3 Control flow statements

A control flow statement in a computer program determines the individual lines of code to be executed and/or the order in which they will be executed. In this chapter, we'll learn about 3 types of control flow statements:

1. if-elif-else
2. for loop
3. while loop

3.1 Conditonal execution

The first type of control flow satement is `if-elif-else`. This statement helps with conditional execution of code, i.e., the piece of code to be executed is selected based on certain condition(s).

3.1.1 Comparison operators

For testing if conditions are true or false, first we need to learn the operators that can be used for comparison. For example, suppose we want to check if two objects are equal, we use the `==` operator:

```
5 == 6
```

```
False
```

```
x = "hi"
y = "hi"
x == y
```

```
True
```

Below are the python comparison operators and their meanings.

Python code	Meaning
x == y	Produce True if … x is equal to y
x != y	… x is not equal to y
x > y	… x is greater than y

x < y	… x is less than y
x >= y	… x is greater than or equal to y
x <= y	… x is less than or equal to y

3.1.2 Logical operators

Sometimes we may need to check multiple conditions simultaneously. The logical operator `and` is used to check if all the conditions are true, while the logical operator `or` is used to check if either of the conditons is true.

```
#Checking if both the conditions are true using 'and'
5 == 5 and 67 == 68
```

```
False
```

```
#Checking if either condition is true using 'or'
x = 6; y = 90
x < 0 or y > 50
```

```
True
```

3.1.3 if-elif-else statement

The `if-elif-else` statements can check several conditions, and execute the code corresponding to the condition that is true. Note that there can be as many `elif` statements as required.

Syntax: Python uses indentation to identify the code to be executed if a condition is true. All the code indented within a condition is executed if the condition is true.

Example: Input an integer. Print whether it is positive or negative.

```
number = input("Enter a number:")  #Input an integer
number_integer = int(number)       #Convert the integer to 'int
if number_integer > 0:               #Check if the integer is p
    print("Number is positive")
else:
    print("Number is negative")
```

```
Enter a number:-9
```

```
Number is negative
```

In the above code, note that anything entered by the user is taken as a string datatype by python. However, a string cannot be positive or negative. So, we converted the number input by the user to integer to check if it was positive or negative.

There may be multiple statements to be executed if a condition is true. See the example below.

Example: Input a number. Print whether it is positive, negative or zero. If it is negative, print its absolute value.

```
number = input("Enter a number:")
number_integer = int(number)
if number_integer > 0:
    print("Number is positive")
elif number_integer == 0:
    print("Number is zero")
else:
    print("Number is negative")
    print("Absolute value of number = ", abs(number_integer))
```

```
Enter a number:0
Number is zero
```

3.1.4 Practice exercise 1

Input a number. Print whether its odd or even.

Solution:

```
num = int(input("Enter a number: "))
if num%2 == 0:           #Checking if the number is divisible by
    print("Number is even")
else:
    print("Number is odd")
```

```
Enter a number: 5
Number is odd
```

3.1.5 Try-except

If we suspect that some lines of code may produce an error, we can put them in a `try` block, and if an error does occur, we can use the `except` block to instead

execute an alternative piece of code. This way the program will not stop if an error occurs within the `try` block, and instead will be directed to execute the code within the `except` block.

Example: Input an integer from the user. If the user inputs a valid integer, print whether it is a multiple of 3. However, if the user does not input a valid integer, print a message saying that the input is invalid.

```
num = input("Enter an integer:")

#The code lines within the 'try' block will execute as long as t
try:
    #Converting the input to integer, as user input is a string
    num_int = int(num)

    #checking if the integer is a multiple of 3
    if num_int % 3 == 0:
        print("Number is a multiple of 3")
    else:
        print("Number is not a multiple of 3")

#The code lines within the 'except' block will execute only if t
except:
    print("Input must be an integer")
```

```
Enter an integer:hi
Input must be an integer
```

3.1.6 Practice exercise 2

3.1.6.1

Ask the user to enter their exam score. Print the grade based on their score as follows:

Score	Grade
(90,100]	A
(80,90]	B
[0,80]	C

If the user inputs a score which is not a number in [0,100], print invalid entry.

