6

Lists

AIM

Understand the role of lists in Python in storing multiple items of different types and perform manipulations using various methods.

LEARNING OUTCOMES

At the end of the chapter, you are expected to

- Create and manipulate items in lists.
- Comprehend Indexing and slicing in lists.
- Use methods associated with lists.
- Using lists as arguments in functions.
- Use *for* loop to access individual items in lists.

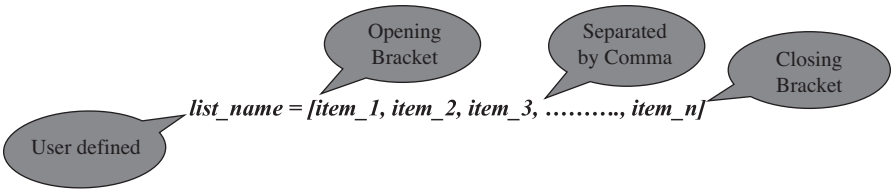
Most of the times a variable can hold a single value. However, in many cases, you need to assign more than that. Consider a Forest scenario where animals belonging to different types of families live. Some of them like Lion, Tiger, Cheetah are Carnivorous and others like Monkeys, Elephants and Buffalos are Herbivorous. If you want to define this habitat computationally, then you have to create multiple variables to represent each animal belonging to a particular animal family which may not make much sense. Instead, you can combine all the animals belonging to the particular animal family under one variable and then use this variable name for further manipulation.

You can think of the list as a container that holds a number of items. Each element or value that is inside a list is called an item. All the items in a list are assigned to a single variable. Lists avoid having a separate variable to store each item which is less efficient and more error prone when you have to perform some operations on these items. Lists can be simple or nested lists with varying types of values. Lists are one of the most flexible data storage formats in Python because they can have values added, removed, and changed.

6.1 Creating Lists

Lists are constructed using square brackets [ ] wherein you can include a list of items separated by commas.

The syntax for creating list is,



1. >>> superstore = ["metro", "tesco", "walmart", "kmart", "carrefour"]
2. >>> superstore
   ['metro', 'tesco', 'walmart', 'kmart', 'carrefour']

In ① each item in the list is a string. The contents of the list variable are displayed by executing the list variable name ②. When you print out the list, the output looks exactly like the list you had created.

You can create an empty list without any items. The syntax is,

list_name = [ ]

For example,

1. >>> number_list = [4, 4, 6, 7, 2, 9, 10, 15]
2. >>> mixed_list = ['dog', 87.23, 65, [9, 1, 8, 1]]
3. >>> type(mixed_list)
   <class 'list'>
4. >>> empty_list = []
5. >>> empty_list
   []
6. >>> type(empty_list)
   <class 'list'>

Here, *number_list* ① contains items of the same type while in *mixed_list* ② the items are a mix of type string, float, integer and another list itself. You can determine the type of a *mixed_list* ③ variable by passing the variable name as an argument to *type()* function. In Python, the list type is called as *list*. An empty list can be created as shown in ④–⑤ and the variable *empty_list* is of *list* type ⑥.

You can store any item in a list like string, number, object, another variable and even another list. You can have a mix of different item types and these item types need not have to be homogeneous. For example, you can have a list which is a mix of type numbers, strings and another list itself.

6.2 Basic List Operations

In Python, lists can also be concatenated using the + sign, and the * operator is used to create a repeated sequence of list items. For example,

```
1. >>> list_1 = [1, 3, 5, 7]
2. >>> list_2 = [2, 4, 6, 8]
3. >>> list_1 + list_2
   [1, 3, 5, 7, 2, 4, 6, 8]
4. >>> list_1 * 3
   [1, 3, 5, 7, 1, 3, 5, 7, 1, 3, 5, 7]
5. >>> list_1 == list_2
   False
```

Two lists containing numbers as items are created ①–②. The *list_1* and *list_2* lists are added to form a new list. The new list has all items of both the lists ③. You can use the multiplication operator on the list. It repeats the items the number of times you specify and in ④ the *list_1* contents are repeated three times. Contents of the lists *list_1* and *list_2* are compared using the == operator ⑤ and the result is Boolean *False* since the items in both the lists are different.

You can check for the presence of an item in the list using *in* and *not in* membership operators. It returns a Boolean *True* or *False*. For example,

```
1. >>> list_items = [1,3,5,7]
2. >>> 5 in list_items
   True
3. >>> 10 in list_items
   False
```

If an item is present in the list then using *in* operator results in *True* ② else returns *False* Boolean value ③.

6.2.1 The *list()* Function

The built-in *list()* function is used to create a list. The syntax for *list()* function is,

list([sequence])

where the sequence can be a string, tuple or list itself. If the optional sequence is not specified then an empty list is created. For example,

```
1. >>> quote = "How you doing?"
2. >>> string_to_list = list(quote)
3. >>> string_to_list
   ['H', 'o', 'w', ' ', 'y', 'o', 'u', ' ', 'd', 'o', 'i', 'n', 'g', '?']
4. >>> friends = ["j", "o", "e", "y"]
```

5. >>> friends + quote
   Traceback (most recent call last):
     File "<stdin>", line 1, in <module>
   TypeError: can only concatenate list (not "str") to list
6. >>> friends + list(quote)
   ['j', 'o', 'e', 'y', 'H', 'o', 'w', ' ', 'y', 'o', 'u', ' ', 'd', 'o', 'i', 'n', 'g', '?']

