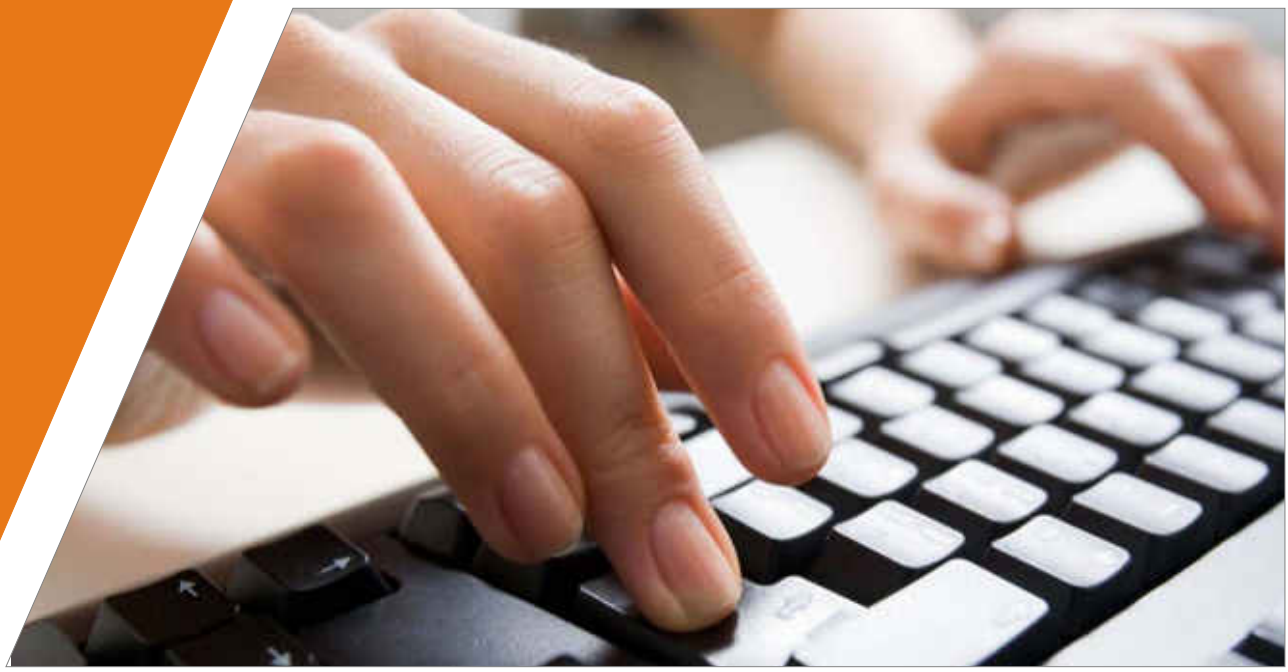
Viva Technologies



PART 2

PYTHON PROGRAMMING

THATIPUR ,CHAUHAN PIYAU, DARPANCOLONY,
VINAY NAGAR, HAZIRA, DD NAGAR Ph: 94257-01888, 02888, 03888

DISCLAIMER

This book has been designed & developed by Viva Technologies for institute purpose only. It is not for sale in any means.

Python Revision Tour-II

2

Python Revision Tour-II

In This Chapter

2.1 INTRODUCTION

The purpose of *Revision Tour chapters* is to brush up all the concepts you have learnt in class XI. The previous chapter – Python Revision Tour I – covered basic concepts like : Python basics, Data handling, and control flow statements. This chapter is going to help you recall and brush up concepts like *Strings, Lists, Tuples* and *Dictionaries*.

2.2 STRINGS IN PYTHON

Strings in Python are stored as individual characters in contiguous locations, with two-way index for each location. Consider following figure (Fig. 2.1).

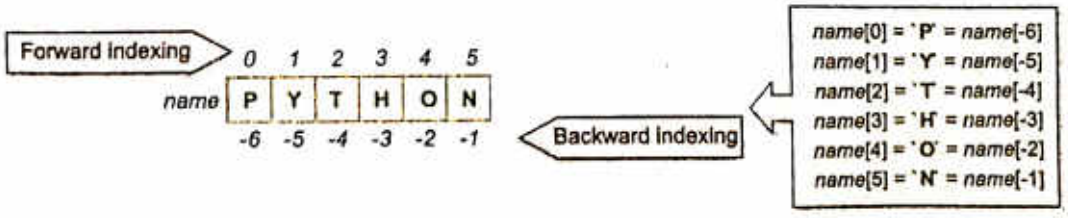


Figure 2.1 Structure of a Python String.

