Python Programming (16ITE01)

UNIT - III

T. PRATHIMA, DEPT. OF IT, CBIT

Syllabus

UNIT-III

Lists and Tuples: Sequences, Introduction to Lists, List slicing, Finding Items in Lists with the in Operator, List Methods and Useful Built-in Functions, Copying Lists, Processing Lists, Two-Dimensional Lists, Tuples.

Strings: Basic String Operations, String Slicing, Testing, Searching, and Manipulating Strings.

Dictionaries and Sets: Dictionaries, Sets, Serializing Objects.

Recursion: Introduction, Problem Solving with Recursion, Examples of Recursive Algorithms.

Sequences

- A sequence is an object that holds multiple items of data, stored one after the other.
- You can perform operations on a sequence to examine and manipulate the items stored in it.
- A *sequence is an object that contains multiple items of data*.
- *The items that are in a sequence* are stored one after the other.
- Python provides various ways to perform operations on the items that are stored in a sequence.
- There are several different types of sequence objects in Python.
- Two of the fundamental sequence types are: lists and tuples.
- Both lists and tuples are sequences that can hold various types of data.
- The difference between lists and tuples is simple: a list is mutable, which means that a program can change its contents, but a tuple is immutable, which means that once it is created, its contents cannot be changed

Introduction to Lists

- A list is an object that contains multiple data items.
- Lists are mutable, which means that their contents can be changed during a program's execution.
- Lists are dynamic data structures, meaning that items may be added to them or removed from them.
- You can use indexing, slicing, and various methods to work with lists in a program.

List Contd..

- A *list is an object that contains multiple data items.*
- *Each item that is stored in a list is called* an *element.*
- *Here is a statement that creates a list of integers:*
- even_numbers = [2, 4, 6, 8, 10]
- The items that are enclosed in brackets and separated by commas are the list elements.

List Contd..

The following is another example:

```
names = ['Molly', 'Steven', 'Will', 'Alicia', 'Adriana']
```

This statement creates a list of five strings. After the statement executes, the `name` variable will reference the list as shown in Figure 7-2.

7-2 A list of strings

A list can hold items of different types, as shown in the following example:

```
info = ['Alicia', 27, 1550.87]
```

This statement creates a list containing a string, an integer, and a floating-point number.

List with different data types

A list holding different types

List()

- Python also has a built-in list() function that can convert certain types of objects to lists.
- range function returns an iterable, which is an object that holds a series of values that can be iterated over.
- You can use a statement such as the following to convert the range function's iterable object to a list:
- numbers = list(range(5))

List()

- When this statement executes, the following things happen:
- The range function is called with 5 passed as an argument.
- The function returns an iterable containing the values 0, 1, 2, 3, 4.
- The iterable is passed as an argument to the list() function.
- The list() function returns the list [0, 1, 2, 3, 4].
- The list [0, 1, 2, 3, 4] is assigned to the numbers variable.
- Here is another example:

```
numbers = list(range(1, 10, 2))
```

- This statement will assign the list [1, 3, 5, 7, 9] to the numbers variable.

The Repetition Operator

- The * symbol multiplies two numbers.
- However, when the operand on the left side of the * symbol is a sequence (such as a list) and the operand on the right side is an integer, it becomes the *repetition operator*.
- *The repetition operator makes* multiple copies of a list and joins them all together.
- Here is the general format:

  *list * n*

Examples

```
1   >>> numbers = [0] * 5 Enter
2   >>> print(numbers) Enter
3   [0, 0, 0, 0, 0]
4   >>>
```

Let's take a closer look at each statement:

- In line 1 the expression `[0] * 5` makes five copies of the list `[0]` and joins them all together in a single list. The resulting list is assigned to the `numbers` variable.
- In line 2 the `numbers` variable is passed to the `print` function. The function's output is shown in line 3.

Here is another interactive mode demonstration:

```
1   >>> numbers = [1, 2, 3] * 3 Enter
2   >>> print(numbers) Enter
3   [1, 2, 3, 1, 2, 3, 1, 2, 3]
4   >>>
```

NOTE: Most programming languages allow you to create sequence structures known as *arrays*, which are similar to lists, but are much more limited in their capabilities. You cannot create traditional arrays in Python because lists serve the same purpose and provide many more built-in capabilities.

Iterating over a List with the for Loop

- You can iterate over a list with the for loop, as shown here

```
numbers = [99, 100, 101, 102]
for n in numbers:
    print(n)
```

If we run this code, it will print:

```
99
100
101
102
```

Indexing

- Another way that you can access the individual elements in a list is with an *index*.
- *Each* element in a list has an index that specifies its position in the list.
- Indexing starts at 0, so the index of the first element is 0, the index of the second element is 1, and so forth.
- The index of the last element in a list is 1 less than the number of elements in the list.
- For example, the following statement creates a list with 4 elements:

```
my_list = [10, 20, 30, 40]
```

Indexing contd..

- The indexes of the elements in this list are 0, 1, 2, and 3.
- We can print the elements of the list with the following statement:
- print(my_list[0], my_list[1], my_list[2], my_list[3])
- The following loop also prints the elements of the list:
- index = 0
- while index < 4:

```
        print(my_list[index])
        index += 1
```

Negative indices

- You can also use negative indexes with lists to identify element positions relative to the end of the list.
- The Python interpreter adds negative indexes to the length of the list to determine the element position.
- The index 21 identifies the last element in a list, 22 identifies the next to last element, and so forth.
- The following code shows an example:
- my_list = [10, 20, 30, 40]
- print(my_list[−1], my_list[−2], my_list[−3], my_list[−4])
- In this example, the print function will display:
- 40 30 20 10

IndexError exception

- An IndexError exception will be raised if you use an invalid index with a list.
- For example, look at the following code:
- # This code will cause an IndexError exception.
- my_list = [10, 20, 30, 40]
- index = 0
- while index < 5:

  print(my_list[index])

  index += 1
- The last time that this loop begins an iteration, the index variable will be assigned the value 4, which is an invalid index for the list.
- As a result, the statement that calls the print function will cause an IndexError exception to be raised.

The len()

- Python has a built-in function named len that returns the length of a sequence, such as a list.
- The following code demonstrates:
- my_list = [10, 20, 30, 40]
- size = len(my_list)
- The first statement assigns the list [10, 20, 30, 40] to the my_list variable.
- The second statement calls the len function, passing the my_list variable as an argument.
- The function returns the value 4, which is the number of elements in the list.
- This value is assigned to the size variable.

Len()...

- The len function can be used to prevent an IndexError exception when iterating over a list with a loop.
- Here is an example:
- my_list = [10, 20, 30, 40]
- index = 0
- while index < len(my_list):

```
        print(my_list[index])
        index += 1
```

Lists Are Mutable

- Lists in Python are *mutable, which means their elements can be changed.*
- *Consequently, an* expression in the form *list[index] can appear on the left side of an assignment operator.*
- The following code shows an example:

```
1 numbers = [1, 2, 3, 4, 5]
2 print(numbers)
3 numbers[0] = 99
4 print(numbers)
```

- The statement in line 2 will display

```
[1, 2, 3, 4, 5]
```

- The statement in line 3 assigns 99 to numbers[0]. This changes the first value in the list to 99.
- When the statement in line 4 executes, it will display

```
[99, 2, 3, 4, 5]
```

Example…

- look at the following code:
- numbers = [1, 2, 3, 4, 5] # Create a list with 5 elements.
- numbers[5] = 99 # This raises an exception!
- The numbers list that is created in the first statement has five elements, with the indexes 0 through 4.
- The second statement will raise an IndexError exception because the numbers list has no element at index 5.

Example

```
1  # The NUM_DAYS constant holds the number of
2  # days that we will gather sales data for.
3  NUM_DAYS = 5
4
5  def main():
6      # Create a list to hold the sales
7      # for each day.
8      sales = [0] * NUM_DAYS
9
10     # Create a variable to hold an index.
11     index = 0
12
13     print('Enter the sales for each day.')
14
15     # Get the sales for each day.
16     while index < NUM_DAYS:
17         print('Day #', index + 1, ': ', sep='', end='')
18         sales[index] = float(input())
19         index += 1
```

```
21      # Display the values entered.
22      print('Here are the values you entered:')
23      for value in sales:
24          print(value)
25
26  # Call the main function.
27  main()
```

Program Output (with input shown in bold)

```
Enter the sales for each day.
Day #1: 1000 Enter
Day #2: 2000 Enter
Day #3: 3000 Enter
Day #4: 4000 Enter
Day #5: 5000 Enter
Here are the values you entered:
1000.0
2000.0
3000.0
4000.0
5000.0
```

Concatenating Lists

- To concatenate means to join two things together. You can use the + operator to concatenate two lists.
- Here is an example:

```
list1 = [1, 2, 3, 4]
list2 = [5, 6, 7, 8]
list3 = list1 + list2
```

- After this code executes, list1 and list2 remain unchanged, and list3 references the following list:

```
[1, 2, 3, 4, 5, 6, 7, 8]
```

Examples

You can also use the += augmented assignment operator to concatenate one list to another. Here is an example:

```
list1 = [1, 2, 3, 4]
list2 = [5, 6, 7, 8]
list1 += list2
```

The last statement appends `list2` to `list1`. After this code executes, `list2` remains unchanged, but `list1` references the following list:

```
[1, 2, 3, 4, 5, 6, 7, 8]
```

The following interactive mode session also demonstrates the += operator used for list concatenation:

```
>>> girl_names = ['Joanne', 'Karen', 'Lori'] Enter
>>> girl_names += ['Jenny', 'Kelly'] Enter
>>> print(girl_names) Enter
['Joanne', 'Karen', 'Lori', 'Jenny', 'Kelly']
>>>
```

NOTE: Keep in mind that you can concatenate lists only with other lists. If you try to concatenate a list with something that is not a list, an exception will be raised.

List Slicing

- A slicing expression selects a range of elements from a sequence
- In Python, you can write expressions that select subsections of a sequence, known as slices.
- A *slice is a span of items that are taken from a sequence.*
- *When you take a slice from a list,* you get a span of elements from within the list.
- To get a slice of a list, you write an expression in the following general format:

  list_name[start : end]

- In the general format, *start is the index of the first element in the slice, and end is the* index marking the end of the slice.
- The expression returns a list containing a copy of the elements from *start up to (but not including) end.*

Example

- days = ['Sunday', 'Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday', 'Saturday']
- The following statement uses a slicing expression to get the elements from indexes 2 up to, but not including, 5:

```
mid_days = days[2:5]
```

- After this statement executes, the mid_days variable references the following list:

```
['Tuesday', 'Wednesday', 'Thursday']
```

Example

1. numbers = [1, 2, 3, 4, 5]
2. print(numbers[1:3])
3. *[2, 3]*

- Here is a summary of each line:
- In line 1 we created the list and [1, 2, 3, 4, 5] and assigned it to the numbers variable.
- In line 2 we sent the slice numbers[1:3] as an argument to the print function.
- The print function displayed the slice in line 3.

Slicing … Starting index

- If you leave out the *start index in a slicing expression, Python uses 0 as the starting index.*

1. *numbers = [1, 2, 3, 4, 5]*
2. *print(numbers)*
3. *[1, 2, 3, 4, 5]*
4. *print(numbers[:3])*
5. *[1, 2, 3]*

- Notice that line 4 sends the slice numbers[:3] as an argument to the print function.
- Because the starting index was omitted, the slice contains the elements from index 0 up to 3.

Slicing … Ending index

- If you leave out the *end index in a slicing expression, Python uses the length of the list as* the *end index.*

1. *numbers = [1, 2, 3, 4, 5]*
2. *print(numbers) e*
3. *[1, 2, 3, 4, 5]*
4. *print(numbers[2:])*
5. *[3, 4, 5]*

- Notice that line 4 sends the slice numbers[2:] as an argument to the print function.
- Because the ending index was omitted, the slice contains the elements from index 2 through the end of the list.
- If you leave out both the start and end index in a slicing expression, you get a copy of the entire list.

1. *numbers = [1, 2, 3, 4, 5]*
2. *print(numbers)*
3. *[1, 2, 3, 4, 5]*
4. *print(numbers[:])*
5. *[1, 2, 3, 4, 5]*

Slicing ...step value

```
1   >>> numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10] Enter
2   >>> print(numbers) Enter
3   [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
4   >>> print(numbers[1:8:2]) Enter
5   [2, 4, 6, 8]
6   >>>
```

In the slicing expression in line 4, the third number inside the brackets is the step value. A step value of 2, as used in this example, causes the slice to contain every second element from the specified range in the list.

You can also use negative numbers as indexes in slicing expressions to reference positions relative to the end of the list. Python adds a negative index to the length of a list to get the position referenced by that index. The following interactive mode session shows an example:

```
1   >>> numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10] Enter
2   >>> print(numbers) Enter
3   [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
4   >>> print(numbers[-5:]) Enter
5   [6, 7, 8, 9, 10]
```

- Slicing expressions can also have step value, which can cause elements to be skipped in the list.

Invalid indices

- Invalid indexes do not cause slicing expressions to raise an exception. For example:
- If the *end index specifies a position beyond the end of the list, Python will use the* length of the list instead.
- If the *start index specifies a position before the beginning of the list, Python will* use 0 instead.
- If the *start index is greater than the end index, the slicing expression will return* an empty list.

