Chapter 2
Data Mapping and Data Dictionaries

Abstract This chapter delves into the key mathematical and computational components of data analytic algorithms. The purpose of these algorithms is to reduce massively large data sets to much smaller data sets with a minimal loss of relevant information. From the mathematical perspective, a data reduction algorithm is a sequence of *data mappings*, that is, functions that consume data in the form of sets and output data in a reduced form. The mathematical perspective is important because it imposes certain desirable attributes on the mappings. However, most of our attention is paid to the practical aspects of turning the mappings into code. The mathematical and computational aspects of data mappings are applied through the use of *data dictionaries*. The tutorials of this chapter help the reader develop familiarity with data mappings and `Python` dictionaries.

2.1 Data Reduction

One of principal reasons that data science has risen in prominence is the accelerating growth of massively large data sets in government and commerce. The potential exists for getting new information and knowledge from the data. Extracting the information is not easy though. The problem is due to the origins of the data. Most of the time, the data are not collected according to a design with the questions of interest in mind. Instead, the data are collected without planning and for purposes tangential to those of the analyst. For example, the variables recorded in retail transaction logs are collected for logistical and accounting purposes. There's information in the data about consumer habits and behaviors, but understanding consumer behavior is not the purpose for collecting the data. The information content is meager with respect to the objective. The analyst will have to work hard

B. Steele et al., *Algorithms for Data Science*,
DOI 10.1007/978-3-319-45797-0_2

to get the information and the strategy must be developed intelligently and executed carefully. Tackling the problem through the use of data mappings will help develop the strategy.

The second major hurdle in extracting information from massively large data sets materializes after the data analyst has developed a strategy. Translating the strategy to action will likely require a substantial programming effort. To limit the effort, we need a language with objects and functions compatible with data mapping. `Python` is the right language, and the `Python` dictionary is the right structure for building data reduction algorithms.

The next section discusses a fairly typical data source and database to provide some context for the discussion of data reduction and data mappings.

2.2 Political Contributions

In April of 2014, the Supreme Court struck down a 40 year-old limit on campaign contributions made by individuals within a single 2-year election cycle.[1] Many people believe that the ruling allows the very wealthy to have undue influence on election outcomes. Anyone that attempts to analyze the relationship between contributions and candidates must recognize that popular candidates attract contributions. A simple analysis of total contributions and election winners cannot produce evidence of causation. However, there is a rich, publicly available data source maintained by the Federal Election Commission that may be mined to learn about the contributors and recipients of money spent in the electoral process.

Let's suppose that you are running an electoral campaign and you have a limited budget with which to raise more money. The money should be spent when potential contributors are most likely to contribute to your campaign. But when? The Federal Election Commission data sets can be used to determine when people contribute. That's not exactly the answer to the question, but it is close. To answer that question, we computed the dollar value of contributions reported to the Federal Election Commission for each day of a 2-year period roughly corresponding to the 2012–2014 election cycle. Figure 2.1 shows that daily total contributions were fairly static for most of the cycle except for temporary increases soon before the end of the year. Closer inspection revealed that contributions increase briefly before December 31, March 31, June 30, and September 30, dates that mark the end of the fiscal quarters. A much greater and lasting increase began in September of 2014, often recognized as the beginning of the political campaign season. There are also substantial differences between weekday and weekend contributions perhaps because individuals and not corporations make donations on weekends.

[1] Election cycles correspond to the 2-year terms of the Representatives to the U.S. Congress.

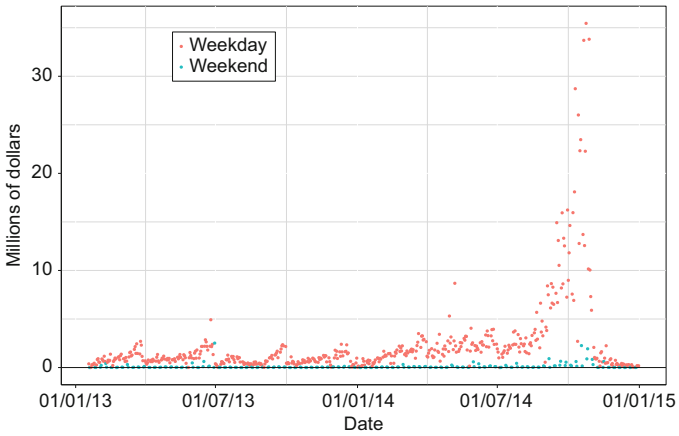

Fig. 2.1 Donation totals reported to the Federal Election Commission by Congressional candidates and Political Action Committees plotted against reporting date

The Federal Election Campaign Act requires candidate committees and political action committees (PACs) to report contributions in excess of $200 that have been received from individuals and committees. Millions of large individual contributions (that is, larger than $200) are reported in a 2-year election cycle. The 2014–2016 data set contained more than 12 million records as of July 1, 2016. Data from 2003 to the most recent election cycle are publicly available from the Federal Election Commission webpage http://www.fec.gov/disclosure.shtml. Three data file types of particular interest are: (a) contributions by individuals; (b) committee master files containing information about Political Action Committees, campaign committees, and other committees raising money to be spent on elections; and (c) candidate master files containing information about the candidates. The contributions by individuals files contain records of large contributions from individuals. The files list the contributor's name and residence, occupation, and employer. They also list the transaction amount and a code identifying the recipient committee. Some data entries are missing or non-informative. A contribution record looks like this:

```
C00110478|N|M3||15|IND|HARRIS, ZACHARY|BUTTE|MT|59701|PEABODY COAL|225
```

We see that Zachary Harris, an employee of Peabody Coal, made a contribution of $225 to the committee identified by `C00110478`.

Another application of the FEC data is motivated by the widely-held belief that big contributions buy political influence [21]. If this is true, then we ought to find out who contributed the largest sums of money. The principal effort in identifying the biggest contributors is the reduction of two million or more contribution records to a short list of contributors and sums. In what follows, reduction is accomplished by creating a `Python` dictionary containing the names and contribution sums of everyone that appears in the contributions by individuals data file.

2.3 Dictionaries

A `Python` dictionary is much like a conventional dictionary. A conventional dictionary is set of pairs. Each pair consists of a word (the `Python` analog is a *key*) and a definition (the `Python` analog is a *value*). Both types of dictionaries are organized around the key in the sense that the key is used to find the value. Keys are unique. That is, a key will only appear once in the dictionary. An example of three key-value pairs from a dictionary of contributors and contributions sums from the 2012–2014 election cycle is

```
'CHARLES G. KOCH 1997 TRUST' : 5000000,
'STEYER, THOMAS' : 5057267,
'ADELSON, SHELDON' : 5141782.
```

The keys are contributor names and the values are the totals made by the contributor during the election cycle. If there was more than one contribution made with the same name, then we store the sum of the contributions with the name. `Python` places some constraints on the types of objects to be used as keys but the values are nearly unconstrained; for example, values may be integers as above, strings, sets, or even dictionaries. In the example above, the number of entries in the contributor dictionary will not be dramatically less than the number of records in the data file. However, if the objective were to identify geographic entities from which the most money originated, then United States zip codes could be used as keys and the number of dictionary entries would be about 43,000.

2.4 Tutorial: Big Contributors

The objective of this tutorial is to construct a `Python` dictionary in which individual contributors are the keys and the values are the sum of all contributions made by the contributor in an election cycle. Some care must be taken in how the dictionary is constructed because big contributors make multiple contributions within an election cycle. Therefore, the value that is stored with the key must be a total, and we have to increment the total whenever second and subsequent records from a particular contributor are encountered in the data file.

After constructing the dictionary, the entries are to be sorted by the contribution totals (the dictionary values). The result will be a list of the names and contribution totals ordered from largest to smallest total contribution.

Proceed as follows:

1. Navigate to http://www.fec.gov/finance/disclosure/ftpdet.shtml, the Federal Election Commission website. Select an election cycle by clicking on one of the election cycle links. The individual contributions file appear as

`indivxx.zip` where `xx` is the last two digits of the last year of the election cycle, e.g., `indiv14.zip` contains data from the 2012–2014 election cycle. Download a file by clicking on the name of the zip file.

