Python iterators

Reuven M. Lerner, PhD
reuven@lerner.co.il

Reversing lists

- You can reverse a list:

```
a = [1,2,3]

a.reverse()

print a
```

- Note that the reversed list is not returned
- The list is (silently) modified!

Reversing strings

- Things are more complicated with strings
- They are immutable — cannot be changed
- Two ways to reverse a string:

```
s = 'abc'

s[::-1]

reversed(s)
```

Reversing strings

- We're not really reversing the string!
- (It's immutable)

```
>>> print s[::-1]
cba
>>> print reversed(s)
<reversed object at 0x10049f590>
```

Huh?

- Where did our data go?
- And what is a “reversed object”?

Iterators

- reversed() returns an “iterator”
- In other words, you can loop over it

```
for element in reversed(alphabet):
    print element
```

- You can also use it as the input to list() or tuple()

Why would we want this?

- Lazy evaluation
  - Saves memory
  - Saves time
  - Why use it before you have to?

Example: Files

- So we can do this:

```
for line in f.readlines():
  print line
```

- But we can also do this:

```
for line in f:
  print line
```

Another example

```
>>> t = reversed((1,2,3))

>>> next(t)

3

... (several more times) ...

>>> next(t)

Traceback (most recent call last):

  File "<stdin>", line 1, in <module>

StopIteration
```

Iterators

- In Python, it's an iterator if:
  - it responds to next()
  - it raises StopIteration when it reaches the end
- If it's an iterator, it can be put in a loop, or a sequence type's constructor
- iter(x) is called to get an x's iterator — which might be x itself

Creating your own iterator

- Define a class
- Define __iter__, which returns self
- Define next, which returns the next item
- next should raise StopIteration at the end

Simple iterator

```
class MyIter(object):

    def __init__(self, data):

        self.data = data

        self.current_index = 0

    def __iter__(self):

        return self
```

Simple iterator, continued

```
def next(self):
    if self.current_index == len(self.data):
        raise StopIteration
    value = self.data[self.current_index]
    self.current_index = self.current_index + 1
    return value
```

Now I can use it!

```
>>> i = MyIter('abc')

>>> for x in i: print x
```

When would I use this?

- Container objects
- Objects that contain data over which you might want to iterate
- File and network abstractions

Iterating again

- The problem with returning "self" from __iter__ is that your object can only be iterated over a single time
- Currently each invocation of iter() returns the same object
- We often want to separate the data from the index tracked by the current iteration

A second class!

- Python's internal objects, when you invoke iter() on them, return a new instance of a secondary class
- This new instance, when you invoke iter() on it, returns itself
- But the object has its own index, and thus tracks the original object separately

Examples

```
>>> iter([1,2,3])
<listiterator object at 0x101206510>
>>> iter([1,2,3])
<listiterator object at 0x101206550>
>>> iter({})
<dictionary-keyiterator object at 0x1011edd60>
>>> iter({})
<dictionary-keyiterator object at 0x1011ede10>
>>> iter('')
<iterator object at 0x101206550>
>>> iter('')
<iterator object at 0x101206590>
```

Implementation

- iter(), on the main class, returns a new instance of the secondary class
- The secondary class gets a reference to the original, main class
- The secondary class has the index, while the original class has the data

```
class MyRealIterator(object):
    def __init__(self, myiter):
        self.myiter = myiter
        self.index = 0
    def next(self):
        if self.index >= len(self.myiter.data):
            raise StopIteration
        value = self.myiter.data[self.index]
        self.index += 1
        return value


class MyIter(object):
    def __init__(self, data)
        self.data = data
   def __iter__(self):
        return MyRealIterator(self)
```

itertools

- This module provides functions that make it easy to create, use iterators
- All take iterables, some take functions

itertools

- Return an iterable object

```
>>> x = itertools.chain([1,2,3], ['a', 'b', 'c'])

>>> type(x)

<type 'itertools.chain'>
```

chain

- Call next() on the first iterable, and then the second, and then... until all are used up

```
for i in chain([1, 2, 3], ['a', 'b', 'c']):

    print i
```

combinations

- Produces all of the combinations (i.e., order doesn't matter) for the iterable

```
>>> for item in itertools.combinations(range(4), 3):
...   print item
...
(0, 1, 2)
(0, 1, 3)
(0, 2, 3)
(1, 2, 3)
```

combinations_with_replacement

- Same as combinations, but items can be repeated in a row

```
for item in itertools.combinations_with_replacement(range(4), 3):
...   print item
(0, 0, 0)
(0, 0, 1)
(0, 0, 2)
(0, 0, 3)
(0, 1, 1)
(0, 1, 2)
...
```

compress

- Return all parallel items that return True

```
>>> list(itertools.compress(['a', 'b', 'c'],
                            (False, False, False)))
[]
>>> list(itertools.compress(['a', 'b', 'c'],
                            (False, True, False)))
['b']
```

