Chapter 8

SETS AND DICTIONARIES

Chapter Goals

- To build and use a set container
- To learn common set operations for processing data
- To build and use a dictionary container
- To work with a dictionary for table lookups
- To work with complex data structures

In this chapter, we will learn how to work with two more types of containers (sets and dictionaries) as well as how to combine containers to model complex structures.

Contents

- Sets
- Dictionaries
- Complex Structures

Sets

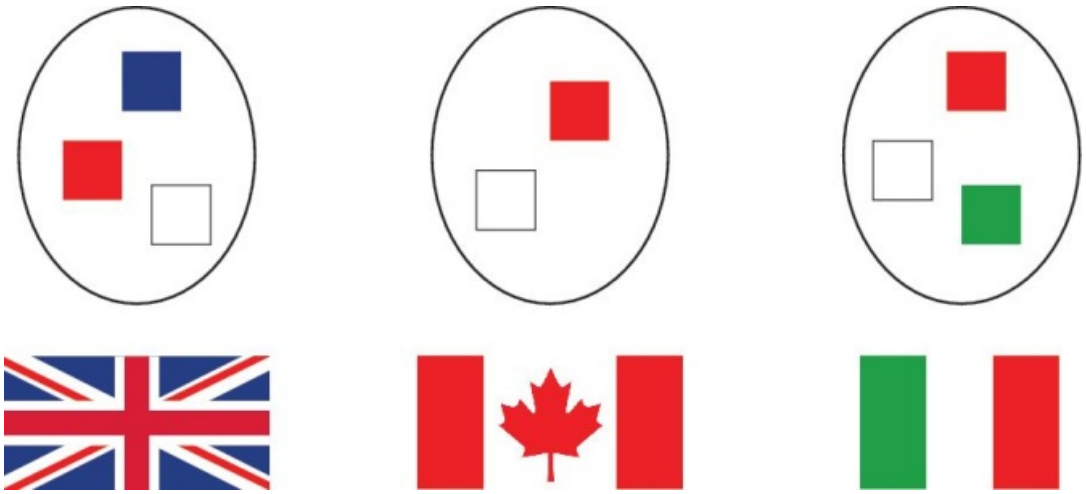
SECTION 8.1

Sets

- A set is a container that stores a collection of unique values
- Unlike a list, the elements or members of the set are not stored in any particular order and cannot be accessed by position
- Operations are the same as the operations performed on sets in mathematics
- Because sets do not need to maintain a particular order, set operations are much faster than the equivalent list operations

Example Set

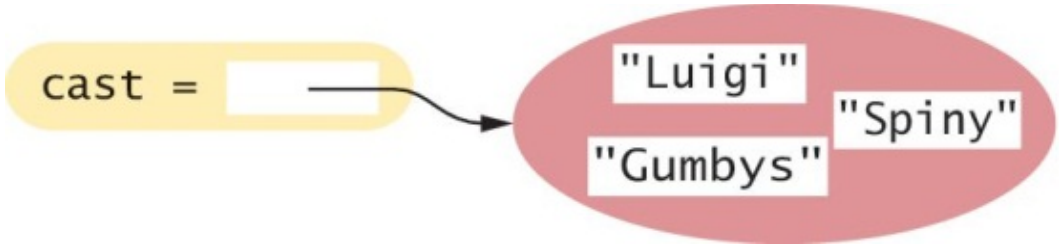
- This set contains three sets of colors—the colors of the British, Canadian, and Italian flags
- In each set, the order does not matter, and the colors are not duplicated in any one of the sets



Creating and Using Sets

- To create a set with initial elements, you can specify the elements enclosed in braces, just like in mathematics:

```
cast = { "Luigi", "Gumbys", "Spiny" }
```



- Alternatively, you can use the set() function to convert any sequence into a set:

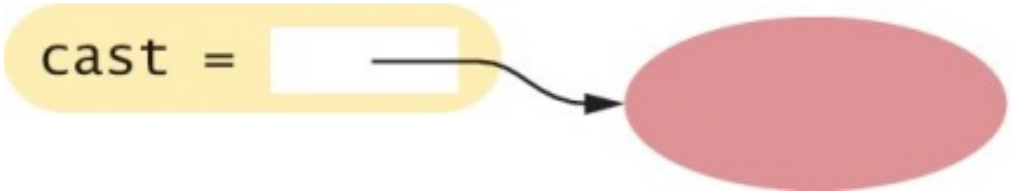
```
names = ["Luigi", "Gumbys", "Spiny"]
cast = set(names)
```

Creating an Empty Set

- For historical reasons, you cannot use `{}` to make an empty set in Python
- Instead, use the `set()` function with no arguments:

```
cast = set()
```

- As with any container, you can use the `len()` function to obtain the number of elements in a set:




```
numberOfCharacters = len(cast)   # In this case it's zero
```

Set Membership: `in`

- To determine whether an element is contained in the set, use the `in` operator or its inverse, the `not in` operator:

```
if "Luigi" in cast :
    print("Luigi is a character in Monty Python's Flying Circus.")
else :
    print("Luigi is not a character in the show.")
```

Accessing Set Elements

- Because sets are unordered, you cannot access the elements of a set by position as you can with a list
- We use a `for` loop to iterate over the individual elements:

```
print("The cast of characters includes:")
for character in cast :
    print(character)
```

- Note that the order in which the elements of the set are visited depends on how they are stored internally

Accessing Elements (2)

- For example, the previous loop above displays the following:

```
The cast of characters includes:
Gumbys
Spiny
Luigi
```

- Note that the order of the elements in the output is different from the order in which the set was created

