The technique we have used thus far is called *user-based collaborative filtering*. An alternative is known as *item-based collaborative filtering*. In cases with very large datasets, item-based collaborative filtering can give better results, and it allows many of the calculations to be performed in advance so that a user needing recommendations can get them more quickly.

The procedure for item-based filtering draws a lot on what we have already discussed. The general technique is to precompute the most similar items for each item. Then, when you wish to make recommendations to a user, you look at his top-rated items and create a weighted list of the items most similar to those. The important difference here is that, although the first step requires you to examine all the data, *comparisons between items will not change as often as comparisons between users*. This means you do not have to continuously calculate each item's most similar items—you can do it at low-traffic times or on a computer separate from your main application.

Building the Item Comparison Dataset

To compare items, the first thing you'll need to do is write a function to build the complete dataset of similar items. Again, this does not have to be done every time a recommendation is needed—instead, you build the dataset once and reuse it each time you need it.

To generate the dataset, add the following function to *recommendations.py*:

```
def calculateSimilarItems(prefs,n=10):
  # Create a dictionary of items showing which other items they
  # are most similar to.
  result={}

  # Invert the preference matrix to be item-centric
  itemPrefs=transformPrefs(prefs)
  c=0
  for item in itemPrefs:
    # Status updates for large datasets
    c+=1
    if c%100==0: print "%d / %d" % (c,len(itemPrefs))
    # Find the most similar items to this one
    scores=topMatches(itemPrefs,item,n=n,similarity=sim_distance)
    result[item]=scores
  return result
```

This function first inverts the score dictionary using the `transformPrefs` function defined earlier, giving a list of items along with how they were rated by each user. It then loops over every item and passes the transformed dictionary to the `topMatches` function to get the most similar items along with their similarity scores. Finally, it creates and returns a dictionary of items along with a list of their most similar items.

In your Python session, build the item similarity dataset and see what it looks like:

```
>>> reload(recommendations)
>>> itemsim=recommendations.calculateSimilarItems(recommendations.critics)
>>> itemsim
{'Lady in the Water': [(0.40000000000000002, 'You, Me and Dupree'),
                       (0.2857142857142857, 'The Night Listener'),...
 'Snakes on a Plane': [(0.22222222222222221, 'Lady in the Water'),
                       (0.18181818181818182, 'The Night Listener'),...
etc.
```

Remember, this function only has to be run frequently enough to keep the item similarities up to date. You will need to do this more often early on when the user base and number of ratings is small, but as the user base grows, the similarity scores between items will usually become more stable.

Getting Recommendations

Now you're ready to give recommendations using the item similarity dictionary without going through the whole dataset. You're going to get all the items that the user has ranked, find the similar items, and weight them according to how similar they are. The items dictionary can easily be used to get the similarities.

Table 2-3 shows the process of finding recommendations using the item-based approach. Unlike Table 2-2, the critics are not involved at all, and instead there is a grid of movies I've rated versus movies I haven't rated.

Table 2-3. Item-based recommendations for Toby

Movie	Rating	Night	R.xNight	Lady	R.xLady	Luck	R.xLuck
Snakes	4.5	0.182	0.818	0.222	0.999	0.105	0.474
Superman	4.0	0.103	0.412	0.091	0.363	0.065	0.258
Dupree	1.0	0.148	0.148	0.4	0.4	0.182	0.182
Total		0.433	1.378	0.713	1.764	0.352	0.914
Normalized			3.183		2.598		2.473

Each row has a movie that I have already seen, along with my personal rating for it. For every movie that I haven't seen, there's a column that shows how similar it is to the movies I have seen—for example, the similarity score between *Superman* and *The Night Listener* is 0.103. The columns starting with R.x show my rating of the movie multiplied by the similarity—since I rated *Superman* 4.0, the value next to Night in the Superman row is 4.0 × 0.103 = 0.412.

The total row shows the total of the similarity scores and the total of the R.x columns for each movie. To predict what my rating would be for each movie, just divide the total for the R.x column by the total for the similarity column. My predicted rating for *The Night Listener* is thus 1.378/0.433 = 3.183.

You can use this functionality by adding one last function to *recommendations.py*:

```
def getRecommendedItems(prefs,itemMatch,user):
  userRatings=prefs[user]
  scores={}
  totalSim={}

  # Loop over items rated by this user
  for (item,rating) in userRatings.items( ):

    # Loop over items similar to this one
    for (similarity,item2) in itemMatch[item]:

      # Ignore if this user has already rated this item
      if item2 in userRatings: continue

      # Weighted sum of rating times similarity
      scores.setdefault(item2,0)
      scores[item2]+=similarity*rating

      # Sum of all the similarities
      totalSim.setdefault(item2,0)
      totalSim[item2]+=similarity

  # Divide each total score by total weighting to get an average
  rankings=[(score/totalSim[item],item) for item,score in scores.items( )]

  # Return the rankings from highest to lowest
  rankings.sort( )
  rankings.reverse( )
  return rankings
```

You can try this function with the similarity dataset you built earlier to get the new recommendations for Toby:

```
>> reload(recommendations)
>> recommendations.getRecommendedItems(recommendations.critics,itemsim,'Toby')
[(3.182, 'The Night Listener'),
 (2.598, 'Just My Luck'),
 (2.473, 'Lady in the Water')]
```

The Night Listener still comes in first by a significant margin, and *Just My Luck* and *Lady in the Water* have changed places although they are still close together. More importantly, the call to getRecommendedItems did not have to calculate the similarities scores for all the other critics because the item similarity dataset was built in advance.

Using the MovieLens Dataset

For the final example, let's look at a real dataset of movie ratings called *MovieLens*. MovieLens was developed by the GroupLens project at the University of Minnesota. You can download the dataset from *http://www.grouplens.org/node/12*. There are two datasets here. Download the 100,000 dataset in either *tar.gz* format or *zip* format, depending on your platform.