False
                multiple += i
    return primes

primes_sieve(1000000000) anyone?
  1 billion
  each number = 64 bits = 8 bytes
def primes_sieve(limit):
    # mark all numbers as prime at first
    prime = [True] * (limit+1)
    primes = []
    # eliminate multiples of previous numbers
    for i in range(2, limit+1):
        if prime[i]:
            primes.append(i)
            multiple = i*i
            while multiple <= limit :
                prime[multiple] = False
                multiple += i
    return primes

primes_sieve(1000000000) anyone?
1 billion
each number = 64 bits = 8 bytes
8 bytes * 1 billion * 2 = 16 billion bytes
def primes_sieve(limit):
    # mark all numbers as prime at first
    prime = [True] * (limit+1)
    primes = []
    # eliminate multiples of previous numbers
    for i in range(2, limit+1):
        if prime[i]:
            primes.append(i)
            multiple = i*i
            while multiple <= limit :
                prime[multiple] = False
                multiple += i
    return primes

primes_sieve(1000000000) anyone?
1 billion
each number = 64 bits = 8 bytes
8 bytes * 1 billion * 2 = 16 billion bytes
= ~14.9 GB of memory
Iterators: another abstraction for sequential data
Iterators: another abstraction for sequential data

Iterators
Iterators: another abstraction for sequential data

Iterators
- Store how to compute items instead of items themselves
Iterators: another abstraction for sequential data

Iterators

- Store how to compute items instead of items themselves
- Give out one item at a time
Iterators: another abstraction for sequential data

Iterators

- Store **how to compute items** instead of items themselves
- Give out one item at a time
- Save the next until asked (lazy evaluation)
Iterators: another abstraction for sequential data

Iterators

- Store how to compute items instead of items themselves
- Give out one item at a time
- Save the next until asked (lazy evaluation)

Compared with sequences
Iterators: another abstraction for sequential data

Iterators

- Store how to compute items instead of items themselves
- Give out one item at a time
- Save the next until asked (lazy evaluation)

Compared with sequences

- Length not explicitly defined
Iterators: another abstraction for sequential data

Iterators
- Store **how to compute items** instead of items themselves
- Give out one item at a time
- Save the next until asked (lazy evaluation)

Compared with sequences
- Length not explicitly defined
- Element selection not supported
Iterators: another abstraction for sequential data

Iterators
- Store how to compute items instead of items themselves
- Give out one item at a time
- Save the next until asked (lazy evaluation)

Compared with sequences
- Length not explicitly defined
- Element selection not supported
  - Element selection -- random access
Iterators: another abstraction for sequential data

Iterators

- Store how to compute items instead of items themselves
- Give out one item at a time
- Save the next until asked (lazy evaluation)

Compared with sequences

- Length not explicitly defined
- Element selection not supported
  - Element selection — random access
  - Iterators — sequential access
Iterators: another abstraction for sequential data

Iterators
- Store **how to compute items** instead of items themselves
- Give out one item at a time
- Save the next until asked (lazy evaluation)

Compared with sequences
- Length not explicitly defined
- Element selection not supported
  - Element selection -- random access
  - Iterators -- sequential access
- No up-front computation of all items
Iterators: another abstraction for sequential data

Iterators
- Store how to compute items instead of items themselves
- Give out one item at a time
- Save the next until asked (lazy evaluation)

Compared with sequences
- Length not explicitly defined
- Element selection not supported
  - Element selection -- random access
  - Iterators -- sequential access
- No up-front computation of all items
- Only one item stored at a time
Iterators: another abstraction for sequential data

Iterators

- Store how to compute items instead of items themselves
- Give out one item at a time
- Save the next until asked (lazy evaluation)

Compared with sequences

- Length not explicitly defined
- Element selection not supported
  - Element selection -- random access
  - Iterators -- sequential access
- No up-front computation of all items
- Only one item stored at a time
- CAN be infinite
Implementation: nested delayed evaluation
Implementation: nested delayed evaluation

Nested pairs
Implementation: nested delayed evaluation

Nested pairs

Stream
Implementation: nested delayed evaluation

Nested pairs

Stream
Implementation: nested delayed evaluation

Nested pairs

Stream

first
Implementation: nested delayed evaluation

Nested pairs

Stream

first

rest
Implementation: nested delayed evaluation

Nested pairs

Stream

now, explicit
Implementation: nested delayed evaluation

Nested pairs

Stream

now, explicit

store how to compute it compute when asked
Streams

Diagram:

- First
- Rest
Streams

class Stream(object):

```
  first
```

```
  rest
```
class Stream(object):
    def __init__(self, first, compute_rest, empty=False):
class Stream(object):

    def __init__(self, first, compute_rest, empty=False):
        self.first = first
class Stream(object):

    def __init__(self, first, compute_rest, empty=False):
        self.first = first
        self._compute_rest = compute_rest
class Stream(object):
    
def __init__(self, first, compute_rest, empty= False):
        self.first = first
        self._compute_rest = compute_rest
        self.empty = empty
class Stream(object):
    def __init__(self, first, compute_rest, empty=False):
        self.first = first
        self._compute_rest = compute_rest
        self.empty = empty
        self._rest = None
class Stream(object):
    def __init__(self, first, compute_rest, empty=False):
        self.first = first
        self._compute_rest = compute_rest
        self.empty = empty
        self._rest = None
        self._computed = False
class Stream(object):

    def __init__(self, first, compute_rest, empty=False):
        self.first = first
        self._compute_rest = compute_rest
        self.empty = empty
        self._rest = None
        self._computed = False

@property
class Stream(object):
    def __init__(self, first, compute_rest, empty=False):
        self.first = first
        self._compute_rest = compute_rest
        self.empty = empty
        self._rest = None
        self._computed = False

@property
def rest(self):
    pass
class Stream(object):

    def __init__(self, first, compute_rest, empty=False):
        self.first = first
        self._compute_rest = compute_rest
        self.empty = empty
        self._rest = None
        self._computed = False

@property
def rest(self):
    assert not self.empty, 'Empty streams have no rest.'
class Stream(object):
    def __init__(self, first, compute_rest, empty=False):
        self.first = first
        self._compute_rest = compute_rest
        self.empty = empty
        self._rest = None
        self._computed = False

@property
def rest(self):
    assert not self.empty, 'Empty streams have no rest.'
    if not self._computed:
class Stream(object):
    def __init__(self, first, compute_rest, empty=False):
        self.first = first
        self._compute_rest = compute_rest
        self.empty = empty
        self._rest = None
        self._computed = False

