CHAPTER 1

Data Structures and Algorithms

Python provides a variety of useful built-in data structures, such as lists, sets, and dictionaries. For the most part, the use of these structures is straightforward. However, common questions concerning searching, sorting, ordering, and filtering often arise. Thus, the goal of this chapter is to discuss common data structures and algorithms involving data. In addition, treatment is given to the various data structures contained in the `collections` module.

1.1. Unpacking a Sequence into Separate Variables

Problem

You have an N-element tuple or sequence that you would like to unpack into a collection of N variables.

Solution

Any sequence (or iterable) can be unpacked into variables using a simple assignment operation. The only requirement is that the number of variables and structure match the sequence. For example:

```
>>> p = (4, 5)
>>> x, y = p
>>> x
4
>>> y
5
>>>

>>> data = [ 'ACME', 50, 91.1, (2012, 12, 21) ]
>>> name, shares, price, date = data
>>> name
```

```
'ACME'
>>> date
(2012, 12, 21)

>>> name, shares, price, (year, mon, day) = data
>>> name
'ACME'
>>> year
2012
>>> mon
12
>>> day
21
>>>
```

If there is a mismatch in the number of elements, you'll get an error. For example:

```
>>> p = (4, 5)
>>> x, y, z = p
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: need more than 2 values to unpack
>>>
```

Discussion

Unpacking actually works with any object that happens to be iterable, not just tuples or lists. This includes strings, files, iterators, and generators. For example:

```
>>> s = 'Hello'
>>> a, b, c, d, e = s
>>> a
'H'
>>> b
'e'
>>> e
'o'
>>>
```

When unpacking, you may sometimes want to discard certain values. Python has no special syntax for this, but you can often just pick a throwaway variable name for it. For example:

```
>>> data = [ 'ACME', 50, 91.1, (2012, 12, 21) ]
>>> _, shares, price, _ = data
>>> shares
50
>>> price
91.1
>>>
```

However, make sure that the variable name you pick isn't being used for something else already.

1.2. Unpacking Elements from Iterables of Arbitrary Length

Problem

You need to unpack N elements from an iterable, but the iterable may be longer than N elements, causing a "too many values to unpack" exception.

Solution

Python "star expressions" can be used to address this problem. For example, suppose you run a course and decide at the end of the semester that you're going to drop the first and last homework grades, and only average the rest of them. If there are only four assignments, maybe you simply unpack all four, but what if there are 24? A star expression makes it easy:

```
def drop_first_last(grades):
    first, *middle, last = grades
    return avg(middle)
```

As another use case, suppose you have user records that consist of a name and email address, followed by an arbitrary number of phone numbers. You could unpack the records like this:

```
>>> record = ('Dave', 'dave@example.com', '773-555-1212', '847-555-1212')
>>> name, email, *phone_numbers = user_record
>>> name
'Dave'
>>> email
'dave@example.com'
>>> phone_numbers
['773-555-1212', '847-555-1212']
>>>
```

It's worth noting that the `phone_numbers` variable will always be a list, regardless of how many phone numbers are unpacked (including none). Thus, any code that uses `phone_numbers` won't have to account for the possibility that it might not be a list or perform any kind of additional type checking.

The starred variable can also be the first one in the list. For example, say you have a sequence of values representing your company's sales figures for the last eight quarters. If you want to see how the most recent quarter stacks up to the average of the first seven, you could do something like this:

```
*trailing_qtrs, current_qtr = sales_record
trailing_avg = sum(trailing_qtrs) / len(trailing_qtrs)
return avg_comparison(trailing_avg, current_qtr)
```

Here's a view of the operation from the Python interpreter:

```
>>> *trailing, current = [10, 8, 7, 1, 9, 5, 10, 3]
>>> trailing
[10, 8, 7, 1, 9, 5, 10]
>>> current
3
```

Discussion

Extended iterable unpacking is tailor-made for unpacking iterables of unknown or arbitrary length. Oftentimes, these iterables have some known component or pattern in their construction (e.g. "everything after element 1 is a phone number"), and star unpacking lets the developer leverage those patterns easily instead of performing acrobatics to get at the relevant elements in the iterable.

It is worth noting that the star syntax can be especially useful when iterating over a sequence of tuples of varying length. For example, perhaps a sequence of tagged tuples:

```
records = [
     ('foo', 1, 2),
     ('bar', 'hello'),
     ('foo', 3, 4),
]

def do_foo(x, y):
    print('foo', x, y)

def do_bar(s):
    print('bar', s)

for tag, *args in records:
    if tag == 'foo':
        do_foo(*args)
    elif tag == 'bar':
        do_bar(*args)
```

Star unpacking can also be useful when combined with certain kinds of string processing operations, such as splitting. For example:

```
>>> line = 'nobody:*:-2:-2:Unprivileged User:/var/empty:/usr/bin/false'
>>> uname, *fields, homedir, sh = line.split(':')
>>> uname
'nobody'
>>> homedir
'/var/empty'
>>> sh
'/usr/bin/false'
>>>
```

Sometimes you might want to unpack values and throw them away. You can't just specify a bare * when unpacking, but you could use a common throwaway variable name, such as _ or `ign` (ignored). For example:

```
>>> record = ('ACME', 50, 123.45, (12, 18, 2012))
>>> name, *_, (*_, year) = record
>>> name
'ACME'
>>> year
2012
>>>
```

There is a certain similarity between star unpacking and list-processing features of various functional languages. For example, if you have a list, you can easily split it into head and tail components like this:

```
>>> items = [1, 10, 7, 4, 5, 9]
>>> head, *tail = items
>>> head
1
>>> tail
[10, 7, 4, 5, 9]
>>>
```

One could imagine writing functions that perform such splitting in order to carry out some kind of clever recursive algorithm. For example:

```
>>> def sum(items):
...     head, *tail = items
...     return head + sum(tail) if tail else head
...
>>> sum(items)
36
>>>
```

However, be aware that recursion really isn't a strong Python feature due to the inherent recursion limit. Thus, this last example might be nothing more than an academic curiosity in practice.

1.3. Keeping the Last N Items

Problem

You want to keep a limited history of the last few items seen during iteration or during some other kind of processing.

Solution

Keeping a limited history is a perfect use for a `collections.deque`. For example, the following code performs a simple text match on a sequence of lines and yields the matching line along with the previous N lines of context when found:

```
from collections import deque

def search(lines, pattern, history=5):
    previous_lines = deque(maxlen=history)
    for line in lines:
        if pattern in line:
            yield line, previous_lines
        previous_lines.append(line)

# Example use on a file
if __name__ == '__main__':
    with open('somefile.txt') as f:
        for line, prevlines in search(f, 'python', 5):
            for pline in prevlines:
                print(pline, end='')
            print(line, end='')
            print('-'*20)
```

Discussion

When writing code to search for items, it is common to use a generator function involving `yield`, as shown in this recipe's solution. This decouples the process of searching from the code that uses the results. If you're new to generators, see Recipe 4.3.

Using `deque(maxlen=N)` creates a fixed-sized queue. When new items are added and the queue is full, the oldest item is automatically removed. For example:

```
>>> q = deque(maxlen=3)
>>> q.append(1)
>>> q.append(2)
>>> q.append(3)
>>> q
deque([1, 2, 3], maxlen=3)
>>> q.append(4)
>>> q
deque([2, 3, 4], maxlen=3)
>>> q.append(5)
>>> q
deque([3, 4, 5], maxlen=3)
```

Although you could manually perform such operations on a list (e.g., appending, deleting, etc.), the queue solution is far more elegant and runs a lot faster.

More generally, a `deque` can be used whenever you need a simple queue structure. If you don't give it a maximum size, you get an unbounded queue that lets you append and pop items on either end. For example:

```
>>> q = deque()
>>> q.append(1)
>>> q.append(2)
>>> q.append(3)
>>> q
```

```
deque([1, 2, 3])
>>> q.appendleft(4)
>>> q
deque([4, 1, 2, 3])
>>> q.pop()
3
>>> q
deque([4, 1, 2])
>>> q.popleft()
4
```

Adding or popping items from either end of a queue has O(1) complexity. This is unlike a list where inserting or removing items from the front of the list is O(N).

1.4. Finding the Largest or Smallest N Items

Problem

You want to make a list of the largest or smallest N items in a collection.

Solution

The `heapq` module has two functions—`nlargest()` and `nsmallest()`—that do exactly what you want. For example:

```
import heapq

nums = [1, 8, 2, 23, 7, -4, 18, 23, 42, 37, 2]
print(heapq.nlargest(3, nums))  # Prints [42, 37, 23]
print(heapq.nsmallest(3, nums)) # Prints [-4, 1, 2]
```

Both functions also accept a key parameter that allows them to be used with more complicated data structures. For example:

```
portfolio = [
   {'name': 'IBM', 'shares': 100, 'price': 91.1},
   {'name': 'AAPL', 'shares': 50, 'price': 543.22},
   {'name': 'FB', 'shares': 200, 'price': 21.09},
   {'name': 'HPQ', 'shares': 35, 'price': 31.75},
   {'name': 'YHOO', 'shares': 45, 'price': 16.35},
   {'name': 'ACME', 'shares': 75, 'price': 115.65}
]

cheap = heapq.nsmallest(3, portfolio, key=lambda s: s['price'])
expensive = heapq.nlargest(3, portfolio, key=lambda s: s['price'])
```

Discussion

If you are looking for the N smallest or largest items and N is small compared to the overall size of the collection, these functions provide superior performance. Underneath

the covers, they work by first converting the data into a list where items are ordered as a heap. For example:

```
>>> nums = [1, 8, 2, 23, 7, -4, 18, 23, 42, 37, 2]
>>> import heapq
>>> heap = list(nums)
>>> heapq.heapify(heap)
>>> heap
[-4, 2, 1, 23, 7, 2, 18, 23, 42, 37, 8]
>>>
```

The most important feature of a heap is that `heap[0]` is always the smallest item. Moreover, subsequent items can be easily found using the `heapq.heappop()` method, which pops off the first item and replaces it with the next smallest item (an operation that requires O(log N) operations where N is the size of the heap). For example, to find the three smallest items, you would do this:

```
>>> heapq.heappop(heap)
-4
>>> heapq.heappop(heap)
1
>>> heapq.heappop(heap)
2
```

The `nlargest()` and `nsmallest()` functions are most appropriate if you are trying to find a relatively small number of items. If you are simply trying to find the single smallest or largest item (N=1), it is faster to use `min()` and `max()`. Similarly, if N is about the same size as the collection itself, it is usually faster to sort it first and take a slice (i.e., use `sorted(items)[:N]` or `sorted(items)[-N:]`). It should be noted that the actual implementation of `nlargest()` and `nsmallest()` is adaptive in how it operates and will carry out some of these optimizations on your behalf (e.g., using sorting if N is close to the same size as the input).

Although it's not necessary to use this recipe, the implementation of a heap is an interesting and worthwhile subject of study. This can usually be found in any decent book on algorithms and data structures. The documentation for the `heapq` module also discusses the underlying implementation details.

1.5. Implementing a Priority Queue

Problem

You want to implement a queue that sorts items by a given priority and always returns the item with the highest priority on each pop operation.

Solution

The following class uses the `heapq` module to implement a simple priority queue:

```
import heapq

class PriorityQueue:
    def __init__(self):
        self._queue = []
        self._index = 0

    def push(self, item, priority):
        heapq.heappush(self._queue, (-priority, self._index, item))
        self._index += 1

    def pop(self):
        return heapq.heappop(self._queue)[-1]
```

Here is an example of how it might be used:

```
>>> class Item:
...     def __init__(self, name):
...         self.name = name
...     def __repr__(self):
...         return 'Item({!r})'.format(self.name)
...
>>> q = PriorityQueue()
>>> q.push(Item('foo'), 1)
>>> q.push(Item('bar'), 5)
>>> q.push(Item('spam'), 4)
>>> q.push(Item('grok'), 1)
>>> q.pop()
Item('bar')
>>> q.pop()
Item('spam')
>>> q.pop()
Item('foo')
>>> q.pop()
Item('grok')
>>>
```

Observe how the first `pop()` operation returned the item with the highest priority. Also observe how the two items with the same priority (`foo` and `grok`) were returned in the same order in which they were inserted into the queue.

Discussion

The core of this recipe concerns the use of the `heapq` module. The functions `heapq.heappush()` and `heapq.heappop()` insert and remove items from a list `_queue` in a way such that the first item in the list has the smallest priority (as discussed in Recipe 1.4). The `heappop()` method always returns the "smallest" item, so that is the key to making the

queue pop the correct items. Moreover, since the push and pop operations have O(log N) complexity where N is the number of items in the heap, they are fairly efficient even for fairly large values of N.

In this recipe, the queue consists of tuples of the form `(-priority, index, item)`. The `priority` value is negated to get the queue to sort items from highest priority to lowest priority. This is opposite of the normal heap ordering, which sorts from lowest to highest value.

The role of the `index` variable is to properly order items with the same priority level. By keeping a constantly increasing index, the items will be sorted according to the order in which they were inserted. However, the index also serves an important role in making the comparison operations work for items that have the same priority level.

To elaborate on that, instances of `Item` in the example can't be ordered. For example:

```
>>> a = Item('foo')
>>> b = Item('bar')
>>> a < b
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unorderable types: Item() < Item()
>>>
```

If you make `(priority, item)` tuples, they can be compared as long as the priorities are different. However, if two tuples with equal priorities are compared, the comparison fails as before. For example:

```
>>> a = (1, Item('foo'))
>>> b = (5, Item('bar'))
>>> a < b
True
>>> c = (1, Item('grok'))
>>> a < c
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unorderable types: Item() < Item()
>>>
```

By introducing the extra index and making `(priority, index, item)` tuples, you avoid this problem entirely since no two tuples will ever have the same value for `index` (and Python never bothers to compare the remaining tuple values once the result of comparison can be determined):

```
>>> a = (1, 0, Item('foo'))
>>> b = (5, 1, Item('bar'))
>>> c = (1, 2, Item('grok'))
>>> a < b
True
>>> a < c
```

```
True
>>>
```

If you want to use this queue for communication between threads, you need to add appropriate locking and signaling. See Recipe 12.3 for an example of how to do this.

The documentation for the `heapq` module has further examples and discussion concerning the theory and implementation of heaps.

1.6. Mapping Keys to Multiple Values in a Dictionary

Problem

You want to make a dictionary that maps keys to more than one value (a so-called "multidict").

Solution

A dictionary is a mapping where each key is mapped to a single value. If you want to map keys to multiple values, you need to store the multiple values in another container such as a list or set. For example, you might make dictionaries like this:

```
d = {
   'a' : [1, 2, 3],
   'b' : [4, 5]
}

e = {
   'a' : {1, 2, 3},
   'b' : {4, 5}
}
```

The choice of whether or not to use lists or sets depends on intended use. Use a list if you want to preserve the insertion order of the items. Use a set if you want to eliminate duplicates (and don't care about the order).

To easily construct such dictionaries, you can use `defaultdict` in the `collections` module. A feature of `defaultdict` is that it automatically initializes the first value so you can simply focus on adding items. For example:

```
from collections import defaultdict

d = defaultdict(list)
d['a'].append(1)
d['a'].append(2)
d['b'].append(4)
...

d = defaultdict(set)
```

```
d['a'].add(1)
d['a'].add(2)
d['b'].add(4)
...
```

One caution with `defaultdict` is that it will automatically create dictionary entries for keys accessed later on (even if they aren't currently found in the dictionary). If you don't want this behavior, you might use `setdefault()` on an ordinary dictionary instead. For example:

```
d = {}    # A regular dictionary
d.setdefault('a', []).append(1)
d.setdefault('a', []).append(2)
d.setdefault('b', []).append(4)
...
```

However, many programmers find `setdefault()` to be a little unnatural—not to mention the fact that it always creates a new instance of the initial value on each invocation (the empty list `[]` in the example).

Discussion

In principle, constructing a multivalued dictionary is simple. However, initialization of the first value can be messy if you try to do it yourself. For example, you might have code that looks like this:

```
d = {}
for key, value in pairs:
    if key not in d:
         d[key] = []
    d[key].append(value)
```

Using a `defaultdict` simply leads to much cleaner code:

```
d = defaultdict(list)
for key, value in pairs:
    d[key].append(value)
```

This recipe is strongly related to the problem of grouping records together in data processing problems. See Recipe 1.15 for an example.

1.7. Keeping Dictionaries in Order

Problem

You want to create a dictionary, and you also want to control the order of items when iterating or serializing.

Solution

To control the order of items in a dictionary, you can use an `OrderedDict` from the `collections` module. It exactly preserves the original insertion order of data when iterating. For example:

```
from collections import OrderedDict

d = OrderedDict()
d['foo'] = 1
d['bar'] = 2
d['spam'] = 3
d['grok'] = 4

# Outputs "foo 1", "bar 2", "spam 3", "grok 4"
for key in d:
    print(key, d[key])
```

An `OrderedDict` can be particularly useful when you want to build a mapping that you may want to later serialize or encode into a different format. For example, if you want to precisely control the order of fields appearing in a JSON encoding, first building the data in an `OrderedDict` will do the trick:

```
>>> import json
>>> json.dumps(d)
'{"foo": 1, "bar": 2, "spam": 3, "grok": 4}'
>>>
```

Discussion

An `OrderedDict` internally maintains a doubly linked list that orders the keys according to insertion order. When a new item is first inserted, it is placed at the end of this list. Subsequent reassignment of an existing key doesn't change the order.

Be aware that the size of an `OrderedDict` is more than twice as large as a normal dictionary due to the extra linked list that's created. Thus, if you are going to build a data structure involving a large number of `OrderedDict` instances (e.g., reading 100,000 lines of a CSV file into a list of `OrderedDict` instances), you would need to study the requirements of your application to determine if the benefits of using an `OrderedDict` outweighed the extra memory overhead.

1.8. Calculating with Dictionaries

Problem

You want to perform various calculations (e.g., minimum value, maximum value, sorting, etc.) on a dictionary of data.

Solution

Consider a dictionary that maps stock names to prices:

```
prices = {
   'ACME': 45.23,
   'AAPL': 612.78,
   'IBM': 205.55,
   'HPQ': 37.20,
   'FB': 10.75
}
```

In order to perform useful calculations on the dictionary contents, it is often useful to invert the keys and values of the dictionary using `zip()`. For example, here is how to find the minimum and maximum price and stock name:

```
min_price = min(zip(prices.values(), prices.keys()))
# min_price is (10.75, 'FB')

max_price = max(zip(prices.values(), prices.keys()))
# max_price is (612.78, 'AAPL')
```

Similarly, to rank the data, use `zip()` with `sorted()`, as in the following:

```
prices_sorted = sorted(zip(prices.values(), prices.keys()))
# prices_sorted is [(10.75, 'FB'), (37.2, 'HPQ'),
#                   (45.23, 'ACME'), (205.55, 'IBM'),
#                   (612.78, 'AAPL')]
```

When doing these calculations, be aware that `zip()` creates an iterator that can only be consumed once. For example, the following code is an error:

```
prices_and_names = zip(prices.values(), prices.keys())
print(min(prices_and_names))   # OK
print(max(prices_and_names))   # ValueError: max() arg is an empty sequence
```

Discussion

If you try to perform common data reductions on a dictionary, you'll find that they only process the keys, not the values. For example:

```
min(prices)   # Returns 'AAPL'
max(prices)   # Returns 'IBM'
```

This is probably not what you want because you're actually trying to perform a calculation involving the dictionary values. You might try to fix this using the `values()` method of a dictionary:

```
min(prices.values())  # Returns 10.75
max(prices.values())  # Returns 612.78
```

Unfortunately, this is often not exactly what you want either. For example, you may want to know information about the corresponding keys (e.g., which stock has the lowest price?).

You can get the key corresponding to the min or max value if you supply a key function to `min()` and `max()`. For example:

```
min(prices, key=lambda k: prices[k])  # Returns 'FB'
max(prices, key=lambda k: prices[k])  # Returns 'AAPL'
```

However, to get the minimum value, you'll need to perform an extra lookup step. For example:

```
min_value = prices[min(prices, key=lambda k: prices[k])]
```

The solution involving `zip()` solves the problem by "inverting" the dictionary into a sequence of `(value, key)` pairs. When performing comparisons on such tuples, the `value` element is compared first, followed by the `key`. This gives you exactly the behavior that you want and allows reductions and sorting to be easily performed on the dictionary contents using a single statement.

