Chapter 4. Iterators and Generators

Iteration is one of Python's strongest features. At a high level, you might simply view iteration as a way to process items in a sequence. However, there is so much more that is possible, such as creating your own iterator objects, applying useful iteration patterns in the `itertools` module, making generator functions, and so forth. This chapter aims to address common problems involving iteration.

Manually Consuming an Iterator

Problem

You need to process items in an iterable, but for whatever reason, you can't or don't want to use a `for`loop.

Solution

To manually consume an iterable, use the `next()`function and write your code to catch the`StopIteration` exception. For example, this example manually reads lines from a file:

```
with open('/etc/passwd') as f:
    try:
        while True:
            line = next(f)
            print(line, end='')
    except StopIteration:
        pass
```

Normally, `StopIteration` is used to signal the end of iteration. However, if you're using `next()` manually (as shown), you can also instruct it to return a terminating value, such as `None`, instead. For example:

```
with open('/etc/passwd') as f:
     while True:
         line = next(f, None)
         if line is None:
             break
         print(line, end='')
```

Discussion

In most cases, the `for` statement is used to consume an iterable. However, every now and then, a problem calls for more precise control over the underlying iteration mechanism. Thus, it is useful to know what actually happens.

The following interactive example illustrates the basic mechanics of what happens during iteration:

```
>>> items = [1, 2, 3]
>>> # Get the iterator
>>> it = iter(items)     # Invokes items.__iter__()
>>> # Run the iterator
>>> next(it)             # Invokes it.__next__()
1
>>> next(it)
2
>>> next(it)
3
>>> next(it)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
>>>
```

Subsequent recipes in this chapter expand on iteration techniques, and knowledge of the basic iterator protocol is assumed. Be sure to tuck this first recipe away in your memory.

Delegating Iteration

Problem

You have built a custom container object that internally holds a list, tuple, or some other iterable. You would like to make iteration work with your new container.

Solution

Typically, all you need to do is define an `__iter__()`method that delegates iteration to the internally held container. For example:

```
class Node:
    def __init__(self, value):
        self._value = value
        self._children = []

    def __repr__(self):
        return 'Node({!r})'.format(self._value)

    def add_child(self, node):
        self._children.append(node)

    def __iter__(self):
        return iter(self._children)

# Example
if __name__ == '__main__':
    root = Node(0)
    child1 = Node(1)
    child2 = Node(2)
    root.add_child(child1)
    root.add_child(child2)
    for ch in root:
        print(ch)
    # Outputs Node(1), Node(2)
```

In this code, the `__iter__()` method simply forwards the iteration request to the internally held `_children`attribute.

Discussion

Python's iterator protocol requires `__iter__()` to return a special iterator object that implements a`__next__()` method to carry out the actual iteration. If all you are doing is iterating over the contents of another container, you don't really need to worry about the underlying details of how it works. All you need to do is to forward the iteration request along.

The use of the `iter()` function here is a bit of a shortcut that cleans up the code. `iter(s)` simply returns the underlying iterator by calling`s.__iter__()`, much in the same way that `len(s)`invokes `s.__len__()`.

Creating New Iteration Patterns with Generators

Problem

You want to implement a custom iteration pattern that's different than the usual built-in functions (e.g.,`range()`, `reversed()`, etc.).

Solution

If you want to implement a new kind of iteration pattern, define it using a generator function. Here's a generator that produces a range of floating-point numbers:

```
def frange(start, stop, increment):
    x = start
    while x < stop:
        yield x
        x += increment
```

To use such a function, you iterate over it using a `for`loop or use it with some other function that consumes an iterable (e.g., `sum()`, `list()`, etc.). For example:

```
>>> for n in frange(0, 4, 0.5):
...     print(n)
...
0
0.5
1.0
1.5
```

```
2.0
2.5
3.0
3.5
>>> list(frange(0, 1, 0.125))
[0, 0.125, 0.25, 0.375, 0.5, 0.625, 0.75, 0.875]
>>>
```

Discussion

The mere presence of the `yield` statement in a function turns it into a generator. Unlike a normal function, a generator only runs in response to iteration. Here's an experiment you can try to see the underlying mechanics of how such a function works:

```
>>> def countdown(n):
...     print('Starting to count from', n)
...     while n > 0:
...             yield n
...             n -= 1
...     print('Done!')
...

>>> # Create the generator, notice no output appears
>>> c = countdown(3)
>>> c
<generator object countdown at 0x1006a0af0>

>>> # Run to first yield and emit a value
>>> next(c)
Starting to count from 3
3

>>> # Run to the next yield
>>> next(c)
2

>>> # Run to next yield
>>> next(c)
1

>>> # Run to next yield (iteration stops)
>>> next(c)
Done!
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
>>>
```

The key feature is that a generator function only runs in response to "next" operations carried out in iteration. Once a generator function returns, iteration stops. However, the `for` statement that's usually used to iterate takes care of these details, so you don't normally need to worry about them.

