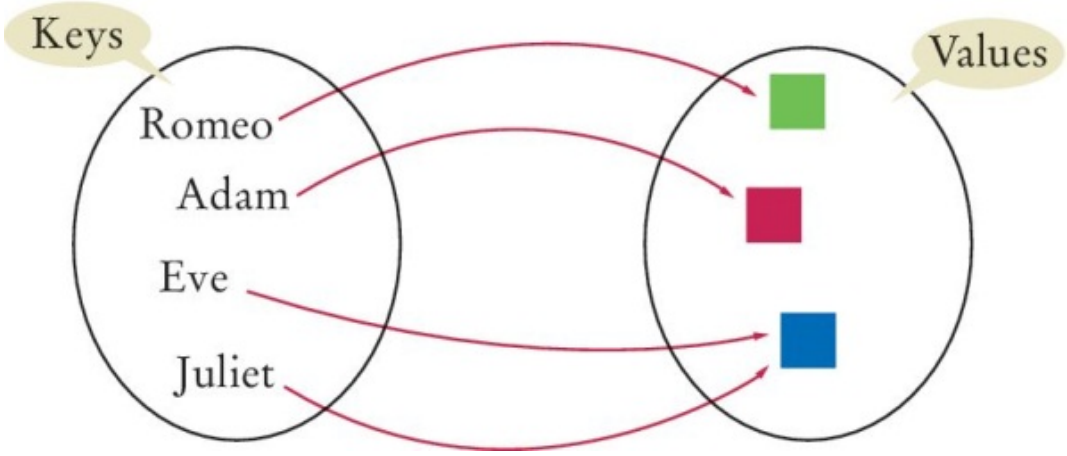
Dictionaries

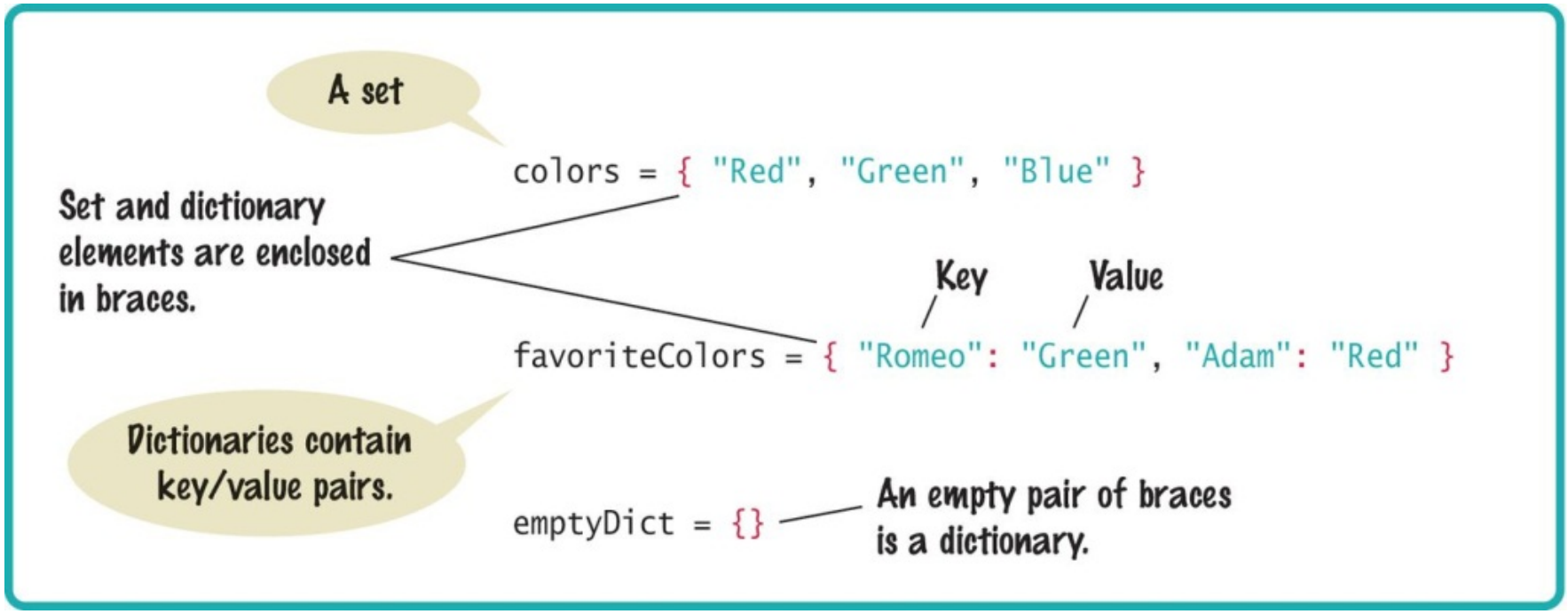
SECTION 8.2

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):