From Fig. 2.1 you can infer that :

- Strings in Python are stored by storing each character separately in contiguous locations.
- The characters of the strings are given two-way indices :
  - 0, 1, 2, ... *size-1* in the *forward direction* and
  - -1, -2, -3, ... *-size* in the *backward direction.*

Thus, you can access any character as **<stringname>[<index>]** *e.g.*, to access the first character of string name shown in Fig. 2.1, you'll write **name[0]**, because the index of first character is 0. You may also write **name [-6]** for the above example *i.e.*, when string name is storing "PYTHON".

Length of string variable can be determined using function **len(<string>)**, *i.e.*, to determine length of above shown string name, you may write :

```
len(name)
```

which will give you 6, indicating that string name stores *six* characters.

2.2.1 Item Assignment not Supported

One important thing about Python strings is that you cannot change the individual letters of a string by assignment because strings are immutable and hence item assignment is not supported, *i.e.*,

```
name = 'hello'
name[0] = 'p'
```

will cause an error like :

```
Traceback (most recent call last):
  File "<pyshell#3>", line 1, in <module>
    name[0] = 'p'
TypeError: 'str' object does not support item assignment
```

2.2.2 Traversing a String

Traversing refers to iterating through the elements of a string, one character at a time.

To traverse through a string, you can write a loop like :

```
code = "Powerful"
for ch in code :
    print(ch, '~', end = ' ')
```

NOTE

Traversing refers to iterating through the elements of a string, one character at a time.

The above code will print :

```
P ~ o ~ w ~ e ~ r ~ f ~ u ~ l ~
```

2.2.3 String Operators

In this section, you'll be learning to work with various operators that can be used to manipulate strings in multiple ways.

1. String Concatenation Operator +

The + operator creates a new string by joining the two operand strings, *e.g.*,

```
"power" + "ful"
```

will result into

```
'powerful'
```

Caution!

The + operator has to have **both operands of the same** type either of number types (for addition) or of string types (for multiplication). It cannot work with one operand as string and one as a number.

2. String Replication Operator *

To use a * operator with strings, you need two types of operands – a string and a number, *i.e.*, as number * string or string * number, where ***string operand*** tells the string to be replicated and ***number operand*** tells the number of times, it is to be repeated; Python will create a new string that is a number of repetitions of the string operand.

For example,

```
3 * "Ha!"
```

will return

```
'Ha!Ha!Ha!'
```

Caution!

The * operator has to either have both operands of the *number* types (for multiplication) or one *string* type and one *number* type (for replication). It cannot work with both operands of string types.

3. Membership Operators

There are *two* membership operators for strings (in fact, for all sequence types). These are **in** and **not in**.

in returns *True* if a character or a substring exists in the given string ; *False* otherwise

not in returns *True* if a character or a substring does not exist in the given string ; *False* otherwise

Both membership operators (when used with strings), require that both operands used with them are of string type, *i.e.*,

```
<string> in <string>
<string> not in <string>
```

Now, let's have a look at some examples :

```
>>> "a" in "heya"
True
>>> "jap" in "heya"
False
```

```
>>> "jap" in "Japan"
False
>>> sub = "help"
False
>>> sub2 not in string
True
>>> string = 'helping hand'
>>> sub2 = 'HELP'
>>> sub in string
True
>>> sub2 in string
False
>>> sub not in string
False
```

4. Comparison Operators

Python's standard comparison operators *i.e.*, all relational operators (<, <=, >, >=, ==, !=,) apply to strings also. The comparisons using these operators are based on the standard character-by-character comparison rules for ASCII or Unicode (*i.e.*, dictionary order). Thus, you can make out that :

```
"a" == "a"      will give      True
"abc" == "abc"  will give      True
"a" != "abc"    will give      True
"A" != "a"      will give      True
"ABC" == "abc"  will give      False (letters' case is different)
"abc" != "Abc"  will give      True  (letters' case is different)
```

String comparison principles are :

- Strings are compared on the basis of lexicographical ordering (ordering in dictionary).
- Upper-case letters are considered smaller than the lower-case letters.

Determining ASCII/Unicode Value of a Single Character

Python offers a built-in function **ord( )** that takes a single character and returns the corresponding ASCII value or Unicode value :

```
ord(<single-character>)  #returns ASCII value of passed character
```

The opposite of **ord( )** function is **chr( )**, *i.e.*, while **ord( )** returns the ASCII value of a character, the **chr( )** takes the ASCII value in integer form and returns the character corresponding to that ASCII value.