Implementing the Iterator Protocol

Problem

You are building custom objects on which you would like to support iteration, but would like an easy way to implement the iterator protocol.

Solution

By far, the easiest way to implement iteration on an object is to use a generator function. In "Delegating Iteration", a `Node` class was presented for representing tree structures. Perhaps you want to implement an iterator that traverses nodes in a depth-first pattern. Here is how you could do it:

```
class Node:
    def __init__(self, value):
        self._value = value
        self._children = []

    def __repr__(self):
        return 'Node({!r})'.format(self._value)

    def add_child(self, node):
        self._children.append(node)

    def __iter__(self):
        return iter(self._children)

    def depth_first(self):
        yield self
        for c in self:
            yield from c.depth_first()

# Example
if __name__ == '__main__':
    root = Node(0)
    child1 = Node(1)
    child2 = Node(2)
    root.add_child(child1)
    root.add_child(child2)
    child1.add_child(Node(3))
    child1.add_child(Node(4))
    child2.add_child(Node(5))

    for ch in root.depth_first():
        print(ch)
    # Outputs Node(0), Node(1), Node(3), Node(4), Node(2), Node(5)
```

In this code, the `depth_first()` method is simple to read and describe. It first yields itself and then iterates over each child yielding the items produced by the child's `depth_first()` method (using `yield from`).

Discussion

Python's iterator protocol requires `__iter__()` to return a special iterator object that implements a `__next__()` operation and uses a `StopIteration`exception to signal completion. However, implementing such objects can often be a messy affair. For example, the following code shows an alternative implementation of the `depth_first()`method using an associated iterator class:

```
class Node:
    def __init__(self, value):
        self._value = value
        self._children = []

    def __repr__(self):
        return 'Node({!r})'.format(self._value)

    def add_child(self, other_node):
        self._children.append(other_node)

    def __iter__(self):
        return iter(self._children)

    def depth_first(self):
        return DepthFirstIterator(self)

class DepthFirstIterator(object):
    '''
    Depth-first traversal
    '''
    def __init__(self, start_node):
        self._node = start_node
        self._children_iter = None
        self._child_iter = None

    def __iter__(self):
        return self

    def __next__(self):
        # Return myself if just started; create an iterator for children
        if self._children_iter is None:
            self._children_iter = iter(self._node)
            return self._node

        # If processing a child, return its next item
        elif self._child_iter:
            try:
                nextchild = next(self._child_iter)
                return nextchild
            except StopIteration:
                self._child_iter = None
                return next(self)

        # Advance to the next child and start its iteration
        else:
            self._child_iter = next(self._children_iter).depth_first()
            return next(self)
```

The `DepthFirstIterator` class works in the same way as the generator version, but it's a mess because the iterator has to maintain a lot of complex state about where it is in the iteration process. Frankly, nobody likes to write mind-bending code like that. Define your iterator as a generator and be done with it.

Iterating in Reverse

Problem

You want to iterate in reverse over a sequence.

Solution

Use the built-in `reversed()` function. For example:

```
>>> a = [1, 2, 3, 4]
>>> for x in reversed(a):
...     print(x)
...
4
3
2
1
```

Reversed iteration only works if the object in question has a size that can be determined or if the object implements a `__reversed__()` special method. If neither of these can be satisfied, you'll have to convert the object into a list first. For example:

```
# Print a file backwards
f = open('somefile')
for line in reversed(list(f)):
    print(line, end='')
```

Be aware that turning an iterable into a list as shown could consume a lot of memory if it's large.