@property
def rest(self):
    assert not self.empty, 'Empty streams have no rest.'
    if not self._computed:
        self._rest = self._compute_rest()
```python
class Stream(object):
    def __init__(self, first, compute_rest, empty=False):
        self.first = first
        self._compute_rest = compute_rest
        self.empty = empty
        self._rest = None
        self._computed = False

    @property
def rest(self):
        assert not self.empty, 'Empty streams have no rest.'
        if not self._computed:
            self._rest = self._compute_rest()
            self._computed = True
```

**Streams**

```plaintext```
```
class Stream(object):
    def __init__(self, first, compute_rest, empty=False):
        self.first = first
        self._compute_rest = compute_rest
        self.empty = empty
        self._rest = None
        self._computed = False

@property
def rest(self):
    assert not self.empty, 'Empty streams have no rest.'
    if not self._computed:
        self._rest = self._compute_rest()
        self._computed = True
    return self._rest
Streams

```python
class Stream(object):
    def __init__(self, first, compute_rest, empty=False):
        self.first = first
        self._compute_rest = compute_rest
        self.empty = empty
        self._rest = None
        self._computed = False

@property
def rest(self):
    assert not self.empty, 'Empty streams have no rest.'
    if not self._computed:
        self._rest = self._compute_rest()
        self._computed = True
    return self._rest

empty_stream = Stream(None, None, True)
```
Sequential data: nested streams
Sequential data: nested streams

Nest streams inside each other
Sequential data: nested streams

Nest streams inside each other
Only compute one element of a sequence at a time
Sequential data: nested streams

Nest streams inside each other
Only compute one element of a sequence at a time
Sequential data: nested streams

Nest streams inside each other
Only compute one element of a sequence at a time
Sequential data: nested streams

Nest streams inside each other
Only compute one element of a sequence at a time
Sequential data: nested streams

Nest streams inside each other
Only compute one element of a sequence at a time

```python
def make_integer_stream(first=1):
```
Sequential data: nested streams

Nest streams inside each other
Only compute one element of a sequence at a time

```python
def make_integer_stream(first=1):
    def compute_rest():
```
Sequential data: nested streams

Nest streams inside each other
Only compute one element of a sequence at a time

```python
def make_integer_stream(first=1):
    def compute_rest():
        return make_integer_stream(first+1)
```
Sequential data: nested streams

Nest streams inside each other
Only compute one element of a sequence at a time

```python
def make_integer_stream(first=1):
    def compute_rest():
        return make_integer_stream(first+1)
    return Stream(first, compute_rest)
```
Sequential data: nested streams

Nest streams inside each other
Only compute one element of a sequence at a time

```python
def make_integer_stream(first=1):
    def compute_rest():
        return make_integer_stream(first+1)
    return Stream(first, compute_rest)
```

live example
Prime numbers with nested streams
Prime numbers with nested streams

def filter_stream(filter_func, stream):

def filter_stream(filter_func, stream):
    def make_filtered_rest():
        pass
Prime numbers with nested streams

def filter_stream(filter_func, stream):
    def make_filtered_rest():
        return filter_stream(filter_func, stream.rest)
def filter_stream(filter_func, stream):
    def make_filtered_rest():
        return filter_stream(filter_func, stream.rest)
    if stream.empty:
        return make_filtered_rest()
def filter_stream(filter_func, stream):
    def make_filtered_rest():
        return filter_stream(filter_func, stream.rest)
    if stream.empty:
        return stream
    return make_filtered_rest()
Prime numbers with nested streams

```python
def filter_stream(filter_func, stream):
    def make_filtered_rest():
        return filter_stream(filter_func, stream.rest)
    if stream.empty:
        return stream
    if filter_func(stream.first):
        return stream.first
```

Monday, November 21, 2011
Prime numbers with nested streams

def filter_stream(filter_func, stream):
    def make_filtered_rest():
        return filter_stream(filter_func, stream.rest)
    if stream.empty:
        return stream
    if filter_func(stream.first):
        return Stream(s.first, make_filtered_rest)
Prime numbers with nested streams

```python
def filter_stream(filter_func, stream):
    def make_filtered_rest():
        return filter_stream(filter_func, stream.rest)
    if stream.empty:
        return stream
    if filter_func(stream.first):
        return Stream(s.first, make_filtered_rest)
    else:
```

Monday, November 21, 2011
def filter_stream(filter_func, stream):
    def make_filtered_rest():
        return filter_stream(filter_func, stream.rest)
    if stream.empty:
        return stream
    if filter_func(stream.first):
        return Stream(s.first, make_filtered_rest)
    else:
        return filter_stream(filter_func, stream.rest)
def filter_stream(filter_func, stream):
    def make_filtered_rest():
        return filter_stream(filter_func, stream.rest)
    if stream.empty:
        return stream
    if filter_func(stream.first):
        return Stream(s.first, make_filtered_rest)
    else:
        return filter_stream(filter_func, stream.rest)

def primes(positive_ints):
def filter_stream(filter_func, stream):
    def make_filtered_rest():
        return filter_stream(filter_func, stream.rest)
    if stream.empty:
        return stream
    if filter_func(stream.first):
        return Stream(s.first, make_filtered_rest)
    else:
        return filter_stream(filter_func, stream.rest)

def primes(positive_ints):
    def not_divible(x):
def filter_stream(filter_func, stream):
    def make_filtered_rest():
        return filter_stream(filter_func, stream.rest)
    if stream.empty:
        return stream
    if filter_func(stream.first):
        return Stream(s.first, make_filtered_rest)
    else:
        return filter_stream(filter_func, stream.rest)

def primes(positive_ints):
    def not_divible(x):
        return (x % positive_integers.first) != 0
Prime numbers with nested streams

```python
def filter_stream(filter_func, stream):
    def make_filtered_rest():
        return filter_stream(filter_func, stream.rest)
    if stream.empty:
        return stream
    if filter_func(stream.first):
        return Stream(s.first, make_filtered_rest)
    else:
        return filter_stream(filter_func, stream.rest)

def primes(positive_ints):
    def not_divible(x):
        return (x % positive_integers.first) != 0
    def sieve():
```
Prime numbers with nested streams

```python
def filter_stream(filter_func, stream):
    def make_filtered_rest():
        return filter_stream(filter_func, stream.rest)
    if stream.empty:
        return stream
    if filter_func(stream.first):
        return Stream(s.first, make_filtered_rest)
    else:
        return filter_stream(filter_func, stream.rest)

def primes(positive_ints):
    def not_divible(x):
        return (x % positive_integers.first) != 0
    def sieve():
        return primes(filter_stream(not_divible, positive_ints.rest))
```