2. Before leaving the website, examine the file structure described under `Format Description`. In particular, note the column positions of the name of the contributor (8 in the 2012–2014 file) and the transaction amount (15 in the 2012–2014 file).

   Beware: the FEC labels the first column in their data sets as one, but the first element in a list in `Python` is indexed by zero. When the `Python` script is written, you will have to subtract one from the column position of a variable to obtain the `Python` list position of the variable.
3. Unzip the file and look at the contents. The file name will be `itcont.txt`. Opening large files in an editor may take a long time, and so it's useful to have a way to quickly look at the first records of the file. You may do the following:

   a. If your operating system is Linux, then open a terminal, navigate to the folder containing file and write the first few records of the file to the terminal using the instruction

```
cat itcont.txt | more
```

   Pressing the enter key will print the next row. Pressing the `Ctrl+C` key combination will terminate the `cat` (concatenate) command. You'll see that attributes are separated by the pipe character: `|`.

   b. If your operating system is Windows 7, then open a command prompt, navigate to the folder containing the file, and write first 20 records of the file to the window using the instruction

```
head 20 itcont.txt
```

   c. If your operating system is Windows 10, then open a `PowerShell`, navigate to the folder containing the file, and write first 20 records of the file to the window using the instruction

```
gc itcont.txt | select -first 20
```

4. Create a `Python` script—a text file with the `py` extension. Instruct the `Python` interpreter to import the `sys` and `operator` modules by entering the following instructions at the top of the file. A module is a collection of functions that extend the core of the `Python` language. The `Python` language has a relatively small number of commands—this is a virtue since it makes it relatively easy to master a substantial portion of the language.

```
import sys
import operator
```

5. Import a function for creating `defaultdict` dictionaries from the module `collections` and initialize a dictionary to store the individual contributor totals:

```
from collections import defaultdict
indivDict = defaultdict(int)
```

   The `int` argument passed to the `defaultdict` function specifies that the dictionary values in `indivDict` will be integers. Specifically, the values will be contribution totals.
6. Specify the path to the data file. For example,

```
path = 'home/Data/itcont.txt'
```

   If you need to know the full path name of a directory, submit the Linux command `pwd` in a terminal. In Windows, right-click on the file name in Windows Explorer and select the Properties option.
7. Create a file object using the `open` function so that the data file may be processed. Process the file one record at a time by reading each record as a single string of characters named `string`. Then, split each string into substrings wherever the pipe symbol appears in the string. The code for opening and processing the file is

```
with open(path) as f:     # The file object is named f.
    for string in f:      # Process each record in the file.
        data = string.split("|")  # Split the character string
                                  # and save as a list named data.
        print(data)
        sys.exit()
```

   The statement `data = string.split("|")` splits the character string at the pipe symbol. The result is a 21-element list named `data`. The instruction `print(len(data))` will print the length of the list named `data`. You can run the print statement from the console or put it in the program.

   The `sys.exit()` instruction will terminate the program. The data file will close when the program execution completes or when execution is terminated. Execute the script and examine the output for errors. You should see a list of 21 elements.

The Python language uses indentation for program flow control. For example, the `for string in f:` instruction is nested below the `with open(path) as f:` statement. Therefore, the `with open(path) as f:` statement executes as long as the file object `f` is open. Likewise, every statement that is indented below the `for string in f:` statement will execute before the flow control returns to the `for` statement. Consequently, `for string in f` reads strings until there are no more strings in `f` to be read. The object `f` will close automatically when the end of the file is reached. At that point, the program flow breaks out of the `with open(path) as f` loop.

The convention in this book is to show indentation in a single code segment, but not carry the indentation down to the following code segments. Therefore, it is up to the reader to understand the program flow and properly indent the code.

Let's return to the script.

8. Remove the termination instruction (`sys.exit()`) and the print instruction. Initialize a record counter using the instruction `n = 0`. Below the `import` and path declaration instructions, the program should appear as so:

```
n = 0
with open(path) as f:     # The file object is named f.
    for string in f:      # Process each record in the file.
        data = string.split("|")
```

9. In the Python lexicon, an *item* is a key-value pair. Each item in the dictionary `indivDict` consists of the name of a contributor (the key) and the total amount that the contributor has contributed to all election committees (the value).

   Items will be added to the dictionary by first testing whether the contributor name is a dictionary key. If the contributor's name is not a key, then the name and contribution amount are added as a key-value pair to the dictionary. On the other hand, if the candidate's name is already a key, then the contribution amount is added to the existing total. These operations are carried out using a *factory* function, a function contained in the module `collections`. This operation is done automatically—we only need one line of code:

```
indivDict[data[7]] += int(data[14])
```

   Add this instruction so that it operates on every list. Therefore, the indentation must be the same as for the instruction `data = string.split("|")`. Set it up like this:

```
with open(path) as f:
    for string in f:
        data = string.split("|")
        indivDict[data[7]] += int(data[14])
```

10. To trace the execution of the script, add the instructions

```
n += 1
if n % 5000 == 0:
    print(n)
```

    within the `for` loop. The instruction `n % 5000` computes the modulus, or integer remainder, of $5000/n$. If n is printed more often than every 5000 lines, then the execution of the program will noticeably slow.
11. After processing the data set, determine the number of contributors by adding the line

```
print(len(indivDict))
```

    This instruction should not be indented.
12. Place the instruction `import operator` at the top of the script. Importing the module allows us to use the functions contained in the module.
13. Sort the dictionary using the instruction

```
sortedSums = sorted(indivDict.items(), key=operator.itemgetter(1))
```

    The statement `indivDict.items()` creates a list (not a dictionary) consisting of the key-value pairs composing `indivDict`. The `key` argument of the function `sorted()` points to the position within the pairs that are to be used for sorting. Setting `key = operator.itemgetter(1)` specifies that the elements in position 1 of the tuples are to be used for determining the ordering. Since zero-indexing is used, `itemgetter(1)` instructs the interpreter to use the values for sorting. Setting `key = operator.itemgetter(0)` will instruct the interpreter to use the keys for sorting.
14. Print the names of the contributors that contributed at least $\$25,000$:

```
for item in sortedSums :
    if item[1] >= 25000 : print(item[1], item[0])
```

15. The list of largest contributors likely will show some individuals.

Transforming the individual contributors data set to the dictionary of contributors did not dramatically reduce the data volume. If we are to draw inferences about groups and behaviors, more reduction is needed. Before proceeding with further analysis, we will develop the principles of data reduction and data mapping in detail.

2.5 Data Reduction

Data reduction algorithms reduce data through a sequence of mappings. Thinking about a data reduction algorithm as a mapping or sequence of mappings is a key step towards insuring that the final algorithm is transparent and computationally efficient. Looking ahead to the next tutorial, consider an algorithm that is to consume an individual contribution data set A. The objective is to produce a set of pairs E in which each pair identifies a major employer, say Microsoft, and the amounts contributed by company employees to Republican, Democratic, and other party candidates. The output of the algorithm is a list of pairs with the same general form as r:

$$r = \big(\text{Microsoft} : \big[(D, 20030), (R, 4150), (\text{other}, 0)\big]\big). \tag{2.1}$$

We say that the algorithm maps A to E. It may not be immediately obvious how to carry out the mapping, but if we break down the mapping as a sequence of simple mappings, then the algorithm will become apparent. One sequence (of several possible sequences) begins by mapping individual contribution records to a dictionary in which each key is an employer and the value is a list of pairs. The pairs consist of the recipient committee code and the amount of the contribution. For instance, a dictionary entry may appear so:

$$\big(\text{Microsoft} : \big[(C20102, 200), (C84088, 1000), \dots\big]\big),$$

where $C20102$ and $C84088$ are recipient committee identifiers. Be aware that the lengths of the lists associated with different employers will vary widely and the lists associated with some of the larger employers may number in the hundreds. Building the dictionary using `Python` is straightforward despite the complexity of the dictionary values.