Discussion

Many programmers don't realize that reversed iteration can be customized on user-defined classes if they implement the `__reversed__()` method. For example:

```
class Countdown:
    def __init__(self, start):
        self.start = start

    # Forward iterator
    def __iter__(self):
        n = self.start
        while n > 0:
            yield n
            n -= 1

    # Reverse iterator
    def __reversed__(self):
        n = 1
        while n <= self.start:
            yield n
            n += 1
```

Defining a reversed iterator makes the code much more efficient, as it's no longer necessary to pull the data into a list and iterate in reverse on the list.

Defining Generator Functions with Extra State

Problem

You would like to define a generator function, but it involves extra state that you would like to expose to the user somehow.

Solution

If you want a generator to expose extra state to the user, don't forget that you can easily implement it as a class, putting the generator function code in the`__iter__()` method. For example:

```
from collections import deque

class linehistory:
    def __init__(self, lines, histlen=3):
        self.lines = lines
        self.history = deque(maxlen=histlen)

    def __iter__(self):
        for lineno, line in enumerate(self.lines,1):
            self.history.append((lineno, line))
            yield line

    def clear(self):
        self.history.clear()
```

To use this class, you would treat it like a normal generator function. However, since it creates an instance, you can access internal attributes, such as the `history` attribute or the `clear()` method. For example:

```
with open('somefile.txt') as f:
     lines = linehistory(f)
     for line in lines:
         if 'python' in line:
             for lineno, hline in lines.history:
                 print('{}:{}'.format(lineno, hline), end='')
```

Discussion

With generators, it is easy to fall into a trap of trying to do everything with functions alone. This can lead to rather complicated code if the generator function needs to interact with other parts of your program in unusual ways (exposing attributes, allowing control via method calls, etc.). If this is the case, just use a class definition, as shown. Defining your generator in the `__iter__()` method doesn't change anything about how you write your algorithm. The fact that it's part of a class makes it easy for you to provide attributes and methods for users to interact with.

One potential subtlety with the method shown is that it might require an extra step of calling `iter()` if you are going to drive iteration using a technique other than a `for` loop. For example:

```
>>> f = open('somefile.txt')
>>> lines = linehistory(f)
>>> next(lines)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'linehistory' object is not an iterator

>>> # Call iter() first, then start iterating
>>> it = iter(lines)
>>> next(it)
'hello world\n'
>>> next(it)
'this is a test\n'
>>>
```

Taking a Slice of an Iterator

Problem

You want to take a slice of data produced by an iterator, but the normal slicing operator doesn't work.

Solution

The `itertools.islice()` function is perfectly suited for taking slices of iterators and generators. For example:

```
>>> def count(n):
...     while True:
...             yield n
...             n += 1
...
>>> c = count(0)
>>> c[10:20]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'generator' object is not subscriptable

>>> # Now using islice()
>>> import itertools
>>> for x in itertools.islice(c, 10, 20):
...     print(x)
...
10
11
12
13
14
15
16
17
18
19
>>>
```

Discussion

Iterators and generators can't normally be sliced, because no information is known about their length (and they don't implement indexing). The result of`islice()` is an iterator that produces the desired slice items, but it does this by consuming and discarding all of the items up to the starting slice index. Further items are then produced by the `islice` object until the ending index has been reached.

It's important to emphasize that `islice()` will consume data on the supplied iterator. Since iterators can't be rewound, that is something to consider. If it's important to go back, you should probably just turn the data into a list first.

Skipping the First Part of an Iterable

Problem

You want to iterate over items in an iterable, but the first few items aren't of interest and you just want to discard them.

Solution

The `itertools` module has a few functions that can be used to address this task. The first is the`itertools.dropwhile()` function. To use it, you supply a function and an iterable. The returned iterator discards the first items in the sequence as long as the supplied function returns `True`. Afterward, the entirety of the sequence is produced.

To illustrate, suppose you are reading a file that starts with a series of comment lines. For example:

```
>>> with open('/etc/passwd') as f:
...     for line in f:
...         print(line, end='')
...
##
# User Database
#
# Note that this file is consulted directly only when the system is running
# in single-user mode.  At other times, this information is provided by
# Open Directory.
...
##
nobody:*:-2:-2:Unprivileged User:/var/empty:/usr/bin/false
root:*:0:0:System Administrator:/var/root:/bin/sh
...
>>>
```

If you want to skip all of the initial comment lines, here's one way to do it:

```
>>> from itertools import dropwhile
>>> with open('/etc/passwd') as f:
...     for line in dropwhile(lambda line: line.startswith('#'), f):
...          print(line, end='')
...
nobody:*:-2:-2:Unprivileged User:/var/empty:/usr/bin/false
root:*:0:0:System Administrator:/var/root:/bin/sh
```

```
...
>>>
```

