Iterators and Generators

Alex Martelli
AB Strakt

This Tutorial’s Audience

- You have a good base knowledge of Python 2.* (say, 2.0 or 2.1)
- You may have no knowledge of iterators, generators, other 2.2 features
- You want to understand exactly how iterators and generators work in 2.2
- You want to learn how best to use them

This Tutorial's Style

- Meant to be *interactive*
- I need feedback on your background knowledge / how well you're following
- You need to ask questions, participate (else, you'd just read a paper!)
- So ***please*** do "interrupt" with questions & comments: it's what we're **here** for!

Iteration before 2.2

```
for item in container:
    any_for_body(item)
```

used to mean (the equivalent of):

```
_hidden_index = 0
while 1:
    try: item = container[_hidden_index]
    except IndexError: break
    _hidden_index = _hidden_index + 1
    any_for_body(item)
```

Iteration before 2.2: yes but...

- OK for *sequences* (which *want* to be randomly indexable, raise `IndexError` when index is out of bounds)
- kludge-ish for *streams* (which do *not* want to simulate random indexability)
- impossible for *mappings* (indexing means something quite different!)

Streams before 2.2

A typical idiom to allow iteration was…:

```
class SomeStream:
    def __init__(self):
        self.current = 0
    def __getitem__(self, index):
        if index != self.current:
            raise TypeError, “sequential only!”
        self.current = self.current + 1
        if self.isFinished():
            raise IndexError
        return self.generateNextItem()
```

Streams before 2.2: problems

- Python and the iterable class are both keeping iteration-indices...
- ...which they only use for error checks!
- no natural way to allow *nested* loops:

```
for x in container:
    for y in container:
        do_something(x, y)
```

Loops before 2.2

Given iterations' issues, one often coded:

```
while 1:
    item = next_iteration_value()
    if iteration_finished(item): break
    some_loop_body(item)
```

or even more clumsily (artificial state flags, code duplication...) just to avoid the `while 1: / break` construct

Iteration since 2.2

```
for item in container:
    any_for_body(item)
```

now means (the equivalent of):

```
_hidden_iterator = iter(container)
while True:
    try: item = _hidden_iterator.next()
    except StopIteration: break
    any_for_body(item)
```

New built-ins: `iter`, class `StopIteration`

2.2 Iterators

- no special iterator/iterable classes/types
- any x “is an iterator” if:
  - can call `x.next()` (`StopIteration` allowed)
  - ideally, `iter(x) is x` (see later)
- any y “is iterable” if it allows `iter(y)`:
  - must return “an iterator” (as above)
  - special method `y.__iter__()` (see later)
  - sequences are acceptable anyway

Other Languages' Iterators

- Ruby, Smalltalk: "other way 'round" (you pass loop body code *into* the iterator; in Python, the iterator yields items *out to* the loop body code)
- Sather: much richer/more special -- Python's iterators are normal objects (Sather's do let you do a lot more, but at a substantial price in complexity)

The new built-in `iter`

- `iter(x)` first tries calling special method `x.__iter__()`, if `x`'s type supplies it
- otherwise, if `x` is a sequence, `iter(x)` creates and returns a wrapper-iterator object that exactly simulates pre-2.2 behavior (see later)
- there's also a two-arguments form, `iter(callable, sentinel)` (see later)

Streams in 2.2

A typical idiom to allow iteration is now...:

```
class SomeStream:
    class _ItsIterator:
        def __init__(self, stream):
            self.stream = stream
        def __iter__(self):
            return self
        def next(self):
            if self.stream.isFinished():
                raise StopIteration
            return self.stream.generateNextItem()
    def __iter__(self):
        return self._ItsIterator(self)
```

Iterables and Iterators

- *Iterables* generally hold “general state” (e.g., a sequence hold items) but no per-iteration state (nor ref to iterators)
- *Iterators* generally hold **only** per-iteration state + reference to iterable
- all iterators are iterable, but...
- conceptual separation allows nested loops on iterable (*not* on an iterator!)

iter(sequence) IS RATHER LIKE…:

```
class SequenceIterator:
    def __init__(self, sequence):
        self.seq = sequence
        self.index = -1
    def __iter__(self):
        return self
    def next(self):
        self.index += 1
        try: return self.seq[self.index]
        except IndexError:
            raise StopIteration
```

iter(sequence) notes

- no implicit copy/snapshot of sequence!
- can’t alter sequence while looping on it
- Python does no implicit copies: if you need a copy, ask for it!

```
for item in mylist:
    mylist.append(item*item)    WRONG!
for item in mylist[:]:
    mylist.append(item*item)    OK!
```

iter(callable,sentinel) is like...:

```python
class SentinelIterator:
    def __init__(self, callable, sentin):
        self.callable = callable
        self.sentinel = sentin
    def __iter__(self):
        return self
    def next(self):
        result = self.callable()
        if result == self.sentinel:
            raise StopIteration
        return result
```

Loops in 2.2 [1]

```
while True:
    item = next_value()
    if item==sentinel: break
    some_loop_body(item)
```

becomes the much-smoother:

```
for item in iter(next_value, sentinel):
    some_loop_body(item)
```

What about general termination tests…:

```
    if iteration_finished(item): break
```

…?