It should be noted that in calculations involving `(value, key)` pairs, the key will be used to determine the result in instances where multiple entries happen to have the same value. For instance, in calculations such as `min()` and `max()`, the entry with the smallest or largest key will be returned if there happen to be duplicate values. For example:

```
>>> prices = { 'AAA' : 45.23, 'ZZZ': 45.23 }
>>> min(zip(prices.values(), prices.keys()))
(45.23, 'AAA')
>>> max(zip(prices.values(), prices.keys()))
(45.23, 'ZZZ')
>>>
```

1.9. Finding Commonalities in Two Dictionaries

Problem

You have two dictionaries and want to find out what they might have in common (same keys, same values, etc.).

Solution

Consider two dictionaries:

```
a = {
   'x' : 1,
   'y' : 2,
   'z' : 3
}
```

```
b = {
   'w' : 10,
   'x' : 11,
   'y' : 2
}
```

To find out what the two dictionaries have in common, simply perform common set operations using the `keys()` or `items()` methods. For example:

```
# Find keys in common
a.keys() & b.keys()   # { 'x', 'y' }

# Find keys in a that are not in b
a.keys() - b.keys()   # { 'z' }

# Find (key,value) pairs in common
a.items() & b.items() # { ('y', 2) }
```

These kinds of operations can also be used to alter or filter dictionary contents. For example, suppose you want to make a new dictionary with selected keys removed. Here is some sample code using a dictionary comprehension:

```
# Make a new dictionary with certain keys removed
c = {key:a[key] for key in a.keys() - {'z', 'w'}}
# c is {'x': 1, 'y': 2}
```

Discussion

A dictionary is a mapping between a set of keys and values. The `keys()` method of a dictionary returns a keys-view object that exposes the keys. A little-known feature of keys views is that they also support common set operations such as unions, intersections, and differences. Thus, if you need to perform common set operations with dictionary keys, you can often just use the keys-view objects directly without first converting them into a set.

The `items()` method of a dictionary returns an items-view object consisting of `(key, value)` pairs. This object supports similar set operations and can be used to perform operations such as finding out which key-value pairs two dictionaries have in common.

Although similar, the `values()` method of a dictionary does not support the set operations described in this recipe. In part, this is due to the fact that unlike keys, the items contained in a values view aren't guaranteed to be unique. This alone makes certain set operations of questionable utility. However, if you must perform such calculations, they can be accomplished by simply converting the values to a set first.

1.10. Removing Duplicates from a Sequence while Maintaining Order

Problem

You want to eliminate the duplicate values in a sequence, but preserve the order of the remaining items.

Solution

If the values in the sequence are hashable, the problem can be easily solved using a set and a generator. For example:

```
def dedupe(items):
    seen = set()
    for item in items:
        if item not in seen:
            yield item
            seen.add(item)
```

Here is an example of how to use your function:

```
>>> a = [1, 5, 2, 1, 9, 1, 5, 10]
>>> list(dedupe(a))
[1, 5, 2, 9, 10]
>>>
```

This only works if the items in the sequence are hashable. If you are trying to eliminate duplicates in a sequence of unhashable types (such as dicts), you can make a slight change to this recipe, as follows:

```
def dedupe(items, key=None):
    seen = set()
    for item in items:
        val = item if key is None else key(item)
        if val not in seen:
            yield item
            seen.add(val)
```

Here, the purpose of the `key` argument is to specify a function that converts sequence items into a hashable type for the purposes of duplicate detection. Here's how it works:

```
>>> a = [ {'x':1, 'y':2}, {'x':1, 'y':3}, {'x':1, 'y':2}, {'x':2, 'y':4}]
>>> list(dedupe(a, key=lambda d: (d['x'],d['y'])))
[{'x': 1, 'y': 2}, {'x': 1, 'y': 3}, {'x': 2, 'y': 4}]
>>> list(dedupe(a, key=lambda d: d['x']))
[{'x': 1, 'y': 2}, {'x': 2, 'y': 4}]
>>>
```

This latter solution also works nicely if you want to eliminate duplicates based on the value of a single field or attribute or a larger data structure.

Discussion

If all you want to do is eliminate duplicates, it is often easy enough to make a set. For example:

```
>>> a
[1, 5, 2, 1, 9, 1, 5, 10]
>>> set(a)
{1, 2, 10, 5, 9}
>>>
```

However, this approach doesn't preserve any kind of ordering. So, the resulting data will be scrambled afterward. The solution shown avoids this.

The use of a generator function in this recipe reflects the fact that you might want the function to be extremely general purpose—not necessarily tied directly to list processing. For example, if you want to read a file, eliminating duplicate lines, you could simply do this:

```
with open(somefile,'r') as f:
    for line in dedupe(f):
        ...
```

The specification of a `key` function mimics similar functionality in built-in functions such as `sorted()`, `min()`, and `max()`. For instance, see Recipes 1.8 and 1.13.

1.11. Naming a Slice

Problem

Your program has become an unreadable mess of hardcoded slice indices and you want to clean it up.

Solution

Suppose you have some code that is pulling specific data fields out of a record string with fixed fields (e.g., from a flat file or similar format):

```
######    0123456789012345678901234567890123456789012345678901234567890'
record = '.....................100          .......513.25     ..........'
cost = int(record[20:32]) * float(record[40:48])
```

Instead of doing that, why not name the slices like this?

```
SHARES = slice(20,32)
PRICE  = slice(40,48)

cost = int(record[SHARES]) * float(record[PRICE])
```

In the latter version, you avoid having a lot of mysterious hardcoded indices, and what you're doing becomes much clearer.

Discussion

As a general rule, writing code with a lot of hardcoded index values leads to a readability and maintenance mess. For example, if you come back to the code a year later, you'll look at it and wonder what you were thinking when you wrote it. The solution shown is simply a way of more clearly stating what your code is actually doing.

In general, the built-in `slice()` creates a slice object that can be used anywhere a slice is allowed. For example:

```
>>> items = [0, 1, 2, 3, 4, 5, 6]
>>> a = slice(2, 4)
>>> items[2:4]
[2, 3]
>>> items[a]
[2, 3]
>>> items[a] = [10,11]
>>> items
[0, 1, 10, 11, 4, 5, 6]
>>> del items[a]
>>> items
[0, 1, 4, 5, 6]
```

If you have a `slice` instance `s`, you can get more information about it by looking at its `s.start`, `s.stop`, and `s.step` attributes, respectively. For example:

```
>>> a = slice(10, 50, 2)
>>> a.start
10
>>> a.stop
50
>>> a.step
2
>>>
```

In addition, you can map a slice onto a sequence of a specific size by using its `indices(size)` method. This returns a tuple `(start, stop, step)` where all values have been suitably limited to fit within bounds (as to avoid `IndexError` exceptions when indexing). For example:

```
>>> s = 'HelloWorld'
>>> a.indices(len(s))
(5, 10, 2)
>>> for i in range(*a.indices(len(s))):
...     print(s[i])
...
W
r
```

```
d
>>>
```

1.12. Determining the Most Frequently Occurring Items in a Sequence

Problem

You have a sequence of items, and you'd like to determine the most frequently occurring items in the sequence.

Solution

The `collections.Counter` class is designed for just such a problem. It even comes with a handy `most_common()` method that will give you the answer.

To illustrate, let's say you have a list of words and you want to find out which words occur most often. Here's how you would do it:

```
words = [
   'look', 'into', 'my', 'eyes', 'look', 'into', 'my', 'eyes',
   'the', 'eyes', 'the', 'eyes', 'the', 'eyes', 'not', 'around', 'the',
   'eyes', "don't", 'look', 'around', 'the', 'eyes', 'look', 'into',
   'my', 'eyes', "you're", 'under'
]

from collections import Counter
word_counts = Counter(words)
top_three = word_counts.most_common(3)
print(top_three)
# Outputs [('eyes', 8), ('the', 5), ('look', 4)]
```

Discussion

As input, `Counter` objects can be fed any sequence of hashable input items. Under the covers, a `Counter` is a dictionary that maps the items to the number of occurrences. For example:

```
>>> word_counts['not']
1
>>> word_counts['eyes']
8
>>>
```

If you want to increment the count manually, simply use addition:

```
>>> morewords = ['why','are','you','not','looking','in','my','eyes']
>>> for word in morewords:
...     word_counts[word] += 1
```

```
...
>>> word_counts['eyes']
9
>>>
```

Or, alternatively, you could use the `update()` method:

```
>>> word_counts.update(morewords)
>>>
```

A little-known feature of `Counter` instances is that they can be easily combined using various mathematical operations. For example:

```
>>> a = Counter(words)
>>> b = Counter(morewords)
>>> a
Counter({'eyes': 8, 'the': 5, 'look': 4, 'into': 3, 'my': 3, 'around': 2,
         "you're": 1, "don't": 1, 'under': 1, 'not': 1})
>>> b
Counter({'eyes': 1, 'looking': 1, 'are': 1, 'in': 1, 'not': 1, 'you': 1,
         'my': 1, 'why': 1})

>>> # Combine counts
>>> c = a + b
>>> c
Counter({'eyes': 9, 'the': 5, 'look': 4, 'my': 4, 'into': 3, 'not': 2,
         'around': 2, "you're": 1, "don't": 1, 'in': 1, 'why': 1,
         'looking': 1, 'are': 1, 'under': 1, 'you': 1})

>>> # Subtract counts
>>> d = a - b
>>> d
Counter({'eyes': 7, 'the': 5, 'look': 4, 'into': 3, 'my': 2, 'around': 2,
         "you're": 1, "don't": 1, 'under': 1})
>>>
```

Needless to say, `Counter` objects are a tremendously useful tool for almost any kind of problem where you need to tabulate and count data. You should prefer this over manually written solutions involving dictionaries.

1.13. Sorting a List of Dictionaries by a Common Key

Problem

You have a list of dictionaries and you would like to sort the entries according to one or more of the dictionary values.

Solution

Sorting this type of structure is easy using the `operator` module's `itemgetter` function. Let's say you've queried a database table to get a listing of the members on your website, and you receive the following data structure in return:

```
rows = [
    {'fname': 'Brian', 'lname': 'Jones', 'uid': 1003},
    {'fname': 'David', 'lname': 'Beazley', 'uid': 1002},
    {'fname': 'John', 'lname': 'Cleese', 'uid': 1001},
    {'fname': 'Big', 'lname': 'Jones', 'uid': 1004}
]
```

It's fairly easy to output these rows ordered by any of the fields common to all of the dictionaries. For example:

```
from operator import itemgetter

rows_by_fname = sorted(rows, key=itemgetter('fname'))
rows_by_uid = sorted(rows, key=itemgetter('uid'))

print(rows_by_fname)
print(rows_by_uid)
```

The preceding code would output the following:

```
[{'fname': 'Big', 'uid': 1004, 'lname': 'Jones'},
 {'fname': 'Brian', 'uid': 1003, 'lname': 'Jones'},
 {'fname': 'David', 'uid': 1002, 'lname': 'Beazley'},
 {'fname': 'John', 'uid': 1001, 'lname': 'Cleese'}]

[{'fname': 'John', 'uid': 1001, 'lname': 'Cleese'},
 {'fname': 'David', 'uid': 1002, 'lname': 'Beazley'},
 {'fname': 'Brian', 'uid': 1003, 'lname': 'Jones'},
 {'fname': 'Big', 'uid': 1004, 'lname': 'Jones'}]
```

The `itemgetter()` function can also accept multiple keys. For example, this code

```
rows_by_lfname = sorted(rows, key=itemgetter('lname','fname'))
print(rows_by_lfname)
```

Produces output like this:

```
[{'fname': 'David', 'uid': 1002, 'lname': 'Beazley'},
 {'fname': 'John', 'uid': 1001, 'lname': 'Cleese'},
 {'fname': 'Big', 'uid': 1004, 'lname': 'Jones'},
 {'fname': 'Brian', 'uid': 1003, 'lname': 'Jones'}]
```

Discussion

In this example, `rows` is passed to the built-in `sorted()` function, which accepts a keyword argument `key`. This argument is expected to be a callable that accepts a single item

from `rows` as input and returns a value that will be used as the basis for sorting. The `itemgetter()` function creates just such a callable.

The `operator.itemgetter()` function takes as arguments the lookup indices used to extract the desired values from the records in `rows`. It can be a dictionary key name, a numeric list element, or any value that can be fed to an object's `__getitem__()` method. If you give multiple indices to `itemgetter()`, the callable it produces will return a tuple with all of the elements in it, and `sorted()` will order the output according to the sorted order of the tuples. This can be useful if you want to simultaneously sort on multiple fields (such as last and first name, as shown in the example).

The functionality of `itemgetter()` is sometimes replaced by `lambda` expressions. For example:

```
rows_by_fname = sorted(rows, key=lambda r: r['fname'])
rows_by_lfname = sorted(rows, key=lambda r: (r['lname'],r['fname']))
```

This solution often works just fine. However, the solution involving `itemgetter()` typically runs a bit faster. Thus, you might prefer it if performance is a concern.

Last, but not least, don't forget that the technique shown in this recipe can be applied to functions such as `min()` and `max()`. For example:

```
>>> min(rows, key=itemgetter('uid'))
{'fname': 'John', 'lname': 'Cleese', 'uid': 1001}
>>> max(rows, key=itemgetter('uid'))
{'fname': 'Big', 'lname': 'Jones', 'uid': 1004}
>>>
```

1.14. Sorting Objects Without Native Comparison Support

Problem

You want to sort objects of the same class, but they don't natively support comparison operations.

Solution

The built-in `sorted()` function takes a `key` argument that can be passed a callable that will return some value in the object that `sorted` will use to compare the objects. For example, if you have a sequence of `User` instances in your application, and you want to sort them by their `user_id` attribute, you would supply a callable that takes a `User` instance as input and returns the `user_id`. For example:

```
>>> class User:
...     def __init__(self, user_id):
...         self.user_id = user_id
```

```
...     def __repr__(self):
...         return 'User({})'.format(self.user_id)
...
>>> users = [User(23), User(3), User(99)]
>>> users
[User(23), User(3), User(99)]
>>> sorted(users, key=lambda u: u.user_id)
[User(3), User(23), User(99)]
>>>
```

Instead of using `lambda`, an alternative approach is to use `operator.attrgetter()`:

```
>>> from operator import attrgetter
>>> sorted(users, key=attrgetter('user_id'))
[User(3), User(23), User(99)]
>>>
```

Discussion

The choice of whether or not to use `lambda` or `attrgetter()` may be one of personal preference. However, `attrgetter()` is often a tad bit faster and also has the added feature of allowing multiple fields to be extracted simultaneously. This is analogous to the use of `operator.itemgetter()` for dictionaries (see Recipe 1.13). For example, if `User` instances also had a `first_name` and `last_name` attribute, you could perform a sort like this:

```
by_name = sorted(users, key=attrgetter('last_name', 'first_name'))
```

It is also worth noting that the technique used in this recipe can be applied to functions such as `min()` and `max()`. For example:

```
>>> min(users, key=attrgetter('user_id')
User(3)
>>> max(users, key=attrgetter('user_id')
User(99)
>>>
```

1.15. Grouping Records Together Based on a Field

Problem

You have a sequence of dictionaries or instances and you want to iterate over the data in groups based on the value of a particular field, such as date.

Solution

The `itertools.groupby()` function is particularly useful for grouping data together like this. To illustrate, suppose you have the following list of dictionaries:

```
rows = [
    {'address': '5412 N CLARK', 'date': '07/01/2012'},
    {'address': '5148 N CLARK', 'date': '07/04/2012'},
    {'address': '5800 E 58TH', 'date': '07/02/2012'},
    {'address': '2122 N CLARK', 'date': '07/03/2012'},
    {'address': '5645 N RAVENSWOOD', 'date': '07/02/2012'},
    {'address': '1060 W ADDISON', 'date': '07/02/2012'},
    {'address': '4801 N BROADWAY', 'date': '07/01/2012'},
    {'address': '1039 W GRANVILLE', 'date': '07/04/2012'},
]
```

Now suppose you want to iterate over the data in chunks grouped by date. To do it, first sort by the desired field (in this case, date) and then use `itertools.groupby()`:

```
from operator import itemgetter
from itertools import groupby

# Sort by the desired field first
rows.sort(key=itemgetter('date'))

# Iterate in groups
for date, items in groupby(rows, key=itemgetter('date')):
    print(date)
    for i in items:
        print('    ', i)
```

This produces the following output:

```
07/01/2012
     {'date': '07/01/2012', 'address': '5412 N CLARK'}
     {'date': '07/01/2012', 'address': '4801 N BROADWAY'}
07/02/2012
     {'date': '07/02/2012', 'address': '5800 E 58TH'}
     {'date': '07/02/2012', 'address': '5645 N RAVENSWOOD'}
     {'date': '07/02/2012', 'address': '1060 W ADDISON'}
07/03/2012
     {'date': '07/03/2012', 'address': '2122 N CLARK'}
07/04/2012
     {'date': '07/04/2012', 'address': '5148 N CLARK'}
     {'date': '07/04/2012', 'address': '1039 W GRANVILLE'}
```

Discussion

The `groupby()` function works by scanning a sequence and finding sequential "runs" of identical values (or values returned by the given key function). On each iteration, it returns the value along with an iterator that produces all of the items in a group with the same value.

An important preliminary step is sorting the data according to the field of interest. Since `groupby()` only examines consecutive items, failing to sort first won't group the records as you want.

If your goal is to simply group the data together by dates into a large data structure that allows random access, you may have better luck using `defaultdict()` to build a multidict, as described in Recipe 1.6. For example:

```
from collections import defaultdict
rows_by_date = defaultdict(list)
for row in rows:
    rows_by_date[row['date']].append(row)
```

This allows the records for each date to be accessed easily like this:

```
>>> for r in rows_by_date['07/01/2012']:
...     print(r)
...
{'date': '07/01/2012', 'address': '5412 N CLARK'}
{'date': '07/01/2012', 'address': '4801 N BROADWAY'}
>>>
```

For this latter example, it's not necessary to sort the records first. Thus, if memory is no concern, it may be faster to do this than to first sort the records and iterate using `groupby()`.

1.16. Filtering Sequence Elements

Problem

You have data inside of a sequence, and need to extract values or reduce the sequence using some criteria.

Solution

The easiest way to filter sequence data is often to use a list comprehension. For example:

```
>>> mylist = [1, 4, -5, 10, -7, 2, 3, -1]
>>> [n for n in mylist if n > 0]
[1, 4, 10, 2, 3]
>>> [n for n in mylist if n < 0]
[-5, -7, -1]
>>>
```

One potential downside of using a list comprehension is that it might produce a large result if the original input is large. If this is a concern, you can use generator expressions to produce the filtered values iteratively. For example:

```
>>> pos = (n for n in mylist if n > 0)
>>> pos
<generator object <genexpr> at 0x1006a0eb0>
>>> for x in pos:
...     print(x)
...
```

```
1
4
10
2
3
>>>
```

Sometimes, the filtering criteria cannot be easily expressed in a list comprehension or generator expression. For example, suppose that the filtering process involves exception handling or some other complicated detail. For this, put the filtering code into its own function and use the built-in `filter()` function. For example:

```
values = ['1', '2', '-3', '-', '4', 'N/A', '5']

def is_int(val):
    try:
        x = int(val)
        return True
    except ValueError:
        return False

ivals = list(filter(is_int, values))
print(ivals)
# Outputs ['1', '2', '-3', '4', '5']
```

`filter()` creates an iterator, so if you want to create a list of results, make sure you also use `list()` as shown.