Finding Items in Lists with the in Operator

- You can search for an item in a list using the **in operator**.
- In Python you can use the in operator to determine whether an item is contained in a list.
- Here is the general format of an expression written with the in operator to search for an item in a list:

  item in list

- In the general format, *item is the item for which you are searching, and list is a list.*
- *The* expression returns true if *item is found in the list or false otherwise*

```
1  # This program demonstrates the in operator
2  # used with a list.
3
4  def main():
5      # Create a list of product numbers.
6      prod_nums = ['V475', 'F987', 'Q143', 'R688']
7
8      # Get a product number to search for.
9      search = input('Enter a product number: ')
10
11     # Determine whether the product number is in the list.
12     if search in prod_nums:
13         print(search, 'was found in the list.')
14     else:
15         print(search, 'was not found in the list.')
16
17 # Call the main function.
18 main()
```

Output

Program Output (with input shown in bold)

```
Enter a product number: Q143 Enter
Q143 was found in the list.
```

Program Output (with input shown in bold)

```
Enter a product number: B000 Enter
B000 was not found in the list.
```

The program gets a product number from the user in line 9 and assigns it to the `search` variable. The `if` statement in line 12 determines whether `search` is in the `prod_nums` list.

You can use the `not in` operator to determine whether an item is *not* in a list. Here is an example:

```
if search not in prod_nums:
    print(search, 'was not found in the list.')
else:
    print(search, 'was found in the list.')
```

List Methods and Useful Built-in Functions

Table 7-1 A few of the list methods

Method	Description
`append(item)`	Adds *item* to the end of the list.
`index(item)`	Returns the index of the first element whose value is equal to item. A `ValueError` exception is raised if item is not found in the list.
`insert(index, item)`	Inserts *item* into the list at the specified *index*. When an item is inserted into a list, the list is expanded in size to accommodate the new item. The item that was previously at the specified index, and all the items after it, are shifted by one position toward the end of the list. No exceptions will occur if you specify an invalid index. If you specify an index beyond the end of the list, the item will be added to the end of the list. If you use a negative index that specifies an invalid position, the item will be inserted at the beginning of the list.
`sort()`	Sorts the items in the list so they appear in ascending order (from the lowest value to the highest value).
`remove(item)`	Removes the first occurrence of *item* from the list. A `ValueError` exception is raised if item is not found in the list.
`reverse()`	Reverses the order of the items in the list.

Append()

```
1  # This program demonstrates how the append
2  # method can be used to add items to a list.
3
4  def main():
5      # First, create an empty list.
6      name_list = []
7
8      # Create a variable to control the loop.
9      again = 'y'
10
11     # Add some names to the list.
12     while again == 'y':
13         # Get a name from the user.
14         name = input('Enter a name: ')
15
16         # Append the name to the list.
17         name_list.append(name)
18
19         # Add another one?
20         print('Do you want to add another name?')
21         again = input('y = yes, anything else = no: ')
22         print()
```

```
24      # Display the names that were entered.
25      print('Here are the names you entered.')
26
27      for name in name_list:
28          print(name)
29
30  # Call the main function.
31  main()
```

Program Output (with input shown in bold)

```
Enter a name: Kathryn [Enter]
Do you want to add another name?
y = yes, anything else = no: y [Enter]

Enter a name: Chris [Enter]
Do you want to add another name?
y = yes, anything else = no: y [Enter]

Enter a name: Kenny [Enter]
Do you want to add another name?
y = yes, anything else = no: y [Enter]

Enter a name: Renee [Enter]
Do you want to add another name?
y = yes, anything else = no: n [Enter]

Here are the names you entered.
Kathryn
Chris
Kenny
Renee
```

index()

```
1   # This program demonstrates how to get the
2   # index of an item in a list and then replace
3   # that item with a new item.
4
5   def main():
6       # Create a list with some items.
7       food = ['Pizza', 'Burgers', 'Chips']
8
9       # Display the list.
10      print('Here are the items in the food list:')
11      print(food)
12
13      # Get the item to change.
14      item = input('Which item should I change? ')
15
16      try:
17          # Get the item's index in the list.
18          item_index = food.index(item)
19
20          # Get the value to replace it with.
21          new_item = input('Enter the new value: ')
22
23          # Replace the old item with the new item.
24          food[item_index] = new_item
25
26          # Display the list.
27          print('Here is the revised list:')
28          print(food)
29      except ValueError:
30          print('That item was not found in the list.')
31
32  # Call the main function.
33  main()
```

Program Output (with input shown in bold)

```
Here are the items in the food list:
['Pizza', 'Burgers', 'Chips']
Which item should I change? Burgers [Enter]
Enter the new value: Pickles [Enter]
Here is the revised list:
['Pizza', 'Pickles', 'Chips']
```

Insert()

```
1  # This program demonstrates the insert method
2
3  def main():
4      # Create a list with some names.
5      names = ['James', 'Kathryn', 'Bill']
6
7      # Display the list.
8      print('The list before the insert:')
9      print(names)
10
11     # Insert a new name at element 0.
12     names.insert(0, 'Joe')
13
14     # Display the list again.
15     print('The list after the insert:')
16     print(names)
17
18 # Call the main function.
19 main()
```

Program Output

```
The list before the insert:
['James', 'Kathryn', 'Bill']
The list after the insert:
['Joe', 'James', 'Kathryn', 'Bill']
```

Sort()

- my_list = [9, 1, 0, 2, 8, 6, 7, 4, 5, 3]
- print('Original order:', my_list)
- my_list.sort()
- print('Sorted order:', my_list)
- When this code runs it will display the following:
- Original order: [9, 1, 0, 2, 8, 6, 7, 4, 5, 3]
- Sorted order: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
- Here is another example:
- my_list = ['beta', 'alpha', 'delta', 'gamma']
- print('Original order:', my_list)
- my_list.sort()
- print('Sorted order:', my_list)
- When this code runs it will display the following:
- Original order: ['beta', 'alpha', 'delta', 'gamma']
- Sorted order: ['alpha', 'beta', 'delta', 'gamma']

Remove()

```
1  # This program demonstrates how to use the remove
2  # method to remove an item from a list.
3
4  def main():
5      # Create a list with some items.
6      food = ['Pizza', 'Burgers', 'Chips']
7
8      # Display the list.
9      print('Here are the items in the food list:')
10     print(food)
11
12     # Get the item to change.
13     item = input('Which item should I remove? ')
14
15     try:
16         # Remove the item.
17         food.remove(item)
18
19         # Display the list.
20         print('Here is the revised list:')
21         print(food)
22
23     except ValueError:
```

```
24          print('That item was not found in the list.')
25
26  # Call the main function.
27  main()
```

Program Output (with input shown in bold)

```
Here are the items in the food list:
['Pizza', 'Burgers', 'Chips']
Which item should I remove? Burgers Enter
Here is the revised list:
['Pizza', 'Chips']
```

The remove method that you saw earlier removes a specific item from a list, if that item is in the list.

Reverse()

- The reverse method simply reverses the order of the items in the list. Here is an example:
- my_list = [1, 2, 3, 4, 5]
- print('Original order:', my_list)
- my_list.reverse()
- print('Reversed:', my_list)
- This code will display the following:
- Original order: [1, 2, 3, 4, 5]
- Reversed: [5, 4, 3, 2, 1]

del Statement

- Some situations might require that you remove an element from a specific index, regardless of the item that is stored at that index.
- This can be accomplished with the del statement
- my_list = [1, 2, 3, 4, 5]
- print('Before deletion:', my_list)
- del my_list[2]
- print('After deletion:', my_list)
- This code will display the following:
- Before deletion: [1, 2, 3, 4, 5]
- After deletion: [1, 2, 4, 5]

The min and max Functions

- Python has two built-in functions named min and max that work with sequences.
- The min function accepts a sequence, such as a list, as an argument and returns the item that has the lowest value in the sequence

```
my_list = [5, 4, 3, 2, 50, 40, 30]
print('The lowest value is', min(my_list))
```

- This code will display the following:

```
The lowest value is 2
```

- The max function accepts a sequence, such as a list, as an argument and returns the item that has the highest value in the sequence.
- Here is an example:

```
my_list = [5, 4, 3, 2, 50, 40, 30]
print('The highest value is', max(my_list))
```

- This code will display the following:

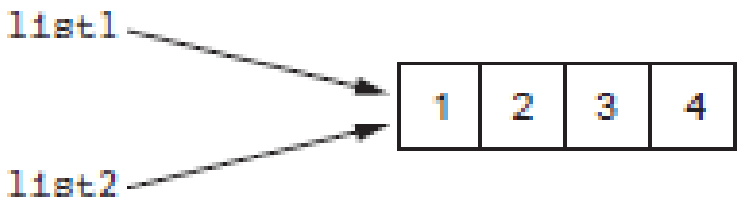
```
The highest value is 50
```

Copying Lists

- To make a copy of a list, you must copy the list's elements.
- Python, assigning one variable to another variable simply makes both variables reference the same object in memory
- list1 = [1, 2, 3, 4] # Create a list.
- list2 = list1 # Assign the list to the list2 variable
- After this code executes, both variables list1 and list2 will reference the same list in memory.

```
 1  >>> list1 = [1, 2, 3, 4] Enter
 2  >>> list2 = list1 Enter
 3  >>> print(list1) Enter
 4  [1, 2, 3, 4]
 5  >>> print(list2) Enter
 6  [1, 2, 3, 4]
 7  >>> list1[0] = 99 Enter
 8  >>> print(list1) Enter
 9  [99, 2, 3, 4]
10  >>> print(list2) Enter
11  [99, 2, 3, 4]
12  >>>
```

`list1` and `list2` reference the same list



Let's take a closer look at each line:

- In line 1 we create a list of integers and assign the list to the `list1` variable.
- In line 2 we assign `list1` to `list2`. After this, both `list1` and `list2` reference the same list in memory.
- In line 3 we print the list referenced by `list1`. The output of the `print` function is shown in line 4.
- In line 5 we print the list referenced by `list2`. The output of the `print` function is shown in line 6. Notice that it is the same as the output shown in line 4.
- In line 7 we change the value of `list[0]` to 99.
- In line 8 we print the list referenced by `list1`. The output of the `print` function is shown in line 9. Notice that the first element is now 99.
- In line 10 we print the list referenced by `list2`. The output of the `print` function is shown in line 11. Notice that the first element is 99

Copying lists

- Suppose you wish to make a copy of the list, so that list1 and list2 reference two separate but identical lists.
- One way to do this is with a loop that copies each element of the list
- # Create a list with values.
- list1 = [1, 2, 3, 4]
- # Create an empty list.
- list2 = []
- # Copy the elements of list1 to list2.
- for item in list1:
- list2.append(item)

More elegant way….copying lists

- # Create a list with values.
- list1 = [1, 2, 3, 4]
- # Create a copy of list1.
- list2 = [] + list1
- The last statement in this code concatenates an empty list with list1 and assigns the resulting list to list2.
- As a result, list1 and list2 will reference two separate but identical lists.

Processing Lists - Totaling the Values in a List

Program 7-8 (total_list.py)

```
1  # This program calculates the total of the values
2  # in a list.
3
4  def main():
5      # Create a list.
6      numbers = [2, 4, 6, 8, 10]
7
8      # Create a variable to use as an accumulator.
9      total = 0
10
11     # Calculate the total of the list elements.
12     for value in numbers:
13         total += value
14
15     # Display the total of the list elements.
16     print('The total of the elements is', total)
17
18 # Call the main function.
19 main()
```

Program Output

```
The total of the elements is 30
```

Averaging the Values in a List

```
1   # This program calculates the average of the values
2   # in a list.
3
4   def main():
5       # Create a list.
6       scores = [2.5, 7.3, 6.5, 4.0, 5.2]
7
8       # Create a variable to use as an accumulator.
9       total = 0.0
10
11      # Calculate the total of the list elements.
12      for value in scores:
13          total += value
14
15      # Calculate the average of the elements.
16      average = total / len(scores)
17
18      # Display the total of the list elements.
19      print('The average of the elements is', average)
20
21  # Call the main function.
22  main()
```

Program Output

```
The average of the elements is 5.3
```

Passing a List as an Argument to a Function

gram 7-10 (total_function.py)

```
# This program uses a function to calculate the
# total of the values in a list.

def main():
    # Create a list.
    numbers = [2, 4, 6, 8, 10]

    # Display the total of the list elements.
    print('The total is', get_total(numbers))

# The get_total function accepts a list as an
# argument returns the total of the values in
# the list.
def get_total(value_list):
    # Create a variable to use as an accumulator.
    total = 0

    # Calculate the total of the list elements.
    for num in value_list:
        total += num

    # Return the total.
    return total

# Call the main function.
main()
```

Program Output

The total is 30

Returning a List from a Function


```
# This program uses a function to create a list.
# The function returns a reference to the list.
def main():
    # Get a list with values stored in it.
    numbers = get_values()

    # Display the values in the list.
    print('The numbers in the list are:')
    print(numbers)

    # The get_values function gets a series of numbers
    # from the user and stores them in a list. The
    # function returns a reference to the list.
def get_values():
    # Create an empty list.
    values = []
    # Create a variable to control the loop.
    again = 'y'

    # Get values from the user and add them to
    # the list.
    while again == 'y':
        # Get a number and add it to the list.
        num = int(input('Enter a number: '))
        values.append(num)

        # Want to do this again?
        print('Do you want to add another number?')
        again = input('y = yes, anything else = no: ')
        print()

    # Return the list.
    return values

 # Call the main function.
main()
```

Output

Program Output (with input shown in bold)

```
Enter a number: 1 [Enter]
Do you want to add another number?
y = yes, anything else = no: y [Enter]

Enter a number: 2 [Enter]
Do you want to add another number?
y = yes, anything else = no: y [Enter]

Enter a number: 3 [Enter]
Do you want to add another number?
y = yes, anything else = no: y [Enter]
```

Program Output *(continued)*

```
Enter a number: 4 [Enter]
Do you want to add another number?
y = yes, anything else = no: y [Enter]

Enter a number: 5 [Enter]
Do you want to add another number?
y = yes, anything else = no: n [Enter]
The numbers in the list are:
[1, 2, 3, 4, 5]
```