The general syntax of **chr( )** function is :

```
chr(<int>)      # Gives character corresponding to passed ASCII
                # value given as integer
```

Consider these examples :

```
>>> ord('A')
65
>>> ord(u'A')
65
>>> chr(65)
'A'
>>> chr(97)
'a'
```

2.2.4 String Slices

STRING SLICE

String Slice refers to part of a string containing some contiguous characters from the string.

The term 'string slice' refers to a part of the string, where strings are sliced using a range of indices. That is, for a string say **name**, if we give **name[n : m]** where **n** and **m** are integers and legal indices, Python will return a slice of the string by returning the characters falling between indices *n* and *m* : starting at *n*, *n* + 1, *n* + 2 ...till *m* – 1.

NOTE

For any index n, s[:n] + s[n:] will give you original string *s*.

Following figure (Fig. 2.2) shows some string slices using the string :

```
helloString ="Hello World" :
```

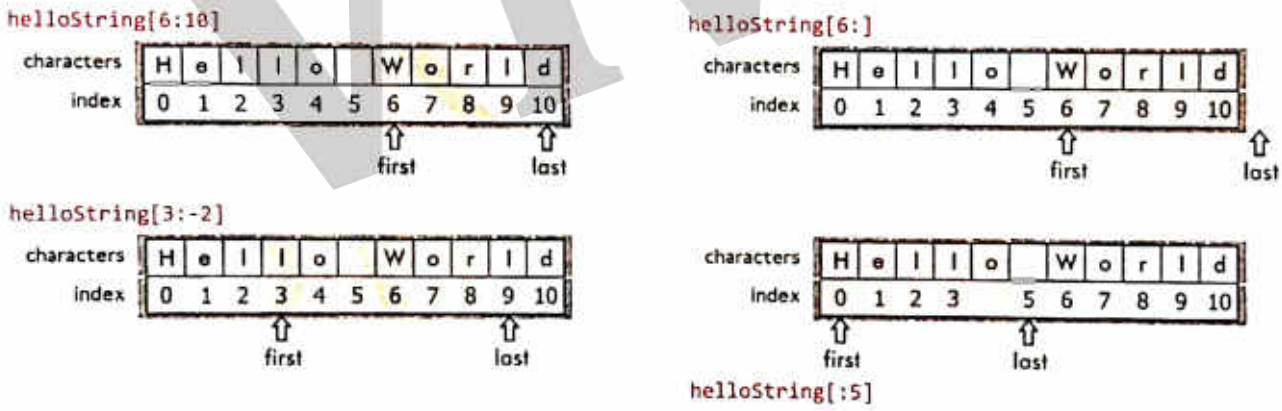


Figure 2.2 String Slicing in Python.

Interesting Inference

Using the same string slicing technique, you will find that

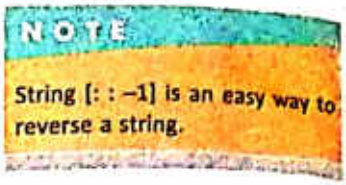
- for any index *n*, s[:n] + s[n:] will give you original string s.

This works even for *n* negative or out of bounds.

Consider the string namely **word** storing *'amazing'*.

```
>>> word[3:], word [:3]
'zing' 'ama'
```

```
>>> word[:3] + word[3:]
'amazing'
>>> word[:-7], word [-7:]
'' 'amazing'
>>> word[:-7] + word[-7:]
'amazing'
```



- Index out of bounds causes error with strings but slicing a string outside the bounds does not cause error.

```
s = "Hello"
print (s[5])
```

Will cause error because 5 is invalid index-out of bounds, for string "Hello"

But if you give

```
s = "Hello"
print (s[4 : 8])
print (s[5 : 10])
```

One limit is outside the bounds (length of Hello is 5 and thus valid indexes are 0–4)

Both limits are outside the bounds

the above will not give any error and print output as :

```
o

```

empty string

i.e., letter o followed by empty string in next line.

2.2.5 String Functions

Python also offers many built-in functions and methods for string manipulation. The string manipulation methods that are being discussed below can be applied to strings as per following syntax :

```
<stringObject>.<method name> ()
```

For instance, if you have a string namely **str** = "Rock the World" and you want to find its length, you will write the code somewhat like shown below :

```
>>> str = "Rock the World."
>>> str.length( )
15
>>> str2 = "New World"
>>> str2.length( )
9
```

*see the string object is **str** and method name is **length( )**.*

Do you know that following websites and web applications have used Python extensively : *Instagram, Dropbox, Google, Netflix, Spotify, Quora, Reddit, Facebook,* and many others ?