Discussion

List comprehensions and generator expressions are often the easiest and most straightforward ways to filter simple data. They also have the added power to transform the data at the same time. For example:

```
>>> mylist = [1, 4, -5, 10, -7, 2, 3, -1]
>>> import math
>>> [math.sqrt(n) for n in mylist if n > 0]
[1.0, 2.0, 3.1622776601683795, 1.4142135623730951, 1.7320508075688772]
>>>
```

One variation on filtering involves replacing the values that don't meet the criteria with a new value instead of discarding them. For example, perhaps instead of just finding positive values, you want to also clip bad values to fit within a specified range. This is often easily accomplished by moving the filter criterion into a conditional expression like this:

```
>>> clip_neg = [n if n > 0 else 0 for n in mylist]
>>> clip_neg
[1, 4, 0, 10, 0, 2, 3, 0]
>>> clip_pos = [n if n < 0 else 0 for n in mylist]
>>> clip_pos
```

```
[0, 0, -5, 0, -7, 0, 0, -1]
>>>
```

Another notable filtering tool is `itertools.compress()`, which takes an iterable and an accompanying Boolean selector sequence as input. As output, it gives you all of the items in the iterable where the corresponding element in the selector is `True`. This can be useful if you're trying to apply the results of filtering one sequence to another related sequence. For example, suppose you have the following two columns of data:

```
addresses = [
    '5412 N CLARK',
    '5148 N CLARK',
    '5800 E 58TH',
    '2122 N CLARK'
    '5645 N RAVENSWOOD',
    '1060 W ADDISON',
    '4801 N BROADWAY',
    '1039 W GRANVILLE',
]

counts = [ 0, 3, 10, 4, 1, 7, 6, 1]
```

Now suppose you want to make a list of all addresses where the corresponding count value was greater than 5. Here's how you could do it:

```
>>> from itertools import compress
>>> more5 = [n > 5 for n in counts]
>>> more5
[False, False, True, False, False, True, True, False]
>>> list(compress(addresses, more5))
['5800 E 58TH', '4801 N BROADWAY', '1039 W GRANVILLE']
>>>
```

The key here is to first create a sequence of Booleans that indicates which elements satisfy the desired condition. The `compress()` function then picks out the items corresponding to `True` values.

Like `filter()`, `compress()` normally returns an iterator. Thus, you need to use `list()` to turn the results into a list if desired.

1.17. Extracting a Subset of a Dictionary

Problem

You want to make a dictionary that is a subset of another dictionary.

Solution

This is easily accomplished using a dictionary comprehension. For example:

```
prices = {
   'ACME': 45.23,
   'AAPL': 612.78,
   'IBM': 205.55,
   'HPQ': 37.20,
   'FB': 10.75
}

# Make a dictionary of all prices over 200
p1 = { key:value for key, value in prices.items() if value > 200 }

# Make a dictionary of tech stocks
tech_names = { 'AAPL', 'IBM', 'HPQ', 'MSFT' }
p2 = { key:value for key,value in prices.items() if key in tech_names }
```

Discussion

Much of what can be accomplished with a dictionary comprehension might also be done by creating a sequence of tuples and passing them to the `dict()` function. For example:

```
p1 = dict((key, value) for key, value in prices.items() if value > 200)
```

However, the dictionary comprehension solution is a bit clearer and actually runs quite a bit faster (over twice as fast when tested on the `prices` dictionary used in the example).

Sometimes there are multiple ways of accomplishing the same thing. For instance, the second example could be rewritten as:

```
# Make a dictionary of tech stocks
tech_names = { 'AAPL', 'IBM', 'HPQ', 'MSFT' }
p2 = { key:prices[key] for key in prices.keys() & tech_names }
```

However, a timing study reveals that this solution is almost 1.6 times slower than the first solution. If performance matters, it usually pays to spend a bit of time studying it. See Recipe 14.13 for specific information about timing and profiling.

1.18. Mapping Names to Sequence Elements

Problem

You have code that accesses list or tuple elements by position, but this makes the code somewhat difficult to read at times. You'd also like to be less dependent on position in the structure, by accessing the elements by name.

Solution

`collections.namedtuple()` provides these benefits, while adding minimal overhead over using a normal tuple object. `collections.namedtuple()` is actually a factory method that returns a subclass of the standard Python `tuple` type. You feed it a type name, and the fields it should have, and it returns a class that you can instantiate, passing in values for the fields you've defined, and so on. For example:

```
>>> from collections import namedtuple
>>> Subscriber = namedtuple('Subscriber', ['addr', 'joined'])
>>> sub = Subscriber('jonesy@example.com', '2012-10-19')
>>> sub
Subscriber(addr='jonesy@example.com', joined='2012-10-19')
>>> sub.addr
'jonesy@example.com'
>>> sub.joined
'2012-10-19'
>>>
```

Although an instance of a `namedtuple` looks like a normal class instance, it is interchangeable with a tuple and supports all of the usual tuple operations such as indexing and unpacking. For example:

```
>>> len(sub)
2
>>> addr, joined = sub
>>> addr
'jonesy@example.com'
>>> joined
'2012-10-19'
>>>
```

A major use case for named tuples is decoupling your code from the position of the elements it manipulates. So, if you get back a large list of tuples from a database call, then manipulate them by accessing the positional elements, your code could break if, say, you added a new column to your table. Not so if you first cast the returned tuples to namedtuples.

To illustrate, here is some code using ordinary tuples:

```
def compute_cost(records):
    total = 0.0
    for rec in records:
        total += rec[1] * rec[2]
    return total
```

References to positional elements often make the code a bit less expressive and more dependent on the structure of the records. Here is a version that uses a `namedtuple`:

```
from collections import namedtuple

Stock = namedtuple('Stock', ['name', 'shares', 'price'])
```

```
def compute_cost(records):
    total = 0.0
    for rec in records:
        s = Stock(*rec)
        total += s.shares * s.price
    return total
```

Naturally, you can avoid the explicit conversion to the `Stock` namedtuple if the `records` sequence in the example already contained such instances.

Discussion

One possible use of a `namedtuple` is as a replacement for a dictionary, which requires more space to store. Thus, if you are building large data structures involving dictionaries, use of a `namedtuple` will be more efficient. However, be aware that unlike a dictionary, a `namedtuple` is immutable. For example:

```
>>> s = Stock('ACME', 100, 123.45)
>>> s
Stock(name='ACME', shares=100, price=123.45)
>>> s.shares = 75
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: can't set attribute
>>>
```

If you need to change any of the attributes, it can be done using the `_replace()` method of a `namedtuple` instance, which makes an entirely new `namedtuple` with specified values replaced. For example:

```
>>> s = s._replace(shares=75)
>>> s
Stock(name='ACME', shares=75, price=123.45)
>>>
```

A subtle use of the `_replace()` method is that it can be a convenient way to populate named tuples that have optional or missing fields. To do this, you make a prototype tuple containing the default values and then use `_replace()` to create new instances with values replaced. For example:

```
from collections import namedtuple

Stock = namedtuple('Stock', ['name', 'shares', 'price', 'date', 'time'])

# Create a prototype instance
stock_prototype = Stock('', 0, 0.0, None, None)

# Function to convert a dictionary to a Stock
def dict_to_stock(s):
    return stock_prototype._replace(**s)
```

Here is an example of how this code would work:

```
>>> a = {'name': 'ACME', 'shares': 100, 'price': 123.45}
>>> dict_to_stock(a)
Stock(name='ACME', shares=100, price=123.45, date=None, time=None)
>>> b = {'name': 'ACME', 'shares': 100, 'price': 123.45, 'date': '12/17/2012'}
>>> dict_to_stock(b)
Stock(name='ACME', shares=100, price=123.45, date='12/17/2012', time=None)
>>>
```

Last, but not least, it should be noted that if your goal is to define an efficient data structure where you will be changing various instance attributes, using `namedtuple` is not your best choice. Instead, consider defining a class using `__slots__` instead (see Recipe 8.4).

1.19. Transforming and Reducing Data at the Same Time

Problem

You need to execute a reduction function (e.g., `sum()`, `min()`, `max()`), but first need to transform or filter the data.

Solution

A very elegant way to combine a data reduction and a transformation is to use a generator-expression argument. For example, if you want to calculate the sum of squares, do the following:

```
nums = [1, 2, 3, 4, 5]
s = sum(x * x for x in nums)
```

Here are a few other examples:

```
# Determine if any .py files exist in a directory
import os
files = os.listdir('dirname')
if any(name.endswith('.py') for name in files):
    print('There be python!')
else:
    print('Sorry, no python.')

# Output a tuple as CSV
s = ('ACME', 50, 123.45)
print(','.join(str(x) for x in s))

# Data reduction across fields of a data structure
portfolio = [
   {'name':'GOOG', 'shares': 50},
   {'name':'YHOO', 'shares': 75},
   {'name':'AOL', 'shares': 20},
```

```
    {'name':'SCOX', 'shares': 65}
]
min_shares = min(s['shares'] for s in portfolio)
```

Discussion

The solution shows a subtle syntactic aspect of generator expressions when supplied as the single argument to a function (i.e., you don't need repeated parentheses). For example, these statements are the same:

```
s = sum((x * x for x in nums))    # Pass generator-expr as argument
s = sum(x * x for x in nums)      # More elegant syntax
```

Using a generator argument is often a more efficient and elegant approach than first creating a temporary list. For example, if you didn't use a generator expression, you might consider this alternative implementation:

```
nums = [1, 2, 3, 4, 5]
s = sum([x * x for x in nums])
```

This works, but it introduces an extra step and creates an extra list. For such a small list, it might not matter, but if `nums` was huge, you would end up creating a large temporary data structure to only be used once and discarded. The generator solution transforms the data iteratively and is therefore much more memory-efficient.

Certain reduction functions such as `min()` and `max()` accept a `key` argument that might be useful in situations where you might be inclined to use a generator. For example, in the `portfolio` example, you might consider this alternative:

```
# Original: Returns 20
min_shares = min(s['shares'] for s in portfolio)

# Alternative: Returns {'name': 'AOL', 'shares': 20}
min_shares = min(portfolio, key=lambda s: s['shares'])
```

1.20. Combining Multiple Mappings into a Single Mapping

Problem

You have multiple dictionaries or mappings that you want to logically combine into a single mapping to perform certain operations, such as looking up values or checking for the existence of keys.

Solution

Suppose you have two dictionaries:

```
a = {'x': 1, 'z': 3 }
b = {'y': 2, 'z': 4 }
```

Now suppose you want to perform lookups where you have to check both dictionaries (e.g., first checking in `a` and then in `b` if not found). An easy way to do this is to use the `ChainMap` class from the `collections` module. For example:

```
from collections import ChainMap
c = ChainMap(a,b)
print(c['x'])      # Outputs 1  (from a)
print(c['y'])      # Outputs 2  (from b)
print(c['z'])      # Outputs 3  (from a)
```

Discussion

A `ChainMap` takes multiple mappings and makes them logically appear as one. However, the mappings are not literally merged together. Instead, a `ChainMap` simply keeps a list of the underlying mappings and redefines common dictionary operations to scan the list. Most operations will work. For example:

```
>>> len(c)
3
>>> list(c.keys())
['x', 'y', 'z']
>>> list(c.values())
[1, 2, 3]
>>>
```

If there are duplicate keys, the values from the first mapping get used. Thus, the entry `c['z']` in the example would always refer to the value in dictionary `a`, not the value in dictionary `b`.

Operations that mutate the mapping always affect the first mapping listed. For example:

```
>>> c['z'] = 10
>>> c['w'] = 40
>>> del c['x']
>>> a
{'w': 40, 'z': 10}
>>> del c['y']
Traceback (most recent call last):
...
KeyError: "Key not found in the first mapping: 'y'"
>>>
```

A `ChainMap` is particularly useful when working with scoped values such as variables in a programming language (i.e., globals, locals, etc.). In fact, there are methods that make this easy:

```
>>> values = ChainMap()
>>> values['x'] = 1
>>> # Add a new mapping
>>> values = values.new_child()
>>> values['x'] = 2
>>> # Add a new mapping
>>> values = values.new_child()
>>> values['x'] = 3
>>> values
ChainMap({'x': 3}, {'x': 2}, {'x': 1})
>>> values['x']
3
>>> # Discard last mapping
>>> values = values.parents
>>> values['x']
2
>>> # Discard last mapping
>>> values = values.parents
>>> values['x']
1
>>> values
ChainMap({'x': 1})
>>>
```

As an alternative to `ChainMap`, you might consider merging dictionaries together using the `update()` method. For example:

```
>>> a = {'x': 1, 'z': 3 }
>>> b = {'y': 2, 'z': 4 }
>>> merged = dict(b)
>>> merged.update(a)
>>> merged['x']
1
>>> merged['y']
2
>>> merged['z']
3
>>>
```

This works, but it requires you to make a completely separate dictionary object (or destructively alter one of the existing dictionaries). Also, if any of the original dictionaries mutate, the changes don't get reflected in the merged dictionary. For example:

```
>>> a['x'] = 13
>>> merged['x']
1
```

A `ChainMap` uses the original dictionaries, so it doesn't have this behavior. For example:

```
>>> a = {'x': 1, 'z': 3 }
>>> b = {'y': 2, 'z': 4 }
>>> merged = ChainMap(a, b)
>>> merged['x']
1
>>> a['x'] = 42
>>> merged['x']   # Notice change to merged dicts
42
>>>
```

CHAPTER 2

Strings and Text

Almost every useful program involves some kind of text processing, whether it is parsing data or generating output. This chapter focuses on common problems involving text manipulation, such as pulling apart strings, searching, substitution, lexing, and parsing. Many of these tasks can be easily solved using built-in methods of strings. However, more complicated operations might require the use of regular expressions or the creation of a full-fledged parser. All of these topics are covered. In addition, a few tricky aspects of working with Unicode are addressed.

2.1. Splitting Strings on Any of Multiple Delimiters

Problem

You need to split a string into fields, but the delimiters (and spacing around them) aren't consistent throughout the string.

Solution

The `split()` method of string objects is really meant for very simple cases, and does not allow for multiple delimiters or account for possible whitespace around the delimiters. In cases when you need a bit more flexibility, use the `re.split()` method:

```
>>> line = 'asdf fjdk; afed, fjek,asdf,      foo'
>>> import re
>>> re.split(r'[;,\s]\s*', line)
['asdf', 'fjdk', 'afed', 'fjek', 'asdf', 'foo']
```

Discussion

The `re.split()` function is useful because you can specify multiple patterns for the separator. For example, as shown in the solution, the separator is either a comma (,),

semicolon (;), or whitespace followed by any amount of extra whitespace. Whenever that pattern is found, the entire match becomes the delimiter between whatever fields lie on either side of the match. The result is a list of fields, just as with `str.split()`.

When using `re.split()`, you need to be a bit careful should the regular expression pattern involve a capture group enclosed in parentheses. If capture groups are used, then the matched text is also included in the result. For example, watch what happens here:

```
>>> fields = re.split(r'(;|,|\s)\s*', line)
>>> fields
['asdf', ' ', 'fjdk', ';', 'afed', ',', 'fjek', ',', 'asdf', ',', 'foo']
>>>
```

Getting the split characters might be useful in certain contexts. For example, maybe you need the split characters later on to reform an output string:

```
>>> values = fields[::2]
>>> delimiters = fields[1::2] + ['']
>>> values
['asdf', 'fjdk', 'afed', 'fjek', 'asdf', 'foo']
>>> delimiters
[' ', ';', ',', ',', ',', '']

>>> # Reform the line using the same delimiters
>>> ''.join(v+d for v,d in zip(values, delimiters))
'asdf fjdk;afed,fjek,asdf,foo'
>>>
```

If you don't want the separator characters in the result, but still need to use parentheses to group parts of the regular expression pattern, make sure you use a noncapture group, specified as `(?:...)`. For example:

```
>>> re.split(r'(?:,|;|\s)\s*', line)
['asdf', 'fjdk', 'afed', 'fjek', 'asdf', 'foo']
>>>
```

2.2. Matching Text at the Start or End of a String

Problem

You need to check the start or end of a string for specific text patterns, such as filename extensions, URL schemes, and so on.

Solution

A simple way to check the beginning or end of a string is to use the `str.startswith()` or `str.endswith()` methods. For example:

```
>>> filename = 'spam.txt'
>>> filename.endswith('.txt')
True
>>> filename.startswith('file:')
False
>>> url = 'http://www.python.org'
>>> url.startswith('http:')
True
>>>
```

If you need to check against multiple choices, simply provide a tuple of possibilities to `startswith()` or `endswith()`:

```
>>> import os
>>> filenames = os.listdir('.')
>>> filenames
[ 'Makefile', 'foo.c', 'bar.py', 'spam.c', 'spam.h' ]
>>> [name for name in filenames if name.endswith(('.c', '.h')) ]
['foo.c', 'spam.c', 'spam.h'
>>> any(name.endswith('.py') for name in filenames)
True
>>>
```

Here is another example:

```
from urllib.request import urlopen

def read_data(name):
    if name.startswith(('http:', 'https:', 'ftp:')):
        return urlopen(name).read()
    else:
        with open(name) as f:
            return f.read()
```

Oddly, this is one part of Python where a tuple is actually required as input. If you happen to have the choices specified in a list or set, just make sure you convert them using `tuple()` first. For example:

```
>>> choices = ['http:', 'ftp:']
>>> url = 'http://www.python.org'
>>> url.startswith(choices)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: startswith first arg must be str or a tuple of str, not list
>>> url.startswith(tuple(choices))
True
>>>
```

Discussion

The `startswith()` and `endswith()` methods provide a very convenient way to perform basic prefix and suffix checking. Similar operations can be performed with slices, but are far less elegant. For example:

```
>>> filename = 'spam.txt'
>>> filename[-4:] == '.txt'
True
>>> url = 'http://www.python.org'
>>> url[:5] == 'http:' or url[:6] == 'https:' or url[:4] == 'ftp:'
True
>>>
```

You might also be inclined to use regular expressions as an alternative. For example:

```
>>> import re
>>> url = 'http://www.python.org'
>>> re.match('http:|https:|ftp:', url)
<_sre.SRE_Match object at 0x101253098>
>>>
```

This works, but is often overkill for simple matching. Using this recipe is simpler and runs faster.

Last, but not least, the `startswith()` and `endswith()` methods look nice when combined with other operations, such as common data reductions. For example, this statement that checks a directory for the presence of certain kinds of files:

```
if any(name.endswith(('.c', '.h')) for name in listdir(dirname)):
    ...
```

2.3. Matching Strings Using Shell Wildcard Patterns

Problem

You want to match text using the same wildcard patterns as are commonly used when working in Unix shells (e.g., `*.py`, `Dat[0-9]*.csv`, etc.).

Solution

The `fnmatch` module provides two functions—`fnmatch()` and `fnmatchcase()`—that can be used to perform such matching. The usage is simple:

```
>>> from fnmatch import fnmatch, fnmatchcase
>>> fnmatch('foo.txt', '*.txt')
True
>>> fnmatch('foo.txt', '?oo.txt')
True
>>> fnmatch('Dat45.csv', 'Dat[0-9]*')
```

```
True
>>> names = ['Dat1.csv', 'Dat2.csv', 'config.ini', 'foo.py']
>>> [name for name in names if fnmatch(name, 'Dat*.csv')]
['Dat1.csv', 'Dat2.csv']
>>>
```

Normally, `fnmatch()` matches patterns using the same case-sensitivity rules as the system's underlying filesystem (which varies based on operating system). For example:

```
>>> # On OS X (Mac)
>>> fnmatch('foo.txt', '*.TXT')
False

>>> # On Windows
>>> fnmatch('foo.txt', '*.TXT')
True
>>>
```

If this distinction matters, use `fnmatchcase()` instead. It matches exactly based on the lower- and uppercase conventions that you supply:

```
>>> fnmatchcase('foo.txt', '*.TXT')
False
>>>
```

An often overlooked feature of these functions is their potential use with data processing of nonfilename strings. For example, suppose you have a list of street addresses like this:

```
addresses = [
    '5412 N CLARK ST',
    '1060 W ADDISON ST',
    '1039 W GRANVILLE AVE',
    '2122 N CLARK ST',
    '4802 N BROADWAY',
]
```

You could write list comprehensions like this:

```
>>> from fnmatch import fnmatchcase
>>> [addr for addr in addresses if fnmatchcase(addr, '* ST')]
['5412 N CLARK ST', '1060 W ADDISON ST', '2122 N CLARK ST']
>>> [addr for addr in addresses if fnmatchcase(addr, '54[0-9][0-9] *CLARK*')]
['5412 N CLARK ST']
>>>
```

Discussion

The matching performed by `fnmatch` sits somewhere between the functionality of simple string methods and the full power of regular expressions. If you're just trying to provide a simple mechanism for allowing wildcards in data processing operations, it's often a reasonable solution.

If you're actually trying to write code that matches filenames, use the `glob` module instead. See Recipe 5.13.

2.4. Matching and Searching for Text Patterns

Problem

You want to match or search text for a specific pattern.