This example is based on skipping the first items according to a test function. If you happen to know the exact number of items you want to skip, then you can use `itertools.islice()` instead. For example:

```
>>> from itertools import islice
>>> items = ['a', 'b', 'c', 1, 4, 10, 15]
>>> for x in islice(items, 3, None):
...     print(x)
...
1
4
10
15
>>>
```

In this example, the last `None` argument to `islice()`is required to indicate that you want everything*beyond* the first three items as opposed to only the first three items (e.g., a slice of `[3:]` as opposed to a slice of `[:3]`).

Discussion

The `dropwhile()` and `islice()` functions are mainly convenience functions that you can use to avoid writing rather messy code such as this:

```
with open('/etc/passwd') as f:
    # Skip over initial comments
    while True:
        line = next(f, '')
        if not line.startswith('#'):
            break

    # Process remaining lines
    while line:
        # Replace with useful processing
        print(line, end='')
        line = next(f, None)
```

Discarding the first part of an iterable is also slightly different than simply filtering all of it. For example, the first part of this recipe might be rewritten as follows:

```
with open('/etc/passwd') as f:
    lines = (line for line in f if not line.startswith('#'))
    for line in lines:
        print(line, end='')
```

This will obviously discard the comment lines at the start, but will also discard all such lines throughout the entire file. On the other hand, the solution only discards items until an item no longer satisfies the supplied test. After that, all subsequent items are returned with no filtering.

Last, but not least, it should be emphasized that this recipe works with all iterables, including those whose size can't be determined in advance. This includes generators, files, and similar kinds of objects.

Iterating Over All Possible Combinations or Permutations

Problem

You want to iterate over all of the possible combinations or permutations of a collection of items.

Solution

The `itertools` module provides three functions for this task. The first of these—`itertools.permutations()`—takes a collection of items and produces a sequence of tuples that rearranges all of the items into all possible permutations (i.e., it shuffles them into all possible configurations). For example:

```
>>> items = ['a', 'b', 'c']
>>> from itertools import permutations
>>> for p in permutations(items):
...     print(p)
...
('a', 'b', 'c')
('a', 'c', 'b')
('b', 'a', 'c')
('b', 'c', 'a')
('c', 'a', 'b')
('c', 'b', 'a')
>>>
```

If you want all permutations of a smaller length, you can give an optional length argument. For example:

```
>>> for p in permutations(items, 2):
...     print(p)
...
('a', 'b')
('a', 'c')
('b', 'a')
('b', 'c')
('c', 'a')
('c', 'b')
>>>
```

Use `itertools.combinations()` to produce a sequence of combinations of items taken from the input. For example:

```
>>> from itertools import combinations
>>> for c in combinations(items, 3):
...     print(c)
...
('a', 'b', 'c')
>>> for c in combinations(items, 2):
...     print(c)
```

```
...
('a', 'b')
('a', 'c')
('b', 'c')
>>> for c in combinations(items, 1):
...     print(c)
...
('a',)
('b',)
('c',)
>>>
```

For `combinations()`, the actual order of the elements is not considered. That is, the combination `('a', 'b')` is considered to be the same as `('b', 'a')`(which is not produced).

When producing combinations, chosen items are removed from the collection of possible candidates (i.e., if `'a'` has already been chosen, then it is removed from consideration). The`itertools.combinations_with_replacement()`function relaxes this, and allows the same item to be chosen more than once. For example:

```
>>> for c in combinations_with_replacement(items, 3):
...     print(c)
...
('a', 'a', 'a')
('a', 'a', 'b')
('a', 'a', 'c')
('a', 'b', 'b')
('a', 'b', 'c')
('a', 'c', 'c')
('b', 'b', 'b')
('b', 'b', 'c')
('b', 'c', 'c')
('c', 'c', 'c')
>>>
```

Discussion

This recipe demonstrates only some of the power found in the `itertools` module. Although you could certainly write code to produce permutations and combinations yourself, doing so would probably require more than a fair bit of thought. When faced with seemingly complicated iteration problems, it always pays to look at `itertools` first. If the problem is common, chances are a solution is already available.

