\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**CSC121 Python Programming**

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Lesson 05 Lists [Part 1]

# **Objectives**

In this lesson, students will learn:

- How to create a list

- How to access list elements

- How to alter a list

- How to iterate over a list

# **5.1 Overview**

We have spent some time learning if statements and while statements. The if and while statements are control structure statements because they control which block of statements to execute and how many times the block will execute. Control structures are very important. We use them frequently in almost all programs.

We are going to switch to a new topic. Instead of studying program flow, now we want to focus on data. Specifically, you will see how to store and use a large group of data in a convenient and efficient way, which is also very important in many programs.

In Python, there are a few ways to store a large group of data. One way is using **lists**. A list stores data items one after another just like listing the things you need to buy from a grocery store on a piece of paper. You will see how to create a list, how to add items to a list, how to access the items, and how to alter them.

# **5.2 Lists**

Sometimes we need to store a large group of data in a program. For example, suppose 10 students are enrolled in a class and they just took a test. The instructor wants a program to analyze their test scores, such as finding the class average, the highest score, the lowest score, counting how many students pass and fail, etc. How do we write this program?

First of all, we need to store the 10 scores. Using what we have learned so far, we can create 10 variables to store them. Suppose these variables are named s0, s1, s2, s3, etc. (Programmers like to start counting from 0 instead of 1). We then write code to calculate different statistics with these 10 variables. For example, we can write a statement like the following to calculate the average score:

average = (s1 + s2 + s3 + s4 + s5 + s6 +s7 + s8 + s9 + s10)/10

This looks fine, but imagine that the class size increases from 10 to 40. We will have to create 40 variables to store the scores and write a very long statement to calculate the average of 40 variables. This is tedious and error prone. Furthermore, what if the class size expands to 100? It is definitely not a good idea to create 100 variables and write an extremely long statement to calculate their average. Obviously we need a more convenient way to store and use a large group of data in situations like this.

In Python, **list** is our answer. A list is a linear data structure. Data items in a list are called **elements**. Elements in a list have a linear ordering, meaning that they line up one after another. We must give a name to a list when it is created. For example, the list that stores test scores may be named score. Each element in the list is identified by a unique number, called a subscript or index, which is assigned by the computer automatically when the list is created. The subscript indicates the element's position in the list. In Python, the subscript starts with 0, and goes up as 1, 2, 3, 4, …, and so on. The first element in the list has a subscript of 0, while the second element in the list has a subscript of 1, and so on. For a 10-element list, the subscripts are from 0 through 9. Notice that the last element in this list has a subscript of 9, not 10. We use the name and the subscript together to refer to an element in the list. For example, score[0] refers to the first element in the list. score[1] is the element right after score[0], and score[2] is the element right after score[1], and so on. The last element in the list is score[9]. The following is an example.

score[0]

score[1]

score[2]

score[3]

score[4]

score[5]

score[6]

score[7]

score[8]

score[9]

97

85

93

76

81

88

100

72

84

75

# **5.3 Creating a List**

In Python, we use an assignment statement to create a list. You put the name of the list on the left hand side, and the elements enclosed in square brackets on the right hand side. The following is an example:

score = [97, 85, 93, 76, 81, 88, 100, 72, 84, 75]

The statement above creates a list named score that holds 10 integers.

The following statement creates a list gpa to store GPA’s of 5 students:

gpa = [3.25, 2.87, 3.10, 3.56, 2.64]

The following statement creates a list dept to store names of 4 departments:

dept = [**'Psychology'**, **'History'**, **'Philosophy'**, **'English'**]

The following statement creates an empty list list1, which has no elements:

list11 = []

Sometimes we create an empty list first and then add elements to it. You will see some examples soon.

Once a list is created, you can use the print function to display it. Examples:

score = [97, 85, 93, 76, 81, 88, 100, 72, 84, 75]  
gpa = [3.25, 2.87, 3.10, 3.56, 2.64]  
dept = [**'Psychology'**, **'History'**, **'Philosophy'**, **'English'**]  
print(score)  
print(gpa)  
print(dept)

When you pass the name of a list to the print function, the whole list will be displayed inside a pair of square brackets. The following is the output of the program:

[97, 85, 93, 76, 81, 88, 100, 72, 84, 75]

[3.25, 2.87, 3.1, 3.56, 2.64]

['Psychology', 'History', 'Philosophy', 'English']

# **5.4 Accessing and Altering List Elements**

## Retrieving list elements

You can retrieve a list element just like you retrieve the value of a variable. Examples:

print(score[2])  
print(gpa[1])  
print(dept[0])

The first statement displays 93, which is the third element in the list score. The second statement displays 2.87, which is the second element in the list gpa. The third statement displays Psychology, which is the first element in the list dept. When we retrieve a list element, it does not remove the element from the list. We just simply look up the value.