Table 2.1 *Python's built-in string manipulation methods*

`string.capitalize()`	Returns a copy of the ***string*** with its first character capitalized.
	Example
	`>>> ' i love my India'.capitalize()`
	`I love my India`

`string.find (sub[, start[, end]])`	Returns the lowest index in the ***string*** where the substring ***sub*** is found within the slice range of *start* and *end*. Returns -1 if ***sub*** is not found.
	Example
	`>>> string = 'it goes as - ringa ringa roses'`
	`>>> sub = 'ringa'`
	`>>> string.find(sub, 15, 22)`
	`-1`
	`>>> string.find(sub, 15, 25)`
	`19`

`string.isalnum()`
`string.isalpha()`
`string.isdigit()`
	Returns **True** if the characters in the ***string*** are alphanumeric (alphabets or numbers) and there is at least one character, **False** otherwise.
	Returns **True** if all characters in the ***string*** are alphabetic and there is at least one character, **False** otherwise.
	Returns **True** if all the characters in the ***string*** are digits. There must be at least one digit, otherwise it returns **False**.
	Examples
	`>>> string = "abc123"`
	`>>> string2 = 'hello'`
	`>>> string3 = '12345'`
	`>>> string4 = ' '`
	`>>> string.isalnum()` `True` `>>> string2.isalnum()` `True` `>>> string3.isalnum()` `True` `>>> string4.isalnum()` `False`
	`>>> string.isalpha()` `False` `>>> string2.isalpha()` `True` `>>> string3.isalpha()` `False` `>>> string4.isalpha()` `False`
	`>>> string.isdigit()` `False` `>>> string2.isdigit()` `False` `>>> string3.isdigit()` `False` `>>> string4.isdigit()` `True`

`string.isspace()`	Returns **True** if there are only whitespace characters in the ***string***. There must be at least one character. It returns **False** otherwise.
	Example
	`>>> string = "   "  # stores three spaces`
	`>>> string2 = ""  # an empty string`
	`>>> string.isspace()`
	`True`
	`>>> string2.isspace()`
	`False`

`string.islower()`
 	
`string.isupper()`
	Returns **True** if all cased characters in the ***string*** are lowercase. There must be at least one cased character. It returns **False** otherwise.
	Tests whether all cased characters in the ***string*** are uppercase and requires that there be at least one cased character. Returns **True** if so and **False** otherwise.
	Examples
	`>>> string = 'hello'`
	`>>> string2 = 'THERE'`
	`>>> string3 = 'Goldy'`
	`>>> string.islower()`
	`True`
	`>>> string2.islower()`
	`False`
	`>>> string3.islower()`
	`False`
	`>>> string = "HELLO"`
	`>>> string2 ="There"`
	`>>> string3 = "goldy"`
	`>>> string.isupper()`
	`True`
	`>>> string2.isupper()`
	`False`
	`>>> string3.isupper()`
	`False`
	`>>> string4.isupper()`
	`True`
	`>>> string5.isupper()`
	`False`

`string.lower()`	Returns a copy of the ***string*** converted to lowercase. **Example**
	`>>> string.lower()  #string = "HELLO"`
	`'hello'`

`string.upper()`	Returns a copy of the ***string*** converted to uppercase. **Example**
	`>>> string.upper()  #string = "hello"`
	`'HELLO'`

`string.lstrip([chars])`
 	
`string.rstrip([chars])`
	Returns a copy of the ***string*** with leading characters removed.
	If used without any argument, it removes the leading whitespaces.
	One can use the optional **chars** argument to specify a set of characters to be removed.
	The **chars** argument is not a prefix ; rather, all combinations of its values (all possible substrings from the given string argument **chars**) are stripped when they lead the ***string***.
	Returns a copy of the *string* with trailing characters removed.
	If used without any argument, it removes the leading whitespaces.
	The **chars** *argument* is a ***string*** specifying the set of characters to be removed.
	The **chars** argument is not a suffix; rather, all combinations of its values are stripped.
	Examples
	`>>> string2 = 'There'`
	`'There'`
	`>>> string2.lstrip( 'The' )`
	`'re'`
	'The', 'Th', 'he', 'Te' , 'T', 'h', 'e'" and their reversed strings are matched, if any of these found, is removed from the left of the string 'The' found , hence removed
	`>>> "saregamapadhanisa".lstrip( "tears" )`
	`'gamapadhanisa'`
	`>>> string2.rstrip('care')`
	`'Th'`
	`>>> "saregamapadhanisa".rstrip( "tears" )`
	`'saregamapadhani'`

