PGR107
Python Programming

Lecture 7 – Sets & Dictionaries

Kristiania University College

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

Sets

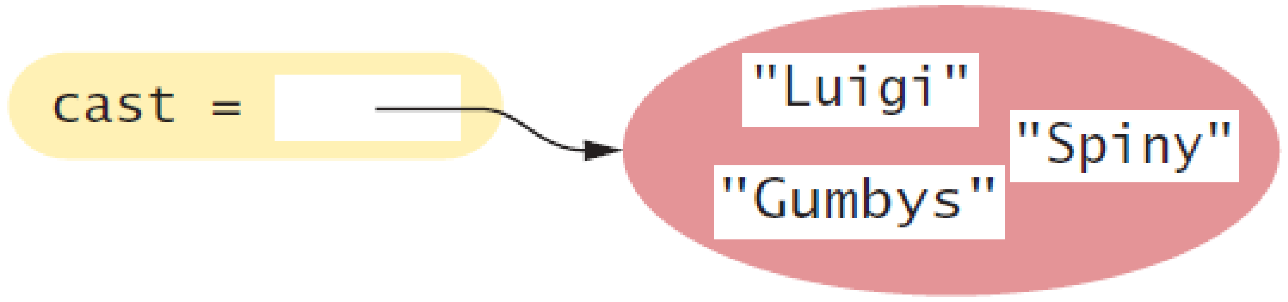
- A **set** is a container that stores a collection of **unique** values. Unlike a list, the elements or members of the set are not stored in any particular order and cannot be accessed by position.

A set stores
a collection of
unique values.

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" }
```



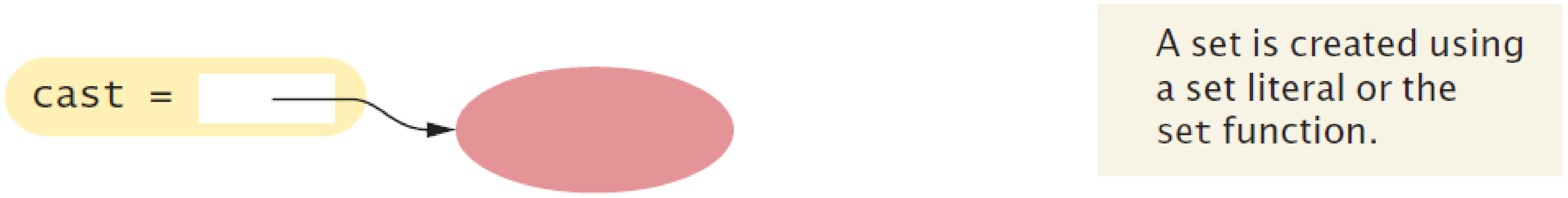
- Alternatively, you can use the **set function** to convert any sequence into a set:

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

Sets

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

```
cast = set()
```



Sets

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

```
numberOfCharacters = len(cast)
```

- 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.")
```

Sets

- Because sets are unordered, you cannot access the elements of a set by position as you can with a list. Instead, 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. For example, the loop above displays the following:

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

Sets

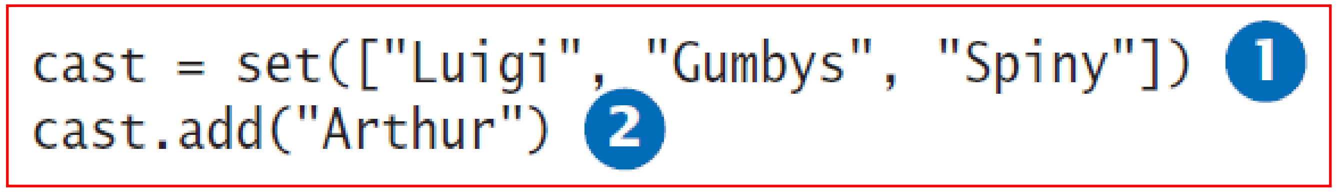
- The fact that sets do not retain the initial ordering is not a problem when working with sets. In fact, the lack of an ordering makes it possible to implement set operations very efficiently.

- However, you usually want to display the elements 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 character in sorted(cast) :
   print(character)
```

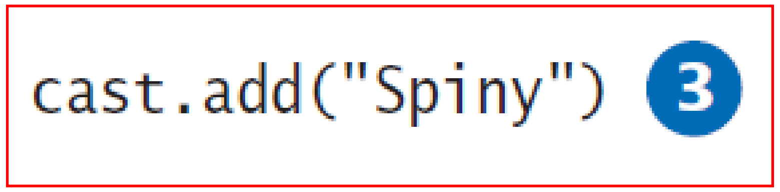
Sets

- **Adding Elements**: Like lists, sets are mutable collections, so you can add elements.
- For example, suppose we need to add more characters to the set cast created in the previous slides. Use the **add method** to add elements:



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

- If the element being added is not already contained in the set, it will be added to the set and the size of the set increased by one. Remember, however, that **a set cannot contain duplicate elements**. If you attempt to add an element that is already in the set, there is no effect, and the set is not changed.