Solution

If the text you're trying to match is a simple literal, you can often just use the basic string methods, such as `str.find()`, `str.endswith()`, `str.startswith()`, or similar. For example:

```
>>> text = 'yeah, but no, but yeah, but no, but yeah'

>>> # Exact match
>>> text == 'yeah'
False

>>> # Match at start or end
>>> text.startswith('yeah')
True
>>> text.endswith('no')
False

>>> # Search for the location of the first occurrence
>>> text.find('no')
10
>>>
```

For more complicated matching, use regular expressions and the `re` module. To illustrate the basic mechanics of using regular expressions, suppose you want to match dates specified as digits, such as "11/27/2012." Here is a sample of how you would do it:

```
>>> text1 = '11/27/2012'
>>> text2 = 'Nov 27, 2012'
>>>
>>> import re
>>> # Simple matching: \d+ means match one or more digits
>>> if re.match(r'\d+/\d+/\d+', text1):
...     print('yes')
... else:
...     print('no')
...
yes
>>> if re.match(r'\d+/\d+/\d+', text2):
...     print('yes')
... else:
```

```
...     print('no')
...
no
>>>
```

If you're going to perform a lot of matches using the same pattern, it usually pays to precompile the regular expression pattern into a pattern object first. For example:

```
>>> datepat = re.compile(r'\d+/\d+/\d+')
>>> if datepat.match(text1):
...     print('yes')
... else:
...     print('no')
...
yes
>>> if datepat.match(text2):
...     print('yes')
... else:
...     print('no')
...
no
>>>
```

`match()` always tries to find the match at the start of a string. If you want to search text for all occurrences of a pattern, use the `findall()` method instead. For example:

```
>>> text = 'Today is 11/27/2012. PyCon starts 3/13/2013.'
>>> datepat.findall(text)
['11/27/2012', '3/13/2013']
>>>
```

When defining regular expressions, it is common to introduce capture groups by enclosing parts of the pattern in parentheses. For example:

```
>>> datepat = re.compile(r'(\d+)/(\d+)/(\d+)')
>>>
```

Capture groups often simplify subsequent processing of the matched text because the contents of each group can be extracted individually. For example:

```
>>> m = datepat.match('11/27/2012')
>>> m
<_sre.SRE_Match object at 0x1005d2750>

>>> # Extract the contents of each group
>>> m.group(0)
'11/27/2012'
>>> m.group(1)
'11'
>>> m.group(2)
'27'
>>> m.group(3)
'2012'
>>> m.groups()
```

```
('11', '27', '2012')
>>> month, day, year = m.groups()
>>>

>>> # Find all matches (notice splitting into tuples)
>>> text
'Today is 11/27/2012. PyCon starts 3/13/2013.'
>>> datepat.findall(text)
[('11', '27', '2012'), ('3', '13', '2013')]
>>> for month, day, year in datepat.findall(text):
...     print('{}-{}-{}'.format(year, month, day))
...
2012-11-27
2013-3-13
>>>
```

The `findall()` method searches the text and finds all matches, returning them as a list. If you want to find matches iteratively, use the `finditer()` method instead. For example:

```
>>> for m in datepat.finditer(text):
...     print(m.groups())
...
('11', '27', '2012')
('3', '13', '2013')
>>>
```

Discussion

A basic tutorial on the theory of regular expressions is beyond the scope of this book. However, this recipe illustrates the absolute basics of using the `re` module to match and search for text. The essential functionality is first compiling a pattern using `re.compile()` and then using methods such as `match()`, `findall()`, or `finditer()`.

When specifying patterns, it is relatively common to use raw strings such as `r'(\d+)/(\d+)/(\d+)'`. Such strings leave the backslash character uninterpreted, which can be useful in the context of regular expressions. Otherwise, you need to use double backslashes such as `'(\\d+)/(\\d+)/(\\d+)'`.

Be aware that the `match()` method only checks the beginning of a string. It's possible that it will match things you aren't expecting. For example:

```
>>> m = datepat.match('11/27/2012abcdef')
>>> m
<_sre.SRE_Match object at 0x1005d27e8>
>>> m.group()
'11/27/2012'
>>>
```

If you want an exact match, make sure the pattern includes the end-marker (`$`), as in the following:

```
>>> datepat = re.compile(r'(\d+)/(\d+)/(\d+)$')
>>> datepat.match('11/27/2012abcdef')
>>> datepat.match('11/27/2012')
<_sre.SRE_Match object at 0x1005d2750>
>>>
```

Last, if you're just doing a simple text matching/searching operation, you can often skip the compilation step and use module-level functions in the `re` module instead. For example:

```
>>> re.findall(r'(\d+)/(\d+)/(\d+)', text)
[('11', '27', '2012'), ('3', '13', '2013')]
>>>
```

Be aware, though, that if you're going to perform a lot of matching or searching, it usually pays to compile the pattern first and use it over and over again. The module-level functions keep a cache of recently compiled patterns, so there isn't a huge performance hit, but you'll save a few lookups and extra processing by using your own compiled pattern.

2.5. Searching and Replacing Text

Problem

You want to search for and replace a text pattern in a string.

Solution

For simple literal patterns, use the `str.replace()` method. For example:

```
>>> text = 'yeah, but no, but yeah, but no, but yeah'

>>> text.replace('yeah', 'yep')
'yep, but no, but yep, but no, but yep'
>>>
```

For more complicated patterns, use the `sub()` functions/methods in the `re` module. To illustrate, suppose you want to rewrite dates of the form "11/27/2012" as "2012-11-27." Here is a sample of how to do it:

```
>>> text = 'Today is 11/27/2012. PyCon starts 3/13/2013.'
>>> import re
>>> re.sub(r'(\d+)/(\d+)/(\d+)', r'\3-\1-\2', text)
'Today is 2012-11-27. PyCon starts 2013-3-13.'
>>>
```

The first argument to `sub()` is the pattern to match and the second argument is the replacement pattern. Backslashed digits such as `\3` refer to capture group numbers in the pattern.

If you're going to perform repeated substitutions of the same pattern, consider compiling it first for better performance. For example:

```
>>> import re
>>> datepat = re.compile(r'(\d+)/(\d+)/(\d+)')
>>> datepat.sub(r'\3-\1-\2', text)
'Today is 2012-11-27. PyCon starts 2013-3-13.'
>>>
```

For more complicated substitutions, it's possible to specify a substitution callback function instead. For example:

```
>>> from calendar import month_abbr
>>> def change_date(m):
...     mon_name = month_abbr[int(m.group(1))]
...     return '{} {} {}'.format(m.group(2), mon_name, m.group(3))
...
>>> datepat.sub(change_date, text)
'Today is 27 Nov 2012. PyCon starts 13 Mar 2013.'
>>>
```

As input, the argument to the substitution callback is a match object, as returned by `match()` or `find()`. Use the `.group()` method to extract specific parts of the match. The function should return the replacement text.

If you want to know how many substitutions were made in addition to getting the replacement text, use `re.subn()` instead. For example:

```
>>> newtext, n = datepat.subn(r'\3-\1-\2', text)
>>> newtext
'Today is 2012-11-27. PyCon starts 2013-3-13.'
>>> n
2
>>>
```

Discussion

There isn't much more to regular expression search and replace than the `sub()` method shown. The trickiest part is specifying the regular expression pattern—something that's best left as an exercise to the reader.

2.6. Searching and Replacing Case-Insensitive Text

Problem

You need to search for and possibly replace text in a case-insensitive manner.

Solution

To perform case-insensitive text operations, you need to use the `re` module and supply the `re.IGNORECASE` flag to various operations. For example:

```
>>> text = 'UPPER PYTHON, lower python, Mixed Python'
>>> re.findall('python', text, flags=re.IGNORECASE)
['PYTHON', 'python', 'Python']
>>> re.sub('python', 'snake', text, flags=re.IGNORECASE)
'UPPER snake, lower snake, Mixed snake'
>>>
```

The last example reveals a limitation that replacing text won't match the case of the matched text. If you need to fix this, you might have to use a support function, as in the following:

```
def matchcase(word):
    def replace(m):
        text = m.group()
        if text.isupper():
            return word.upper()
        elif text.islower():
            return word.lower()
        elif text[0].isupper():
            return word.capitalize()
        else:
            return word
    return replace
```

Here is an example of using this last function:

```
>>> re.sub('python', matchcase('snake'), text, flags=re.IGNORECASE)
'UPPER SNAKE, lower snake, Mixed Snake'
>>>
```

Discussion

For simple cases, simply providing the `re.IGNORECASE` is enough to perform case-insensitive matching. However, be aware that this may not be enough for certain kinds of Unicode matching involving case folding. See Recipe 2.10 for more details.

2.7. Specifying a Regular Expression for the Shortest Match

Problem

You're trying to match a text pattern using regular expressions, but it is identifying the longest possible matches of a pattern. Instead, you would like to change it to find the shortest possible match.

Solution

This problem often arises in patterns that try to match text enclosed inside a pair of starting and ending delimiters (e.g., a quoted string). To illustrate, consider this example:

```
>>> str_pat = re.compile(r'\"(.*)\"')
>>> text1 = 'Computer says "no."'
>>> str_pat.findall(text1)
['no.']
>>> text2 = 'Computer says "no." Phone says "yes."'
>>> str_pat.findall(text2)
['no." Phone says "yes.']
>>>
```

In this example, the pattern `r'\"(.*)\"'` is attempting to match text enclosed inside quotes. However, the `*` operator in a regular expression is greedy, so matching is based on finding the longest possible match. Thus, in the second example involving `text2`, it incorrectly matches the two quoted strings.

To fix this, add the `?` modifier after the `*` operator in the pattern, like this:

```
>>> str_pat = re.compile(r'\"(.*?)\"')
>>> str_pat.findall(text2)
['no.', 'yes.']
>>>
```

This makes the matching nongreedy, and produces the shortest match instead.

Discussion

This recipe addresses one of the more common problems encountered when writing regular expressions involving the dot (.) character. In a pattern, the dot matches any character except a newline. However, if you bracket the dot with starting and ending text (such as a quote), matching will try to find the longest possible match to the pattern. This causes multiple occurrences of the starting or ending text to be skipped altogether and included in the results of the longer match. Adding the `?` right after operators such as `*` or `+` forces the matching algorithm to look for the shortest possible match instead.

2.8. Writing a Regular Expression for Multiline Patterns

Problem

You're trying to match a block of text using a regular expression, but you need the match to span multiple lines.

Solution

This problem typically arises in patterns that use the dot (.) to match any character but forget to account for the fact that it doesn't match newlines. For example, suppose you are trying to match C-style comments:

```
>>> comment = re.compile(r'/\*(.*?)\*/')
>>> text1 = '/* this is a comment */'
>>> text2 = '''/* this is a
...               multiline comment */
... '''
>>>
>>> comment.findall(text1)
[' this is a comment ']
>>> comment.findall(text2)
[]
>>>
```

To fix the problem, you can add support for newlines. For example:

```
>>> comment = re.compile(r'/\*((?:.|\n)*?)\*/')
>>> comment.findall(text2)
[' this is a\n              multiline comment ']
>>>
```

In this pattern, `(?:.|\n)` specifies a noncapture group (i.e., it defines a group for the purposes of matching, but that group is not captured separately or numbered).

Discussion

The `re.compile()` function accepts a flag, `re.DOTALL`, which is useful here. It makes the `.` in a regular expression match all characters, including newlines. For example:

```
>>> comment = re.compile(r'/\*(.*?)\*/', re.DOTALL)
>>> comment.findall(text2)
[' this is a\n              multiline comment ']
```

Using the `re.DOTALL` flag works fine for simple cases, but might be problematic if you're working with extremely complicated patterns or a mix of separate regular expressions that have been combined together for the purpose of tokenizing, as described in Recipe 2.18. If given a choice, it's usually better to define your regular expression pattern so that it works correctly without the need for extra flags.

2.9. Normalizing Unicode Text to a Standard Representation

Problem

You're working with Unicode strings, but need to make sure that all of the strings have the same underlying representation.

Solution

In Unicode, certain characters can be represented by more than one valid sequence of code points. To illustrate, consider the following example:

```
>>> s1 = 'Spicy Jalape\u00f1o'
>>> s2 = 'Spicy Jalapen\u0303o'
>>> s1
'Spicy Jalapeño'
>>> s2
'Spicy Jalapeño'
>>> s1 == s2
False
>>> len(s1)
14
>>> len(s2)
15
>>>
```

Here the text "Spicy Jalapeño" has been presented in two forms. The first uses the fully composed "ñ" character (U+00F1). The second uses the Latin letter "n" followed by a "~" combining character (U+0303).

Having multiple representations is a problem for programs that compare strings. In order to fix this, you should first normalize the text into a standard representation using the `unicodedata` module:

```
>>> import unicodedata
>>> t1 = unicodedata.normalize('NFC', s1)
>>> t2 = unicodedata.normalize('NFC', s2)
>>> t1 == t2
True
>>> print(ascii(t1))
'Spicy Jalape\xf1o'

>>> t3 = unicodedata.normalize('NFD', s1)
>>> t4 = unicodedata.normalize('NFD', s2)
>>> t3 == t4
True
>>> print(ascii(t3))
'Spicy Jalapen\u0303o'
>>>
```

The first argument to `normalize()` specifies how you want the string normalized. `NFC` means that characters should be fully composed (i.e., use a single code point if possible). `NFD` means that characters should be fully decomposed with the use of combining characters.

Python also supports the normalization forms `NFKC` and `NFKD`, which add extra compatibility features for dealing with certain kinds of characters. For example:

```
>>> s = '\ufb01'   # A single character
>>> s
'fi'
>>> unicodedata.normalize('NFD', s)
'fi'

# Notice how the combined letters are broken apart here
>>> unicodedata.normalize('NFKD', s)
'fi'
>>> unicodedata.normalize('NFKC', s)
'fi'
>>>
```

Discussion

Normalization is an important part of any code that needs to ensure that it processes Unicode text in a sane and consistent way. This is especially true when processing strings received as part of user input where you have little control of the encoding.

Normalization can also be an important part of sanitizing and filtering text. For example, suppose you want to remove all diacritical marks from some text (possibly for the purposes of searching or matching):

```
>>> t1 = unicodedata.normalize('NFD', s1)
>>> ''.join(c for c in t1 if not unicodedata.combining(c))
'Spicy Jalapeno'
>>>
```

This last example shows another important aspect of the `unicodedata` module—namely, utility functions for testing characters against character classes. The `combining()` function tests a character to see if it is a combining character. There are other functions in the module for finding character categories, testing digits, and so forth.

Unicode is obviously a large topic. For more detailed reference information about normalization, visit Unicode's page on the subject. Ned Batchelder has also given an excellent presentation on Python Unicode handling issues at his website.

2.10. Working with Unicode Characters in Regular Expressions

Problem

You are using regular expressions to process text, but are concerned about the handling of Unicode characters.

Solution

By default, the `re` module is already programmed with rudimentary knowledge of certain Unicode character classes. For example, `\d` already matches any unicode digit character:

```
>>> import re
>>> num = re.compile('\d+')
>>> # ASCII digits
>>> num.match('123')
<_sre.SRE_Match object at 0x1007d9ed0>

>>> # Arabic digits
>>> num.match('\u0661\u0662\u0663')
<_sre.SRE_Match object at 0x101234030>
>>>
```

If you need to include specific Unicode characters in patterns, you can use the usual escape sequence for Unicode characters (e.g., `\uFFFF` or `\UFFFFFFF`). For example, here is a regex that matches all characters in a few different Arabic code pages:

```
>>> arabic = re.compile('[\u0600-\u06ff\u0750-\u077f\u08a0-\u08ff]+')
>>>
```

When performing matching and searching operations, it's a good idea to normalize and possibly sanitize all text to a standard form first (see Recipe 2.9). However, it's also important to be aware of special cases. For example, consider the behavior of case-insensitive matching combined with case folding:

```
>>> pat = re.compile('stra\u00dfe', re.IGNORECASE)
>>> s = 'straße'
>>> pat.match(s)              # Matches
<_sre.SRE_Match object at 0x10069d370>
>>> pat.match(s.upper())      # Doesn't match
>>> s.upper()                 # Case folds
'STRASSE'
>>>
```

Discussion

Mixing Unicode and regular expressions is often a good way to make your head explode. If you're going to do it seriously, you should consider installing the third-party `regex` library, which provides full support for Unicode case folding, as well as a variety of other interesting features, including approximate matching.

2.11. Stripping Unwanted Characters from Strings

Problem

You want to strip unwanted characters, such as whitespace, from the beginning, end, or middle of a text string.

Solution

The `strip()` method can be used to strip characters from the beginning or end of a string. `lstrip()` and `rstrip()` perform stripping from the left or right side, respectively. By default, these methods strip whitespace, but other characters can be given. For example:

```
>>> # Whitespace stripping
>>> s = '   hello world  \n'
>>> s.strip()
'hello world'
>>> s.lstrip()
'hello world  \n'
>>> s.rstrip()
'   hello world'
>>>

>>> # Character stripping
>>> t = '-----hello====='
>>> t.lstrip('-')
'hello====='
>>> t.strip('-=')
'hello'
>>>
```

Discussion

The various `strip()` methods are commonly used when reading and cleaning up data for later processing. For example, you can use them to get rid of whitespace, remove quotations, and other tasks.

Be aware that stripping does not apply to any text in the middle of a string. For example:

```
>>> s = '  hello       world   \n'
>>> s = s.strip()
>>> s
'hello       world'
>>>
```

If you needed to do something to the inner space, you would need to use another technique, such as using the `replace()` method or a regular expression substitution. For example:

```
>>> s.replace(' ', '')
'helloworld'
>>> import re
>>> re.sub('\s+', ' ', s)
'hello world'
>>>
```

It is often the case that you want to combine string stripping operations with some other kind of iterative processing, such as reading lines of data from a file. If so, this is one area where a generator expression can be useful. For example:

```
with open(filename) as f:
    lines = (line.strip() for line in f)
    for line in lines:
        ...
```

Here, the expression `lines = (line.strip() for line in f)` acts as a kind of data transform. It's efficient because it doesn't actually read the data into any kind of temporary list first. It just creates an iterator where all of the lines produced have the stripping operation applied to them.

For even more advanced stripping, you might turn to the `translate()` method. See the next recipe on sanitizing strings for further details.

2.12. Sanitizing and Cleaning Up Text

Problem

Some bored script kiddie has entered the text "pýt̂höñ" into a form on your web page and you'd like to clean it up somehow.

Solution

The problem of sanitizing and cleaning up text applies to a wide variety of problems involving text parsing and data handling. At a very simple level, you might use basic string functions (e.g., `str.upper()` and `str.lower()`) to convert text to a standard case. Simple replacements using `str.replace()` or `re.sub()` can focus on removing or

changing very specific character sequences. You can also normalize text using `unicodedata.normalize()`, as shown in Recipe 2.9.

However, you might want to take the sanitation process a step further. Perhaps, for example, you want to eliminate whole ranges of characters or strip diacritical marks. To do so, you can turn to the often overlooked `str.translate()` method. To illustrate, suppose you've got a messy string such as the following:

```
>>> s = 'pýt̂höñ\fis\tawesome\r\n'
>>> s
'pýt̂höñ\x0cis\tawesome\r\n'
>>>
```

The first step is to clean up the whitespace. To do this, make a small translation table and use `translate()`:

```
>>> remap = {
...     ord('\t') : ' ',
...     ord('\f') : ' ',
...     ord('\r') : None      # Deleted
... }
>>> a = s.translate(remap)
>>> a
'pýt̂höñ is awesome\n'
>>>
```

As you can see here, whitespace characters such as `\t` and `\f` have been remapped to a single space. The carriage return `\r` has been deleted entirely.

You can take this remapping idea a step further and build much bigger tables. For example, let's remove all combining characters:

```
>>> import unicodedata
>>> import sys
>>> cmb_chrs = dict.fromkeys(c for c in range(sys.maxunicode)
...                          if unicodedata.combining(chr(c)))
...
>>> b = unicodedata.normalize('NFD', a)
>>> b
'pýt̂höñ is awesome\n'
>>> b.translate(cmb_chrs)
'python is awesome\n'
>>>
```

In this last example, a dictionary mapping every Unicode combining character to `None` is created using the `dict.fromkeys()`.

The original input is then normalized into a decomposed form using `unicodedata.normalize()`. From there, the translate function is used to delete all of the accents. Similar techniques can be used to remove other kinds of characters (e.g., control characters, etc.).