Displaying Sets In Sorted Order

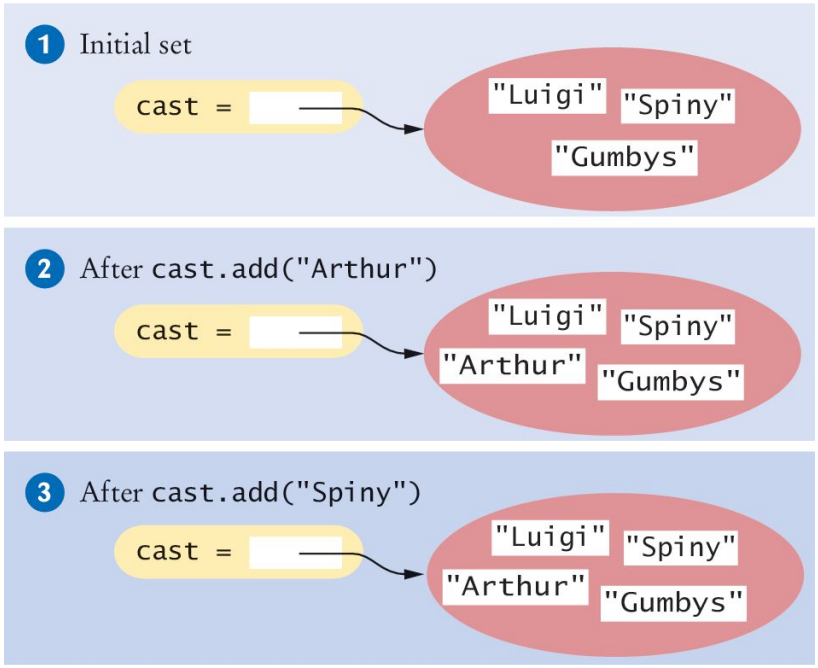
- Use the `sorted()` function, which returns a list ***(not a set)*** of the elements in sorted order
- The following loop prints the cast in sorted order:

```
for actor in sorted(cast) :
    print(actor)
```

Adding Elements

- Sets are ***mutable*** collections, so you can add elements by using the `add()` method:

```
cast = set(["Luigi", "Gumbys", "Spiny"])   #1
cast.add("Arthur")                         #2
cast.add("Spiny")                          #3
```



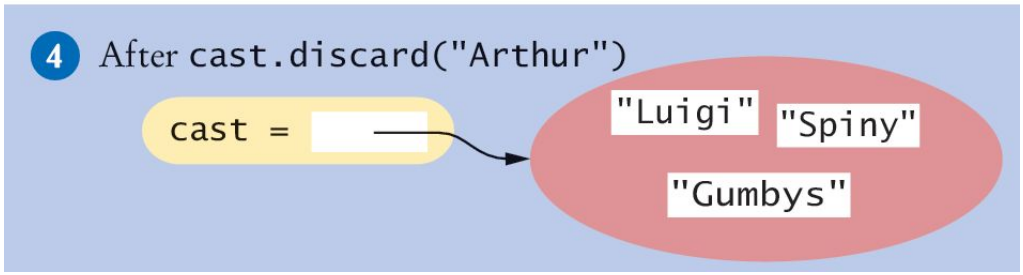
Arthur is not in the set, so it is added to the set and the size of the set is increased by one

Spiny is already in the set, so there is no effect on the set

Removing Elements: `discard()`

- The `discard()` method removes an element if the element exists:

```
cast.discard("Arthur")   #4
```



- It has no effect if the given element is not a member of the set:

```
cast.discard("The Colonel")   # Has no effect
```

Removing Elements: `remove()`

- The `remove()` method, on the other hand, removes an element if it exists, but raises an exception if the given element is not a member of the set:

```
cast.remove("The Colonel")   # Raises an exception
```

- For this class we will use the discard() method

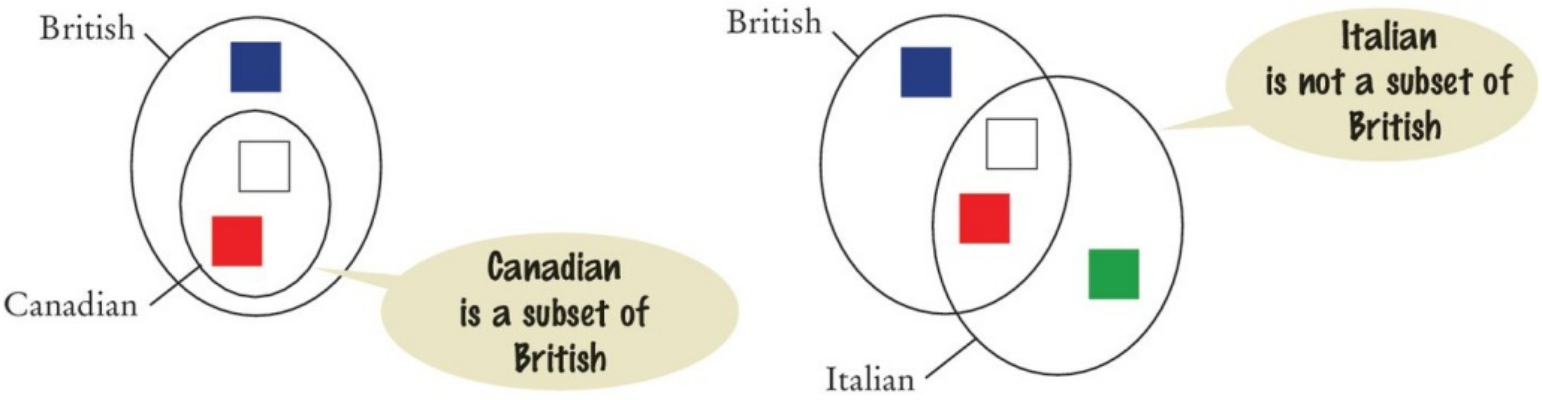
Removing Elements: `clear()`

- Finally, the `clear()` method removes *all* elements of a set, leaving the empty set:

```
cast.clear() # cast now has size 0
```

Subsets

- A set is a *subset* of another set *if and only if* every element of the first set is also an element of the second set
- In the image below, the Canadian flag colors are a subset of the British colors
- The Italian flag colors are not.



