Complex Structures

SECTIONS 8.3

Complex Structures

- Containers are very useful for storing collections of values
  - In Python, the list and dictionary containers can contain any type of data, including other containers
- Some data collections, however, may require more complex structures.
  - In this section, we explore problems that require the use of a complex structure

A Dictionary of Sets

- The index of a book specifies on which pages each term occurs
- Build a book index from page numbers and terms contained in a text file with the following format:

```
6:type
7:example
7:index
7:program
8:type
10:example
11:program
20:set
```

A Dictionary of Sets

- The file includes every occurrence of every term to be included in the index and the page on which the term occurs
- If a term occurs on the same page more than once, the index includes the page number only once

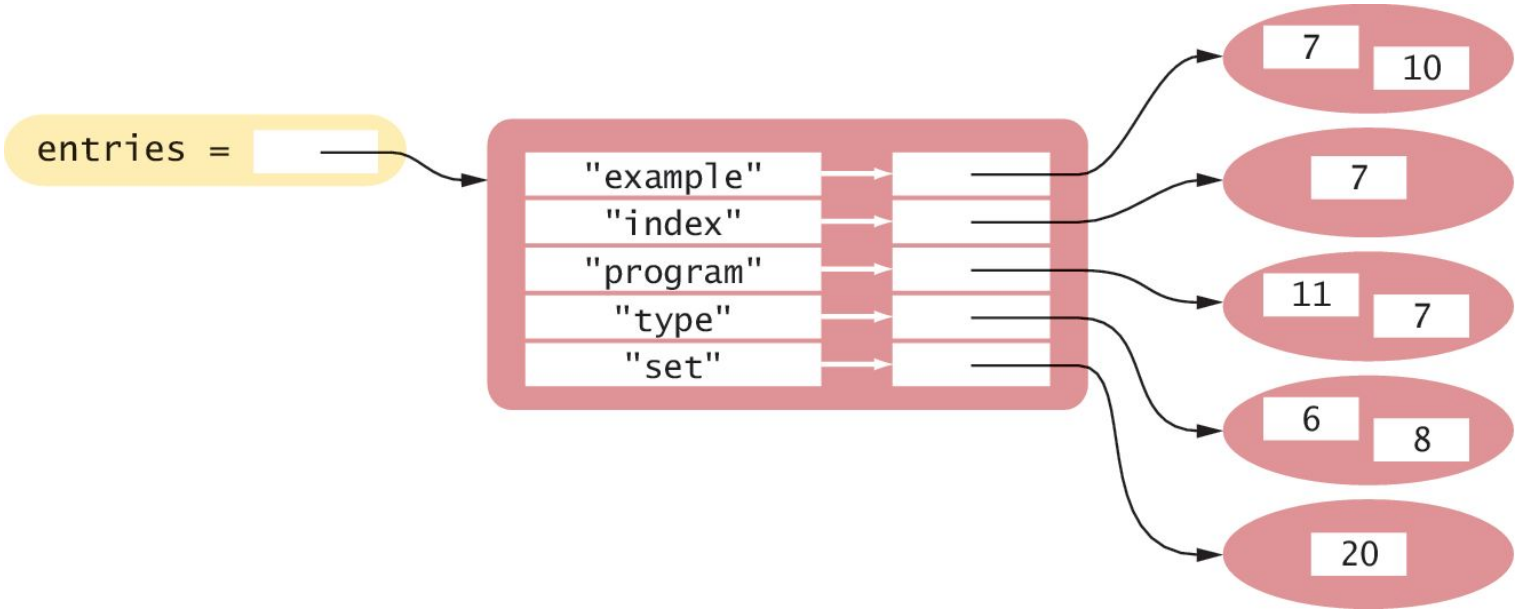
A Dictionary of Sets

- The output of the program should be a list of terms in alphabetical order followed by the page numbers on which the term occurs, separated by commas, like this:

```
example: 7, 10
index: 7
program: 7, 11
type: 6, 8
set: 20
```

A Dictionary of Sets

- A dictionary of sets would be appropriate for this problem
- Each key can be a term and its corresponding value a set of the page numbers where it occurs



Why Use a Dictionary?