The string variable *quote* ① is converted to a list using the *list()* function ②. Now, let's see whether a string can be concatenated with a list ⑤. The result is an Exception which says *TypeError: can only concatenate list (not "str") to list*. That means you cannot concatenate a list with a string value. In order to concatenate a string with the list, you must convert the string value to list type, using Python built-in *list()* function ⑥.

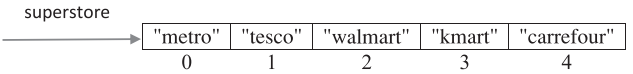
6.3 Indexing and Slicing in Lists

As an ordered sequence of elements, each item in a list can be called individually, through indexing. The expression inside the bracket is called the index. Lists use square brackets [ ] to access individual items, with the first item at index 0, the second item at index 1 and so on. The index provided within the square brackets indicates the value being accessed.

The syntax for accessing an item in a list is,

list_name[index]

where index should always be an integer value and indicates the item to be selected. For the list *superstore*, the index breakdown is shown below.

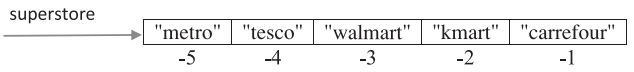


1. >>> superstore = ["metro", "tesco", "walmart", "kmart", "carrefour"]
2. >>> superstore[0]
   'metro'
3. >>> superstore[1]
   'tesco'
4. >>> superstore[2]
   'walmart'
5. >>> superstore[3]
   'kmart'
6. >>> superstore[4]
   'carrefour'
7. >>> superstore[9]
   Traceback (most recent call last):
     File "<stdin>", line 1, in <module>
   IndexError: list index out of range

The *superstore* list has five items. To print the first item in the list use square brackets immediately after list name with an index value of zero ②. The index numbers for this *superstore* list range from 0 to 4 ②–⑥. If the index value is more than the number of items in the list ⑦ then it results in *"IndexError: list index out of range"* error.

In addition to positive index numbers, you can also access items from the list with a negative index number, by counting backwards from the end of the list, starting at –1. Negative indexing is useful if you have a long list and you want to locate an item towards the end of a list.

For the same list *superstore*, the negative index breakdown is shown below.




```
1. >>> superstore[-3]
   'walmart'
```

If you would like to print out the item 'walmart' by using its negative index number, you can do so as in ①.

6.3.1 Modifying Items in Lists

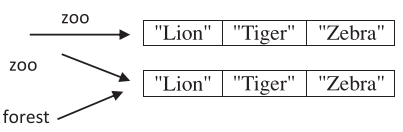
Lists are mutable in nature as the list items can be modified after you have created a list. You can modify a list by replacing the older item with a newer item in its place and without assigning the list to a completely new variable. For example,

```
1. >>> fauna = ["pronghorn", "alligator", "bison"]
2. >>> fauna[0] = "groundhog"
3. >>> fauna
   ['groundhog', 'alligator', 'bison']
4. >>> fauna[2] = "skunk"
5. >>> fauna
   ['groundhog', 'alligator', 'skunk']
6. >>> fauna[-1] = "beaver"
7. >>> fauna
   ['Groundhog', 'alligator', 'beaver']
```

You can change the string item at index 0 from 'pronghorn' to 'groundhog' as shown in ②. Now when you display *fauna*, the list items will be different ③. The item at index 2 is changed from "bison" to "skunk" ④. You can also change the value of an item by using a negative index number –1 ⑥ which corresponds to the positive index number of 2. Now "skunk" is replaced with "beaver" ⑦.

When you assign an existing list variable to a new variable, an assignment (=) on lists does not make a new copy. Instead, assignment makes both the variable names point to the same list in memory. For example,

```
1. >>> zoo = ["Lion", "Tiger", "Zebra"]
2. >>> forest = zoo
3. >>> type(zoo)
```

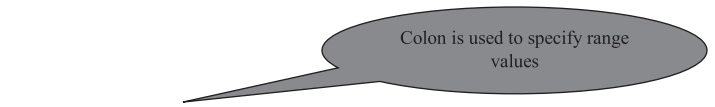


```
    <class 'list'>
4. >>> type(forest)
    <class 'list'>
5. >>> forest
    ['Lion', 'Tiger', 'Zebra']
6. >>> zoo[0] = "Fox"
7. >>> zoo
    ['Fox', 'Tiger', 'Zebra']
8. >>> forest
    ['Fox', 'Tiger', 'Zebra']
9. >>> forest[1] = "Deer"
10. >>> forest
    ['Fox', 'Deer', 'Zebra']
11. >>> zoo
    ['Fox', 'Deer', 'Zebra']
```

The above code proves that assigning an existing list variable to a new variable does not create a new copy of the existing list items ①–⑪.

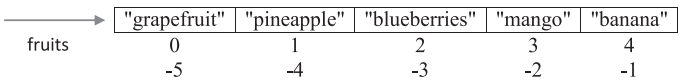
Slicing of lists is allowed in Python wherein a part of the list can be extracted by specifying index range along with the colon (:) operator which itself is a list.