Solution:

```
score = input("Enter exam score:")
try:

    #As exam score can be a floating point number (such as 90.5
    score_num = float(score)
    if score_num > 90 and score_num <= 100:
        print("Grade: A")
    elif score_num > 80 and score_num <= 90:
        print("Grade: B")
    elif score_num >= 0 and score_num <= 80:
        print("Grade: C")
    else:
        print("Invalid score")      #If a number is less than 0 (
except:
    print("Invalid input")          #If the input is not a numbe
```

```
Enter exam score:90
Grade: B
```

3.1.6.2

Nested if-elif-else statements: This question will lead you to create nested `if` statements, i.e., an `if` statement within another `if` statement.

Think of a number in [1,5]. Ask the user to guess the number.

- If the user guesses the number correctly, print “Correct in the first attempt!”, and stop the program. Otherwise, print “Incorrect! Try again” and give them another chance to guess the number.
- If the user guesses the number correctly in the second attempt, print “Correct in the second attempt”, otherwise print “Incorrect in both the attempts, the correct number is:”, and print the correct number.

Solution:

```
#Let us say we think of the number. Now the user has to guess th
rand_no = 3
guess = input("Guess the number:")
if int(guess)==rand_no:
    print("Correct in the first attempt!")

#If the guess is incorrect, the program will execute the code bl
else:
    guess = input("Incorrect! Try again:")
```

```
    if int(guess) == rand_no:
        print("Correct in the second attempt")
    else:
        print("Incorrect in the both the attempts, the correct r
```

3.2 Loops

With loops, a piece of code can be executed repeatedly for a fixed number of times or until a condition is satistified.

3.2.1 `for` loop

With a `for` loop, a piece of code is executed a fixed number of times.

We typically use `for` loops with an in-built python function called `range()` that supports `for` loops. Below is its description.

range(): The range() function creates an iterative object that represents an immutable sequence of numbers and is commonly used for looping a specific number of times in for loops.

The advantage of the range type over a regular list or tuple is that a range object will always take the same (small) amount of memory, no matter the size of the range it represents (as it only stores the start, stop and step values, calculating individual items and subranges as needed).

Below is an example where the `range()` function is used to print over integers from 0 to 4.

```
for i in range(5):
    print(i)
```

```
0
1
2
3
4
```

Note that the range function itself doesn't store the list of integers from 0 to 4; it is more memory-efficient by generating values on the fly.

Note that the last element is one less than the integer specified in the `range()` function.

Using the `range()` function, the `for` loop can iterate over a sequence of numbers. See the example below.

Example: Print the first *n* elements of the Fibonacci sequence, where *n* is an integer input by the user, such that *n>2*. In a fibonacci sequence, each number is the sum of the preceding two numbers, and the sequence starts from 0,1. The sequence is as follows:

0,1,1,2,3,5,8,13,….

```
n = int(input("Enter number of elements:"))

#Initializing the sequence to start from 0, 1
n1 = 0;n2 = 1

#Printing the first two numbers of the sequence
print(n1)
print(n2)

for i in range(n-2):  #Since two numbers of the sequence are al

    #Computing the next number of the sequence as the summation
    n3 = n1 + n2
    print(n3)

    #As 'n3' is already printed, it is no longer the next numbe
    #Thus, we move the values of the variables n1 and n2 one pl
    n1 = n2
    n2 = n3

print("These are the first", n, "elements of the fibonacci serie
```

```
Enter number of elements:6
0
1
1
2
3
5
These are the first 6 elements of the fibonacci series
```

As in the `if-elif-else` statement, the `for` loop uses identation to indicate the piece of code to be run repeatedly.