Monday, November 21, 2011
def filter_stream(filter_func, stream):
    def make_filtered_rest():
        return filter_stream(filter_func, stream.rest)
    if stream.empty:
        return stream
    if filter_func(stream.first):
        return Stream(s.first, make_filtered_rest)
    else:
        return filter_stream(filter_func, stream.rest)

def primes(positive_ints):
    def not_divible(x):
        return (x % positive_integers.first) != 0
    def sieve():
        return primes(filter_stream(not_divible, positive_ints.rest))
    return Stream(pos_stream.first, sieve)
Prime numbers with nested streams
Prime numbers with nested streams

def primes(positive_ints):

Prime numbers with nested streams

def primes(positive_ints):
    def not_divible(x):
        # Function definitions
Prime numbers with nested streams

def primes(positive_ints):
    def not_divible(x):
        return (x % positive_integers.first) != 0
Prime numbers with nested streams

```python
def primes(positive_ints):
    def not_divible(x):
        return (x % positive_integers.first) != 0
    def sieve():
```

Monday, November 21, 2011
Prime numbers with nested streams

def primes(positive_ints):
    def not_divible(x):
        return (x % positive_integers.first) != 0
    def sieve():
        return primes(filter_stream(not_divible, positive_ints.rest)
Prime numbers with nested streams

def primes(positive_ints):
    def not_divible(x):
        return (x % positive_integers.first) != 0
    def sieve():
        return primes(filter_stream(not_divible, positive_ints.rest))
    return Stream(pos_stream.first, sieve)
Prime numbers with nested streams

def primes(positive_ints):
    def not_divible(x):
        return (x % positive_integers.first) != 0
    def sieve():
        return primes(filter_stream(not_divible, positive_ints.rest))
    return Stream(pos_stream.first, sieve)

>>> p = primes(make_integer_stream(5))
Prime numbers with nested streams

```python
def primes(positive_ints):
    def not_divible(x):
        return (x % positive_integers.first) != 0
    def sieve():
        return primes(filter_stream(not_divible, positive_ints.rest))
    return Stream(pos_stream.first, sieve)

>>> p = primes(make_integer_stream(5))
>>> p.first
```
def primes(positive_ints):
    def not_divible(x):
        return (x % positive_integers.first) != 0
    def sieve():
        return primes(filter_stream(not_divible, positive_ints.rest))
    return Stream(pos_stream.first, sieve)

>>> p = primes(make_integer_stream(5))

>>> p.first
Prime numbers with nested streams

```python
def primes(positive_ints):
    def not_divible(x):
        return (x % positive_integers.first) != 0
    def sieve():
        return primes(filter_stream(not_divible, positive_ints.rest))
    return Stream(pos_stream.first, sieve)

>>> p = primes(make_integer_stream(5))
>>> p.first
5
```
Prime numbers with nested streams

```python
def primes(positive_ints):
    def not_divible(x):
        return (x % positive_integers.first) != 0
    def sieve():
        return primes(filter_stream(not_divible, positive_ints.rest))
    return Stream(pos_stream.first, sieve)

>>> p = primes(make_integer_stream(5))
>>> p.first

5

>>> p.rest
```

Monday, November 21, 2011
def primes(positive_ints):
    def not_divible(x):
        return (x % positive_integers.first) != 0
    def sieve():
        return primes(filter_stream(not_divible, positive_ints.rest))
    return Stream(pos_stream.first, sieve)

>>> p = primes(make_integer_stream(5))
>>> p.first
5
>>> p.rest
def primes(positive_ints):
    def not_divible(x):
        return (x % positive_integers.first) != 0
    def sieve():
        return primes(filter_stream(not_divible, positive_ints.rest))
    return Stream(pos_stream.first, sieve)

>>> p = primes(make_integer_stream(5))
>>> p.first
5
>>> p.rest
<Stream instance at ... >
Prime numbers with nested streams

def primes(positive_ints):
    def not_divible(x):
        return (x % positive_integers.first) != 0
    def sieve():
        return primes(filter_stream(not_divible, positive_ints.rest))
    return Stream(pos_stream.first, sieve)

>>> p = primes(make_integer_stream(5))
>>> p.first
5
>>> p.rest
<Stream instance at ... >
>>> p.rest.first
Prime numbers with nested streams

```python
def primes(positive_ints):
    def not_divible(x):
        return (x % positive_integers.first) != 0
    def sieve():
        return primes(filter_stream(not_divible, positive_ints.rest))
    return Stream(pos_stream.first, sieve)

>>> p = primes(make_integer_stream(5))
>>> p.first
5
>>> p.rest
<Stream instance at ... >
>>> p.rest.first
7
```
Native python iterators

Python natively supports iterators
Native python iterators

Python natively supports iterators

The Iterator interface in python:
Native python iterators

Python natively supports iterators

The Iterator interface in python:

- __iter__
Native python iterators

Python natively supports iterators

The Iterator interface in python:

- __iter__
  - should return an iterator object
Native python iterators

Python natively supports iterators

The Iterator interface in python:

- `__iter__`
  - should return an iterator object
- `__next__`
Native python iterators

Python natively supports iterators

The Iterator interface in python:

- **__iter__**
  - should return an iterator object
- **__next__**
  - should return a value OR
Native python iterators

Python natively supports iterators

The Iterator interface in python:

- __iter__
  - should return an iterator object
- __next__
  - should return a value OR
- raise StopIteration
Native python iterators

Python natively supports iterators

The Iterator interface in python:

- __iter__
  - should return an iterator object
- __next__
  - should return a value OR
- raise StopIteration
  - when end of sequence is reached
Native python iterators

Python natively supports iterators

The Iterator interface in python:

- `__iter__`
  - should return an iterator object
- `__next__`
  - should return a value OR
  - raise StopIteration
    - when end of sequence is reached
    - on all subsequent calls
Native python iterators: example
Native python iterators: example

class Letters(object):

