3

Control Flow Statements

AIM

Understand if, if…else and if…elif…else statements and use of control flow statements that provide a means for looping over a section of code multiple times within the program.

LEARNING OUTCOMES

At the end of this chapter, you are expected to

- Use if, if…else and if…elif…else statements to transfer the control from one part of the program to another.
- Write while and for loops to run one or more statements repeatedly.
- Control the flow of execution using *break* and *continue* statements.
- Improve the reliability of code by incorporating exception handling mechanisms through try-except blocks.

Python supports a set of control flow statements that you can integrate into your program. The statements inside your Python program are generally executed *sequentially* from top to bottom, in the order that they appear. Apart from sequential control flow statements you can employ decision making and looping control flow statements to break up the flow of execution thus enabling your program to conditionally execute particular blocks of code. The term *control flow* details the direction the program takes.

The control flow statements (FIGURE 3.1) in Python Programming Language are

1. **Sequential Control Flow Statements:** This refers to the line by line execution, in which the statements are executed sequentially, in the same order in which they appear in the program.
2. **Decision Control Flow Statements:** Depending on whether a condition is True or False, the decision structure may skip the execution of an entire block of statements or even execute one block of statements instead of other (if, if…else and if…elif…else).
3. **Loop Control Flow Statements:** This is a control structure that allows the execution of a block of statements multiple times until a loop termination condition is met (*for* loop and *while* loop). Loop Control Flow Statements are also called Repetition statements or Iteration statements.

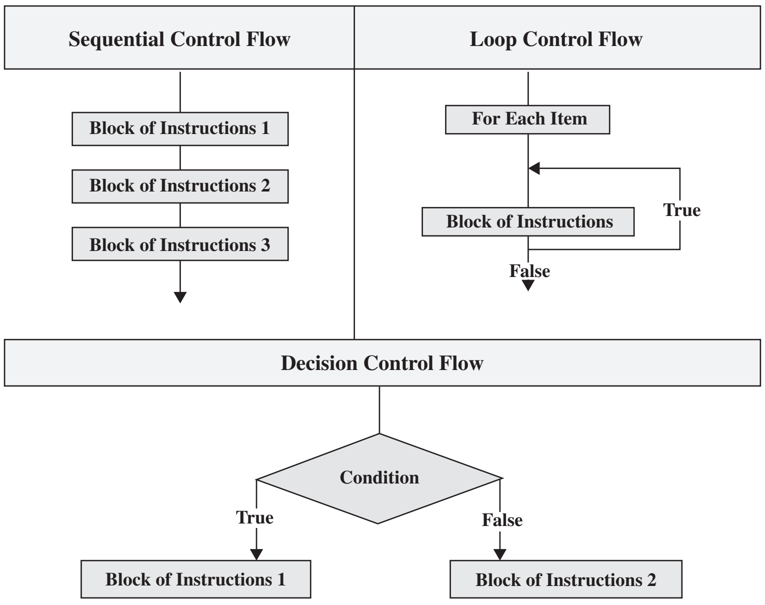
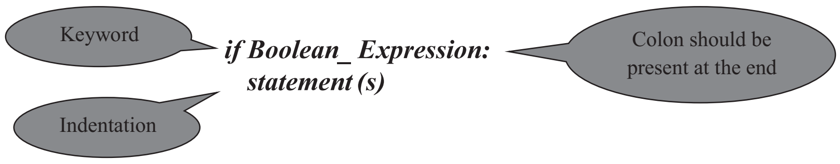


FIGURE 3.1
Forms of control flow statements.

3.1 The *if* Decision Control Flow Statement

The syntax for ***if*** statement is,



if Boolean_ Expression:
    statement (s)

The *if* decision control flow statement starts with *if* keyword and ends with a colon. The expression in an *if* statement should be a Boolean expression. The *if* statement decides whether to run some particular statement or not depending upon the value of the Boolean expression. If the Boolean expression evaluates to *True* then statements in the *if* block will be executed; otherwise the result is *False* then none of the statements are executed. In Python, the *if* block statements are determined through indentation and the first unindented statement marks the end. You don't need to use the == operator explicitly to check if a variable's value evaluates to *True* as the variable name can itself be used as a condition.

For example,

```
>>> if 20 > 10:
...      print(f"20 is greater than 10")
```

Output

```
20 is greater than 10
```

In ①, the Boolean expression 20 > 10 is evaluated to Boolean True and the print statement ② is executed.

Program 3.1: Program Reads a Number and Prints a Message If It Is Positive

```
1. number = int(input("Enter a number"))
2. if number >= 0:
3.     print(f"The number entered by the user is a positive number")
```

Output

```
Enter a number 67
The number entered by the user is a positive number
```

The value entered by the user is read and stored in the *number* variable ①, the value in the *number* variable ② is checked to determine if it is greater than or equal to 0, if *True* then print the message ③.

