B Sc Computer Science – VI Semester

Elective Papers - CS6PET01: Python and LateX

Module II - Control Flow and Data Structures

Logical operators, if, If-Else, While loop, For loop, List value, length, operation and deletion, Dictionary operation & methods, Tuples

Flow of Control

The order of execution of the statements in a program is known as flow of control. The flow of control can be implemented using control structures. Python supports two types of control structures—selection and repetition.

```
Program to print the difference of two numbers.
#Program to print the difference of two input numbers
num1 = int(input("Enter first number: "))
num2 = int(input("Enter second number: "))
diff = num1 - num2
print("The difference of",num1,"and",num2,"is",diff)
Output:
Enter first number 5
Enter second number 7
The difference of 5 and 7 is -2
```

1. Selection

A selection is a decision involves selecting from one of the two or more possible options. In programming, this concept of decision making or selection is implemented with the help of if ..else statement.

1. Simple if statement

The syntax of if statement is:

```
if condition:
        statement(s)
```

In the following example, if the age entered by the user is greater than 18, then print that the user is eligible to vote. If the condition is true, then the indented statement(s) are executed.

The indentation implies that its execution is dependent on the condition. There is no limit on the number of statements that can appear as a block under the if statement.

Example 6.1

```
age = int(input("Enter your age "))
if age >= 18:
    print("Eligible to vote")
```

2. if .. else Statement

A variant of if statement called if..else statement allows us to write two alternative paths and the control condition determines which path gets executed. The syntax for if..else statement is as follows.

```
if condition:
        statement(s)
else:
   statement(s)
```

Let us now modify the example on voting with the condition that if the age entered by the user is greater than 18, then to display that the user is eligible to vote. Otherwise display that the user is not eligible to vote.

```
age = int(input("Enter your age: "))
if age >= 18:
       print("Eligible to vote")
else:
  print("Not eligible to vote")
```

3. if .. elif .. else Statement

Many a times there are situations that require multiple conditions to be checked and it may lead to many alternatives. In such cases we can chain the conditions using if..elif (elif means else..if).

The syntax for a selection structure using elif is as shown below.

```
if condition:
    statement(s)
elif condition:
    statement(s)
elif condition:
    statement(s)
else:
statement(s)
```

Number of elif is dependent on the number of conditions to be checked. If the first condition is false, then the next condition is checked, and so on. If one of the conditions is true, then the corresponding indented block executes, and the if statement terminates.

Example Check whether a number is positive, negative, or zero.

```
number = int(input("Enter a number: ")
if number > 0:
  print("Number is positive")
elif number < 0:
  print("Number is negative")
else:
print("Number is zero")
```

Example Display the appropriate message as per the colour of signal at the road crossing.

```
signal = input("Enter the colour: ")
if signal == "red" or signal == "RED":
    print("STOP")
elif signal == "orange" or signal == "ORANGE":
    print("Be Slow")
elif signal == "green" or signal == "GREEN":
   print("Go!")
```

Program 6-3 Write a program to create a simple calculator performing only four basic operations.

```
#Program to create a four function calculator
result = 0
val1 = float(input("Enter value 1: "))
val2 = float(input("Enter value 2: "))
op = input("Enter any one of the operator (+,-,*,/): ")
if op == "+":
    result = val1 + val2
elif op == "-":
    if val1 > val2:
        result = val1 - val2
    else:
        result = val2 - val1
elif op == "*":
    result = val1 * val2
elif op == "/":
    if val2 == 0:
        print("Error! Division by zero is not allowed.
            Program terminated")
    else:
        result = val1/val2
else:
    print("Wrong input,program terminated")
```

```
        print("The result is ",result)
```

Output:

```
Enter value 1: 84
Enter value 2: 4
Enter any one of the operator (+,-,*,/): /
The result is 21.0
```

4. Nested if Statements

Nested if .. statements is one if… statement inside another if… statement.

In the program, for the operators "-" and "/", there exists an if..else condition within the elif block. This is called nested if. We can have many levels of nesting inside if..else statements.

2. INDENTATION

In most programming languages, the statements within a block are put inside curly brackets. However, Python uses indentation for block as well as for nested block structures. Leading whitespace (spaces and tabs) at the beginning of a statement is called indentation. In Python, the same level of indentation associate statements into a single block of code. The interpreter checks indentation levels very strictly and throws up syntax errors if indentation is not correct. It is a common practice to use a single tab for each level of indentation.

Program : Program to find the larger of the two pre-specified numbers.

```
#Program to find larger of the two numbers
num1 = 5
num2 = 6
if num1 > num2: #Block1
   print("first number is larger")
   print("Bye")
else: #Block2
   print("second number is larger")
   print("Bye Bye")
Output:
second number is larger
Bye Bye
```

3. Repetition

Often, we repeat a tasks, for example, payment of electricity bill, which is done every month. Consider the life cycle of butterfly that involves four stages, i.e., a butterfly lays eggs, turns into a caterpillar, becomes a pupa, and finally matures as a butterfly. The cycle starts again with laying of eggs by the butterfly.
This kind of repetition is also called iteration. **Repetition of a set of statements in a program is made possible using looping constructs.**

1. The 'for' Loop

The for statement is used to iterate over a range of values or a sequence. The for loop is executed for each of the items in the range. These values can be either numeric, or, as we shall see in later chapters, they can be elements of a data type like a string, list, or tuple. With every iteration of the loop, the control variable checks whether each of the values in the range have been traversed or not. When all the items in the range are exhausted, the statements within loop are not executed; the control is then transferred to the statement immediately following the for loop. While using for loop, it is known in advance.

(A) Syntax of the For Loop

```
for <control-variable> in <sequence/items in range>:
        <statements inside body of the loop>
else: #optional
        <statements inside else>
```

Program to print the characters in the string 'PYTHON' using for loop.

```
#Print the characters in word PYTHON using for loop
for letter in 'PYTHON':
   print(letter)
Output:
PYTHON
```

Program to print the numbers in a given sequence using for loop.

```
#Print the given sequence of numbers using for loop
count = [10,20,30,40,50]
for num in count:
   print(num)
```

Output:

```
10
20
30
40
50
```

Program to print even numbers in a given sequence using for loop.

```
#Print even numbers in the given sequence
numbers = [1,2,3,4,5,6,7,8,9,10]
for num in numbers:
     if (num % 2) == 0:
               print(num,'is an even Number')
```

Output:

```
2 is an even Number
4 is an even Number
6 is an even Number
8 is an even Number
10 is an even Number
```

(B) The Range() Function

The range() is a built-in function in Python. Syntax of range() function is:

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

It is used to create a list containing a sequence of integers from the given start value up to stop value (excluding stop value), with a difference of the given step value. We will learn about functions in the next chapter. To begin with, simply remember that function takes parameters to work on. In function range(), start, stop and step are parameters.

The start and step parameters are optional. If start value is not specified, by default the list starts from 0. If step is also not specified, by default the value increases by 1 in each iteration. All parameters of range() function must be integers. The step parameter can be a positive or a negative integer excluding zero.

Example

```
#start and step not specified
>>> list(range(10))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
#default step value is 1
>>> list(range(2, 10))
[2, 3, 4, 5, 6, 7, 8, 9]
#step value is 5
>>> list(range(0, 30, 5))
[0, 5, 10, 15, 20, 25]
#step value is -1. Hence, decreasing sequence is generated
>>> list (range (0, -9, -1))
[0, -1, -2, -3, -4, -5, -6, -7, -8]
```

The function range() is often used in for loops for generating a sequence of numbers.