Native python iterators: example

class Letters(object):
    def __init__(self, start, finish):
Native python iterators: example

class Letters(object):
    def __init__(self, start, finish):
        self.current = start
Native python iterators: example

```python
class Letters(object):
    def __init__(self, start, finish):
        self.current = start
        self.finish = finish
```
class Letters(object):
    def __init__(self, start, finish):
        self.current = start
        self.finish = finish

    def __next__(self):
class Letters(object):
    def __init__(self, start, finish):
        self.current = start
        self.finish = finish

    def __next__(self):
        if self.current > self.finish:
            raise StopIteration
Native python iterators: example

```python
class Letters(object):
    def __init__(self, start, finish):
        self.current = start
        self.finish = finish

    def __next__(self):
        if self.current > self.finish:
            raise StopIteration
```
Native python iterators: example

class Letters(object):
    def __init__(self, start, finish):
        self.current = start
        self.finish = finish

    def __next__(self):
        if self.current > self.finish:
            raise StopIteration
        result = self.current
Native python iterators: example

class Letters(object):
    def __init__(self, start, finish):
        self.current = start
        self.finish = finish

    def __next__(self):
        if self.current > self.finish:
            raise StopIteration
        result = self.current
        self.current = chr(ord(result)+1)
Native python iterators: example

class Letters(object):
    def __init__(self, start, finish):
        self.current = start
        self.finish = finish

    def __next__(self):
        if self.current > self.finish:
            raise StopIteration
        result = self.current
        self.current = chr(ord(result)+1)
        return result
Native python iterators: example

class Letters(object):
    def __init__(self, start, finish):
        self.current = start
        self.finish = finish

    def __next__(self):
        if self.current > self.finish:
            raise StopIteration
        result = self.current
        self.current = chr(ord(result)+1)
        return result

    def __iter__(self):
Native python iterators: example

class Letters(object):
    def __init__(self, start, finish):
        self.current = start
        self.finish = finish

    def __next__(self):
        if self.current > self.finish:
            raise StopIteration
        result = self.current
        self.current = chr(ord(result)+1)
        return result

    def __iter__(self):
        return self
Native python iterators: example

class Letters(object):
    def __init__(self, start, finish):
        self.current = start
        self.finish = finish

    def __next__(self):
        if self.current > self.finish:
            raise StopIteration
        result = self.current
        self.current = chr(ord(result)+1)
        return result

    def __iter__(self):
        return self

>>> letters = Letters('a', 'd')
Native python iterators: example

class Letters(object):
    def __init__(self, start, finish):
        self.current = start
        self.finish = finish

    def __next__(self):
        if self.current > self.finish:
            raise StopIteration
        result = self.current
        self.current = chr(ord(result)+1)
        return result

    def __iter__(self):
        return self

>>> letters = Letters('a', 'd')
>>> letters.__next__()
Native python iterators: example

class Letters(object):
    def __init__(self, start, finish):
        self.current = start
        self.finish = finish

    def __next__(self):
        if self.current > self.finish:
            raise StopIteration
        result = self.current
        self.current = chr(ord(result)+1)
        return result

    def __iter__(self):
        return self

letters = Letters('a', 'd')
print(letters.__next__())
Native python iterators: example

class Letters(object):
    def __init__(self, start, finish):
        self.current = start
        self.finish = finish

    def __next__(self):
        if self.current > self.finish:
            raise StopIteration
        result = self.current
        self.current = chr(ord(result)+1)
        return result

    def __iter__(self):
        return self
def __init__(self, start, finish):
    def __next__(self):
        if self.current > self.finish:
            raise StopIteration
        result = self.current
        self.current = chr(ord(result)+1)
        return result

    def __iter__(self):
        return self

letters = Letters('a', 'd')
letters.__next__()
letters.__next__()
Native python iterators: example

class Letters(object):
    def __init__(self, start, finish):
        self.current = start
        self.finish = finish

    def __next__(self):
        if self.current > self.finish:
            raise StopIteration
        result = self.current
        self.current = chr(ord(result)+1)
        return result

    def __iter__(self):
        return self

>>> letters = Letters('a', 'd')
>>> letters.__next__()
'a'

>>> letters.__next__()
'b'
Native python iterators: example

class Letters(object):
    def __init__(self, start, finish):
        self.current = start
        self.finish = finish

    def __next__(self):
        if self.current > self.finish:
            raise StopIteration
        result = self.current
        self.current = chr(ord(result)+1)
        return result

    def __iter__(self):
        return self

>>> letters = Letters('a', 'd')
>>> letters.__next__()
'a'

>>> letters.__next__()
'b'

>>> letters.__next__()
Native python iterators: example

```python
class Letters(object):
    def __init__(self, start, finish):
        self.current = start
        self.finish = finish

    def __next__(self):
        if self.current > self.finish:
            raise StopIteration
        result = self.current
        self.current = chr(ord(result)+1)
        return result

    def __iter__(self):
        return self

letters = Letters('a', 'd')
letters.__next__()
letters.__next__()
letters.__next__()
```

Monday, November 21, 2011
Native python iterators: example

class Letters(object):
    def __init__(self, start, finish):
        self.current = start
        self.finish = finish

    def __next__(self):
        if self.current > self.finish:
            raise StopIteration
        result = self.current
        self.current = chr(ord(result)+1)
        return result

    def __iter__(self):
        return self

letters = Letters('a', 'd')
>>> letters.__next__()
'b'
>>> letters.__next__()
'c'
>>> letters.__next__()
Native python iterators: example

class Letters(object):
    def __init__(self, start, finish):
        self.current = start
        self.finish = finish

    def __next__(self):
        if self.current > self.finish:
            raise StopIteration
        result = self.current
        self.current = chr(ord(result)+1)
        return result

    def __iter__(self):
        return self

>>> letters = Letters('a', 'd')
>>> letters.__next__()
'a'

>>> letters.__next__()
'b'

>>> letters.__next__()
'c'

>>> letters.__next__()
'd'
Native python iterators: example

class Letters(object):
    def __init__(self, start, finish):
        self.current = start
        self.finish = finish

    def __next__(self):
        if self.current > self.finish:
            raise StopIteration
        result = self.current
        self.current = chr(ord(result)+1)
        return result

    def __iter__(self):
        return self

>>> letters = Letters('a', 'd')
>>> letters.__next__()
'a'

>>> letters.__next__()
'b'

>>> letters.__next__()
'c'

>>> letters.__next__()
'd'

>>> letters.__next__()
Native python iterators: example

```python
class Letters(object):
    def __init__(self, start, finish):
        self.current = start
        self.finish = finish

    def __next__(self):
        if self.current > self.finish:
            raise StopIteration
        result = self.current
        self.current = chr(ord(result)+1)
        return result

    def __iter__(self):
        return self

>>> letters = Letters('a', 'd')
>>> letters.__next__()
'a'
>>> letters.__next__()
'b'
>>> letters.__next__()
'c'
>>> letters.__next__()
'd'
>>> letters.__next__()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 12, in next
StopIteration
```
From a native python iterator to a nested stream
empty_stream = Stream(None, None, True)
empty_stream = Stream(None, None, True)

def iterator_to_stream(iterator):