More compress

```
>>> list(itertools.compress(a,
        [letter in 'aeiou' for letter in a]))
['a', 'e', 'i', 'o', 'u']
>>> list(itertools.compress(a,
        [letter in 'AEIOU' for letter in a]))
[]
```

count

- Count forever, starting at x, in steps of y

```
>>> x = itertools.count(1, 5)
>>> x.next()
1
>>> x.next()
6
>>> x.next()
11
>>> x.next()
16
```

cycle

```
>>> x = itertools.cycle(['a', 'b', 'c'])
>>> x.next()
'a'
>>> x.next()
'b'
>>> x.next()
'c'
>>> x.next()
'a'
```

dropwhile

```
>>> for item in itertools.dropwhile(lambda
x: x < 'b', ['a' 'b', 'c']):

...   print item

...

c
```

groupby

```
for key, item in itertools.groupby([1,2,2,2,2,2,3,4]):
  print "Key:{}".format(key)
  for subitem in item:
    print subitem
Key: 1
1
Key: 2
2
2
2
2
2
Key: 3
3
Key: 4
4
```

ifilter

```
iseven = itertools.ifilter(lambda x: x%2,
                           range(10))

for i in iseven:
    print i
```

ifilterfalse

```
isodd = itertools.ifilterfalse(
                                lambda x: not x%2,
                               range(10))

for i in isodd:
    print i
```

imap

```
>>> list(itertools.imap(
    lambda x, y: x*y, ([1,2], [5,6])))
[2, 30]
```

islice

```
for item in itertools.islice(alphabet, 3, 20, 3):
  print item
```

```
d
g
j
m
p
s
```

izip

```
>>> for item in itertools.izip(
          ['a', 'b', 'c'],
          [1,2,3],
          [10,20,30, 40]):
  print item

('a', 1, 10)
('b', 2, 20)
('c', 3, 30)
```

izip_longest

```
>>> for item in itertools.izip_longest(['a', 'b', 'c'],
[1,2,3], [10,20,30, 40]):
...   print item
...
('a', 1, 10)
('b', 2, 20)
('c', 3, 30)
(None, None, 40)
```

itertools.permutations

- Generate all of the permutations for an iterator!

```
for p in itertools.permutations(range(10)):
  print p
```

Output

```
...
(1, 6, 0, 2, 5, 9, 8, 3, 7, 4)
(1, 6, 0, 2, 5, 9, 8, 4, 3, 7)
(1, 6, 0, 2, 5, 9, 8, 4, 7, 3)
(1, 6, 0, 2, 5, 9, 8, 7, 3, 4)
(1, 6, 0, 2, 5, 9, 8, 7, 4, 3)
(1, 6, 0, 2, 7, 3, 4, 5, 8, 9)
(1, 6, 0, 2, 7, 3, 4, 5, 9, 8)
(1, 6, 0, 2, 7, 3, 4, 8, 5, 9)
...
```

product

```
>>> list(itertools.product(
            ['a', 'b', 'c'],
            ['d', 'e', 'f']))

[('a', 'd'), ('a', 'e'), ('a', 'f'), ('b',
'd'), ('b', 'e'), ('b', 'f'), ('c', 'd'),
('c', 'e'), ('c', 'f')]
```

repeat

```
>>> list(itertools.repeat('a', 5))

['a', 'a', 'a', 'a', 'a']
```

starmap

```
>>> list(itertools.starmap(lambda x, y: x*y,
                           ([1,2], [5,6])))
[2, 30]
```

takewhile

```
>>> for item in itertools.takewhile(lambda x: x < 'b',
                                    ['a', 'b', 'c']):
     print item

ab
```