We're not done of course. We need the political parties instead of the recipient committee identifiers. The second mapping consumes the just-built dictionary and replaces the recipient committee code with a political party affiliation if an affiliation can be identified. Otherwise, the pair is deleted. The final mapping maps each of the long lists to a shorter list. The shorter list consists of three pairs. The three-pair list is a little complicated: the first element of a pair identifies the political party (e.g., Republican) and the second

element is the sum of all employee contributions received by committees with the corresponding party affiliation. Line 2.1 shows a hypothetical entry in the final reduced data dictionary.

It may be argued that using three maps to accomplish data reduction is computationally expensive, and that computational efficiency would be improved by using fewer maps. The argument is often misguided since the effort needed to program and test an algorithm consisting of fewer but more sophisticated and complicated mappings shifts the workload from the computer to the programmer.

2.5.1 Notation and Terminology

A *data mapping* is a function $f : A \mapsto B$ where A and B are data sets. The set B is called the *image* of A under f and A is the called *preimage* of B. Suppose that $\mathbf{y}$ is a vector of length p (Chap. 1, Sect. 1.10.1 provides a review of matrices and vectors). For instance, $\mathbf{y}$ may be constructed from a record or row in a data file. To show the mapping of an element $\mathbf{y} \in A$ to an element $\mathbf{z} \in B$ by f, we write

$$f : \mathbf{y} \mapsto \mathbf{z} \text{ or } f(\mathbf{y}) = \mathbf{z}.$$

We also write $B = f(A)$ to indicate that B is the image of A. The statement $B = f(A)$ implies that for every $\mathbf{b} \in B$ there exists $\mathbf{a} \in A$ such that $\mathbf{a} \mapsto \mathbf{b}$.

For the example above, it's possible to construct a single map f from A to E though it's less complicated to define two mappings that when applied in sequence, map A to E. If the first mapping is g and the second is h, then data reduction is carried out according to $g : A \mapsto B$ and then $h : B \mapsto E$, or more simply by realizing that f is the composite of g and h, i.e., $f = h \circ g$.

As a point of clarification, a *list* of length p in the `Python` lexicon is equivalent to a tuple of length p in mathematics. The elements of a list may be changed by replacement or a mathematical operation. If we may change the values of an object, then the object is said to be *mutable*. For instance, a `Python` list is *mutable*. The term *tuple* in the `Python` lexicon refers to an immutable list. The elements of a tuple cannot be changed. You may think of a tuple as a universal constant while the program executes.

The order or position of elements within a tuple is important when working with tuples and `Python` lists; hence, $\mathbf{x} \neq \mathbf{y}$ if $\mathbf{x} = (1, 2)$ and $\mathbf{y} = (2, 1)$. In contrast, the elements of sets may be rearranged without altering the set, as illustrated by $\{1, 2\} = \{2, 1\}$. The `Python` notation for a list uses brackets, so that mathematical tuples are expressed as `Python` lists by writing `x = [1,2]` and `y = [2,1]`. The beginning and end of a `Python` tuple (an immutable list) is marked by parentheses. For example, `z = (1,2)` is a `Python` two-tuple. You may verify that `x` $\neq$ `z` (submit `[1,2]==(1,2)` in a `Python` console). You can create a tuple named `a` from a list using the `tuple` function; e.g.,

`a = tuple([1,2])`; likewise, a list may be formed from a tuple using the statement `a = list[(1,2)]`. However, if you submit `[(1,2)]`, the result is not a list consisting of the elements 1 and 2. Instead the result is a list containing the tuple $(1, 2)$.

2.5.2 The Political Contributions Example

To make these ideas concrete we'll continue with the political contributions example and define three data mappings that reduce an input data file on the order of a million records to an output file with about 1000 records. The first mapping $g : A \mapsto B$ is defined to take a contribution record $\mathbf{y} \in A$ and map it to a three-tuple $\mathbf{b} = (b_1, b_2, b_3) \in B$ where b_1 is the employer of the contributor, b_2 is the political party of the recipient of the contribution, and b_3 is the contribution amount.

The set B will be only slightly smaller than A since the mapping does little more than discard entries in A with missing values for the attributes of interest. The second mapping h takes the set B as an input and computes the total contribution for each employer by political party. Supposing that the political parties are identified as Republican, Democratic, and Other, then the image of B under $h : B \mapsto C$ may appear as $C = \{\mathbf{c}_1, \mathbf{c}_2, \ldots, \mathbf{c}_n\}$ where, for example,

$$\begin{aligned}
\mathbf{c}_1 &= (\text{Microsoft}, \text{Republican}, 1000) \\
\mathbf{c}_2 &= (\text{Microsoft}, \text{Democratic}, 70000) \\
\mathbf{c}_3 &= (\text{Microsoft}, \text{Other}, 350) \\
&\vdots \\
\mathbf{c}_n &= (\text{Google}, \text{Other}, 5010).
\end{aligned}$$

The data set C is convenient for producing summary tables or figures. However, C may be further reduced by a third mapping k that combines the records for a particular employer as a single pair with a somewhat elaborate structure. For example, the key-value pair $\mathbf{d}_1 \in D = k(C)$, may appear as

$$\mathbf{d}_1 = (d_{11}, \mathbf{d}_{12}) = \Big(\text{Microsoft}, \big((D, 20030), (R, 4150), (\text{other}, 0)\big)\Big) \tag{2.2}$$

where d_{11} =Microsoft. The second element of the pair $\mathbf{d}_1$ is a three-tuple $\mathbf{d}_{12} = (\mathbf{d}_{121}, \mathbf{d}_{122}, \mathbf{d}_{123})$ where the elements of the three-tuple are pairs. Hence, the first element of the three-tuple, $\mathbf{d}_{121} = (D, 20030)$, is a pair in which the first element identifies the Democratic party and the second element is the total contribution made by employees to candidates affiliated with the Democratic party.

The distribution of contributions to Democratic and Republican candidates made by employees of 20 companies are shown in Fig. 2.2 as an illustration of the results of the data reduction process described above. There is

a substantial degree of variation between companies with respect to the distribution; for instance, the split between Democratic and Republican parties is nearly equal for Morgan Stanley, but very lopsided for Google (which appears twice) and Harvard. Those companies that reveal the least amount of diversity in party contributions contribute predominantly to Democratic party candidates.

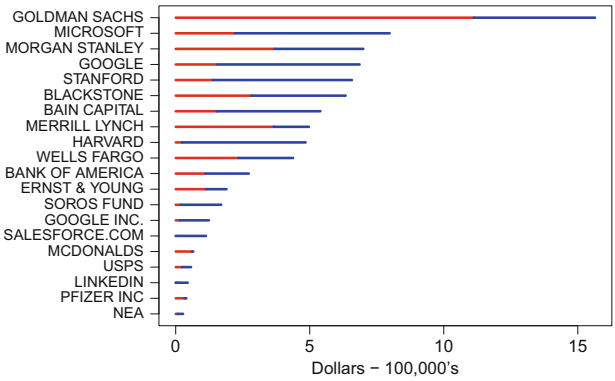

Fig. 2.2 Contributions to committees by individual contributors aggregated by employer. Length of the *red lines* represents the total amount to Republican party candidates and the length of the *blue lines* represents total contributions to Democratic party candidates

2.5.3 Mappings

Let us take a mathematical view of data mappings to help identify some properties, that if lacking, imply that an algorithm is at risk of not achieving its intended purpose.

The first of these mathematical properties is that a mapping must be *well-defined*. A mapping $f : D \to E$ is well-defined if, for every $\mathbf{x} \in D$ there is one and only one output $\mathbf{y} = f(\mathbf{x})$.

If an algorithm is treated as a well-defined mapping, then there can only be one and only one output of the algorithm for every possible input. A second essential property of a data mapping is that every output $\mathbf{y}$ must belong to the image set E. This implies that all possible outputs of the function must be anticipated and that the algorithm does not produce an unexpected or unusable output, say, an empty set instead of the expected four-element tuple. This condition avoids the possibility of an subsequent error later in the program. Computer programs that are intended for general use typically contain a significant amount of code dedicated to checking and eliminating unexpected algorithm output.