```
cast.add("Spiny") 3
```

Sets

- **Removing Elements**: Like lists, sets are mutable collections, so you can remove elements. There are two methods that can be used to remove individual elements from a set.

✓ The **discard method** removes an element if the element exists

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

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

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

Sets

- **Removing Elements**:

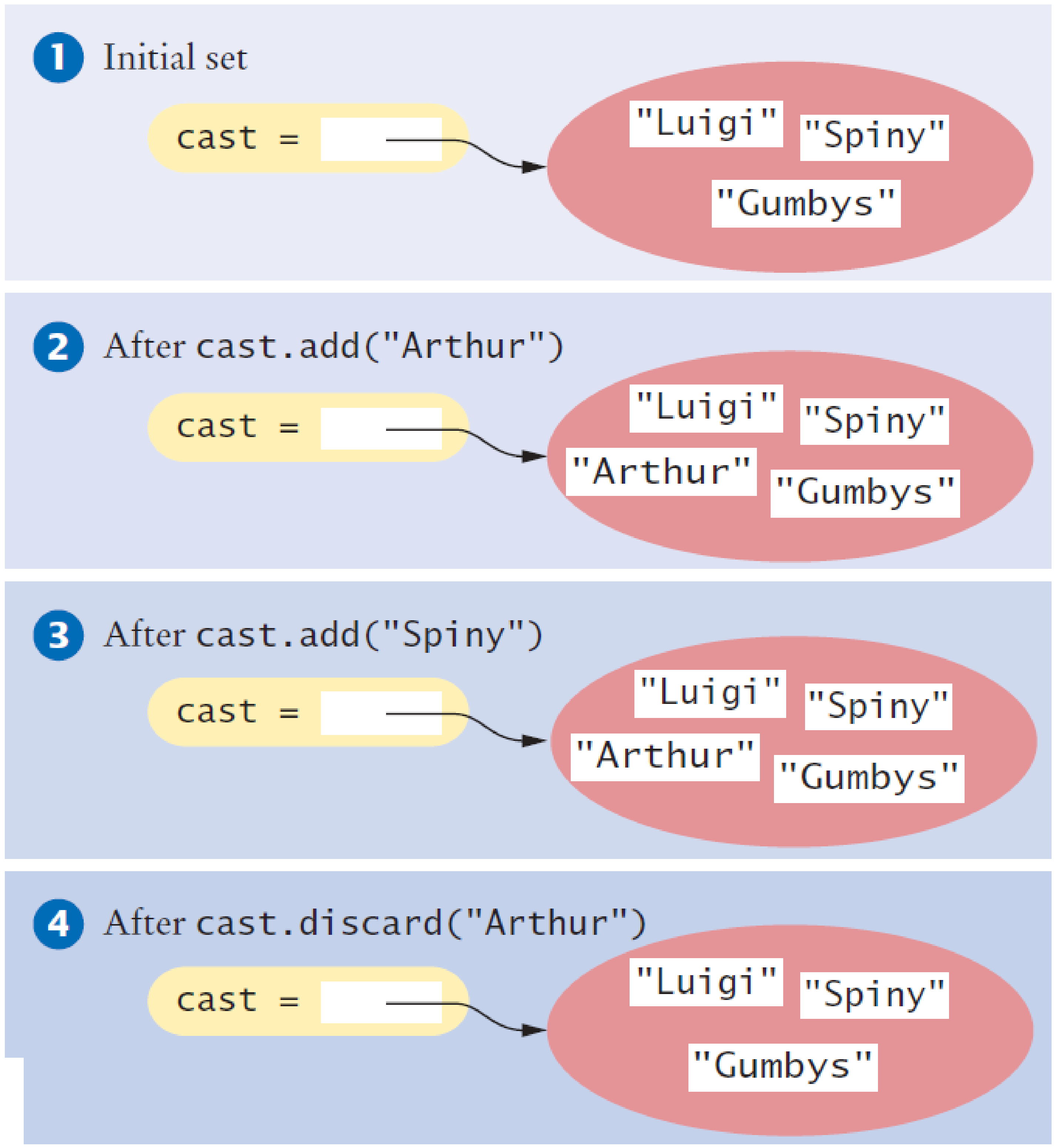
✓The **remove method**, on the other hand, removes an element if it exists, but raises an exception if the given element is not a member of the set:

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

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

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

Sets


1 Initial set
cast =
"Luigi"
"Spiny"
"Gumbys"
2 After cast.add("Arthur")
cast =
"Luigi"
"Spiny"
"Arthur"
"Gumbys"
3 After cast.add("Spiny")
cast =
"Luigi"
"Spiny"
"Arthur"
"Gumbys"
4 After cast.discard("Arthur")
cast =
"Luigi"
"Spiny"
"Gumbys"

Sets

- **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.

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

The issubset method tests whether one set is a subset of another set.