Program to print the multiples of 10 for numbers in a given range.

```
#Print multiples of 10 for numbers in a given range
for num in range(5):
        if num > 0:
                print(num * 10)
```

Output:

```
10
20
30
40
```

2. while loop

The while statement executes a block of code repeatedly as long as the control condition of the loop is true. The control condition of the while loop is executed before any statement inside the loop is executed. After each iteration, the control condition is tested again and the loop continues as long as the condition remains true. When this condition becomes false, the statements in the body of loop are not executed and the control is transferred to the statement immediately following the body of while loop. If the condition of the while loop is initially false, the body is not executed even once.

The statements within the body of the while loop must ensure that the condition eventually becomes false; otherwise the loop will become an infinite loop, leading to a logical error in the program.

Syntax of while Loop

```
while test_condition:
        <body of while>
else:
        <body of else>
```

The else block just after for/while is executed only when the loop is NOT terminated by a break statement

Program to print first 5 natural numbers using while loop.

```
# Print first 5 natural numbers using while loop
count = 1
while count <= 5:
 print(count) count += 1
Output:
1
2
3
4
5
```

Python program to show how to use a while loop

```
counter = 0

# Initiating the loop

while counter < 10: # giving the condition

   counter = counter + 3

   print("Python Loops")


#Python program to show how to use else statement with the while loop
counter = 0

# Iterating through the while loop
while (counter < 10):
   counter = counter + 3
   print("Python Loops") # Executed until condition is met
# Once the condition of while loop gives False this statement will be executed
else:
   print("Code block inside the else statement")
```

Jumps in Loops

Looping constructs allow programmers to repeat tasks efficiently. In certain situations, when some particular condition occurs, we may want to exit from a loop or skip some statements of the loop before continuing further in the loop. These requirements can be achieved by using break and continue statements, respectively. Python provides these statements as a tool to give more flexibility to the programmer to control the flow of execution of a program. Jumps statements in Python are:

1. **break**
2. **continue**
3. **pass**

1. Break Statements

The break statement alters the normal flow of execution as it terminates the current loop and resumes execution of the statement following that loop.

```
Program to demonstrate use of break statement.
num = 0
for num in range(10):
        num = num + 1
        if num == 8:
                break
                print('Num has value ' + str(num))
print('Encountered break!! Out of loop')

Output:
Num has value 1
Num has value 2
Num has value 3
Num has value 4
Num has value 5
Num has value 6
Num has value 7
Encountered break!! Out of loop
```

Note: When value of num becomes 8, the break statement is executed and the for loop terminates.

2. Continue Statement

When a continue statement is encountered, the control skips the execution of remaining statements inside the body of the loop for the current iteration and jumps to the beginning of the loop for the next iteration. If the loop's condition is still true, the loop is entered again, else the control is transferred to the statement immediately following the loop.

```
#Prints values from 0 to 6 except 3
num = 0
for num in range(6):
    num = num + 1
    if num == 3:
        continue
    print('Num has value ' + str(num))
print('End of loop')

Output:
Num has value 1
Num has value 2
Num has value 4
Num has value 5
Num has value 6
End of loop
```

Note: Observe that the value 3 is not printed in the output, but the loop continues after the continue statement to print other values till the for loop terminates.

3. Pass Statement

In Python programming, pass is a null statement. The difference between a comment and pass statement is that, while the interpreter ignores a comment entirely, pass is not ignored. However, nothing happens when pass is executed. It results into no operation (NOP).

```
# pass is just a placeholder for functionality to be added later

sequence = {'p','a','s','s'}
for val in sequence:
        pass
```

STRINGS

String is a sequence which is made up of one or more UNICODE characters. Here the character can be a letter, digit, whitespace or any other symbol. A string can be created by enclosing one or more characters in single, double or triple quote.

Example

```
>>> str1 = 'Hello World!'
>>> str2 = "Hello World!"
>>> str3 = """Hello World!"""
>>> str4 = '''Hello World!'''
```

str1, str2, str3, str4 are all string variables having the same value 'Hello World!'. Values stored in str3 and str4 can be extended to multiple lines using triple codes as can be seen in the following example:

```
>>> str3 = """Hello World!
welcome to the world of Python"""
>>> str4 = '''Hello World!
welcome to the world of Python'''
```

Accessing Characters in a String

Each individual character in a string can be accessed using a technique called indexing. The index specifies the character to be accessed in the string and is written in square brackets ([ ]). The index of the first character (from left) in the string is 0 and the last character is n-1 where n is the length of the string. If we give index value out of this range then we get an *IndexError*. The index must be an integer (positive, zero or negative).

```
#initializes a string str1
>>> str1 = 'Hello World!'
#gives the first character of str1
>>> str1[0]
'H'
#gives seventh character of str1
>>> str1[6]
'W'
#gives last character of str1
>>> str1[11]
'!'
#gives error as index is out of range
>>> str1[15]
```

```
IndexError: string index out of range
```

The index can also be an expression including variables and operators but the expression must evaluate to an integer.

```
#an expression resulting in an integer index
#so gives 6th character of str1
>>> str1[2+4]
'W'

#gives error as index must be an integer
>>> str1[1.5]
TypeError: string indices must be integers
```

Python allows an index value to be negative also. Negative indices are used when we want to access the characters of the string from right to left. Starting from right hand side, the first character has the index as -1 and the last character has the index –n where n is the length of the string. Table 8.1 shows the indexing of characters in the string 'Hello World!' in both the cases, i.e., positive and negative indices.

```
>>> str1[-1] #gives first character from right
'!'
>>> str1[-12]#gives last character from right
'H'
```

Positive Indices	0	1	2	3	4	5	6	7	8	9	10	11
String	H	e	l	l	o		W	o	r	l	d	!
Negative Indices	-12	-11	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1

An inbuilt function len() in Python returns the length of the string that is passed as parameter. For example, the length of string str1 = 'Hello World!' is 12.

```
#gives the length of the string str1
>>> len(str1)
12
#length of the string is assigned to n
>>> n = len(str1)
>>> print(n)
12
#gives the last character of the string
>>> str1[n-1]
'!'
#gives the first character of the string
>>> str1[-n]
'H'
```

String is Immutable

A string is an immutable data type. It means that the contents of the string cannot be changed after it has been created. An attempt to do this would lead to an error.

```
>>> str1 = "Hello World!"
#if we try to replace character 'e' with 'a'
>>> str1[1] = 'a'
TypeError: 'str' object does not support item assignment
```

STRING OPERATIONS

As we know that string is a sequence of characters. Python allows certain operations on string data type, such as concatenation, repetition, membership and slicing.

1. Concatenation

To concatenate means to join. Python allows us to join two strings using concatenation operator plus which is denoted by symbol +.

```
>>> str1 = 'Hello' #First string
>>> str2 = 'World!' #Second string
>>> str1 + str2 #Concatenated strings
'HelloWorld!'
#str1 and str2 remain same
>>> str1 #after this operation.
'Hello'
>>> str2
'World!
```

2. Repetition

Python allows us to repeat the given string using repetition operator which is denoted by symbol *

```
#assign string 'Hello' to str1
>>> str1 = 'Hello'
#repeat the value of str1 2 times
>>> str1 * 2
'HelloHello'
#repeat the value of str1 5 times
>>> str1 * 5
'HelloHelloHelloHelloHello'
```

Note: str1 still remains the same after the use of repetition operator.