Working with Lists and Files


```
1  # This program uses the writelines method to save
2  # a list of strings to a file.
3
4  def main():
5      # Create a list of strings.
6      cities = ['New York', 'Boston', 'Atlanta', 'Dallas']
7
8      # Open a file for writing.
9      outfile = open('cities.txt', 'w')
10
11     # Write the list to the file.
12     outfile.writelines(cities)
13
14     # Close the file.
15     outfile.close()
16
17 # Call the main function.
18 main()
```

Working with Lists and Files

```
1  # This program saves a list of strings to a file.
2
3  def main():
4      # Create a list of strings.
5      cities = ['New York', 'Boston', 'Atlanta', 'Dallas']
6
7      # Open a file for writing.
8      outfile = open('cities.txt', 'w')
9
10     # Write the list to the file.
11     for item in cities:
12         outfile.write(item + '\n')
13
14     # Close the file.
15     outfile.close()
16
17 # Call the main function.
18 main()
```

Reads a file's contents into a list

```
1  # This program reads a file's contents into a list.
2
3  def main():
4      # Open a file for reading.
5      infile = open('cities.txt', 'r')
6
7      # Read the contents of the file into a list.
8      cities = infile.readlines()
9
10     # Close the file.
11     infile.close()
12
13     # Strip the \n from each element.
14     index = 0
15     while index < len(cities):
16         cities[index] = cities[index].rstrip('\n')
17         index += 1
18
19     # Print the contents of the list.
20     print(cities)
21
22 # Call the main function.
23 main()
```

program saves a list of numbers to a file

```
1  # This program saves a list of numbers to a file.
2
3  def main():
4      # Create a list of numbers.
5      numbers = [1, 2, 3, 4, 5, 6, 7]
6
7      # Open a file for writing.
8      outfile = open('numberlist.txt', 'w')
9
10     # Write the list to the file.
11     for item in numbers:
12         outfile.write(str(item) + '\n')
13
14     # Close the file.
15     outfile.close()
16
17 # Call the main function.
18 main()
```

This program reads numbers from a file into a list.

```
1  # This program reads numbers from a file into a list.
2
3  def main():
4      # Open a file for reading.
5      infile = open('numberlist.txt', 'r')
6
7      # Read the contents of the file into a list.
8      numbers = infile.readlines()
9
10     # Close the file.
11     infile.close()
12
13     # Convert each element to an int.
14     index = 0
15     while index < len(numbers):
16         numbers[index] = int(numbers[index])
17         index += 1
18
19     # Print the contents of the list.
20     print(numbers)
21
22 # Call the main function.
23 main()
```

Two-Dimensional Lists

- A two-dimensional list is a list that has other lists as its elements

```
1  >>> students = [['Joe', 'Kim'], ['Sam', 'Sue'], ['Kelly', 'Chris']] Enter
2  >>> print(students) Enter
3  [['Joe', 'Kim'], ['Sam', 'Sue'], ['Kelly', 'Chris']]
4  >>> print(students[0]) Enter
5  ['Joe', 'Kim']
6  >>> print(students[1]) Enter
```

Two-Dimensional Lists

```
 7   ['Sam', 'Sue']
 8   >>> print(students[2]) Enter
 9   ['Kelly', 'Chris']
10   >>>
```

's take a closer look at each line.

- Line 1 creates a list and assigns it to the `students` variable. The list has three elements, and each element is also a list. The element at `students[0]` is
  `['Joe', 'Kim']`

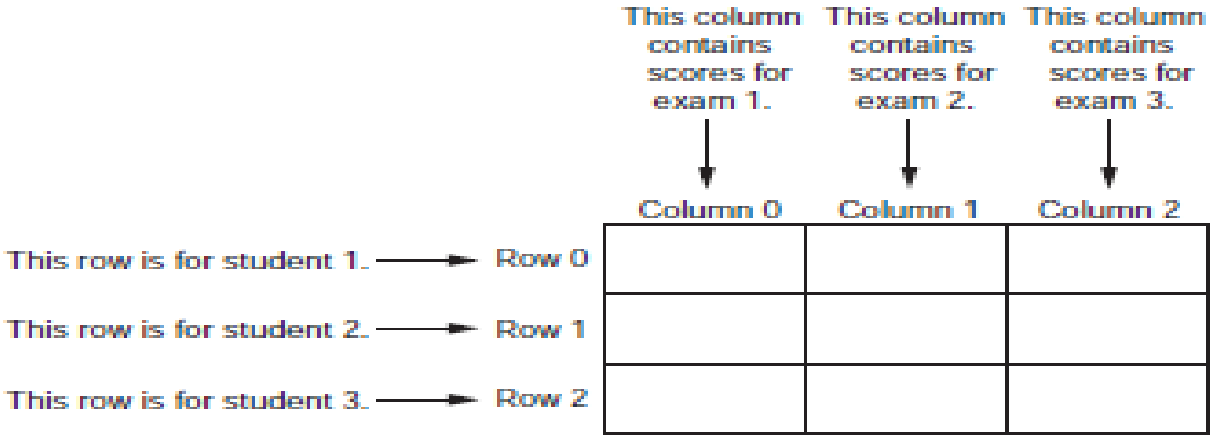
  The element at `students[1]` is
  `['Sam', 'Sue']`

  The element at `students[2]` is
  `['Kelly', 'Chris']`
- Line 2 prints the entire `students` list. The output of the `print` function is shown in line 3.
- Line 4 prints the `students[0]` element. The output of the `print` function is shown in line 5.
- Line 6 prints the `students[1]` element. The output of the `print` function is shown in line 7.
- Line 8 prints the `students[2]` element. The output of the `print` function is shown in line 9.

- Lists of lists are also known as *nested lists, or two-dimensional lists.*
- *It is common to think* of a two-dimensional list as having rows and columns of elements

	Column 0	Column 1
Row 0	'Joe'	'Kim'
Row 1	'Sam'	'Sue'
Row 2	'Kelly'	'Chris'

ure 7-6 Two-dimensional list with three rows and three columns



When processing the data in a two-dimensional list, you need two subscripts: one for the rows and one for the columns. For example, suppose we create a two-dimensional list with the following statement:

```
scores = [[0, 0, 0],
          [0, 0, 0],
          [0, 0, 0]]
```

The elements in row 0 are referenced as follows:

```
scores[0][0]
scores[0][1]
scores[0][2]
```

The elements in row 1 are referenced as follows:

```
scores[1][0]
scores[1][1]
scores[1][2]
```

And, the elements in row 2 are referenced as follows:

```
scores[2][0]
scores[2][1]
scores[2][2]
```

```
1   # This program assigns random numbers to
2   # a two-dimensional list.
3   import random
4
5   # Constants for rows and columns
6   ROWS = 3
7   COLS = 4
8
9   def main():
10      # Create a two-dimensional list.
11      values = [[0, 0, 0, 0],
12                [0, 0, 0, 0],
13                [0, 0, 0, 0]]
14
15      # Fill the list with random numbers.
16      for r in range(ROWS):
17          for c in range(COLS):
18              values[r][c] = random.randint(1, 100)
19
20      # Display the random numbers.
21      print(values)
22
23  # Call the main function.
24  main()
```

Program Output

```
[[4, 17, 34, 24], [46, 21, 54, 10], [54, 92, 20, 100]]
```

Tuples

- A tuple is an immutable sequence, which means that its contents cannot be changed.
- A *tuple is a sequence, very much like a list. The primary difference between tuples and lists* is that tuples are immutable.
- That means that once a tuple is created, it cannot be changed.
- When you create a tuple, you enclose its elements in a set of parentheses
- >>> my_tuple = (1, 2, 3, 4, 5)
- >>> print(my_tuple)
- (1, 2, 3, 4, 5)

Examples

- >>> names = ('Holly', 'Warren', 'Ashley')
- >>> for n in names:
- print(n)
- Holly
- Warren
- Ashle

- Like lists, tuples support indexing, as shown in the following session:
- >>> names = ('Holly', 'Warren', 'Ashley')
- >>> for i in range(len(names)):
- print(names[i])
- Holly
- Warren
- Ashley

In fact, tuples support all the same operations as lists, except those that change the contents of the list. Tuples support the following:

- Subscript indexing (for retrieving element values only)
- Methods such as `index`
- Built-in functions such as `len`, `min`, and `max`
- Slicing expressions
- The `in` operator
- The `+` and `*` operators

Tuples do not support methods such as `append`, `remove`, `insert`, `reverse`, and `sort`.

Create a tuple with one element

NOTE: If you want to create a tuple with just one element, you must write a trailing comma after the element's value, as shown here:

```
my_tuple = (1,)     # Creates a tuple with one element.
```

If you omit the comma, you will not create a tuple. For example, the following statement simply assigns the integer value 1 to the `value` variable:

```
value = (1)         # Creates an integer.
```

Why tuples?

- If the only difference between lists and tuples is immutability, you might wonder why tuples exist.
- One reason that tuples exist is performance.
- Processing a tuple is faster than processing a list, so tuples are good choices when you are processing lots of data and that data will not be modified.
- Another reason is that tuples are safe. Because you are not allowed to change the contents of a tuple, you can store data in one and rest assured that it will not be modified (accidentally or otherwise) by any code in your program.

Converting Between Lists and Tuples


```
1 >>> number_tuple = (1, 2, 3)
2 >>> number_list = list(number_tuple)
3 >>> print(number_list)
4 [1, 2, 3]
5 >>> str_list = ['one', 'two', 'three']
6 >>> str_tuple = tuple(str_list)
7 >>> print(str_tuple)
8 ('one', 'two', 'three')
9 >>>
```

More About Strings

- Python provides several ways to access the individual characters in a string.
- Strings also have methods that allow you to perform operations on them.
- Python provides a wide variety of tools and programming techniques that you can use to examine and manipulate strings.
- In fact, strings are a type of sequence, so many of the concepts that you learned about sequences

Accessing the Individual Characters in a String

- One of the easiest ways to access the individual characters in a string is to use the for loop.
- Here is the general format:
- for *variable in string:*

  statement

  statement

  etc.
- In the general format, *variable is the name of a variable and string is either a string* literal or a variable that references a string.
- Each time the loop iterates, *variable will reference*
- a copy of a character in *string, beginning with the first character. We say that the* loop iterates over the characters in the string.

Example

- name = 'Juliet'
- for ch in name:
- print(ch)
- When the code executes, it will display the following:
- J
- u
- l
- i
- e
- t

e 8-1 Iterating over the string 'Juliet'

1st Iteration

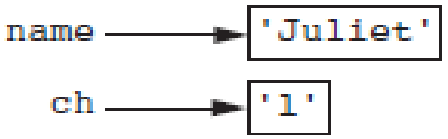
```
for ch in name:
    print(ch)
```

2nd Iteration

```
for ch in name:
    print(ch)
```

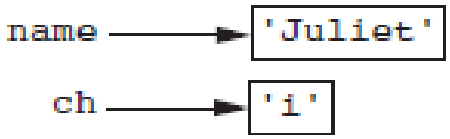
3rd Iteration

```
for ch in name:
    print(ch)
```



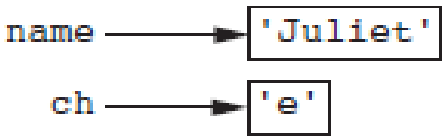
4th Iteration

```
for ch in name:
    print(ch)
```



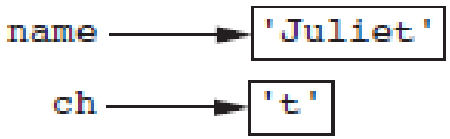
5th Iteration

```
for ch in name:
    print(ch)
```



6th Iteration

```
for ch in name:
    print(ch)
```



NOTE: Figure 8-1 illustrates how the `ch` variable references a copy of a character from the string as the loop iterates. If we change the value that `ch` references in the loop, it has no effect on the string referenced by `name`. To demonstrate, look at the following:

```
1 name = 'Juliet'
2 for ch in name:
3      ch = 'X'
4      print(name)
```

The statement in line 3 merely reassigns the `ch` variable to a different value each time the loop iterates. It has no effect on the string `'Juliet'` that is referenced by `name`, and it has no effect on the number of times the loop iterates. When this code executes, the statement in line 4 will print:

```
Juliet
```

```
1   # This program counts the number of times
2   # the letter T (uppercase or lowercase)
3   # appears in a string.
4
5   def main():
6       # Create a variable to use to hold the count.
7       # The variable must start with 0.
8       count = 0
9
10      # Get a string from the user.
11      my_string = input('Enter a sentence: ')
12
13      # Count the Ts.
14      for ch in my_string:
15          if ch == 'T' or ch == 't':
16              count += 1
17
18      # Print the result.
19      print('The letter T appears', count, 'times.')
20
21  # Call the main function.
22  main()
```

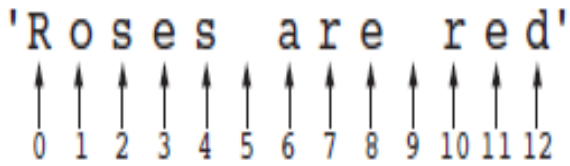
Program Output (with input shown in bold)

```
Enter a sentence: Today we sold twenty-two toys. Enter
The letter T appears 5 times.
```

Indexing

- Another way that you can access the individual characters in a string is with an index.
- Each character in a string has an index that specifies its position in the string.
- Indexing starts at 0, so the index of the first character is 0, the index of the second character is 1, and so forth

8-2 String indexes



You can use an index to retrieve a copy of an individual character in a string, as shown here:

```
my_string = 'Roses are red'
ch = my_string[6]
```

- Here is another example:

```
my_string = 'Roses are red'
print(my_string[0], my_string[6], my_string[10])
```


- This code will print the following:

```
R a r
```

- You can also use negative numbers as indexes, to identify character positions relative to the end of the string.
- The Python interpreter adds negative indexes to the length of the string to determine the character position.
- The index –1 identifies the last character in a string, –2 identifies the next to last character, and so forth.
- The following code shows an example:

```
my_string = 'Roses are red'
print(my_string[-1], my_string[-2], my_string[-13])
```

- This code will print the following:

```
d e R
```

IndexError Exceptions

- An IndexError exception will occur if you try to use an index that is out of range for a particular string.
- For example, the string 'Boston' has 6 characters, so the valid indexes are 0 through 5.
- (The valid negative indexes are –1 through –6.)
- The following is an example of code that causes an IndexError exception.
- city = 'Boston'
- print(city[6])
- This type of error is most likely to happen when a loop incorrectly iterates beyond the end of a string, as shown here:
- city = 'Boston'
- index = 0
- while index < 7:
  print(city[index])
  index += 1
- The last time that this loop iterates, the index variable will be assigned the value 6, which is an invalid index for the string 'Boston'.
- As a result, the print function will cause an IndexError exception to be raised.