```
if canadian.issubset(british) :
   print("All Canadian flag colors occur in the British flag.")
if not italian.issubset(british) :
   print("At least one of the colors in the Italian flag does not.")
```

- You can also test for **equality** between two sets using the **==** and **!=** operators. Two sets are equal if and only if they have exactly the same elements.

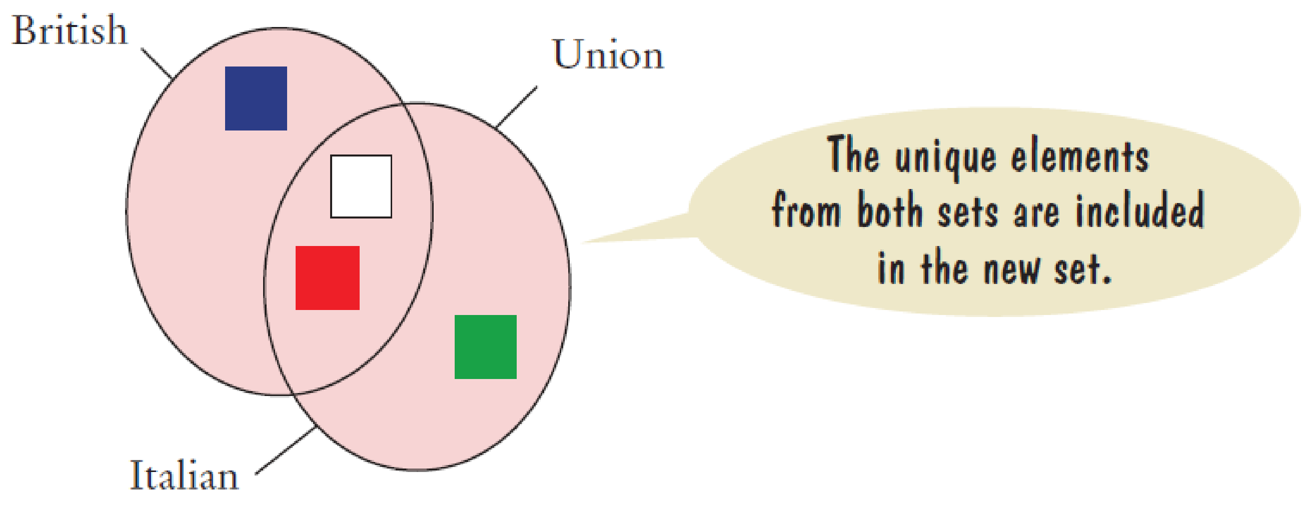
```
french = { "Red", "White", "Blue" }
if british == french :
   print("The British and French flags use the same colors.")
```

Sets

- **Set Union, Intersection, and Difference**:

- **Union**: The union of two sets contains all of the elements from both sets, with duplicates removed (use the **union method**).

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

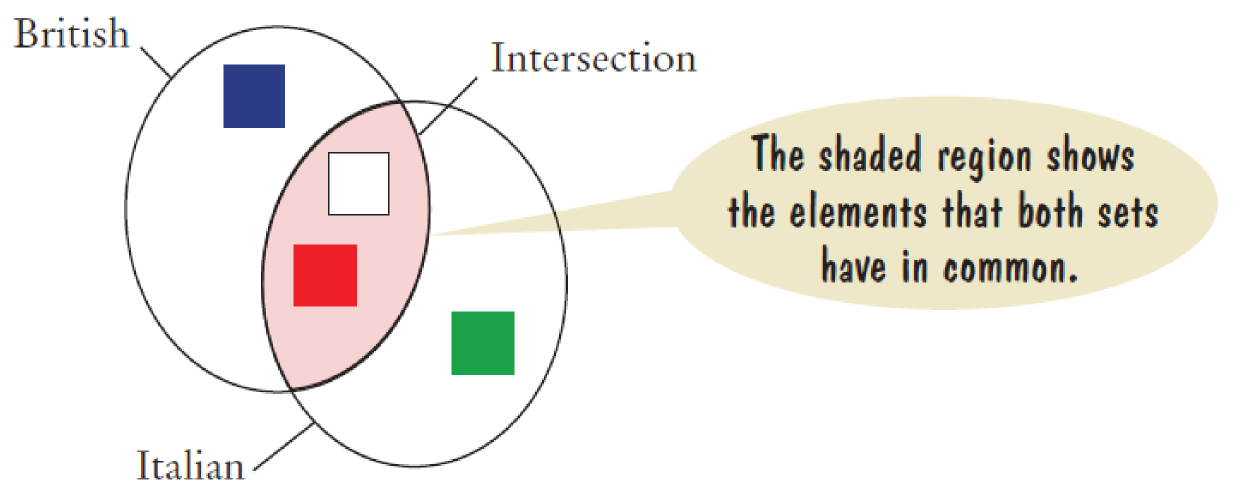

British
Union
Italian
The unique elements from both sets are included in the new set.

Sets

- **Set Union, Intersection, and Difference**:

- **Intersection**: The intersection of two sets contains all of the elements that are in both sets (use the **intersection method**).

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