We define a *dictionary mapping* to be a mapping that produces a key-value pair. The key is a label or index that identifies the key-value pair. The key serves as a focal point about which the mapping and algorithm is constructed. In the example above, a natural choice for keys are the employer names because the objective is to summarize contributions to Democratic and Republican political parties by employers. A collection of key-value pairs will be called a *dictionary* in accordance with `Python` terminology.[2] A dictionary item is a key-value pair, say $\mathbf{e}_i = (\text{key}_i, \text{value}_i)$, and if the mapping is $f : D \mapsto E$, then we will refer to $f(D) = \{\mathbf{e}_1, \ldots, \mathbf{e}_n\}$ as the dictionary produced by the application of f to the data set D. Keep in mind that a `Python` dictionary is not a set. It has it's own type–`dict`. However, since the items in a dictionary are unique, we tend to think of dictionaries as sets. You may determine the type of an object `u` with the instruction `type(u)`.

2.6 Tutorial: Election Cycle Contributions

The aim of this tutorial is translate the logical and mathematical descriptions of the data reduction mappings above into `Python` code for the problem of summarizing the political orientation of company employees. It has been conjectured that the existence a corporate culture within a company fosters insularity, uniformity of thought, and suppression of originality, all of which are presumed to have negative effects on creativity and innovation. But, it's difficult to uncover objective evidence that supports this conjecture. We may, however, investigate the political complexion of a corporation, more specifically, the political preferences of the employees of a company by studying the recipients of their contributions. If the results of the investigation reveal that one political party is the dominant recipient of contributions, then we have some evidence of a tendency for employees to align politically.

The first task of this investigation is to build a dictionary that identifies a political party associated with an individual's contribution. The keys, however, are not individuals but instead are the employers of the individuals. To accomplish the task, a Federal Elections Commission Individual Contributions data file will be mapped to a dictionary in which a key is an employer and the associated value is a list of n pairs, where each pair consists of a recipient political party and contribution amount. The second task is to build another dictionary to store contribution totals by employer and political party. This task is accomplished by a mapping the n pairs associated with an employer to three pairs (corresponding to Republican, Democratic, and other parties)

As with Tutorial 2.4, you'll process an Individual Contributions data file for some election cycle. From each record, extract the employer, the

[2] `Python` dictionaries are equivalent to `Java` hashmaps.

contribution amount and the recipient code. If there is an employer listed, then determine whether there is a political party associated with the recipient code.

We'll need another dictionary that links political party to recipient. If there is a political party associated with the recipient, it will be recorded in one of two FEC files since there are two types of recipients: candidate committees and other committees. If the recipient is a candidate committee, then the committee information is in the *Candidate Master* file and there's a good chance that a political party will be identified in the file. The other recipients, or other committees, encompass political action committees, party committees, campaign committees, or other organizations spending money related to an election. Information on other committees is contained in the *Committee Master* file and a political party sometimes is identified with the committee in this file. Therefore, once the recipient code is extracted from the Individual Contributions data file, you'll have to check for the recipient code in the Candidate Master file; if it's not there, then check the Committee Master file.

The program is somewhat complicated because two dictionaries must be searched to determine the party affiliation of a receiving candidate. We wind up processing three files and building three dictionaries. Table 2.1 is shown for reference.

Table 2.1 Files and dictionaries used in Tutorial 2.6. The file `cn.txt` is a Candidate Master file and `cm.txt` is a Committee Master file. The file `itcont.txt` is a Contributions by Individual file. Field positions are identified using zero-indexing

		Attributes and field positions			
File	Dictionary	Key	Column	Value	Column
`cn.txt`	`canDict`	Committee code	9	Political party[a]	2
`cm.txt`	`comDict`	Committee code	0	Political party	10
`itcont.txt`	`employerDict`	Employer	11	Amount	14

[a]The recipient's political party is determined by searching the `canDict` and `comDict` dictionaries for the committee code associated with the individual's contribution

1. Decide on an election cycle and, for that cycle, download and unzip the (a) Candidate Master File; (b) Committee Master File; and (c) Contributions by Individuals Files from the Federal Elections Commission webpage http://www.fec.gov/finance/disclosure/ftpdet.shtml.
2. Build a candidate committee dictionary from the Candidate Master file using the principal campaign committee codes in field position 9 (the zero-indexed column) as keys and party affiliation in position 2 as values. We will refer to this dictionary by the name `canDict`.

```
canDict = {}
path = '../cn.txt'
with open(path) as f:
    for line in f:
        data = line.split("|")
        canDict[data[9]] = data[2]
```

Candidate identifiers appear only once in the Candidate Master file so it's not necessary to test whether `data[9]` is a dictionary key before creating the key-value pair.

3. Build an *other* committees dictionary from the Committee Master file. We'll use the `Python` indexing convention of indexing the first element of a list with 0. The keys are to be the committee identification codes located in field position 0 of Committee Master file and the values are the committee parties located in field position 10. We will refer to this dictionary by the name `otherDict`.

```
otherDict = {}
path = '../cm.txt'
with open(path) as f:
    for line in f:
        data = line.split("|")
        otherDict[data[0]] = data[10]
```

4. Build the dictionary `employerDict`. Keys will be employers and values will be a list of pairs. Each pair in a list will be a political party and a contribution amount. To build the dictionary, process the Contributions by Individuals (`itcont.txt`) data file one record at a time.[3]

   The first task to perform with each record is to determine what, if any political party affiliation is associated with the recipient of the contribution. The search begins with the Filer Identification number—`data[0]`. The filer is the recipient committee and the entity that filed the report to the Federal Elections Commission (the individual does not file the report). Determine if there is a party listed for the recipient committee. First, check the candidate committee dictionary in case the recipient committee is a candidate committee. Look for a party name entry in the dictionary named `canDict`. If the filer identification number is not a key in this dictionary, then look for an entry in the other committee dictionary named `otherDict`. Then create a two-tuple, `x`, with the party and the amount. Use the `int` function to convert the contribution amount (stored as a string) to an integer. The code follows:

[3] You may be able to reuse your code from the tutorial of Sect. 2.4.

```
path = '../itcont14.txt'
n = 0
employerDict = {}
with open(path) as f:
    for line in f:
        data = line.split("|")
        party = canDict.get(data[0])
        if data[0] is None:
            party = otherDict[data[0]]
        x = (party, int(data[14]))
```

5. The next step is to save `x` in the dictionary named `employerDict`. Each key is to be the employer of a contributor and the value will be a list of contribution pairs (the `x`'s) built in step 4. If there is an entry in `data` for employer, it will be in position 11. So, extract `data[11]` and if it is not empty string, then assign the string to `employer`. If `employer` is an empty string, then ignore the record and process the next record. Assuming that there is an entry for `employer`, test whether it is a dictionary key. If `employer` is not a dictionary key, then create the dictionary entry by assigning a list containing `x` using the instruction `employerDict[employer] = [x]`. On the other hand, if `employer` is a key, then append `x` to the dictionary value associated with `employer` using the instruction `employerDict[employer].append(x)`:

```
employer = data[11]
if employer != '':
    value = employerDict.get(employer)
    if value is None:
        employerDict[employer] = [x]
    else:
        employerDict[employer].append(x)
```

Don't forget to indent the code segment. It must be aligned with the `x = (party, int(data[14]))` statement because it is to execute every time that a new record is processed. The pair `x` is appended in the last statement of the code segment.

By construction, the value associated with the key will be a list. For instance, a value may appear as so:

$$\texttt{value} = \texttt{[('DEM', 1000), (", 500),('REP', 500)]}. \tag{2.3}$$

Note that one of the entries in `value` is an empty string which implies that the contribution was received by a committee without a political party listed in either the Candidate Master or Committee Master file.