The len Function

- city = 'Boston'
- size = len(city)
- The second statement calls the len function, passing the city variable as an argument.
- The function returns the value 6, which is the length of the string 'Boston'.
- This value is assigned to the size variable.
- The len function is especially useful to prevent loops from iterating beyond the end of a string, as shown here:
- city = 'Boston'
- index = 0
- while index < len(city):
  print(city[index])
  index += 1
- Notice that the loop iterates as long as index is *less than the length of the string*.
- *This is because* the index of the last character in a string is always 1 less than the length of the string.

String Concatenation

- A common operation that performed on strings is *concatenation, or appending one string* to the end of another string.
- The + operator produces a string that is the combination of the two strings used as its operands.

```
>>> first_name = 'Emily'
>>> last_name = 'Yeager'
>>> full_name = first_name + ' ' + last_name
>>> print(full_name)
Emily Yeager
>>>
```

Examples

1. >>> *letters = 'abc‘*
2. >>> *letters += 'def‘*
3. >>> *print(letters)*
4. *abcdef*
5. >>>

- >>> name = 'Kelly' e # name is 'Kelly'
- >>> name += ' ' e # name is 'Kelly '
- >>> name += 'Yvonne' e # name is 'Kelly Yvonne'
- >>> name += ' ' e # name is 'Kelly Yvonne '
- >>> name += 'Smith' e # name is 'Kelly Yvonne Smith'
- >>> print(name)
- Kelly Yvonne Smith
- >>>

Strings Are Immutable

- In Python, strings are immutable, which means that once they are created, they cannot be changed.
- Some operations, such as concatenation, give the impression that they modify strings, but in reality they do not.
- Because strings are immutable, you cannot use an expression in the form *string[index] on* the left side of an assignment operator.
- For example, the following code will cause an error:
- # Assign 'Bill' to friend.
- friend = 'Bill'
- # Can we change the first character to 'J'?
- friend[0] = 'J' # No, this will cause an error!
- The last statement in this code will raise an exception because it attempts to change the value of the first character in the string 'Bill'.

Example


```
1   # This program concatenates strings
2
3   def main():
4       name = 'Carmen'
5       print('The name is', name)
6       name = name + ' Brown'
7       print('Now the name is', name)
8
9   # Call the main function.
10  main()
```

Program Output

```
The name is Carmen
Now the name is Carmen Brown
```

- The original string 'Carmen' is not modified.
- Instead, a new string containing 'Carmen Brown' is created and assigned to the name variable.
- The original string, 'Carmen' is no longer usable because no variable references it.
- The Python interpreter will eventually remove the unusable string from memory.

String Slicing

- You can use slicing expressions to select a range of characters from a string
- A slice is a span of items that are taken from a sequence.
- When you take a slice from a string, you get a span of characters from within the string.
- String slices are also called *substrings*.
- To get a slice of a string, you write an expression in the following general format:

  string[start : end]
- In the general format, *start is the index of the first character in the slice, and end is the index* marking the end of the slice.
- The expression will return a string containing a copy of the characters from *start up to (but not including) end.*
- *For example, suppose we have the following:*

  full_name = 'Patty Lynn Smith'

  middle_name = full_name[6:10]

The second statement assigns the string 'Lynn' to the middle_name variable.

String Slicing

- If you leave out the *start index in a slicing expression, Python uses 0 as the starting index.*
- *Here is an example:*

```
full_name = 'Patty Lynn Smith‘
first_name = full_name[:5]
```

- The second statement assigns the string 'Patty' to first_name.
- If you leave out the *end index* in a slicing expression, Python uses the length of the string as the *end index*.
- *Here is an example:*

```
full_name = 'Patty Lynn Smith'
last_name = full_name[11:]
```

- The second statement assigns the string 'Smith' to last_name. What do you think the following code will assign to the my_string variable?

```
full_name = 'Patty Lynn Smith'
my_string = full_name[:]
```

- The second statement assigns the entire string 'Patty Lynn Smith' to my_string. The statement is equivalent to:

```
my_string = full_name[0 : len(full_name)]
```

String Slicing

- The slicing examples we have seen so far get slices of consecutive characters from strings.
- Slicing expressions can also have step value, which can cause characters to be skipped in the string.
- Here is an example of code that uses a slicing expression with a step value:

```
letters = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
print(letters[0:26:2])
```

- The third number inside the brackets is the step value. A step value of 2, as used in this example, causes the slice to contain every second character from the specified range in the string.
- The code will print the following:

```
ACEGIKMOQSUWY
```

String Slicing

- You can also use negative numbers as indexes in slicing expressions to reference positions relative to the end of the string.
- Here is an example:

```
full_name = 'Patty Lynn Smith'
last_name = full_name[-5:]
```

- Recall that Python adds a negative index to the length of a string to get the position referenced by that index.
- The second statement in this code assigns the string 'Smith' to the last_name variable.
- Can you extract branch, year details from your roll nos

String Slicing

- Invalid indexes do not cause slicing expressions to raise an exception. For example:
- If the *end index specifies a position beyond the end of the string, Python will use* the length of the string instead.
- If the *start index specifies a position before the beginning of the string, Python* will use 0 instead.
- If the *start index is greater than the end index, the slicing expression will return* an empty string.

Testing, Searching, and Manipulating Strings

- Python provides operators and methods for testing strings, searching the contents of strings, and getting modified copies of strings.

Testing Strings with in and not in

- In Python you can use the in operator to determine whether one string is contained in another string.
- Here is the general format of an expression using the in operator with two strings:

  string1 in string2
- *string1 and string2 can be either string literals or variables referencing strings.*
- *The expression* returns true if *string1 is found in string2.*
- *For example, look at the following code:*

```
text = 'Four score and seven years ago'
if 'seven' in text:
        print('The string "seven" was found.')
else:
        print('The string "seven" was not found.')
```

- This code determines whether the string 'Four score and seven years ago' contains the string 'seven'.
- If we run this code it will display:

  The string "seven" was found.

not in operator

- You can use the not in operator to determine whether one string is *not contained in* another string.
- Here is an example:

```
names = 'Bill Joanne Susan Chris Juan Katie'
        if 'Pierre' not in names:
                print('Pierre was not found.')
        else:
                print('Pierre was found.')
```

- If we run this code it will display:

```
Pierre was not found.
```

String Methods

- A method is a function that belongs to an object and performs some operation on that object.
- Strings in Python have numerous methods for performing the following types of operations:
  - Testing the values of strings
  - Performing various modifications
  - Searching for substrings and replacing sequences of characters
- Here is the general format of a string method call:

  stringvar.method(arguments)
- In the general format, *stringvar is a variable that references a string*, *method is the name* of the method that is being called, and *arguments is one or more arguments being passed* to the method.

String Testing Methods

Method	Description
`isalnum()`	Returns true if the string contains only alphabetic letters or digits and is at least one character in length. Returns false otherwise.
`isalpha()`	Returns true if the string contains only alphabetic letters and is at least one character in length. Returns false otherwise.
`isdigit()`	Returns true if the string contains only numeric digits and is at least one character in length. Returns false otherwise.
`islower()`	Returns true if all of the alphabetic letters in the string are lowercase, and the string contains at least one alphabetic letter. Returns false otherwise.
`isspace()`	Returns true if the string contains only whitespace characters and is at least one character in length. Returns false otherwise. (Whitespace characters are spaces, newlines (`\n`), and tabs (`\t`).
`isupper()`	Returns true if all of the alphabetic letters in the string are uppercase, and the string contains at least one alphabetic letter. Returns false otherwise.

Examples

```
string1 = '1200'
if string1.isdigit():
    print(string1, 'contains only digits.')
else:
    print(string1, 'contains characters other than digits.')
```

This code will display

```
1200 contains only digits.
```

Here is another example:

```
string2 = '123abc'
if string2.isdigit():
    print(string2, 'contains only digits.')
else:
    print(string2, 'contains characters other than digits.')
```

This code will display

```
123abc contains characters other than digits.
```

Table 8-2 String Modification Methods

Method	Description
`lower()`	Returns a copy of the string with all alphabetic letters converted to lowercase. Any character that is already lowercase, or is not an alphabetic letter, is unchanged.
`lstrip()`	Returns a copy of the string with all leading whitespace characters removed. Leading whitespace characters are spaces, newlines (`\n`), and tabs (`\t`) that appear at the beginning of the string.
`lstrip(char)`	The *`char`* argument is a string containing a character. Returns a copy of the string with all instances of *`char`* that appear at the beginning of the string removed.
`rstrip()`	Returns a copy of the string with all trailing whitespace characters removed. Trailing whitespace characters are spaces, newlines (`\n`), and tabs (`\t`) that appear at the end of the string.
`rstrip(char)`	The *`char`* argument is a string containing a character. The method returns a copy of the string with all instances of *`char`* that appear at the end of the string removed.
`strip()`	Returns a copy of the string with all leading and trailing whitespace characters removed.
`strip(char)`	Returns a copy of the string with all instances of *`char`* that appear at the beginning and the end of the string removed.
`upper()`	Returns a copy of the string with all alphabetic letters converted to uppercase. Any character that is already uppercase, or is not an alphabetic letter, is unchanged.

```
1  # This program demonstrates several string testing methods.
2
3  def main():
4      # Get a string from the user.
5      user_string = input('Enter a string: ')
6
7      print('This is what I found about that string:')
8
9      # Test the string.
10     if user_string.isalnum():
11        print('The string is alphanumeric.')
12     if user_string.isdigit():
13        print('The string contains only digits.')
14     if user_string.isalpha():
15        print('The string contains only alphabetic characters.')
16     if user_string.isspace():
17        print('The string contains only whitespace characters.')
18     if user_string.islower():
19        print('The letters in the string are all lowercase.')
20     if user_string.isupper():
21        print('The letters in the string are all uppercase.')
22
23 # Call the string.
24 main()
```

Program Output (with input shown in bold)

```
Enter a string: abc [Enter]
This is what I found about that string:
The string is alphanumeric.
The string contains only alphabetic characters.
The letters in the string are all lowercase.
```

Program Output (with input shown in bold)

```
Enter a string: 123 [Enter]
This is what I found about that string:
The string is alphanumeric.
The string contains only digits.
```

Program Output (with input shown in bold)

```
Enter a string: 123ABC [Enter]
This is what I found about that string:
The string is alphanumeric.
The letters in the string are all uppercase.
```

Modification Methods

Although strings are immutable, meaning they cannot be modified, they do have a number of methods that return modified versions of themselves. Table 8-2 lists several of these methods.

Table 8-2 String Modification Methods

Method	Description
`lower()`	Returns a copy of the string with all alphabetic letters converted to lowercase. Any character that is already lowercase, or is not an alphabetic letter, is unchanged.
`lstrip()`	Returns a copy of the string with all leading whitespace characters removed. Leading whitespace characters are spaces, newlines (`\n`), and tabs (`\t`) that appear at the beginning of the string.
`lstrip(char)`	The *`char`* argument is a string containing a character. Returns a copy of the string with all instances of *`char`* that appear at the beginning of the string removed.
`rstrip()`	Returns a copy of the string with all trailing whitespace characters removed. Trailing whitespace characters are spaces, newlines (`\n`), and tabs (`\t`) that appear at the end of the string.
`rstrip(char)`	The *`char`* argument is a string containing a character. The method returns a copy of the string with all instances of *`char`* that appear at the end of the string removed.
`strip()`	Returns a copy of the string with all leading and trailing whitespace characters removed.
`strip(char)`	Returns a copy of the string with all instances of *`char`* that appear at the beginning and the end of the string removed.
`upper()`	Returns a copy of the string with all alphabetic letters converted to uppercase. Any character that is already uppercase, or is not an alphabetic letter, is unchanged.

Examples

- For example, the lower method returns a copy of a string with all of its alphabetic letters converted to lowercase. Here is an example:

```
letters = 'WXYZ'
print(letters, letters.lower())
```

- This code will print:

```
WXYZ wxyz
```

- The upper method returns a copy of a string with all of its alphabetic letters converted to uppercase. Here is an example:

```
letters = 'abcd'
print(letters, letters.upper())
```

- This code will print:

```
abcd ABCD
```

- The lower and upper methods are useful for making case-insensitive string comparisons.
- String comparisons are case-sensitive

For example, look at the following code:

```
again = 'y'
while again.lower() == 'y':
    print('Hello')
    print('Do you want to see that again?')
    again = input('y = yes, anything else = no: ')
```

Notice that the last statement in the loop asks the user to enter `y` to see the message displayed again. The loop iterates as long as the expression `again.lower() == 'y'` is true. The expression will be true if the `again` variable references either `'y'` or `'Y'`.