3. Membership

Python has two membership operators 'in' and 'not in'. The 'in' operator takes two strings and returns True if the first string appears as a substring in the second string, otherwise it returns False.

```
>>> str1 = 'Hello World!'
>>> 'W' in str1
True
>>> 'Wor' in str1
```

```
True
>>> 'My' in str1
False
```

The 'not in' operator also takes two strings and returns True if the first string does not appear as a substring in the second string, otherwise returns False.

```
>>> str1 = 'Hello World!'
>>> 'My' not in str1
True
>>> 'Hello' not in str1
False
```

4. Slicing

In Python, to access some part of a string or substring, we use a method called slicing. This can be done by specifying an index range. Given a string str1, the slice operation str1[n:m] returns the part of the string str1 starting from index n (inclusive) and ending at m (exclusive). In other words, we can say that str1[n:m] returns all the characters starting from str1[n] till str1[m-1]. The numbers of characters in the substring will always be equal to difference of two indices m and n, i.e., (m-n).

```
>>> str1 = 'Hello World!'
#gives substring starting from index 1 to 4
>>> str1[1:5]
'ello'
#gives substring starting from 7 to 9
>>> str1[7:10]
'orl'
#index that is too big is truncated down to
#the end of the string
>>> str1[3:20]
'lo World!'
#first index > second index results in an
#empty '' string
>>> str1[7:2]
If the first index is not mentioned, the slice starts
from index.
#gives substring from index 0 to 4
>>> str1[:5]
'Hello'
```

If the second index is not mentioned, the slicing is done till the length of the string.

```
#gives substring from index 6 to end
>>> str1[6:]
'World!'
```

The slice operation can also take a third index that specifies the 'step size'. For example, str1[n:m:k], means every k_{th} character has to be extracted from the string str1 starting from n and ending at m-1. By default, the step size is one.

```
>>> str1[0:10:2]
'HloWr'
>>> str1[0:10:3]
'HlWl'
```

Negative indexes can also be used for slicing.

```
#characters at index -6,-5,-4,-3 and -2 are
#sliced
>>> str1[-6:-1]
'World'
```

If we ignore both the indexes and give step size as -1

```
#str1 string is obtained in the reverse order
>>> str1[::-1]
'!dlroW olleH'
```

5. TRAVERSING A STRING

We can access each character of a string or traverse a string using for loop and while loop.

(A) String Traversal Using `for` Loop:

```
>>> str1 = 'Hello World!'
>>> for ch in str1:
print(ch,end = '')
Hello World! #output of for loop
```

In the above code, the loop starts from the first character of the string str1 and automatically ends when the last character is accessed.

(B) String Traversal Using `while` Loop:

```
>>> str1 = 'Hello World!'
>>> index = 0
#len(): a function to get length of string
>>> while index < len(str1):
print(str1[index],end = '')
index += 1
Hello World! #output of while loop
```

Here while loop runs till the condition index < len(str) is True, where index varies from 0 to len(str1) -1.

Method	Description	Example

len()	Returns the length of the given string	`>>> str1 = 'Hello World!'`
		`>>> len(str1)`
		`12`

title()	Returns the string with first letter of every word in the string in uppercase and rest in lowercase	`>>> str1 = 'hello WORLD!'`
		`>>> str1.title()`
		`'Hello World!'`

lower()	Returns the string with all uppercase letters converted to lowercase	`>>> str1 = 'hello WORLD!'`
		`>>> str1.lower()`
		`'hello world!'`

upper()	Returns the string with all lowercase letters converted to uppercase	`>>> str1 = 'hello WORLD!'`
		`>>> str1.upper()`
		`'HELLO WORLD!'`

count(str, start, end)	Returns number of times substring `str` occurs in the given string. If we do not give start index and end index then searching starts from index 0 and ends at length of the string	`>>> str1 = 'Hello World! Hello Hello'`
		`>>> str1.count('Hello',12,25)`
		`2`
		`>>> str1.count('Hello')`
		`3`

find(str,start, end)	Returns the first occurrence of index of substring `str` occurring in the given string. If we do not give start and end then searching starts from index 0 and ends at length of the string. If the substring is not present in the given string, then the function returns -1	`>>> str1= 'Hello World! Hello Hello'`
		`>>> str1.find('Hello',10,20)`
		`13`
		`>>> str1.find('Hello',15,25)`
		`19`
		`>>> str1.find('Hello')`
		`0`
		`>>> str1.find('Hee')`
		`-1`

index(str, start, end)	Same as find() but raises an exception if the substring is not present in the given string	>>> str1 = 'Hello World! Hello Hello'
		>>> str1.index('Hello')
		0
		>>> str1.index('Hee')
		ValueError: substring not found

endswith()	Returns True if the given string ends with the supplied substring otherwise returns False	>>> str1 = 'Hello World!'
		>>> str1.endswith('World!')
		True
		>>> str1.endswith('!')
		True
		>>> str1.endswith('lde')
		False

startswith()	Returns True if the given string starts with the supplied substring otherwise returns False	>>> str1 = 'Hello World!'
		>>> str1.startswith('He')
		True
		>>> str1.startswith('Hee')
		False

isalnum()	Returns True if characters of the given string are either alphabets or numeric. If whitespace or special symbols are part of the given string or the string is empty it returns False	>>> str1 = 'HelloWorld'
		>>> str1.isalnum()
		True
		>>> str1 = 'HelloWorld2'
		>>> str1.isalnum()
		True
		>>> str1 = 'HelloWorld!!'
		>>> str1.isalnum()
		False

islower()	Returns True if the string is non-empty and has all lowercase alphabets, or has at least one character as lowercase alphabet and rest are non-alphabet characters	>>> str1 = 'hello world!'
		>>> str1.islower()
		True
		>>> str1 = 'hello 1234'
		>>> str1.islower()
		True
		>>> str1 = 'hello ??'
		>>> str1.islower()
		True
		>>> str1 = '1234'
		>>> str1.islower()
		False
		>>> str1 = 'Hello World!'
		>>> str1.islower()
		False

isupper()	Returns True if the string is non-empty and has all uppercase alphabets, or has at least one character as uppercase character and rest are non-alphabet characters	`>>> str1 = 'HELLO WORLD!'`
		`>>> str1.isupper()`
		`True`
		`>>> str1 = 'HELLO 1234'`
		`>>> str1.isupper()`
		`True`
		`>>> str1 = 'HELLO ??'`
		`>>> str1.isupper()`
		`True`
		`>>> str1 = '1234'`
		`>>> str1.isupper()`
		`False`
		`>>> str1 = 'Hello World!'`
		`>>> str1.isupper()`
		`False`

isspace()	Returns True if the string is non-empty and all characters are white spaces (blank, tab, newline, carriage return)	`>>> str1 = '  \n  \t \r'`
		`>>> str1.isspace()`
		`True`
		`>>> str1 = 'Hello  \n'`
		`>>> str1.isspace()`
		`False`

istitle()	Returns True if the string is non-empty and title case, i.e., the first letter of every word in the string in uppercase and rest in lowercase	`>>> str1 = 'Hello World!'`
		`>>> str1.istitle()`
		`True`
		`>>> str1 = 'hello World!'`
		`>>> str1.istitle()`
		`False`

lstrip()	Returns the string after removing the spaces only on the left of the string	`>>> str1 = '  Hello World! '`
		`>>> str1.lstrip()`
		`'Hello World!  '`