The `issubset()` Method

- The `issubset()` method returns `True` or `False` to report whether one set is a subset of another:

```
canadian = { "Red", "White" }
british = { "Red", "Blue", "White" }
italian = { "Red", "White", "Green" }

# True
if canadian.issubset(british) :
    print("All Canadian flag colors occur in the British flag.")

# True
if not italian.issubset(british) :
    print("At least one of the colors in the Italian flag does
       not.")
```

Set Equality / Inequality

- We test set equality with the “==“ and “!=“ operators
- Two sets are equal ***if and only if*** they have exactly the same elements

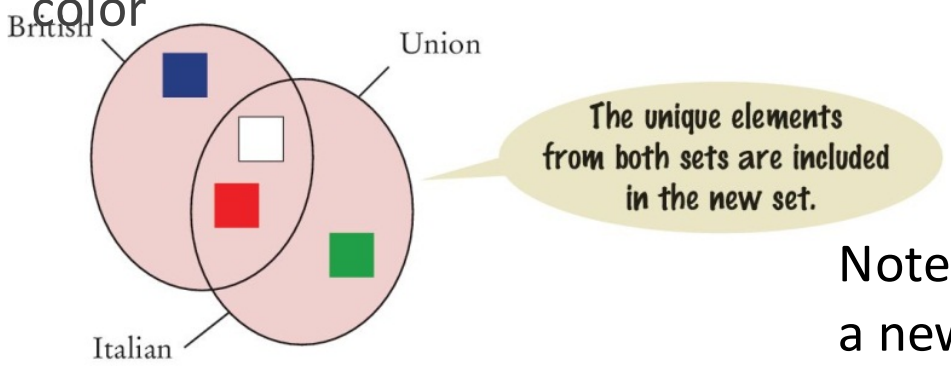
```
french = { "Red", "White", "Blue" }
if british == french :
    print("The British and French flags use the same colors.")
```

Set Union: `union()`

- The *`union`* of two sets contains all of the elements from both sets, with duplicates removed

```
# inEither: The set {"Blue", "Green", "White", "Red"}
inEither = british.union(italian)
```

- Both the British and Italian sets contain the colors Red and White, but the union is a set and therefore contains only one instance of each color

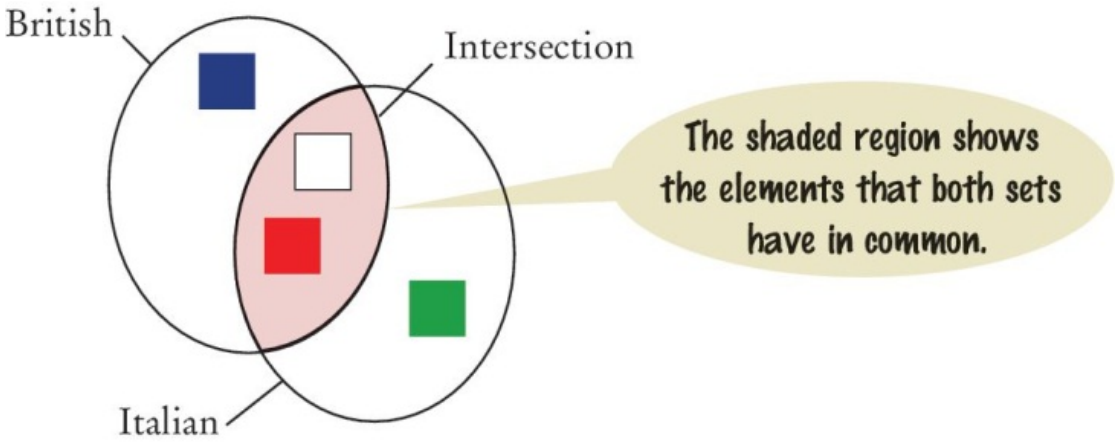


Note that the union() method returns a new set. It does not modify either of the sets
in the call

Set Intersection: `intersection()`

- The *`intersection`* of two sets contains all of the elements that are in *both* sets

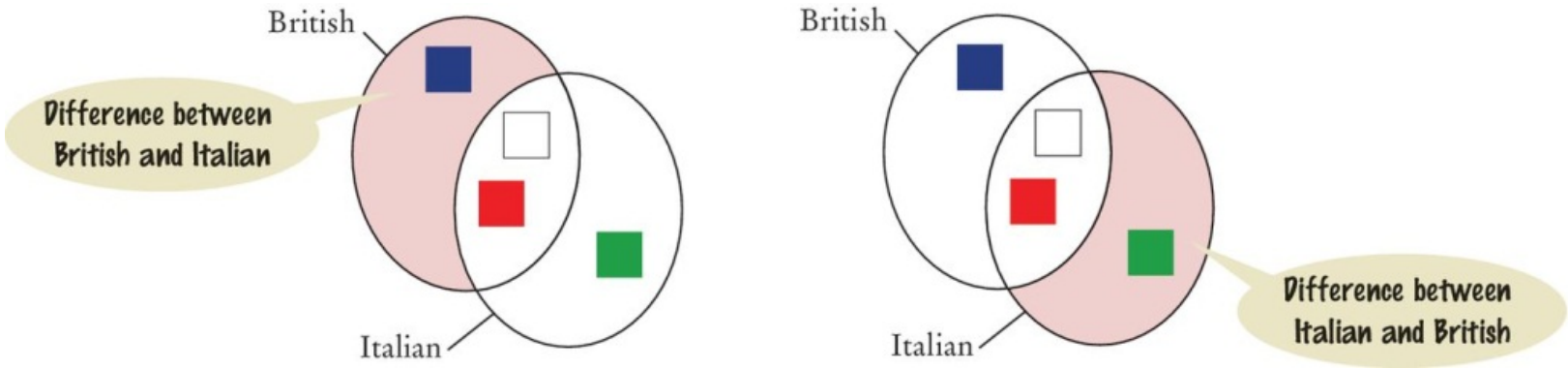
```
# inBoth: The set {"White", "Red"}
inBoth = british.intersection(italian)
```