Iterating Over the Index-Value Pairs of a Sequence

Problem

You want to iterate over a sequence, but would like to keep track of which element of the sequence is currently being processed.

Solution

The built-in `enumerate()` function handles this quite nicely:

```
>>> my_list = ['a', 'b', 'c']
>>> for idx, val in enumerate(my_list):
...     print(idx, val)
...
0 a
1 b
2 c
```

For printing output with canonical line numbers (where you typically start the numbering at 1 instead of 0), you can pass in a `start` argument:

```
>>> my_list = ['a', 'b', 'c']
>>> for idx, val in enumerate(my_list, 1):
...     print(idx, val)
...
1 a
2 b
3 c
```

This case is especially useful for tracking line numbers in files should you want to use a line number in an error message:

```
def parse_data(filename):
    with open(filename, 'rt') as f:
         for lineno, line in enumerate(f, 1):
             fields = line.split()
             try:
                 count = int(fields[1])
                 ...
             except ValueError as e:
                 print('Line {}: Parse error: {}'.format(lineno, e))
```

`enumerate()` can be handy for keeping track of the offset into a list for occurrences of certain values, for example. So, if you want to map words in a file to the lines in which they occur, it can easily be accomplished using `enumerate()` to map each word to the line offset in the file where it was found:

```
word_summary = defaultdict(list)

with open('myfile.txt', 'r') as f:
    lines = f.readlines()

for idx, line in enumerate(lines):
    # Create a list of words in current line
    words = [w.strip().lower() for w in line.split()]
    for word in words:
        word_summary[word].append(idx)
```

If you print `word_summary` after processing the file, it'll be a dictionary (a `defaultdict` to be precise), and it'll have a key for each word. The value for each word-key will be a list of line numbers that word occurred on. If the word occurred twice on a single line, that line number will be listed twice, making it possible to identify various simple metrics about the text.

Discussion

`enumerate()` is a nice shortcut for situations where you might be inclined to keep your own counter variable. You could write code like this:

```
lineno = 1
for line in f:
    # Process line
    ...
    lineno += 1
```

But it's usually much more elegant (and less error prone) to use `enumerate()` instead:

```
for lineno, line in enumerate(f):
    # Process line
    ...
```

The value returned by `enumerate()` is an instance of an `enumerate` object, which is an iterator that returns successive tuples consisting of a counter and the value returned by calling `next()` on the sequence you've passed in.

Although a minor point, it's worth mentioning that sometimes it is easy to get tripped up when applying`enumerate()` to a sequence of tuples that are also being unpacked. To do it, you have to write code like this:

```
data = [ (1, 2), (3, 4), (5, 6), (7, 8) ]

# Correct!
for n, (x, y) in enumerate(data):
    ...

# Error!
for n, x, y in enumerate(data):
    ...
```

Iterating Over Multiple Sequences Simultaneously

Problem

You want to iterate over the items contained in more than one sequence at a time.

Solution

To iterate over more than one sequence simultaneously, use the `zip()` function. For example:

```
>>> xpts = [1, 5, 4, 2, 10, 7]
>>> ypts = [101, 78, 37, 15, 62, 99]
>>> for x, y in zip(xpts, ypts):
...     print(x,y)
...
1 101
5 78
4 37
2 15
10 62
7 99
>>>
```

`zip(a, b)` works by creating an iterator that produces tuples `(x, y)` where `x` is taken from `a` and yis taken from `b`. Iteration stops whenever one of the input sequences is exhausted. Thus, the length of the iteration is the same as the length of the shortest input. For example:

```
>>> a = [1, 2, 3]
>>> b = ['w', 'x', 'y', 'z']
>>> for i in zip(a,b):
...     print(i)
...
(1, 'w')
(2, 'x')
(3, 'y')
>>>
```

If this behavior is not desired, use`itertools.zip_longest()` instead. For example:

```
>>> from itertools import zip_longest
>>> for i in zip_longest(a,b):
...     print(i)
...
(1, 'w')
(2, 'x')
(3, 'y')
(None, 'z')
>>> for i in zip_longest(a, b, fillvalue=0):
...     print(i)
...
(1, 'w')
(2, 'x')
(3, 'y')
(0, 'z')
>>>
```

Discussion

`zip()` is commonly used whenever you need to pair data together. For example, suppose you have a list of column headers and column values like this:

```
headers = ['name', 'shares', 'price']
```

```
values = ['ACME', 100, 490.1]
```