Note that we have used an in-built python function

3.2.2 `while` loop

With a `while` loops, a piece of code is executed repeatedly until certain condition(s) hold.

Example: Print all the elements of the Fibonacci sequence less than *n*, where *n* is an integer input by the user, such that *n>2*. In a fibonacci sequence, each number is the sum of the preceding two numbers, and the sequence starts from 0,1. The sequence is as follows:

0,1,1,2,3,5,8,13,….

```
n = int(input("Enter the value of n:"))

#Initializing the sequence to start from 0, 1
n1 = 0; n2 = 1

#Printing the first number of the sequence
print(n1)

while n2 < n:

    #Print the next number of the sequence
    print(n2)

    #Comptuing the next number of the sequence as the summation
    n3 = n1 + n2

    #As n2 is already printed, assigning n2 to n3, so that the n
    #Assigning n1 to n2 as n1 has already been used to compute t
    n1 = n2
    n2 = n3
print("These are all the elements of the fibonacci series less t
```

```
Enter the value of n:50
0
1
1
2
3
5
8
13
21
34
These are all the elements of the fibonacci series less than 50
```

3.2.3 Practice exercise 3

3.2.3.1

Write a program that identifies whether a number input by the user is prime or not.

Solution:

```
number = int(input("Enter a positive integer:"))

#Defining a variable that will have a value of 0 if there are no
num_divisors = 0

#Checking if the number has any divisors from 2 to half of the n
for divisor in range(2,int(number/2+1)):
        if number % divisor == 0:

            #If the number has a divisor, setting num_divisors t
            num_divisors = 1

            #If a divisor has been found, there is no need to ch
            #Even if the number has a single divisor, it is not
            #If you don't 'break', your code will still be corre
            break

#If there are no divisors of the number, it is prime, else not p
if num_divisors == 0:
    print("Prime")
else:
    print("Not prime")
```

```
Enter a positive integer:97
Prime
```

3.2.3.2

Update the program above to print the prime numbers starting from 2, and less than *n* where *n* is a positive integer input by the user.

Solution:

```
n = int(input("Enter a positive integer:"))

#Defining a variable - number_iterator. We will use this variabl
#While iterating over each integer from 2 to n, we will check if
```

```
number_iterator = 2

print(number_iterator) #Since '2' is a prime number, we can pri

#Continue to check for prime numbers until n (but not including
while(number_iterator < n):

    #After each check, increment the number_iterator to check i
    number_iterator = number_iterator + 1

    #Defining a variable that will have a value of 0 if there a
    num_divisors = 0

    #Checking if the integer has any divisors from 2 to half of
    for divisor in range(2,int(number_iterator/2 + 1)):
        if number_iterator % divisor == 0:

            #If the integer has a divisor, setting num_divisors
            num_divisors = 1

            #If a divisor has been found, there is no need to ch
            #Even if the integer has a single divisor, it is not
            #Thus, we 'break' out of the loop that checks for di
            break

    #If there are no divisors of the integer being checked, the
    if num_divisors == 0:
        print(number_iterator)
```

```
Enter a positive integer:100
2
3
5
7
11
13
17
19
23
29
31
37
41
43
47
53
```

```
59
61
67
71
73
79
83
89
97
```

3.3 `break` statement

The break statement is used to unconditionally exit the innermost loop.

For example, suppose we need to keep asking the user to input year of birth and compute the corresponding age, until the user enters 1900 as the year of birth.

```
#The loop will continue to run indefinitely as the condition 'T
while True:
    year = int(input("Enter year of birth:"))
    if year == 1900:
        break        #If the user inputs 1900, then break out o
    else:
        print("Age = ", 2022 - year)    #Otherwise compute and
```

```
Enter year of birth:1987
Age =  35
Enter year of birth:1995
Age =  27
Enter year of birth:2001
Age =  21
Enter year of birth:1900
```

3.3.1 Practice exercise 4

Write a program that finds and prints the largest factor of a number input by the user. Check the output if the user inputs 133.