Program 3.2: Program to Read Luggage Weight and Charge the Tax Accordingly

```
1. weight = int(input("How many pounds does your suitcase weigh?"))
2. if weight > 50:
3.     print(f"There is a $25 surcharge for heavy luggage")
4.     print(f"Thank you")
```

Output

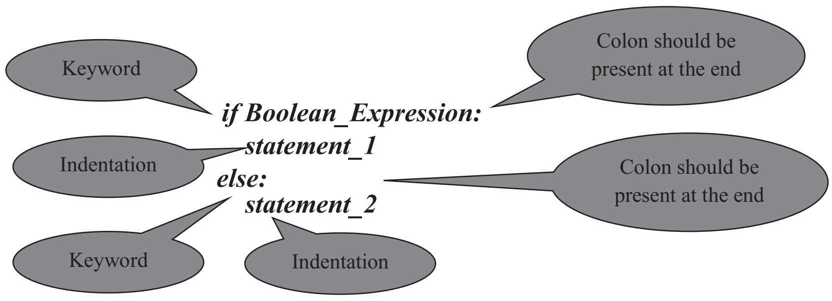
```
How many pounds does your suitcase weigh? 75
There is a $25 surcharge for heavy luggage
Thank you
```

The *weight* of the luggage is read ① and if it is greater than 50 ② then extra charges are collected. Lines ③ and ④ are present in the indentation block of *if* statement. If the weight of the luggage is less than 50 then nothing is printed.

3.2 The *if…else* Decision Control Flow Statement

An *if* statement can also be followed by an *else* statement which is optional. An *else* statement does not have any condition. Statements in the *if* block are executed if the Boolean_Expression is *True*. Use the optional *else* block to execute statements if the Boolean_Expression is *False*. The ***if…else*** statement allows for a two-way decision.

The syntax for ***if…else*** statement is,


Keyword
Colon should be present at the end
Indentation
Colon should be present at the end
Keyword
Indentation

```
if Boolean_Expression:
    statement_1
else:
    statement_2
```

If the Boolean_Expression evaluates to *True*, then statement_1 is executed, otherwise it is evaluated to *False* then statement_2 is executed. Indentation is used to separate the blocks. After the execution of either statement_1 or statement_2, the control is transferred to the next statement after the *if* statement. Also, *if* and *else* keywords should be aligned at the same column position.

Here, statement, statement_1, statement_2 and so on can be either a single statement or multiple statements. Boolean_Expression, Boolean_Expression_1, Boolean_Expression_2 and so on are expressions of the Boolean type which gets evaluated to either *True* or *False*.

Program 3.3: Program to Find If a Given Number Is Odd or Even

```
1. number = int(input("Enter a number"))
2. if number % 2 == 0:
3.     print(f"{number} is Even number")
4. else:
5.     print(f"{number} is Odd number")
```

Output

```
Enter a number: 45
45 is Odd number
```

A number is read and stored in the variable named *number* ①. The *number* is checked using modulus operator ② to determine whether the *number* is perfectly divisible by 2 or not. If the *number* is perfectly divisible by 2, then the expression is evaluated to *True* and *number* is even ③. However, ④ if the expression evaluates to *False*, the *number* is odd ⑤.

Program 3.4: Program to Find the Greater of Two Numbers

```
1. number_1 = int(input("Enter the first number"))
2. number_2 = int(input("Enter the second number"))
```

```
3. if number_1 > number_2:
4.     print(f"{number_1} is greater than {number_2}")
5. else:
6.     print(f"{number_2} is greater than {number_1}")
```

Output

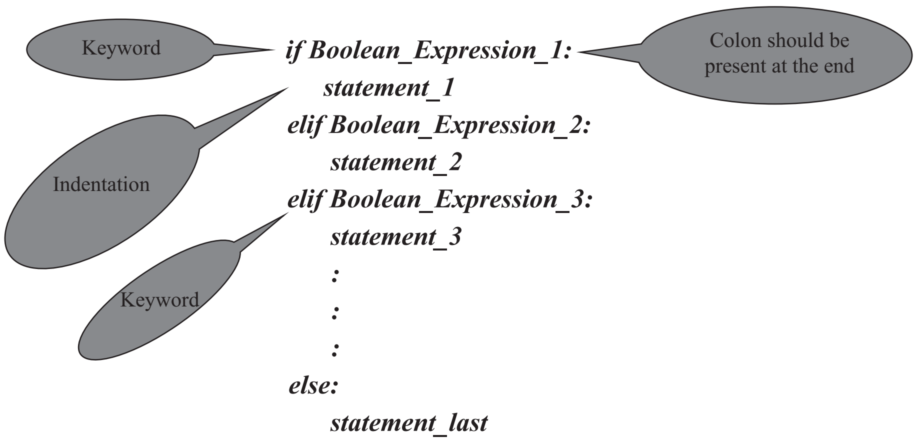
```
Enter the first number 8
Enter the second number 10
10 is greater than 8
```

Two numbers are read using the input function and the values are stored in the variables *number_1* ① and *number_2* ②. The Boolean expression is evaluated ③ and if it is *True*, then line ④ is executed else ⑤ if the Boolean expression is evaluated to *False*, then line ⑥ is executed.

3.3 The *if…elif…else* Decision Control Statement

The ***if…elif…else*** is also called as multi-way decision control statement. When you need to choose from several possible alternatives, then an *elif* statement is used along with an *if* statement. The keyword *'elif'* is short for 'else if' and is useful to avoid excessive indentation. The *else* statement must always come last, and will again act as the default action.

The syntax for ***if…elif…else*** statement is,



```
if Boolean_Expression_1:
    statement_1
elif Boolean_Expression_2:
    statement_2
elif Boolean_Expression_3:
    statement_3
    :
    :
    :
else:
    statement_last
```

This *if…elif…else* decision control statement is executed as follows:

- In the case of multiple Boolean expressions, only the first logical Boolean expression which evaluates to True will be executed.
- If Boolean_Expression_1 is *True*, then statement_1 is executed.
- If Boolean_Expression_1 is *False* and Boolean_Expression_2 is *True*, then statement_2 is executed.