Difference of Two Sets: `difference()`

- The *`difference`* of two sets results in a new set that contains those elements in the first set that are not in the second set

```
print("Colors that are in the Italian flag but not the
   British:")
print(italian.difference(british)) # Prints {'Green'}
```



Common Set Operations

Table 1 Common Set Operations

Operation	Description

`s = set()`
`s = set(`*seq*`)`
`s = {`e_1`, `e_2`, ..., `e_n`}`
	Creates a new set that is either empty, a duplicate copy of sequence *seq*, or that contains the initial elements provided.

`len(`*s*`)`	Returns the number of elements in set *s*.
element `in` *s* *element* `not in` *s*	Determines if *element* is in the set.
s`.add(`*element*`)`	Adds a new element to the set. If the element is already in the set, no action is taken.
s`.discard(`*element*`)` *s*`.remove(`*element*`)`	Removes an element from the set. If the element is not a member of the set, `discard` has no effect, but `remove` will raise an exception.
s`.clear()`	Removes all elements from a set.
s`.issubset(`*t*`)`	Returns a Boolean indicating whether set *s* is a subset of set *t*.

Common Set Operations (2)

Table 1 Common Set Operations

s == *t* *s* != *t*	Returns a Boolean indicating whether set *s* is equal to set *t*.
s.union(*t*)	Returns a new set that contains all elements in set *s* and set *t*.
s.intersection(*t*)	Returns a new set that contains elements that are in *both* set *s* and set *t*.
s.difference(*t*)	Returns a new set that contains elements in *s* that are not in set *t*.

Remember: `union`, `intersection` and `difference` return new sets
They do not modify the set they are applied to

Simple Examples

- Open the file: set examples.py

Set Example: Spell Checking

- The program spellcheck.py reads a file that contains correctly spelled words and places the words in a set
- It then reads all words from a document—here, the book *Alice in Wonderland*—into a second set
- Finally, it prints all words from the document that are not in the set of correctly spelled words
- Open the file spellcheck.py

Example: Spellcheck.py

```
1  ##
2  #  This program checks which words in a file are not present in a list of
3  #  correctly spelled words.
4  #
5
6  #  Import the split function from the regular expression module.
7  from re import split
8
9  def main() :
10    # Read the word list and the document.
11    correctlySpelledWords = readWords("words")
12    documentWords = readWords("alice30.txt")
13
14    # Print all words that are in the document but not the word list.
15    misspellings = documentWords.difference(correctlySpelledWords)
16    for word in sorted(misspellings) :
17       print(word)
```

Example: Spellcheck.py

```
24  def readWords(filename) :
25     wordSet = set()
26     inputFile = open(filename, "r")
27
28     for line in inputFile :
29        line = line.strip()
30        # Use any character other than a-z or A-Z as word delimiters.
31        parts = split("[^a-zA-Z]+", line)
32        for word in parts :
33           if len(word) > 0 :
34              wordSet.add(word.lower())
35
36     inputFile.close()
37     return wordSet
38
39  # Start the program.
40  main()
```

Execution: `Spellcheck.py`

```
...
champaign
chatte
clamour
comfits
conger
croqueted
croqueting
cso
daresay
dinn
dir
draggled
dutchess
...
```

Programming Tip

- When you write a program that manages a collection of unique items, sets are far more efficient than lists
- Some programmers prefer to use the familiar lists, replacing

```
itemSet.add(item)
```

with:

```
if (item not in itemList)
    itemList.append(item)
```

- However, the resulting program is much slower.
  - The speed factor difference is over 10 times

Counting Unique Words

Problem Statement

- We want to be able to count the number of unique words in a text document
  - "Mary had a little lamb" has 57 unique words
- Our task is to write a program that reads in a text document and determines the number of unique words in the document

Step One: Understand the Task

- To count the number of unique words in a text document we need to be able to determine if a word has been encountered earlier in the document
  - Only the first occurrence of a word should be counted
- The easiest way to do this is to read each word from the file and add it to the set
  - Because a set cannot contain duplicates we can use the add method
  - The add method will prevent a word that was encountered earlier from being added to the set
- After we process every word in the document the size of the set will be the number of unique words contained in the document

Step Two: Decompose the Problem

The problem can be split into several simple steps:

Create an empty set

for each word in the text document
    Add the word to the set

Number of unique words = the size of the set

- Creating the empty set, adding an element to the set, and determining the size of the set are standard set operations
- Reading the words in the file can be handled as a separate task

Step Three: Build the Set

- We need to read individual words from the file. For simplicity in our example we will use a literal file name

```
inputFile = open(“nurseryrhyme.txt”, “r”)
For line in inputFile :
    theWords = line.split()
    For words in theWords :
        Process word
```

- To count unique words we need to remove any nonletters and remove capitalization
- We will design a function to “clean” the words before we add them to the set