6. It's helpful to trace the progress as the script executes. Count the number of records as the file is processed and print the count at regular intervals. Process the entire file.
7. The last substantive set of code will reduce `employerDict` to the dictionary illustrated by Eq. (2.2). Having become familiar with dictionaries, we may describe the reduced dictionary, `reducedDict`, as a dictionary of dictionaries. The keys of `reducedDict` are to be employers and the values are to be dictionaries. The keys of these internal or sub-dictionaries are the party labels `'Other'`, `'DEM'`, and `'REP'` and the values are the total of contributions to each party made by employees of a company. Suppose that `GMC` is a key of `reducedDict`. Then, the value associated with `GMC` may appear as so:

   `reducedDict['GMC']`$=\{$`'Other'` $: 63000,$ `'DEM'` $: 73040,$ `'REP'`$: 103750\}$.

   The reduced dictionary will be formed by building a dictionary named `totals` for each employer. The dictionary `totals` will then be stored as the value associated with the employer key in `reducedDict`. Begin by initializing the reduced dictionary and iterating over each key-value pair in `employerDict`:

```
reducedDict = {}
for key in employerDict:                # Iterate over employerDict.
    totals = {'REP':0,'DEM':0,'Other':0} # Initialize the dictionary.
    for value in employerDict[key]:
        try :
            totals[value[0]] += value[1]
        except KeyError:
            totals['Other'] += value[1]
    reducedDict[key] = totals
```

   The value associated with `employerDict[key]` is a list of pairs (political party and contribution amount). The `for` loop that iterates over `employerDict[key]` extracts each pair as `value`. The first element `value[0]` is a party and the second element `value[1]` is a dollar amount. The first two lines of this code segment are not indented. The code

```
try :
    totals[value[0]] += value[1]
```

   attempts to add the contribution amount stored in `value[1]` with the party name stored in `value[0]`. If the party name is not `'REP'`, `'DEM'` or `'Other'`, then the `Python` interpreter will produce a `KeyError` exception (an error) and program flow will be directed to the instruction `totals['Other'] += value[1]`. The result is that the contribution amount is added to the other party total. The `try` and `except` construct is called an *exception handler*.

8. We will want to sort the entries in the employer dictionary according to the total of all contributions made by employees of an employer. Immediately after the statement `reducedDict = {}`, initialize a dictionary named `sumDict` to contain the total of all contributions made by employees of each employer. Add an instruction that computes the sum of the three dictionary values and stores it with the employer key. The instruction is

```
sumDict[key] = totals['REP'] + totals['DEM'] + totals['Other']
```

   The instruction executes immediately after the assignment instruction `reducedDict[key] = totals` from step 7 and it should be aligned with it.
9. Add an instruction so that the execution of the script may be monitored, say,

```
if sumDict[key] > 10000 : print(key, totals)
```

   Indent this instruction so that it executes on every iteration of the `for` loop. This print statement is the last instruction in the `for` loop.
10. Now that `sumDict` has been built, we will create a list from the dictionary in which the largest contribution sums are the first elements. Specifically, sorting `sumDict` with respect to the sums will create the sorted list. The resulting list consists of key-value pairs in the form $[(\mathtt{k}_1, \mathtt{v}_1), \ldots, (\mathtt{k}_n, \mathtt{v}_n)]$ where $\mathtt{k}_i$ is the ith key and $\mathtt{v}_i$ is the ith value. Because the list has been sorted, $\mathtt{v}_1 \geq \mathtt{v}_2 \geq \cdots \geq \mathtt{v}_n$. Since `sortedList` is a list, we can print the employers with the 100 largest sums using the code

```
sortedList = sorted(sumDict.items(), key=operator.itemgetter(1))
n = len(sortedList)
print(sortedList[n-100:])
```

    If `a` is a list, then the expression `a[:10]` extracts the first ten elements and `a[len(a)-10:]` extracts the last 10. These operations are called *slicing*.
11. Write a short list of the largest 200 employers to a text file. We'll use `R` to construct a plot similar to Fig. 2.2. The code follows.

```
path = '../employerMoney.txt'
with open(path,'w') as f:
   for i in range(n-200, n):
        employerName = sortedList[i][0].replace("'", "")
        totals = reducedDict[employerName]
        outputRecord = [employerName] + [str(x) for x in totals.
                       values()] + [str(sortedSums[i][1])]
        string = ';'.join(outputRecord) +'\n'
        f.write(string)
```

The `'w'` argument must be passed in the call to `open` to be able to write to the file. Some employer names contain apostrophes and will create errors when `R` reads the file so we must remove the apostrophes from the employer name before the data is written to the output file. Applying the `replace` operator to the string `sortedList[i][0]` removes the apostrophes wherever they are found in the string.

The list `outputRecord` is created by concatenating three lists (each is enclosed by brackets) using the `+` operator. The middle list is created by list comprehension. List comprehension is discussed in detail in the Sect. 2.7.1. For now, it's a method of creating a list using a `for` loop.

In the instruction

```
string = ';'.join(outputRecord)+'\n'
```

`outputRecord` is converted to a string using the `.join` operator. A semicolon joins each list element in the creation of the string. The semicolon serves as a delimiter so that the output file is easy to process in `R`. A delimiter is a symbol that is used to separate variables. For example, commas are used to separate variables in a comma-delimited file (often known as a csv file.) Lastly, the end-of-line marker `\n` is concatenated to the list.

12. Use the following `R` code to generate a figure showing the largest contribution totals to Republican and Democratic parties from company employees. The instruction `s = 160:190` effectively ignores the largest ten employers (you can see that most of the largest 10 are not true employers but instead identify contributors that are self-employed, retired, and so on). The instruction `order(v)` returns a vector of indices that orders the vector `v` from smallest to largest.

```
Data = read.table('../Data/employerMoney.txt' ,sep=';', as.is = TRUE)
colnames(Data) = c('Company', 'Rep', 'Dem', 'Other', 'Total')
head(Data)
print(Data[,1])
s = 160:190  # Select specific rows to plot.
D = Data[s,]                # Take the subset.
D = D[order(D$Rep+D$Dem),]  # Re-order the data according to the total.
rep = D$Rep/10^5            # Scale the values.
dem = D$Dem/10^5
mx = max(rep+dem)
names = D[,1]
n = length(rep)
# Fix the plot window for long names.

plot(x = c(0,mx),y=c(1,n),yaxt = 'n',xlab
  = "Dollars - 100,000's",cex.axis = .65,typ = 'n',ylab='',cex.lab=.8)
axis(side = 2, at = seq(1,n),labels = names, las = 2, cex.axis = .65)
for (i in 1:n) {
  lines(y=c(i,i),x=c(0,rep[i]),col='red',lwd=3)
  lines(y=c(i,i),x=c(rep[i],rep[i]+dem[i]),col='blue',lwd=3)
}
par(oma=c(0,0,0,0))  # Reset the plotting window to default values.
```

2.7 Similarity Measures

Similarity measurements are routinely used to compare individuals and objects. They're used extensively by recommendation engines—algorithms that recommend products and services to potential customers. The underlying premise is that if viewer **A** is similar to viewer **B** with respect to the movies that they have watched, then movies watched by **A** and not by **B** may be recommended to **B**. In the case of political fund-raising, knowing that contributors to candidate **A** are likely to contribute to candidate **B** allows a campaign committee to more effectively direct their fund raising efforts.

Similarity measurements may also provide insight to a new or little-understood entity, say, a campaign committee that is secretive about its objectives. Suppose that a campaign committee **A** has received contributions from committees collected in the set $A = \{a_1, \ldots, a_n\}$. An analyst can gain an understanding of a secretive committee by determining its similarity to known committees. To proceed, a function is needed that will measure similarity between entities **A** and **B** based on sets A and B. More specifically, we need at least one similarity measure. We will examine two measures of similarity: Jaccard similarity and conditional probabilities.

Let $|A|$ denote the cardinality of the set A. If S is finite, then $|S|$ is the number of elements in S. The Jaccard similarity between sets A and B is the

number of elements in both A and B relative to the number of elements in either A or B. Mathematically, the Jaccard similarity is

$$J(A,B) = \frac{|A \cap B|}{|A \cup B|}. \tag{2.4}$$

Jaccard similarity possesses several desirable attributes:

1. If the sets are the same then the Jaccard similarity is 1. Mathematically, if $A = B$, then $A \cap B = A \cup B$ and $J(A,B) = 1$.
2. If the sets have no elements in common, then $A \cap B = \emptyset$ and $J(A,B) = 0$.
3. $J(A,B)$ is bounded by 0 and 1 because $0 \leq |A \cap B| \leq |A \cup B|$.

