Chapter 7: Lists and Tuples

7.1 Sequences

Concept:

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.

Two of the fundamental sequence types: lists and tuples.

A list is mutable, which means that a program can change its contents.

A tuple is immutable, which means that once it is created, its contents cannot be changed.

7.2 Introduction to Lists

Concept:

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.

A *list* is an object that contains multiple data items. Each item that is stored in a list is called an *element*.

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.

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

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

Printing:

```
numbers = [5, 10, 15, 20]
print(numbers)
```

Output:

```
[5, 10, 15, 20]
```

A built-in list() function converts certain types of objects to lists.

Examples:

numbers = list(range(5))

- The list [0, 1, 2, 3, 4] is assigned to the numbers variable.

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

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

The Repetition Operator

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.

The general format:

*list * n*

Example:

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

Iterating over a List with the for Loop

Example:

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

Output:

```
99
100
101
102
```

Indexing

Accessing the individual elements in a list 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.

Example:

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

Printing the elements:

```
print(my_list[0], my_list[1], my_list[2], my_list[3])
```

or:

```
index = 0
while index < 4:
    print(my_list[index])
    index += 1
```

Negative indexes:

Example:

```
my_list = [10, 20, 30, 40]
print(my_list[-1], my_list[-2], my_list[-3], my_list[-4])
```

Output:

```
40   30   20   10
```

The `len` Function

Python has a built-in function named len that returns the length of a sequence, such as a list.

Example:

```
my_list = [10, 20, 30, 40]
size = len(my_list)
```

The function returns the value 4, which is the number of elements in the list. This value is assigned to the size variable.

Lists Are Mutable

Lists in Python are *mutable*, which means their elements can be changed.

Example:

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

Output:

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

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

Concatenating Lists

To concatenate means to join two things together. You can use the + operator to concatenate two lists.

Example:

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

list3 references   [1, 2, 3, 4, 5, 6, 7, 8]

You can also use the += augmented assignment operator to concatenate one list to another.

Example:

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

list1 references   [1, 2, 3, 4, 5, 6, 7, 8]

7.3 List Slicing

Concept:

A slicing expression selects a range of elements from a sequence.

We 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.

Example:

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']

mid_days = days[2:5]
```

The mid_days variable references   ['Tuesday', 'Wednesday', 'Thursday']

Options for slicing:

- If you leave out the *start* index in a slicing expression, Python uses 0 as the starting index.
- If you leave out the *end* index in a slicing expression, Python uses the length of the list as the *end* index.
- If you leave out both the start and end index in a slicing expression, you get a copy of the entire list.
- Slicing expressions can also have step value, which can cause elements to be skipped 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.

7.4 Finding Items in Lists with the in Operator

Concept:

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.

The general format:

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.

Example:

```
 def main():
 prod_nums = ['V475', 'F987', 'Q143', 'R688']

search = input('Enter a product number: ')