Using `zip()`, you can pair the values together to make a dictionary like this:

```
s = dict(zip(headers,values))
```

Alternatively, if you are trying to produce output, you can write code like this:

```
for name, val in zip(headers, values):
    print(name, '=', val)
```

It’s less common, but `zip()` can be passed more than two sequences as input. For this case, the resulting tuples have the same number of items in them as the number of input sequences. For example:

```
>>> a = [1, 2, 3]
>>> b = [10, 11, 12]
>>> c = ['x','y','z']
>>> for i in zip(a, b, c):
...     print(i)
...
(1, 10, 'x')
(2, 11, 'y')
(3, 12, 'z')
>>>
```

Last, but not least, it’s important to emphasize that`zip()` creates an iterator as a result. If you need the paired values stored in a list, use the `list()` function. For example:

```
>>> zip(a, b)
<zip object at 0x1007001b8>
>>> list(zip(a, b))
[(1, 10), (2, 11), (3, 12)]
>>>
```

Iterating on Items in Separate Containers

Problem

You need to perform the same operation on many objects, but the objects are contained in different containers, and you’d like to avoid nested loops without losing the readability of your code.

Solution

The `itertools.chain()` method can be used to simplify this task. It takes a list of iterables as input, and returns an iterator that effectively masks the fact that you’re really acting on multiple containers. To illustrate, consider this example:

```
>>> from itertools import chain
>>> a = [1, 2, 3, 4]
>>> b = ['x', 'y', 'z']
>>> for x in chain(a, b):
...     print(x)
...
1
2
3
4
x
y
z
>>>
```

A common use of `chain()` is in programs where you would like to perform certain operations on all of the items at once but the items are pooled into different working sets. For example:

```
# Various working sets of items
active_items = set()
inactive_items = set()

# Iterate over all items
for item in chain(active_items, inactive_items):
    # Process item
    ...
```

This solution is much more elegant than using two separate loops, as in the following:

```
for item in active_items:
    # Process item
    ...

for item in inactive_items:
    # Process item
    ...
```

Discussion

`itertools.chain()` accepts one or more iterables as arguments. It then works by creating an iterator that successively consumes and returns the items produced by each of the supplied iterables you provided. It's a subtle distinction, but `chain()` is more efficient than first combining the sequences and iterating. For example:

```
# Inefficent
for x in a + b:
    ...

# Better
for x in chain(a, b):
    ...
```

In the first case, the operation `a + b` creates an entirely new sequence and additionally requires `a`and `b` to be of the same type. `chain()` performs no such operation, so it's far more efficient with memory if the input sequences are large and it can be easily applied when the iterables in question are of different types.

Creating Data Processing Pipelines

Problem

You want to process data iteratively in the style of a data processing pipeline (similar to Unix pipes). For instance, you have a huge amount of data that needs to be processed, but it can't fit entirely into memory.

Solution

Generator functions are a good way to implement processing pipelines. To illustrate, suppose you have a huge directory of log files that you want to process:

```
foo/
   access-log-012007.gz
   access-log-022007.gz
   access-log-032007.gz
   ...
   access-log-012008
bar/
   access-log-092007.bz2
   ...
   access-log-022008
```

Suppose each file contains lines of data like this:

```
124.115.6.12 - - [10/Jul/2012:00:18:50 -0500] "GET /robots.txt ..." 200 71
210.212.209.67 - - [10/Jul/2012:00:18:51 -0500] "GET /ply/ ..." 200 11875
210.212.209.67 - - [10/Jul/2012:00:18:51 -0500] "GET /favicon.ico ..." 404 369
61.135.216.105 - - [10/Jul/2012:00:20:04 -0500] "GET /blog/atom.xml ..." 304 -
...
```