Similar results can be achieved by using the `upper` method, as shown here:

```
again = 'y'
while again.upper() == 'Y':
    print('Hello')
    print('Do you want to see that again?')
    again = input('y = yes, anything else = no: ')
```

Table 8-3 Search and replace methods

Method	Description
`endswith(substring)`	The *substring* argument is a string. The method returns true if the string ends with *substring*.
`find(substring)`	The *substring* argument is a string. The method returns the lowest index in the string where *substring* is found. If *substring* is not found, the method returns –1.
`replace(old, new)`	The *old* and *new* arguments are both strings. The method returns a copy of the string with all instances of *old* replaced by *new*.
`startswith(substring)`	The *substring* argument is a string. The method returns true if the string starts with *substring*.

The endswith method determines whether a string ends with a specified substring. Here is an example:

```
filename = input('Enter the filename: ')
if filename.endswith('.txt'):
    print('That is the name of a text file.')
elif filename.endswith('.py'):
    print('That is the name of a Python source file.')
elif filename.endswith('.doc'):
    print('That is the name of a word processing document.')
else:
    print('Unknown file type.')
```

The startswith method works like the endswith method, but determines whether a string begins with a specified substring.

The find method searches for a specified substring within a string. The method returns the lowest index of the substring, if it is found. If the substring is not found, the method returns –1. Here is an example:

```
string = 'Four score and seven years ago'
position = string.find('seven')
if position != -1:
    print('The word "seven" was found at index', position)
else:
    print('The word "seven" was not found.')
```

This code will display:

```
The word "seven" was found at index 15
```

replace()

- The replace method returns a copy of a string, where every occurrence of a specified substring has been replaced with another string.
- For example, look at the following code:

```
string = 'Four score and seven years ago'
new_string = string.replace('years', 'days')
print(new_string)
```

- This code will display:

```
Four score and seven days ago
```

- Try validating a password

The Repetition Operator

- *string_to_copy * n*
- The repetition operator creates a string that contains *n repeated copies of string_to_copy*.
- Here is an example:
- my_string = 'w' * 5
- After this statement executes, my_string will reference the string 'wwwww'.
- Here is another example:
- print('Hello' * 5)
- This statement will print:
- HelloHelloHelloHelloHello

```
1  # This program demonstrates the repetition operator
2
3  def main():
4      # Print nine rows increasing in length.
5      for count in range(1, 10):
6          print('Z' * count)
7
8      # Print nine rows decreasing in length.
9      for count in range(8, 0, -1):
10         print('Z' * count)
11
12 # Call the main function.
13 main()
```

Program Output

```
Z
ZZ
ZZZ
ZZZZ
ZZZZZ
ZZZZZZ
ZZZZZZZ
ZZZZZZZZ
ZZZZZZZZZ
ZZZZZZZZ
ZZZZZZZ
```

Splitting a String

Strings in Python have a method named `split` that returns a list containing the words in the string. Program 8-9 shows an example.

Program 8-9 (string_split.py)

```
1  # This program demonstrates the split method.
2
3  def main():
4      # Create a string with multiple words.
5      my_string = 'One two three four'
6
7      # Split the string.
8      word_list = my_string.split()
9
10     # Print the list of words.
11     print(word_list)
12
13 # Call the main function.
14 main()
```

Program Output

```
['One', 'two', 'three', 'four']
```

```
1  # This program calls the split method, using the
2  # '/' character as a separator.
3
4  def main():
5      # Create a string with a date.
6      date_string = '11/26/2014'
7
8      # Split the date.
9      date_list = date_string.split('/')
10
11     # Display each piece of the date.
12     print('Month:', date_list[0])
13     print('Day:', date_list[1])
14     print('Year:', date_list[2])
15
16 # Call the main function.
17 main()
```

Program Output

```
Month: 11
Day: 26
Year: 2014
```

Dictionaries and Sets

UNIT-III

Dictionaries

- A dictionary is an object that stores a collection of data. Each element in a dictionary has two parts: a key and a value. You use a key to locate a specific value.
- In Python, a *dictionary is an object that stores a collection of data*.
- *Each element that is* stored in a dictionary has two parts: a *key and a value*.
- *In fact, dictionary elements are commonly* referred to as *key-value pairs*.
- *When you want to retrieve a specific value from a* dictionary, you use the key that is associated with that value
- Key-value pairs are often referred to as *mappings because each key is mapped* to a value.

Creating a Dictionary

- You can create a dictionary by enclosing the elements inside a set of curly braces ({}).
- An element consists of a key, followed by a colon, followed by a value.
- The elements are separated by commas.
- The following statement shows an example:
- phonebook = {'Chris':'555–1111', 'Katie':'555–2222', 'Joanne':'555-3333'}
- This statement creates a dictionary and assigns it to the phonebook variable.

Example

- The dictionary contains the following three elements:
- The first element is 'Chris':'555–1111'. In this element the key is 'Chris' and the value is '555–1111'.
- The second element is 'Katie':'555–2222'. In this element the key is 'Katie' and the value is '555–2222'.
- …

Keys

- In this example the keys and the values are strings.
- The values in a dictionary can be objects of any type, but the keys must be immutable objects.
- For example, keys can be strings, integers, floating-point values, or tuples.
- Keys cannot be lists or any other type of immutable object.

Retrieving a Value from a Dictionary


```
>>> phonebook = {'Chris':'555-1111', 'Katie':'555-2222',
'Joanne':'555-3333'} Enter
>>> phonebook Enter
{'Chris': '555-1111', 'Joanne': '555-3333', 'Katie': '555-2222'}
>>>
```

- The elements in a dictionary are not stored in any particular order
- Notice that the order in which the elements are displayed is different than the order in which they were created.
- This illustrates how dictionaries are not sequences, like lists, tuples, and strings.
- As a result, you cannot use a numeric index to retrieve a value by its position from a dictionary.
- Instead, you use a key to retrieve a value.

Example

- To retrieve a value from a dictionary, you simply write an expression in the following general format:

  dictionary_name[key]

- In the general format, *dictionary_name is the variable that references the dictionary, and key* is a key.
- If the key exists in the dictionary, the expression returns the value that is associated with the key.
- If the key does not exist, a KeyError exception is raised.

```
1   >>> phonebook = {'Chris':'555-1111', 'Katie':'555-2222',
    'Joanne':'555-3333'} Enter
2   >>> phonebook['Chris'] Enter
3   '555-1111'
4   >>> phonebook['Joanne'] Enter
5   '555-3333'
6   >>> phonebook['Katie'] Enter
7   '555-2222'
8   >>> phonebook['Kathryn'] Enter
Traceback (most recent call last):
    File "<pyshell#5>", line 1, in <module>
      phonebook['Kathryn']
KeyError: 'Kathryn'
>>>
```

Let's take a closer look at the session:

- Line 1 creates a dictionary containing names (as keys) and phone numbers (as values).
- In line 2, the expression `phonebook['Chris']` returns the value from the `phonebook` dictionary that is associated with the key `'Chris'`. The value is displayed in line 3.
- In line 4, the expression `phonebook['Joanne']` returns the value from the `phonebook` dictionary that is associated with the key `'Joanne'`. The value is displayed in line 5.
- In line 6, the expression `phonebook['Katie']` returns the value from the `phonebook` dictionary that is associated with the key `'Katie'`. The value is displayed in line 7.
- In line 8, the expression `phonebook['Kathryn']` is entered. There is no such key as `'Kathryn'` in the `phonebook` dictionary, so a `KeyError` exception is raised.

Using the in and not in Operators to Test for a Value in a Dictionary

- A KeyError exception is raised if you try to retrieve a value from a dictionary using a nonexistent key.
- To prevent such an exception, you can use the in operator to determine whether a key exists before you try to use it to retrieve a value.

```
1  >>> phonebook = {'Chris':'555-1111', 'Katie':'555-2222',
   'Joanne':'555-3333'} Enter
2  >>> if 'Chris' in phonebook: Enter
3          print(phonebook['Chris']) Enter Enter
4
5  555-1111
6  >>>
```

You can also use the `not in` operator to determine whether a key does not exist, as demonstrated in the following session:

```
1  >>> phonebook = {'Chris':'555-1111', 'Katie':'555-2222'} Enter
2  >>> if 'Joanne' not in phonebook: Enter
3          print('Joanne is not found.') Enter Enter
4
5  Joanne is not found.
6  >>>
```

NOTE: Keep in mind that string comparisons with the `in` and `not in` operators are case sensitive.

Adding Elements to an Existing Dictionary

- Dictionaries are mutable objects
- *dictionary_name[key] = value*
- In the general format, *dictionary_name is the variable that references the dictionary, and key is a key.*
- *If key already exists in the dictionary, its associated value will be changed to value.*
- *If the key does not exist, it will be added to the dictionary, along with value as its* associated value.
- You cannot have duplicate keys in a dictionary. When you assign a value to an existing key, the new value replaces the existing value.

```
1  >>> phonebook = {'Chris':'555-1111', 'Katie':'555-2222',
   'Joanne':'555-3333'} Enter
2  >>> phonebook['Joe'] = '555-0123' Enter
3  >>> phonebook['Chris'] = '555-4444' Enter
4  >>> phonebook Enter
5  {'Chris': '555-4444', 'Joanne': '555-3333', 'Joe': '555-0123',
   'Katie': '555-2222'}
6  >>>
```

Let's take a closer look at the session:

- Line 1 creates a dictionary containing names (as keys) and phone numbers (as values).
- The statement in line 2 adds a new key-value pair to the `phonebook` dictionary. Because there is no key `'Joe'` in the dictionary, this statement adds the key `'Joe'`, along with its associated value `'555-0123'`.
- The statement in line 3 changes the value that is associated with an existing key. Because the key `'Chris'` already exists in the `phonebook` dictionary, this statement changes its associated value to `'555-4444'`.
- Line 4 displays the contents of the `phonebook` dictionary. The output is shown in line 5.

Deleting Elements

- You can delete an existing key-value pair from a dictionary with the del statement
- In the general format, *dictionary_name is the variable that references the dictionary, and key is a key.*
- *After the statement executes, the key and its associated value will be deleted* from the dictionary.
- If the *key does not exist, a KeyError exception is raised.*

```
1   >>> phonebook = {'Chris':'555-1111', 'Katie':'555-2222',
    'Joanne':'555-3333'} Enter
2   >>> phonebook Enter
3   {'Chris': '555-1111', 'Joanne': '555-3333', 'Katie': '555-2222'}
4   >>> del phonebook['Chris'] Enter
5   >>> phonebook Enter
6   {'Joanne': '555-3333', 'Katie': '555-2222'}
7   >>> del phonebook['Chris'] Enter
8   Traceback (most recent call last):
9       File "<pyshell#5>", line 1, in <module>
10          del phonebook['Chris']
11  KeyError: 'Chris'
12  >>>
```

Let's take a closer look at the session:

- Line 1 creates a dictionary, and line 2 displays its contents.
- Line 4 deletes the element with the key `'Chris'`, and line 5 displays the contents of the dictionary. You can see in the output in line 6 that the element no longer exists in the dictionary.
- Line 7 tries to delete the element with the key `'Chris'` again. Because the element no longer exists, a `KeyError` exception is raised.

To prevent a KeyError exception from being raised, you should use the in operator to determine whether a key exists before you try to delete it and its associated value. The following interactive session demonstrates:

```
1  >>> phonebook = {'Chris':'555-1111', 'Katie':'555-2222',
   'Joanne':'555-3333'} Enter
2  >>> if 'Chris' in phonebook: Enter
3          del phonebook['Chris'] Enter Enter
4
5  >>> phonebook Enter
6  {'Joanne': '555-3333', 'Katie': '555-2222'}
7  >>>
```

Mixing Data Types in a Dictionary

- >>> phonebook = {'Chris':'555-1111', 'Katie':'555-2222'}
- >>> *num_items = len(phonebook)*
- >>> print(num_items)
- *2*

Mixing Data Types in a Dictionary

- The keys in a dictionary must be immutable objects, but their associated values can be any type of object.

```
1  >>> test_scores = { 'Kayla' : [88, 92, 100], Enter
2                   'Luis' : [95, 74, 81], Enter
3                   'Sophie' : [72, 88, 91], Enter
4                   'Ethan' : [70, 75, 78] } Enter
5  >>> test_scores Enter
6  {'Kayla': [88, 92, 100], 'Sophie': [72, 88, 91], 'Ethan': [70,
   75, 78],
7  'Luis': [95, 74, 81]}
8  >>> test_scores['Sophie'] Enter
9  [72, 88, 91]
10 >>> kayla_scores = test_scores['Kayla'] Enter
11 >>> print(kayla_scores) Enter
12 [88, 92, 100]
13 >>>
```

Let's take a closer look at the session. This statement in lines 1 through 4 creates a dictionary and assigns it to the `test_scores` variable. The dictionary contains the following four elements:

- The first element is `'Kayla' : [88, 92, 100]`. In this element the key is `'Kayla'` and the value is the list `[88, 92, 100]`.
- The second element is `'Luis' : [95, 74, 81]`. In this element the key is `'Luis'` and the value is the list `[95, 74, 81]`.
- The third element is `'Sophie' : [72, 88, 91]`. In this element the key is `'Sophie'` and the value is the list `[72, 88, 91]`.
- The fourth element is `'Ethan' : [70, 75, 78]`. In this element the key is `'Ethan'` and the value is the list `[70, 75, 78]`.

```
1  >>> mixed_up = {'abc':1, 999:'yada yada', (3, 6, 9):[3, 6, 9]} Enter
2  >>> mixed_up Enter
3  {(3, 6, 9): [3, 6, 9], 'abc': 1, 999: 'yada yada'}
4  >>>
```

This statement in line 1 creates a dictionary and assigns it to the `mixed_up` variable. The dictionary contains the following elements:

- The first element is `'abc':1`. In this element the key is the string `'abc'` and the value is the integer 1.
- The second element is `999:'yada yada'`. In this element the key is the integer 999 and the value is the string `'yada yada'`.
- The third element is `(3, 6, 9):[3, 6, 9]`. In this element the key is the tuple `(3, 6, 9)` and the value is the list `[3, 6, 9]`.