Unlike other major programming languages, Python allows us to use negative indices to access list elements. Examples:

print(score[-2])  
print(gpa[-1])  
print(dept[-3])

When negative indices are used, we are counting backward from the end of the list. The index -1 means the last element in the list. The index -2 means the second-to-last element in the list. The index -3 means the third-to-last element in the list, and so on. Using this system, the first statement displays 84, which is the second-to-last element in the list score. The second statement displays 2.64, which is the last element in the list gpa. The third statement displays History, which is the third-to-last element in the list dept. Notice that the index -0 is the same as 0. It refers to the first, not the last, element of the list. Therefore, the statement

print(dept[-0])

is the same as the statement

print(dept[0])

Both of them display Psychology, which is the first element of the list dept.

The examples above retrieve list elements to display them. You can retrieve list elements for other purposes such as using them for calculations. Examples:

total\_first\_two\_scores = score[0] + score[1]  
avg\_last\_three\_gpas = (gpa[-1] + gpa[-2] + gpa[-3])/3

The first statement calculates the total of first two elements in the list score, while the second statement calculates the average of the last three elements in the list gpa.

## Altering list elements

We can use assignment statements to change elements of a list. Examples:

score[1] = 100  
gpa[-1] = 3.87  
dept[2] = **'Music'**

The first statement changes the second element of score to 100. The second statement changes the last element of gpa to 3.87. The third statement changes the third element of dept to Music.

# **5.5 Adding Elements to a List**

## The append function

We can use append to add a new element to the end of a list. For example, the third statement in the following example adds 2.97 as the sixth element to the 5-element list gpa:

gpa = [3.25, 2.87, 3.10, 3.56, 2.64]  
print(**"List before append:"**, gpa)  
gpa.append(2.97)  
print(**"List after append: "**, gpa)

The list gpa has 5 elements originally. It has 6 elements after the append function adds 2.97 to the end of the list. This is shown by the print statements:

List before append: [3.25 2.87 3.1 3.56 2.64]

List after append: [3.25 2.87 3.1 3.56 2.64 2.97]

More examples of append:

score.append(65)  
dept.append(**'Math'**)

The first statement adds 65 as the eleventh element to the 10-element list score. The second statement adds math as the fifth element to the 4-element list dept.

The append function is a useful tool when we want to input data from the user and store them in a list.

score = []  
new\_score = int(input(**"Enter a score: "**))  
score.append(new\_score)  
new\_score = int(input(**"Enter a score: "**))  
score.append(new\_score)  
new\_score = int(input(**"Enter a score: "**))  
score.append(new\_score)  
print(**"The list:"**, score)

The program above starts with an empty list. It then reads a test score from the keyboard and appends it to the list. It does the same things two more times and we end up having three scores in the list. The last statement displays the three scores stored in the list. The following is a sample test run:

Enter a score: 97

Enter a score: 82

Enter a score: 91

The list: [97, 82, 91]

The program can be shortened by using a loop. In Lesson 04 we learned how to use a counter-controlled loop to repeat the same action a specified number of times:

*Set number of iterations to 0*

*While number of iterations < preset limit:*

*Statements to process one case*

*Increase number of iterations by one*

The key here is to use a variable to keep track of how many times the loop has completed so far. It is set to 0 before the loop and increases by 1 before the end of the loop body. The loop executes repeatedly before iterations reaches the preset limit. The variable that counts how many iterations have completed is called a counter variable.

We can apply that technique here:

score = []  
i = 0 # number of iterations completed  
**while** i < 3:  
 new\_score = int(input(**"Enter a score: "**))  
 score.append(new\_score)  
 i = i + 1  
print(**"The list:"**, score)

The counter variable i is set to 0 before the loop. In the loop body, we read one new score from the keyboard and append it to the list immediately. The counter variable i increases by 1 before the end of the loop body. The loop repeats again and again as long as i is less than 3. When i reaches 3, the loop will stop.