The syntax for list slicing is,



list_name[start:stop[:step]]

where both *start* and *stop* are integer values (positive or negative values). List slicing returns a part of the list from the *start* index value to *stop* index value which includes the *start* index value but excludes the *stop* index value. *Step* specifies the increment value to slice by and it is optional. For the list *fruits*, the positive and negative index breakdown is shown below.




```
1. >>> fruits = ["grapefruit", "pineapple", "blueberries", "mango", "banana"]
2. >>> fruits[1:3]
    ['pineapple', 'blueberries']
3. >>> fruits[:3]
    ['grapefruit', 'pineapple', 'blueberries']
4. >>> fruits[2:]
    ['blueberries', 'mango', 'banana']
5. >>> fruits[1:4:2]
    ['pineapple', 'mango']
```

6. >>> fruits[:]
   ['grapefruit', 'pineapple', 'blueberries', 'mango', 'banana']
7. >>> fruits[::2]
   ['grapefruit', 'blueberries', 'banana']
8. >>> fruits[::-1]
   ['banana', 'mango', 'blueberries', 'pineapple', 'grapefruit']
9. >>> fruits[-3:-1]
   ['blueberries', 'mango']

All the items in the *fruits* list starting from an index value of 1 up to index value of 3 but excluding the index value of 3 is sliced ②. If you want to access the *start* items, then there is no need to specify the index value of zero. You can skip the *start* index value and specify only the *stop* index value ③. Similarly, if you want to access the last *stop* items, then there is no need to specify the *stop* value and you have to mention only the *start* index value ④. When *stop* index value is skipped the range of items accessed extends up to the last item. If you skip both the *start* and *stop* index values ⑥ and specify only the colon operator within the brackets, then the entire items in the list are displayed. The number after the second colon tells Python that you would like to choose your slicing increment. By default, Python sets this increment to 1, but that number after second colon allows you to specify what you want it to be. Using double colon as shown in ⑦ means no value for *start* index and no value for *stop* index and jump the items by two steps. Every second item of the list is extracted starting from an index value of zero. All the items in a list can be displayed in reverse order by specifying a double colon followed by an index value of –1 ⑧. Negative index values can also be used for *start* and *stop* index values ⑨.

6.4 Built-In Functions Used on Lists

There are many built-in functions for which a list can be passed as an argument (TABLE 6.1).

TABLE 6.1

Built-In Functions Used on Lists

Built-In Functions	Description
len()	The *len()* function returns the numbers of items in a list.
sum()	The *sum()* function returns the sum of numbers in the list.
any()	The *any()* function returns *True* if any of the Boolean values in the list is *True*.
all()	The *all()* function returns *True* if all the Boolean values in the list are *True*, else returns *False*.
sorted()	The *sorted()* function returns a modified copy of the list while leaving the original list untouched.

For example,

1. >>> lakes = ['superior', 'erie', 'huron', 'ontario', 'powell']
2. >>> len(lakes)
   5

```
3. >>> numbers = [1, 2, 3, 4, 5]
4. >>> sum(numbers)
   15
5. >>> max(numbers)
   5
6. >>> min(numbers)
   1
7. >>> any([1, 1, 0, 0, 1, 0])
   True
8. >>> all([1, 1, 1, 1])
   True
9. >>> lakes_sorted_new = sorted(lakes)
10. >>> lakes_sorted_new
    ['erie', 'huron', 'ontario', 'powell', 'superior']
```

You can find the number of items in the list *lakes* using the *len()* function ②. Using the *sum()* function results in adding up all the numbers in the list ④. Maximum and minimum numbers in a list are returned using *max()* ⑤ and *min()* functions ⑥. If any of the items is 1 in the list then *any()* function returns Boolean *True* value ⑦. If all the items in the list are 1 then *all()* function returns *True* else returns *False* Boolean value ⑧. The *sorted()* function returns the sorted list of items without modifying the original list which is assigned to a new list variable ⑨. In the case of string items in the list, they are sorted based on their ASCII values.

6.5 List Methods

The list size changes dynamically whenever you add or remove the items and there is no need for you to manage it yourself. You can get a list of all the methods (TABLE 6.2) associated with the *list* by passing the list function to *dir()*.

```
1. >>> dir(list)
   ['__add__', '__class__', '__contains__', '__delattr__', '__delitem__', '__dir__', '__doc__',
   '__eq__', '__format__', '__ge__', '__getattribute__', '__getitem__', '__gt__', '__hash__',
   '__iadd__', '__imul__', '__init__', '__init_subclass__', '__iter__', '__le__', '__len__',
   '__lt__', '__mul__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__',
   '__reversed__', '__rmul__', '__setattr__', '__setitem__', '__sizeof__', '__str__', '__sub-
   classhook__', 'append', 'clear', 'copy', 'count', 'extend', 'index', 'insert', 'pop', 'remove',
   'reverse', 'sort']
```

Various methods associated with *list* is displayed ①.

TABLE 6.2

Various List Methods

List Methods	Syntax	Description
append()	list.append(item)	The *append()* method adds a single item to the end of the list. This method does not return new list and it just modifies the original.
count()	list.count(item)	The *count()* method counts the number of times the item has occurred in the list and returns it.
insert()	list.insert(index, item)	The *insert()* method inserts the item at the given index, shifting items to the right.
extend()	list.extend(list2)	The *extend()* method adds the items in list2 to the end of the list.
index()	list.index(item)	The *index()* method searches for the given item from the start of the list and returns its index. If the value appears more than once, you will get the index of the first one. If the item is not present in the list then *ValueError* is thrown by this method.
remove()	list.remove(item)	The *remove()* method searches for the first instance of the given item in the list and removes it. If the item is not present in the list then *ValueError* is thrown by this method.
sort()	list.sort()	The *sort()* method sorts the items *in place* in the list. This method modifies the original list and it does not return a new list.
reverse()	list.reverse()	The *reverse()* method reverses the items *in place* in the list. This method modifies the original list and it does not return a new list.
pop()	list.pop([index])	The *pop()* method removes and returns the item at the given index. This method returns the rightmost item if the index is omitted.

Note: Replace the word *"list"* mentioned in the syntax with your *actual list name* in your code.