From a native python iterator to a nested stream

empty_stream = Stream(None, None, True)

def iterator_to_stream(iterator):
    def streamify():

empty_stream = Stream(None, None, True)

def iterator_to_stream(iterator):
    def streamify():
        try:
empty_stream = Stream(None, None, True)

def iterator_to_stream(iterator):
    def streamify():
        try:
            first = iterator._next_()
empty_stream = Stream(None, None, True)

def iterator_to_stream(iterator):
    def streamify():
        try:
            first = iterator.__next__()
        return Stream(first, streamify)
        return Stream(first, streamify)
From a native python iterator to a nested stream

empty_stream = Stream(None, None, True)

def iterator_to_stream(iterator):
    def streamify():
        try:
            first = iterator.__next__()
            return Stream(first, streamify)
        except:
            return Stream(None, None, True)
empty_stream = Stream(None, None, True)

def iterator_to_stream(iterator):
    def streamify():
        try:
            first = iterator.__next__()
            return Stream(first, streamify)
        except:
            return empty_stream
From a native python iterator to a nested stream

empty_stream = Stream(None, None, True)

def iterator_to_stream(iterator):
    def streamify():
        try:
            first = iterator.__next__()
            return Stream(first, streamify)
        except:
            return empty_stream
    stream = streamify()
empty_stream = Stream(None, None, True)

def iterator_to_stream(iterator):
    def streamify():
        try:
            first = iterator.__next__()
            return Stream(first, streamify)
        except:
            return empty_stream
    stream = streamify()
    return stream

From a native python iterator to a nested stream

Monday, November 21, 2011
More support: for loops!
More support: for loops!

for item in obj:
do stuff
More support: for loops!

```
for item in obj:
  do stuff
```

“for” loops use iterators
More support: for loops!

“for” loops use iterators

- Step 1: get an iterator

for item in obj:
do stuff
More support: for loops!

```
for item in obj:
do stuff
```

“for” loops use iterators

- **Step 1: get an iterator**
  - `iterator = obj.__iter__()`
"for" loops use iterators

- Step 1: get an iterator
  - iterator = obj.__iter__()
- Step 2:
More support: for loops!

for item in obj:
do stuff

“for” loops use iterators

- Step 1: get an iterator
  - iterator = obj.__iter__()
- Step 2:
  - try iterator.__next__()
More support: for loops!

```
for item in obj:
    do stuff
```

“for” loops use iterators

1. Step 1: get an iterator
   - `iterator = obj.__iter__()`

2. Step 2:
   - `try iterator.__next__()`
   - Assign value to “item”
More support: for loops!

```
for item in obj:
    do stuff
```

“for” loops use iterators

- **Step 1:** get an iterator
  - `iterator = obj.__iter__()`
- **Step 2:**
  - `try iterator.__next__()`
  - assign value to “item”
  - `do body of loop`
More support: for loops!

```
for item in obj:
    do stuff
```

"for" loops use iterators

- Step 1: get an iterator
  - `iterator = obj.__iter__()`
- Step 2:
  - `try iterator.__next__()`
  - assign value to "item"
  - do body of loop
  - until StopIteration is raised
More support: for loops!

```
for item in obj:
    do stuff
```

“for” loops use iterators

- Step 1: get an iterator
  - `iterator = obj.__iter__()`
- Step 2:
  - `try iterator.__next__()`
  - assign value to “item”
  - do body of loop
  - until StopIteration is raised

```python
def for_each(sequence, function):
```
More support: for loops!

“for” loops use iterators

- Step 1: get an iterator
  - iterator = obj.__iter__()

- Step 2:
  - try iterator.__next__()
  - assign value to “item”
  - do body of loop
  - until StopIteration is raised

```python
def for_each(sequence, function):
    iterator = sequence.__iter__()
    for item in obj:
        do stuff
```

Monday, November 21, 2011
“for” loops use iterators

• Step 1: get an iterator
  • iterator = obj.__iter__()

• Step 2:
  • try iterator.__next__()
  • assign value to “item”
  • do body of loop
  • until StopIteration is raised

```python
def for_each(sequence, function):
    iterator = sequence.__iter__()
    try:
        for item in obj:
            do stuff
```
More support: for loops!

```
def for_each(sequence, function):
    iterator = sequence.__iter__()
    try:
        while True:
            for item in obj:
                do stuff
```

“for” loops use iterators

- **Step 1:** get an iterator
  - `iterator = obj.__iter__()`

- **Step 2:**
  - `try iterator.__next__()`
  - assign value to “item”
  - do body of loop
  - until StopIteration is raised
More support: for loops!

for item in obj:
    do stuff

“for” loops use iterators

- Step 1: get an iterator
  - iterator = obj.__iter__()

- Step 2:
  - try iterator.__next__()
  - assign value to “item”
  - do body of loop
  - until StopIteration is raised

```python
def for_each(sequence, function):
    iterator = sequence.__iter__()
    try:
        while True:
            element = iterator.__next__()
```
More support: for loops!

```
def for_each(sequence, function):
    iterator = sequence.__iter__()
    try:
        while True:
            element = iterator.__next__()
            function(element)
```
More support: for loops!

```
for item in obj:
    do stuff
```

"for" loops use iterators

- **Step 1:** get an iterator
  ```python
  iterator = obj.__iter__()
  ```
- **Step 2:**
  ```python
  try iterator.__next__()
  assign value to "item"
  do body of loop
  until StopIteration is raised
  ```

```python
def for_each(sequence, function):
    iterator = sequence.__iter__()
    try:
        while True:
            element = iterator.__next__()
            function(element)
    except StopIteration as e:
```
More support: for loops!

```
for item in obj:
    do stuff
```

“for” loops use iterators

- Step 1: get an iterator
  - `iterator = obj.__iter__()`

- Step 2:
  - `try iterator.__next__()`
  - assign value to “item”
  - do body of loop
  - until StopIteration is raised

```python
def for_each(sequence, function):
    iterator = sequence.__iter__()
    try:
        while True:
            element = iterator.__next__()
            function(element)
    except StopIteration as e:
        pass
```
More support: for loops!