## The insert function

We can also use the insert function to insert an element at a specified position of a list. The third statement in the following example inserts 3.88 as the second element of the list gpa:

gpa = [3.25, 2.87, 3.10, 3.56, 2.64]  
print(**"List before insert:"**,gpa)  
gpa.insert(1, 3.88)  
print(**"List after insert: "**, gpa)

The insert function requires two pieces of information: the position in the list where the new element will be inserted, and the data to be inserted. The syntax gpa.insert(1, 3.88) tells the computer to insert 3.88 in the position with index equal to 1, which means adding 3.88 as the second element. Let’s look at the list before and after the insert statement:

List before insert: [3.25, 2.87, 3.1, 3.56, 2.64]

List after insert: [3.25, 3.88, 2.87, 3.1, 3.56, 2.64]

The first element 3.25 remains the same. After 3.88 is inserted as the second element, all other elements shift one position down. For example, 2.87 was originally the second element, now it is the third.

More example of insert:

score.insert(0, 88)  
dept(3, **'Math'**)

The first statement inserts 88 as the first element of the list score. The second statement inserts Math as the fourth element of the list dept.

# **5.6 More List Operations**

In the last section we saw how to add elements to a list. In this section, we will see a few more list operations.

## The len function

The len function finds the length of a list. Example:

gpa = [3.25, 2.87, 3.10, 3.56, 2.64]  
list\_length = len(gpa)  
print(**"Number of elements in the list:"**, list\_length)

The len function in the second statement finds the length of the list gpa and stores it in the variable list\_length, which is displayed by the third statement:

Number of elements in the list: 5

Another example:

dept = [**'Psychology'**, **'History'**, **'Philosophy'**, **'English'**]

list\_length = len(dept)  
print(**"Number of elements in the list:"**, list\_length)

The len function in the second statement finds the length of the list dept and stores it in the variable list\_length, which is displayed by the third statement:

Number of elements in the list: 4

## The in operator

The in operator tests whether a test value is in an element or not. Example:

gpa = [3.25, 2.87, 3.10, 3.56, 2.64]  
**if** 3.10 **in** gpa:  
 print(**"3.10 found in the list"**)  
**else**:  
 print(**"3.10 not in the list"**)

The in operator in the if statement tests whether 3.10 is an element in the list gpa or not. If 3.10 is found in the list, the condition is true. If 3.10 is not in the list, the condition is false.

Another example:

dept = [**'Psychology'**, **'History'**, **'Philosophy'**, **'English'**]

**if 'History' in** dept:  
 print(**"History is found in the list"**)  
**else**:  
 print(**"History is not in the list"**)

The in operator in the if statement tests whether the string History is an element in the list dept or not. If History is found in the list, the condition is true. If History is not in the list, the condition is false.

We can also ask the user to enter a target to search:

dept = [**'Psychology'**, **'History'**, **'Philosophy'**, **'English'**]

target = input(**"Enter search target: "**)  
**if** target **in dept**:  
 print(target, **"is found in the list"**)  
**else**:  
 print(target, **"is not in the list"**)

Sample test run:

Enter search target: History

History is found in the list

Another test run:

Enter search target: Music

Music is not in the list

## The del keyword

Sometimes we want to remove elements from a list. There are two ways to remove elements. The first way is using del. Let’s look at an example:

gpa = [3.25, 2.87, 3.10, 3.56, 2.64]  
print(**"List before delete:"**,gpa)  
**del** gpa[2]  
print(**"List after delete: "**, gpa)

The third statement in the program above deletes the element with index 2, i.e. the third element, from the list gpa. Before the delete operation, 3.10 was the third element. After delete, this element is no longer in the list. The list is shortened from 5 elements to 4 elements.

List before delete: [3.25 2.87 3.1 3.56 2.64]

List after delete : [3.25 2.87 3.56 2.64]

More examples of del:

del score[0]  
del dept[**-2**]

The first statement deletes the first element from the list score. The second statement deletes the second-to-last element from the list dept.