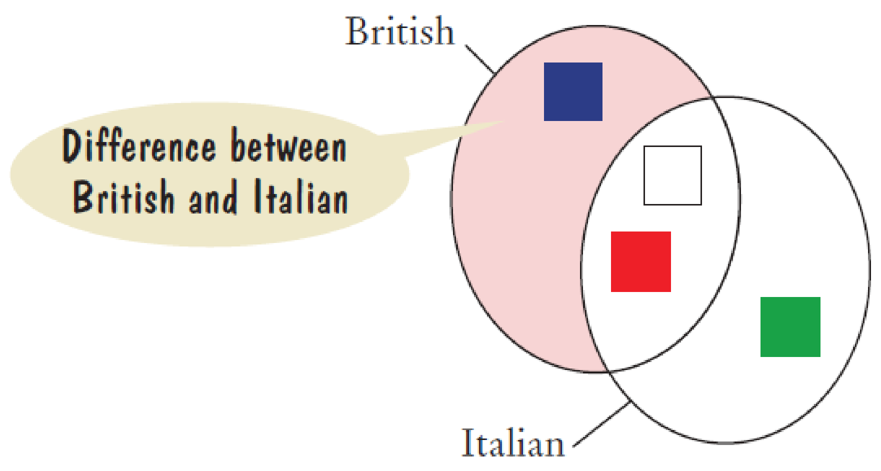

British
Intersection
Italian
The shaded region shows the elements that both sets have in common.

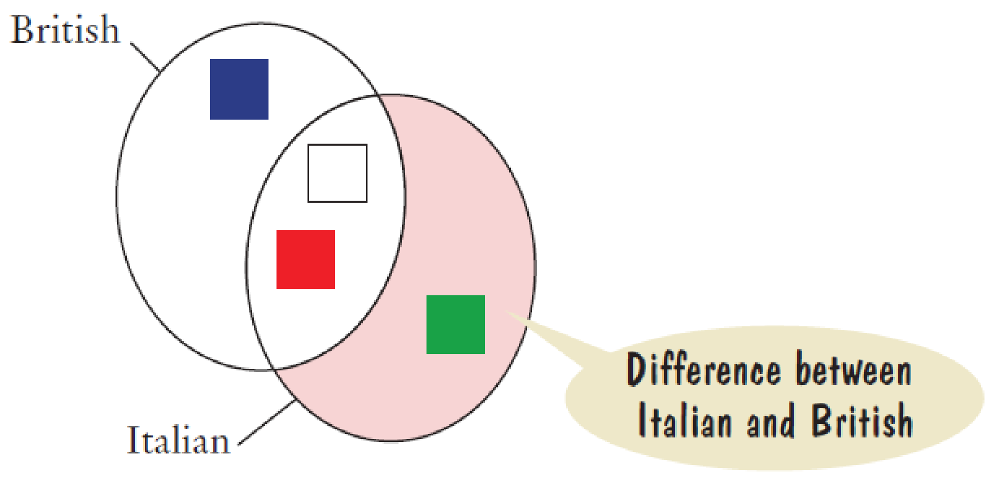
Sets

- **Set Union, Intersection, and Difference**:

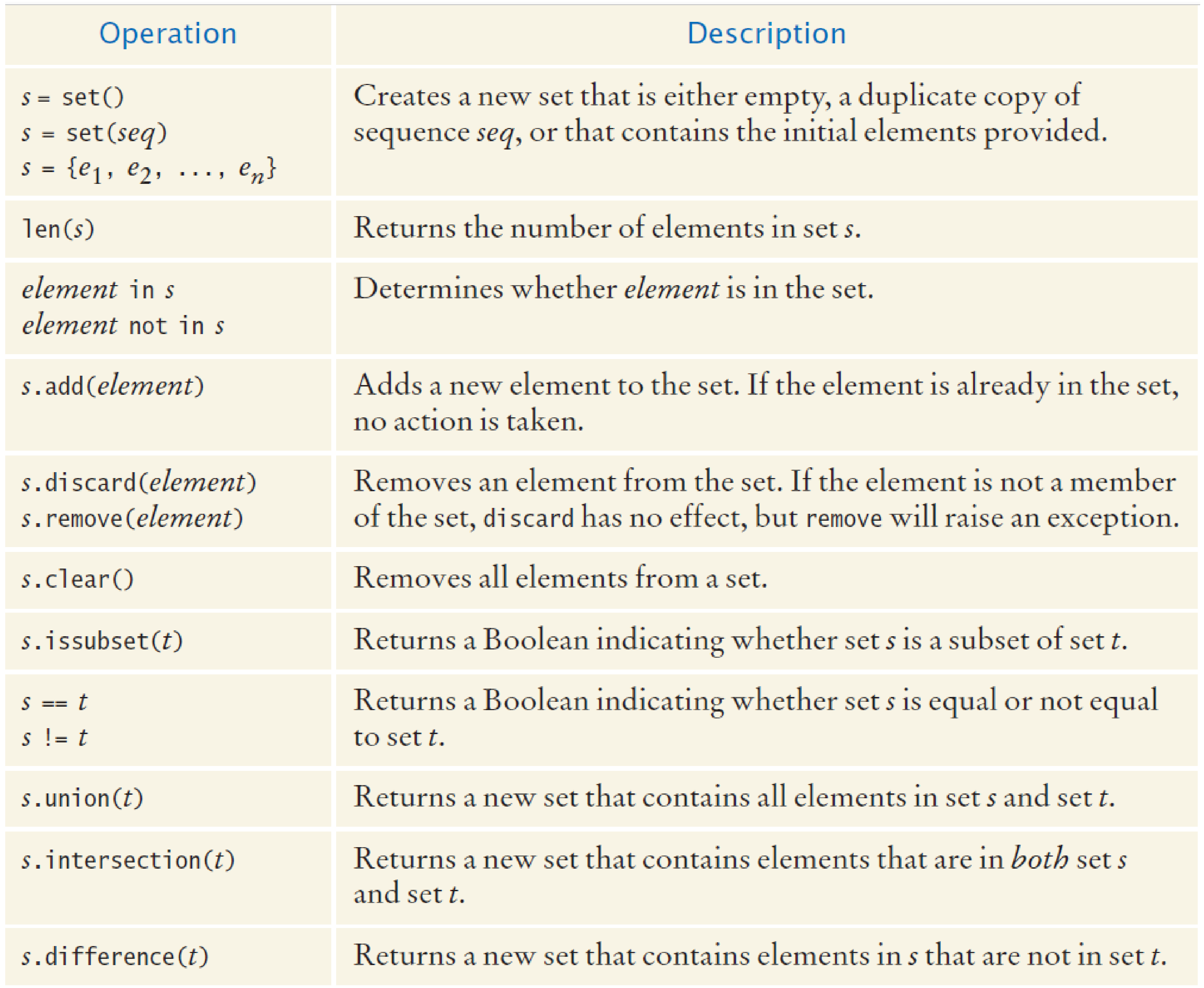
- **Difference**: The difference of two sets results in a new set that contains those elements in the first set that are not in the second set (use the **difference method**).

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





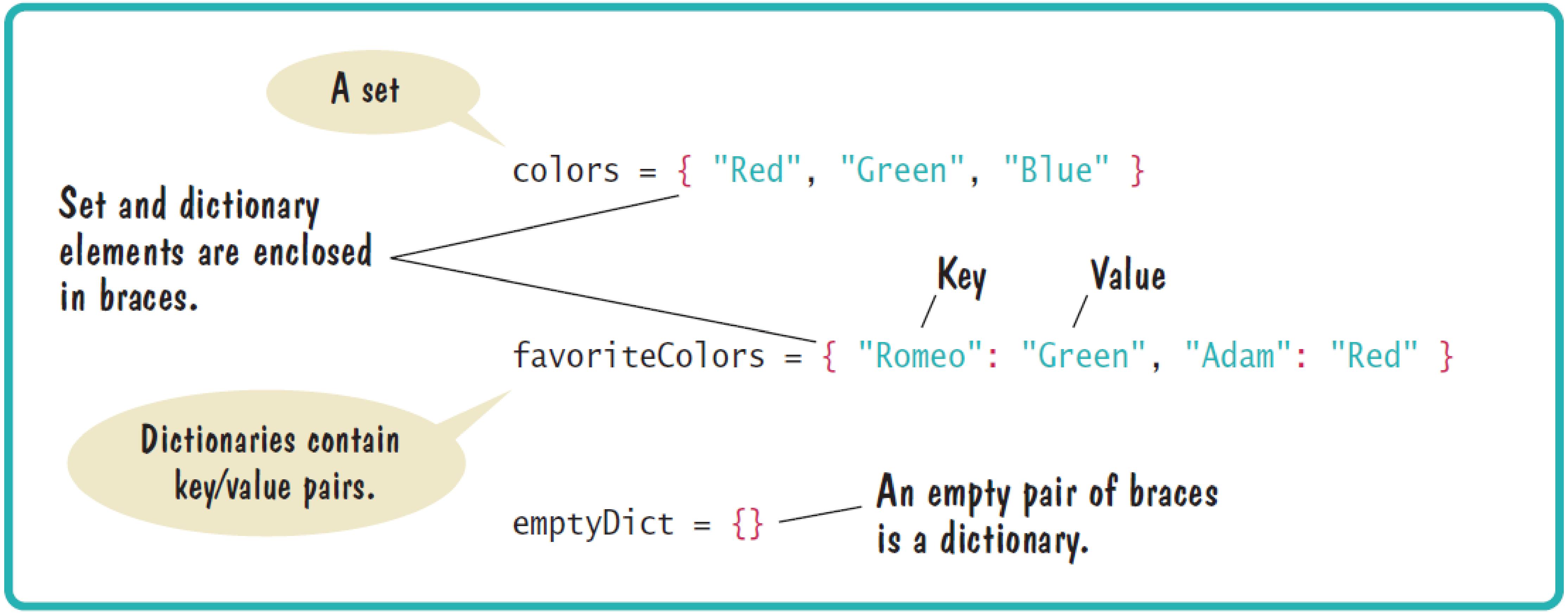
Common Set Operations



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 whether *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*.
s `==` *t* *s* `!=` *t*	Returns a Boolean indicating whether set *s* is equal or not 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*.

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.


A set
Set and dictionary elements are enclosed in braces.
Key
Value
Dictionaries contain key/value pairs.
An empty pair of braces is a dictionary.

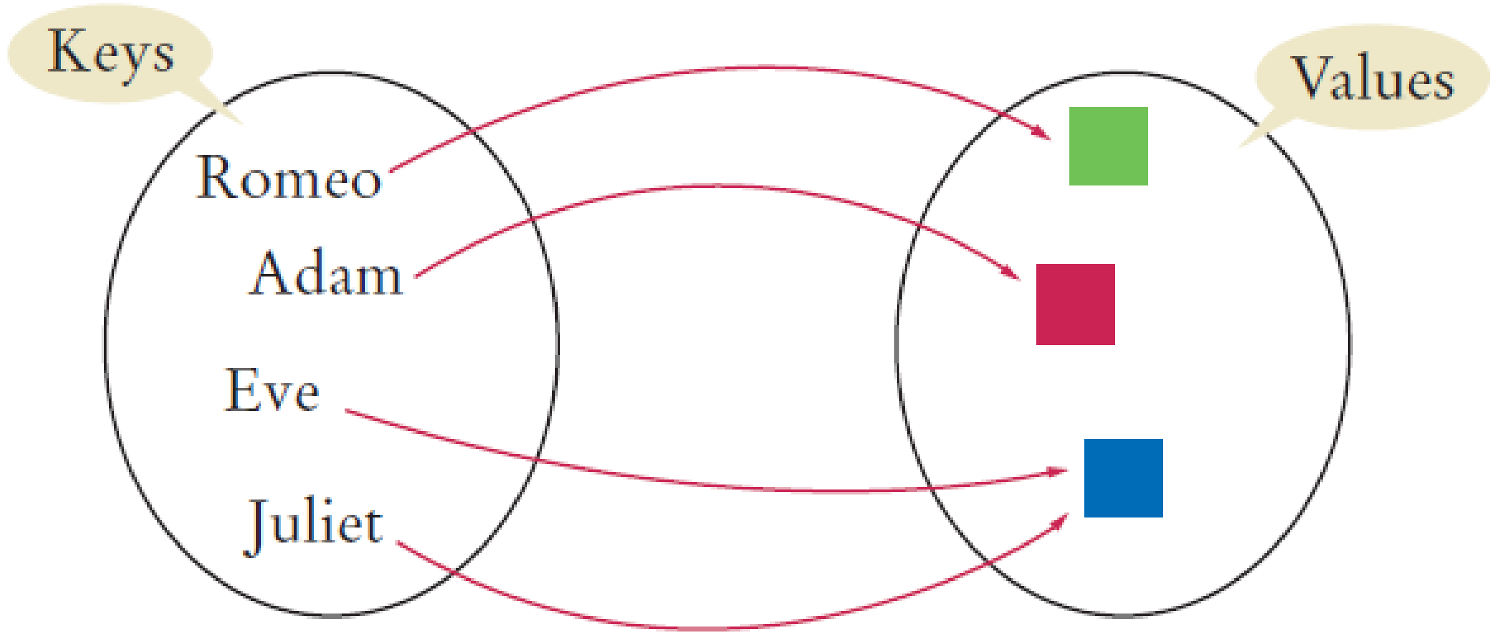
```
colors = { "Red", "Green", "Blue" }