- The terms in the index must be unique
  - By making each term a dictionary key, there will be only one instance of each term.
- The index listing must be provided in alphabetical order by term
  - We can iterate over the keys of the dictionary in sorted order to produce the listing
- Duplicate page numbers for a term should only be included once
  - By adding each page number to a set, we ensure that no duplicates will be added

Dictionary Sets: `Buildindex.py`

```
 5  def main() :
 6      # Create an empty dictionary.
 7      indexEntries = {}
 8
 9      # Extract the data from the text file.
10      infile = open("indexdata.txt", "r")
11      fields = extractRecord(infile)
12      while len(fields) > 0 :
13          addWord(indexEntries, fields[1], fields[0])
14          fields = extractRecord(infile)
15
16      infile.close()
17
18      # Print the index listing.
19      printIndex(indexEntries)
```

Dictionary Sets: `Buildindex.py`

```
26  def extractRecord(infile) :
27      line = infile.readline()
28      if line != "" :
29          fields = line.split(":")
30          page = int(fields[0])
31          term = fields[1].rstrip()
32          return [page, term]
33      else :
34          return []
```

Dictionary Sets: `Buildindex.py`

```
41  def addWord(entries, term, page) :
42      # If the term is already in the dictionary, add the page to the set.
43      if term in entries :
44          pageSet = entries[term]
45          pageSet.add(page)
46
47      # Otherwise, create a new set that contains the page and add an entry.
48      else :
49          pageSet = set([page])
50          entries[term] = pageSet
```

Dictionary Sets: `Buildindex.py`

```
56      for key in sorted(entries) :
57          print(key, end=" ")
58          pageSet = entries[key]
59          first = True
60          for page in sorted(pageSet) :
61              if first :
62                  print(page, end="")
63                  first = False
64              else :
65                  print(",", page, end="")
66
67          print()
```

A Dictionary of Lists

- A common use of dictionaries in Python is to store a collection of lists in which each list is associated with a unique name or key
- For example, consider the problem of extracting data from a text file that represents the yearly sales of different ice cream flavors in multiple stores of a retail ice cream company
  - `vanilla:8580.0:7201.25:8900.0`
  - `chocolate:10225.25:9025.0:9505.0`
  - `rocky road:6700.1:5012.45:6011.0`
  - `strawberry:9285.15:8276.1:8705.0`
  - `cookie dough:7901.25:4267.0:7056.5`

A Dictionary of Lists

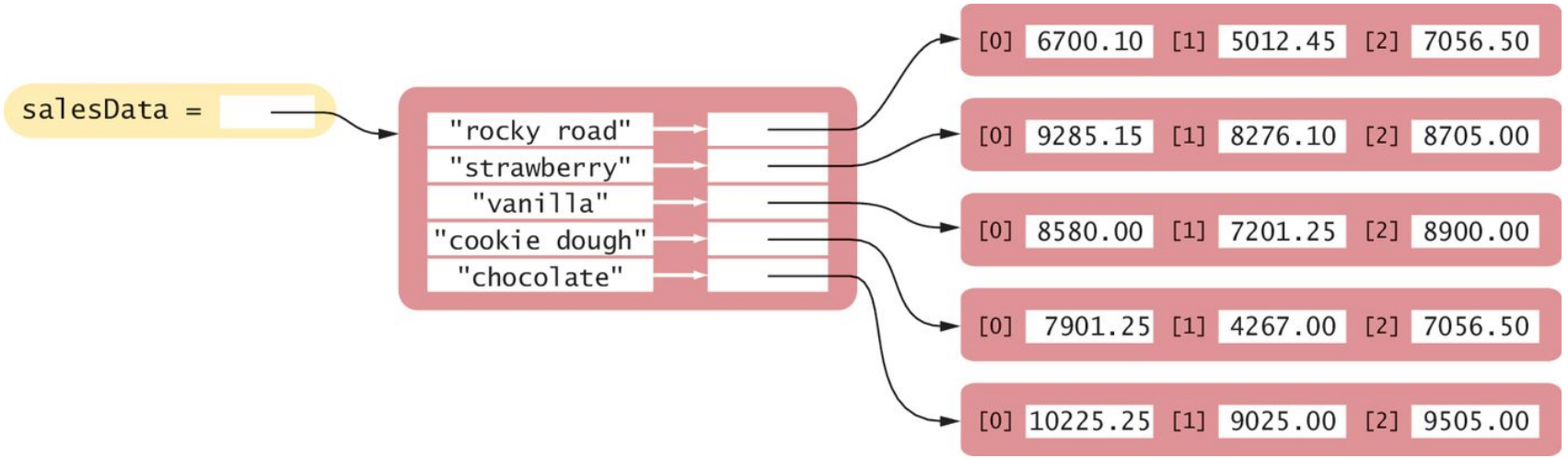
- The data is to be processed to produce a report similar to the following:

chocolate	10225.25	9025.00	9505.00	28755.25
cookie dough	7901.25	4267.00	7056.50	19224.75
rocky road	6700.10	5012.45	6011.00	17723.55
strawberry	9285.15	8276.10	8705.00	26266.25
vanilla	8580.00	7201.25	8900.00	24681.25
	42691.75	33781.80	40177.50	