Solution:

```
num = int(input("Enter an integer:"))

#Looping from the half the integer to 0 as the highest factor is
for i in range(int(num/2) + 1, 0, -1):
    if num%i == 0:
```

```
        print("Largest factor = ", i)

        #Exiting the loop if the largest integer is found
        break
```

```
Enter an integer:133
Largest factor =  19
```

3.4 `continue` statement

The continue statement is used to continue with the next iteration of the loop without executing the lines of code below it.

For example, consider the following code:

```
for i in range(10):
    if i%2 == 0:
        continue
    print(i)
```

```
1
3
5
7
9
```

When the control flow reads the statement `continue`, it goes back to the beginning of the `for` loop, and ignores the lines of code below the statement.

3.4.1 Practice exercise 5:

Write a program that asks the user the question, “How many stars are in the Milky Way (in billions)?”. If the user answers 100, the program should print correct, and stop. However, if the user answers incorrectly, the program should print “incorrect”, and ask them if they want to try again. The program should continue to run until the user answers correctly, or they want to stop trying.

```
#Defining an infinite while loop as the loop may need to run inc
while True:
    answer = input("How many stars are there in the Milky Way? '
    if answer == '100':
        print("Correct")

        #Exiting the loop if the user answers correctly
```

```
        break
    else:
        print("Incorrect")
        try_again = input("Do you want to try again? (Y/N) ")
        if try_again == 'Y':

            #Continuing with the infinite loop if the user wants
            continue
        else:

            #Exiting the infinite loop if the user wants to stop
            break
```

```
How many stars are there in the Milky Way? 101
Incorrect
Do you want to try again? (Y/N) Y
How many stars are there in the Milky Way? 7
Incorrect
Do you want to try again? (Y/N) Y
How many stars are there in the Milky Way? 5
Incorrect
Do you want to try again? (Y/N) Y
How many stars are there in the Milky Way? 100
Correct
```

3.5 Loops with strings

Loops can be used to iterate over a string, just like we used them to iterate over a sequence of integers.

Consider the following string:

```
sentence = "She sells sea shells on the sea shore"
```

The i^{th} character of the string can be retrieved by its index. For example, the first character of the string `sentence` is:

```
sentence[0]
```

```
'S'
```

Slicing a string:

A part of the string can be sliced by passing the starting index (say `start`) and the

stopping index (say `stop`) as `start:stop` to the index operator []. This is called slicing a string. For a string `S`, the characters starting from the index `start` upto the index `stop`, but not including `stop`, can be sliced as `S[start:stop]`.

For example, the slice of the string `sentence` from index `4` to index `9`, but not including `9` is:

```
sentence[4:9]
```

```
'sells'
```

Example:

Input a string, and count and print the number of "*t*"s.

```
string = input("Enter a sentence:")

#Initializing a variable 'count_t' which will store the number o
count_t = 0

#Iterating over the entire length of the string.
#The length of the string is given by the len() function
for i in range(len(string)):

    #If the ith character of the string is 't', then we count it
    if string[i] == 't':
        count_t = count_t + 1

print("Number of 't's in the string = ", count_t)
```

```
Enter a sentence:Getting a tatto is not a nice experience
Number of 't's in the string =  6
```

3.5.1 Practice exercise 6

Write a program that asks the user to input a string, and print the number of "*the*" s in the string.

```
string = input("Enter a sentence:")

#Defining a variable to store the count of the word 'the'
count_the = 0

#Looping through the entire length of the string except the last
#As we are checking three letters at a time starting from the in
```

```
for i in range(len(string) - 3):

    #Slicing 3 letters of the string and checking if they are '
    if string[i:(i+3)] == 'the':

        #Counting the words that are 'the'
        count_the = count_the + 1
print("Number of 'the's in the string = ", count_the)
```

```
Enter a sentence:She sells the sea shells on the sea shore in
the spring
Number of 'the's in the string =  3
```