favoriteColors = { "Romeo": "Green", "Adam": "Red" }

emptyDict = {}
```

Dictionaries

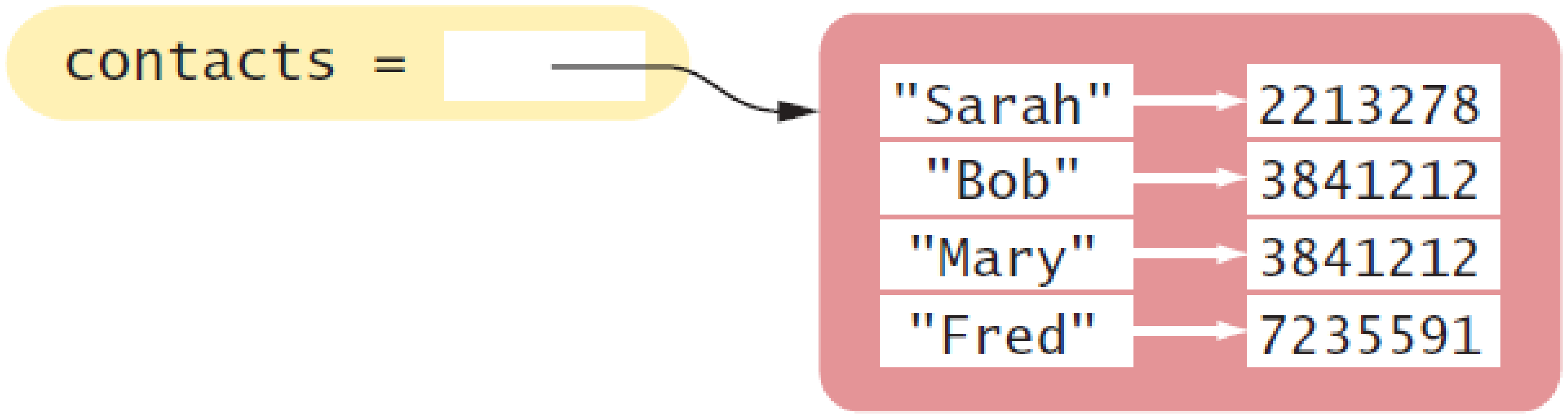
- The dictionary structure is also known as a **map** because it maps a unique key to a value. It stores the keys, values, and the associations between them.



Dictionaries

- **Creating Dictionaries:**

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



- You can create a duplicate copy of a dictionary using the **dict** function:

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

Dictionaries

- **Accessing Dictionary Values**: The **subscript operator []** is used to return the value associated with a key.