“for” loops use iterators

- Step 1: get an iterator
  - iterator = obj.__iter__()

- Step 2:
  - try iterator.__next__()
  - assign value to “item”
  - do body of loop
  - until StopIteration is raised

```python
def for_each(sequence, function):
    iterator = sequence.__iter__()
    try:
        while True:
            element = iterator.__next__()
            function(element)
    except StopIteration as e:
        pass
```

live example
Even more support: generator functions
class Letters(object):
    def __init__(self, start, finish):
        self.current = start
        self.finish = finish

    def __next__(self):
        if self.current > self.finish:
            raise StopIteration
        result = self.current
        self.current = chr(ord(result)+1)
        return result

    def __iter__(self):
        return self
class Letters(object):
    def __init__(self, start, finish):
        self.current = start
        self.finish = finish

    def __next__(self):
        if self.current > self.finish:
            raise StopIteration
        result = self.current
        self.current = chr(ord(result)+1)
        return result

    def __iter__(self):
        return self
Even more support: generator functions

class Letters(object):
    def __init__(self, start, finish):
        self.current = start
        self.finish = finish

    def __next__(self):
        if self.current > self.finish:
            raise StopIteration
        result = self.current
        self.current = chr(ord(result)+1)
        return result

    def __iter__(self):
        return self

Generator version

def letters(start, finish):
    current = start
    while current <= finish:
        yield current
        current = chr(ord(current)+1)
Even more support: generator functions

class Letters(object):
    def __init__(self, start, finish):
        self.current = start
        self.finish = finish

    def __next__(self):
        if self.current > self.finish:
            raise StopIteration
        result = self.current
        self.current = chr(ord(result)+1)
        return result

    def __iter__(self):
        return self

Generator version
class Letters(object):
    def __init__(self, start, finish):
        self.current = start
        self.finish = finish

    def __next__(self):
        if self.current > self.finish:
            raise StopIteration
        result = self.current
        self.current = chr(ord(result)+1)
        return result

    def __iter__(self):
        return self

Generator version

def letters(start, finish):

class Letters(object):
    def __init__(self, start, finish):
        self.current = start
        self.finish = finish

    def __next__(self):
        if self.current > self.finish:
            raise StopIteration
        result = self.current
        self.current = chr(ord(result)+1)
        return result

    def __iter__(self):
        return self

Generator version

def letters(start, finish):
    current = start
Even more support: generator functions

class Letters(object):
    def __init__(self, start, finish):
        self.current = start
        self.finish = finish

    def __next__(self):
        if self.current > self.finish:
            raise StopIteration
        result = self.current
        self.current = chr(ord(result)+1)
        return result

    def __iter__(self):
        return self

Generator version

def letters(start, finish):
    current = start
    while current <= finish:
Even more support: generator functions

class Letters(object):
    def __init__(self, start, finish):
        self.current = start
        self.finish = finish

    def __next__(self):
        if self.current > self.finish:
            raise StopIteration
        result = self.current
        self.current = chr(ord(result) + 1)
        return result

    def __iter__(self):
        return self

Generator version

def letters(start, finish):
    current = start
    while current <= finish:
        yield current
**Even more support: generator functions**

```python
class Letters(object):
    def __init__(self, start, finish):
        self.current = start
        self.finish = finish

    def __next__(self):
        if self.current > self.finish:
            raise StopIteration
        result = self.current
        self.current = chr(ord(result)+1)
        return result

    def __iter__(self):
        return self

Generator version

def letters(start, finish):
    current = start
    while current <= finish:
        yield current
        current = chr(ord(current)+1)
```

---

*Monday, November 21, 2011*
Yield: a built-in flow-control statement

```python
def letters(start, finish):
    current = start
    while current <= finish:
        yield current
        current = chr(ord(current)+1)
```
Yield: a built-in flow-control statement

def letters(start, finish):
    current = start
    while current <= finish:
        yield current
        current = chr(ord(current)+1)
Yield: a built-in flow-control statement

def letters(start, finish):
    current = start
    while current <= finish:
        yield current
        current = chr(ord(current)+1)

Generator function.
When called, creates a Generator object
Yield: a built-in flow-control statement

def letters(start, finish):
    current = start
    while current <= finish:
        yield current
        current = chr(ord(current)+1)

Generator function. When called, creates a Generator object

>>> l = letters('a', 'd')
Yield: a built-in flow-control statement

```python
def letters(start, finish):
    current = start
    while current <= finish:
        yield current
        current = chr(ord(current)+1)
```

Generator function. When called, creates a Generator object

```python
>>> l = letters('a', 'd')
>>> l
```

Monday, November 21, 2011
Yield: a built-in flow-control statement

```python
def letters(start, finish):
    current = start
    while current <= finish:
        yield current
        current = chr(ord(current)+1)
```

Generator function. When called, creates a Generator object

```python
>>> l = letters('a', 'd')
>>> l
<generator instance at..>
```
Yield: a built-in flow-control statement

def letters(start, finish):
    current = start
    while current <= finish:
        yield current
        current = chr(ord(current)+1)

Generator function. When called, creates a Generator object

>>> l = letters('a', 'd')
>>> l
<generator instance at..>

Automatically creates:
Yield: a built-in flow-control statement

```python
def letters(start, finish):
    current = start
    while current <= finish:
        yield current
        current = chr(ord(current)+1)
```

Generator function. When called, creates a Generator object

```python
>>> l = letters('a', 'd')
>>> l
<generator instance at...>
Automatically creates:
l.__iter__()
```
Yield: a built-in flow-control statement

```python
def letters(start, finish):
    current = start
    while current <= finish:
        yield current
        current = chr(ord(current)+1)
```

Generator function. When called, creates a Generator object

```python
>>> l = letters('a', 'd')
>>> l
<generator instance at..>
```

Automatically creates:

- `l.__iter__()`
- `l.__next__()`

Monday, November 21, 2011
Yield: a built-in flow-control statement

```python
def letters(start, finish):
    current = start
    while current <= finish:
        yield current
        current = chr(ord(current)+1)
```

Does nothing at first

Generator function. When called, creates a Generator object

```python
>>> l = letters('a', 'd')
>>> l
<generator instance at..>
```

Automatically creates:

```
l.__iter__()
l.__next__()
```
Yield: a built-in flow-control statement

def letters(start, finish):
    current = start
    while current <= finish:
        yield current
        current = chr(ord(current)+1)

Does nothing at first

when __next__() is called, starts

Generator function. When called, creates a Generator object

>>> l = letters('a', 'd')
>>> l
<generator instance at...>

Automatically creates:
l.__iter__()
l.__next__()
Yield: a built-in flow-control statement

```python
def letters(start, finish):
    current = start
    while current <= finish:
        yield current
        current = chr(ord(current)+1)
```

Does nothing at first

when `__next__()` is called, starts

Goes through executing body of function

Generator function. When called, creates a Generator object

```python
>>> l = letters('a', 'd')
>>> l
<generator instance at...>
```