Loops in 2.2 [2]

```
class TestingIterator:
    def __init__(self, callable, finish):
        self.callable = callable
        self.finish = finish
    def __iter__(self):
        return self
    def next(self):
        result = self.callable()
        if self.finish(result):
            raise StopIteration
        return result
```

Where can you use iterables

- basically, wherever you could use sequences in earlier Pythons:
  - `for` statements
  - `for` clauses of list comprehensions
  - built-ins: `map, zip, reduce, filter`, ...
  - type ctors: `list, tuple, dict` (new!)
  - operator `in` (e.g., `if x in y:` ...)
  - methods (`‘’.join(x), `...)

An aside: `dict`

- type (and thus also type-constructor) of dictionaries (much like `list`, `tuple`)
- accepts an optional mapping argument (for a dict `D`, `dict(D)` is like `D.copy()`)
- also accepts any iterable of *pairs* (two-items tuples) `(key, value)`
- “make a set”: `set=dict(zip(seq,seq))` (great for then doing many fast in tests)

Non-sequence built-in iterables

- `file`: iteration on a file object yields the *lines* one by one (must be text...!)
- `dict`: iteration on a dictionary yields the dictionary's *keys* one by one
- each dictionary `d` also has methods `d.iterkeys(), d.itervalues(), d.iteritems()`, which return iterators with the same contents as the lists `d.keys(), d.values(), d.items()`

Altering-while-iterating dicts

Dict methods `.keys()` &c do "snapshot":

```
for k in adict.keys():
    if blah(k): del adict[k]
```

But, iterators don't! So, you cannot code:

```
for k in adict:
    if blah(k): del adict[k]
```

However, no problem with:

```
for k in adict:
    if blah(k): adict[k] = 23
```

Need `StopIteration` ever come?

Not necessarily...:

```
class Ints:
    def __init__(self, start=0, step=1):
        self.current = start - step
        self.step = step
    def __iter__(self):
        return self
    def next(self):
        self.current += self.step
        return self.current
```

Such *unbounded* iterators are OK...

Unbounded iterators: yes **but...**

...*not* to be used just like this*:*

```
for x in Ints(7,12):
    print x
```

This would *never* stop! (`OverflowError` has gone, now OF promotes int→`long`)

```
for x in Ints(7,12):
    print x
    if x % 5 == 0: break
```

To use unbounded iterators, terminate the iteration separately and explicitly.

Is there no `prev` / pushback?

- No! That’s the flip side of iterators’ simplicity: they’re *very* lightweight
- Your own iterators can provide any extras you want (only your code will know how to use those extras)
- You can *wrap* arbitrary iterators to provide extras (for your code, only)

pushback iterator-wrapper

```
class PushbackWrapper:
    def __init__(self, it):
        self.it = iter(it)
        self.q = []
    def __iter__(self): return self
    def next(self):
        if self.q: return self.q.pop()
        else: return self.it.next()
    def pushback(self, back):
        self.q.append(back)
```

Generators

- enable by placing at start of module:

```
from __future__ import generators
```

- this transforms `yield` into a keyword
- a *generator* is any function whose body contains one or more statements:

  `yield` <expression>

- (may also have 0+ `return`, but not any `return` <expression>)

Generator mechanics [1]

- calling a generator G does *not* yet execute G’s body
- rather, it returns an iterator `I` wrapping an “execution frame” for G’s body, i.e.:
  - a reference to G’s body code
  - a set of G’s locals (including arguments)
  - “point-of-execution” (POE) (at code start)
- now, calling `I.next()`...

Generator mechanics [2]

- ...each call to `I.next()` continues G's body code from the last-saved "POE"
- execution proceeds until it encounters a `yield` <expr> statement
- then, it returns the value of <expr> as the result of `I.next()`
- execution suspends (locals and POE)

Generator mechanics [3]

- if, before a `yield` <expr> executes in a call to `I.next()`, a `return` executes, the iterator raises `StopIteration`
- “falling off the end” is like a `return`
- after a `StopIteration`, iterator `I` can “forget” the rest of its state (if `I.next()` is called again, `StopIteration` again)

Generators are compact...!

```
def Ints(start=0, step=1):
    while True:
        yield start
        start += step
def SentinelIter(callable, sentin):
    while True:
        result = callable()
        if result == sentin: return
        yield result
```

Generator equivalence rule

- change a (bounded) generator into an equivalent function with these rules...:
  - add (e.g.) `_list=[]` as the first statement
  - change every `yield <expr>` statement into `_list.append(expr)`
  - change every `return` statement (including function end), and `raise StopIteration`, into `return iter(_list)`
- takes more memory, gives same results
- (use: just to help understanding!)