To process these files, you could define a collection of small generator functions that perform specific self-contained tasks. For example:

```
import os
import fnmatch
import gzip
import bz2
import re

def gen_find(filepat, top):
    '''
    Find all filenames in a directory tree that match a shell wildcard pattern
    '''
    for path, dirlist, filelist in os.walk(top):
        for name in fnmatch.filter(filelist, filepat):
            yield os.path.join(path,name)
```

```python
def gen_opener(filenames):
    '''
    Open a sequence of filenames one at a time producing a file object.
    The file is closed immediately when proceeding to the next iteration.
    '''
    for filename in filenames:
        if filename.endswith('.gz'):
            f = gzip.open(filename, 'rt')
        elif filename.endswith('.bz2'):
            f = bz2.open(filename, 'rt')
        else:
            f = open(filename, 'rt')
        yield f
        f.close()

def gen_concatenate(iterators):
    '''
    Chain a sequence of iterators together into a single sequence.
    '''
    for it in iterators:
        yield from it

def gen_grep(pattern, lines):
    '''
    Look for a regex pattern in a sequence of lines
    '''
    pat = re.compile(pattern)
    for line in lines:
        if pat.search(line):
            yield line
```

You can now easily stack these functions together to make a processing pipeline. For example, to find all log lines that contain the word *python*, you would just do this:

```python
lognames = gen_find('access-log*', 'www')
files = gen_opener(lognames)
lines = gen_concatenate(files)
pylines = gen_grep('(?i)python', lines)
for line in pylines:
    print(line)
```

If you want to extend the pipeline further, you can even feed the data in generator expressions. For example, this version finds the number of bytes transferred and sums the total:

```python
lognames = gen_find('access-log*', 'www')
files = gen_opener(lognames)
lines = gen_concatenate(files)
pylines = gen_grep('(?i)python', lines)
bytecolumn = (line.rsplit(None,1)[1] for line in pylines)
bytes = (int(x) for x in bytecolumn if x != '-')
print('Total', sum(bytes))
```

Discussion

Processing data in a pipelined manner works well for a wide variety of other problems, including parsing, reading from real-time data sources, periodic polling, and so on.

In understanding the code, it is important to grasp that the `yield` statement acts as a kind of data producer whereas a `for` loop acts as a data consumer. When the generators are stacked together, each`yield` feeds a single item of data to the next stage of the pipeline that is consuming it with iteration. In the last example, the `sum()` function is actually driving the entire program, pulling one item at a time out of the pipeline of generators.

One nice feature of this approach is that each generator function tends to be small and self-contained. As such, they are easy to write and maintain. In many cases, they are so general purpose that they can be reused in other contexts. The resulting code that glues the components together also tends to read like a simple recipe that is easily understood.

The memory efficiency of this approach can also not be overstated. The code shown would still work even if used on a massive directory of files. In fact, due to the iterative nature of the processing, very little memory would be used at all.

There is a bit of extreme subtlety involving the`gen_concatenate()` function. The purpose of this function is to concatenate input sequences together into one long sequence of lines. The`itertools.chain()` function performs a similar function, but requires that all of the chained iterables be specified as arguments. In the case of this particular recipe, doing that would involve a statement such as `lines = itertools.chain(*files)`, which would cause the`gen_opener()` generator to be fully consumed. Since that generator is producing a sequence of open files that are immediately closed in the next iteration step,`chain()` can't be used. The solution shown avoids this issue.

Also appearing in the `gen_concatenate()` function is the use of `yield from` to delegate to a subgenerator. The statement `yield from it` simply makes`gen_concatenate()` emit all of the values produced by the generator `it`. This is described further in"Flattening a Nested Sequence".

Last, but not least, it should be noted that a pipelined approach doesn't always work for every data handling problem. Sometimes you just need to work with all of the data at once. However, even in that case, using generator pipelines can be a way to logically break a problem down into a kind of workflow.

David Beazley has written extensively about these techniques in his "Generator Tricks for Systems Programmers" tutorial presentation. Consult that for even more examples.

Flattening a Nested Sequence

Problem

You have a nested sequence that you want to flatten into a single list of values.

Solution

This is easily solved by writing a recursive generator function involving a `yield from` statement. For example:

```
from collections import Iterable

def flatten(items, ignore_types=(str, bytes)):
    for x in items:
        if isinstance(x, Iterable) and not isinstance(x, ignore_types):
            yield from flatten(x)
        else:
            yield x

items = [1, 2, [3, 4, [5, 6], 7], 8]

# Produces 1 2 3 4 5 6 7 8
for x in flatten(items):
    print(x)
```

In the code, the `isinstance(x, Iterable)` simply checks to see if an item is iterable. If so, `yield from`is used to emit all of its values as a kind of subroutine. The end result is a single sequence of output with no nesting.