```
print("Fred's number is", contacts["Fred"])
```

- Note that the dictionary is not a sequence-type container like a list. Even though the subscript operator is used with a dictionary, you cannot access the items by index or position. A value can only be accessed using its associated key.

Use the [] operator to access the value associated with a key.

Dictionaries

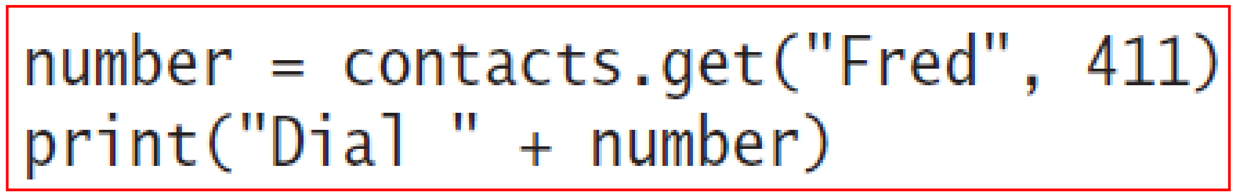
- **Accessing Dictionary Values**: The key supplied to the subscript operator must be a **valid key** in the dictionary or a **KeyError exception** will be raised. To find out whether a key is present in the dictionary, use the **in (or not in) operator**:

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

The in operator is used to test whether a key is in a dictionary.

Dictionaries

- **Accessing Dictionary Values**: Often, you want to use a default value if a key is not present.
- For example, if there is no number for Fred, you want to dial the directory assistance number instead. 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)
```

Dictionaries

- **Adding and Modifying Items**: A dictionary is a mutable container. That is, you can change its contents after it has been created. You can add a new item using the **subscript operator []** much as you would with a list.

```
contacts["John"] = 4578102 ❶
```

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

```
contacts["John"] = 2228102 ❷
```

Dictionaries

- **Adding and Modifying Items**: 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:

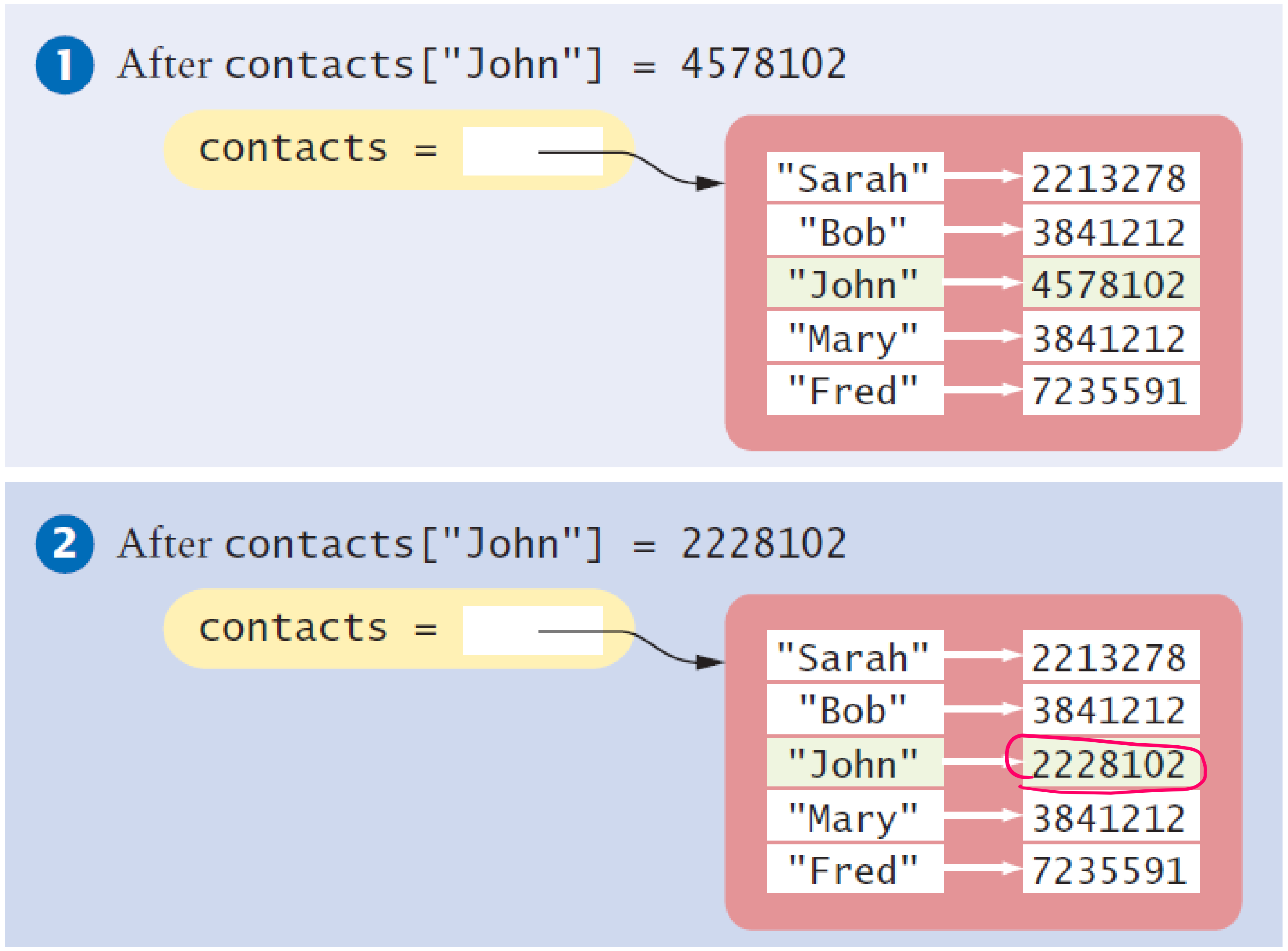
```
favoriteColors = {}
```

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

New items can be added or modified using the [] operator.

Dictionaries

- **Adding and Modifying Items:**



Dictionaries

- **Removing Items**: To remove an item from a dictionary, call the pop method with the key as the argument:

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

Use the pop method to remove a dictionary entry.