Jaccard similarity is particularly useful if all possible elements of $A \cup B$ are difficult or expensive to determine. For example, suppose that individuals are to be grouped according to the similarity of their gut microbiota.[4] The possible number of resident species of microorganisms may number in the hundreds of thousands but the number that is expected be found in a stool sample will tend to be many times less. In this case, similarity should be measured on the basis of the species that are present and not on the basis of species that are absent since the vast majority of possible resident species will not be present. Jaccard similarity depends only on the distribution of species in $A \cup B$ and those species absent from the union have no bearing on the value of $J(A,B)$.

Jaccard similarity is an imperfect similarity measure since if the numbers of elements $|A|$ and $|B|$ are much different, say $|A| << |B|$,[5] then

$$|A \cap B| \leq |A| << |B| \leq |A \cup B|. \tag{2.5}$$

Inequality (2.5) implies that the denominator of the Jaccard similarity (formula (2.4)) will be much larger than the numerator and hence, $J(A,B) \approx 0$. This situation will occur if a new customer, **A**, is very much like **B** in purchasing habits and has made only a few purchases (recorded in A). Suppose that all of these purchases have been made by **B** and so whatever **B** has purchased ought to be recommended to **A**. We recognize that **A** is similar **B**, given the information contained in A. But, $J(A,B)$ is necessarily small because the combined set of purchases $A \cup B$ will be much larger in number than the set of common purchases $A \cap B$. There's no way to distinguish this situation between that of two individuals with dissimilar buying habits. Thus, it's beneficial to have an alternative similarity measure that will reveal the relationship.

An alternate measure of similarity that will meaningfully reflect substantial differences in the cardinalities of the sets A and B is the *conditional*

[4] The gut microbiota consists of the microorganism species populating the digestive tract of an organism.

[5] This notation conveys that $|A|$ is much smaller than $|B|$.

probability of an event. The conditional probability of the event A given B is the probability that A will occur given that B has occurred. This conditional probability is denoted as $\Pr(A|B)$ and it defined by

$$\Pr(A|B) = \frac{\Pr(A \cap B)}{\Pr(B)}, \tag{2.6}$$

provided that $\Pr(B) \neq 0$. If $\Pr(B) = 0$, then the conditional probability is undefined and without interest since the event B will not occur. If there are substantive differences between the unconditional probability of A, $(\Pr(A))$ and the conditional probability of A given B, then B is informative with respect to the occurrence of A. On the other hand, if $\Pr(A|B) \approx \Pr(A)$, then B provides little information about the occurrence of A. Furthermore, A and B are said to be independent events whenever $\Pr(A|B) = \Pr(A)$. Lastly, events A and B are independent if and only if $\Pr(A \cap B) = \Pr(A)\Pr(B)$, a statement that may be deduced from formula (2.6) and the definition of independence.

To utilize conditional probabilities in the analysis of political committees, consider a hypothetical experiment in which a committee is randomly drawn from among a list of all committees that have contributed in a particular election cycle. The event A is that of drawing a committee that has contributed to committee **A**. Since committees are randomly selected, $\Pr(A)$ is the proportion of committees that have identified **A** as a recipient of one or more of their contributions. Let $|A|$ denote the number of committees contributing to **A** and n denote the total number of committees. Then,

$$\Pr(A) = \frac{|A|}{n}.$$

The event B is defined in the same manner as A and so $\Pr(B)$ is the proportion of committees contributing to **B**. The probability that a randomly selected committee has contributed to both **A** and **B** during a particular election cycle is $\Pr(A \cap B) = |A \cap B|/n$. The conditional probability of A given B is defined by Eq. (2.6) but can be rewritten in terms of the numbers of contributors:

$$\begin{aligned} \Pr(A|B) &= \frac{|A \cap B|/n}{|B|/n} \\ &= \frac{|A \cap B|}{|B|}. \end{aligned} \tag{2.7}$$

Conditional probabilities may be used to address the deficiencies of the Jaccard similarity measure. Recall that when there are large differences in the frequencies of the events A and B, say $|A| << |B|$, then $J(A, B)$ must be small even if every contributor to **A** also contributed to **B** so that $A \subset B$. It's desirable to have a measure that conveys similarity of **B** to **A**. The conditional

probability $\Pr(B|A)$ does so. To see why, suppose that $0 < |A| << |B|$ and that nearly every contributor to **A** also contributed to **B** and, hence, $A \cap B \approx A$. Then,

$$\begin{aligned} \Pr(A|B) &= \frac{|A \cap B|}{|B|} \approx \frac{|A|}{|B|} \approx 0 \\ \text{and } \Pr(B|A) &= \frac{|A \cap B|}{|A|} \approx 1. \end{aligned} \tag{2.8}$$

Since $\Pr(B|A)$ is nearly 1, it's very likely that any committee that contributes to **A** will also contribute to **B**. In summary, an analysis of the committee similarity will be improved by utilizing conditional probabilities in addition to Jaccard similarities.

It's appropriate to think of the set of Federal Elections Commission records for a particular election cycle as a population and compute the exact probabilities of the events of interest given the experiment of randomly sampling from the list of all committees. Viewing the data set as a population usually is not justified. Instead, the usual situation is that the data are a sample from a larger population or process. For instance, if the data consist of all point-of-sale records collected by a business during a sample window, say, a single day or week, then the data should viewed as a sample from a larger population of records spanning a time span of interest, say, a fiscal quarter or a year. In this sample context, the proportions used above must be viewed as probability estimates, and usually we would define $|A|$ as the number times that the event A occurred during the sample window and n as the number of outcomes (sales) observed during the sample window. The *hat* notation, e.g., $\widehat{\Pr}(A) = |A|/n$ is used to emphasize the uncertainty and error associated with using the estimator $\widehat{\Pr}(A)$ in place of the true, unknown probability $\Pr(A)$. Finally, when probability estimates are computed as relative frequencies, as in this example, the term *empirical probabilities* is often used.

2.7.1 Computation

We now to turn the matter of computing the Jaccard similarity and the conditional probabilities for a large set of committee pairs. An algorithm for forming pairs is needed since the set of committees $\{\mathbf{A}, \mathbf{B}, \mathbf{C}, \ldots\}$ for a particular election cycle is too large to form the set manually. Once the set of pairs has been constructed, then an algorithm will process the pairs and compute the numbers of committees that have contributed to a particular committee, the number of contributors to both committees, and the number of contributors to at least one of the two committees. From these counts, the three similarity measures are computed. The last operation is to sort the pairs with respect to Jaccard similarity, from least to largest and print the committee names, Jaccard similarity, and conditional probabilities.

The `Python` method of *list comprehension* is a concise and computationally efficient method for building the set of pairs $\{(\mathbf{A},\mathbf{B}),(\mathbf{A},\mathbf{C}),(\mathbf{B},\mathbf{C}),\ldots\}$. List comprehension is an analogue of the mathematical syntax for set definition in which the elements of a population or sample space are identified as members of the set by whether or not they satisfy a specified condition. For example, the even integers are $W = \{x|x \bmod 2 = 0\}$.The mathematical definition of W is translated as *the set of values x such that x modulo 2 is 0*. The advantage of list comprehension is that the code is compact yet readily understood and computationally faster than other methods of building a list. A list comprehension expression is contained by a pair of brackets since brackets define lists in `Python`. The brackets contain an expression to be computed followed by one or more `for` or `if` clauses. For example, a list of squares can be constructed using the list comprehension `s = [i**2 for i in range(5)]`. The expression is `i**2` and there is a single `for` clause. An alternative to using list comprehension initializes `s` and then fills it:

```
s = [0]*5  # Create a 5-element list filled with zeros.
for i in range(5):
    s[i] = i**2
```

Another example builds the set of all pairs (i,j) with $i < j$ formed from the set $\{0,1,\ldots,n\}$. Without list comprehension, a pair of nested `for` clauses are used in which the outer `for` clause iterates over $0,1,\ldots,n-1$ and the inner `for` clause iterates over $i+1,\ldots,n$. Running the inner iteration from $i+1$ to n insures that i will always be less than j and that all pairs will be formed. Code for forming the set without list comprehension follows:

```
pairs = set({})  # Create the empty set.
for i in range(n):
    for j in range(i+1, n+1, 1):
        pairs = pairs.union({(i,j)})
```

Note that a singleton set `{(i,j)}` is created containing the pair (i,j) before putting the pair into the set `pairs` using the `union` operator. The `union` operator requires that the two objects to be combined as one set are both sets.