Step Four: Clean the Words

- To strip out all the characters that are not letters we will iterate through the string, one character at a time, and build a new “clean” word

```
def clean(string) :
  result = “”
  for char in string :
    if char.isalpha() :
      result = result + char
return result.lower()
```

Step Five: Some Assembly Required

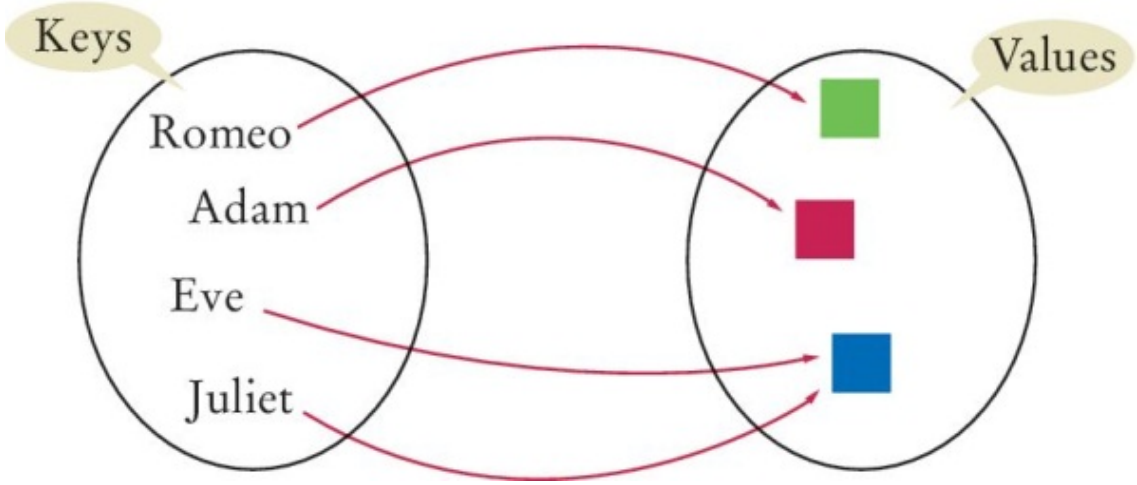
- Implement the main() function and combine it with the other functions
- Open the file: countwords.py

Dictionaries

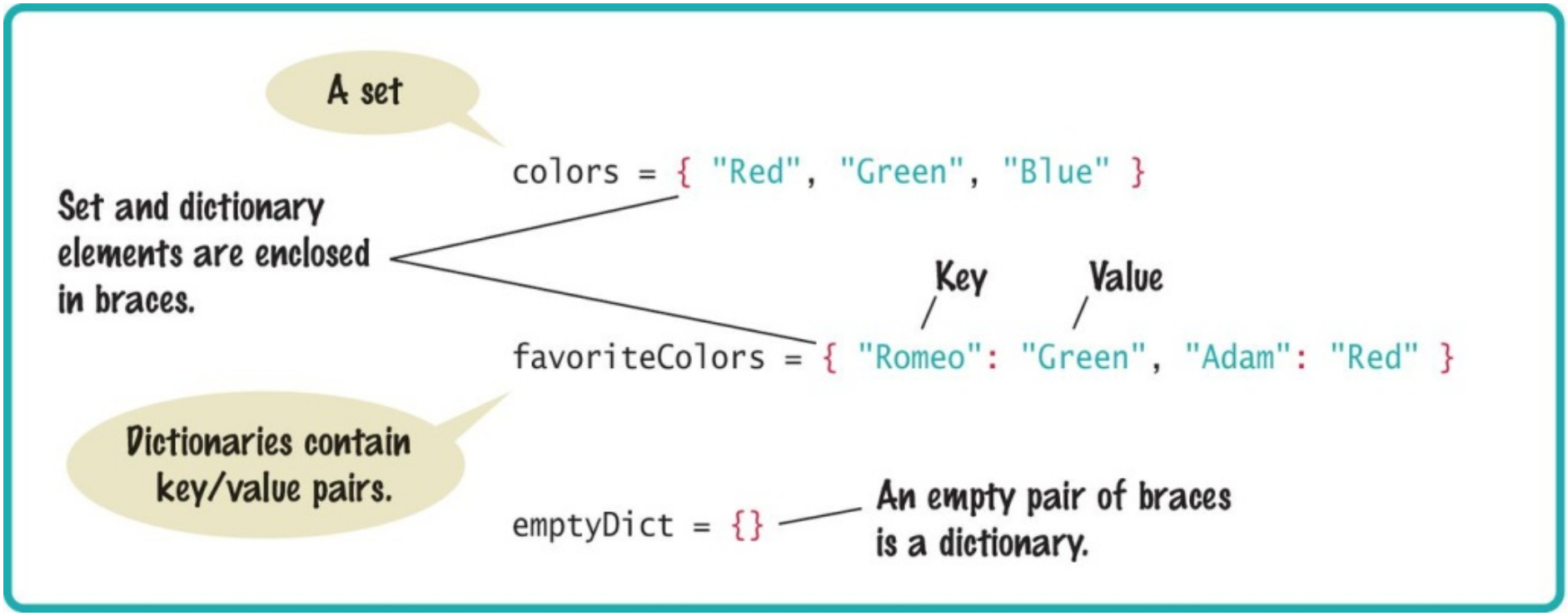
SECTION 8.2

Dictionaries

- A dictionary is a container that keeps associations between *keys* and *values*
- Every key in the dictionary has an associated value
- Keys are unique, but a value may be associated with several keys
- Example (the mapping between the key and value is indicated by an arrow):