For example,

```
1. >>> cities = ["oslo", "delhi", "washington", "london", "seattle", "paris", "washington"]
2. >>> cities.count('seattle')
   1
3. >>> cities.index('washington')
   2
4. >>> cities.reverse()
5. >>> cities
   ['washington', 'paris', 'seattle', 'london', 'washington', 'delhi', 'oslo']
6. >>> cities.append('brussels')
7. >>> cities
   ['washington', 'paris', 'seattle', 'london', 'washington', 'delhi', 'oslo', 'brussels']
8. >>> cities.sort()
9. >>> cities
   ['brussels', 'delhi', 'london', 'oslo', 'paris', 'seattle', 'washington', 'washington']
10. >>> cities.pop()
    'washington'
```

```
11. >>> cities
    ['brussels', 'delhi', 'london', 'oslo', 'paris', 'seattle', 'washington']
12. >>> more_cities = ["brussels", "copenhagen"]
13. >>> cities.extend(more_cities)
14. >>> cities
    ['brussels', 'delhi', 'london', 'oslo', 'paris', 'seattle', 'washington', 'brussels', 'copenhagen']
15. >>> cities.remove("brussels")
16. >>> cities
    ['delhi', 'london', 'oslo', 'paris', 'seattle', 'washington', 'brussels', 'copenhagen']
```

Various operations on lists are carried out using list methods ①–⑯.

6.5.1 Populating Lists with Items

One of the popular way of populating lists is to start with an empty list [ ], then use the functions *append()* or *extend()* to add items to the list. For example,

```
1. >>> continents = []
2. >>> continents.append("Asia")
3. >>> continents.append("Europe")
4. >>> continents.append("Africa")
5. >>> continents
   ['Asia', 'Europe', 'Africa']
```

Create an empty list *continents* ① and start populating items to the *continents* list using append() function ②–④ and finally display the items of the *continents* list ⑤.

Program 6.1: Program to Dynamically Build User Input as a List

```
1. list_items = input("Enter list items separated by a space ").split()
2. print(f"List items are {list_items}")
3. items_of_list = []
4. total_items = int(input("Enter the number of items "))
5. for i in range(total_items):
6.     item = input("Enter list item: ")
7.     items_of_list.append(item)
8. print(f"List items are {items_of_list}")
```

Output

```
Enter list items separated by a space Asia Europe Africa
List items are ['Asia', 'Europe', 'Africa']
```

```
Enter the number of items 2
Enter list item: Australia
Enter list item: Americas
List items are ['Australia', 'Americas']
```

You can build a list from user-entered values in two ways. In the first method ①–② you chain *input()* and *split()* together using dot notation. The user has to enter the items separated by spaces. Even though the user entered input has multiple items separated by spaces, by default the user-entered input is considered a single string. You benefit from the fact that the *split()* method can be used with the string. This user-entered input string value is split based on spaces and the *split()* method returns a list of string items. The advantage of this method is, there is no need for you to specify the total number of items that you are planning to insert into the list. In the second method ③–⑧ you have to specify the total number of items that you are planning to insert into the list beforehand itself ④. Based on this number we iterate through the *for* loop as many times using *range()* function ⑤. During each iteration, you need to append ⑦ the user entered value to the list ⑥.

6.5.2 Traversing of Lists

Using a *for* loop you can iterate through each item in a list.

Program 6.2: Program to Illustrate Traversing of Lists Using the *for* loop

```
1. fast_food = ["waffles", "sandwich", "burger", "fries"]
2. for each_food_item in fast_food:
3.     print(f"I like to eat {each_food_item}")
4. for each_food_item in ["waffles", "sandwich", "burger", "fries"]:
5.     print(f"I like to eat {each_food_item}")
```

Output

```
I like to eat waffles
I like to eat sandwich
I like to eat burger
I like to eat fries
I like to eat waffles
I like to eat sandwich
I like to eat burger
I like to eat fries
```

The *for* statement makes it easy to loop over the items in a list. A list variable is created ① and the list variable is specified in the *for* loop ②. Instead of specifying a list variable, you can specify a list directly in the *for* loop ④.

You can obtain the index value of each item in the list by using *range()* along with *len()* function.

Program 6.3: Program to Display the Index Values of Items in List

```
1. silicon_valley = ["google", "amd", "yahoo", "cisco", "oracle"]
2. for index_value in range(len(silicon_valley)):
3.     print(f"The index value of '{silicon_valley[index_value]}' is {index_value}")
```

Output

```
The index value of 'google' is 0
The index value of 'amd' is 1
The index value of 'yahoo' is 2
The index value of 'cisco' is 3
The index value of 'oracle' is 4
```

You need to pass the list name as an argument to *len()* function and the resulting value will be passed as an argument to *range()* function to obtain the index value ② of each item in the list ①.