- If Boolean_Expression_1 and Boolean_Expression_2 are *False* and Boolean_Expression_3 is *True*, then statement_3 is executed and so on.
- If none of the Boolean_Expression is *True*, then statement_last is executed.

There can be zero or more *elif* parts each followed by an indented block, and the *else* part is optional. There can be only one *else* block. An *if…elif…else* statement is a substitute for the switch or case statements found in other programming languages.

Program 3.5: Write a Program to Prompt for a Score between 0.0 and 1.0. If the Score Is Out of Range, Print an Error. If the Score Is between 0.0 and 1.0, Print a Grade Using the Following Table

Score	Grade
>= 0.9	A
>= 0.8	B
>= 0.7	C
>= 0.6	D
< 0.6	F

```
1. score = float(input("Enter your score"))
2. if score < 0 or score > 1:
3.     print('Wrong Input')
4. elif score >= 0.9:
5.     print('Your Grade is "A" ')
6. elif score >= 0.8:
7.     print('Your Grade is "B" ')
8. elif score >= 0.7:
9.     print('Your Grade is "C" ')
10. elif score >= 0.6:
11.     print('Your Grade is "D" ')
12. else:
13.     print('Your Grade is "F" ')
```

Output

```
Enter your score0.92
Your Grade is "A"
```

A number is read and is assigned to the variable *score* ①. If the *score* value is greater than 1 or less than 0 ② then we display an error message indicating to the user that it is a wrong input ③. If not, the *score* value is checked for different conditions based on the score table and the grade statements are printed accordingly ④–⑬.

Program 3.6: Program to Display the Cost of Each Type of Fruit

```
 1. fruit_type = input("Enter the Fruit Type:")
 2. if fruit_type == "Oranges":
 3.     print('Oranges are $0.59 a pound')
 4. elif fruit_type == "Apples":
 5.     print('Apples are $0.32 a pound')
 6. elif fruit_type == "Bananas":
 7.     print('Bananas are $0.48 a pound')
 8. elif fruit_type == "Cherries":
 9.     print('Cherries are $3.00 a pound')
10. else:
11.     print(f'Sorry, we are out of {fruit_type}')
```

Output

```
Enter the Fruit Type: Cherries
Cherries are $3.00 a pound
```

A string value is read and assigned to the variable *fruit_type* ①. The value of the *fruit_type* variable is checked against different strings ②–⑨. If the *fruit_type* value matches with the existing string, then a message is printed else inform the user of the unavailability of the fruit ⑩–⑪.

3.4 Nested *if* Statement

In some situations, you have to place an *if* statement inside another statement. An *if* statement that contains another *if* statement either in its *if* block or *else* block is called a Nested *if* statement.

The syntax of the nested *if* statement is,

```
               if Boolean_Expression_1:
Nested if  →      if Boolean_Expression_2:
                      statement_1
                  else:
                      statement_2
               else:
                   statement_3
```

If the Boolean_Expression_1 is evaluated to *True*, then the control shifts to Boolean_Expression_2 and if the expression is evaluated to *True*, then statement_1 is executed, if the Boolean_Expression_2 is evaluated to *False* then the statement_2 is executed. If the Boolean_Expression_1 is evaluated to *False*, then statement_3 is executed.

Program 3.7: Program to Check If a Given Year Is a Leap Year

```
 1. year = int(input('Enter a year'))
 2. if year % 4 == 0:
 3.     if year % 100 == 0:
 4.         if year % 400 == 0:
 5.             print(f'{year} is a Leap Year')
 6.         else:
 7.             print(f'{year} is not a Leap Year')
 8.     else:
 9.         print(f'{year} is a Leap Year')
10. else:
11.     print(f'{year} is not a Leap Year')
```

Output

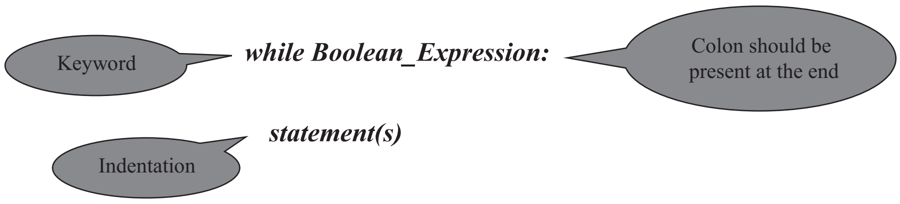
```
Enter a year 2014
2014 is not a Leap Year
```

All years which are perfectly divisible by 4 are leap years except for century years (years ending with 00) which is a leap year only it is perfectly divisible by 400. For example, years like 2012, 2004, 1968 are leap years but 1971, 2006 are not leap years. Similarly, 1200, 1600, 2000, 2400 are leap years but 1700, 1800, 1900 are not.

Read the value for *year* as input ①. Check whether the given *year* is divisible by 4 ② and also by 400 ④. If the condition is *True*, then the *year* is a leap year ⑤. If the *year* is divisible by 4 and not divisible by 100 ⑧ then the *year* is a leap year ⑨. If the condition at ② or ④ becomes *False*, then the *year* is not a leap year ⑦ and ⑪.