As another example, here is a translation table that maps all Unicode decimal digit characters to their equivalent in ASCII:

```
>>> digitmap = { c: ord('0') + unicodedata.digit(chr(c))
...              for c in range(sys.maxunicode)
...              if unicodedata.category(chr(c)) == 'Nd' }
...
>>> len(digitmap)
460
>>> # Arabic digits
>>> x = '\u0661\u0662\u0663'
>>> x.translate(digitmap)
'123'
>>>
```

Yet another technique for cleaning up text involves I/O decoding and encoding functions. The idea here is to first do some preliminary cleanup of the text, and then run it through a combination of `encode()` or `decode()` operations to strip or alter it. For example:

```
>>> a
'pýẗhön is awesome\n'
>>> b = unicodedata.normalize('NFD', a)
>>> b.encode('ascii', 'ignore').decode('ascii')
'python is awesome\n'
>>>
```

Here the normalization process decomposed the original text into characters along with separate combining characters. The subsequent ASCII encoding/decoding simply discarded all of those characters in one fell swoop. Naturally, this would only work if getting an ASCII representation was the final goal.

Discussion

A major issue with sanitizing text can be runtime performance. As a general rule, the simpler it is, the faster it will run. For simple replacements, the `str.replace()` method is often the fastest approach—even if you have to call it multiple times. For instance, to clean up whitespace, you could use code like this:

```
def clean_spaces(s):
    s = s.replace('\r', '')
    s = s.replace('\t', ' ')
    s = s.replace('\f', ' ')
    return s
```

If you try it, you'll find that it's quite a bit faster than using `translate()` or an approach using a regular expression.

On the other hand, the `translate()` method is very fast if you need to perform any kind of nontrivial character-to-character remapping or deletion.

In the big picture, performance is something you will have to study further in your particular application. Unfortunately, it's impossible to suggest one specific technique that works best for all cases, so try different approaches and measure it.

Although the focus of this recipe has been text, similar techniques can be applied to bytes, including simple replacements, translation, and regular expressions.

2.13. Aligning Text Strings

Problem

You need to format text with some sort of alignment applied.

Solution

For basic alignment of strings, the `ljust()`, `rjust()`, and `center()` methods of strings can be used. For example:

```
>>> text = 'Hello World'
>>> text.ljust(20)
'Hello World         '
>>> text.rjust(20)
'         Hello World'
>>> text.center(20)
'    Hello World     '
>>>
```

All of these methods accept an optional &&65.180&&fill character as well. For example:

```
>>> text.rjust(20,'=')
'=========Hello World'
>>> text.center(20,'*')
'****Hello World*****'
>>>
```

The `format()` function can also be used to easily align things. All you need to do is use the <, >, or ^ characters along with a desired width. For example:

```
>>> format(text, '>20')
'         Hello World'
>>> format(text, '<20')
'Hello World         '
>>> format(text, '^20')
'    Hello World     '
>>>
```

If you want to include a fill character other than a space, specify it before the alignment character:

```
>>> format(text, '=>20s')
'=========Hello World'
```

```
>>> format(text, '*^20s')
'****Hello World*****'
>>>
```

These format codes can also be used in the `format()` method when formatting multiple values. For example:

```
>>> '{:>10s} {:>10s}'.format('Hello', 'World')
'     Hello      World'
>>>
```

One benefit of `format()` is that it is not specific to strings. It works with any value, making it more general purpose. For instance, you can use it with numbers:

```
>>> x = 1.2345
>>> format(x, '>10')
'    1.2345'
>>> format(x, '^10.2f')
'   1.23   '
>>>
```

Discussion

In older code, you will also see the `%` operator used to format text. For example:

```
>>> '%-20s' % text
'Hello World         '
>>> '%20s' % text
'         Hello World'
>>>
```

However, in new code, you should probably prefer the use of the `format()` function or method. `format()` is a lot more powerful than what is provided with the `%` operator. Moreover, `format()` is more general purpose than using the `jlust()`, `rjust()`, or `center()` method of strings in that it works with any kind of object.

For a complete list of features available with the `format()` function, consult the online Python documentation.

2.14. Combining and Concatenating Strings

Problem

You want to combine many small strings together into a larger string.

Solution

If the strings you wish to combine are found in a sequence or iterable, the fastest way to combine them is to use the `join()` method. For example:

```
>>> parts = ['Is', 'Chicago', 'Not', 'Chicago?']
>>> ' '.join(parts)
'Is Chicago Not Chicago?'
>>> ','.join(parts)
'Is,Chicago,Not,Chicago?'
>>> ''.join(parts)
'IsChicagoNotChicago?'
>>>
```

At first glance, this syntax might look really odd, but the `join()` operation is specified as a method on strings. Partly this is because the objects you want to join could come from any number of different data sequences (e.g., lists, tuples, dicts, files, sets, or generators), and it would be redundant to have `join()` implemented as a method on all of those objects separately. So you just specify the separator string that you want and use the `join()` method on it to glue text fragments together.

If you're only combining a few strings, using + usually works well enough:

```
>>> a = 'Is Chicago'
>>> b = 'Not Chicago?'
>>> a + ' ' + b
'Is Chicago Not Chicago?'
>>>
```

The + operator also works fine as a substitute for more complicated string formatting operations. For example:

```
>>> print('{} {}'.format(a,b))
Is Chicago Not Chicago?
>>> print(a + ' ' + b)
Is Chicago Not Chicago?
>>>
```

If you're trying to combine string literals together in source code, you can simply place them adjacent to each other with no + operator. For example:

```
>>> a = 'Hello' 'World'
>>> a
'HelloWorld'
>>>
```

Discussion

Joining strings together might not seem advanced enough to warrant an entire recipe, but it's often an area where programmers make programming choices that severely impact the performance of their code.

The most important thing to know is that using the + operator to join a lot of strings together is grossly inefficient due to the memory copies and garbage collection that occurs. In particular, you never want to write code that joins strings together like this:

```
s = ''
for p in parts:
    s += p
```

This runs quite a bit slower than using the `join()` method, mainly because each `+=` operation creates a new string object. You're better off just collecting all of the parts first and then joining them together at the end.

One related (and pretty neat) trick is the conversion of data to strings and concatenation at the same time using a generator expression, as described in Recipe 1.19. For example:

```
>>> data = ['ACME', 50, 91.1]
>>> ','.join(str(d) for d in data)
'ACME,50,91.1'
>>>
```

Also be on the lookout for unnecessary string concatenations. Sometimes programmers get carried away with concatenation when it's really not technically necessary. For example, when printing:

```
print(a + ':' + b + ':' + c)       # Ugly
print(':'.join([a, b, c]))         # Still ugly

print(a, b, c, sep=':')            # Better
```

Mixing I/O operations and string concatenation is something that might require study in your application. For example, consider the following two code fragments:

```
# Version 1 (string concatenation)
f.write(chunk1 + chunk2)

# Version 2 (separate I/O operations)
f.write(chunk1)
f.write(chunk2)
```

If the two strings are small, the first version might offer much better performance due to the inherent expense of carrying out an I/O system call. On the other hand, if the two strings are large, the second version may be more efficient, since it avoids making a large temporary result and copying large blocks of memory around. Again, it must be stressed that this is something you would have to study in relation to your own data in order to determine which performs best.

Last, but not least, if you're writing code that is building output from lots of small strings, you might consider writing that code as a generator function, using `yield` to emit fragments. For example:

```
def sample():
    yield 'Is'
    yield 'Chicago'
    yield 'Not'
    yield 'Chicago?'
```

The interesting thing about this approach is that it makes no assumption about how the fragments are to be assembled together. For example, you could simply join the fragments using `join()`:

```
text = ''.join(sample())
```

Or you could redirect the fragments to I/O:

```
for part in sample():
    f.write(part)
```

Or you could come up with some kind of hybrid scheme that's smart about combining I/O operations:

```
def combine(source, maxsize):
    parts = []
    size = 0
    for part in source:
        parts.append(part)
        size += len(part)
        if size > maxsize:
            yield ''.join(parts)
            parts = []
            size = 0
    yield ''.join(parts)

for part in combine(sample(), 32768):
    f.write(part)
```

The key point is that the original generator function doesn't have to know the precise details. It just yields the parts.

2.15. Interpolating Variables in Strings

Problem

You want to create a string in which embedded variable names are substituted with a string representation of a variable's value.

Solution

Python has no direct support for simply substituting variable values in strings. However, this feature can be approximated using the `format()` method of strings. For example:

```
>>> s = '{name} has {n} messages.'
>>> s.format(name='Guido', n=37)
'Guido has 37 messages.'
>>>
```

Alternatively, if the values to be substituted are truly found in variables, you can use the combination of `format_map()` and `vars()`, as in the following:

```
>>> name = 'Guido'
>>> n = 37
>>> s.format_map(vars())
'Guido has 37 messages.'
>>>
```

One subtle feature of `vars()` is that it also works with instances. For example:

```
>>> class Info:
...     def __init__(self, name, n):
...         self.name = name
...         self.n = n
...
>>> a = Info('Guido',37)
>>> s.format_map(vars(a))
'Guido has 37 messages.'
>>>
```

One downside of `format()` and `format_map()` is that they do not deal gracefully with missing values. For example:

```
>>> s.format(name='Guido')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'n'
>>>
```

One way to avoid this is to define an alternative dictionary class with a `__missing__()` method, as in the following:

```
class safesub(dict):
    def __missing__(self, key):
        return '{' + key + '}'
```

Now use this class to wrap the inputs to `format_map()`:

```
>>> del n     # Make sure n is undefined
>>> s.format_map(safesub(vars()))
'Guido has {n} messages.'
>>>
```

If you find yourself frequently performing these steps in your code, you could hide the variable substitution process behind a small utility function that employs a so-called "frame hack." For example:

```
import sys

def sub(text):
    return text.format_map(safesub(sys._getframe(1).f_locals))
```

Now you can type things like this:

```
>>> name = 'Guido'
>>> n = 37
>>> print(sub('Hello {name}'))
Hello Guido
>>> print(sub('You have {n} messages.'))
You have 37 messages.
>>> print(sub('Your favorite color is {color}'))
Your favorite color is {color}
>>>
```

Discussion

The lack of true variable interpolation in Python has led to a variety of solutions over the years. As an alternative to the solution presented in this recipe, you will sometimes see string formatting like this:

```
>>> name = 'Guido'
>>> n = 37
>>> '%(name) has %(n) messages.' % vars()
'Guido has 37 messages.'
>>>
```

You may also see the use of template strings:

```
>>> import string
>>> s = string.Template('$name has $n messages.')
>>> s.substitute(vars())
'Guido has 37 messages.'
>>>
```

However, the `format()` and `format_map()` methods are more modern than either of these alternatives, and should be preferred. One benefit of using `format()` is that you also get all of the features related to string formatting (alignment, padding, numerical formatting, etc.), which is simply not possible with alternatives such as `Template` string objects.

Parts of this recipe also illustrate a few interesting advanced features. The little-known `__missing__()` method of mapping/dict classes is a method that you can define to handle missing values. In the `safesub` class, this method has been defined to return missing values back as a placeholder. Instead of getting a `KeyError` exception, you would see the missing values appearing in the resulting string (potentially useful for debugging).

The `sub()` function uses `sys._getframe(1)` to return the stack frame of the caller. From that, the `f_locals` attribute is accessed to get the local variables. It goes without saying that messing around with stack frames should probably be avoided in most code. However, for utility functions such as a string substitution feature, it can be useful. As an aside, it's probably worth noting that `f_locals` is a dictionary that is a copy of the local variables in the calling function. Although you can modify the contents of `f_locals`,

the modifications don't actually have any lasting effect. Thus, even though accessing a different stack frame might look evil, it's not possible to accidentally overwrite variables or change the local environment of the caller.

2.16. Reformatting Text to a Fixed Number of Columns

Problem

You have long strings that you want to reformat so that they fill a user-specified number of columns.

Solution

Use the `textwrap` module to reformat text for output. For example, suppose you have the following long string:

```
s = "Look into my eyes, look into my eyes, the eyes, the eyes, \
the eyes, not around the eyes, don't look around the eyes, \
look into my eyes, you're under."
```

Here's how you can use the `textwrap` module to reformat it in various ways:

```
>>> import textwrap
>>> print(textwrap.fill(s, 70))
Look into my eyes, look into my eyes, the eyes, the eyes, the eyes,
not around the eyes, don't look around the eyes, look into my eyes,
you're under.

>>> print(textwrap.fill(s, 40))
Look into my eyes, look into my eyes,
the eyes, the eyes, the eyes, not around
the eyes, don't look around the eyes,
look into my eyes, you're under.

>>> print(textwrap.fill(s, 40, initial_indent='    '))
    Look into my eyes, look into my
eyes, the eyes, the eyes, the eyes, not
around the eyes, don't look around the
eyes, look into my eyes, you're under.

>>> print(textwrap.fill(s, 40, subsequent_indent='    '))
Look into my eyes, look into my eyes,
    the eyes, the eyes, the eyes, not
    around the eyes, don't look around
    the eyes, look into my eyes, you're
    under.
```

Discussion

The `textwrap` module is a straightforward way to clean up text for printing—especially if you want the output to fit nicely on the terminal. On the subject of the terminal size, you can obtain it using `os.get_terminal_size()`. For example:

```
>>> import os
>>> os.get_terminal_size().columns
80
>>>
```

The `fill()` method has a few additional options that control how it handles tabs, sentence endings, and so on. Look at the documentation for the `textwrap.TextWrapper` class for further details.

2.17. Handling HTML and XML Entities in Text

Problem

You want to replace HTML or XML entities such as `&entity;` or `&#code;` with their corresponding text. Alternatively, you need to produce text, but escape certain characters (e.g., <, >, or &).

Solution

If you are producing text, replacing special characters such as < or > is relatively easy if you use the `html.escape()` function. For example:

```
>>> s = 'Elements are written as "<tag>text</tag>".'
>>> import html
>>> print(s)
Elements are written as "<tag>text</tag>".
>>> print(html.escape(s))
Elements are written as &quot;&lt;tag&gt;text&lt;/tag&gt;&quot;.

>>> # Disable escaping of quotes
>>> print(html.escape(s, quote=False))
Elements are written as "&lt;tag&gt;text&lt;/tag&gt;".
>>>
```

If you're trying to emit text as ASCII and want to embed character code entities for non-ASCII characters, you can use the `errors='xmlcharrefreplace'` argument to various I/O-related functions to do it. For example:

```
>>> s = 'Spicy Jalapeño'
>>> s.encode('ascii', errors='xmlcharrefreplace')
b'Spicy Jalape&#241;o'
>>>
```

To replace entities in text, a different approach is needed. If you're actually processing HTML or XML, try using a proper HTML or XML parser first. Normally, these tools will automatically take care of replacing the values for you during parsing and you don't need to worry about it.

If, for some reason, you've received bare text with some entities in it and you want them replaced manually, you can usually do it using various utility functions/methods associated with HTML or XML parsers. For example:

```
>>> s = 'Spicy &quot;Jalape&#241;o&quot.'
>>> from html.parser import HTMLParser
>>> p = HTMLParser()
>>> p.unescape(s)
'Spicy "Jalapeño".'
>>>

>>> t = 'The prompt is &gt;&gt;&gt;'
>>> from xml.sax.saxutils import unescape
>>> unescape(t)
'The prompt is >>>'
>>>
```

Discussion

Proper escaping of special characters is an easily overlooked detail of generating HTML or XML. This is especially true if you're generating such output yourself using `print()` or other basic string formatting features. Using a utility function such as `html.escape()` is an easy solution.

If you need to process text in the other direction, various utility functions, such as `xml.sax.saxutils.unescape()`, can help. However, you really need to investigate the use of a proper parser. For example, if processing HTML or XML, using a parsing module such as `html.parser` or `xml.etree.ElementTree` should already take care of details related to replacing entities in the input text for you.

2.18. Tokenizing Text

Problem

You have a string that you want to parse left to right into a stream of tokens.

Solution

Suppose you have a string of text such as this:

```
text = 'foo = 23 + 42 * 10'
```

To tokenize the string, you need to do more than merely match patterns. You need to have some way to identify the kind of pattern as well. For instance, you might want to turn the string into a sequence of pairs like this:

```
tokens = [('NAME', 'foo'), ('EQ','='), ('NUM', '23'), ('PLUS','+'),
          ('NUM', '42'), ('TIMES', '*'), ('NUM', 10')]
```

To do this kind of splitting, the first step is to define all of the possible tokens, including whitespace, by regular expression patterns using named capture groups such as this:

```
import re
NAME = r'(?P<NAME>[a-zA-Z_][a-zA-Z_0-9]*)'
NUM  = r'(?P<NUM>\d+)'
PLUS = r'(?P<PLUS>\+)'
TIMES = r'(?P<TIMES>\*)'
EQ    = r'(?P<EQ>=)'
WS    = r'(?P<WS>\s+)'

master_pat = re.compile('|'.join([NAME, NUM, PLUS, TIMES, EQ, WS]))
```

In these `re` patterns, the `?P<TOKENNAME>` convention is used to assign a name to the pattern. This will be used later.

Next, to tokenize, use the little-known `scanner()` method of pattern objects. This method creates a scanner object in which repeated calls to `match()` step through the supplied text one match at a time. Here is an interactive example of how a scanner object works:

```
>>> scanner = master_pat.scanner('foo = 42')
>>> scanner.match()
<_sre.SRE_Match object at 0x100677738>
>>> _.lastgroup, _.group()
('NAME', 'foo')
>>> scanner.match()
<_sre.SRE_Match object at 0x100677738>
>>> _.lastgroup, _.group()
('WS', ' ')
>>> scanner.match()
<_sre.SRE_Match object at 0x100677738>
>>> _.lastgroup, _.group()
('EQ', '=')
>>> scanner.match()
<_sre.SRE_Match object at 0x100677738>
>>> _.lastgroup, _.group()
('WS', ' ')
>>> scanner.match()
<_sre.SRE_Match object at 0x100677738>
>>> _.lastgroup, _.group()
('NUM', '42')
>>> scanner.match()
>>>
```

To take this technique and put it into code, it can be cleaned up and easily packaged into a generator like this:

```
from collections import namedtuple

Token = namedtuple('Token', ['type','value'])

def generate_tokens(pat, text):
    scanner = pat.scanner(text)
    for m in iter(scanner.match, None):
        yield Token(m.lastgroup, m.group())

# Example use
for tok in generate_tokens(master_pat, 'foo = 42'):
    print(tok)

# Produces output
# Token(type='NAME', value='foo')
# Token(type='WS', value=' ')
# Token(type='EQ', value='=')
# Token(type='WS', value=' ')
# Token(type='NUM', value='42')
```

If you want to filter the token stream in some way, you can either define more generator functions or use a generator expression. For example, here is how you might filter out all whitespace tokens.

```
tokens = (tok for tok in generate_tokens(master_pat, text)
          if tok.type != 'WS')
for tok in tokens:
    print(tok)
```

Discussion

Tokenizing is often the first step for more advanced kinds of text parsing and handling. To use the scanning technique shown, there are a few important details to keep in mind. First, you must make sure that you identify every possible text sequence that might appear in the input with a correponding `re` pattern. If any nonmatching text is found, scanning simply stops. This is why it was necessary to specify the whitespace (`WS`) token in the example.

The order of tokens in the master regular expression also matters. When matching, `re` tries to match pattens in the order specified. Thus, if a pattern happens to be a substring of a longer pattern, you need to make sure the longer pattern goes first. For example:

```
LT = r'(?P<LT><)'
LE = r'(?P<LE><=)'
EQ = r'(?P<EQ>=)'

master_pat = re.compile('|'.join([LE, LT, EQ]))    # Correct
# master_pat = re.compile('|'.join([LT, LE, EQ]))  # Incorrect
```

The second pattern is wrong because it would match the text <= as the token LT followed by the token EQ, not the single token LE, as was probably desired.

Last, but not least, you need to watch out for patterns that form substrings. For example, suppose you have two pattens like this:

```
PRINT = r'(P<PRINT>print)'
NAME  = r'(P<NAME>[a-zA-Z_][a-zA-Z_0-9]*)'

master_pat = re.compile('|'.join([PRINT, NAME]))

for tok in generate_tokens(master_pat, 'printer'):
    print(tok)

# Outputs :
#  Token(type='PRINT', value='print')
#  Token(type='NAME', value='er')
```

For more advanced kinds of tokenizing, you may want to check out packages such as PyParsing or PLY. An example involving PLY appears in the next recipe.