rstrip()	Returns the string after removing the spaces only on the right of the string	`>>> str1 = '  Hello World!'`
		`>>> str1.rstrip()`
		`'  Hello World!'`

strip()	Returns the string after removing the spaces both on the left and the right of the string	`>>> str1 = '  Hello World!'`
		`>>> str1.strip()`
		`'Hello World!'`

replace(oldstr, newstr)	Replaces all occurrences of old string with the new string	`>>> str1 = 'Hello World!'`
		`>>> str1.replace('o','*')`
		`'Hell* W*rld!'`
		`>>> str1 = 'Hello World!'`
		`>>> str1.replace('World','Country')`
		`'Hello Country!'`
		`>>> str1 = 'Hello World! Hello'`
		`>>> str1.replace('Hello','Bye')`
		`'Bye World! Bye'`

join()	Returns a string in which the characters in the string have been joined by a separator	`>>> str1 = ('HelloWorld!')`
		`>>> str2 = '-'  #separator`
		`>>> str2.join(str1)`
		`'H-e-l-l-o-W-o-r-l-d-!'`

partition())	Partitions the given string at the first occurrence of the substring (separator) and returns the string partitioned into three parts.
	1. Substring before the separator
	2. Separator
	3. Substring after the separator
	If the separator is not found in the string, it returns the whole string itself and two empty strings
		`>>> str1 = 'India is a Great Country'`
		`>>> str1.partition('is')`
		`('India ', 'is', ' a Great Country')`
		`>>> str1.partition('are')`
		`('India is a Great Country',' ',' ')`

split()	Returns a list of words delimited by the specified substring. If no delimiter is given then words are separated by space.	`>>> str1 = 'India is a Great Country'`
		`>>> str1.split()`
		`['India','is','a','Great', 'Country']`
		`>>> str1 = 'India is a Great Country'`
		`>>> str1.split('a')`
		`['Indi', ' is ', ' Gre', 't Country']`

Lists

The data type list is an ordered sequence which is mutable and made up of one or more elements. Unlike a string which consists of only characters, a list can have elements of different data types, such as integer, float, string, tuple or even another list. A list is very useful to group together elements of mixed data types. Elements of a list are enclosed in square brackets and are separated by comma. Like string indices, list indices also start from 0.

Example

```
#list1 is the list of six even numbers
>>> list1 = [2,4,6,8,10,12]
>>> print(list1)
[2, 4, 6, 8, 10, 12]

#list2 is the list of vowels
>>> list2 = ['a','e','i','o','u']
>>> print(list2)
['a', 'e', 'i', 'o', 'u']

#list3 is the list of mixed data types
>>> list3 = [100,23.5,'Hello']
>>> print(list3)
[100, 23.5, 'Hello']
```

```
#list4 is the list of lists called nested
#list
>>> list4 =[['Physics',101],['Chemistry',202],
['Maths',303]]
>>> print(list4)
[['Physics', 101], ['Chemistry', 202],
['Maths', 303]]
```

Accessing Elements in a List

The elements of a list are accessed in the same way as characters are accessed in a string.

```
#initializes a list list1
>>> list1 = [2,4,6,8,10,12]
>>> list1[0] #return first element of list1
2
>>> list1[3] #return fourth element of list1
8
#return error as index is out of range
>>> list1[15]
IndexError: list index out of range
#an expression resulting in an integer index
>>> list1[1+4]
12
>>> list1[-1] #return first element from right
12
#length of the list list1 is assigned to n
>>> n = len(list1)
>>> print(n)
6
#return the last element of the list1
>>> list1[n-1]
12
#return the first element of list1
>>> list1[-n]
2
```

Lists are Mutable

In Python, lists are mutable. It means that the contents of the list can be changed after it has been created.

```
#List list1 of colors
>>> list1 = ['Red','Green','Blue','Orange']
#change/override the fourth element of list1
>>> list1[3] = 'Black'
>>> list1 #print the modified list list1
['Red', 'Green', 'Blue', 'Black']
```

LIST OPERATIONS

The data type list allows manipulation of its contents through various operations as shown below.

1. Concatenation

Python allows us to join two or more lists using concatenation operator depicted by the symbol +.

```
#list1 is list of first five odd integers
>>> list1 = [1,3,5,7,9]
#list2 is list of first five even integers
>>> list2 = [2,4,6,8,10]
#elements of list1 followed by list2

>>> list1 + list2
[1, 3, 5, 7, 9, 2, 4, 6, 8, 10]
>>> list3 = ['Red','Green','Blue']
>>> list4 = ['Cyan', 'Magenta', 'Yellow'
,'Black']
>>> list3 + list4
['Red','Green','Blue','Cyan','Magenta','Yellow','Black']
```

Note that, there is no change in ongoing lists, i.e., `list1, list2, list3, list4` remain the same after concatenation operation. If we want to merge two lists, then we should use an assignment statement to assign the merged list to another list. The concatenation operator '+' requires that the operands should be of list type only. If we try to concatenate a list with elements of some other data type, `TypeError` occurs.

```
>>> list1 = [1,2,3]
>>> str1 = "abc"
>>> list1 + str1
TypeError: can only concatenate list (not "str") to list
```

2. Repetition

Python allows us to replicate a list using repetition operator depicted by symbol *.

```
>>> list1 = ['Hello']
#elements of list1 repeated 4 times
>>> list1 * 4
['Hello', 'Hello', 'Hello', 'Hello']
```

3. Membership

Like strings, the membership operators `in` checks if the element is present in the list and returns `True`, else returns `False`.

```
>>> list1 = ['Red','Green','Blue']
>>> 'Green' in list1
```

```
True
>>> 'Cyan' in list1
False
```

The `not in` operator returns `True` if the element is not present in the list, else it returns `False`.

```
>>> list1 = ['Red','Green','Blue']
>>> 'Cyan' not in list1
True
>>> 'Green' not in list1
     False
```

4. Slicing

Like strings, the slicing operation can also be applied to lists.

```
>>> list1 =['Red','Green','Blue','Cyan','Magenta','Yellow','Black']
>>> list1[2:6]
['Blue', 'Cyan', 'Magenta', 'Yellow']
#list1 is truncated to the end of the list

>>> list1[2:20] #second index is out of range
['Blue', 'Cyan', 'Magenta', 'Yellow','Black']
>>> list1[7:2] #first index > second index

[] #results in an empty list
#return sublist from index 0 to 4

>>> list1[:5] #first index missing
['Red','Green','Blue','Cyan','Magenta']
#slicing with a given step size

>>> list1[0:6:2]
['Red','Blue','Magenta']

#negative indexes
#elements at index -6,-5,-4,-3 are sliced
>>> list1[-6:-2]
['Green','Blue','Cyan','Magenta']

#both first and last index missing
>>> list1[::2] #step size 2 on entire list
['Red','Blue','Magenta','Black']

#negative step size
#whole list in the reverse order
>>> list1[::-1]
['Black','Yellow','Magenta','Cyan','Blue','Green','Red']
```

TRAVERSING A LIST

We can access each element of the list or traverse a list using a `for` loop or a `while` loop.