The following interactive session gives a more practical example. It creates a dictionary that contains various pieces of data about an employee:

```
1  >>> employee = {'name' : 'Kevin Smith', 'id' : 12345, 'payrate' :
   25.75 } Enter
2  >>> employee Enter
3  {'payrate': 25.75, 'name': 'Kevin Smith', 'id': 12345}
4  >>>
```

This statement in line 1 creates a dictionary and assigns it to the `employee` variable. The dictionary contains the following elements:

- The first element is `'name' : 'Kevin Smith'`. In this element the key is the string `'name'` and the value is the string `'Kevin Smith'`.
- The second element is `'id' : 12345`. In this element the key is the string `'id'` and the value is the integer 12345.
- The third element is `'payrate' : 25.75`. In this element the key is the string `'payrate'` and the value is the floating-point number 25.75.

Creating an Empty Dictionary

- Sometimes you need to create an empty dictionary and then add elements to it as the program executes.
- phonebook = {}
- *>>> phonebook['Chris'] = '555-1111‘*
- >>> phonebook['Katie'] = '555-2222‘
- *>>> phonebook['Joanne'] = '555-3333‘*
- *>>> phonebook*
- *{'Chris': '555-1111', 'Joanne': '555–3333', 'Katie': '555–2222'}*
- >>>

- You can also use the built-in dict() method to create an empty dictionary, as shown in the following statement:

  phonebook = dict()

Using the for Loop to Iterate over a Dictionary

```
for var in dictionary:
    statement
    statement
    etc.
```

In the general format, *var* is the name of a variable and *dictionary* is the name of a dictionary. This loop iterates once for each element in the dictionary. Each time the loop iterates, *var* is assigned a key. The following interactive session demonstrates:

```
1  >>> phonebook = {'Chris':'555-1111', Enter
2                 'Katie':'555-2222', Enter
3                 'Joanne':'555-3333'} Enter
4  >>> for key in phonebook: Enter
5          print(key) Enter Enter
6
7
8  Chris
9  Joanne
10 Katie
11 >>> for key in phonebook: Enter
12         print(key, phonebook[key]) Enter Enter
13
14
15 Chris 555-1111
16 Joanne 555-3333
17 Katie 555-2222
18 >>>
```

Table 9-1 Some of the dictionary methods

Method	Description
`clear`	Clears the contents of a dictionary.
`get`	Gets the value associated with a specified key. If the key is not found, the method does not raise an exception. Instead, it returns a default value.
`items`	Returns all the keys in a dictionary and their associated values as a sequence of tuples.
`keys`	Returns all the keys in a dictionary as a sequence of tuples.
`pop`	Returns the value associated with a specified key and removes that key-value pair from the dictionary. If the key is not found, the method returns a default value.
`popitem`	Returns a randomly selected key-value pair as a tuple from the dictionary and removes that key-value pair from the dictionary.
`values`	Returns all the values in the dictionary as a sequence of tuples.

The `clear` Method

The `clear` method deletes all the elements in a dictionary, leaving the dictionary empty. The method's general format is

```
dictionary.clear()
```

The following interactive session demonstrates the method:

```
1  >>> phonebook = {'Chris':'555-1111', 'Katie':'555-2222'} Enter
2  >>> phonebook Enter
3  {'Chris': '555-1111', 'Katie': '555-2222'}
4  >>> phonebook.clear() Enter
5  >>> phonebook Enter
6  {}
7  >>>
```

The get Method

You can use the `get` method as an alternative to the `[]` operator for getting a value from a dictionary. The `get` method does not raise an exception if the specified key is not found. Here is the method's general format:

```
dictionary.get(key, default)
```

In the general format, *`dictionary`* is the name of a dictionary, *`key`* is a key to search for in the dictionary, and *`default`* is a default value to return if the *`key`* is not found. When the method is called, it returns the value that is associated with the specified *`key`*. If the specified *`key`* is not found in the dictionary, the method returns *`default`*. The following interactive session demonstrates:

```
1  >>> phonebook = {'Chris':'555-1111', 'Katie':'555-2222'} Enter
2  >>> value = phonebook.get('Katie', 'Entry not found') Enter
3  >>> print(value) Enter
4  555-2222
5  >>> value = phonebook.get('Andy', 'Entry not found') Enter
6  >>> print(value) Enter
7  Entry not found
8  >>>
```

The `items` Method

The `items` method returns all of a dictionary's keys and their associated values. They are returned as a special type of sequence known as a *dictionary view*. Each element in the dictionary view is a tuple, and each tuple contains a key and its associated value. For example, suppose we have created the following dictionary:

```
phonebook = {'Chris':'555-1111', 'Katie':'555-2222', 'Joanne':'555-3333'}
```

If we call the `phonebook.items()` method, it returns the following sequence:

```
[('Chris', '555-1111'), ('Joanne', '555-3333'), ('Katie', '555-2222')]
```

Notice the following:

- The first element in the sequence is the tuple `('Chris', '555-1111')`.
- The second element in the sequence is the tuple `('Joanne', '555-3333')`.
- The third element in the sequence is the tuple `('Katie', '555-2222')`.

You can use the `for` loop to iterate over the tuples in the sequence. The following interactive session demonstrates:

```
1  >>> phonebook = {'Chris':'555-1111', Enter
2                   'Katie':'555-2222', Enter
3                   'Joanne':'555-3333'} Enter
4  >>> for key, value in phonebook.items(): Enter
5          print(key, value) Enter Enter
6
7
8  Chris 555-1111
9  Joanne 555-3333
10 Katie 555-2222
11 >>>
```

Here is a summary of the statements in the session:

- Lines 1 through 3 create a dictionary with three elements and assign it to the `phonebook` variable.
- The `for` loop in lines 4 through 5 calls the `phonebook.items()` method, which returns a sequence of tuples containing the key-value pairs in the dictionary. The loop iterates once for each tuple in the sequence. Each time the loop iterates, the values of a tuple are assigned to the `key` and `value` variables. Line 5 prints the value of the `key` variable, followed by the value of the `value` variable. Lines 8 through 10 show the output of the loop.

The keys Method

The keys method returns all of a dictionary's keys as a dictionary view, which is a type of sequence. Each element in the dictionary view is a key from the dictionary. For example, suppose we have created the following dictionary:

```
phonebook = {'Chris':'555-1111', 'Katie':'555-2222', 'Joanne':'555-3333'}
```

If we call the phonebook.keys() method, it will return the following sequence:

```
['Chris', 'Joanne', 'Katie']
```

The following interactive session shows how you can use a for loop to iterate over the sequence that is returned from the keys method:

```
1  >>> phonebook = {'Chris':'555-1111', Enter
2                   'Katie':'555-2222', Enter
3                   'Joanne':'555-3333'} Enter
4  >>> for key in phonebook.keys(): Enter
5          print(key) Enter Enter
6
7
8  Chris
9  Joanne
10 Katie
```

The pop Method

The pop method returns the value associated with a specified key and removes that key-value pair from the dictionary. If the key is not found, the method returns a default value. Here is the method's general format:

```
dictionary.pop(key, default)
```

In the general format, *dictionary* is the name of a dictionary, *key* is a key to search for in the dictionary, and *default* is a default value to return if the *key* is not found. When the method is called, it returns the value that is associated with the specified *key*, and it removes that key-value pair from the dictionary. If the specified *key* is not found in the dictionary, the method returns *default*. The following interactive session demonstrates:

```
>>> phonebook = {'Chris':'555-1111', Enter
                 'Katie':'555-2222', Enter
                 'Joanne':'555-3333'} Enter
>>> phone_num = phonebook.pop('Chris', 'Entry not found') Enter
>>> phone_num Enter
'555-1111'
>>> phonebook Enter
{'Joanne': '555-3333', 'Katie': '555-2222'}
>>> phone_num = phonebook.pop('Andy', 'Element not found') Enter
>>> phone_num Enter
'Element not found'
>>> phonebook Enter
{'Joanne': '555-3333', 'Katie': '555-2222'}
```

Here is a summary of the statements in the session:

- Lines 1 through 3 create a dictionary with three elements and assign it to the `phonebook` variable.
- Line 4 calls the `phonebook.pop()` method, passing `'Chris'` as the key to search for. The value that is associated with the key `'Chris'` is returned and assigned to the `phone_num` variable. The key-value pair containing the key `'Chris'` is removed from the dictionary.
- Line 5 displays the value assigned to the `phone_num` variable. The output is displayed in line 6. Notice that this is the value that was associated with the key `'Chris'`.
- Line 7 displays the contents of the `phonebook` dictionary. The output is shown in line 8. Notice the key-value pair that contained the key `'Chris'` is no longer in the dictionary.
- Line 9 calls the `phonebook.pop()` method, passing `'Andy'` as the key to search for. The key is not found, so the string `'Entry not found'` is assigned to the `phone_num` variable.
- Line 10 displays the value assigned to the `phone_num` variable. The output is displayed in line 11.
- Line 12 displays the contents of the `phonebook` dictionary. The output is shown in line 13.

The popitem Method

- The popitem method returns a randomly selected key-value pair, and it removes that keyvalue pair from the dictionary.
- The key-value pair is returned as a tuple. Here is the method's general format:

  dictionary.popitem()

- You can use an assignment statement in the following general format to assign the returned key and value to individual variables:

  k, v = dictionary.popitem()

- This type of assignment is known as a *multiple assignment because multiple variables are* being assigned at once.
- In the general format, *k and v are variables. After the statement* executes, *k is assigned a randomly selected key from the dictionary, and v is assigned the* value associated with that key.
- The key-value pair is removed from the *dictionary*.

The following interactive session demonstrates:

```
1   >>> phonebook = {'Chris':'555-1111', Enter
2                    'Katie':'555-2222', Enter
3                    'Joanne':'555-3333'} Enter
4   >>> phonebook Enter
5   {'Chris': '555-1111', 'Joanne': '555-3333', 'Katie': '555-2222'}
6   >>> key, value = phonebook.popitem() Enter
7   >>> print(key, value) Enter
8   Chris 555-1111
9   >>> phonebook Enter
10  {'Joanne': '555-3333', 'Katie': '555-2222'}
11  >>>
```

Here is a summary of the statements in the session:

- Lines 1 through 3 create a dictionary with three elements and assign it to the `phonebook` variable.
- Line 4 displays the dictionary's contents, shown in line 5.
- Line 6 calls the `phonebook.popitem()` method. The key and value that are returned from the method are assigned to the variables `key` and `value`. The key-value pair is removed from the dictionary.
- Line 7 displays the values assigned to the `key` and `value` variables. The output is shown in line 8.
- Line 9 displays the contents of the dictionary. The output is shown in line 10. Notice that the key-value pair that was returned from the `popitem` method in line 6 has been removed.

Keep in mind that the `popitem` method raises a `KeyError` exception if it is called on an empty dictionary.

The values Method

- The values method returns all a dictionary's values (without their keys) as a dictionary view, which is a type of sequence.
- Each element in the dictionary view is a value from the dictionary.
- For example, suppose we have created the following dictionary:
- phonebook = {'Chris':'555−1111', 'Katie':'555−2222', 'Joanne':'555−3333'}

```
>>> phonebook = {'Chris':'555-1111', Enter
                 'Katie':'555-2222', Enter
                 'Joanne':'555-3333'} Enter
>>> for val in phonebook.values(): Enter
        print(val) Enter Enter


555-1111
555-3333
555-2222
```

Sets

- A set contains a collection of unique values and works like a mathematical set.
- A *set is an object that stores a collection of data in the same way as mathematical sets*.
- *Here* are some important things to know about sets:
  - All the elements in a set must be unique. No two elements can have the same value.
  - Sets are unordered, which means that the elements in a set are not stored in any particular order.
  - The elements that are stored in a set can be of different data types.

Creating a Set

- A set contains a collection of unique values and works like a mathematical set.
- All the elements in a set must be unique. No two elements can have the same value.
- Sets are unordered, which means that the elements in a set are not stored in any particular order.
- The elements that are stored in a set can be of different data types.
- Sets cannot contain duplicate elements.
- If you pass an argument containing duplicate elements
- to the set function, only one of the duplicated elements will appear in the set.

Creating a Set

To create a set, you have to call the built-in `set` function. Here is an example of how you create an empty set:

```
myset = set()
```

After this statement executes, the `myset` variable will reference an empty set. You can also pass one argument to the `set` function. The argument that you pass must be an object that contains iterable elements, such as a list, a tuple, or a string. The individual elements of the object that you pass as an argument become elements of the set. Here is an example:

```
myset = set(['a', 'b', 'c'])
```

In this example we are passing a list as an argument to the `set` function. After this statement executes, the `myset` variable references a set containing the elements `'a'`, `'b'`, and `'c'`.

If you pass a string as an argument to the set function, each individual character in the string becomes a member of the set. Here is an example:

```
myset = set('abc')
```

After this statement executes, the `myset` variable will reference a set containing the elements `'a'`, `'b'`, and `'c'`.

Example

- myset = set('aaabc')
- The character 'a' appears multiple times in the string, but it will appear only once in the set.
- After this statement executes, the myset variable will reference a set containing the elements 'a', 'b', and 'c‘.