Program 6.4: Write Python Program to Sort Numbers in a List in Ascending Order Using Bubble Sort by Passing the List as an Argument to the Function Call

```
1. def bubble_sort(list_items):
2.     for i in range(len(list_items)):
3.         for j in range(len(list_items)-i-1):
4.             if list_items[j] > list_items[j+1]:
5.                 temp = list_items[j]
6.                 list_items[j] = list_items[j+1]
7.                 list_items[j+1] = temp
8.     print(f"The sorted list using Bubble Sort is {list_items}")
9. def main():
10.    items_to_sort = [5, 4, 3, 2, 1]
11.    bubble_sort(items_to_sort)
12. if __name__ == "__main__":
13.    main()
```

Output

```
The sorted list using Bubble Sort is [1, 2, 3, 4, 5]
```

Bubble sort is an elementary sorting algorithm that repeatedly steps through the items in the list to be sorted by comparing each item with its successor item and swaps them if they are in the wrong order (FIGURE 6.1). The pass through the list is repeated until no swaps are needed, which indicates that the list is sorted. Bubble sort requires a maximum of n – 1 passes if there are n items in the list. To sort the items in the list we require two loops. One for running through the passes ② and another for comparing

1st Pass, Index Value 0	2nd Pass, Index Value 1	3rd Pass, Index Value 2	4th Pass, Index Value 3
(5,4,3,2,1) → (4,**5**,3,2,1) (4,5,3,2,1) → (4,3,**5**,2,1) (4,3,5,2,1) → (4,3,2,**5**,1) (4,3,2,5,1) → (4,3,2,1,**5**)	(4,3,2,1,5) → (3,**4**,2,1,5) (3,4,2,1,5) → (3,2,**4**,1,5) (3,2,4,1,5) → (3,2,1,**4**,5)	(3,2,1,4,5) → (2,**3**,1,4,5) (2,3,1,4,5) → (2,1,**3**,4,5)	(2,1,3,4,5) → (1,**2**,3,4,5)

FIGURE 6.1
Sorting steps in Bubble Sort algorithm.

consecutive items in each pass ③. The indexing value for each item in the list is obtained by *range(len(list_items)-i-1)*. While traversing the list using *for* loop, swap ⑤–⑦ if the current item is found to be greater than the next item ④. You can also pass a list ⑩ as an argument to the function call ⑪ which is assigned to the *list_items* parameter in the function definition ①.

Program 6.5: Write Python Program to Conduct a Linear Search for a Given Key Number in the List and Report Success or Failure

```
1. def read_key_item():
2.     key_item = int(input("Enter the key item to search: "))
3.     return key_item

4. def linear_search(search_key):
5.     list_of_items = [10, 20, 30, 40, 50]
6.     found = False
7.     for item_index in range(len(list_of_items)):
8.         if list_of_items[item_index] == search_key:
9.             found = True
10.            break
11.    if found:
12.        print(f"{search_key} found at position {item_index + 1}")
13.    else:
14.        print("Item not found in the list")

15. def main():
16.    key_in_list = read_key_item()
17.    linear_search(key_in_list)

18. if __name__ == "__main__":
19.    main()
```

Output

```
Enter the key item to search: 50
50 found at position 5
```

Linear search is a method for finding a key item in a list. It sequentially checks each item of the list for the key value until a match is found or until all the items have been searched. The function *read_key_item()* is used to read search key from user ①–③. The function *linear_search()* ④ searches for a key item in the list and reports success or failure. A *for* loop is required to run through all the items in the list ⑦. An *if* condition is used to check for the presence of the key item in the list ⑧. Here the variable *found* keeps track of the presence of the item in the list. Initially *found* is set to False ⑥. If the key item is present, then the variable *found* is assigned with Boolean *True* ⑨. Then the key item along with its position is displayed ⑫.

Program 6.6: Input Five Integers (+ve and −ve). Find the Sum of Negative Numbers, Positive Numbers and Print Them. Also, Find the Average of All the Numbers and Numbers Above Average

```
1. def find_sum(list_items):
2.     positive_sum = 0
3.     negative_sum = 0
4.     for item in list_items:
5.         if item > 0:
6.             positive_sum = positive_sum + item
7.         else:
8.             negative_sum = negative_sum + item
9.     average = (positive_sum + negative_sum) / 5
10.    print(f"Sum of Positive numbers in list is {positive_sum}")
11.    print(f"Sum of Negative numbers in list is {negative_sum}")
12.    print(f"Average of item numbers in list is {average}")
13.    print("Items above average are")
14.    for item in list_items:
15.        if item > average:
16.            print(item)
17. def main():
18.    find_sum([-1, -2, -3, 4, 5])
19. if __name__ == "__main__":
20.    main()
```

Output

```
Sum of Positive numbers in list is 9
Sum of Negative numbers in list is -6
Average of item numbers in list is 0.6
Items above average are
4
5
```

Traverse through the list of five items of integer type ④. If a number is greater than zero then it is positive or if a number is less than zero then it is negative ⑤. Sum of positive

and negative numbers are stored in the variables *positive_sum* ⑥ and *negative_sum* ⑧ respectively. Traverse through each item in the list and based on positive and negative numbers, add to the variables *positive_sum* and *negative_sum* and print it ⑩–⑪. Average is calculated by (positive_sum + negative_sum) / 5 ⑨ and print it ⑫. After getting the *average* value, traverse through each item in the list ⑭ and check whether the numbers are above average ⑮. If so, print those numbers ⑯.

Program 6.7: Check If the Items in the List Are Sorted in Ascending or Descending Order and Print Suitable Messages Accordingly. Otherwise, Print "Items in list are not sorted"

```
1. def check_for_sort_order(list_items):
2.     ascending = descending = True
3.     for i in range(len(list_items) - 1):
4.         if list_items[i] < list_items[i+1]:
5.             descending = False
6.         if list_items[i] > list_items[i+1]:
7.             ascending = False
8.     if ascending:
9.         print("Items in list are in Ascending order")
10.    elif descending:
11.        print("Items in list are in Descending order")
12.    else:
13.        print("Items in list are not sorted")
14. def main():
15.    check_for_sort_order([1, 4, 2, 5, 3])
16. if __name__ == "__main__":
17.    main()
```

Output

Items in list are not sorted

In the beginning, it is assumed that the list is sorted either in ascending order or in descending order by setting the variables *ascending* and *descending* to Boolean *True* value ②. Now traverse through each item of the list ③. If the first item is less than the second item, then the list is not in descending order, so set the variable descending to *False* Boolean value. If the second item is greater than the third item, then the list is not in ascending order, so set the variable *ascending* to *False* Boolean value ④–⑦. By applying the above logic to each item in the list results in *ascending* variable is set to *True* and *descending* variable is set to a *False* Boolean value for items stored in ascending order. If the list items are in descending order, then the *descending* variable is set to *True* while *ascending* variable is set to a *False* Boolean value. If items are not in any order in the list, then both *ascending* and *descending* variables are set to *False*.