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

main()
```

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

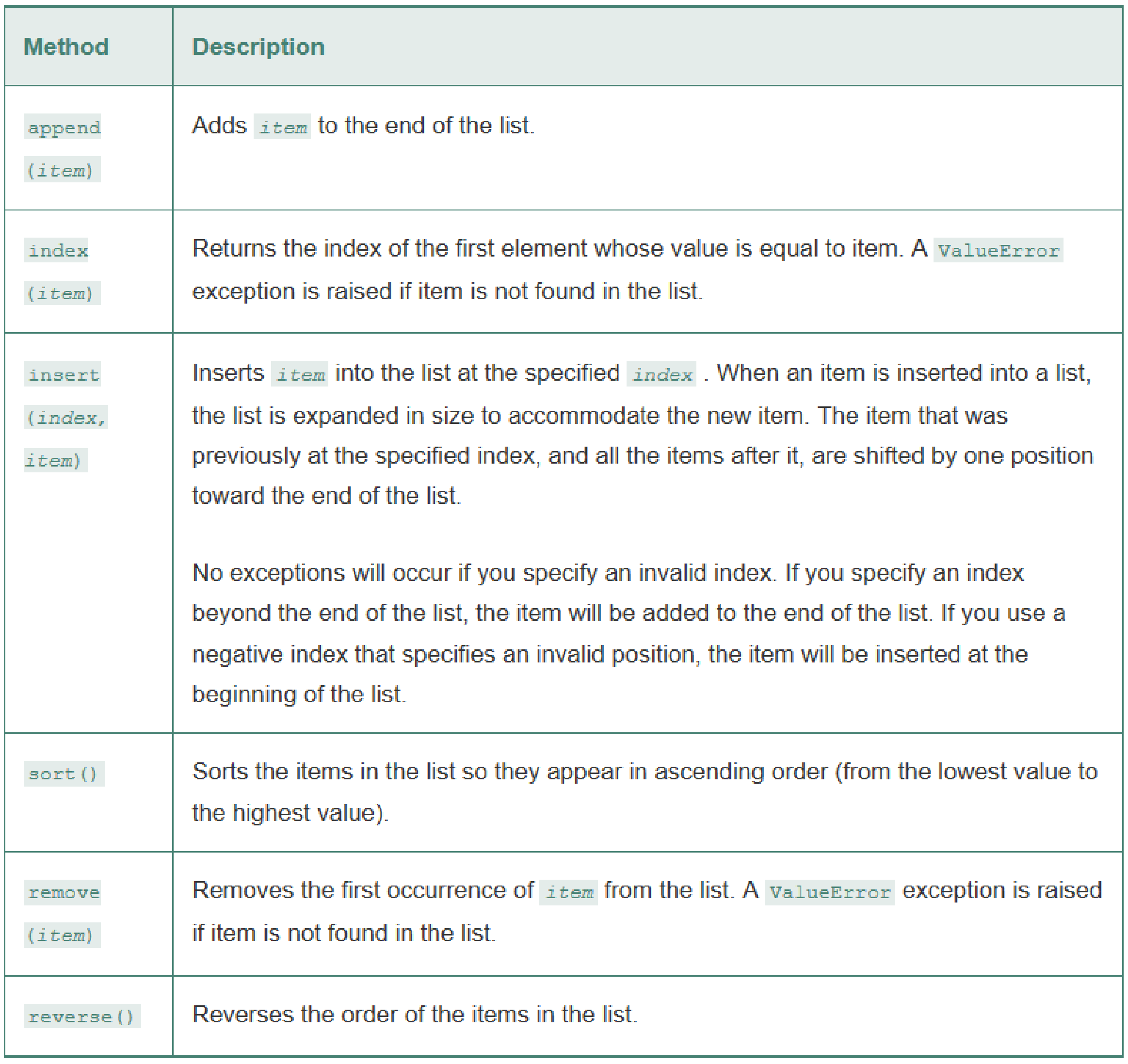
7.5 List Methods and Useful Built-in Functions

Concept:

Lists have numerous methods that allow you to work with the elements that they contain. Python also provides some built-in functions that are useful for working with lists.

Lists have numerous methods that allow you to add elements, remove elements, change the ordering of elements, and so forth.

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.

The append Method

The append method is commonly used to add items to a list. The item that is passed as an argument is appended to the end of the list's existing elements.

The index Method

Sometimes you need to know not only whether an item is in a list, but where it is located. You pass an argument to the index method, and it returns the index of the first element in the list containing that item.

The insert Method

The insert method allows you to insert an item into a list at a specific position. You pass two arguments to the insert method: an index specifying where the item should be inserted and the item that you want to insert.

The sort Method

The sort method rearranges the elements of a list so they appear in ascending order (from the lowest value to the highest value).

Here is an example:

```
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]
```

The remove Method

The remove method removes an item from the list. You pass an item to the method as an argument, and the first element containing that item is removed. This reduces the size of the list by one element. All of the elements after the removed element are shifted one position toward the beginning of the list. A ValueError exception is raised if the item is not found in the list

The reverse Method

The reverse method simply reverses the order of the items in the list.

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]
```

The del Statement

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

Example:

```
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.

Example:

```
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.

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
```

7.6 Copying Lists

Concept:

To make a copy of a list, you must copy the list’s elements.

Recall that in Python, assigning one variable to another variable simply makes both variables reference the same object in memory. For example, look at the following code:

```
# Create a list.
list1 = [1, 2, 3, 4]
# Assign the list to the list2 variable.
list2 = list1
```

or:

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. Here is an example:

```
# 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)
```

or:

A simpler and more elegant way to accomplish the same task is to use the concatenation operator, as shown here:

```
# Create a list with values.
list1 = [1, 2, 3, 4]
# Create a copy of list1.
list2 = [] + list1
```

Working with Lists and Files

Some tasks may require you to save the contents of a list to a file so the data can be used at a later time. Likewise, some situations may require you to read the data from a file into a list. For example, suppose you have a file that contains a set of values that appear in random order and you want to sort the values. One technique for sorting the values in the file would be to read them into a list, call the list's sort method, and then write the values in the list back to the file.

Saving the contents of a list to a file is a straightforward procedure. In fact, Python file objects have a method named writelines that writes an entire list to a file. A drawback to the writelines method, however, is that it does not automatically write a newline ('\n') at the end of each item.

An alternative approach is to use the for loop to iterate through the list, writing each element with a terminating newline character.

7.8 Two-Dimensional Lists

Concept:

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

The elements of a list can be virtually anything, including other lists.

7.9 Tuples

Concept:

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, as shown in the following interactive session:

```
>>> my_tuple = (1, 2, 3, 4, 5) Enter
>>> print(my_tuple) Enter
(1, 2, 3, 4, 5)
>>>
```

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.

Converting Between Lists and Tuples

You can use the built-in list() function to convert a tuple to a list and the built-in tuple() function to convert a list to a tuple. The following interactive session demonstrates:

```
>>> number_tuple = (1, 2, 3) Enter
>>> number_list = list(number_tuple) Enter
>>> print(number_list) Enter
[1, 2, 3]
>>> str_list = ['one', 'two', 'three'] Enter
>>> str_tuple = tuple(str_list) Enter
>>> print(str_tuple) Enter
('one', 'two', 'three')
>>>
```