## The remove function

We can also use the remove function to remove elements. Let’s look at an example:

gpa = [3.25, 2.87, 3.10, 3.56, 2.64]  
print(**"List before remove:"**,gpa)  
gpa.remove(3.10)  
print(**"List after remove:"**, gpa)

The third statement in the program above removes the element 3.10 from the list gpa. Before the remove operation, 3.10 was the third element. After remove, this element is no longer in the list. The list is shortened from 5 elements to 4 elements.

List before remove: [3.25 2.87 3.1 3.56 2.64]

List after remove: [3.25 2.87 3.56 2.64]

More examples of remove:

score.remove(97)  
dept.remove(**'History'**)

The first statement removes the score 97 from the list score. The second statement removes the name History from the list dept.

If two or more elements have the same value, the remove function only removes the first one. Here is an example:

high\_temp = [92, 89, 92]  
print(**"List before remove:"**, high\_temp)  
high\_temp.remove(92)  
print(**"List after remove: "**, high\_temp)

The list high\_temp originally has three elements 92, 89 and 92. There are two 92’s in the list. The remove function in the third statement only removes the first. There are two elements in high\_temp after remove:

List before remove: [92 89 92]

List after remove: [89 92]

## The sort function

The sort function sorts a list in ascending order, i.e., elements are ordered from small to large. Example:

gpa = [3.25, 2.87, 3.10, 3.56, 2.64]  
print(**"List before sort:"**, gpa)  
gpa.sort()  
print(**"List after sort: "**, gpa[0])

Before the sort, elements in the list are not ordered by values. After the sort, the smallest element is placed at the beginning of the list and each element is larger than the one right before it:

List before sort: [3.25 2.87 3.1 3.56 2.64]

List after sort: [2.64 2.87 3.1 3.25 3.56]

Sort works even if elements in a list are strings. When two strings are compared to decide which one is smaller than the other one, they are compared character by character starting from the first character from both strings. Character that has a smaller ASCII value is considered as smaller than the other character. The string that has a smaller character first is considered as the smaller string. The following is an example:

dept = [**'Psychology'**, **'History'**, **'Philosophy'**, **'English'**]  
print(**"List before sort:"**, dept)  
dept.sort()  
print(**"List after sort: "**, dept)

The third statement in the program above sorts the elements in the list:

List before sort: [Psychology History Philosophy English]

List after sort: [English History Philosophy Psychology]

## The reverse function

The reverse function reverses the order of the elements in a list. Example:

gpa = [3.25, 2.87, 3.10, 3.56, 2.64]  
print(**"List before reverse:"**, gpa)  
gpa.reverse()  
print(**"List after reverse: "**, gpa)

The third statement in the program above reverses the elements in the list. The first element becomes the last, the second element becomes the second-to-last, and so on:

List before reverse: [3.25 2.87 3.1 3.56 2.64]

List after reverse: [2.64 3.56 3.1 2.87 3.25]

Another example:

dept = [**'Psychology'**, **'History'**, **'Philosophy'**, **'English'**]  
print(**"List before reverse:"**,dept)  
dept.reverse()  
print(**"List after reverse: "**,dept)

The third statement in the program above reverses the elements in the list:

List before reverse: [Psychology History Philosophy English]

List after reverse: [English Philosophy History Psychology]

# **5.7 Iterating Over a List**

Sometimes we need to access every list element, e.g. using every element to do some calculations. There are two ways to do it in Python. In this section, we will see how to use a while statement to achieve this goal.

In an earlier section we saw Python code that uses a while loop to input and append three new elements to a list:

score = []  
i = 0 # number of iterations completed  
**while** i < 3:  
 new\_score = int(input(**"Enter a score: "**))  
 score.append(new\_score)  
 i = i + 1

print(**"The list:"**, score)

Suppose we want to modify the program so each element of the list is displayed in a separate line. We can replace the last print statement with three print statements:

score = []  
i = 0 *# number of iterations completed***while** i < 3:  
 new\_score = int(input(**"Enter a score: "**))  
 score.append(new\_score)  
 i = i + 1  
print(score[0])  
print(score[1])  
print(score[2])

A better way is to write a while loop to iterate over the list and display a different element each time. The loop will be a counter-controlled loop that iterates exactly three times. The logical structure of this while loop will be the same as the while loop that input and append three elements:

**while** i < 3:  
 *# use a print statement to display an element* i = i + 1

Now we need to figure out how to write the print statement. Our goal is to display all list elements. In the first iteration, i.e. when the counter variable i is 0, we want to display score[0]. In the second iteration when i is 1, we want to display score[1]. In the third iteration when i is 2, we want to display score[2], and so on. In general, the element we want to display is simply score[i]:

i = 0  
**while** i < 3:  
 print(score[i])  
 i = i + 1

Let’s write another loop for the same program. Suppose we want to give every student 5 extra points. We can use the same technique to do that:

i = 0  
**while** i < 3:  
 score[i] = score[i] + 5  
 i = i + 1

The following program combines what we learned so far about iterating over a list. It uses loops to read 3 scores, give students extra points and display new scores.

score = []  
i = 0 *# number of iterations completed***while** i < 3:  
 new\_score = int(input(**"Enter a score: "**))  
 score.append(new\_score)  
 i = i + 1  
  
i = 0 *# reset i to 0 before loop***while** i < 3:  
 score[i] = score[i] + 5  
 i = i + 1  
  
print(**'Scores after 5 extra points:'**)  
i = 0 *# reset i to 0 before loop***while** i < 3:  
 print(score[i])  
 i = i + 1

The following is a sample test run:

Enter a score: 77

Enter a score: 89

Enter a score: 81

Scores after 5 extra points:

82

94

86

Here is another example. This program asks the user to enter as many student names as he wants. It converts these names into all uppercase letters and then displays them.

student = []  
new\_element = input(**"Enter student name: "**)  
student.append(new\_element)  
again = input(**"Add another student? [y/n]"**)  
  
**while** again == **'y'**:  
 new\_element = input(**"Enter student name: "**)  
 student.append(new\_element)  
 again = input(**"Add another student? [y/n]"**)  
  
length = len(student)  
i = 0  
**while** (i < length):  
 student[i] = student[i].upper()  
 i = i + 1  
  
print(**"Student names in uppercase letters:"**)  
i = 0  
**while** (i < length):  
 print(student[i])  
 i = i + 1

Just like before, we start with an empty list. Every time a new student name is entered, it is appended to the list. This repeats until the user no longer wants to add another name. We then use a counter-controlled loop to convert the names into uppercase letters. Converting letters to uppercase can be done by the upper function of the string. This function returns the all-cap version of the string, and we use it in an assignment statement to save the all-cap string in the list to replace the original string.

Another new thing in this program is using the len function to find out the number of elements in this list. This is necessary because we do not know how many names the user is going to enter. This number may be different every time this program runs.

The following is a test run of the program:

Enter student name: Amy Cohen

Add another student? [y/n]y

Enter student name: Ben Simon

Add another student? [y/n]y

Enter student name: Cathy Harris

Add another student? [y/n]y

Enter student name: Don Nelson

Add another student? [y/n]n

Student names in uppercase letters:

AMY COHEN

BEN SIMON

CATHY HARRIS

DON NELSON

# **5.8 For Loops**

A **for** statement is an iterative control statement that iterates once for each element in a list. It works like a counter-controlled while loop, but its syntax is a little easier and it has a limitation. Before we look at the for statement, let’s review how we use a counter-controlled while loop to iterate over list elements.

student\_list = [**'Pete Li'**, **'Al Davis'**, **'Den White'**, **'Dave Fox'**]  
list\_length = len(student\_list)  
print(**"Elements in the list:"**)  
i = 0  
**while** i < list\_length:  
 print(student\_list[i])  
 i = i + 1

The program above uses a counter-controlled while loop to display all the elements in the list. The counter variable i is set to 0 before the loop. It increases by 1 every time the loop iterates and the loop stops when i reaches the length of the list. The print statement displays a different element in the list each time the loop body executes.

Let’s add one extra statement in the loop to store the element student\_list[i]. We will explain why this statement is added later:

student\_list = [**'Pete Li'**, **'Al Davis'**, **'Den White'**, **'Dave Fox'**]  
list\_length = len(student\_list)  
print(**"Elements in the list:"**)  
i = 0  
**while** i < list\_length:  
 student = student\_list[i]  
 print(student)  
 i = i + 1

The statement

student = student\_list[i]

creates a variable student and copies the element student\_list[i] to it. The print statement then displays student.