Example

- What if you want to create a set in which each element is a string containing more than one character?
- For example, how would you create a set containing the elements 'one', 'two', and 'three'? The following code does not accomplish the task because you can pass no more than one argument to the set function:

```
# This is an ERROR!
myset = set('one', 'two', 'three')
```

- The following does not accomplish the task either:

```
# This does not do what we intend.
myset = set('one two three')
```

- After this statement executes, the myset variable will reference a set containing the elements 'o', 'n', 'e', ' ', 't', 'w', 'h', and 'r'.
- To create the set that we want, we have to pass a list containing the strings ‘one’, ‘two’, and ‘three’ as an argument to the set function.
- Here is an example:

```
# OK, this works.
myset = set(['one', 'two', 'three'])
```

- After this statement executes, the myset variable will reference a set containing the elements 'one', 'two', and 'three’

Example

Getting the Number of Elements in a Set

As with lists, tuples, and dictionaries, you can use the `len` function to get the number of elements in a set. The following interactive session demonstrates:

```
1  >>> myset = set([1, 2, 3, 4, 5]) Enter
2  >>> len(myset) Enter
3  5
4  >>>
```

Adding and Removing Elements

Sets are mutable objects, so you can add items to them and remove items from them. You use the add method to add an element to a set. The following interactive session demonstrates:

```
1   >>> myset = set() Enter
2   >>> myset.add(1) Enter
3   >>> myset.add(2) Enter
4   >>> myset.add(3) Enter
5   >>> myset Enter
6   {1, 2, 3}
7   >>> myset.add(2) Enter
8   >>> myset
9   {1, 2, 3}
```

- The statement in line 1 creates an empty set and assigns it to the myset variable.
- The statements in lines 2 through 4 add the values 1, 2, and 3 to the set.
- Line 5 displays the contents of the set, which is shown in line 6.
- The statement in line 7 attempts to add the value 2 to the set. The value 2 is already in the set, however.
- If you try to add a duplicate item to a set with the add method, the method does not raise an exception.
- It simply does not add the item.

Update()

You can add a group of elements to a set all at one time with the `update` method. When you call the `update` method as an argument, you pass an object that contains iterable elements, such as a list, a tuple, string, or another set. The individual elements of the object that you pass as an argument become elements of the set. The following interactive session demonstrates:

```
1   >>> myset = set([1, 2, 3]) Enter
2   >>> myset.update([4, 5, 6]) Enter
3   >>> myset Enter
4   {1, 2, 3, 4, 5, 6}
5   >>>
```

The statement in line 1 creates a set containing the values 1, 2, and 3. Line 2 adds the values 4, 5, and 6. The following session shows another example:

```
1   >>> set1 = set([1, 2, 3]) Enter
2   >>> set2 = set([8, 9, 10]) Enter
3   >>> set1.update(set2) Enter
4   >>> set1
5 {1, 2, 3, 8, 9, 10}
6 >>> set2
7 {8, 9, 10}
8 >>>
```

Example

- >>> myset = set([1, 2, 3])
- *2 >>> myset.update('abc')*
- *3 >>> myset*
- *4 {'a', 1, 2, 3, 'c', 'b'}*
- *5 >>>*
- The statement in line 1 creates a set containing the values 1, 2, and 3.
- Line 2 calls the myset.update method, passing the string 'abc' as an argument.
- This causes the each character of the string to be added as an element to myset.

Remove()

- You can remove an item from a set with either the remove method or the discard method.
- You pass the item that you want to remove as an argument to either method, and that item is removed from the set.
- The only difference between the two methods is how they behave when the specified item is not found in the set.
- The remove method raises a KeyError exception, but the discard method does not raise an exception

Example

```
>>> myset = set([1, 2, 3, 4, 5]) Enter
>>> myset Enter
{1, 2, 3, 4, 5}
>>> myset.remove(1) Enter
>>> myset Enter
{2, 3, 4, 5}
>>> myset.discard(5) Enter
>>> myset Enter
{2, 3, 4}
>>> myset.discard(99) Enter
>>> myset.remove(99) Enter
Traceback (most recent call last):
  File "<pyshell#12>", line 1, in <module>
    myset.remove(99)
KeyError: 99
>>>
```

- Line 1 creates a set with the elements 1, 2, 3, 4, and 5.
- Line 2 displays the contents of the set, which is shown in line 3.
- Line 4 calls the remove method to remove the value 1 from the set.
- You can see in the output shown in line 6 that the value 1 is no longer in the set.
- Line 7 calls the discard method to remove the value 5 from the set.
- You can see in the output in line 9 that the value 5 is no longer in the set.
- Line 10 calls the discard method to remove the value 99 from the set.
- The value is not found in the set, but the discard method does not raise an exception.
- Line 11 calls the remove method to remove the value 99 from the set.
- Because the value is not in the set, a KeyError exception is raised, as shown in lines 12 through 15.

Clear()

You can clear all the elements of a set by calling the `clear` method. The following interactive session demonstrates:

```
1  >>> myset = set([1, 2, 3, 4, 5]) Enter
2  >>> myset Enter
3  {1, 2, 3, 4, 5}
4  >>> myset.clear() Enter
5  >>> myset Enter
6  set()
7  >>>
```

The statement in line 4 calls the `clear` method to clear the set. Notice in line 6 that when we display the contents of an empty set, the interpreter displays `set()`.

Using the for Loop to Iterate over a Set

You can use the `for` loop in the following general format to iterate over all the elements in a set:

```
for var in set:
    statement
    statement
    etc.
```

In the general format, *`var`* is the name of a variable and *`set`* is the name of a set. This loop iterates once for each element in the set. Each time the loop iterates, *`var`* is assigned an element. The following interactive session demonstrates:

```
1  >>> myset = set(['a', 'b', 'c']) Enter
2  >>> for val in myset: Enter
3          print(val) Enter Enter
4
5  a
6  c
7  b
8  >>>
```

Lines 2 through 3 contain a `for` loop that iterates once for each element of the `myset` set. Each time the loop iterates, an element of the set is assigned to the `val` variable. Line 3 prints the value of the `val` variable. Lines 5 through 7 show the output of the loop.

Using the in and not in Operators to Test for a Value in a Set

```
1  >>> myset = set([1, 2, 3]) Enter
2  >>> if 1 in myset: Enter
3          print('The value 1 is in the set.') Enter Enter
4
5  The value 1 is in the set.
6  >>>
```

The `if` statement in line 2 determines whether the value 1 is in the `myset` set. If it is, the statement in line 3 displays a message.

You can also use the `not in` operator to determine if a value does not exist in a set, as demonstrated in the following session:

```
1  >>> myset = set([1, 2, 3]) Enter
2  >>> if 99 not in myset: Enter
3          print('The value 99 is not in the set.') Enter Enter
4
5  The value 99 is not in the set.
6  >>>
```

Finding the Union of Sets

The union of two sets is a set that contains all the elements of both sets. In Python, you can call the `union` method to get the union of two sets. Here is the general format:

```
set1.union(set2)
```

In the general format, *`set1`* and *`set2`* are sets. The method returns a set that contains the elements of both *`set1`* and *`set2`*. The following interactive session demonstrates:

```
1   >>> set1 = set([1, 2, 3, 4]) Enter
2   >>> set2 = set([3, 4, 5, 6]) Enter
3   >>> set3 = set1.union(set2) Enter
4   >>> set3 Enter
5   {1, 2, 3, 4, 5, 6}
6   >>>
```

The statement in line 3 calls the `set1` object's `union` method, passing `set2` as an argument. The method returns a set that contains all the elements of `set1` and `set2` (without duplicates, of course). The resulting set is assigned to the `set3` variable.

You can also use the `|` operator to find the union of two sets. Here is the general format of an expression using the `|` operator with two sets:

```
set1 | set2
```

Example

- In the general format, *set1 and set2 are sets.*
- *The expression returns a set that contains* the elements of both *set1 and set2.*

- *1 >>> set1 = set([1, 2, 3, 4])*
- *2 >>> set2 = set([3, 4, 5, 6])*
- *3 >>> set3 = set1 | set2*
- *4 >>> set3*
- *5 {1, 2, 3, 4, 5, 6}*
- *6 >>>*

Finding the Intersection of Sets

The intersection of two sets is a set that contains only the elements that are found in both sets. In Python, you can call the `intersection` method to get the intersection of two sets. Here is the general format:

```
set1.intersection(set2)
```

In the general format, *set1* and *set2* are sets. The method returns a set that contains the elements that are found in both *set1* and *set2*. The following interactive session demonstrates:

```
1   >>> set1 = set([1, 2, 3, 4]) Enter
2   >>> set2 = set([3, 4, 5, 6]) Enter
3   >>> set3 = set1.intersection(set2) Enter
4   >>> set3 Enter
5   {3, 4}
6   >>>
```

The statement in line 3 calls the `set1` object's `intersection` method, passing `set2` as an argument. The method returns a set that contains the elements that are found in both `set1` and `set2`. The resulting set is assigned to the `set3` variable.

You can also use the `&` operator to find the intersection of two sets. Here is the general format of an expression using the `&` operator with two sets:

```
set1 & set2
```

Example

- In the general format, *set1 and set2 are sets.*
- *The expression returns a set that contains* the elements that are found in both *set1 and set2.*
- *1 >>> set1 = set([1, 2, 3, 4])*
- *2 >>> set2 = set([3, 4, 5, 6])*
- *3 >>> set3 = set1 & set2*
- *4 >>> set3*
- *5 {3, 4}*

Finding the Difference of Sets

The difference of `set1` and `set2` is the elements that appear in `set1` but do not appear in `set2`. In Python, you can call the `difference` method to get the difference of two sets. Here is the general format:

```
set1.difference(set2)
```

In the general format, *`set1`* and *`set2`* are sets. The method returns a set that contains the elements that are found in *`set1`* but not in *`set2`*. The following interactive session demonstrates:

```
1   >>> set1 = set([1, 2, 3, 4]) Enter
2   >>> set2 = set([3, 4, 5, 6]) Enter
3   >>> set3 = set1.difference(set2) Enter
4   >>> set3 Enter
5   {1, 2}
```

You can also use the – operator to find the difference of two sets. Here is the general format of an expression using the – operator with two sets:

```
set1 - set2
```

In the general format, *`set1`* and *`set2`* are sets. The expression returns a set that contains the elements that are found in *`set1`* but not in *`set2`*. The following interactive session demonstrates:

```
1   >>> set1 = set([1, 2, 3, 4]) Enter
2   >>> set2 = set([3, 4, 5, 6]) Enter
3   >>> set3 = set1 - set2 Enter
4   >>> set3 Enter
5   {1, 2}
```

Finding the Symmetric Difference of Sets

- The symmetric difference of two sets is a set that contains the elements that are not shared by the sets.
- In other words, it is the elements that are in one set but not in both.
- In Python, you can call the symmetric_difference method to get the symmetric difference of two sets.
- Here is the general format:

  set1.symmetric_difference(set2)
- In the general format, *set1 and set2 are sets.*
- *The method returns a set that contains the* elements that are found in either *set1 or set2 but not both sets.*
- *1 >>> set1 = set([1, 2, 3, 4])*
- *2 >>> set2 = set([3, 4, 5, 6])*
- *3 >>> set3 = set1.symmetric_difference(set2)*
- *4 >>> set3*
- *5 {1, 2, 5, 6}*
- You can also use the ^ operator to find the symmetric difference of two sets.
- Here is the general format of an expression using the ^ operator with two sets:

  set1 ^ set2

Example

- In the general format, *set1 and set2 are sets.*
- *The expression returns a set that contains* the elements that are found in either *set1 or set2 but not both sets.*
- *The following interactive* session demonstrates:
- *1 >>> set1 = set([1, 2, 3, 4])*
- *2 >>> set2 = set([3, 4, 5, 6])*
- *3 >>> set3 = set1 ^ set2*
- *4 >>> set3*
- *5 {1, 2, 5, 6}*
- *6 >>>*

Finding Subsets and Supersets

Suppose you have two sets and one of those sets contains all of the elements of the other set. Here is an example:

```
set1 = set([1, 2, 3, 4])
set2 = set([2, 3])
```

In this example, `set1` contains all the elements of `set2`, which means that `set2` is a *subset* of `set1`. It also means that `set1` is a *superset* of `set2`. In Python, you can call the `issubset` method to determine whether one set is a subset of another. Here is the general format:

```
set2.issubset(set1)
```

In the general format, *`set1`* and *`set2`* are sets. The method returns `True` if `set2` is a subset of `set1`. Otherwise, it returns `False`. You can call the `issuperset` method to determine whether one set is a superset of another. Here is the general format:

```
set1.issuperset(set2)
```

In the general format, *set1* and *set2* are sets. The method returns `True` if `set1` is a superset of `set2`. Otherwise, it returns `False`. The following interactive session demonstrates:

```
1   >>> set1 = set([1, 2, 3, 4]) Enter
2   >>> set2 = set([2, 3]) Enter
3   >>> set2.issubset(set1) Enter
4   True
5   >>> set1.issuperset(set2) Enter
6   True
7   >>>
```

You can also use the <= operator to determine whether one set is a subset of another and the >= operator to determine whether one set is a superset of another. Here is the general format of an expression using the <= operator with two sets:

```
set2 <= set1
```

In the general format, *set1* and *set2* are sets. The expression returns `True` if `set2` is a subset of `set1`. Otherwise, it returns `False`. Here is the general format of an expression using the >= operator with two sets:

```
set1 >= set2
```

In the general format, *set1* and *set2* are sets. The expression returns `True` if `set1` is a subset of `set2`. Otherwise, it returns `False`. The following interactive session demonstrates:

```
1   >>> set1 = set([1, 2, 3, 4]) Enter
2   >>> set2 = set([2, 3]) Enter
3   >>> set2 <= set1 Enter
4   True
5   >>> set1 >= set2 Enter
6   True
7   >>> set1 <= set2 Enter
```

Serializing Objects

- Serializing a object is the process of converting the object to a stream of bytes that can be saved to a file for later retrieval.
- In Python, object serialization is called pickling.