3.5 The *while* Loop

The syntax for *while* loop is,



The *while* loop starts with the *while* keyword and ends with a colon. With a *while* statement, the first thing that happens is that the Boolean expression is evaluated before the statements in the *while* loop block is executed. If the Boolean expression evaluates to *False*, then the statements in the *while* loop block are never executed. If the Boolean expression evaluates to *True*, then the *while* loop block is executed. After each iteration of the loop block, the Boolean expression is again checked, and if it is *True*, the loop is iterated again.

Each repetition of the loop block is called an iteration of the loop. This process continues until the Boolean expression evaluates to *False* and at this point the *while* statement exits. Execution then continues with the first statement after the *while* loop.

Program 3.8: Write Python Program to Display First 10 Numbers Using *while* Loop Starting from 0

```
1. i = 0
2. while i < 10:
3.     print(f"Current value of i is {i}")
4.     i = i + 1
```

Output

```
Current value of i is 0
Current value of i is 1
Current value of i is 2
Current value of i is 3
Current value of i is 4
Current value of i is 5
Current value of i is 6
Current value of i is 7
Current value of i is 8
Current value of i is 9
```

Variable *i* is assigned with 0 outside the loop ①. The expression *i < 10* is evaluated ②. If the value of *i* is less than 10 (i.e., *True*) then the body of the loop is executed. Value of *i* is printed ③ and *i* is incremented by 1 ④. This continues until the expression in *while* loop becomes *False*.

Program 3.9: Write a Program to Find the Average of *n* Natural Numbers Where *n* Is the Input from the User

```
 1. number = int(input("Enter a number up to which you want to find the average"))
 2. i = 0
 3. sum = 0
 4. count = 0
 5. while i < number:
 6.     i = i + 1
 7.     sum = sum + i
 8.     count = count + 1
 9. average = sum/count
10. print(f"The average of {number} natural numbers is {average}")
```

Output

```
Enter a number up to which you want to find the average 5
The average of 5 natural numbers is 3.0
```

The variables *i*, *sum* and *count* are assigned with zero ②, ③, ④. The expression *i* < *number* is evaluated ⑤. Since it is *True* for the first iteration, the body of the *while* loop gets executed. Variable *i* gets incremented ⑥ by value of 1 and it generates the required natural numbers. The *sum* variable adds the value of *sum* variable with the value of *i* variable ⑦, while the *count* variable keeps track of number of times the body of the loop gets executed ⑧. The loop gets repeated until the test expression becomes *False*. The *average* is calculated as *sum*/*count* ⑨ and displayed ⑬.

Program 3.10: Program to Find the GCD of Two Positive Numbers

```
 1. m = int(input("Enter first positive number"))
 2. n = int(input("Enter second positive number"))
 3. if m == 0 and n == 0:
 4.     print("Invalid Input")
 5. if m == 0:
 6.     print(f"GCD is {n}")
 7. if n == 0:
 8.     print(f"GCD is {m}")
 9. while m != n:
10.     if m > n:
11.         m = m – n
12.     if n > m:
13.         n = n – m
14. print(f"GCD of two numbers is {m}")
```

Output

```
Enter first positive number8
Enter second positive number12
GCD of two numbers is 4
```

Read the value for *m* and *n* ①–②. If both *m* and *n* are zero then it is invalid input because zero cannot be divided by zero which is indeterminate ③–④. If either *m* or *n* is zero then the other one is gcd ⑤–⑧. If the value of $m > n$ then $m = m - n$ or if $n > m$ then $n = n - m$. The logic in line ⑩–⑬ is repeated until the value of *m* is equal to the value of *n* ⑨. Then gcd will be either *m* or *n* ⑭.

Program 3.11: Write Python Program to Find the Sum of Digits in a Number

```
1. number = int(input('Enter a number'))
2. result = 0
3. remainder = 0
4. while number != 0:
5.     remainder = number % 10
```

```
6.     result = result + remainder
7.     number = int(number / 10)
8. print(f"The sum of all digits is {result}")
```

Output

```
Enter a number1234
The sum of all digits is 10
```

Read a number from user ① and store it in a variable called *number*. Assign zero to the variables *result* and *remainder* ②–③. Find the last digit of the number. To get the last digit of the *number* use modulus division by 10 and assign it the variable *remainder* ⑤. Add the last digit that you obtained with the *result* variable ⑥. Then remove the last digit from the *number* by dividing the *number* by 10 and cast it as int ⑦. Repeat the logic in line ⑤–⑦ till the variable *number* becomes 0 ④. Finally, you will be left with the sum of digits in the *result* variable ⑧.

Program 3.12: Write a Program to Display the Fibonacci Sequences up to *n*th Term Where *n* is Provided by the User

```
nterms = int(input('How many terms?'))
current = 0
previous = 1
count = 0
next_term = 0
if nterms <= 0:
    print('Please enter a positive number')
elif nterms == 1:
    print('Fibonacci sequence')
    print('0')
else:
    print("Fibonacci sequence")
    while count < nterms:
        print(next_term)
        current = next_term
        next_term = previous + current
        previous = current
        count += 1
```

Output

```
How many terms? 5
Fibonacci sequence
0
1
1
2
3
```

In a Fibonacci sequence, the next number is obtained by adding the previous two numbers. The first two numbers of the Fibonacci sequence are 0 and 1. The next number is obtained by adding 0 and 1 which is 1. Again, the next number is obtained by adding 1 and 1 which is 2 and so on. Get a number from user ① up to which you want to generate the Fibonacci sequence. Assign values to variables *current*, *previous*, *next_term* and *count* ②–⑤. The variable *count* keeps track of number of times the *while* block is executed. User is required to enter a positive number to generate the Fibonacci sequence ⑥–⑦. If the user asks to generate a single number in the sequence, then print zero ⑧–⑩. The *next_term* is obtained ⑭ by adding the *previous* and *current* variables and the logic from ⑮–⑱ is repeated until *while* block conditional expression becomes *False* ⑬.