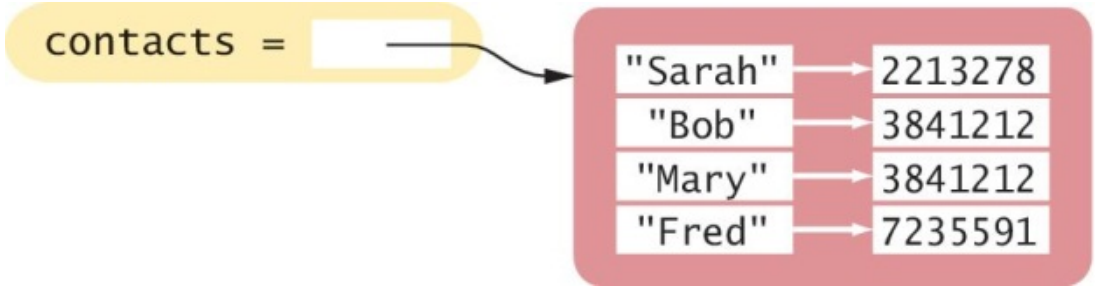
Syntax: Sets and Dictionaries



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)
```

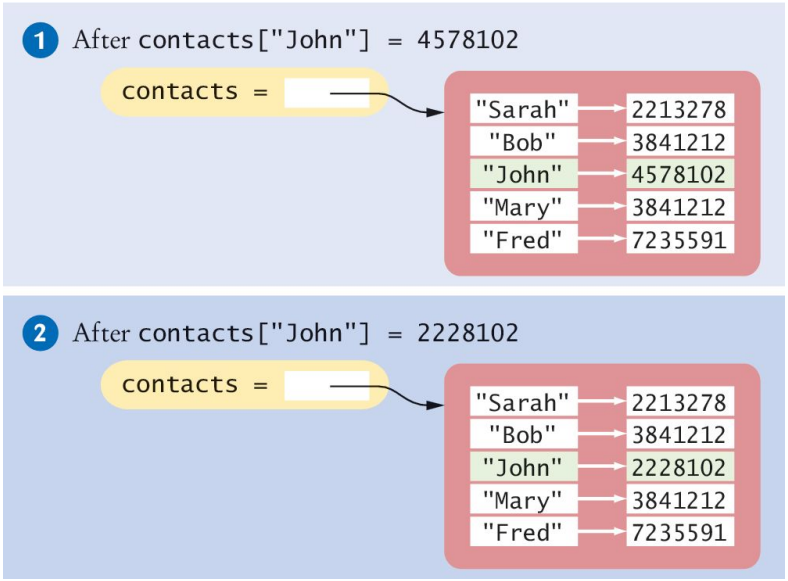
Adding/Modifying Items

- 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 = {}
```

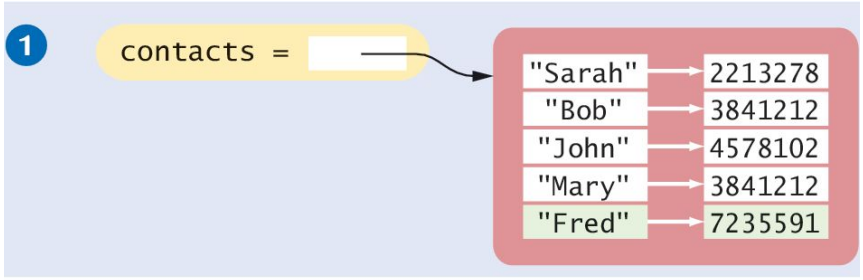
- and add new items as needed:

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

Removing Elements

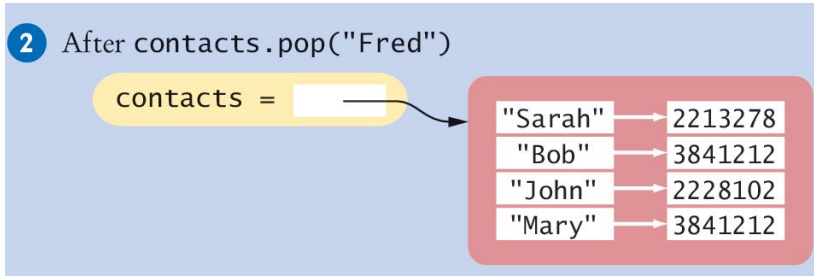
- To remove an item from a dictionary, call the `pop()` method with the key as the argument:

```
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.