Serializing Objects

- Sometimes you need to store the contents of a complex object, such as a dictionary or a set, to a file.
- The easiest way to save an object to a file is to serialize the object.
- When an object is *serialized, it is converted to a* stream of bytes that can be easily stored in a file for later retrieval.
- In Python, the process of serializing an object is referred to as *pickling*.
- *The Python standard* library provides a module named pickle that has various functions for serializing, or pickling, objects.

- Once you import the pickle module, you perform the following steps to pickle an object:
  - You open a file for binary writing.
  - You call the pickle module's dump method to pickle the object and write it to the specified file.
  - After you have pickled all the objects that you want to save to the file, you close the file.
- To open a file for binary writing, you use 'wb' as the mode when you call the open function.

How to do it

- The following statement opens a file named mydata.dat for binary writing:

  outputfile = open('mydata.dat', 'wb')
- Once you have opened a file for binary writing, you call the pickle module's dump function.
- Here is the general format of the dump method:

  pickle.dump(*object, file)*
- In the general format, *object is a variable that references the object you want to pickle,* and *file is a variable that references a file object.*
- *After the function executes, the object* referenced by *object will be serialized and written to the file.*
- *You can pickle just about* any type of object, including lists, tuples, dictionaries, sets, strings, integers, and floating point numbers.

You can save as many pickled objects as you want to a file. When you are finished, you call the file object's `close` method to close the file. The following interactive session provides a simple demonstration of pickling a dictionary:

```
1  >>> import pickle Enter
2  >>> phonebook = {'Chris' : '555-1111', Enter
3                   'Katie' : '555-2222', Enter
4                   'Joanne' : '555-3333'} Enter
5  >>> output_file = open('phonebook.dat', 'wb') Enter
6  >>> pickle.dump(phonebook, output_file) Enter
7  >>> output_file.close() Enter
8  >>>
```

Let's take a closer look at the session:

- Line 1 imports the `pickle` module.
- Lines 2 through 4 create a dictionary containing names (as keys) and phone numbers (as values).
- Line 5 opens a file named `phonebook.dat` for binary writing.
- Line 6 calls the `pickle` module's `dump` function to serialize the `phonebook` dictionary and write it to the `phonebook.dat` file.
- Line 7 closes the `phonebook.dat` file.

unpickle

- To retrieve, or unpickle, the objects that you have pickled.
  - You open a file for binary reading.
  - You call the pickle module's load function to retrieve an object from the file and unpickle it.
  - After you have unpickled all the objects that you want from the file, you close the file.

unpickle

- To open a file for binary reading, you use 'rb' as the mode when you call the open function.
- For example, the following statement opens a file named mydata.dat for binary reading:
  inputfile = open('mydata.dat', 'rb')
- Once you have opened a file for binary reading, you call the pickle module's load function.
- Here is the general format of a statement that calls the load function:
  object = pickle.load(file)
- In the general format, *object is a variable, and file is a variable that references a file* object.
- After the function executes, the *object variable will reference an object that was* retrieved from the file and unpickled.
- You can unpickle as many objects as necessary from the file.
- (If you try to read past the end of the file, the load function will raise an EOFError exception.)
- When you are finished, you call the file object's close method to close the file.

```
1  >>> import pickle Enter
2  >>> input_file = open('phonebook.dat', 'rb') Enter
3  >>> pb = pickle.load(inputfile) Enter
4  >>> pb Enter
5  {'Chris': '555-1111', 'Joanne': '555-3333', 'Katie': '555-2222'}
6  >>> input_file.close() Enter
7  >>>
```

Let's take a closer look at the session:

- Line 1 imports the `pickle` module.
- Line 2 opens a file named `phonebook.dat` for binary reading.
- Line 3 calls the `pickle` module's `load` function to retrieve and unpickle an object from the `phonebook.dat` file. The resulting object is assigned to the `pb` variable.
- Line 4 displays the dictionary referenced by the `pb` variable. The output is shown in line 5.
- Line 6 closes the `phonebook.dat` file.

RECURSION

Introduction to Recursion

- A recursive function is a function that calls itself.
- If you run this program, you will have to press Ctrl+C on the keyboard to interrupt its execution.

Program 12-1 (endless_recursion.py)

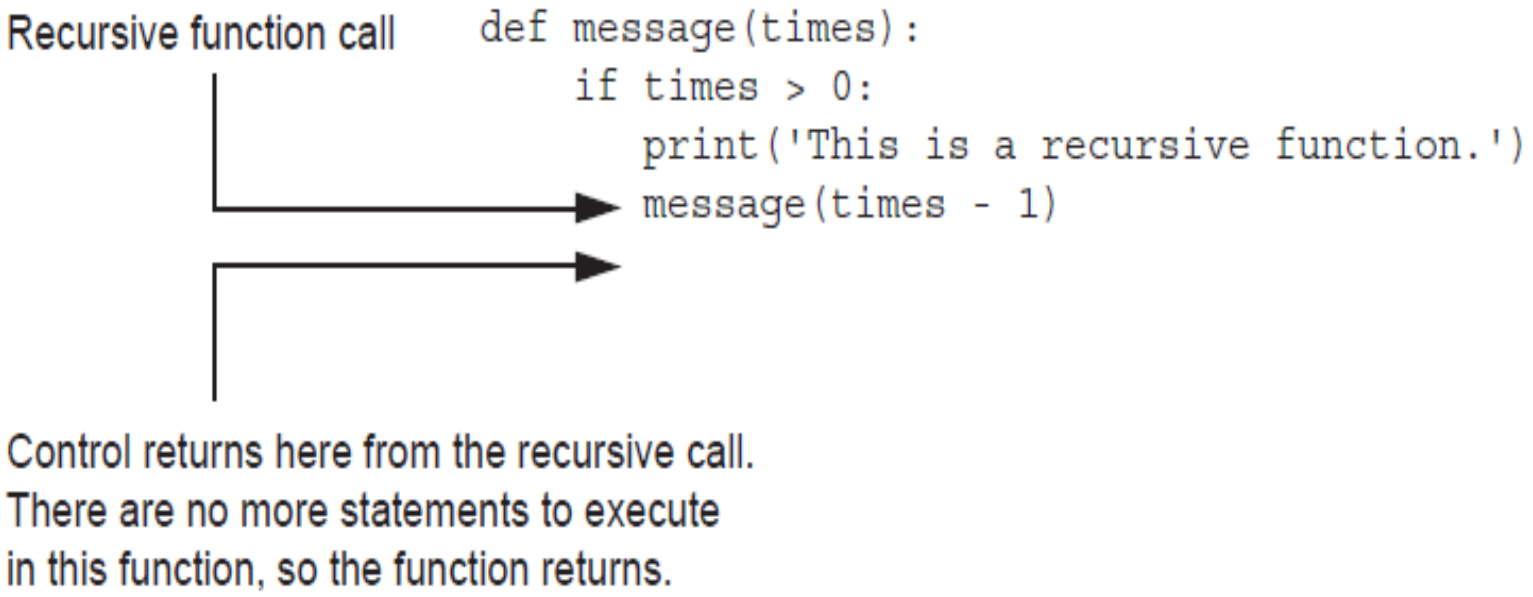
```
1   # This program has a recursive function.
2
3   def main():
4       message()
5
6   def message():
7       print('This is a recursive function.')
8       message()
9
10  # Call the main function.
11  main()
```

```
1  # This program has a recursive function.
2
3  def main():
4      # By passing the argument 5 to the message
5      # function we are telling it to display the
6      # message five times.
7      message(5)
8
9  def message(times):
10     if times > 0:
11         print('This is a recursive function.')
12         message(times - 1)
13
14 # Call the main function.
15 main()
```

Program Output

```
This is a recursive function.
This is a recursive function.
This is a recursive function.
This is a recursive function.
This is a recursive function.
```

12-3 Control returns to the point after the recursive function call



Because there are no more statements to be executed after the function call, the fifth instance of the function returns control of the program back to the fourth instance. This repeats until all instances of the function return.

Problem Solving with Recursion

- **A problem can be solved with recursion if it can be broken down into smaller problems that are identical in structure to the overall problem.**
- First, note that recursion is never required to solve a problem. Any problem that can be solved recursively can also be solved with a loop.
- In fact, recursive algorithms are usually less efficient than iterative algorithms.
- This is because the process of calling a function requires several actions to be performed by the computer.
- These actions include allocating memory for parameters and local variables and storing the address of the program location where control returns after the function terminates.
- These actions, which are sometimes referred to as *overhead, take place with each function call.*
- *Such overhead is not necessary* with a loop.

Problem Solving with Recursion

- Some repetitive problems, however, are more easily solved with recursion than with
- a loop. Where a loop might result in faster execution time, the programmer might be able to design a recursive algorithm faster.
- In general, a recursive function works as follows:
  - If the problem can be solved now, without recursion, then the function solves it and Returns
  - If the problem cannot be solved now, then the function reduces it to a smaller but similar problem and calls itself to solve the smaller problem
- In order to apply this approach, first, we identify at least one case in which the problem can be solved without recursion. This is known as the *base case*.
- *Second, we determine a way* to solve the problem in all other circumstances using recursion. This is called the *recursive case*.
- *In the recursive case, we must always reduce the problem to a smaller version of the* original problem.
- By reducing the problem with each recursive call, the base case will eventually be reached and the recursion will stop.

```
1   # This program uses recursion to calculate
2   # the factorial of a number.
3
4   def main():
5       # Get a number from the user.
6       number = int(input('Enter a nonnegative integer: '))
7
8       # Get the factorial of the number.
9       fact = factorial(number)
10
11      # Display the factorial.
12      print('The factorial of', number, 'is', fact)
13
14  # The factorial function uses recursion to
15  # calculate the factorial of its argument,
16  # which is assumed to be nonnegative.
17  def factorial(num):
18      if num == 0:
19          return 1
20      else:
21          return num * factorial(num - 1)
22
23  # Call the main function.
24  main()
```

Program Output (with input shown in bold)

```
Enter a nonnegative integer: 4 Enter
The factorial of 4 is 24
```

Direct and Indirect Recursion

- The examples we have discussed so far show recursive functions or functions that directly call themselves.
- This is known as *direct recursion*.
- *There is also the possibility of creating* indirect recursion in a program.
- This occurs when function A calls function B, which in turn calls function A.
- There can even be several functions involved in the recursion.
- For example, function A could call function B, which could call function C, which calls function A.

Summing a Range of List Elements with Recursion

```
1  # This program demonstrates the range_sum function.
2
3  def main():
4      # Create a list of numbers.
5      numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9]
6
7      # Get the sum of the items at indexes 2
8      # through 5.
9      my_sum = range_sum(numbers, 2, 5)
10
11     # Display the sum.
12     print('The sum of items 2 through 5 is', my_sum)
13
14 # The range_sum function returns the sum of a specified
15 # range of items in num_list. The start parameter
16 # specifies the index of the starting item. The end
17 # parameter specifies the index of the ending item.
18 def range_sum(num_list, start, end):
19     if start > end:
20         return 0
21     else:
22         return num_list[start] + range_sum(num_list, start + 1, end)
23
24 # Call the main function.
25 main()
```

Program Output

```
The sum of elements 2 through 5 is 18
```

The Fibonacci Series

```
1   # This program uses recursion to print numbers
2   # from the Fibonacci series.
3
4   def main():
5       print('The first 10 numbers in the')
6       print('Fibonacci series are:')
7
8       for number in range(1, 11):
9           print(fib(number))
10
11  # The fib function returns the nth number
12  # in the Fibonacci series.
13  def fib(n):
14      if n == 0:
15          return 0
16      elif n == 1:
17          return 1
18      else:
19          return fib(n - 1) + fib(n - 2)
20
21  # Call the main function.
22  main()
```

Program Output

```
The first 10 numbers in the
Fibonacci series are:
1
```

```
1  # This program uses recursion to find the GCD
2  # of two numbers.
3
4  def main():
5      # Get two numbers.
6      num1 = int(input('Enter an integer: '))
7      num2 = int(input('Enter another integer: '))
8
9      # Display the GCD.
10     print('The greatest common divisor of')
11     print('the two numbers is', gcd(num1, num2))
12
13 # The gcd function returns the greatest common
14 # divisor of two numbers.
15 def gcd(x, y):
16     if x % y == 0:
17         return y
18     else:
19         return gcd(x, x % y)
20
21 # Call the main function.
22 main()
```

Recursion versus Looping

- Any algorithm that can be coded with recursion can also be coded with a loop.
- Both approaches achieve repetition, but which is best to use?
- There are several reasons not to use recursion. Recursive function calls are certainly less efficient than loops.
- Each time a function is called, the system incurs overhead that is not necessary with a loop.
- Also, in many cases, a solution using a loop is more evident than a recursive solution. In fact, the majority of repetitive programming tasks are best done with loops.
- Some problems, however, are more easily solved with recursion than with a loop.
- For example, the mathematical definition of the GCD formula is well suited to a recursive approach.
- If a recursive solution is evident for a particular problem, and the recursive algorithm does not slow system performance an intolerable amount, then recursion would be a good design choice.
- If a problem is more easily solved with a loop, however, you should take that approach.

References

- Tony Gaddis, “Starting out with Python”, 3rd Edition, Global Edition, Pearson Education