(A) List Traversal Using `for` Loop:

```
>>> list1 = ['Red','Green','Blue','Yellow','Black']
>>> for item in list1:
      print(item)
```

Output:

```
Red
Green
Blue
Yellow
Black
```

Another way of accessing the elements of the list is using range() and len() functions:

```
>>> for i in range(len(list1)):
print(list1[i])
```

Output:

```
Red
Green
Blue
Yellow
Black
```

(B) List Traversal Using `while` Loop:

```
>>> list1 = ['Red','Green','Blue','Yellow',
'Black']
>>> i = 0
>>> while i < len(list1):
          print(list1[i])
          i += 1
```

Output:

```
Red
Green
Blue
Yellow
Black
```

LIST METHODS AND BUILT-IN FUNCTIONS

The data type list has several built-in methods that are useful in programming. Some of them are listed in Table.

Method	Description	Example

len()	Returns the length of the list passed as the argument	`>>> list1 = [10,20,30,40,50]`
		`>>> len(list1)`
		`5`

list()	Creates an empty list if no argument is passed	`>>> list1 = list()` `>>> list1`

	Creates a list if a sequence is passed as an argument	`[ ]`
		`>>> str1 = 'aeiou'`
		`>>> list1 = list(str1)`
		`>>> list1`
		`['a', 'e', 'i', 'o', 'u']`

append()	Appends a single element passed as an argument at the end of the list
	
	The single element can also be a list
		`>>> list1 = [10,20,30,40]`
		`>>> list1.append(50)`
		`>>> list1`
		`[10, 20, 30, 40, 50]`
		`>>> list1 = [10,20,30,40]`
		`>>> list1.append([50,60])`
		`>>> list1`
		`[10, 20, 30, 40, [50, 60]]`

extend()	Appends each element of the list passed as argument to the end of the given list	`>>> list1 = [10,20,30]`
		`>>> list2 = [40,50]`
		`>>> list1.extend(list2)`
		`>>> list1`
		`[10, 20, 30, 40, 50]`

insert()	Inserts an element at a particular index in the list	`>>> list1 = [10,20,30,40,50]`
		`>>> list1.insert(2,25)`
		`>>> list1`
		`[10, 20, 25, 30, 40, 50]`
		`>>> list1.insert(0,5)`
		`>>> list1`
		`[5, 10, 20, 25, 30, 40, 50]`

count()	Returns the number of times a given element appears in the list	`>>> list1 = [10,20,30,10,40,10]`
		`>>> list1.count(10)`
		`3`
		`>>> list1.count(90)`
		`0`

index()	Returns index of the first occurrence of the element in the list. If the element is not present, ValueError is generated	`>>> list1 = [10,20,30,20,40,10]`
		`>>> list1.index(20)`
		`1`
		`>>> list1.index(90)`
		`ValueError: 90 is not in list`

remove()	Removes the given element from the list. If the element is present multiple times, only the first occurrence is removed. If the element is not present, then ValueError is generated	`>>> list1 = [10,20,30,40,50,30]`
		`>>> list1.remove(30)`
		`>>> list1`
		`[10, 20, 40, 50, 30]`
		
		`>>> list1.remove(90)`
		`ValueError:list.remove(x):x not in list`

pop()	Returns the element whose index is passed as parameter to this function and also removes it from the list. If no parameter is given, then it returns and removes the last element of the list	`>>> list1 = [10,20,30,40,50,60]`
		`>>> list1.pop(3)`
		`40`
		`>>> list1`
		`[10, 20, 30, 50, 60]`
		`>>> list1 = [10,20,30,40,50,60]`
		`>>> list1.pop()`
		`60`

		`>>> list1` `[10, 20, 30, 40, 50]`

reverse()	Reverses the order of elements in the given list	`>>> list1 = [34,66,12,89,28,99]`
		`>>> list1.reverse()`
		`>>> list1`
		`[ 99, 28, 89, 12, 66, 34]`
		
		`>>> list1 = [ 'Tiger' ,'Zebra' , 'Lion' , 'Cat' ,'Elephant' ,'Dog']`
		`>>> list1.reverse()`
		`>>> list1`
		`['Dog', 'Elephant', 'Cat', 'Lion', 'Zebra', 'Tiger']`

sort()	Sorts the elements of the given list in-place	`>>>list1=['Tiger','Zebra','Lion', 'Cat', 'Elephant' ,'Dog']`
		`>>> list1.sort()`
		`>>> list1`
		`['Cat', 'Dog', 'Elephant', 'Lion', 'Tiger', 'Zebra']`
		
		`>>> list1 = [34,66,12,89,28,99]`
		`>>> list1.sort(reverse = True)`
		`>>> list1`
		`[99,89,66,34,28,12]`

sorted()	It takes a list as parameter and creates a new list consisting of the same elements arranged in sorted order	`>>> list1 = [23,45,11,67,85,56]`
		`>>> list2 = sorted(list1)`
		`>>> list1`
		`[23, 45, 11, 67, 85, 56]`
		`>>> list2`
		`[11, 23, 45, 56, 67, 85]`

min()
 	
max()
 	
sum()
	Returns minimum or smallest element of the list
	
	Returns maximum or largest element of the list
	Returns sum of the elements of the list
		`>>> list1 = [34,12,63,39,92,44]`
		`>>> min(list1)`
		`12`
		`>>> max(list1)`
		`92`
		`>>> sum(list1)`
		`284`

NESTED LISTS

When a list appears as an element of another list, it is called a nested list.

Example

```
>>> list1 = [1,2,'a','c',[6,7,8],4,9]
#fifth element of list is also a list
>>> list1[4]
[6, 7, 8]
```

To access the element of the nested list of `list1`, we have to specify two indices `list1[i][j]`. The first index i will take us to the desired nested list and second index `j` will take us to the desired element in that nested list.

```
>>> list1[4][1]
7
#index i gives the fifth element of list1
#which is a list
#index j gives the second element in the
#nested list
```

COPYING LISTS

Given a list, the simplest way to make a copy of the list is to assign it to another list.

```
>>> list1 = [1,2,3]
>>> list2 = list1
>>> list1
[1, 2, 3]
>>> list2
[1, 2, 3]
```

The statement `list2 = list1` does not create a new list. Rather, it just makes `list1` and `list2` refer to the same list object. Here `list2` actually becomes an alias of `list1`. Therefore, any changes made to either of them will be reflected in the other list.

```
>>> list1.append(10)
>>> list1
[1, 2, 3, 10]
>>> list2
[1, 2, 3, 10]
```

We can also create a copy or clone of the list as a distinct object by three methods. The first method uses slicing, the second method uses built-in function list() and the third method uses copy() function of python library `copy`.

Method 1

We can slice our original list and store it into a new variable as follows:

```
newList = oldList[:]
```

Example

```
>>> list1 = [1,2,3,4,5]
>>> list2 = list1[:]
>>> list2
[1, 2, 3, 4, 5]
```

Method 2

We can use the built-in function list() as follows:

```
newList = list(oldList)
```

Example

```
>>> list1 = [10,20,30,40]
>>> list2 = list(list1)
>>> list2
[10, 20, 30, 40]
```

Method 3

We can use the copy () function as follows:

```
import copy #import the library copy
#use copy()function of library copy
newList = copy.copy(oldList)
```

Example

```
>>> import copy
>>> list1 = [1,2,3,4,5]
>>> list2 = copy.copy(list1)
>>> list2
[1, 2, 3, 4, 5]
```

LIST MANIPULATION

In the following programs, we will apply the various list manipulation methods.