2.19. Writing a Simple Recursive Descent Parser

Problem

You need to parse text according to a set of grammar rules and perform actions or build an abstract syntax tree representing the input. The grammar is small, so you'd prefer to just write the parser yourself as opposed to using some kind of framework.

Solution

In this problem, we're focused on the problem of parsing text according to a particular grammar. In order to do this, you should probably start by having a formal specification of the grammar in the form of a BNF or EBNF. For example, a grammar for simple arithmetic expressions might look like this:

```
expr ::= expr + term
     |   expr - term
     |   term

term ::= term * factor
     |   term / factor
     |   factor

factor ::= ( expr )
       |   NUM
```

Or, alternatively, in EBNF form:

```
expr ::= term { (+|-) term }*

term ::= factor { (*|/) factor }*

factor ::= ( expr )
       |   NUM
```

In an EBNF, parts of a rule enclosed in `{ ... }*` are optional. The `*` means zero or more repetitions (the same meaning as in a regular expression).

Now, if you're not familiar with the mechanics of working with a BNF, think of it as a specification of substitution or replacement rules where symbols on the left side can be replaced by the symbols on the right (or vice versa). Generally, what happens during parsing is that you try to match the input text to the grammar by making various substitutions and expansions using the BNF. To illustrate, suppose you are parsing an expression such as `3 + 4 * 5`. This expression would first need to be broken down into a token stream, using the techniques described in Recipe 2.18. The result might be a sequence of tokens like this:

```
NUM + NUM * NUM
```

From there, parsing involves trying to match the grammar to input tokens by making substitutions:

```
expr
expr ::= term { (+|-) term }*
expr ::= factor { (*|/) factor }* { (+|-) term }*
expr ::= NUM { (*|/) factor }* { (+|-) term }*
expr ::= NUM { (+|-) term }*
expr ::= NUM + term { (+|-) term }*
expr ::= NUM + factor { (*|/) factor }* { (+|-) term }*
expr ::= NUM + NUM { (*|/) factor}* { (+|-) term }*
expr ::= NUM + NUM * factor { (*|/) factor }* { (+|-) term }*
expr ::= NUM + NUM * NUM { (*|/) factor }* { (+|-) term }*
expr ::= NUM + NUM * NUM { (+|-) term }*
expr ::= NUM + NUM * NUM
```

Following all of the substitution steps takes a bit of coffee, but they're driven by looking at the input and trying to match it to grammar rules. The first input token is a `NUM`, so substitutions first focus on matching that part. Once matched, attention moves to the next token of `+` and so on. Certain parts of the righthand side (e.g., `{ (*/) factor }*`) disappear when it's determined that they can't match the next token. In a successful parse, the entire righthand side is expanded completely to match the input token stream.

With all of the preceding background in place, here is a simple recipe that shows how to build a recursive descent expression evaluator:

```
import re
import collections
```

```
# Token specification
NUM    = r'(?P<NUM>\d+)'
PLUS   = r'(?P<PLUS>\+)'
MINUS  = r'(?P<MINUS>-)'
TIMES  = r'(?P<TIMES>\*)'
DIVIDE = r'(?P<DIVIDE>/)'
LPAREN = r'(?P<LPAREN>\()'
RPAREN = r'(?P<RPAREN>\))'
WS     = r'(?P<WS>\s+)'

master_pat = re.compile('|'.join([NUM, PLUS, MINUS, TIMES,
                                  DIVIDE, LPAREN, RPAREN, WS]))

# Tokenizer
Token = collections.namedtuple('Token', ['type','value'])

def generate_tokens(text):
    scanner = master_pat.scanner(text)
    for m in iter(scanner.match, None):
        tok = Token(m.lastgroup, m.group())
        if tok.type != 'WS':
            yield tok

# Parser
class ExpressionEvaluator:
    '''
    Implementation of a recursive descent parser.   Each method
    implements a single grammar rule.  Use the ._accept() method
    to test and accept the current lookahead token.  Use the ._expect()
    method to exactly match and discard the next token on on the input
    (or raise a SyntaxError if it doesn't match).
    '''

    def parse(self,text):
        self.tokens = generate_tokens(text)
        self.tok = None             # Last symbol consumed
        self.nexttok = None         # Next symbol tokenized
        self._advance()             # Load first lookahead token
        return self.expr()

    def _advance(self):
        'Advance one token ahead'
        self.tok, self.nexttok = self.nexttok, next(self.tokens, None)

    def _accept(self,toktype):
        'Test and consume the next token if it matches toktype'
        if self.nexttok and self.nexttok.type == toktype:
            self._advance()
            return True
        else:
            return False
```

```
    def _expect(self,toktype):
        'Consume next token if it matches toktype or raise SyntaxError'
        if not self._accept(toktype):
            raise SyntaxError('Expected ' + toktype)

    # Grammar rules follow

    def expr(self):
        "expression ::= term { ('+'|'-') term }*"

        exprval = self.term()
        while self._accept('PLUS') or self._accept('MINUS'):
            op = self.tok.type
            right = self.term()
            if op == 'PLUS':
                exprval += right
            elif op == 'MINUS':
                exprval -= right
        return exprval

    def term(self):
        "term ::= factor { ('*'|'/') factor }*"

        termval = self.factor()
        while self._accept('TIMES') or self._accept('DIVIDE'):
            op = self.tok.type
            right = self.factor()
            if op == 'TIMES':
                termval *= right
            elif op == 'DIVIDE':
                termval /= right
        return termval

    def factor(self):
        "factor ::= NUM | ( expr )"

        if self._accept('NUM'):
            return int(self.tok.value)
        elif self._accept('LPAREN'):
            exprval = self.expr()
            self._expect('RPAREN')
            return exprval
        else:
            raise SyntaxError('Expected NUMBER or LPAREN')
```

Here is an example of using the `ExpressionEvaluator` class interactively:

```
>>> e = ExpressionEvaluator()
>>> e.parse('2')
2
>>> e.parse('2 + 3')
5
```

```
>>> e.parse('2 + 3 * 4')
14
>>> e.parse('2 + (3 + 4) * 5')
37
>>> e.parse('2 + (3 + * 4)')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "exprparse.py", line 40, in parse
    return self.expr()
  File "exprparse.py", line 67, in expr
    right = self.term()
  File "exprparse.py", line 77, in term
    termval = self.factor()
  File "exprparse.py", line 93, in factor
    exprval = self.expr()
  File "exprparse.py", line 67, in expr
    right = self.term()
  File "exprparse.py", line 77, in term
    termval = self.factor()
  File "exprparse.py", line 97, in factor
    raise SyntaxError("Expected NUMBER or LPAREN")
SyntaxError: Expected NUMBER or LPAREN
>>>
```

If you want to do something other than pure evaluation, you need to change the `ExpressionEvaluator` class to do something else. For example, here is an alternative implementation that constructs a simple parse tree:

```
class ExpressionTreeBuilder(ExpressionEvaluator):
    def expr(self):
        "expression ::= term { ('+'|'-') term }"

        exprval = self.term()
        while self._accept('PLUS') or self._accept('MINUS'):
            op = self.tok.type
            right = self.term()
            if op == 'PLUS':
                exprval = ('+', exprval, right)
            elif op == 'MINUS':
                exprval = ('-', exprval, right)
        return exprval

    def term(self):
        "term ::= factor { ('*'|'/') factor }"

        termval = self.factor()
        while self._accept('TIMES') or self._accept('DIVIDE'):
            op = self.tok.type
            right = self.factor()
            if op == 'TIMES':
                termval = ('*', termval, right)
            elif op == 'DIVIDE':
```

```
                termval = ('/', termval, right)
        return termval

    def factor(self):
        'factor ::= NUM | ( expr )'

        if self._accept('NUM'):
            return int(self.tok.value)
        elif self._accept('LPAREN'):
            exprval = self.expr()
            self._expect('RPAREN')
            return exprval
        else:
            raise SyntaxError('Expected NUMBER or LPAREN')
```

The following example shows how it works:

```
>>> e = ExpressionTreeBuilder()
>>> e.parse('2 + 3')
('+', 2, 3)
>>> e.parse('2 + 3 * 4')
('+', 2, ('*', 3, 4))
>>> e.parse('2 + (3 + 4) * 5')
('+', 2, ('*', ('+', 3, 4), 5))
>>> e.parse('2 + 3 + 4')
('+', ('+', 2, 3), 4)
>>>
```

Discussion

Parsing is a huge topic that generally occupies students for the first three weeks of a compilers course. If you are seeking background knowledge about grammars, parsing algorithms, and other information, a compilers book is where you should turn. Needless to say, all of that can't be repeated here.

Nevertheless, the overall idea of writing a recursive descent parser is generally simple. To start, you take every grammar rule and you turn it into a function or method. Thus, if your grammar looks like this:

```
expr ::= term { ('+'|'-') term }*

term ::= factor { ('*'|'/') factor }*

factor ::= '(' expr ')'
       |   NUM
```

You start by turning it into a set of methods like this:

```
class ExpressionEvaluator:
    ...
    def expr(self):
        ...
```

```
    def term(self):
        ...

    def factor(self):
        ...
```

The task of each method is simple—it must walk from left to right over each part of the grammar rule, consuming tokens in the process. In a sense, the goal of the method is to either consume the rule or generate a syntax error if it gets stuck. To do this, the following implementation techniques are applied:

- If the next symbol in the rule is the name of another grammar rule (e.g., `term` or `factor`), you simply call the method with the same name. This is the "descent" part of the algorithm—control descends into another grammar rule. Sometimes rules will involve calls to methods that are already executing (e.g., the call to `expr` in the `factor ::= '(' expr ')'` rule). This is the "recursive" part of the algorithm.
- If the next symbol in the rule has to be a specific symbol (e.g., `(`), you look at the next token and check for an exact match. If it doesn't match, it's a syntax error. The `_expect()` method in this recipe is used to perform these steps.
- If the next symbol in the rule could be a few possible choices (e.g., `+` or `-`), you have to check the next token for each possibility and advance only if a match is made. This is the purpose of the `_accept()` method in this recipe. It's kind of like a weaker version of the `_expect()` method in that it will advance if a match is made, but if not, it simply backs off without raising an error (thus allowing further checks to be made).
- For grammar rules where there are repeated parts (e.g., such as in the rule `expr ::= term { ('+'|'-') term }*`), the repetition gets implemented by a `while` loop. The body of the loop will generally collect or process all of the repeated items until no more are found.
- Once an entire grammar rule has been consumed, each method returns some kind of result back to the caller. This is how values propagate during parsing. For example, in the expression evaluator, return values will represent partial results of the expression being parsed. Eventually they all get combined together in the topmost grammar rule method that executes.

Although a simple example has been shown, recursive descent parsers can be used to implement rather complicated parsers. For example, Python code itself is interpreted by a recursive descent parser. If you're so inclined, you can look at the underlying grammar by inspecting the file *Grammar/Grammar* in the Python source. That said, there are still numerous pitfalls and limitations with making a parser by hand.

One such limitation of recursive descent parsers is that they can't be written for grammar rules involving any kind of left recursion. For example, suppose you need to translate a rule like this:

```
items ::= items ',' item
        |  item
```

To do it, you might try to use the `items()` method like this:

```
def items(self):
    itemsval = self.items()
    if itemsval and self._accept(','):
         itemsval.append(self.item())
    else:
         itemsval = [ self.item() ]
```

The only problem is that it doesn't work. In fact, it blows up with an infinite recursion error.

You can also run into tricky issues concerning the grammar rules themselves. For example, you might have wondered whether or not expressions could have been described by this more simple grammar:

```
expr ::= factor { ('+'|'-'|'*'|'/') factor }*

factor ::= '(' expression ')'
        |   NUM
```

This grammar technically "works," but it doesn't observe the standard arithmetic rules concerning order of evaluation. For example, the expression "3 + 4 * 5" would get evaluated as "35" instead of the expected result of "23." The use of separate "expr" and "term" rules is there to make evaluation work correctly.

For really complicated grammars, you are often better off using parsing tools such as PyParsing or PLY. This is what the expression evaluator code looks like using PLY:

```
from ply.lex import lex
from ply.yacc import yacc

# Token list
tokens = [ 'NUM', 'PLUS', 'MINUS', 'TIMES', 'DIVIDE', 'LPAREN', 'RPAREN' ]

# Ignored characters

t_ignore = ' \t\n'

# Token specifications (as regexs)
t_PLUS   = r'\+'
t_MINUS  = r'-'
t_TIMES  = r'\*'
t_DIVIDE = r'/'
t_LPAREN = r'\('
```

```
t_RPAREN = r'\)'

# Token processing functions
def t_NUM(t):
    r'\d+'
    t.value = int(t.value)
    return t

# Error handler
def t_error(t):
    print('Bad character: {!r}'.format(t.value[0]))
    t.skip(1)

# Build the lexer
lexer = lex()

# Grammar rules and handler functions
def p_expr(p):
    '''
    expr : expr PLUS term
         | expr MINUS term
    '''
    if p[2] == '+':
        p[0] = p[1] + p[3]
    elif p[2] == '-':
        p[0] = p[1] - p[3]

def p_expr_term(p):
    '''
    expr : term
    '''
    p[0] = p[1]

def p_term(p):
    '''
    term : term TIMES factor
         | term DIVIDE factor
    '''
    if p[2] == '*':
        p[0] = p[1] * p[3]
    elif p[2] == '/':
        p[0] = p[1] / p[3]

def p_term_factor(p):
    '''
    term : factor
    '''
    p[0] = p[1]

def p_factor(p):
    '''
    factor : NUM
```

```
        '''
        p[0] = p[1]

    def p_factor_group(p):
        '''
        factor : LPAREN expr RPAREN
        '''
        p[0] = p[2]

    def p_error(p):
        print('Syntax error')

    parser = yacc()
```

In this code, you'll find that everything is specified at a much higher level. You simply write regular expressions for the tokens and high-level handling functions that execute when various grammar rules are matched. The actual mechanics of running the parser, accepting tokens, and so forth is implemented entirely by the library.

Here is an example of how the resulting parser object gets used:

```
>>> parser.parse('2')
2
>>> parser.parse('2+3')
5
>>> parser.parse('2+(3+4)*5')
37
>>>
```

If you need a bit more excitement in your programming, writing parsers and compilers can be a fun project. Again, a compilers textbook will have a lot of low-level details underlying theory. However, many fine resources can also be found online. Python's own `ast` module is also worth a look.

2.20. Performing Text Operations on Byte Strings

Problem

You want to perform common text operations (e.g., stripping, searching, and replacement) on byte strings.

Solution

Byte strings already support most of the same built-in operations as text strings. For example:

```
>>> data = b'Hello World'
>>> data[0:5]
b'Hello'
>>> data.startswith(b'Hello')
True
>>> data.split()
[b'Hello', b'World']
>>> data.replace(b'Hello', b'Hello Cruel')
b'Hello Cruel World'
>>>
```

Such operations also work with byte arrays. For example:

```
>>> data = bytearray(b'Hello World')
>>> data[0:5]
bytearray(b'Hello')
>>> data.startswith(b'Hello')
True
>>> data.split()
[bytearray(b'Hello'), bytearray(b'World')]
>>> data.replace(b'Hello', b'Hello Cruel')
bytearray(b'Hello Cruel World')
>>>
```

You can apply regular expression pattern matching to byte strings, but the patterns themselves need to be specified as bytes. For example:

```
>>>
>>> data = b'FOO:BAR,SPAM'
>>> import re
>>> re.split('[:,]',data)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/usr/local/lib/python3.3/re.py", line 191, in split
    return _compile(pattern, flags).split(string, maxsplit)
TypeError: can't use a string pattern on a bytes-like object

>>> re.split(b'[:,]',data)     # Notice: pattern as bytes
[b'FOO', b'BAR', b'SPAM']
>>>
```

Discussion

For the most part, almost all of the operations available on text strings will work on byte strings. However, there are a few notable differences to be aware of. First, indexing of byte strings produces integers, not individual characters. For example:

```
>>> a = 'Hello World'     # Text string
>>> a[0]
'H'
>>> a[1]
'e'
>>> b = b'Hello World'    # Byte string
```

```
>>> b[0]
72
>>> b[1]
101
>>>
```

This difference in semantics can affect programs that try to process byte-oriented data on a character-by-character basis.

Second, byte strings don't provide a nice string representation and don't print cleanly unless first decoded into a text string. For example:

```
>>> s = b'Hello World'
>>> print(s)
b'Hello World'               # Observe b'...'
>>> print(s.decode('ascii'))
Hello World
>>>
```

Similarly, there are no string formatting operations available to byte strings.

```
>>> b'%10s %10d %10.2f' % (b'ACME', 100, 490.1)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for %: 'bytes' and 'tuple'

>>> b'{} {} {}'.format(b'ACME', 100, 490.1)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'bytes' object has no attribute 'format'
>>>
```

If you want to do any kind of formatting applied to byte strings, it should be done using normal text strings and encoding. For example:

```
>>> '{:10s} {:10d} {:10.2f}'.format('ACME', 100, 490.1).encode('ascii')
b'ACME              100     490.10'
>>>
```

Finally, you need to be aware that using a byte string can change the semantics of certain operations—especially those related to the filesystem. For example, if you supply a filename encoded as bytes instead of a text string, it usually disables filename encoding/decoding. For example:

```
>>> # Write a UTF-8 filename
>>> with open('jalape\xf1o.txt', 'w') as f:
...     f.write('spicy')
...

>>> # Get a directory listing
>>> import os
>>> os.listdir('.')          # Text string (names are decoded)
['jalapeño.txt']
```

```
>>> os.listdir(b'.')         # Byte string (names left as bytes)
[b'jalapen\xcc\x83o.txt']
>>>
```

Notice in the last part of this example how giving a byte string as the directory name caused the resulting filenames to be returned as undecoded bytes. The filename shown in the directory listing contains raw UTF-8 encoding. See Recipe 5.15 for some related issues concerning filenames.

As a final comment, some programmers might be inclined to use byte strings as an alternative to text strings due to a possible performance improvement. Although it's true that manipulating bytes tends to be slightly more efficient than text (due to the inherent overhead related to Unicode), doing so usually leads to very messy and non-idiomatic code. You'll often find that byte strings don't play well with a lot of other parts of Python, and that you end up having to perform all sorts of manual encoding/decoding operations yourself to get things to work right. Frankly, if you're working with text, use normal text strings in your program, not byte strings.

CHAPTER 3

Numbers, Dates, and Times

Performing mathematical calculations with integers and floating-point numbers is easy in Python. However, if you need to perform calculations with fractions, arrays, or dates and times, a bit more work is required. The focus of this chapter is on such topics.

3.1. Rounding Numerical Values

Problem

You want to round a floating-point number to a fixed number of decimal places.

Solution

For simple rounding, use the built-in `round(value, ndigits)` function. For example:

```
>>> round(1.23, 1)
1.2
>>> round(1.27, 1)
1.3
>>> round(-1.27, 1)
-1.3
>>> round(1.25361,3)
1.254
>>>
```

When a value is exactly halfway between two choices, the behavior of round is to round to the nearest even digit. That is, values such as 1.5 or 2.5 both get rounded to 2.

The number of digits given to `round()` can be negative, in which case rounding takes place for tens, hundreds, thousands, and so on. For example:

```
>>> a = 1627731
>>> round(a, -1)
1627730
```

```
>>> round(a, -2)
1627700
>>> round(a, -3)
1628000
>>>
```

Discussion

Don't confuse rounding with formatting a value for output. If your goal is simply to output a numerical value with a certain number of decimal places, you don't typically need to use `round()`. Instead, just specify the desired precision when formatting. For example:

```
>>> x = 1.23456
>>> format(x, '0.2f')
'1.23'
>>> format(x, '0.3f')
'1.235'
>>> 'value is {:0.3f}'.format(x)
'value is 1.235'
>>>
```

Also, resist the urge to round floating-point numbers to "fix" perceived accuracy problems. For example, you might be inclined to do this:

```
>>> a = 2.1
>>> b = 4.2
>>> c = a + b
>>> c
6.300000000000001
>>> c = round(c, 2)      # "Fix" result (???)
>>> c
6.3
>>>
```

For most applications involving floating point, it's simply not necessary (or recommended) to do this. Although there are small errors introduced into calculations, the behavior of those errors are understood and tolerated. If avoiding such errors is important (e.g., in financial applications, perhaps), consider the use of the `decimal` module, which is discussed in the next recipe.

