Iterators and Generators

it ain't your gramps' loop any more

Python Iteration Protocol

- the old way: "for x in y: ..." used to mean:
  - at each leg of the loop:
    - y got indexed (by 0, 1, 2, ...) to get x
    - ending when y[i] raised IndexError
- the new way: "for x in y: ..." now means:
  - an iterator on y is gotten, "_it = iter(y)"
  - then, at each leg of the loop:
    - _it.next() is called to get x
    - ending when it raises StopIteration

Making (building) iterators

- the iter built-in
- a class with methods next and __iter__
  - def __iter__(self): return self
- generators
- generator expressions
- itertools
- other built-ins
  - enumerate, file, re.finditer

Using (consuming) iterators

- the for statement
- the in operator (membership test)
- many built-ins:
  - list, tuple, set, dict (when items are pairs)
  - enumerate
  - sum, max, min
- itertools module

The iter built-in

- iter(x) tries delegating to x.__iter__
  - if none, falls back to simulating old-way:

```
def seqiter(x):
    i = 0
    while True:
        try: yield x[i]
        except IndexError: break
        i += 1
```

generator vs class

```
def seqiter(x):
  i = 0
  while True:
    try: yield x[i]
    except IndexError: break
    i += 1

class Sqiter(object):
  def __init__(self, x):
    self.x, self.i = x, -1
  def __iter__(self): return self
  def next(self):
    if self.i is not None:
        self.i += 1
        try: return self.x[self.i]
        except IndexError: self.i = None
    raise StopIteration
```

iter's Sentinel form

- iter(f, sentinel) gives another common idiom

```
def sentiter(f, sentinel):
    while True:
        r = f()
        if r == sentinel: break
        yield r
```

- a think-outside-of-the-box usage...:

```
import random
randiter = iter(random.random, -1.0)
```

  - note: an infinite iterator!

Infinite iterators

```
class Fibonacci(object):
  def __init__(self):
    self.i, self.j = 1, 1
  def __iter__(self): return self
  def next(self):
    r = self.i
    self.i = self.j
    self.j = r + self.i
    return r

for f in Fibonacci():
    print f,
    if f>100: break
print
```

Peeking ahead

```
class peekable(object):
  def __init__(self, seq):
    self._it, self._c = iter(seq), queue()
  def __iter__(self): return self
  def _fillcache(self, n):
    while len(self._c)<n
      self._c.append(self._it.next())
  def next(self):
    self._fillcache(1)
    return self._c.popleft()
  def peek(self, n=None):
    self._fillcache(n or 1)
    if n is None: return self._c[0]
    return [self._c[i] for i in range(n)]
```

Generators

- a function containing keyword `yield...`:
  - is a generator
  - when called, returns an iterator x
  - first `x.next()` starts executing body
  - until it meets a `yield y`
    - then, it returns y and "freezes" state
    - further next calls "thaw" it & go on
  - or, meets a `return` (or falls off the end)
    - then, it raises `StopIteration`

Generator Expressions

- like list comprehensions, but...:
  - no [square] brackets around them
  - (round) parentheses instead
  - produce generator-iterators
    - suitable for looping, one item at a time
    - save memory
- [xpr for x in L] == list(xpr for x in L)

Make any loop a for

```
some, init = here
while whatevr(some):
  item, some = comput(init, some)
  if some < item: break
  dosome = processing(item)

def fancy(here, whatevr, comput):
  some, init = here
  while whatevr(some):
    item, some = comput(init, some)
    if some < item: break
    yield item
for item in fancy(here, whatevr, comput):
  dosome = processing(item)
```

Encapsulate recursion

```
def is_seq(x):
  return isinstance(x, (list, tuple))
def flat(tree, isntleaf=is_seq):
  for item in tree:
    if isntleaf(item):
      for leaf in flat(item, isntleaf):
        yield leaf
    else:
      yield item

for anitem in flat(sometree):
  dosome = processing(anitem)
```

Iterators: not just 'for's

```
def alternate(one, other):
  other = iter(other)
  for item in one:
    yield item
    yield other.next()

def alternate_with_hoist(one, other):
  next_other = iter(other).next
  for item in one:
    yield item
    yield next_other()
```

Infinite generators

```
def fibonacci():
  i, j = 1, 1
  while True:
    yield i
    i, j = j, i+j

for f in Fibonacci():
    print f,
    if f>100: break
print
```

A primes generator

```
def eratosthenes():
  D = {}  # 1st-found prime factor of k
  q = 1
  while True:
    p = D.pop(q, None)
    if p is None:
      yield q
      D[q*q] = q
    else:
      x = p + q
      while x in D:
        x += p
      D[x] = p
    q += 1
```

itertools

- module in standard library
- build iterators, mostly from other iterators, sequences, or other iterables:
  - izip, ifilter, islice, imap, starmap, tee, chain, takewhile, dropwhile, groupby
- build potentially infinite iterators:
  - count, cycle, repeat
- fast, fundamental, composable "building blocks" for many kinds of loops

itertools are *fast*

for example, say we need 1000 steps...:

```
for x in range(1000): pass
  144 microseconds per loop
for x in xrange(1000): pass
  123 microseconds per loop
for x in itertools.repeat(0, 1000): pass
  104 microseconds per loop
```

Truncating infinities

```
import itertools as it

# first (up to N) items, as a list:
primes_lst = list(it.islice(primes(), N))

# items < X (monotonically increasing):
list(it.takewhile(lambda x:x<X,primes()))

# stop when == X (excluded):
list(iter(primes().next, X))
```

"Striding" w/itertools

```
def strider1(seq, n):
  return [list(seq[i::n])
          for i in range(n)]

def strider5(seq, n):
  res = [ [] for r in it.repeat(0, n) ]
  riter = it.cycle(res)
  for item, subl in it.izip(seq, riter):
    subl.append(item)
  return res
```