Classic “tree-flatten” example

```
def flat(tree, scalarp):
    for node in tree:
        if scalarp(node): yield node
        else:
            for x in flat(node, scalarp):
                yield x
```

Note that defining “scalarp” is not trivial (strings are iterable, but we usually want to consider them as “scalar” anyway...)

De-generator’ed “tree-flatten”

```
def flat(tree, scalarp):
    _list = []
    for node in tree:
        if scalarp(node):
            _list.append(node)
        else:
            for x in flat(node, scalarp):
                _list.append(x)
    return iter(_list)
```

Aside: the scalarp predicate

```
def scalarp(obj):
    # deem string-like objects ‘scalar’
    try: obj+’’
    except: pass # not string-like, go on
    else: return True
    # now, ‘scalar’←→‘not iterable’
    try: iter(obj)
    except: return True
    else: return False
```

Iterators may be “lazy”

- an iterator may do “lazy” evaluation (AKA “just-in-time” evaluation)
- the “lazy” paradigm (AKA the “streams” paradigm) is central to functional languages such as Haskell
- iterator are a “foot in the door” for “lazy evaluation” in Python

Taking a(nother) Haskell idea

- fundamental stream operations, e.g.:

```
def take(N, stream):
    while N > 0:
        yield stream.next()
        N -= 1
```

- to “concretize” a bounded stream:

```
L = list(stream)  # built-in!
```

Sequence Idioms: spreading

```
import re
wds = re.compile(r’[\w-]+’).findall

def byWords(stream, wordsOf=wds):
    for line in stream:
        for w in wordsOf(line):
            yield w
```

Sequence Idioms: bunching

```
def byParagraphs(stream):
    p = []
    for line in stream:
        if line.isspace():
            if p: yield ‘’.join(p)
            p = []
        else: p.append(line)
    if p: yield ‘’.join(p)
```

Sorting a huge stream

- Classic algorithm "mergesort":
  - read the stream, a "chunk" at a time
    - sort chunk in-memory (Python list `sort`)
    - write sorted chunk to a temporary file
  - *merge* temporary files back to a stream
- very good fit for streams paradigm
- not all that lazy here (sort can't be...)

Merging sorted streams

```
def merge(streams):
    L = []
    for s in streams:
        try: L.append([s.next(), s.next])
        except StopIteration: pass
    while L:
        L.sort()
        yield L[0][0]
        try: L[0][0] = L[0][1]()
        except StopIteration: del L[0]
```

Lines-stream to sorted-pieces

```
def sortPieces(stream, N=1000*1000):
    while True:
        chunk = list(take(N, stream))
        if not chunk: return
        chunk.sort()
        tempFile = os.tmpfile()
        tempFile.writelines(L)
        tempFile.seek(0)
        del chunk
        yield tempFile
```

What if items are not lines...?

- just refactor with slight generalization:

```
def saveLines(lines):
    tempFile = os.tmpfile()
    tempFile.writelines(lines)
    tempFile.seek(0)
    return tempFile
def sortPieces(stream, saver, N):
    ...
    yield saver(chunk)
    del chunk
```

E.g., float items

```
def saveFloats(floats):
    tempFile = os.tmpfile()  # Win OK too
    array.array(‘d’, floats
        ).tofile(tempFile)
    tempFile.seek(0)
    return tempFile
def streamFloats(F, N=8*1000*1000):
    while True:
        buf = array.array(‘d’, F.read(N))
        if not buf: return
        for aFloat in buf: yield aFloat
```

Mergesort: putting it together

```
def mergesort(stream,
              saver=saveLines,
              N=1000*1000):
    pcs = sortPieces(stream, saver, N)
    for item in merge(pcs): yield item
```

E.g.:

```
m = mergesort(streamFloats(‘x.dat’,’rb’),
    saveFloats, 10*1000*1000)
for x in m: ...
```

One last little mint...

```
def makeSaver(typecode):
    def saver(data):
        tempFile = os.tmpfile()
        array.array(typecode, data
            ).tofile(tempFile)
        tempFile.seek(0)
        return tempFile
    return saver
saveFloats = makeSaver(‘d’)
saveUlongs = makeSaver(‘L’)
```