This program generates the same output:

Elements in the list:

Pete Li

Al Davis

Den White

Dave Fox

Using the variable student to store a list element before it is displayed is not necessary. This extra statement is added because now the loop can be rewritten by using a for statement:

student\_list = [**'Pete Li'**, **'Al Davis'**, **'Den White'**, **'Dave Fox'**]  
list\_length = len(student\_list)  
print(**"Elements in the list:"**)  
**for** student **in** student\_list:  
 print(student)

The computer executes the for loop exactly like it executes the while loop with that extra assignment statement. Every time the loop body executes, the computer copies a different list element to the variable student, which is then displayed by the print statement. The new syntax is a lot shorter and cleaner. There is no need to create, initialize and update a counter variable. That is done by the computer automatically. Furthermore, we do not need to use the index notation to refer to a list element. The computer will automatically copy a different element to the variable student in different iterations of the loop. The variable student is referred to as a **loop variable**. You can interpret the for statement like this: for each student in the student list, print student.

Let’s look at another example.

high\_temp = [92, 89, 90, 95]  
total = 0  
**for** temp **in** high\_temp:  
 total = total + temp  
avg\_temp = total/len(high\_temp)  
print(**"Average daily high temperature:"**, avg\_temp)

This program calculates the average daily high temperature. It uses a for loop to add every element of the list high\_temp to the variable total. The total is then divided by the number of elements to calculate average. When the computer executes the for loop, a loop variable temp is created and a different element of the list is copied to it every time. Therefore, you can interpret the for loop in this program like this: for every temperature in the list high\_temp, add the temperature to total.

Although the for loop is easy to use, it has a limitation. We are unable to use a for loop to alter the elements of the list. For example, suppose you want to add 2 degrees to every element in the list high\_temp, the for loop does not work.

high\_temp = [92, 89, 90, 95]  
print(**"Elements before raising temperature:"**)  
**for** temp **in** high\_temp:  
 print(temp)  
  
**for** temp **in** high\_temp:  
 temp = temp + 2 *# does not change list element*print(**"Elements after raising temperature:"**)  
**for** temp **in** high\_temp:  
 print(temp)

The program above tries to use a for loop to add 2 to every element of high\_temp. However, the statement

temp = temp + 2

only adds 2 to the loop variable temp. It does not add 2 to the loop element temp is copied from. This is shown by comparing the elements before and after the loop:

Elements before raising temperature:

92

89

90

95

Elements after raising temperature:

92

89

90

95

The elements are exactly the same before and after the loop. That means the assignment statement inside the for loop has no effect on the list elements.

# **5.9 The Range Function And More About For Loops**

The range function generates a sequence of integers. Example:

sequence = range(5)  
print(**"Integers generated by range(5):"**)  
**for** element **in** sequence:  
 print(element)

The range function in the program above generates the sequence of integers 0, 1, 2, 3, 4. The following is the output of the program:

Integers generated by range(5):

0

1

2

3

4

The integers generated by the range function are determined by what you put inside the parentheses. You can put one, two or three numbers there. If you put just one number there, it is the final value of the sequence. The sequence starts with 0. Each number in the sequence is 1 larger than the number right before it. The sequence stops right before the final value, i.e. the final value is not included in the sequence. For example, range(4) generates the sequence 0, 1, 2, 3, and range(7) generates the sequence 0, 1, 2, 3, 4, 5, 6.

If you put two numbers inside the parentheses, the first number will be the starting value while the second number will be ending value. Example:

sequence = range(4, 9)  
print(**"Integers generated by range(4, 9):"**)  
**for** element **in** sequence:  
 print(element)

The following is the output of the program:

Integers generated by range(4, 9):

4

5

6

7

8

Similarly, range(2, 6) generates the sequence 2, 3, 4, 5, while range(-2, 4) generates the sequence -2, -1, 0, 1, 2, 3.

If you put three numbers inside the parentheses, the first number will be the starting value, the second number will be ending value, while the third number will be the step value. The step value determines how much larger each number is compared to the number right before it. Example:

sequence = range(3, 11, 2)  
print(**"Integers generated by range(3, 11, 2):"**)  
**for** element **in** sequence:  
 print(element)

The following is the output of the program:

Integers generated by range(3, 11, 2):

3

5

7

9

Similarly, range(1, 9, 3) generates the sequence 1, 4, 7, while range(-5, 8, 4) generates the sequence -5, -1, 3, 7.