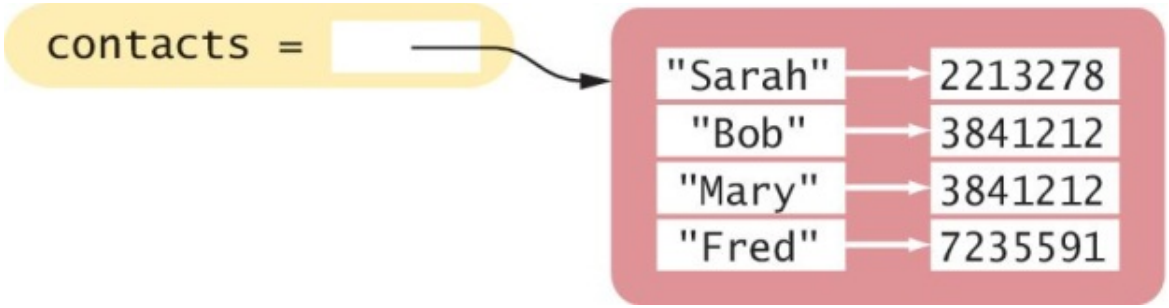
Syntax: Sets and Dictionaries



Creating Dictionaries

- Suppose you need to write a program that looks up the phone number for a person in your mobile phone's contact list
- You can use a dictionary where the names are keys and the phone numbers are values

```
contacts = { "Fred": 7235591, "Mary": 3841212, "Bob":
   3841212, "Sarah": 2213278 }
```



Duplicating Dictionaries: `Dict()`

- You can create a duplicate copy of a dictionary using the `dict()` function:

```
oldContacts = dict(contacts)
```

Accessing Dictionary Values [ ]

- The subscript operator [ ] is used to return the value associated with a key
- The statement

```
# prints 7235591.
print("Fred's number is",
   contacts["Fred"])
```

- Note that the dictionary is not a sequence-type container like a list.
  - You cannot access the items by index or position
  - A value can only be accessed using its associated key

The key supplied to the subscript operator must be a valid key in the dictionary or a KeyError exception will be raised

Dictionaries: Checking Membership

- To find out whether a key is present in the dictionary, use the `in` (or `not in`) operator:

```
if "John" in contacts :
    print("John's number is", contacts["John"])
else :
    print("John is not in my contact list.")
```

Default Keys

- Often, you want to use a default value if a key is not present
- Instead of using the `in` operator, you can simply call the `get()` method and pass the `key` and a `default value`
- The default value is returned if there is no matching key

```
number = contacts.get("Fred", 411)
print("Dial " + number)
```

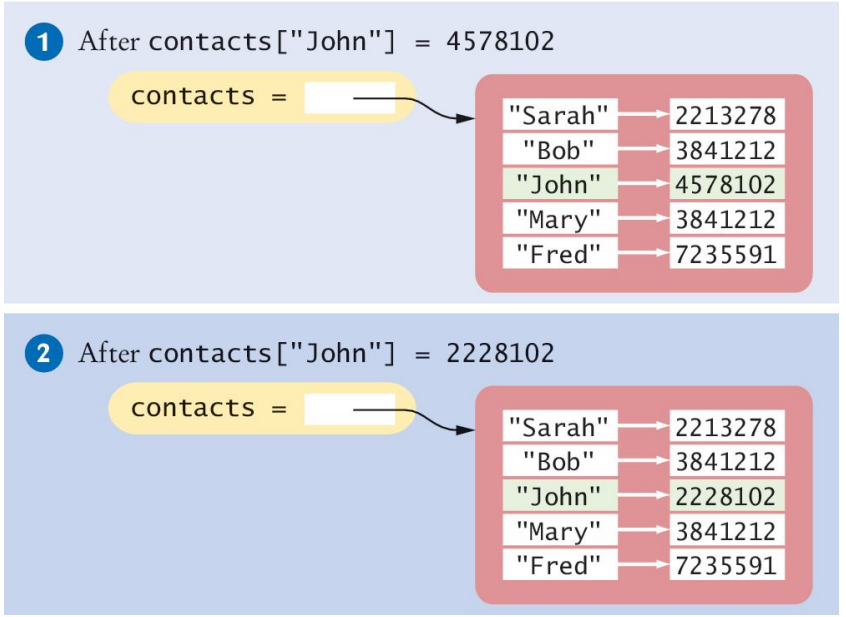
Adding/Modifying Items

- A dictionary is a mutable container
- You can add a new item using the subscript operator [] much as you would with a list

```
contacts["John"] = 4578102   #1
```

- To change the value associated with a given key, set a new value using the [] operator on an existing key:

```
contacts["John"] = 2228102   #2
```



Adding New Elements Dynamically

- Sometimes you may not know which items will be contained in the dictionary when it's created
- You can create an empty dictionary like this:

```
favoriteColors = {}
```

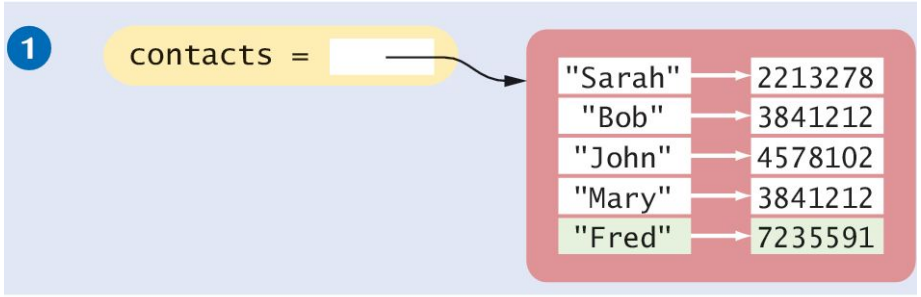
- and add new items as needed:

```
favoriteColors["Juliet"] = "Blue"
favoriteColors["Adam"] = "Red"
favoriteColors["Eve"] = "Blue"
favoriteColors["Romeo"] = "Green"
```