- A simple list is not the best choice:
  - The entries consist of strings and floating-point values, and they have to be sorted by the flavor name

A Dictionary of Lists

- With this structure, each row of the table is an item in the dictionary
- The name of the ice cream flavor is the key used to identify a particular row in the table.
- The value for each key is a list that contains the sales, by store, for that flavor of ice cream



Example: Icecreamsales.py

```
6  def main() :
7      salesData = readData("icecream.txt")
8      printReport(salesData)
```

Example: `Icecreamsales.py`

```
14  def readData(filename) :
15      # Create an empty dictionary.
16      salesData = {}
17
18      infile = open(filename, "r")
19
20      # Read each record from the file.
21      for line in infile :
22          fields = line.split(":")
23          flavor = fields[0]
24          salesData[flavor] = buildList(fields)
25
26      infile.close()
27      return salesData
```

Example: `Icecreamsales.py`

```
33  def buildList(fields) :
34      storeSales = []
35      for i in range(1, len(fields)) :
36          sales = float(fields[i])
37          storeSales.append(sales)
38
39      return storeSales
```

Example: `Icecreamsales.py`

```
44  def printReport(salesData) :
45     # Find the number of stores as the length of the longest store sales list.
46     numStores = 0
47     for storeSales in salesData.values() :
48        if len(storeSales) > numStores :
49           numStores = len(storeSales)
50
51     # Create a list of store totals.
52     storeTotals = [0.0] * numStores
53
54     # Print the flavor sales.
55     for flavor in sorted(salesData) :
56        print("%-15s" % flavor, end="")
57
```

Example: `Icecreamsales.py`

```
58          flavorTotal = 0.0
59          storeSales = salesData[flavor]
60          for i in range(len(storeSales)) :
61             sales = storeSales[i]
62             flavorTotal = flavorTotal + sales
63             storeTotals[i] = storeTotals[i] + sales
64             print("%10.2f" % sales, end="")
65
66          print("%15.2f" % flavorTotal)
67
68       # Print the store totals.
69       print("%15s" % " ", end="")
70       for i in range(numStores) :
71          print("%10.2f" % storeTotals[i], end="")
72       print()
```

Modules

SPLITTING OUR PROGRAMS INTO PIECES

Modules

- When you write small programs, you can place all of your code into a single source file
- When your programs get larger or you work in a team, that situation changes
- You will want to structure your code by splitting it into separate source files (a “module”)

Reasons for Employing Modules

- Large programs can consist of hundreds of functions that become difficult to manage and debug if they are all in one source file
  - By distributing the functions over several source files and grouping related functions together, it becomes easier to test and debug the various functions
- The second reason becomes apparent when you work with other programmers in a team
  - It would be very difficult for multiple programmers to edit a single source file simultaneously
  - The program code is broken up so that each programmer is solely responsible for a unique set of files

Typical Division Into Modules

- Large Python programs typically consist of a **driver module** and one or more supplemental modules
- The driver module contains the `main()` function or the first executable statement if no main function is used
- The supplemental modules contain supporting functions and constant variables

Modules Example

- Splitting the dictionary of lists into modules
- The `tabulardata.py` module contains functions for reading the data from a file and printing a dictionary of lists with row and column totals
- The `salesreport.py` module is the driver (or main) module that contains the main function
- By splitting the program into two modules, the functions in the `tabulardata.py` module can be reused in another program that needs to process named lists of numbers

Using Code That are in Modules

- To call a function or use a constant variable that is defined in a user module, you can first import the module in the same way that you imported a standard library module:

```
from tabulardata import readData, printReport
```

- However, if a module defines many functions, it is easier to use the form:

```
import tabulardata
```

- With this form, you must prepend the name of the module to the function name:

```
tabulardata.printReport(salesData)
```

Review

Python Sets

- A set stores a collection of unique values
- A set is created using a set literal or the set function
- The `in` operator is used to test whether an element is a member of a set
- New elements can be added using the `add()` method
- Use the `discard()` method to remove elements from a set
- The `issubset()` method tests whether one set is a subset of another set

Python Sets

- The `union()` method produces a new set that contains the elements in both sets
- The `intersection()` method produces a new set with the elements that are contained in both sets
- The `difference()` method produces a new set with the elements that belong to the first set but not the second
- The implementation of sets arrange the elements in the set so that they can be located quickly

Python Dictionaries

- A dictionary keeps associations between keys and values
- Use the `[]` operator to access the value associated with a key
- The `in` operator is used to test whether a key is in a dictionary
- New entries can be added or modified using the `[]` operator
- Use the `pop()` method to remove a dictionary entry

Complex Structures

- Complex structures can help to better organize data for processing
- The code of complex programs is distributed over multiple files