Program 3.13: Program to Repeatedly Check for the Largest Number Until the User Enters "done"

```
1. largest_number = int(input("Enter the largest number initially"))
2. check_number = input("Enter a number to check whether it is largest or not")
3. while check_number != "done":
4.     if largest_number > int(check_number):
5.         print(f"Largest Number is {largest_number}")
6.     else:
7.         largest_number = int(check_number)
8.         print(f"Largest Number is {largest_number}")
9.     check_number = input("Enter a number to check whether it is largest or not")
```

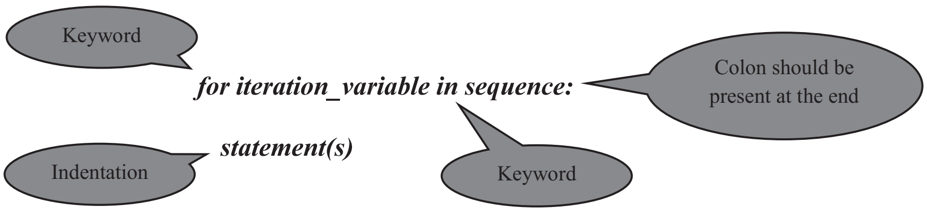
Output

```
Enter the largest number initially 5
Enter a number to check whether it is largest or not 1
Largest Number is 5
Enter a number to check whether it is largest or not 10
Largest Number is 10
Enter a number to check whether it is largest or not 8
Largest Number is 10
Enter a number to check whether it is largest or not done
```

A number is read initially which is assumed to be the *largest_number* ①. Then, the user is prompted to enter another number which is assigned to variable *check_number* ②. Within the *while* loop ③ the value of *check_variable* is compared with that of *largest_ variable* ④–⑤. If the *check_variable* has a larger value then that value is assigned to *largest_ variable* ⑥–⑧. The user is again prompted to enter another value which is compared against the *largest_number* value ⑨. This continues until the user enters the string "done" instead of a numerical value.

3.6 The *for* Loop

The syntax for the *for* loop is,



```
for iteration_variable in sequence:
    statement(s)
```

The *for* loop starts with *for* keyword and ends with a colon. The first item in the sequence gets assigned to the iteration variable *iteration_variable*. Here, *iteration_variable* can be any valid variable name. Then the statement block is executed. This process of assigning items from the sequence to the *iteration_variable* and then executing the statement continues until all the items in the sequence are completed.

We take the liberty of introducing you to *range()* function which is a built-in function at this stage as it is useful in demonstrating *for* loop. The *range()* function generates a sequence of numbers which can be iterated through using *for* loop. The syntax for *range()* function is,

range([start ,] stop [, step])

Both start and step arguments are optional and the range argument value should always be an integer.

start → value indicates the beginning of the sequence. If the start argument is not specified, then the sequence of numbers start from zero by default.

stop → Generates numbers up to this value but not including the number itself.

step → indicates the difference between every two consecutive numbers in the sequence. The step value can be both negative and positive but not zero.

NOTE: The square brackets in the syntax indicate that these arguments are optional. You can leave them out.

Program 3.14: Demonstrate for Loop Using *range()* Function

```
print("Only "stop" argument value specified in range function")
for i in range(3):
    print(f"{i}")
print("Both "start" and "stop" argument values specified in range function")
for i in range(2, 5):
    print(f"{i}")
print("All three arguments "start", "stop" and "step" specified in range function")
for i in range(1, 6, 3):
    print(f"{i}")
```

Output

```
Only "stop" argument value specified in range function
0
1
2
Both "start" and "stop" argument values specified in range function
2
3
4
All three arguments "start", "stop" and "step" specified in range function
1
4
```

The function *range(3)* generates numbers starting from 0 to 2 ②. During the first iteration, the 0th value gets assigned to the iteration variable *i* and the same gets printed out ③. This continues to execute until all the numbers generated using *range()* function are assigned to the variable *i*. The function *range(2, 5)* ⑤ generates a sequence of numbers starting from 2 to 4 and the function *range(1, 6, 3)* generates ⑧ all the numbers starting from 1 up to 5 but the difference between each number is 2.

Program 3.15: Program to Iterate through Each Character in the String Using *for* Loop

```
1. for each_character in "Blue":
2.     print(f"Iterate through character {each_character} in the string 'Blue'")
```

Output

```
Iterate through character B in the string 'Blue'
Iterate through character l in the string 'Blue'
Iterate through character u in the string 'Blue'
Iterate through character e in the string 'Blue'
```

The iteration variable *each_character* is used to iterate through each character of the string "Blue" ① and each character is printed out in separate line ②.

Program 3.16: Write a Program to Find the Sum of All Odd and Even Numbers up to a Number Specified by the User.

```
1. number = int(input("Enter a number"))
2. even = 0
3. odd = 0
4. for i in range(number):
5.     if i % 2 == 0:
6.         even = even + i
7.     else:
8.         odd = odd + i
9. print(f"Sum of Even numbers are {even} and Odd numbers are {odd}")
```

Output

```
Enter a number 10
Sum of Even numbers are 20 and Odd numbers are 25
```

A range of numbers are generated using *range()* function ④. The numbers are segregated as odd or even by using the modulus operator ⑤. All the even numbers are added up and assigned to *even* variable and odd numbers are added up and assigned to *odd* variable ⑥–⑧ and print the result ⑨.