Removing Elements

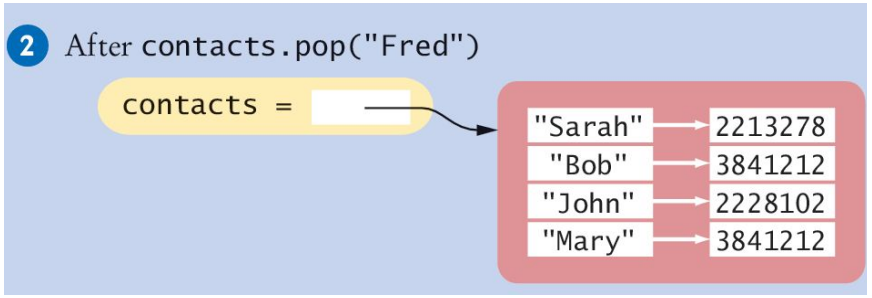
- To remove an item from a dictionary, call the `pop()` method with the key as the argument:

```
contacts = { "Fred":
   7235591, "Mary": 3841212,
   "Bob": 3841212, "Sarah":
   2213278 }
```



- This removes the entire item, both the key and its associated value.

```
contacts.pop("Fred")
```



Removing and Storing Elements

- The `pop()` method returns the value of the item being removed, so you can use it or store it in a variable:

```
fredsNumber = contacts.pop("Fred")
```

- Note: If the key is not in the dictionary, the pop method raises a `KeyError` exception
  - To prevent the exception from being raised, you should test for the key in the dictionary:

```
if "Fred" in contacts :
    contacts.pop("Fred")
```

Traversing a Dictionary

- You can iterate over the individual keys in a dictionary using a `for` loop:

```
print("My Contacts:")
for key in contacts :
    print(key)
```

- The result of this code fragment is shown below:

```
My Contacts:
Sarah
Bob
John
Mary
Fred
```

Note that the dictionary stores its items in an order that is optimized for efficiency, which may not be the order in which they were added

Traversing a Dictionary: In Order

- To iterate through the keys in sorted order, you can use the `sorted()` function as part of the for loop :

```
print("My Contacts:")
for key in sorted(contacts) :
    print("%-10s %d" % (key, contacts[key]))
```

- Now, the contact list will be printed in order by name:

```
My Contacts:
Bob 3841212
Fred 7235591
John 4578102
Mary 3841212
Sarah 2213278
```

Iterating Dictionaries More Efficiently

- Python allows you to iterate over the items in a dictionary using the `items()` method
- This is a bit more efficient than iterating over the keys and then looking up the value of each key
- The `items()` method returns a sequence of tuples that contain the keys and values of all items
  - Here the loop variable `item` will be assigned a tuple that contains the key in the first slot and the value in the second slot

```
for item in contacts.items() :
    print(item[0], item[1])
```

Storing Data Records

- Data records, in which each record consists of multiple fields, are very common
- In some instances, the individual fields of the record were stored in a list to simplify the storage
- But this requires remembering in which element of the list each field is stored
  - This can introduce run-time errors into your program if you use the wrong list element when processing the record
- In Python, it is common to use a dictionary to store a data record

Dictionaries: Data Records

- You create an item for each data record in which the key is the field name and the value is the data value for that field
- For example, this dictionary named `record` stores a single student record with fields for ID, name, class, and GPA:

```
record = { "id": 100, "name": "Sally Roberts", "class": 2,
   "gpa": 3.78 }
```

Dictionaries: Data Records

- To extract records from a file, we can define a function that reads a single record and returns it as a dictionary
- The file to be read contains records made up of country names and population data separated by a colon:

```
def extractRecord(infile) :
    record = {}
    line = infile.readline()
    if line != "" :
        fields = line.split(":")
        record["country"] = fields[0]
        record["population"] = int(fields[1])
    return record
```

Dictionaries: Data Records

- The dictionary `record` that is returned has two items, one with the key "`country`" and the other with the key "`population`"
- This function's result can be used to print all of the records to the terminal

```
infile = open("populations.txt", "r")
record = extractRecord(infile)
while len(record) > 0 :
    print("%-20s %10d" % (record["country"],
       record["population"]))
    record = extractRecord(infile)
```

Common Dictionary Operations (1)

Table 2 Common Dictionary Operations

Operation	Returns
`d = dict()` `d = dict(c)`	Creates a new empty dictionary or a duplicate copy of dictionary *c*.
`d = {}` `d = {k1: v1, k2: v2, ..., kn: vn}`	Creates a new empty dictionary or a dictionary that contains the initial items provided. Each item consists of a key (*k*) and a value (*v*) separated by a colon.
`len(d)`	Returns the number of items in dictionary *d*.
key `in` *d* *key* `not in` *d*	Determines if the key is in the dictionary.
d`[`*key*`] = `*value*	Adds a new *key*/*value* item to the dictionary if the *key* does not exist. If the key does exist, it modifies the value associated with the key.
`x = `*d*`[`*key*`]`	Returns the value associated with the given key. The key must exist or an exception is raised.

Common Dictionary Operations (2)

Table 2 Common Dictionary Operations

d.get(*key*, *default*)	Returns the value associated with the given key, or the default value if the key is not present.
d.pop(*key*)	Removes the key and its associated value from the dictionary that contains the given key or raises an exception if the key is not present.
d.values()	Returns a sequence containing all values of the dictionary.