3.2. Performing Accurate Decimal Calculations

Problem

You need to perform accurate calculations with decimal numbers, and don't want the small errors that naturally occur with floats.

Solution

A well-known issue with floating-point numbers is that they can't accurately represent all base-10 decimals. Moreover, even simple mathematical calculations introduce small errors. For example:

```
>>> a = 4.2
>>> b = 2.1
>>> a + b
6.300000000000001
>>> (a + b) == 6.3
False
>>>
```

These errors are a "feature" of the underlying CPU and the IEEE 754 arithmetic performed by its floating-point unit. Since Python's float data type stores data using the native representation, there's nothing you can do to avoid such errors if you write your code using `float` instances.

If you want more accuracy (and are willing to give up some performance), you can use the `decimal` module:

```
>>> from decimal import Decimal
>>> a = Decimal('4.2')
>>> b = Decimal('2.1')
>>> a + b
Decimal('6.3')
>>> print(a + b)
6.3
>>> (a + b) == Decimal('6.3')
True
>>>
```

At first glance, it might look a little weird (i.e., specifying numbers as strings). However, `Decimal` objects work in every way that you would expect them to (supporting all of the usual math operations, etc.). If you print them or use them in string formatting functions, they look like normal numbers.

A major feature of `decimal` is that it allows you to control different aspects of calculations, including number of digits and rounding. To do this, you create a local context and change its settings. For example:

```
>>> from decimal import localcontext
>>> a = Decimal('1.3')
>>> b = Decimal('1.7')
>>> print(a / b)
0.7647058823529411764705882353
>>> with localcontext() as ctx:
...     ctx.prec = 3
...     print(a / b)
...
```

```
0.765
>>> with localcontext() as ctx:
...     ctx.prec = 50
...     print(a / b)
...
0.76470588235294117647058823529411764705882352941176
>>>
```

Discussion

The `decimal` module implements IBM's "General Decimal Arithmetic Specification." Needless to say, there are a huge number of configuration options that are beyond the scope of this book.

Newcomers to Python might be inclined to use the `decimal` module to work around perceived accuracy problems with the `float` data type. However, it's really important to understand your application domain. If you're working with science or engineering problems, computer graphics, or most things of a scientific nature, it's simply more common to use the normal floating-point type. For one, very few things in the real world are measured to the 17 digits of accuracy that floats provide. Thus, tiny errors introduced in calculations just don't matter. Second, the performance of native floats is significantly faster—something that's important if you're performing a large number of calculations.

That said, you can't ignore the errors completely. Mathematicians have spent a lot of time studying various algorithms, and some handle errors better than others. You also have to be a little careful with effects due to things such as subtractive cancellation and adding large and small numbers together. For example:

```
>>> nums = [1.23e+18, 1, -1.23e+18]
>>> sum(nums)    # Notice how 1 disappears
0.0
>>>
```

This latter example can be addressed by using a more accurate implementation in `math.fsum()`:

```
>>> import math
>>> math.fsum(nums)
1.0
>>>
```

However, for other algorithms, you really need to study the algorithm and understand its error propagation properties.

All of this said, the main use of the `decimal` module is in programs involving things such as finance. In such programs, it is extremely annoying to have small errors creep into the calculation. Thus, `decimal` provides a way to avoid that. It is also common to encounter `Decimal` objects when Python interfaces with databases—again, especially when accessing financial data.

3.3. Formatting Numbers for Output

Problem

You need to format a number for output, controlling the number of digits, alignment, inclusion of a thousands separator, and other details.

Solution

To format a single number for output, use the built-in `format()` function. For example:

```
>>> x = 1234.56789

>>> # Two decimal places of accuracy
>>> format(x, '0.2f')
'1234.57'

>>> # Right justified in 10 chars, one-digit accuracy
>>> format(x, '>10.1f')
'    1234.6'

>>> # Left justified
>>> format(x, '<10.1f')
'1234.6    '

>>> # Centered
>>> format(x, '^10.1f')
'  1234.6  '

>>> # Inclusion of thousands separator
>>> format(x, ',')
'1,234.56789'
>>> format(x, '0,.1f')
'1,234.6'
>>>
```

If you want to use exponential notation, change the `f` to an `e` or `E`, depending on the case you want used for the exponential specifier. For example:

```
>>> format(x, 'e')
'1.234568e+03'
>>> format(x, '0.2E')
'1.23E+03'
>>>
```

The general form of the width and precision in both cases is `'[<>^]?width[,]?(.digits)?'` where `width` and `digits` are integers and `?` signifies optional parts. The same format codes are also used in the `.format()` method of strings. For example:

```
>>> 'The value is {:0,.2f}'.format(x)
'The value is 1,234.57'
>>>
```

Discussion

Formatting numbers for output is usually straightforward. The technique shown works for both floating-point numbers and `Decimal` numbers in the `decimal` module.

When the number of digits is restricted, values are rounded away according to the same rules of the `round()` function. For example:

```
>>> x
1234.56789
>>> format(x, '0.1f')
'1234.6'
>>> format(-x, '0.1f')
'-1234.6'
>>>
```

Formatting of values with a thousands separator is not locale aware. If you need to take that into account, you might investigate functions in the `locale` module. You can also swap separator characters using the `translate()` method of strings. For example:

```
>>> swap_separators = { ord('.'):',', ord(','):'.' }
>>> format(x, ',').translate(swap_separators)
'1.234,56789'
>>>
```

In a lot of Python code, numbers are formatted using the `%` operator. For example:

```
>>> '%0.2f' % x
'1234.57'
>>> '%10.1f' % x
'    1234.6'
>>> '%-10.1f' % x
'1234.6    '
>>>
```

This formatting is still acceptable, but less powerful than the more modern `format()` method. For example, some features (e.g., adding thousands separators) aren't supported when using the `%` operator to format numbers.

3.4. Working with Binary, Octal, and Hexadecimal Integers

Problem

You need to convert or output integers represented by binary, octal, or hexadecimal digits.

Solution

To convert an integer into a binary, octal, or hexadecimal text string, use the `bin()`, `oct()`, or `hex()` functions, respectively:

```
>>> x = 1234
>>> bin(x)
'0b10011010010'
>>> oct(x)
'0o2322'
>>> hex(x)
'0x4d2'
>>>
```

Alternatively, you can use the `format()` function if you don't want the `0b`, `0o`, or `0x` prefixes to appear. For example:

```
>>> format(x, 'b')
'10011010010'
>>> format(x, 'o')
'2322'
>>> format(x, 'x')
'4d2'
>>>
```

Integers are signed, so if you are working with negative numbers, the output will also include a sign. For example:

```
>>> x = -1234
>>> format(x, 'b')
'-10011010010'
>>> format(x, 'x')
'-4d2'
>>>
```

If you need to produce an unsigned value instead, you'll need to add in the maximum value to set the bit length. For example, to show a 32-bit value, use the following:

```
>>> x = -1234
>>> format(2**32 + x, 'b')
'11111111111111111111101100101110'
>>> format(2**32 + x, 'x')
```

```
'fffffb2e'
>>>
```

To convert integer strings in different bases, simply use the `int()` function with an appropriate base. For example:

```
>>> int('4d2', 16)
1234
>>> int('10011010010', 2)
1234
>>>
```

Discussion

For the most part, working with binary, octal, and hexadecimal integers is straightforward. Just remember that these conversions only pertain to the conversion of integers to and from a textual representation. Under the covers, there's just one integer type.

Finally, there is one caution for programmers who use octal. The Python syntax for specifying octal values is slightly different than many other languages. For example, if you try something like this, you'll get a syntax error:

```
>>> import os
>>> os.chmod('script.py', 0755)
  File "<stdin>", line 1
    os.chmod('script.py', 0755)
                          ^
SyntaxError: invalid token
>>>
```

Make sure you prefix the octal value with `0o`, as shown here:

```
>>> os.chmod('script.py', 0o755)
>>>
```

3.5. Packing and Unpacking Large Integers from Bytes

Problem

You have a byte string and you need to unpack it into an integer value. Alternatively, you need to convert a large integer back into a byte string.

Solution

Suppose your program needs to work with a 16-element byte string that holds a 128-bit integer value. For example:

```
data = b'\x00\x124V\x00x\x90\xab\x00\xcd\xef\x01\x00#\x004'
```

To interpret the bytes as an integer, use `int.from_bytes()`, and specify the byte ordering like this:

```
>>> len(data)
16
>>> int.from_bytes(data, 'little')
69120565665751139577663547927094891008
>>> int.from_bytes(data, 'big')
94522842520747284487117727783387188
>>>
```

To convert a large integer value back into a byte string, use the `int.to_bytes()` method, specifying the number of bytes and the byte order. For example:

```
>>> x = 94522842520747284487117727783387188
>>> x.to_bytes(16, 'big')
b'\x00\x124V\x00x\x90\xab\x00\xcd\xef\x01\x00#\x004'
>>> x.to_bytes(16, 'little')
b'4\x00#\x00\x01\xef\xcd\x00\xab\x90x\x00V4\x12\x00'
>>>
```

Discussion

Converting large integer values to and from byte strings is not a common operation. However, it sometimes arises in certain application domains, such as cryptography or networking. For instance, IPv6 network addresses are represented as 128-bit integers. If you are writing code that needs to pull such values out of a data record, you might face this problem.

As an alternative to this recipe, you might be inclined to unpack values using the `struct` module, as described in Recipe 6.11. This works, but the size of integers that can be unpacked with `struct` is limited. Thus, you would need to unpack multiple values and combine them to create the final value. For example:

```
>>> data
b'\x00\x124V\x00x\x90\xab\x00\xcd\xef\x01\x00#\x004'
>>> import struct
>>> hi, lo = struct.unpack('>QQ', data)
>>> (hi << 64) + lo
94522842520747284487117727783387188
>>>
```

The specification of the byte order (`little` or `big`) just indicates whether the bytes that make up the integer value are listed from the least to most significant or the other way around. This is easy to view using a carefully crafted hexadecimal value:

```
>>> x = 0x01020304
>>> x.to_bytes(4, 'big')
b'\x01\x02\x03\x04'
>>> x.to_bytes(4, 'little')
```

```
b'\x04\x03\x02\x01'
>>>
```

If you try to pack an integer into a byte string, but it won't fit, you'll get an error. You can use the `int.bit_length()` method to determine how many bits are required to store a value if needed:

```
>>> x = 523 ** 23
>>> x
335381300113661875107536852714019056160355655333978849017944067
>>> x.to_bytes(16, 'little')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
OverflowError: int too big to convert
>>> x.bit_length()
208
>>> nbytes, rem = divmod(x.bit_length(), 8)
>>> if rem:
...     nbytes += 1
...
>>>
>>> x.to_bytes(nbytes, 'little')
b'\x03X\xf1\x82iT\x96\xac\xc7c\x16\xf3\xb9\xcf...\xd0'
>>>
```

3.6. Performing Complex-Valued Math

Problem

Your code for interacting with the latest web authentication scheme has encountered a singularity and your only solution is to go around it in the complex plane. Or maybe you just need to perform some calculations using complex numbers.

Solution

Complex numbers can be specified using the `complex(real, imag)` function or by floating-point numbers with a `j` suffix. For example:

```
>>> a = complex(2, 4)
>>> b = 3 - 5j
>>> a
(2+4j)
>>> b
(3-5j)
>>>
```

The real, imaginary, and conjugate values are easy to obtain, as shown here:

```
>>> a.real
2.0
```

```
>>> a.imag
4.0
>>> a.conjugate()
(2-4j)
>>>
```

In addition, all of the usual mathematical operators work:

```
>>> a + b
(5-1j)
>>> a * b
(26+2j)
>>> a / b
(-0.411764705882353529+0.6470588235294118j)
>>> abs(a)
4.47213595499958
>>>
```

To perform additional complex-valued functions such as sines, cosines, or square roots, use the `cmath` module:

```
>>> import cmath
>>> cmath.sin(a)
(24.83130584894638-11.356612711218174j)
>>> cmath.cos(a)
(-11.36423470640106-24.814651485634187j)
>>> cmath.exp(a)
(-4.829809383269385-5.5920560936409816j)
>>>
```

Discussion

Most of Python's math-related modules are aware of complex values. For example, if you use `numpy`, it is straightforward to make arrays of complex values and perform operations on them:

```
>>> import numpy as np
>>> a = np.array([2+3j, 4+5j, 6-7j, 8+9j])
>>> a
array([ 2.+3.j,  4.+5.j,  6.-7.j,  8.+9.j])
>>> a + 2
array([  4.+3.j,   6.+5.j,   8.-7.j,  10.+9.j])
>>> np.sin(a)
array([    9.15449915  -4.16890696j,   -56.16227422 -48.50245524j,
        -153.20827755-526.47684926j,  4008.42651446-589.49948373j])
>>>
```

Python's standard mathematical functions do not produce complex values by default, so it is unlikely that such a value would accidentally show up in your code. For example:

```
>>> import math
>>> math.sqrt(-1)
Traceback (most recent call last):
```

```
  File "<stdin>", line 1, in <module>
ValueError: math domain error
>>>
```

If you want complex numbers to be produced as a result, you have to explicitly use `cmath` or declare the use of a complex type in libraries that know about them. For example:

```
>>> import cmath
>>> cmath.sqrt(-1)
1j
>>>
```

3.7. Working with Infinity and NaNs

Problem

You need to create or test for the floating-point values of infinity, negative infinity, or NaN (not a number).

Solution

Python has no special syntax to represent these special floating-point values, but they can be created using `float()`. For example:

```
>>> a = float('inf')
>>> b = float('-inf')
>>> c = float('nan')
>>> a
inf
>>> b
-inf
>>> c
nan
>>>
```

To test for the presence of these values, use the `math.isinf()` and `math.isnan()` functions. For example:

```
>>> math.isinf(a)
True
>>> math.isnan(c)
True
>>>
```

Discussion

For more detailed information about these special floating-point values, you should refer to the IEEE 754 specification. However, there are a few tricky details to be aware of, especially related to comparisons and operators.

Infinite values will propagate in calculations in a mathematical manner. For example:

```
>>> a = float('inf')
>>> a + 45
inf
>>> a * 10
inf
>>> 10 / a
0.0
>>>
```

However, certain operations are undefined and will result in a NaN result. For example:

```
>>> a = float('inf')
>>> a/a
nan
>>> b = float('-inf')
>>> a + b
nan
>>>
```

NaN values propagate through all operations without raising an exception. For example:

```
>>> c = float('nan')
>>> c + 23
nan
>>> c / 2
nan
>>> c * 2
nan
>>> math.sqrt(c)
nan
>>>
```

A subtle feature of NaN values is that they never compare as equal. For example:

```
>>> c = float('nan')
>>> d = float('nan')
>>> c == d
False
>>> c is d
False
>>>
```

Because of this, the only safe way to test for a NaN value is to use `math.isnan()`, as shown in this recipe.

Sometimes programmers want to change Python's behavior to raise exceptions when operations result in an infinite or NaN result. The `fpectl` module can be used to adjust this behavior, but it is not enabled in a standard Python build, it's platform-dependent, and really only intended for expert-level programmers. See the online Python documentation for further details.

3.8. Calculating with Fractions

Problem

You have entered a time machine and suddenly find yourself working on elementary-level homework problems involving fractions. Or perhaps you're writing code to make calculations involving measurements made in your wood shop.

Solution

The `fractions` module can be used to perform mathematical calculations involving fractions. For example:

```
>>> from fractions import Fraction
>>> a = Fraction(5, 4)
>>> b = Fraction(7, 16)
>>> print(a + b)
27/16
>>> print(a * b)
35/64

>>> # Getting numerator/denominator
>>> c = a * b
>>> c.numerator
35
>>> c.denominator
64

>>> # Converting to a float
>>> float(c)
0.546875

>>> # Limiting the denominator of a value
>>> print(c.limit_denominator(8))
4/7

>>> # Converting a float to a fraction
>>> x = 3.75
>>> y = Fraction(*x.as_integer_ratio())
>>> y
Fraction(15, 4)
>>>
```

Discussion

Calculating with fractions doesn't arise often in most programs, but there are situations where it might make sense to use them. For example, allowing a program to accept units of measurement in fractions and performing calculations with them in that form might alleviate the need for a user to manually make conversions to decimals or floats.

3.9. Calculating with Large Numerical Arrays

Problem

You need to perform calculations on large numerical datasets, such as arrays or grids.

Solution

For any heavy computation involving arrays, use the NumPy library. The major feature of NumPy is that it gives Python an array object that is much more efficient and better suited for mathematical calculation than a standard Python list. Here is a short example illustrating important behavioral differences between lists and NumPy arrays:

```
>>> # Python lists
>>> x = [1, 2, 3, 4]
>>> y = [5, 6, 7, 8]
>>> x * 2
[1, 2, 3, 4, 1, 2, 3, 4]
>>> x + 10
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can only concatenate list (not "int") to list
>>> x + y
[1, 2, 3, 4, 5, 6, 7, 8]

>>> # Numpy arrays
>>> import numpy as np
>>> ax = np.array([1, 2, 3, 4])
>>> ay = np.array([5, 6, 7, 8])
>>> ax * 2
array([2, 4, 6, 8])
>>> ax + 10
array([11, 12, 13, 14])
>>> ax + ay
array([ 6,  8, 10, 12])
>>> ax * ay
array([ 5, 12, 21, 32])
>>>
```

As you can see, basic mathematical operations involving arrays behave differently. Specifically, scalar operations (e.g., `ax * 2` or `ax + 10`) apply the operation on an element-by-element basis. In addition, performing math operations when both operands are arrays applies the operation to all elements and produces a new array.

The fact that math operations apply to all of the elements simultaneously makes it very easy and fast to compute functions across an entire array. For example, if you want to compute the value of a polynomial:

```
>>> def f(x):
...     return 3*x**2 - 2*x + 7
```

```
...
>>> f(ax)
array([ 8, 15, 28, 47])
>>>
```

NumPy provides a collection of "universal functions" that also allow for array operations. These are replacements for similar functions normally found in the `math` module. For example:

```
>>> np.sqrt(ax)
array([ 1.        ,  1.41421356,  1.73205081,  2.        ])
>>> np.cos(ax)
array([ 0.54030231, -0.41614684, -0.9899925 , -0.65364362])
>>>
```

Using universal functions can be hundreds of times faster than looping over the array elements one at a time and performing calculations using functions in the `math` module. Thus, you should prefer their use whenever possible.

Under the covers, NumPy arrays are allocated in the same manner as in C or Fortran. Namely, they are large, contiguous memory regions consisting of a homogenous data type. Because of this, it's possible to make arrays much larger than anything you would normally put into a Python list. For example, if you want to make a two-dimensional grid of 10,000 by 10,000 floats, it's not an issue:

```
>>> grid = np.zeros(shape=(10000,10000), dtype=float)
>>> grid
array([[ 0.,  0.,  0., ...,  0.,  0.,  0.],
       [ 0.,  0.,  0., ...,  0.,  0.,  0.],
       [ 0.,  0.,  0., ...,  0.,  0.,  0.],
       ...,
       [ 0.,  0.,  0., ...,  0.,  0.,  0.],
       [ 0.,  0.,  0., ...,  0.,  0.,  0.],
       [ 0.,  0.,  0., ...,  0.,  0.,  0.]])
>>>
```

All of the usual operations still apply to all of the elements simultaneously:

```
>>> grid += 10
>>> grid
array([[ 10.,  10.,  10., ...,  10.,  10.,  10.],
       [ 10.,  10.,  10., ...,  10.,  10.,  10.],
       [ 10.,  10.,  10., ...,  10.,  10.,  10.],
       ...,
       [ 10.,  10.,  10., ...,  10.,  10.,  10.],
       [ 10.,  10.,  10., ...,  10.,  10.,  10.],
       [ 10.,  10.,  10., ...,  10.,  10.,  10.]])
>>> np.sin(grid)
array([[-0.54402111, -0.54402111, -0.54402111, ..., -0.54402111,
        -0.54402111, -0.54402111],
       [-0.54402111, -0.54402111, -0.54402111, ..., -0.54402111,
        -0.54402111, -0.54402111],
```

```
       [-0.54402111, -0.54402111, -0.54402111, ..., -0.54402111,
        -0.54402111, -0.54402111],
       ...,
       [-0.54402111, -0.54402111, -0.54402111, ..., -0.54402111,
        -0.54402111, -0.54402111],
       [-0.54402111, -0.54402111, -0.54402111, ..., -0.54402111,
        -0.54402111, -0.54402111],
       [-0.54402111, -0.54402111, -0.54402111, ..., -0.54402111,
        -0.54402111, -0.54402111]])
>>>
```

One extremely notable aspect of NumPy is the manner in which it extends Python's list indexing functionality—especially with multidimensional arrays. To illustrate, make a simple two-dimensional array and try some experiments:

```
>>> a = np.array([[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]])
>>> a
array([[ 1,  2,  3,  4],
       [ 5,  6,  7,  8],
       [ 9, 10, 11, 12]])

>>> # Select row 1
>>> a[1]
array([5, 6, 7, 8])

>>> # Select column 1
>>> a[:,1]
array([ 2,  6, 10])

>>> # Select a subregion and change it
>>> a[1:3, 1:3]
array([[ 6,  7],
       [10, 11]])
>>> a[1:3, 1:3] += 10
>>> a
array([[ 1,  2,  3,  4],
       [ 5, 16, 17,  8],
       [ 9, 20, 21, 12]])

>>> # Broadcast a row vector across an operation on all rows
>>> a + [100, 101, 102, 103]
array([[101, 103, 105, 107],
       [105, 117, 119, 111],
       [109, 121, 123, 115]])
>>> a
array([[ 1,  2,  3,  4],
       [ 5, 16, 17,  8],
       [ 9, 20, 21, 12]])

>>> # Conditional assignment on an array
>>> np.where(a < 10, a, 10)
array([[ 1,  2,  3,  4],
```

```
        [ 5, 10, 10,  8],
        [ 9, 10, 10, 10]])
>>>
```

Discussion

NumPy is the foundation for a huge number of science and engineering libraries in Python. It is also one of the largest and most complicated modules in widespread use. That said, it's still possible to accomplish useful things with NumPy by starting with simple examples and playing around.