Program 3.17: Write a Program to Find the Factorial of a Number

```
 1. number = int(input('Enter a number'))
 2. factorial = 1
 3. if number < 0:
 4.     print("Factorial doesn't exist for negative numbers")
 5. elif number == 0:
 6.     print('The factorial of 0 is 1')
 7. else:
 8.     for i in range(1, number + 1):
 9.         factorial = factorial * i
10. print(f"The factorial of number {number} is {factorial}")
```

Output

```
Enter a number 5
The factorial of number 5 is 120
```

The factorial of a non-negative integer n is denoted by n! which is the product of all positive integers less than or equal to n i.e., n! = n * (n – 1) * (n – 2) * (n – 3)… 3 * 2 * 1. For example,

$5! = 5 \times 4 \times 3 \times 2 \times 1 = 120.$

The value of 0! is 1

Read a *number* from user ①. A value of 1 is assigned to variable *factorial* ②. To find the *factorial* of a number it has to be checked for a non-negative integer ③–④. If the user entered number is zero then the factorial is 1 ⑤–⑥. To generate numbers from 1 to the user entered number *range()* function is used. Every number is multiplied with the *factorial* variable and is assigned to the *factorial* variable itself inside the *for* loop ⑦–⑨. The *for* loop block is repeated for all the numbers starting from 1 up to the user entered number. Finally, the factorial value is printed ⑩.

3.7 The *continue* and *break* Statements

The *break* and *continue* statements provide greater control over the execution of code in a loop. Whenever the *break* statement is encountered, the execution control immediately jumps to the first instruction following the loop. To pass control to the next iteration without exiting the loop, use the *continue* statement. Both *continue* and *break* statements can be used in *while* and *for* loops.

Program 3.18: Program to Demonstrate Infinite *while* Loop and *break*

```
1. n = 0
2. while True:
3.     print(f"The latest value of n is {n}")
4.     n = n + 1
```

Here the *while* loop evaluates to *True* logical value always ② and it prints the latest value ③–④. This is an infinite loop with no end in sight. You need to press Ctrl + C to terminate this program. One way of ending this infinite loop is to use *break* statement along with *if* condition as shown in the following code.

```
1. n = 0
2. while True:
3.     print(f"The latest value of n is {n}")
4.     n = n + 1
5.     if n > 5:
6.         print(f"The value of n is greater than 5")
7.         break
```

Output

```
The latest value of n is 0
The latest value of n is 1
The latest value of n is 2
The latest value of n is 3
The latest value of n is 4
The latest value of n is 5
The value of n is greater than 5
```

While this is an infinite loop ①–④, you can use this pattern to build useful loops as long as you explicitly add code to the body of the loop to ensure to exit from the loop using *break* statement upon satisfying a condition ⑤–⑦.

Program 3.19: Write a Program to Check Whether a Number Is Prime or Not

```
1. number = int(input('Enter a number > 1: '))
2. prime = True
3. for i in range(2, number):
4.     if number % i == 0:
5.         prime = False
6.         break
```

```
 7. if prime:
 8.     print(f"{number} is a prime number")
 9. else:
10.     print(f"{number} is not a prime number")
```

Output

```
Enter a number > 1: 7
7 is a prime number
```

A prime number is a number which is divided by one and itself. For example, the number 7 is prime as it can be divided by 1 and itself, while number 10 can be divided by 2 and 5 other than 1 and itself, thus 10 can't be a prime number.

The user shall enter a value greater than one ①. Initially, the variable *prime* is assigned with *True* Boolean value ②. Use *range()* function to generate numbers starting from 2 up to number – 1, excluding the user entered *number* ③. The user entered *number* is checked using modulus operator ④ to determine whether the *number* is perfectly divisible by any number other than 1 and by itself. If it is divisible, then the variable *prime* is assigned *False* Boolean value ⑤ and we break out of the loop ⑥. But after completing the loop if *prime* remains *True* ⑦ then the user entered *number* is prime number ⑧ else ⑨ it is not a prime number ⑩.

Program 3.20: Program to Demonstrate *continue* Statement

```
1. n = 10
2. while n > 0:
3.     print(f"The current value of number is {n}")
4.     if n == 5:
5.         print(f"Breaking at {n}")
6.         n = 10
7.         continue
8.     n = n – 1
```

Output

```
The current value of number is 10
The current value of number is 9
The current value of number is 8
The current value of number is 7
The current value of number is 6
The current value of number is 5
Breaking at 5
The current value of number is 10
The current value of number is 9
The current value of number is 8
The current value of number is 7
The current value of number is 6
The current value of number is 5
```

In the above program, the while block is executed when the value of *n* is greater than zero ①–②. The value in the variable *n* is decremented ⑧ and printed in descending order ③ and when *n* becomes five ④–⑦ the control goes back to the beginning of the loop.

3.8 Catching Exceptions Using *try* and *except* Statement

There are at least two distinguishable kinds of errors:

1. Syntax Errors
2. Exceptions

3.8.1 Syntax Errors

Syntax errors, also known as parsing errors, are perhaps the most common kind of complaint you get while you are still learning Python. For example,

```
1. while True
2.     print("Hello World)
```

Output

```
File "<ipython-input-3-c231969faf4f>", line 1
  while True
            ^
SyntaxError: invalid syntax
```

In the output, the offending line is repeated and displays a little 'arrow' pointing at the earliest point in the line where the error was detected ①. The error is caused by a missing colon (':'). File name and line number are also printed so you know where to look in case the input came from a Python program file.