Set comprehension is the analogue of list comprehension for building sets instead of lists. For building a set of pairs using set comprehension, two nested for clauses are required:

```
pairs = {(i,j) for i in range(n) for j in range(i+1, n+1, 1)}
```

Since the number of pairs is $n(n-1)/2$, where n is the number of elements from which to form the sets, the number of possible pairs is of the order n^2. When n is much larger than 100, the computational demands of building and processing the set of pairs may be excessive.

Returning to the problem of identifying pairs of similar committees among a large collection of committees, it will be necessary to determine the number of committee pairs that must be examined before computing similarities. Furthermore, it will be necessary to reduce the number of contributing committees in the contributor dictionary by removing some of the committees. For example, if the analysis is limited to political action committees that have made contributions to many recipients, then the population of interest is effectively limited to committees with expansive agendas such as protecting Second Amendment rights. Alternatively, the analysis may be limited to narrow-focus committees by selecting those committees that contributed to fewer than 50 recipients.

The next tutorial provides practice in working with dictionaries and programming the similarity measures discussed above.

2.8 Tutorial: Computing Similarity

The objective of this tutorial is to identify political campaign committees that are alike with respect to their contributors. For each political campaign committee, you are to construct a set of other committees from which the committee in question has received contributions. Let A and B denote two sets of contributing committees. Then, the similarity between the two committees, say, **A** and **B**, will be determined based on the numbers of common contributor committees, and three measures of similarity will be computed: $J(A, B)$, $\Pr(A|B)$ and $\Pr(B|A)$. The final step is to sort the committee pairs with respect to similarity and produce a short list of the most similar pairs of committees.

1. Decide upon an election cycle and retrieve the following files from the FEC website:
   a. The Committee Master file linking the committee identification code (field position 0) with the committee name (field position 1). The compressed Committee Master files are named `cm.zip`.
   b. The between-committee transaction file[6] linking the recipient committee (field position 0) with the contributing committee (field position 7). Between-committee transaction files are named `oth.zip`. The decompressed file has the name `itoth.txt`.

[6] Named *Any Transaction from One Committee to Another* by the FEC.

2. Process the Committee Master file and build a dictionary linking committee identification codes with committee names by setting the dictionary keys to be committee identification codes (located in field position 0 of the data file) and the dictionary values to be the committee names (field position 1 of the data file).

```
path = '../cm.txt'
nameDict = {}
with open(path) as f:
    for line in f:
        data = line.split("|")
        if data[1] != '':
            nameDict[data[0]] = data[1]
```

3. Create a set named `committees` containing the committee identification numbers:

```
print('Number of committees = ',len(nameDict))
committees = set(nameDict.keys())
```

   The `.keys()` operator extracts the keys from a dictionary.
4. Construct another dictionary, call it `contributorDict`, in which the dictionary keys are the committee identification codes and the values are sets containing the names of the committees that have contributed to the committee identified by identification code. The value for a new key is created using braces to contain the contributing committee name. Additional committees are combined with the set using the instruction `A.union({a})`, where A is a set and a is an element. For example, $\{a\}$ is the singleton set containing a.

```
path = '../itoth.txt'
contributorDict = {}
with open(path) as f:
    for line in f:
        data = line.split("|")
        contributor = data[0]
        if contributorDict.get(contributor) is None:
            contributorDict[contributor] = {data[7]}
        else:
            contributorDict[contributor]
                  = contributorDict[contributor].union({data[7]})
```

5. Determine the number of contributors by finding the number of keys in `contributorDict`. Compute the number of pairs.

```
n = len(contributorDict)
print('N pairs = ',n*(n-1)/2)
```

If the number of pairs is large, say greater than 10^5, then its best to reduce the number of committees by selecting a subset. For example, you may limit the set of committees to only those that made at least m contributions in an election cycle. In the following code, key-value pairs are removed using the `pop()` function when the number of committees to which contributions were made is less than or equal to 500. We iterate over a list of dictionary keys because items cannot be deleted from a dictionary while iterating over the dictionary. Check that the number of remaining committees is sufficiently small, say less than 300.

```
for key in list(contributorDict.keys()):
    if len(contributorDict[key]) <= 500:
        contributorDict.pop(key, None)
n = len(contributorDict)
print('N pairs = ',n*(n-1)/2)
```

What's left in `contributorDict` are committees that are dispersing a lot of money to many political committees, and hence might be viewed as influential.

6. Iterate over `contributorDict` and print the lengths of each value to check the last block of code.

```
for key in contributorDict:
    print(nameDict[key], len(contributorDict[key]))
```

The length of a contributor value is the number of committees that the committee has contributed to in the election cycle.

7. Extract the keys and save as a list:

```
contributors = list(contributorDict.keys())
```

8. Use list comprehension to build a list containing the $n(n-1)/2$ pairs:

```
pairs = [(contributors[i], contributors[j]) for i in range(n-1)
         for j in range(i+1, n, 1)]
```

9. Compute the similarity between pairs of committees by iterating over `pairs`. For each pair in the list `pairs`, extract the sets of contributors and the compute Jaccard similarity, $\Pr(A|B)$, and $\Pr(B|A)$. Then store the three similarity measures in a dictionary named `simDict`.

```
simDict = {}
for commA, commB in pairs:
    A = contributorDict[commA]  # Set of contributors to commA.
    nameA = nameDict[commA]
    B = contributorDict[commB]
    nameB = nameDict[commB]

    nIntersection = len(A.intersection(B))
    jAB = nIntersection/len(A.union(B))
    pAGivenB = nIntersection/len(B)
    pBGivenA = nIntersection/len(A)

    simDict[(nameA, nameB)] = (jAB, pAGivenB, pBGivenA)
```

The keys for the similarity dictionary are the pairs `(nameA, nameB)` consisting of the names of the committees rather than the committee identification codes.

Pairs may be used as dictionary keys because tuples are immutable. The statement `nIntersection/len(A.union(B))` will perform integer division if the `Python` version is less than 3.0; if your `Python` version is less than 3.0, then the denominator must be cast as a floating point number before division takes place, say:

```
    jAB = nIntersection/float(len(A.union(B)))
```

10. Sort the similarity dictionary using the instruction

```
sortedList = sorted(simDict.items(), key=operator.itemgetter(1),
             reverse=True)
```

The function `sorted` produces a list containing the key-value pairs of `simDict` sorted according to the magnitude of the Jaccard similarity since the `itemgetter()` argument is 1. We've used the optional argument `reverse = True` so that the sorted list goes from largest to smallest Jaccard similarity.

11. Since `sortedList` is a list, we can iterate over the pairs in the list in the following code segment. We refer to the keys of `simDict` as `committees` and the value as `simMeasures` in the `for` statement. `Python` allows the programmer to extract and name the elements of a list or tuple using an

assignment statement such as `nameA, nameB = committees`. Print the committee names, Jaccard similarity, and conditional probabilities from smallest Jaccard similarity to largest:

```
for committees, simMeasures in sortedList:
    nameA, nameB = committees
    jAB, pAB, pBA = simMeasures
    if jAB > .5:
        print(round(jAB, 3), round(pAB, 3),
            round(pBA, 3), nameA + ' | ' + nameB)
```

We could use indices to extract values from the similarity dictionary. The ith Jaccard similarity value is `sortedList[i][1][0]`, for example. The triple indexing of `sortedList` can be understood by noting that `sortedList` is a list in which the ith element is a pair, `sortedList[i]` is a key-value pair, `sortedList[i][0]` is the pair of committee names, and `sortedList[i][0][1]` is the second committee name.