Program 6.8: Find Mean, Variance and Standard Deviation of List Numbers

```
1. import math
2. def statistics(list_items):
3.     mean = sum(list_items)/len(list_items)
4.     print(f"Mean is {mean}")
5.     variance = 0
6.     for item in list_items:
7.         variance += (item-mean)**2
8.     variance /= len(list_items)
9.     print(f"Variance is {variance}")
10.    standard_deviation = math.sqrt(variance)
11.    print(f"Standard Deviation is {standard_deviation}")
12. def main():
13.    statistics([1, 2, 3, 4])
14. if __name__ == "__main__":
15.    main()
```

Output

```
Mean is 2.5
Variance is 1.25
Standard Deviation is 1.118033988749895
```

Mean is calculated as the sum of all items in the list / total number of items in the list ③. Variance is defined as the average of the squared differences from the mean. Steps to find the *variance* is to first calculate the *mean* ③–④, then traverse through each element in the list and subtract it with *mean* and square the result which is also called as the squared difference. Finally, find the mean of those squared differences ⑥–⑧ to get the variance ⑨. Standard deviation is the square root of variance ⑩–⑪.

Program 6.9: Write Python Program to Implement Stack Operations

```
1. stack = []
2. stack_size = 3
3. def display_stack_items():
4.     print("Current stack items are: ")
5.     for item in stack:
6.         print(item)
7. def push_item_to_stack(item):
8.     print(f"Push an item to stack {item}")
```

```
    if len(stack) < stack_size:
        stack.append(item)
    else:
        print("Stack is full!")

def pop_item_from_stack():
    if len(stack) > 0:
        print(f"Pop an item from stack {stack.pop()}")
    else:
        print("Stack is empty.")

def main():
    push_item_to_stack(1)
    push_item_to_stack(2)
    push_item_to_stack(3)
    display_stack_items()
    push_item_to_stack(4)
    pop_item_from_stack()
    display_stack_items()
    pop_item_from_stack()
    pop_item_from_stack()
    pop_item_from_stack()

if __name__ == "__main__":
    main()
```

Output

```
Push an item to stack 1
Push an item to stack 2
Push an item to stack 3
Current stack items are:
1
2
3
Push an item to stack 4
Stack is full!
Pop an item from stack 3
Current stack items are:
1
2
Pop an item from stack 2
Pop an item from stack 1
Stack is empty.
```

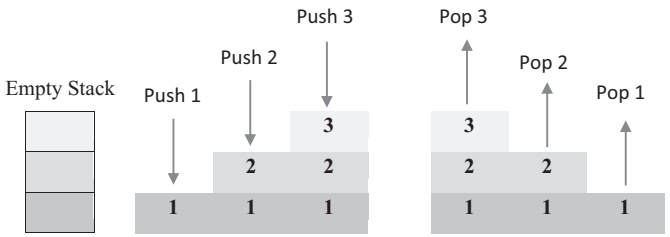


FIGURE 6.2
Stack push and pop operations.

The list methods make it very easy to use a list as a stack (FIGURE 6.2), where the last item added is the first item retrieved ("last-in, first-out"). To add an item to the top of the stack, use the *append()* method. To retrieve an item from the top of the stack, use *pop()* without an explicit index. Set the stack size to three. The function *display_stack_items()* ③–⑥ displays current items in the stack. The function *push_item_to_stack()* ⑦ is used to push an item to stack ⑩ if the total length of the stack is less than the stack size ⑨ else display *"Stack is full"* ⑫ message. The function *pop_item_from_stack()* ⑬ pops an item from the stack ⑮ if the stack is not empty ⑭ else display *"Stack is empty"* message ⑰.

Program 6.10: Write Python Program to Perform Queue Operations

```
1. from collections import deque

2. def queue_operations():
3.     queue = deque(["Eric", "John", "Michael"])
4.     print(f"Queue items are {queue}")
5.     print("Adding few items to Queue")
6.     queue.append("Terry")
7.     queue.append("Graham")
8.     print(f"Queue items are {queue}")
9.     print(f"Removed item from Queue is {queue.popleft()}")
10.    print(f"Removed item from Queue is {queue.popleft()}")
11.    print(f"Queue items are {queue}")

12. def main():
13.    queue_operations()

14. if __name__ == "__main__":
15.    main()
```

OUTPUT

```
Queue items are deque(['Eric', 'John', 'Michael'])
Adding few items to Queue
Queue items are deque(['Eric', 'John', 'Michael', 'Terry', 'Graham'])
```

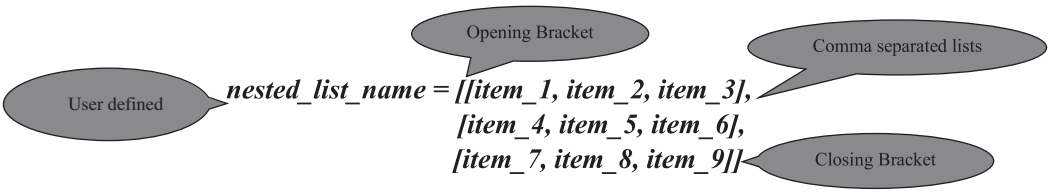
Removed item from Queue is Eric
Removed item from Queue is John
Queue items are deque(['Michael', 'Terry', 'Graham'])

It is also possible to use a list as a queue, where the first item added is the first item retrieved ("first-in, first-out"). However, lists are not effective for this purpose. While appends and pops are fast from the end of the list, doing inserts or pops from the beginning of a list is slow because all of the other items have to be shifted by one. To implement a queue ③, use *collections.deque* ① which was designed to have fast appends ⑨–⑩ and pops ⑥–⑦ from both ends.