Program : Write a menu driven program to perform various list operations, such as:
- Append an element
- Insert an element
- Append a list to the given list
- Modify an existing element
- Delete an existing element from its position
- Delete an existing element with a given value
- Sort the list in ascending order
- Sort the list in descending order
- Display the list.

```
#Program
#Menu driven program to do various list operations
myList = [22,4,16,38,13] #myList already has 5 elements
choice = 0
while True:
   print("The list 'myList' has the following elements", myList)
   print("\nL I S T O P E R A T I O N S")
   print(" 1. Append an element")
   print(" 2. Insert an element at the desired position")
   print(" 3. Append a list to the given list")
   print(" 4. Modify an existing element")
   print(" 5. Delete an existing element by its position")
   print(" 6. Delete an existing element by its value")
   print(" 7. Sort the list in ascending order")
   print(" 8. Sort the list in descending order")
   print(" 9. Display the list")
   print(" 10. Exit")
   choice = int(input("ENTER YOUR CHOICE (1-10): "))

   #append element
   if choice == 1:
      element = int(input("Enter the element to be appended: "))
      myList.append(element)
      print("The element has been appended\n")

    #insert an element at desired position
    elif choice == 2:
      element = int(input("Enter the element to be inserted: "))
      pos = int(input("Enter the position:"))
      myList.insert(pos,element)
      print("The element has been inserted\n")

    #append a list to the given list
    elif choice == 3:
     newList = eval(input( "Enter the elements separated by commas"))
     myList.extend(list(newList))
     print("The list has been appended\n")
```

```
#modify an existing element
elif choice == 4:
 i = int(input("Enter the position of the element to be modified: "))
 if i < len(myList):
   newElement = int(input("Enter the new element: "))
   oldElement = myList[i]
   myList[i] = newElement
   print("The element",oldElement,"has been modified\n")
 else:
  print("Position of the element is more than the length of list")

#delete an existing element by position
elif choice == 5:
  i = int(input("Enter the position of the element to be deleted: "))
  if i < len(myList):
    element = myList.pop(i)
    print("The element",element,"has been deleted\n")
  else:
     print("\nPosition of the element is more than the length of list")

#delete an existing element by value
elif choice == 6:
  element = int(input("\nEnter the element to be deleted: "))
  if element in myList:
     myList.remove(element)
     print("\nThe element",element,"has been deleted\n")
  else:
     print("\nElement",element,"is not present in the list")

#list in sorted order
elif choice == 7:
  myList.sort()
  print("\nThe list has been sorted")

#list in reverse sorted order
elif choice == 8:
 myList.sort(reverse = True)
 print("\nThe list has been sorted in reverse order")

#display the list
elif choice == 9:
 print("\nThe list is:", myList)

#exit from the menu
elif choice == 10:
   break
else:
   print("Choice is not valid")
   print("\n\nPress any key to continue.............")
   ch = input()
```

Tuples and Dictionaries

Tuples and Dictionaries

TUPLES

A tuple is an ordered sequence of elements of different data types, such as integer, float, string, list or even a tuple. Elements of a tuple are enclosed in parenthesis (round brackets) and are separated by commas. Like list and string, elements of a tuple can be accessed using index values, starting from 0.

Example

```
#tuple1 is the tuple of integers
>>> tuple1 = (1,2,3,4,5)
>>> tuple1
(1, 2, 3, 4, 5)
#tuple2 is the tuple of mixed data types
>>> tuple2 =('Economics',87,'Accountancy',89.6)
>>> tuple2
('Economics', 87, 'Accountancy', 89.6)
#tuple3 is the tuple with list as an element
>>> tuple3 = (10,20,30,[40,50])
>>> tuple3
(10, 20, 30, [40, 50])
#tuple4 is the tuple with tuple as an element
>>> tuple4 = (1,2,3,4,5,(10,20))
>>> tuple4
(1, 2, 3, 4, 5, (10, 20))
```

If there is only a single element in a tuple then the element should be followed by a comma. If we assign the value without comma it is treated as integer. It should be noted that a sequence without parenthesis is treated as tuple by default.

```
#incorrect way of assigning single element to
#tuple
#tuple5 is assigned a single element
>>> tuple5 = (20)
>>> tuple5
20

>>>type(tuple5) #tuple5 is not of type tuple
<class 'int'> #it is treated as integer
#Correct Way of assigning single element to
#tuple
#tuple5 is assigned a single element
>>> tuple5 = (20,) #element followed by comma
>>> tuple5
(20,)

>>>type(tuple5) #tuple5 is of type tuple
<class 'tuple'>
#a sequence without parentheses is treated as
#tuple by default
>>> seq = 1,2,3 #comma separated elements
```

```
>>> type(seq) #treated as tuple
<class 'tuple'>
>>> print(seq) #seq is a tuple
(1, 2, 3)
```

1. Accessing Elements in a Tuple

Elements of a tuple can be accessed in the same way as a list or string using indexing and slicing.

```
>>> tuple1 = (2,4,6,8,10,12)
#initializes a tuple tuple1
#returns the first element of tuple1
>>> tuple1[0]
2
#returns fourth element of tuple1
>>> tuple1[3]
8
#returns error as index is out of range
>>> tuple1[15]
IndexError: tuple index out of range
#an expression resulting in an integer index
>>> tuple1[1+4]
12
#returns first element from right
>>> tuple1[-1]
12
```

2. Tuple is Immutable

Tuple is an immutable data type. It means that the elements of a tuple cannot be changed after it has been created. An attempt to do this would lead to an error.

```
>>> tuple1 = (1,2,3,4,5)
TypeError: 'tuple' object does not support
item assignment
```

However an element of a tuple may be of mutable type, e.g., a list.

```
#4th element of the tuple2 is a list
>>> tuple2 = (1,2,3,[8,9])
#modify the list element of the tuple tuple2
>>> tuple2[3][1] = 10
#modification is reflected in tuple2
>>> tuple2
(1, 2, 3, [8, 10])
```

TUPLE OPERATIONS

1 Concatenation

Python allows us to join tuples using concatenation operator depicted by symbol +. We can also create a new tuple which contains the result of this concatenation operation.

```
>>> tuple1 = (1,3,5,7,9)
>>> tuple2 = (2,4,6,8,10)
>>> tuple1 + tuple2
#concatenates two tuples
```

```
(1, 3, 5, 7, 9, 2, 4, 6, 8, 10)
>>> tuple3 = ('Red','Green','Blue')
>>> tuple4 = ('Cyan', 'Magenta', 'Yellow','Black')
#tuple5 stores elements of tuple3 and tuple4
>>> tuple5 = tuple3 + tuple4
>>> tuple5
('Red','Green','Blue','Cyan','Magenta','Yellow','Black')
```

Concatenation operator can also be used for extending an existing tuple. When we extend a tuple using concatenation a new tuple is created.

```
>>> tuple6 = (1,2,3,4,5)
#single element is appended to tuple6
>>> tuple6 = tuple6 + (6,)
>>> tuple6
(1, 2, 3, 4, 5, 6)
#more than one elements are appended
>>> tuple6 = tuple6 + (7,8,9)
>>> tuple6
(1, 2, 3, 4, 5, 6, 7, 8, 9)
```

2 Repetition

Repetition operation is depicted by the symbol *. It isused to repeat elements of a tuple. We can repeat the tuple elements. The repetition operator requires the first operand to be a tuple and the second operand to be an integer only.

```
>>> tuple1 = ('Hello','World')
>>> tuple1 * 3
('Hello', 'World', 'Hello', 'World', 'Hello',
'World')
#tuple with single element
>>> tuple2 = ("Hello",)
>>> tuple2 * 4
('Hello', 'Hello', 'Hello', 'Hello')
```