- This removes the entire item, both the key and its associated value. The pop method returns the value of the item being removed, so you can use it or store it in a variable:

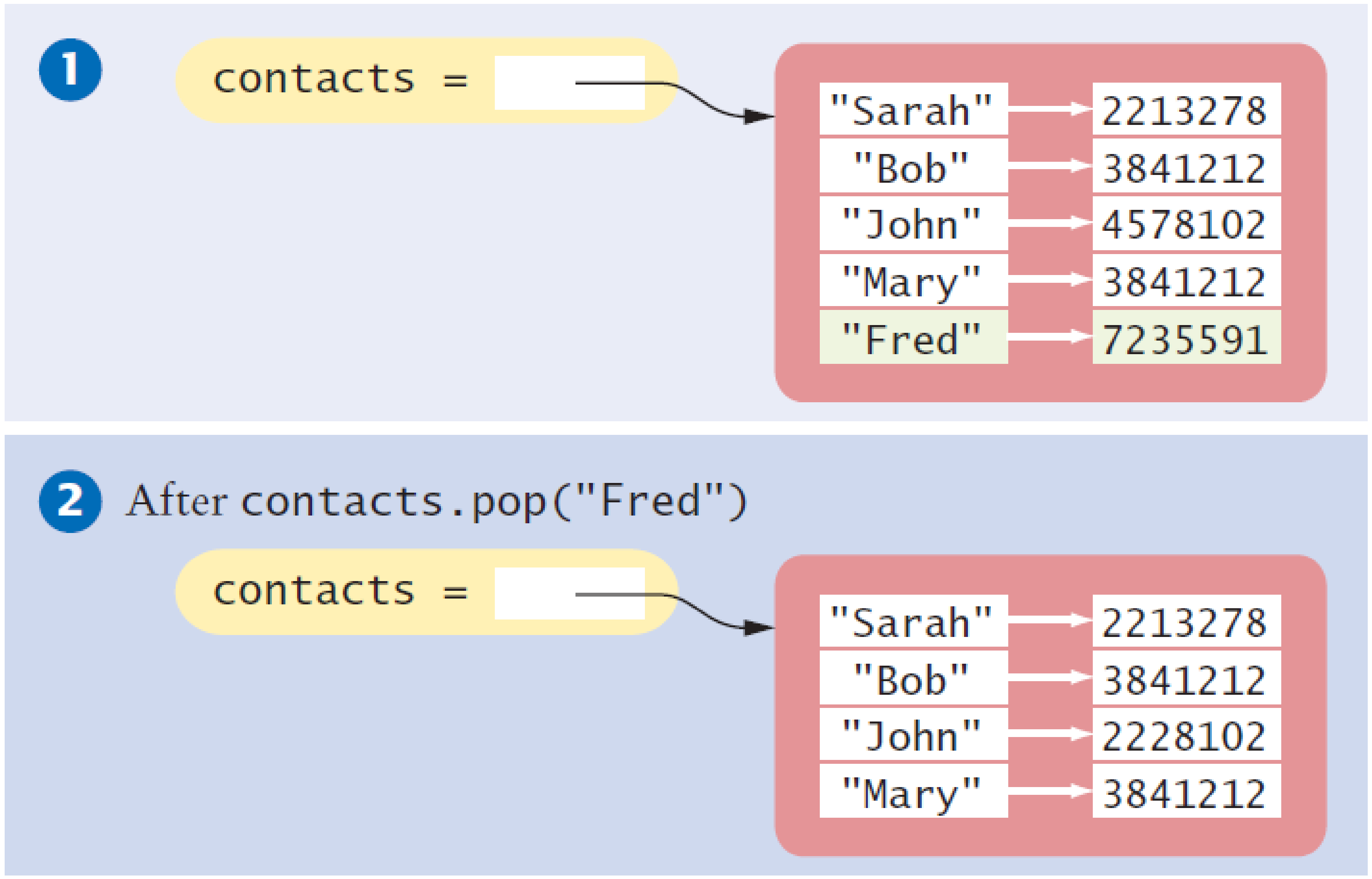
```
fredsNumber = contacts.pop("Fred")
```