3.8.2 Exceptions

Exception handling is one of the most important feature of Python programming language that allows us to handle the errors caused by exceptions. Even if a statement or expression is syntactically correct, it may cause an error when an attempt is made to execute it. Errors detected during execution are called exceptions.

An exception is an unwanted event that interrupts the normal flow of the program. When an exception occurs in the program, execution gets terminated. In such cases, we get a system-generated error message. However, these exceptions can be handled in Python. By handling the exceptions, we can provide a meaningful message to the user about the issue rather than a system-generated message, which may not be understandable to the user.

Exceptions can be either built-in exceptions or user-defined exceptions. The interpreter or built-in functions can generate the built-in exceptions while user-defined exceptions are custom exceptions created by the user.

When the exceptions are not handled by programs it results in error messages as shown below.

1.
```
>>> 10 * (1/0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
```
2.
```
>>> 4 + spam*3
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'spam' is not defined
```
3.
```
>>> '2' + 2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Can't convert 'int' object to str implicitly
```

The last line of the error message indicates what happened. Exceptions come in different types, and the type is printed as part of the message: the types in the example are *ZeroDivisionError* ①, *NameError* ② and *TypeError* ③. The string printed as the exception type is the name of the *built-in exception* that occurred. The preceding part of the error message shows the context where the exception happened, in the form of a stack traceback.

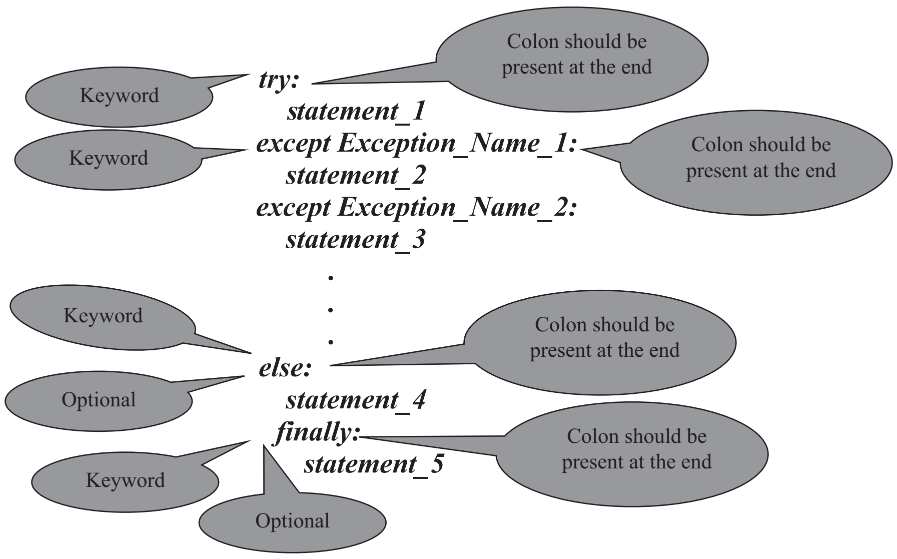
3.8.3 Exception Handling Using *try…except…finally*

Handling of exception ensures that the flow of the program does not get interrupted when an exception occurs which is done by trapping run-time errors. Handling of exceptions results in the execution of all the statements in the program.

Run-time errors are those errors that occur during the execution of the program. These errors are not detected by the Python interpreter, because the code is syntactically correct.

It is possible to write programs to handle exceptions by using ***try…except…finally*** statements.

The syntax for ***try…except…finally*** is,



```
try:
   statement_1
except Exception_Name_1:
   statement_2
except Exception_Name_2:
   statement_3
      .
      .
      .
else:
   statement_4
finally:
   statement_5
```

Keyword
Colon should be present at the end
Keyword
Colon should be present at the end
Keyword
Colon should be present at the end
Optional
Colon should be present at the end
Keyword
Optional

A *try* block consisting of one or more statements is used by Python programmers to partition code that might be affected by an exception. The associated *except* blocks are used to handle any resulting exceptions thrown in the *try* block. That is, you want the *try* block to succeed, and if it does not succeed, you want the control to pass to the *catch* block. If any statement within the *try* block throws an exception, control immediately shifts to the *catch* block. If no exception is thrown in the *try* block, the *catch* block is skipped.

There can be one or more *except* blocks. Multiple *except* blocks with different exception names can be chained together. The *except* blocks are evaluated from top to bottom in your code, but only one *except* block is executed for each exception that is thrown. The first *except* block that specifies the exact exception name of the thrown exception is executed. If no *except* block specifies a matching exception name then an *except* block that does not have an exception name is selected, if one is present in the code. Instead of having multiple except blocks with multiple exception names for different exceptions, you can combine multiple exception names together separated by a comma (also called parenthesized tuples) in a single except block. The syntax for combining multiple exception names in an except block is,

```
except (Exception_Name_1, Exception_Name_2, Exception_Name_3):
        statement(s)
```

where Exception_Name_1, Exception_Name_2 and Exception_Name_3 are different exception names.

You shall learn more about tuples in Chapter 8.