6.5.3 Nested Lists

A list inside another list is called a nested list and you can get the behavior of nested lists in Python by storing lists within the elements of another list. You can traverse through the items of nested lists using the *for* loop.

The syntax for nested lists is,



Each list inside another list is separated by a comma. For example,

```
1. >>> asia = [["India", "Japan", "Korea"],
                ["Srilanka", "Myanmar", "Thailand"],
                ["Cambodia", "Vietnam", "Israel"]]
2. >>> asia[0]
   ['India', 'Japan', 'Korea']
3. >>> asia[0][1]
   'Japan'
4. >>> asia[1][2] = "Philippines"
5. >>> asia
   [['India', 'Japan', 'Korea'], ['Srilanka', 'Myanmar', 'Philippines'], ['Cambodia', 'Vietnam',
   'Israel']]
```

You can access an item inside a list that is itself inside another list by chaining two sets of square brackets together. For example, in the above list variable *asia* you have three lists ① which represent a 3 × 3 matrix. If you want to display the items of the first list then specify the list variable followed by the index of the list which you need to access within the brackets, like *asia[0]* ②. If you want to access "Japan" item inside the list then you need to specify the index of the list within the list and followed by the index of the item in the list,

like *asia[0][1]* ③. You can even modify the contents of the list within the list. For example, to replace "Thailand" with "Philippines" use the code in ④.

Program 6.11: Write a Program to Find the Transpose of a Matrix

```
1. matrix = [[10, 20],
             [30, 40],
             [50, 60]]
2. matrix_transpose = [[0, 0, 0],
                       [0, 0, 0]]
3. def main():
4.     for rows in range(len(matrix)):
5.         for columns in range(len(matrix[0])):
6.             matrix_transpose[columns][rows] = matrix[rows][columns]
7.     print("Transposed Matrix is")
8.     for items in matrix_transpose:
9.         print(items)
10. if __name__ == "__main__":
11.     main()
```

Output

```
Transposed Matrix is
[10, 30, 50]
[20, 40, 60]
```

The transpose of a matrix is a new matrix whose rows are the columns and columns are the rows of the original matrix. To find the transpose of a matrix ① in Python, you need an empty matrix initially to store the transposed matrix ②. This empty matrix should have one column greater and one row less than the original matrix. You need two *for* loops corresponding to rows ④ and columns of the matrix ⑤. Using *for* loops you need to iterate through each row and each column. At each point we place the *matrix[rows][columns]* item into *matrix_transpose[columns][rows]* ⑥. Finally, print the transposed result using *for* loop ⑧–⑨.

Program 6.12: Write Python Program to Add Two Matrices

```
1. matrix_1 = [[1, 2, 3],
               [4, 5, 6],
               [7, 8, 9]]
2. matrix_2 = [[1, 2, 3],
               [4, 5, 6],
               [7, 8, 9]]
```

```
3. matrix_result = [[0, 0, 0],
                    [0, 0, 0],
                    [0, 0, 0]]
4. for rows in range(len(matrix_1)):
5.     for columns in range(len(matrix_2[0])):
6.         matrix_result[rows][columns] = matrix_1[rows][columns] + matrix_2[rows]
           [columns]
7. print("Addition of two matrices is")
8. for items in matrix_result:
9.     print(items)
```

Output

```
Addition of two matrices is
[2, 4, 6]
[8, 10, 12]
[14, 16, 18]
```

To add two matrices in Python, you need to have two matrices which need to be added ①–② and another empty matrix ③. The empty matrix is used to store the addition result of the two matrices. To perform addition of matrices you need two *for* loops corresponding to rows ④ and columns ⑤ of *matrix_1* and *matrix_2* respectively. Using *for* loops you need to iterate through each row and each column. At each point we add *matrix_1[rows][columns]* item with *matrix_2[rows][columns]* item ⑥. Finally, print the matrix result of adding two matrices using *for* loop ⑧–⑨.

6.6 The *del* Statement

You can remove an item from a list based on its index rather than its value. The difference between *del* statement and *pop()* function is that the *del* statement does not return any value while the *pop()* function returns a value. The *del* statement can also be used to remove slices from a list or clear the entire list.

```
1. >>> a = [5, -8, 99.99, 432, 108, 213]
2. >>> del a[0]
3. >>> a
   [-8, 99.99, 432, 108, 213]
4. >>> del a[2:4]
5. >>> a
   [-8, 99.99, 213]
6. >>> del a[:]
7. >>> a
   []
```

An item at an index value of zero is removed ②. Now the number of items in the original list is reduced ③. Items starting from an index value of 2 up to 4 but excluding the index value of 4 is removed from the list ④. All the items in the list can be removed by specifying only the colon operator without *start* or *stop* index values ⑥–⑦.