Automatically creates:

```
l.__iter__()
l.__next__()
```
Yield: a built-in flow-control statement

def letters(start, finish):
    current = start
    while current <= finish:
        yield current
        current = chr(ord(current)+1)

Does nothing at first
when __next__() is called, starts
Goes through executing body of function
Pauses at “yield” -- returns value

Generator function. When called, creates a Generator object

>>> l = letters('a', 'd')
>>> l
<generator instance at..>
Automatically creates:

l.__iter__()
l.__next__()
Yield: a built-in flow-control statement

```python
def letters(start, finish):
    current = start
    while current <= finish:
        yield current
        current = chr(ord(current)+1)
```

Does nothing at first

when `__next__()` is called, starts

Goes through executing body of function

Pauses at “yield” -- returns value

All local state is preserved

Generator function. When called, creates a Generator object

```python
>>> l = letters('a', 'd')
>>> l
<generator instance at...>
Automatically creates:

l.__iter__()
l.__next__()
```
Def letters(start, finish):
    current = start
    while current <= finish:
        yield current
        current = chr(ord(current)+1)

Does nothing at first

When __next__() is called, starts

Goes through executing body of function

Pauses at “yield” -- returns value

All local state is preserved

When __next__() is called, resumes.

Generator function.
When called, creates a Generator object

>>> l = letters('a', 'd')
>>> l
<generator instance at..>

Automatically creates:

l.__iter__()
l.__next__()
Yield: a built-in flow-control statement

```python
def letters(start, finish):
    current = start
    while current <= finish:
        yield current
        current = chr(ord(current)+1)
```

Does nothing at first

when `__next__()` is called, starts

Goes through executing body of function

Pauses at “yield” -- returns value

All local state is preserved

When `__next__()` is called, resumes.

Generator function. When called, creates a Generator object

```python
>>> l = letters('a', 'd')
>>> l
<generator instance at...>
```

Automatically creates:

- `l.__iter__()`
- `l.__next__()`
Iterators get used up
Iterators get used up

```python
>>> letters = Letters('a', 'd')
```
Iterators get used up

```python
>>> letters = Letters('a', 'd')
>>> letters.__next__()
```
Iterators get used up

```python
>>> letters = Letters('a', 'd')
>>> letters.__next__()
'a'
```
Iterators get used up

```python
>>> letters = Letters('a', 'd')
>>> letters.__next__()
'a'
>>> letters.__next__()
```

Monday, November 21, 2011
Iterators get used up

```python
>>> letters = Letters('a', 'd')
>>> letters.__next__()
'a'
>>> letters.__next__()
'b'
```
Iterators get used up

```python
>>> letters = Letters('a', 'd')
>>> letters.__next__()
'a'
>>> letters.__next__()
'b'
```
Iterators get used up

```python
>>> letters = Letters('a', 'd')
>>> letters.__next__()
'a'
>>> letters.__next__()
'b'
>>> letters.__next__()
'c'
```
Iterators get used up

```python
>>> letters = Letters('a', 'd')
```

```python
>>> letters.__next__()
'a'
```  

```python
>>> letters.__next__()
'b'
```  

```python
>>> letters.__next__()
'c'
```  

```python
>>> letters.__next__()
```
Iterators get used up

```python
>>> letters = Letters('a', 'd')
>>> letters.__next__()
'a'
>>> letters.__next__()
'b'
>>> letters.__next__()
'c'
>>> letters.__next__()
'd'
```
Iterators get used up

```python
>>> letters = Letters('a', 'd')
>>> letters.__next__()
'a'
>>> letters.__next__()
'b'
>>> letters.__next__()
'c'
>>> letters.__next__()
'd'
>>> letters.__next__()
```

Monday, November 21, 2011
Iterators get used up

```python
>>> letters = Letters('a', 'd')
>>> letters.__next__()
'a'
>>> letters.__next__()
'b'
>>> letters.__next__()
'c'
>>> letters.__next__()
'd'
>>> letters.__next__()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 12, in next
StopIteration
```
Iterators get used up

```python
>>> letters = Letters('a', 'd')
>>> letters.__next__()
'a'
>>> letters.__next__()
'b'
>>> letters.__next__()
'c'
>>> letters.__next__()
'd'
>>> letters.__next__()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 12, in next
StopIteration
```

The Iterator interface in python:

- **__iter__**
  - should return an iterator object
- **__next__**
  - should return a value OR
  - raise StopIteration
  - when end of sequence is reached
  - on all subsequent calls
Iterators get used up

```python
>>> letters = Letters('a', 'd')
>>> letters.__next__()
'a'
>>> letters.__next__()
'b'
>>> letters.__next__()
'c'
>>> letters.__next__()
'd'
>>> letters.__next__()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 12, in next
StopIteration
```

The Iterator interface in python:

- __iter__
  - should return an iterator object
- __next__
  - should return a value OR
  - raise StopIteration
  - when end of sequence is reached on all subsequent calls
Iterators get used up

```python
>>> letters = Letters('a', 'd')
>>> letters.__next__()
'a'
>>> letters.__next__()
'b'
>>> letters.__next__()
'c'
>>> letters.__next__()
'd'
>>> letters.__next__()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 12, in next
StopIteration

>>> letters.__next__()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 12, in next
StopIteration
```

The Iterator interface in python:

- **__iter__**
  - should return an iterator object
- **__next__**
  - should return a value OR
  - raise StopIteration
  - when end of sequence is reached
  - on all subsequent calls
Iterators get used up

>>> letters = Letters('a', 'd')
>>> letters.__next__()
'a'
>>> letters.__next__()
'b'
>>> letters.__next__()
'c'
>>> letters.__next__()
'd'
>>> letters.__next__()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    File "<stdin>", line 12, in next
StopIteration
>>> letters.__next__()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    File "<stdin>", line 12, in next
StopIteration
>>> letters.__next__()
Iterators get used up

The Iterator interface in python:

- __iter__
  - should return an iterator object
- __next__
  - should return a value OR raise StopIteration
  - when end of sequence is reached
  - on all subsequent calls

```python
>>> letters = Letters('a', 'd')
>>> letters.__next__()
'a'
>>> letters.__next__()
'b'
>>> letters.__next__()
'c'
>>> letters.__next__()
'd'
>>> letters.__next__()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 12, in next
StopIteration
>>> letters.__next__()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 12, in next
StopIteration
>>> letters.__next__()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 12, in next
StopIteration
```
Iterables -- new iterator for every `__iter__()`
Iterables -- new iterator for every __iter__()}

class Letters(object):
    def __init__(self, start, finish):
        self.current = start
        self.finish = finish

    def __next__(self):
        if self.current > self.finish:
            raise StopIteration
        result = self.current
        self.current = chr(ord(result)+1)
        return result

    def __iter__(self):
        return self
class Letters(object):
    def __init__(self, start, finish):
        self.current = start
        self.finish = finish

    def __next__(self):
        if self.current > self.finish:
            raise StopIteration
        result = self.current
        self.current = chr(ord(result)+1)
        return result

def __iter__(self):
    return self

class LetterSequence(object):
    pass
class Letters(object):
    def __init__(self, start, finish):
        self.current = start
        self.finish = finish

    def __next__(self):
        if self.current > self.finish:
            raise StopIteration
        result = self.current
        self.current = chr(ord(result)+1)
        return result

def __iter__(self):
    return self

class LetterSequence(object):
    def __init__(self, start, finish):
        self.current = start
        self.finish = finish

    def __next__(self):
        if self.current > self.finish:
            raise StopIteration
        result = self.current
        self.current = chr(ord(result)+1)
        return result

def __iter__(self):
    return self
Iterables -- new iterator for every `__iter__()`

```python
class Letters(object):
    def __init__(self, start, finish):
        self.current = start
        self.finish = finish

    def __next__(self):
        if self.current > self.finish:
            raise StopIteration
        result = self.current
        self.current = chr(ord(result)+1)
        return result

    def __iter__(self):
        return self
class LetterSequence(object):
    def __init__(self, start, finish):
        self.start = start

def __iter__(self):
    return self
```

Monday, November 21, 2011
class Letters(object):
    def __init__(self, start, finish):
        self.current = start
        self.finish = finish

    def __next__(self):
        if self.current > self.finish:
            raise StopIteration
        result = self.current
        self.current = chr(ord(result)+1)
        return result

    def __iter__(self):
        return self

class LetterSequence(object):
    def __init__(self, start, finish):
        self.start = start
        self.finish = finish

    def __next__(self):
        pass

    def __iter__(self):
        return self
class LetterSequence(object):
    def __init__(self, start, finish):
        self.start = start
        self.finish = finish

    def __iter__(self):
        return self

class Letters(object):
    def __init__(self, start, finish):
        self.current = start
        self.finish = finish

    def __next__(self):
        if self.current > self.finish:
            raise StopIteration
        result = self.current
        self.current = chr(ord(result)+1)
        return result

    def __iter__(self):
        return self
Iterables -- new iterator for every `__iter__()`

```python
class LetterSequence(object):
    def __init__(self, start, finish):
        self.start = start
        self.finish = finish

    def __iter__(self):
        return self.forward()

    def __next__(self):
        result = self.current
        self.current = chr(ord(result)+1)
        return result

def __iter__(self):
    return self
```

```python
class Letters(object):
    def __init__(self, start, finish):
        self.current = start
        self.finish = finish

    def __next__(self):
        if self.current > self.finish:
            raise StopIteration
        result = self.current
        self.current = chr(ord(result)+1)
        return result

    def __iter__(self):
        return self
```

Monday, November 21, 2011
**Iterables -- new iterator for every `__iter__()`**

class Letters(object):
    def __init__(self, start, finish):
        self.current = start
        self.finish = finish

    def __next__(self):
        if self.current > self.finish:
            raise StopIteration
        result = self.current
        self.current = chr(ord(result)+1)
        return result

    def __iter__(self):
        return self

class LetterSequence(object):
    def __init__(self, start, finish):
        self.start = start
        self.finish = finish

    def __iter__(self):
        return self.forward()
Iterables -- new iterator for every __iter__()
class LetterSequence(object):
    def __init__(self, start, finish):
        self.start = start
        self.finish = finish
    def __iter__(self):
        return self.forward()
    def forward(self):
        current = self.start
        if current < self.finish:
            raise StopIteration
        result = current
        self.current = chr(ord(result)+1)
        return result
    def __iter__(self):
        return self

class Letters(object):
    def __init__(self, start, finish):
        self.current = start
        self.finish = finish
    def __next__(self):
        if self.current > self.finish:
            raise StopIteration
        result = self.current
        self.current = chr(ord(result)+1)
        return result
    def __iter__(self):
        return self.forward()
class Letters(object):
    def __init__(self, start, finish):
        self.current = start
        self.finish = finish

    def __next__(self):
        if self.current > self.finish:
            raise StopIteration
        result = self.current
        self.current = chr(ord(result)+1)
        return result

    def __iter__(self):
        return self

class LetterSequence(object):
    def __init__(self, start, finish):
        self.start = start
        self.finish = finish

    def __iter__(self):
        return self.forward()

    def forward(self):
        current = self.start
        if current < self.finish:
            yield current
Iterables -- new iterator for every `__iter__()`

```python
class LetterSequence(object):
    def __init__(self, start, finish):
        self.start = start
        self.finish = finish

    def __iter__(self):
        return self.forward()

    def forward(self):
        current = self.start
        if current < self.finish:
            yield current
            current = chr(ord(current)+1)

class Letters(object):
    def __init__(self, start, finish):
        self.current = start
        self.finish = finish

    def __next__(self):
        if self.current > self.finish:
            raise StopIteration
        result = self.current
        self.current = chr(ord(result)+1)
        return result

    def __iter__(self):
        return self
```

Monday, November 21, 2011
Iterables -- new iterator for every __iter__()
class Letters(object):
    def __init__(self, start, finish):
        self.current = start
        self.finish = finish

    def __next__(self):
        if self.current > self.finish:
            raise StopIteration
        result = self.current
        self.current = chr(ord(result)+1)
        return result

    def __iter__(self):
        return self

class LetterSequence(object):
    def __init__(self, start, finish):
        self.start = start
        self.finish = finish

    def __iter__(self):
        return self

    def forward(self):
        current = self.start
        if current < self.finish:
            yield current
            current = chr(ord(current)+1)

a generator function

a new generator object every time
Iterables -- new iterator for every `__iter__()`

```python
class LetterSequence(object):
    def __init__(self, start, finish):
        self.start = start
        self.finish = finish

    def __iter__(self):
        return self.forward()

    def forward(self):
        current = self.start
        if current < self.finish:
            yield current
        current = chr(ord(current)+1)

class Letters(object):
    def __init__(self, start, finish):
        self.current = start
        self.finish = finish

    def __next__(self):
        if self.current > self.finish:
            raise StopIteration
        result = self.current
        self.current = chr(ord(result)+1)
        return result

    def __iter__(self):
        return self
```

Any questions?
Processing pipelines for sequential data

Next time