The *try…except* statement has an optional *else* block, which, when present, must follow all except blocks. It is useful for code that must be executed if the *try* block does not raise an exception. The use of the *else* block is better than adding additional code to the *try* block because it avoids accidentally catching an exception that wasn't raised by the code being protected by the *try…except* statement.

The *try* statement has another optional block which is intended to define clean-up actions that must be executed under all circumstances. A *finally* block is always executed before leaving the *try* statement, whether an exception has occurred or not. When an exception has occurred in the *try* block and has not been handled by an *except* block, it is re-raised after the *finally* block has been executed. The *finally* clause is also executed "on the way out" when any other clause of the *try* statement is left via a *break*, *continue* or *return* statement.

You can also leave out the name of the exception after the *except* keyword. This is generally not recommended as the code will now be catching different types of exceptions and handling them in the same way. This is not optimal as you will be handling a TypeError exception the same way as you would have handled a ZeroDivisionError exception. When handling exceptions, it is better to be as specific as possible and only catch what you can handle.

Program 3.21: Program to Check for *ValueError* Exception

```
1. while True:
2.     try:
3.         number = int(input("Please enter a number: "))
4.         print(f"The number you have entered is {number}")
5.         break
6.     except ValueError:
7.         print("Oops! That was no valid number. Try again…")
```

Output

```
Please enter a number: g
Oops! That was no valid number. Try again…
Please enter a number: 4
The number you have entered is 4
```

First, the *try* block (the statement(s) between the try and except keywords) is executed ②–⑤ inside the *while* loop ①. If no exception occurs, the *except* block is skipped and execution of the *try* statement is finished. If an exception occurs during execution of the *try* block statements, the rest of the statements in the *try* block is skipped. Then if its type matches the exception named after the *except* keyword, the *except* block is executed ⑥–⑦, and then execution continues after the *try* statement. When a variable receives an inappropriate value then it leads to *ValueError* exception.

Program 3.22: Program to Check for *ZeroDivisionError* Exception

```
1. x = int(input("Enter value for x: "))
2. y = int(input("Enter value for y: "))
```

```
3. try:
4.     result = x / y
5. except ZeroDivisionError:
6.     print("Division by zero!")
7. else:
8.     print(f"Result is {result}")
9. finally:
10.    print("Executing finally clause")
```

Output

```
Case 1
Enter value for x: 8
Enter value for y: 0
Division by zero!
Executing finally clause
Case 2
Enter value for x: p
Enter value for y: q
Executing finally clause
ValueError Traceback (most recent call last)
<ipython-input-16-271d1f4e02e8> in <module>()
ValueError: invalid literal for int() with base 10: 'p'
```

In the above example divide by zero exception is handled. In line ① and ②, the user entered values are assigned to *x* and *y* variables. Line ④ is enclosed within line ③ which is a *try* clause. If the statements enclosed within the *try* clause raise an exception then the control is transferred to line ⑤ and divide by zero error is printed ⑥. As can be seen in Case 1, *ZeroDivisionError* occurs when the second argument of a division or modulo operation is zero. If no exception occurs, the except block is skipped and result is printed out ⑦–⑧. In Case 2, as you can see, the *finally* clause is executed in any event ⑨–⑩. The *ValueError* raised by dividing two strings is not handled by the *except* clause and therefore re-raised after the *finally* clause has been executed.

Program 3.23: Write a Program Which Repeatedly Reads Numbers Until the User Enters 'done'. Once 'done' Is Entered, Print Out the Total, Count, and Average of the Numbers. If the User Enters Anything Other Than a Number, Detect Their Mistake Using *try* and *except* and Print an Error Message and Skip to the Next Number

```
total = 0
count = 0
while True:
    num = input("Enter a number: ")
    if num == 'done':
        print(f"Sum of all the entered numbers is {total}")
        print(f"Count of total numbers entered {count}")
        print(f"Average is {total / count}")
```

```
 9.         break
10.     else:
11.         try:
12.             total += float(num)
13.         except:
14.             print("Invalid input")
15.             continue
16.         count += 1
```

Output

```
Enter a number: 1
Enter a number: 2
Enter a number: 3
Enter a number: 4
Enter a number: 5
Enter a number: done
Sum of all the entered numbers is 15.0
Count of total numbers entered 5
Average is 3.0
```

The program prompts the user ④ to enter a series of numbers until the user enters the word *"done"* ③. Assign zero to the variables *total* and *count* ①–②. Check whether the user has entered the word "done" or a numerical value ⑤. If it is other than the word "done" then the value entered by the user is added to the *total* variable ⑩–⑫. If the value entered is a value other than numerical value and other than "done" string value then an exception is raised ⑬–⑭ and the program continues ⑮ with the next iteration prompting the user to enter the next value. The *count* variable ⑯ keeps track of number of times the user has entered a value. If the user enters "done" string value ⑤ then calculate and display the average ⑥–⑧. At this stage break from the loop and stop the execution of the program ⑨.

3.9 Summary

- An if statement always starts with *if* clause. It can also have one or more *elif* clauses and a concluding *else* clause, but those clauses are optional.
- When an error occurs at run time, an exception is thrown and exceptions must be handled to end the programs gracefully.
- Python allows try-except and can have multiple except blocks for a single try block. Developers can create and raise their own exceptions.
- The while statement loops through a block of statements until the condition becomes false.
- The for statement can loop through a block of statements once for each item sequence.
- A break statement breaks out of a loop by jumping to the end of it.
- A continue statement continues a loop by jumping to the start of it.