One note about usage is that it is relatively common to use the statement `import numpy as np`, as shown in the solution. This simply shortens the name to something that's more convenient to type over and over again in your program.

For more information, you definitely need to visit *http://www.numpy.org*.

3.10. Performing Matrix and Linear Algebra Calculations

Problem

You need to perform matrix and linear algebra operations, such as matrix multiplication, finding determinants, solving linear equations, and so on.

Solution

The NumPy library has a `matrix` object that can be used for this purpose. Matrices are somewhat similar to the array objects described in Recipe 3.9, but follow linear algebra rules for computation. Here is an example that illustrates a few essential features:

```
>>> import numpy as np
>>> m = np.matrix([[1,-2,3],[0,4,5],[7,8,-9]])
>>> m
matrix([[ 1, -2,  3],
        [ 0,  4,  5],
        [ 7,  8, -9]])

>>> # Return transpose
>>> m.T
matrix([[ 1,  0,  7],
        [-2,  4,  8],
        [ 3,  5, -9]])

>>> # Return inverse
>>> m.I
matrix([[ 0.33043478, -0.02608696,  0.09565217],
        [-0.15217391,  0.13043478,  0.02173913],
        [ 0.12173913,  0.09565217, -0.0173913 ]])
```

```
>>> # Create a vector and multiply
>>> v = np.matrix([[2],[3],[4]])
>>> v
matrix([[2],
        [3],
        [4]])
>>> m * v
matrix([[ 8],
        [32],
        [ 2]])
>>>
```

More operations can be found in the `numpy.linalg` subpackage. For example:

```
>>> import numpy.linalg

>>> # Determinant
>>> numpy.linalg.det(m)
-229.99999999999983

>>> # Eigenvalues
>>> numpy.linalg.eigvals(m)
array([-13.11474312,   2.75956154,   6.35518158])

>>> # Solve for x in mx = v
>>> x = numpy.linalg.solve(m, v)
>>> x
matrix([[ 0.96521739],
        [ 0.17391304],
        [ 0.46086957]])
>>> m * x
matrix([[ 2.],
        [ 3.],
        [ 4.]])
>>> v
matrix([[2],
        [3],
        [4]])
>>>
```

Discussion

Linear algebra is obviously a huge topic that's far beyond the scope of this cookbook. However, if you need to manipulate matrices and vectors, NumPy is a good starting point. Visit *http://www.numpy.org* for more detailed information.

3.11. Picking Things at Random

Problem

You want to pick random items out of a sequence or generate random numbers.

Solution

The `random` module has various functions for random numbers and picking random items. For example, to pick a random item out of a sequence, use `random.choice()`:

```
>>> import random
>>> values = [1, 2, 3, 4, 5, 6]
>>> random.choice(values)
2
>>> random.choice(values)
3
>>> random.choice(values)
1
>>> random.choice(values)
4
>>> random.choice(values)
6
>>>
```

To take a sampling of N items where selected items are removed from further consideration, use `random.sample()` instead:

```
>>> random.sample(values, 2)
[6, 2]
>>> random.sample(values, 2)
[4, 3]
>>> random.sample(values, 3)
[4, 3, 1]
>>> random.sample(values, 3)
[5, 4, 1]
>>>
```

If you simply want to shuffle items in a sequence in place, use `random.shuffle()`:

```
>>> random.shuffle(values)
>>> values
[2, 4, 6, 5, 3, 1]
>>> random.shuffle(values)
>>> values
[3, 5, 2, 1, 6, 4]
>>>
```

To produce random integers, use `random.randint()`:

```
>>> random.randint(0,10)
2
```

```
>>> random.randint(0,10)
5
>>> random.randint(0,10)
0
>>> random.randint(0,10)
7
>>> random.randint(0,10)
10
>>> random.randint(0,10)
3
>>>
```

To produce uniform floating-point values in the range 0 to 1, use `random.random()`:

```
>>> random.random()
0.9406677561675867
>>> random.random()
0.133129581343897
>>> random.random()
0.4144991136919316
>>>
```

To get N random-bits expressed as an integer, use `random.getrandbits()`:

```
>>> random.getrandbits(200)
335837000776573622800628485064121869519521710558559406913275
>>>
```

Discussion

The `random` module computes random numbers using the Mersenne Twister algorithm. This is a deterministic algorithm, but you can alter the initial seed by using the `random.seed()` function. For example:

```
random.seed()           # Seed based on system time or os.urandom()
random.seed(12345)      # Seed based on integer given
random.seed(b'bytedata') # Seed based on byte data
```

In addition to the functionality shown, `random()` includes functions for uniform, Gaussian, and other probabality distributions. For example, `random.uniform()` computes uniformly distributed numbers, and `random.gauss()` computes normally distributed numbers. Consult the documentation for information on other supported distributions.

Functions in `random()` should not be used in programs related to cryptography. If you need such functionality, consider using functions in the `ssl` module instead. For example, `ssl.RAND_bytes()` can be used to generate a cryptographically secure sequence of random bytes.

3.12. Converting Days to Seconds, and Other Basic Time Conversions

Problem

You have code that needs to perform simple time conversions, like days to seconds, hours to minutes, and so on.

Solution

To perform conversions and arithmetic involving different units of time, use the `datetime` module. For example, to represent an interval of time, create a `timedelta` instance, like this:

```
>>> from datetime import timedelta
>>> a = timedelta(days=2, hours=6)
>>> b = timedelta(hours=4.5)
>>> c = a + b
>>> c.days
2
>>> c.seconds
37800
>>> c.seconds / 3600
10.5
>>> c.total_seconds() / 3600
58.5
>>>
```

If you need to represent specific dates and times, create `datetime` instances and use the standard mathematical operations to manipulate them. For example:

```
>>> from datetime import datetime
>>> a = datetime(2012, 9, 23)
>>> print(a + timedelta(days=10))
2012-10-03 00:00:00
>>>
>>> b = datetime(2012, 12, 21)
>>> d = b - a
>>> d.days
89
>>> now = datetime.today()
>>> print(now)
2012-12-21 14:54:43.094063
>>> print(now + timedelta(minutes=10))
2012-12-21 15:04:43.094063
>>>
```

When making calculations, it should be noted that `datetime` is aware of leap years. For example:

```
>>> a = datetime(2012, 3, 1)
>>> b = datetime(2012, 2, 28)
>>> a - b
datetime.timedelta(2)
>>> (a - b).days
2
>>> c = datetime(2013, 3, 1)
>>> d = datetime(2013, 2, 28)
>>> (c - d).days
1
>>>
```

Discussion

For most basic date and time manipulation problems, the `datetime` module will suffice. If you need to perform more complex date manipulations, such as dealing with time zones, fuzzy time ranges, calculating the dates of holidays, and so forth, look at the `dateutil` module.

To illustrate, many similar time calculations can be performed with the `dateutil.relativedelta()` function. However, one notable feature is that it fills in some gaps pertaining to the handling of months (and their differing number of days). For instance:

```
>>> a = datetime(2012, 9, 23)
>>> a + timedelta(months=1)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'months' is an invalid keyword argument for this function
>>>

>>> from dateutil.relativedelta import relativedelta
>>> a + relativedelta(months=+1)
datetime.datetime(2012, 10, 23, 0, 0)
>>> a + relativedelta(months=+4)
datetime.datetime(2013, 1, 23, 0, 0)
>>>

>>> # Time between two dates
>>> b = datetime(2012, 12, 21)
>>> d = b - a
>>> d
datetime.timedelta(89)
>>> d = relativedelta(b, a)
>>> d
relativedelta(months=+2, days=+28)
>>> d.months
2
>>> d.days
28
>>>
```

3.13. Determining Last Friday's Date

Problem

You want a general solution for finding a date for the last occurrence of a day of the week. Last Friday, for example.

Solution

Python's `datetime` module has utility functions and classes to help perform calculations like this. A decent, generic solution to this problem looks like this:

```
from datetime import datetime, timedelta

weekdays = ['Monday', 'Tuesday', 'Wednesday', 'Thursday',
            'Friday', 'Saturday', 'Sunday']

def get_previous_byday(dayname, start_date=None):
    if start_date is None:
        start_date = datetime.today()
    day_num = start_date.weekday()
    day_num_target = weekdays.index(dayname)
    days_ago = (7 + day_num - day_num_target) % 7
    if days_ago == 0:
        days_ago = 7
    target_date = start_date - timedelta(days=days_ago)
    return target_date
```

Using this in an interpreter session would look like this:

```
>>> datetime.today()  # For reference
datetime.datetime(2012, 8, 28, 22, 4, 30, 263076)
>>> get_previous_byday('Monday')
datetime.datetime(2012, 8, 27, 22, 3, 57, 29045)
>>> get_previous_byday('Tuesday') # Previous week, not today
datetime.datetime(2012, 8, 21, 22, 4, 12, 629771)
>>> get_previous_byday('Friday')
datetime.datetime(2012, 8, 24, 22, 5, 9, 911393)
>>>
```

The optional `start_date` can be supplied using another `datetime` instance. For example:

```
>>> get_previous_byday('Sunday', datetime(2012, 12, 21))
datetime.datetime(2012, 12, 16, 0, 0)
>>>
```

Discussion

This recipe works by mapping the start date and the target date to their numeric position in the week (with `Monday` as day 0). Modular arithmetic is then used to figure out how many days ago the target date last occurred. From there, the desired date is calculated from the start date by subtracting an appropriate `timedelta` instance.

If you're performing a lot of date calculations like this, you may be better off installing the `python-dateutil` package instead. For example, here is an example of performing the same calculation using the `relativedelta()` function from `dateutil`:

```
>>> from datetime import datetime
>>> from dateutil.relativedelta import relativedelta
>>> from dateutil.rrule import *
>>> d = datetime.now()
>>> print(d)
2012-12-23 16:31:52.718111

>>> # Next Friday
>>> print(d + relativedelta(weekday=FR))
2012-12-28 16:31:52.718111
>>>

>>> # Last Friday
>>> print(d + relativedelta(weekday=FR(-1)))
2012-12-21 16:31:52.718111
>>>
```

3.14. Finding the Date Range for the Current Month

Problem

You have some code that needs to loop over each date in the current month, and want an efficient way to calculate that date range.

Solution

Looping over the dates doesn't require building a list of all the dates ahead of time. You can just calculate the starting and stopping date in the range, then use `datetime.time` `delta` objects to increment the date as you go.

Here's a function that takes any `datetime` object, and returns a tuple containing the first date of the month and the starting date of the next month:

```
from datetime import datetime, date, timedelta
import calendar

def get_month_range(start_date=None):
    if start_date is None:
```

```
        start_date = date.today().replace(day=1)
    _, days_in_month = calendar.monthrange(start_date.year, start_date.month)
    end_date = start_date + timedelta(days=days_in_month)
    return (start_date, end_date)
```

With this in place, it's pretty simple to loop over the date range:

```
>>> a_day = timedelta(days=1)
>>> first_day, last_day = get_month_range()
>>> while first_day < last_day:
...     print(first_day)
...     first_day += a_day
...
2012-08-01
2012-08-02
2012-08-03
2012-08-04
2012-08-05
2012-08-06
2012-08-07
2012-08-08
2012-08-09
#... and so on...
```

Discussion

This recipe works by first calculating a date correponding to the first day of the month. A quick way to do this is to use the `replace()` method of a `date` or `datetime` object to simply set the `days` attribute to 1. One nice thing about the `replace()` method is that it creates the same kind of object that you started with. Thus, if the input was a `date` instance, the result is a `date`. Likewise, if the input was a `datetime` instance, you get a `datetime` instance.

After that, the `calendar.monthrange()` function is used to find out how many days are in the month in question. Any time you need to get basic information about calendars, the `calendar` module can be useful. `monthrange()` is only one such function that returns a tuple containing the day of the week along with the number of days in the month.

Once the number of days in the month is known, the ending date is calculated by adding an appropriate `timedelta` to the starting date. It's subtle, but an important aspect of this recipe is that the ending date is not to be included in the range (it is actually the first day of the next month). This mirrors the behavior of Python's slices and range operations, which also never include the end point.

To loop over the date range, standard math and comparison operators are used. For example, `timedelta` instances can be used to increment the date. The `<` operator is used to check whether a date comes before the ending date.

Ideally, it would be nice to create a function that works like the built-in `range()` function, but for dates. Fortunately, this is extremely easy to implement using a generator:

```
def date_range(start, stop, step):
    while start < stop:
        yield start
        start += step
```

Here is an example of it in use:

```
>>> for d in date_range(datetime(2012, 9, 1), datetime(2012,10,1),
                        timedelta(hours=6)):
...     print(d)
...
2012-09-01 00:00:00
2012-09-01 06:00:00
2012-09-01 12:00:00
2012-09-01 18:00:00
2012-09-02 00:00:00
2012-09-02 06:00:00
...
>>>
```

Again, a major part of the ease of implementation is that dates and times can be manipulated using standard math and comparison operators.

3.15. Converting Strings into Datetimes

Problem

Your application receives temporal data in string format, but you want to convert those strings into `datetime` objects in order to perform nonstring operations on them.

Solution

Python's standard `datetime` module is typically the easy solution for this. For example:

```
>>> from datetime import datetime
>>> text = '2012-09-20'
>>> y = datetime.strptime(text, '%Y-%m-%d')
>>> z = datetime.now()
>>> diff = z - y
>>> diff
datetime.timedelta(3, 77824, 177393)
>>>
```

Discussion

The `datetime.strptime()` method supports a host of formatting codes, like `%Y` for the four-digit year and `%m` for the two-digit month. It's also worth noting that these format-

ting placeholders also work in reverse, in case you need to represent a `datetime` object in string output and make it look nice.

For example, let's say you have some code that generates a datetime object, but you need to format a nice, human-readable date to put in the header of an auto-generated letter or report:

```
>>> z
datetime.datetime(2012, 9, 23, 21, 37, 4, 177393)
>>> nice_z = datetime.strftime(z, '%A %B %d, %Y')
>>> nice_z
'Sunday September 23, 2012'
>>>
```

It's worth noting that the performance of `strptime()` is often much worse than you might expect, due to the fact that it's written in pure Python and it has to deal with all sorts of system locale settings. If you are parsing a lot of dates in your code and you know the precise format, you will probably get much better performance by cooking up a custom solution instead. For example, if you knew that the dates were of the form "YYYY-MM-DD," you could write a function like this:

```
from datetime import datetime
def parse_ymd(s):
    year_s, mon_s, day_s = s.split('-')
    return datetime(int(year_s), int(mon_s), int(day_s))
```

When tested, this function runs over seven times faster than `datetime.strptime()`. This is probably something to consider if you're processing large amounts of data involving dates.

3.16. Manipulating Dates Involving Time Zones

Problem

You had a conference call scheduled for December 21, 2012, at 9:30 a.m. in Chicago. At what local time did your friend in Bangalore, India, have to show up to attend?

Solution

For almost any problem involving time zones, you should use the `pytz` module. This package provides the Olson time zone database, which is the de facto standard for time zone information found in many languages and operating systems.

A major use of `pytz` is in localizing simple dates created with the `datetime` library. For example, here is how you would represent a date in Chicago time:

```
>>> from datetime import datetime
>>> from pytz import timezone
```

```
>>> d = datetime(2012, 12, 21, 9, 30, 0)
>>> print(d)
2012-12-21 09:30:00
>>>

>>> # Localize the date for Chicago
>>> central = timezone('US/Central')
>>> loc_d = central.localize(d)
>>> print(loc_d)
2012-12-21 09:30:00-06:00
>>>
```

Once the date has been localized, it can be converted to other time zones. To find the same time in Bangalore, you would do this:

```
>>> # Convert to Bangalore time
>>> bang_d = loc_d.astimezone(timezone('Asia/Kolkata'))
>>> print(bang_d)
2012-12-21 21:00:00+05:30
>>>
```

If you are going to perform arithmetic with localized dates, you need to be particularly aware of daylight saving transitions and other details. For example, in 2013, U.S. standard daylight saving time started on March 13, 2:00 a.m. local time (at which point, time skipped ahead one hour). If you're performing naive arithmetic, you'll get it wrong. For example:

```
>>> d = datetime(2013, 3, 10, 1, 45)
>>> loc_d = central.localize(d)
>>> print(loc_d)
2013-03-10 01:45:00-06:00
>>> later = loc_d + timedelta(minutes=30)
>>> print(later)
2013-03-10 02:15:00-06:00       # WRONG! WRONG!
>>>
```

The answer is wrong because it doesn't account for the one-hour skip in the local time. To fix this, use the `normalize()` method of the time zone. For example:

```
>>> from datetime import timedelta
>>> later = central.normalize(loc_d + timedelta(minutes=30))
>>> print(later)
2013-03-10 03:15:00-05:00
>>>
```

Discussion

To keep your head from completely exploding, a common strategy for localized date handling is to convert all dates to UTC time and to use that for all internal storage and manipulation. For example:

```
>>> print(loc_d)
2013-03-10 01:45:00-06:00
>>> utc_d = loc_d.astimezone(pytz.utc)
>>> print(utc_d)
2013-03-10 07:45:00+00:00
>>>
```

Once in UTC, you don't have to worry about issues related to daylight saving time and other matters. Thus, you can simply perform normal date arithmetic as before. Should you want to output the date in localized time, just convert it to the appropriate time zone afterward. For example:

```
>>> later_utc = utc_d + timedelta(minutes=30)
>>> print(later_utc.astimezone(central))
2013-03-10 03:15:00-05:00
>>>
```

One issue in working with time zones is simply figuring out what time zone names to use. For example, in this recipe, how was it known that "Asia/Kolkata" was the correct time zone name for India? To find out, you can consult the `pytz.country_timezones` dictionary using the ISO 3166 country code as a key. For example:

```
>>> pytz.country_timezones['IN']
['Asia/Kolkata']
>>>
```

By the time you read this, it's possible that the `pytz` module will be deprecated in favor of improved time zone support, as described in PEP 431. Many of the same issues will still apply, however (e.g., advice using UTC dates, etc.).

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.

4.1. 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.

4.2. 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__()`.

4.3. 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.

4.4. 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 Recipe 4.2, 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.

4.5. 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.

4.6. 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'
>>>
```

4.7. 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.

4.8. 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.

4.9. 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.

4.10. 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):
    ...
```

4.11. 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 `y` is 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)]
>>>
```

4.12. 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.

4.13. 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)

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:

```
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:

```
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 Recipe 4.14.

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.

4.14. 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 Recipe 12.12 for another example.

4.15. 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.

4.16. 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()`.