3 Membership

The `in` operator checks if the element is present in the tuple and returns `True`, else it returns `False`.

```
>>> tuple1 = ('Red','Green','Blue')
>>> 'Green' in tuple1
True
```

The `not in` operator returns `True` if the element isnot present in the tuple, else it returns `False`.

```
>>> tuple1 = ('Red','Green','Blue')
>>> 'Green' not in tuple1
False
```

4 Slicing

Like string and list, slicing can be applied to tuples also.

```
#tuple1 is a tuple
>>> tuple1 = (10,20,30,40,50,60,70,80)
#elements from index 2 to index 6
>>> tuple1[2:7]
(30, 40, 50, 60, 70)
#all elements of tuple are printed
>>> tuple1[0:len(tuple1)]
(10, 20, 30, 40, 50, 60, 70, 80)
#slice starts from zero index
>>> tuple1[:5]
(10, 20, 30, 40, 50)
#slice is till end of the tuple
>>> tuple1[2:]
(30, 40, 50, 60, 70, 80)

#step size 2
>>> tuple1[0:len(tuple1):2]
(10, 30, 50, 70)
#negative indexing
>>> tuple1[-6:-4]
(30, 40)
#tuple is traversed in reverse order
>>> tuple1[::-1]
(80, 70, 60, 50, 40, 30, 20, 10)
```

TUPLE METHODS AND BUILT-IN FUNCTIONS

Python provides many functions to work on tuples. Table list some of the commonly used tuple methods and built-in functions.

Method	Description	Example

len()	Returns the length or the number of elements of the tuple passed as the argument	`>>> tuple1 = (10,20,30,40,50)`
		`>>> len(tuple1)`
		`5`

tuple()	Creates an empty tuple if no argument is passed
	
	Creates a tuple if a sequence is passed as argument
		`>>> tuple1 = tuple()`
		`>>> tuple1`
		`( )`
		`>>> tuple1 = tuple('aeiou')#string`
		`>>> tuple1`
		`('a', 'e', 'i', 'o', 'u')`
		
		`>>> tuple2 = tuple([1,2,3]) #list`
		`>>> tuple2`
		`(1, 2, 3)`
		
		`>>> tuple3 = tuple(range(5))`
		`>>> tuple3`
		`(0, 1, 2, 3, 4)`

sorted()	Takes elements in the tuple and returns a new sorted list. It should be noted that, sorted() does not make any change to the original tuple	>>> tuple1 = ("Rama","Heena","Raj", "Mohsin","Aditya")
		>>> sorted(tuple1)
		['Aditya', 'Heena', 'Mohsin', 'Raj', 'Rama']

min()
 	
max()
 	
sum()
	Returns minimum or smallest element of the tuple
	
	Returns maximum or largest element of the tuple
	
	Returns sum of the elements of the tuple
		>>> tuple1 = (19,12,56,18,9,87,34)
		>>> min(tuple1)
		9
		>>> max(tuple1)
		87
		>>> sum(tuple1)
		235

TUPLE ASSIGNMENT

Assignment of tuple is a useful feature in Python. It allows a tuple of variables on the left side of the assignment operator to be assigned respective values from a tuple on the right side. The number of variables on the left should be same as the number of elements in the tuple.

Example

```
#The first element 10 is assigned to num1 and
#the second element 20 is assigned to num2.
>>> (num1,num2) = (10,20)
>>> print(num1)
10
>>> print(num2)
20
>>> record = ( "Pooja",40,"CS")
>>> (name,rollNo,subject) = record
>>> name
'Pooja'
>>> rollNo
40
>>> subject
'CS'
>>> (a,b,c,d) = (5,6,8)
ValueError: not enough values to unpack (expected 4, got 3)
```

If there is an expression on the right side then first that expression is evaluated and finally the result is assigned to the tuple.

Example 10.3

```
#15 is assigned to num3 and
#25 is assigned to num4
>>> (num3,num4) = (10+5,20+5)
>>> print(num3)
15
>>> print(num4)
25
```

NESTED TUPLES

A tuple inside another tuple is called a nested tuple. In the following program, roll number, name and marks (in percentage) of students are saved in a tuple. To store details of many such students we can create a nested tuple.

Program 10 This is a program to create a nested tuple to store roll number, name and marks of students

```
#To store records of students in tuple and print them
st=((101,"Aman",98),(102,"Geet",95),(103,"Sahil",87),(104,"Pawan",79))
print("S_No"," Roll_No"," Name"," Marks")
for i in range(0,len(st)):
    print((i+1),'\t',st[i][0],'\t',st[i][1],'\t',st[i][2])
```

Output:

```
S_No Roll_No Name Marks
1 101 Aman 98
2 102 Geet 95
3 103 Sahil 87
4 104 Pawan 79
```

TUPLE HANDLING

Program : Write a program to swap two numbers without using a temporary variable.

```
#Program 10-2
#Program to swap two numbers
num1 = int(input('Enter the first number: '))
num2 = int(input('Enter the second number: '))
print("\nNumbers before swapping:")
print("First Number:",num1)
print("Second Number:",num2)
(num1,num2) = (num2,num1)
print("\nNumbers after swapping:")
print("First Number:",num1)
print("Second Number:",num2)
```

Output:

```
Enter the first number: 5
Enter the second number: 10
Numbers before swapping:
First Number: 5
Second Number: 10
Numbers after swapping:
First Number: 10
Second Number: 5
```

INTRODUCTION TO DICTIONARIES

The data type *dictionary* fall under mapping. It is a mapping between a *set of keys* and a *set of values*. The key-value pair is called an *item*. A key is separated from its value by a colon(:) and consecutive items are separated by commas. Items in dictionaries are unordered, so we may not get back the data in the same order in which we had entered the data initially in the dictionary.

Creating a Dictionary

To create a dictionary, the items entered are separated by commas and enclosed in curly braces. Each item is a key value pair, separated through colon (:). The keys in the dictionary must be unique and should be of any immutable data type, i.e., number, string or tuple. The values can be repeated and can be of any data type.

Example

```
#dict1 is an empty Dictionary created
#curly braces are used for dictionary
>>> dict1 = {}
>>> dict1
{}
#dict2 is an empty dictionary created using
#built-in function
>>> dict2 = dict()
>>> dict2
{}
#dict3 is the dictionary that maps names
#of the students to respective marks in
#percentage
>>> dict3 = {'Mohan':95,'Ram':89,'Suhel':92,'Sangeeta':85}
>>> dict3
{'Mohan': 95, 'Ram': 89, 'Suhel': 92,'Sangeeta': 85}
```

Accessing Items in a Dictionary

We have already seen that the items of a sequence(string, list and tuple) are accessed using a technique called indexing. The items of a dictionary are accessed via the keys rather than via their relative positions or indices. Each key serves as the index and maps to a value.
The following example shows how a dictionary returns the value corresponding to the given key:

```
>>> dict3 = {'Mohan':95,'Ram':89,'Suhel':92,
'Sangeeta':85}
>>> dict3['Ram']
89
>>> dict3['Sangeeta']
85
#the key does not exist
>>> dict3['Shyam']
KeyError: 'Shyam'
```

In the above examples the key `'Ram'` always maps to the value 89 and key `'Sangeeta'` always maps to the value 85. So the order of items does not matter. If the key is not present in the dictionary we get KeyError.