The extra argument `ignore_types` and the check for`not isinstance(x, ignore_types)` is there to prevent strings and bytes from being interpreted as iterables and expanded as individual characters. This allows nested lists of strings to work in the way that most people would expect. For example:

```
>>> items = ['Dave', 'Paula', ['Thomas', 'Lewis']]
>>> for x in flatten(items):
...     print(x)
...
Dave
Paula
Thomas
Lewis
>>>
```

Discussion

The `yield from` statement is a nice shortcut to use if you ever want to write generators that call other generators as subroutines. If you don't use it, you need to write code that uses an extra `for` loop. For example:

```
def flatten(items, ignore_types=(str, bytes)):
    for x in items:
        if isinstance(x, Iterable) and not isinstance(x, ignore_types):
            for i in flatten(x):
                yield i
        else:
            yield x
```

Although it's only a minor change, the `yield from`statement just feels better and leads to cleaner code.

As noted, the extra check for strings and bytes is there to prevent the expansion of those types into individual characters. If there are other types that you don't want expanded, you can supply a different value for the `ignore_types` argument.

Finally, it should be noted that `yield from` has a more important role in advanced programs involving coroutines and generator-based concurrency. See“Using Generators As an Alternative to Threads” for another example.

Iterating in Sorted Order Over Merged Sorted Iterables

Problem

You have a collection of sorted sequences and you want to iterate over a sorted sequence of them all merged together.

Solution

The `heapq.merge()` function does exactly what you want. For example:

```
>>> import heapq
>>> a = [1, 4, 7, 10]
>>> b = [2, 5, 6, 11]
>>> for c in heapq.merge(a, b):
...     print(c)
...
1
2
4
5
6
7
10
11
```

Discussion

The iterative nature of `heapq.merge` means that it never reads any of the supplied sequences all at once. This means that you can use it on very long sequences with very little overhead. For instance, here is an example of how you would merge two sorted files:

```
import heapq

with open('sorted_file_1', 'rt') as file1, \
     open('sorted_file_2') 'rt' as file2, \
     open('merged_file', 'wt') as outf:

    for line in heapq.merge(file1, file2):
        outf.write(line)
```

It’s important to emphasize that `heapq.merge()`requires that all of the input sequences already be sorted. In particular, it does not first read all of the data into a heap or do any preliminary sorting. Nor does it perform any kind of validation of the inputs to check if they meet the ordering requirements. Instead, it simply examines the set of items from the front of each input sequence and emits the smallest one found. A new item from the chosen sequence is then read, and the process repeats itself until all input sequences have been fully consumed.

Replacing Infinite while Loops with an Iterator

Problem

You have code that uses a `while` loop to iteratively process data because it involves a function or some kind of unusual test condition that doesn't fall into the usual iteration pattern.

Solution

A somewhat common scenario in programs involving I/O is to write code like this:

```
CHUNKSIZE = 8192

def reader(s):
    while True:
        data = s.recv(CHUNKSIZE)
        if data == b'':
            break
        process_data(data)
```

Such code can often be replaced using `iter()`, as follows:

```
def reader(s):
    for chunk in iter(lambda: s.recv(CHUNKSIZE), b''):
        process_data(data)
```

If you're a bit skeptical that it might work, you can try a similar example involving files. For example:

```
>>> import sys
>>> f = open('/etc/passwd')
>>> for chunk in iter(lambda: f.read(10), ''):
...     n = sys.stdout.write(chunk)
...
nobody:*:-2:-2:Unprivileged User:/var/empty:/usr/bin/false
root:*:0:0:System Administrator:/var/root:/bin/sh
daemon:*:1:1:System Services:/var/root:/usr/bin/false
_uucp:*:4:4:Unix to Unix Copy Protocol:/var/spool/uucp:/usr/sbin/uucico
...
>>>
```

Discussion

A little-known feature of the built-in `iter()` function is that it optionally accepts a zero-argument callable and sentinel (terminating) value as inputs. When used in this way, it creates an iterator that repeatedly calls the supplied callable over and over again until it returns the value given as a sentinel.

This particular approach works well with certain kinds of repeatedly called functions, such as those involving I/O. For example, if you want to read data in chunks from sockets or files, you usually have to repeatedly execute `read()` or `recv()` calls followed by an end-of-file test. This recipe simply takes these two features and combines them together into a single`iter()` call. The use of `lambda` in the solution is needed to create a callable that takes no arguments, yet still supplies the desired size argument to `recv()`or `read()`.