You can also use negative step values. When a negative step value is used, each number in the sequence is smaller than the number right before it. Example,

sequence = range(7, 2, -1)  
print(**"Elements generated by range(7, 2 -1):"**)  
**for** element **in** sequence:  
 print(element)

The following is the output of the program:

Elements generated by range(7, 2 -1):

7

6

5

4

3

Similarly, range(11, 4, -2) generates the sequence 11, 9, 7, 5, while range(25, 4, -5) generates the sequence 25, 20, 15, 10, 5.

Now we know that the range function generates a sequence of integers. When do we use it? There are many applications of the range function. First of all, we can use it to generate a list. Example:

sequence = range(5)  
list1 = list(sequence)  
print(**"List generated by range(5):"**, list1)

The second statement in the program above creates a new list from the sequence generated by the range function. The following is the output:

List generated by range(5): [0, 1, 2, 3, 4]

The first and the second statements can be combined to make the program shorter:

list1 = list(range(5))  
print(**"List generated by range(5):"**, list1)

Another example:

list1 = list(range(4, 9))  
print(**"List generated by range(4, 9):"**, list1)

The following is the output:

List generated by range(4, 9): [4, 5, 6, 7, 8]

Another application of the range function is using it like a counter variable. Sometimes a counter-controlled while loop can be rewritten as a for loop with a range function. Let’s look at an example.

We saw this counter-controlled while loop before:

iterations = 0  
**while** iterations < 5:  
 score = float(input(**"Enter score: "**))  
 new\_score = score + 10  
 print(**"New score:"**, new\_score)  
 iterations = iterations + 1

This program uses a counter-controlled loop to repeat these actions 5 times: input a score, add 10 points to it and display the new score. It uses a counter variable iterations to control when the loop executes and when it stops.

We can rewrite this program with a for loop. First, we create the sequence 0, 1, 2, 3 and 4. These values are exactly the same as the values of the counter variable iterations in the program above. We then write a for loop to iterate over this sequence. Since this sequence has 5 elements, the loop will execute 5 times. Inside the loop body we input, increase and display one score. The following is the Python code:

sequence = range(5)  
**for** i **in** sequence:  
 score = float(input(**"Enter score: "**))  
 new\_score = score + 10  
 print(**"New score:"**, new\_score)

The first statement uses the range function to generate the sequence 0, 1, 2, 3, 4 and store the numbers in sequence, which is then used in the for loop. Many programmers will write the program like this to eliminate the first statement:

**for** i **in** range(5):  
 score = float(input(**"Enter score: "**))  
 new\_score = score + 10  
 print(**"New score:"**, new\_score)

Here is another example:

score = [85, 77, 64, 80]  
i = 0  
print(**"New scores:"**)  
**while** i < 4:  
 score[i] = score[i] + 5  
 print(score[i])  
 i = i + 1

This program uses a counter-controlled while loop to add extra points and display new scores. We can rewrite it with for loop and range function:

score = [85, 77, 64, 80]  
print(**"New scores:"**)  
**for** i **in** range(4):  
 score[i] = score[i] + 5  
 print(score[i])

The following is the output of the program:

New scores:

90

82

69

85

# **5.10 Generating Random Integers**

Random numbers are needed in many applications. In this section, you will see how to generate a sequence of random integers in Python.

First, let’s see how to generate one random integer. The following is an example:

**import** random  
  
num = random.randint(1, 20)

The first line is an import statement. These type of statements import modules into a program and they are usually placed right at the beginning of the code file. The module we want to import here is the random module. This module has a collection of functions that generate different types of random values. We will explain more about import statements in a later lesson.

Once the random module is imported, we can use the function random.randint to generate a random integer in a specified range. The two numbers inside the parenthesis are the starting and ending limits. The syntax random.randint(1, 20) generates a random integer that can be as small as 1 and as large as 20.

We can use a loop to generate multiple random numbers. For example, the following program uses a for loop to generate and display 4 random integers in the range of 5 to 15:

**import** random  
  
**for** i **in** range(4):  
 num = random.randint(5, 15)  
 print(num)

The following shows a sample test run of the program:

8

5

11

9