DICTIONARIES ARE MUTABLE

Dictionaries are mutable which implies that the contents of the dictionary can be changed after it has been created.

Adding a new item

We can add a new item to the dictionary as shown in the following example:

```
>>> dict1 = {'Mohan':95,'Ram':89,'Suhel':92,'Sangeeta':85}
>>> dict1['Meena'] = 78
```

```
>>> dict1
{'Mohan': 95, 'Ram': 89, 'Suhel': 92,'Sangeeta': 85, 'Meena': 78}
```

Modifying an Existing Item

The existing dictionary can be modified by just overwriting the key-value pair. Example to modify a given item in the dictionary:

```
>>> dict1 = {'Mohan':95,'Ram':89,'Suhel':92,
'Sangeeta':85}
#Marks of Suhel changed to 93.5
>>> dict1['Suhel'] = 93.5
>>> dict1
{'Mohan': 95, 'Ram': 89, 'Suhel': 93.5,'Sangeeta': 85}
```

DICTIONARY OPERATIONS

Membership

The membership operator `in` checks if the key is presentin the dictionary and returns `True`, else it returns `False`.

```
>>> dict1 = {'Mohan':95,'Ram':89,'Suhel':92,'Sangeeta':85}
>>> 'Suhel' in dict1
True
```

The `not in` operator returns `True` if the key is not present inthe dictionary, else it returns `False`.

```
>>> dict1 = {'Mohan':95,'Ram':89,'Suhel':92,'Sangeeta':85}
>>> 'Suhel' not in dict1
False
```

TRAVERSING A DICTIONARY

We can access each item of the dictionary or traverse adictionary using `for` loop.

```
>>> dict1 = {'Mohan':95,'Ram':89,'Suhel':92,'Sangeeta':85}
```

Method 1

```
>>> for key in dict1:
print(key,':',dict1[key])
Mohan: 95
Ram: 89
Suhel: 92
Sangeeta: 85
```

Method 2

```
>>> for key,value in dict1.items():
print(key,':',value)
Mohan: 95
Ram: 89
Suhel: 92
Sangeeta: 85
```

DICTIONARY METHODS AND BUILT-IN FUNCTIONS

Python provides many functions to work on dictionaries. Table lists some of the commonly used dictionary methods.

Method	Description	Example

len()	Returns the length or number of key: value pairs of the dictionary passed as the argument	`>>> dict1 = {'Mohan':95,'Ram':89, 'Suhel':92, 'Sangeeta':85}`
		`>>> len(dict1)`
		`4`

dict()	Creates a dictionary from a sequence of key-value pairs	`pair1 = [('Mohan',95),('Ram',89), ('Suhel',92),('Sangeeta',85)]`
		`>>> pair1`
		`[('Mohan', 95), ('Ram', 89), ('Suhel', 92), ('Sangeeta', 85)]`
		`>>> dict1 = dict(pair1)`
		`>>> dict1`
		`{'Mohan': 95, 'Ram': 89, 'Suhel': 92, 'Sangeeta': 85}`

keys()	Returns a list of keys in the dictionary	`>>> dict1 = {'Mohan':95, 'Ram':89, 'Suhel':92, 'Sangeeta':85}`
		`>>> dict1.keys()`
		`dict_keys(['Mohan', 'Ram', 'Suhel', 'Sangeeta'])`

values()	Returns a list of values in the dictionary	`>>> dict1 = {'Mohan':95, 'Ram':89, 'Suhel':92, 'Sangeeta':85}`
		`>>> dict1.values()`
		`dict_values([95, 89, 92, 85])`

items()	Returns a list of tuples(key – value) pair	`>>> dict1 = {'Mohan':95, 'Ram':89, 'Suhel':92, 'Sangeeta':85}`
		`>>> dict1.items()`
		`dict_items([( 'Mohan', 95), ('Ram', 89), ('Suhel', 92), ('Sangeeta', 85)])`

get()	Returns the value corresponding to the key passed as the argument
	
	If the key is not present in the dictionary it will return None
		`>>> dict1 = {'Mohan':95, 'Ram':89, 'Suhel':92, 'Sangeeta':85}`
		`>>> dict1.get('Sangeeta')`
		`85`
		`>>> dict1.get('Sohan')`
		`>>>`

update()	appends the key-value pair of the dictionary passed as the argument to the key-value pair of the given dictionary	`>>> dict1 = {'Mohan':95, 'Ram':89, 'Suhel':92, 'Sangeeta':85}`
		`>>> dict2 = {'Sohan':79,'Geeta':89}`
		`>>> dict1.update(dict2)`
		`>>> dict1`
		`{'Mohan': 95, 'Ram': 89, 'Suhel': 92, 'Sangeeta': 85, 'Sohan': 79, 'Geeta': 89}`
		`>>> dict2`
		`{'Sohan': 79, 'Geeta': 89}`

del()	Deletes the item with the given key
	To delete the dictionary from the memory we write:
	`del Dict_name`
		`>>> dict1 = {'Mohan':95,'Ram':89, 'Suhel':92, 'Sangeeta':85}`
		`>>> del dict1['Ram']`
		`>>> dict1`
		`{'Mohan':95,'Suhel':92, 'Sangeeta': 85}`
		
		`>>> del dict1 ['Mohan']`
		`>>> dict1`
		`{'Suhel': 92, 'Sangeeta': 85}`
		`>>> del dict1`
		`>>> dict1`
		`NameError: name 'dict1' is not defined`

clear()	Deletes or clear all the items of the dictionary	`>>> dict1 = {'Mohan':95,'Ram':89, 'Suhel':92, 'Sangeeta':85}`
		`>>> dict1.clear()`
		`>>> dict1`
		`{ }`

MANIPULATING DICTIONARIES

In this chapter, we have learnt how to create a dictionary and apply various methods to manipulate it. The following programs show the application of those manipulation methods on dictionaries.

Program : Create a dictionary 'ODD' of odd numbers between 1 and 10, where the key is the decimal number and the value is the corresponding number in words. Perform the following operations on this dictionary:

(a) Display the keys
(b) Display the values
(c) Display the items
(d) Find the length of the dictionary
(e) Check if 7 is present or not
(f) Check if 2 is present or not
(g) Retrieve the value corresponding to the key 9
(h) Delete the item from the dictionary corresponding to the key 9

```
>>> ODD = {1:'One',3:'Three',5:'Five',7:'Seven',9:'Nine'}
>>> ODD
{1: 'One', 3: 'Three', 5: 'Five', 7: 'Seven', 9: 'Nine'}
```

(a) Display the keys

```
>>> ODD.keys()
dict_keys([1, 3, 5, 7, 9])
```

(b) Display the values

```
>>> ODD.values()
dict_values(['One', 'Three', 'Five', 'Seven', 'Nine'])
```

(c) Display the items

```
>>> ODD.items()
dict_items([(1, 'One'), (3, 'Three'), (5, 'Five'), (7, 'Seven'),(9,
'Nine')])
```

(d) Find the length of the dictionary

```
>>> len(ODD)
5
```

(e) Check if 7 is present or not

```
>>> 7 in ODD
True
```

(f) Check if 2 is present or not

```
>>> 2 in ODD
False
```

(g) Retrieve the value corresponding to the key 9

```
>>> ODD.get(9)
'Nine'
```

(h) Delete the item from the dictionary corresponding to the key 9

```
>>> del ODD[9]
>>> ODD
{1: 'One', 3: 'Three', 5: 'Five', 7: 'Seven'}
```