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

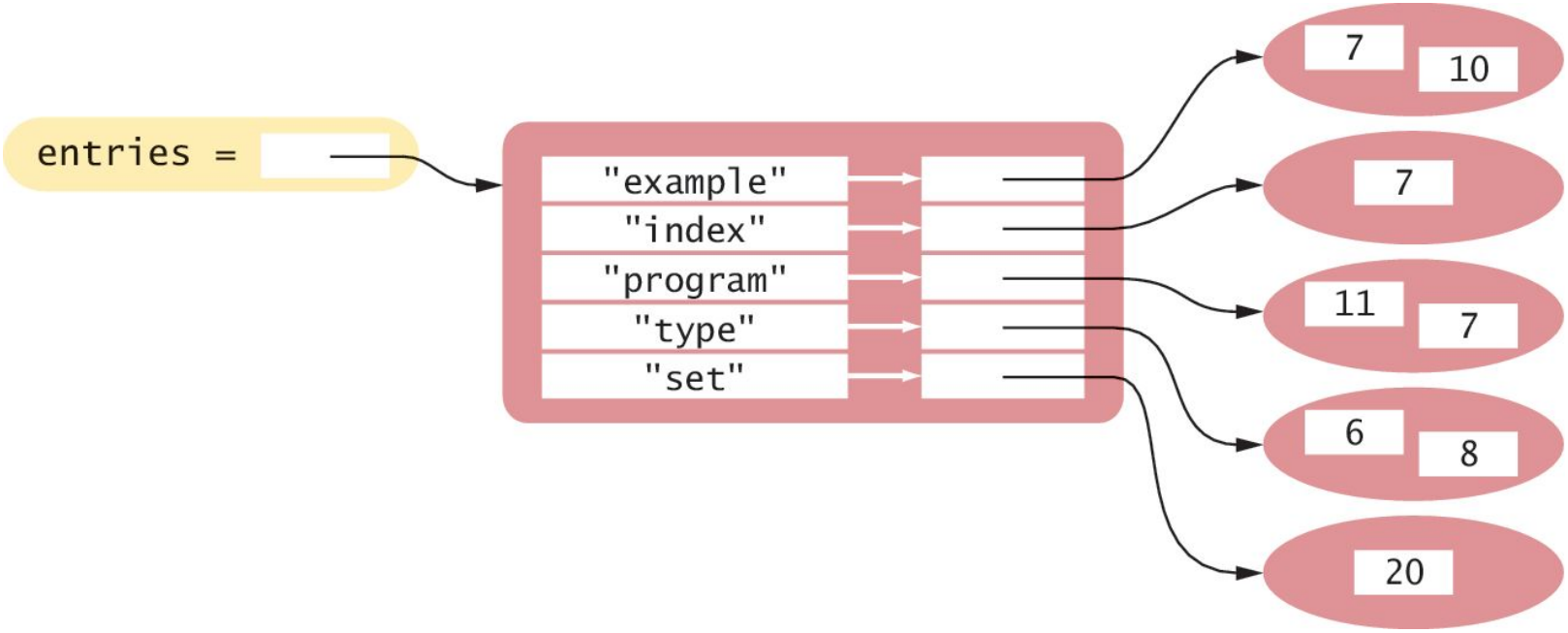
A Dictionary of Sets

- 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

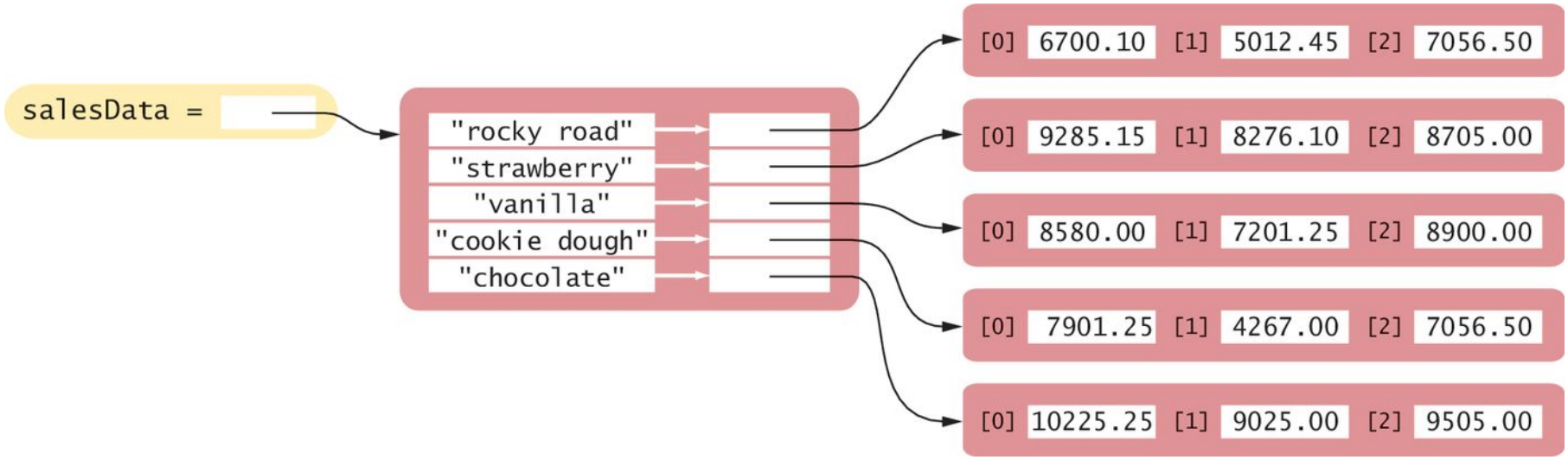
- The data is to be processed to produce a report similar to the following:

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