The results of our analysis of the major contributors of 2012 election cycle data are summarized in Table 2.2. The analysis was limited to those committees that made at least 200 contributions during the 2012 election cycle. Because of this prerequisite, Table 2.2 consists almost entirely committees affiliated with large groups of individuals, more specifically, corporations, unions, and associations. The Comcast Corp & NBCUniversal PAC and the Verizon PAC both appeared twice in Table 2.2 showing that these political action committees were very active and also much alike with respect to the recipients of their contributions. The last line of the table shows the entry $\Pr(B|A) = .794$, from which it may be concluded that if the Johnson & Johnson PAC (committee **A**) contributed to a particular entity, then the probability that Pfizer PAC (committee **B**) also contributed to the same entity is .794. On the other hand, given that the Pfizer PAC has contributed to a particular entity, the probability that Johnson & Johnson PAC contributed as well to the committee is much less, .415. Both companies are global pharmaceutical companies. The similarity between the beer wholesalers and realtors is puzzling.

2.9 Concluding Remarks About Dictionaries

A irrevocable property of dictionary keys is that they are immutable. An immutable object resides in a fixed memory location. Immutability is key for optimizing dictionary operations. If we attempted to use a list consisting of two committee identification codes, for instance, `[commA, commB]`, then the `Python` interpreter will produce a `TypeError` because lists are mutable

Table 2.2 The five major committee pairs from the 2012 election cycle with the largest Jaccard similarity. Also shown are the conditional probabilities $\Pr(A|B)$ and $\Pr(B|A)$

Committee **A**	Committee **B**	$J(A,B)$	$\Pr(B\|A)$	$\Pr(A\|B)$
Verizon PAC	Comcast Corp & NBCUniversal PAC	.596	.693	.809
General Electric PAC	Comcast Corp & NBCUniversal PAC	.582	.713	.76
NEA Fund for Children and Public Education	Letter Carriers Political Action Fund	.571	.82	.653
National Beer Wholesalers Association PAC	National Association of Realtors PAC	.57	.651	.821
Verizon PAC	AT&T Federal PAC	.588	.656	.788
⋮	⋮	⋮	⋮	⋮
Johnson & Johnson PAC	Pfizer PAC	.375	.415	.794

and cannot be used as dictionary keys. Dictionary keys are stored in semi-permanent memory locations—the location does not change as long as the key-value pair is in the dictionary. If the key were to change, then the amount of memory needed to contain the key might change, and a different location would be needed. Semi-permanent memory locations allows the dictionary to be optimized and hence, operations on dictionaries are fast. Fast operations are important when working with large dictionaries and why dictionaries are used at every opportunity in this text.

2.10 Exercises

2.10.1 Conceptual

2.1. Show that the number of pairs satisfying $i < j$ that may be formed from $i, j \in \{1, 2, \ldots, n\}$ is $\frac{n(n-1}{2}$ by rearranging the expression

$$\sum_{i=1}^{n}\sum_{j=1}^{n} 1 = \sum_{i=1}^{n-1}\sum_{j=i+1}^{n} 1 + \sum_{j=1}^{n-1}\sum_{i=j+1}^{n} 1 + \sum_{i=j=1}^{n} 1.$$

Then solve for $\sum_{i=1}^{n-1}\sum_{j=i+1}^{n} 1$.

2.2. a. Give a formula for $C_{n,3}$, the number of sets or combinations that can be formed by choosing three items from n without replication. Recall that the standard convention in mathematics is to disallow an item to appear in a set more than once. For example, $\{a, b, b\}$ is not used since $\{a, b, b\} = \{a, b\}$. Furthermore, the order of arranging the elements of a set does not change the set since $\{a, b, c\} = \{c, a, b\}$. Therefore, there's only one set that consists of the items a, b, c. Compute $C_{5,3}$ and $C_{100,3}$.

b. Using list comprehension, generate all three-tuples (i, j, k) such that $0 \leq i < j < k < 5$.
c. Using set comprehension, generate all sets of three elements from the set $\{0, 1, 2, 3, 4\}$.

2.3. Consider the list constructed using the list comprehension

```
lst1 = [{(i, j),(j,i)} for i in range(4) for j in range(i, 4, 1)]
```

a. Describe the structure and contents of `lst1`.
b. Explain why the set contains a single pair when $i = j$.
c. Consider the list created from the instruction

```
lst2 = [{(i, j),(j,i)} for i in range(4)
        for j in range(i, 4, 1) if i !=j]
```

Explain why `lst1` $\neq$ `lst2`.

2.4. Initialize a `Python` list to be `L = [(1,2,3),(4,1,5),(0,0,6)]`. Give the `Python` instructions for sorting `L` according to the first coordinate (or position) of the three-tuples. Give the `Python` instructions for sorting `L` according to the third coordinate of the three-tuples. Give the instruction for sorting in descending order (largest to smallest) using the second coordinate of the three-tuples.

2.10.2 Computational

2.5. This exercise is aimed at determining where political action committee (PAC) money goes. Usually, a PAC collects donations from individual contributors by mail or online solicitation and then donates money to candidates of their choice or to other PACs. Sometimes, the intentions of a PAC may not be transparent, and their donations to other PACs or candidates might not be in accord with the intention of the contributors. For this reason, there is interest in tracking the donations of PACs. Table 2.3 is an example showing a few of the PACs that received donations from the Bachmann for Congress PAC during the 2012 election cycle.

Choose an election cycle, identify a PAC of interest, and create a recipient dictionary listing all of the contributions that were received from the PAC. A convenient format for the recipient dictionary uses the recipient committee code as a key. The value should be a two-element list containing the total

Table 2.3 The top eight recipients of contributions from the Bachmann for Congress PAC during the 2010–2012 election cycle

Recipient	Amount ($)
National Republican Congressional Committee	115,000
Republican Party Of Minnesota	41,500
Susan B Anthony List Inc. Candidate Fund	12,166
Freedom Club Federal Pac	10,000
Citizens United Political Victory Fund	10,000
Republican National Coalition For Life Political Action Committee	10,000
Koch Industries Inc Political Action Committee (Kochpac)	10,000
American Crystal Sugar Company Political Action Committee	10,000

amount received by the recipient PAC and the name of the recipient PAC. PACs also contribute to individual candidates, so if you wanted, you could also track money that was received by individual candidates from the PAC of interest.

A couple of reminders may help:

1. Contributions made by PACs are listed in the FEC data files named *Any Transaction from One Committee to Another* and contained in zip files with the name `oth.zip`.
2. To convert a dictionary `C` into a sorted list named `sC`, largest to smallest, by sorting on the values, use

```
sC = sorted(C.iteritems(), key=operator.itemgetter(1),
    reverse = True)
```

   This will work if the value is a list and the first element of the lists are numeric.

2.6. Use the `timeit` module and function to compare execution times for two algorithms used for constructing sets of pairs $\{(i,j\}|0 \leq i < j \leq n$, for integers i and $j\}$. The first algorithm should use list comprehension and the second should join each pair to the set of pairs using the `union` operator. Set $n \in \{100, 200, 300\}$. Report on the computational time for both algorithms and choice of n. Comment on the differences.

2.7. The Consumer Financial Protection Bureau is a federal agency tasked with enforcing federal consumer financial laws and protecting consumers of financial services and products from malfeasance on the part of the providers of these services and products. The Consumer Financial Protection Bureau maintains a database documenting consumer complaints. Download the database from
http://www.consumerfinance.gov/complaintdatabase/#download-the-data.
Determine which ten companies were most often named in the product category *Mortgage* and the issue category of *Loan modification,collection, foreclosure*. The first record of the data file is a list of attributes.

http://www.springer.com/978-3-319-45795-6

Algorithms for Data Science

Steele, B.; Chandler, J.; Reddy, S.

2016, XXIII, 430 p. 48 illus., 30 illus. in color., Hardcover
ISBN: 978-3-319-45795-6