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

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

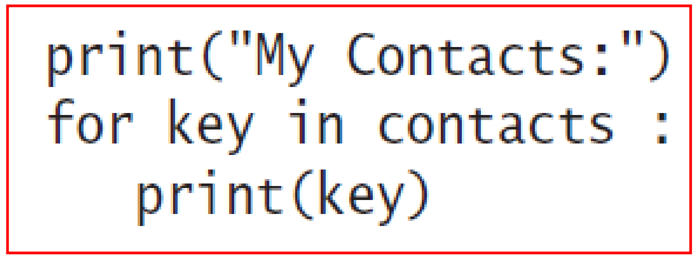
Dictionaries

- **Removing Items:**



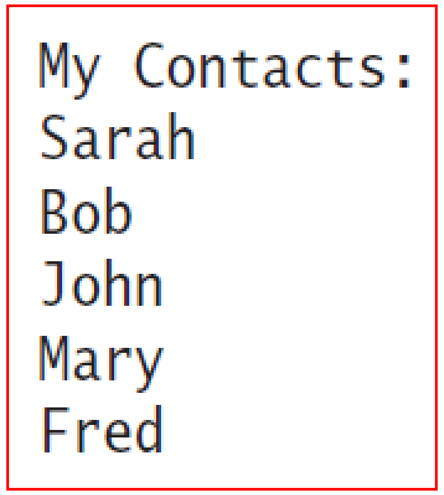
Dictionaries

- **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
```

Dictionaries

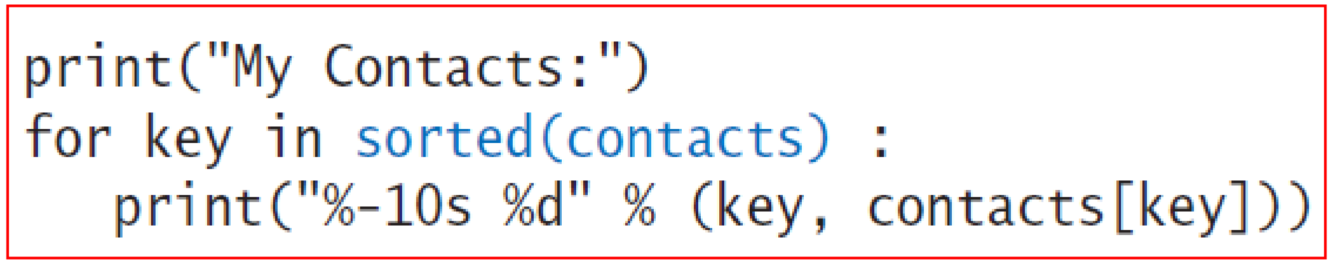
- To access the value associated with a key in the body of the loop, you can use the loop variable with the subscript operator. For example, these statements print both the name and phone number of your contacts:

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

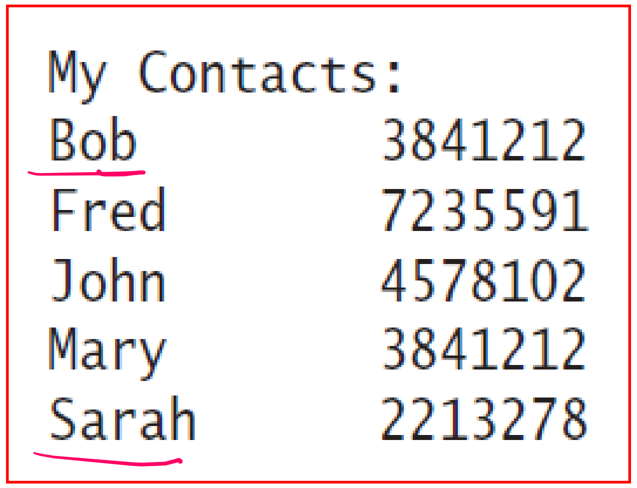
```
My Contacts:
Sarah      2213278
Bob        3841212
John       4578102
Mary       3841212
Fred       7235591
```

Dictionaries

- **Traversing a Dictionary**: The order in which the keys are visited is based on the order in which the items are stored internally. 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]))
```



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

Dictionaries

- **Traversing a Dictionary**: You can also iterate over the values of the items, instead of the keys, using the values method. This can be useful for creating a list that contains all of the phone numbers in our dictionary:

```
phoneNumbers = []    # Create an empty list.
for number in contacts.values() :
   phoneNumbers.append(number)
```

- As an alternative, you can pass the result of the values method to the list function to create the same list:

```
phoneNumbers = list(contacts.values())
```

Dictionaries

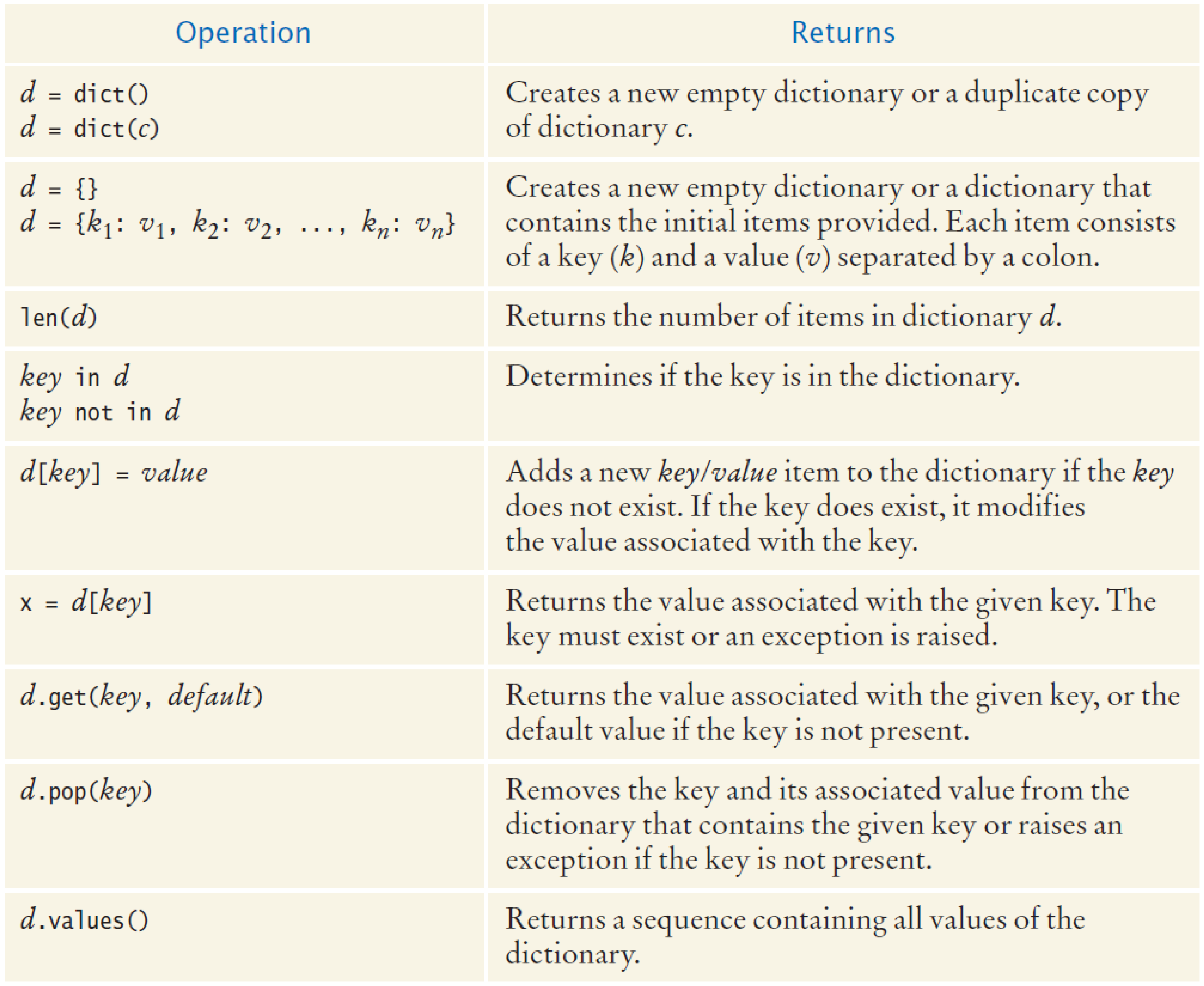
- **Iterating over Dictionary Items**: Python allows you to iterate over the items in a dictionary using the items method. This is a bit more efficient than iterating over the keys and then looking up the value of each key.
- The items method returns a sequence of tuples that contain the keys and values of all items.

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

- 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 (key, value) in contacts.items() :
   print(key, value)
```

Common Dictionary Operations



Operation	Returns
d = dict() *d* = dict(*c*)	Creates a new empty dictionary or a duplicate copy of dictionary *c*.

d = {}
d = {k_1: v_1, k_2: v_2, ..., k_n: v_n}
	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.
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.

End of Chapter 8