6.7 Summary

- Lists are a basic and useful data structure built into the Python language.
- Built-in functions include *len()*, which returns the length of the list; *max()*, which returns the maximum element in the list; *min()*, which returns the minimum element in the list and *sum()*, which returns the sum of all the elements in the list.
- An individual elements in the list can be accessed using the index operator [].
- Lists are mutable sequences which can be used to add, delete, sort and even reverse list elements.
- The *sort()* method is used to sort items in the list.
- The *split()* method can be used to split a string into a list.
- Nested list means a list within another list.

Multiple-Choice Questions

1. The statement that creates the list is
   a. superstore = list()
   b. superstore = []
   c. superstore = list([1,2,3])
   d. All of the above
2. Suppose continents = [1,2,3,4,5], what is the output of len(continents)?
   a. 5
   b. 4
   c. None
   d. error
3. What is the output of the following code snippet?
   islands = [111,222,300,411,546]
   max(islands)
   a. 300
   b. 222
   c. 546
   d. 111

4. Assume the list superstore is [1,2,3,4,5], which of the following is correct syntax for slicing operation?
   a. print(superstore[0:])
   b. print(superstore[:2])
   c. print(superstore[:-2])
   d. All of these
5. If zoo = ["lion", "tiger"], what will be zoo * 2?
   a. ['lion']
   b. ['lion', 'lion', 'tiger', 'tiger']
   c. ['lion', 'tiger', 'lion', 'tiger']
   d. ['tiger']
6. To add a new element to a list the statement used is?
   a. zoo. add(5)
   b. zoo.append("snake")
   c. zoo.addLast(5)
   d. zoo.addend(4)
7. To insert the string "snake" to the third position in zoo, which of the following statement is used?
   a. zoo.insert(3, "snake")
   b. zoo. insert(2, "snake")
   c. zoo.add(3, "snake")
   d. zoo.append(3, "snake")
8. Consider laptops = [3, 4, 5, 20, 5, 25, 1, 3], what will be the output of laptops.reverse()?
   a. [3, 4, 5, 20, 5, 25, 1, 3]
   b. [1, 3, 3, 4, 5, 5, 20, 25]
   c. [25, 20, 5, 5, 4, 3, 3, 1]
   d. [3, 1, 25, 5, 20, 5, 4, 3]
9. Assume quantity = [3, 4, 5, 20, 5, 25, 1, 3], then what will be the items of quantity list after quantity.pop(1)?
   a. [3, 4, 5, 20, 5, 25, 1, 3]
   b. [1, 3, 3, 4, 5, 5, 20, 25]
   c. [3, 5, 20, 5, 25, 1, 3]
   d. [1, 3, 4, 5, 20, 5, 25]
10. What is the output of the following code snippet?

    ```
    letters = ['a', 'b', 'c', 'd', 'e']
    letters[::-2]
    ```

    a. ['d', 'c', 'b']
    b. ['a', 'c', 'e']
    c. ['a', 'b', 'd']
    d. ['e', 'c', 'a']

11. Suppose list_items is [3, 4, 5, 20, 5, 25, 1, 3], then what is the result of list_items. remove(4)?
    a. 3, 5, 29, 5
    b. 3, 5, 20, 5, 25, 1, 3
    c. 5, 20, 1, 3
    d. 1, 3, 25
12. Find the output of the following code.

```
matrix= [[1,2,3],[4,5,6]]
v = matrix[0][0]
for row in range(0, len(matrix)):
    for column in range(0, len(matrix[row])):
        if v < matrix[row][column]:
            v = matrix[row][column]
print(v)
```

    a. 3
    b. 5
    c. 6
    d. 33
13. Gauge the output of the following.

```
matrix = [[1, 2, 3, 4],
          [4, 5, 6, 7],
          [8, 9, 10, 11],
          [12, 13, 14, 15]]

for i in range(0, 4):
    print(matrix[i][1])
```

    a. 1 2 3 4
    b. 4 5 6 7
    c. 1 3 8 12
    d. 2 5 9 13
14. What will be the output of the following?

```
data = [[[1, 2], [3, 4]], [[5, 6], [7, 8]]]
print(data[1][0][0])
```

    a. 1
    b. 2
    c. 4
    d. 5

15. The list function that inserts the item at the given index after shifting the items to the right is
    a. sort()
    b. index()
    c. insert()
    d. append()
16. The method that is used to count the number of times an item has occurred in the list is
    a. count()
    b. len()
    c. length()
    d. extend()

Review Questions

1. Explain the advantages of the list.
2. Explain the different ways in which the lists can be created.
3. Explain the different list methods with an example.
4. With the help of an example explain the concept of nested lists.
5. Explain the ways of indexing and slicing the list with examples.
6. Differentiate between the following:
    a. pop() and remove() methods of list.
    b. Del statement and pop() method of list.
    c. append() and insert() methods of list.
7. Write a program that creates a list of 10 random integers. Then create two lists by name odd_list and even_list that have all odd and even values of the list respectively.
8. Write a program to sort the elements in ascending order using insertion sort.
9. Write a Python program to use binary search to find the key element in the list.
10. Make a list of the first eight letters of the alphabet, then using the slice operation do the following operations:
    a. Print the first three letters of the alphabet.
    b. Print any three letters from the middle.
    c. Print the letters from any particular index to the end of the list.
11. Write a program to sort the elements in ascending order using selection sort.
12. Write a program that prints the maximum value of the second half of the list.
13. Write a program that creates a list of numbers 1–100 that are either divisible by 5 or 6.
14. Write a function that prompts the user to enter five numbers, then invoke a function to find the GCD of these numbers.