Program 2.1 *Program that reads a line and prints its statistics like :*

Number of uppercase letters :
Number of lowercase letters:
Number of alphabets :
Number of digits:

```
line = input( "Enter a line :" )
lowercount = uppercount = 0
digitcount = alphacount = 0
for a in line :
    if a.islower() :
        lowercount += 1
    elif a.isupper() :
        uppercount += 1
    elif a.isdigit() :
        digitcount += 1
    if a.isalpha() :
        alphacount += 1
print("Number of uppercase letters :", uppercount)
print("Number of lowercase letters :", lowercount)
print("Number of alphabets :", alphacount)
print("Number of digits :", digitcount)
```

```
Enter a line : Hello 123, ZIPPY zippy Zap
Number of uppercase letters :  7
Number of lowercase letters : 11
Number of alphabets :  18
Number of digits :  3
```

2.3 LISTS IN PYTHON

A list is a standard data type of Python that can store a sequence of values belonging to any type. The Lists are depicted through square brackets, *e.g.*, following are some lists in Python :

```
[]                          #list with no member, empty list
[1, 2, 3]                   #list of integers
[1, 2.5, 3.7, 9]            #list of numbers (integers and floating point)
['a', 'b', 'c']             # list of characters
['a', 1, 'b', 3.5, 'zero']  # list of mixed value types
['One', 'Two', 'Three']     # list of strings
```

Lists are mutable (*i.e.*, modifiable) *i.e.*, you can change elements of a list in place. List is one of the two mutable types of Python – *Lists* and *Dictionaries* are mutable types ; all other data types of Python are immutable.

2.3.1 Creating Lists

To create a list, put a number of expressions, separated by commas in square brackets. That is, to create a list you can write in the form given below :

```
L = []
L = [value, ...]
```

This construct is known as a list display construct.

Creating Empty List

The empty list is [ ]. You can also create an empty list as :

```
L = list()                    #It will generate an empty list and name that list as L.
```

Creating Lists from Existing Sequences

You can also use the built-in list type object to create lists from sequences as per the syntax given below :

```
L = list(<sequence>)
```

where *<sequence>* can be any kind of sequence object including *strings, tuples,* and *lists.*

Consider following examples :

```
>>> l1 = list('hello')          #creating list from string
>>> l1
['h', 'e', 'l', 'l', 'o']
>>> t = ('w', 'e', 'r', 't', 'y')
>>> l2 = list(t)                #creating list from tuple
>>> l2
['w', 'e', 'r', 't', 'y']
```

Creating List from Keyboard Input

You can use this method of creating lists of single characters or single digits via keyboard input. Consider the code below :

```
>>> l1 = list( input('Enter list elements:'))
Enter list elements: 234567
>>> l1
['2', '3', '4', '5', '6', '7']
```

Notice, this way the data type of all characters entered is string even though we entered digits. To enter a list of integers through keyboard, you can use the method given below.

Most commonly used method to input lists is **eval(input( ))**[1] as shown below :

```
list = eval(input("Enter list to be added :"))
print("List you entered :", list)
```

when you execute it, it will work somewhat like :

```
Enter list to be added : [67, 78, 46, 23]
List you entered : [67, 78, 46, 23]
```

1. Please note, sometimes (not always) eval( ) does not work in Python shell. At that time, you can run it through a script too.

2.3.2 Lists vs. Strings

Lists and strings have lots in common yet key differences too. This section is going to summarize the similarities and differences between lists and strings.

2.3.2A Similarity between Lists and Strings

Lists are similar to strings in following ways :

- *Length* — Function *len*(*L*) returns the number of items (count) in the list L.
- *Indexing and Slicing*

  L[*i*] returns the item at index *i* (the first item has index 0), and
  L[*i*:*j*] returns a new list, containing the objects between *i* and *j*.
- *Membership operators*

  Both **'in'** and **'not in'** operators work on Lists just like they work for other sequences such as strings. (Operator **in** tells if an element is present in the list or not, and **not in** does the opposite.)
- *Concatenation and Replication operators + and **

  The + operator adds one list to the end of another. The * operator repeats a list.
- *Accessing Individual Elements*

  Like strings, the individual elements of a list are accessed through their indexes.

  Consider following examples :

```
>>> vowels = ['a', 'e', 'i', 'o', 'u']
>>> vowels[0]
'a'
>>> vowels[4]
'u'
>>> vowels[-1]
'u'
>>> vowels[-5]
'a'
```

2.3.2B Difference between Lists and Strings

The lists and strings are different from one another in following ways :

- *Storage* — Lists are stored in memory exactly like strings, except that because some of their objects are larger than others, they store a reference at each index instead of single character as in strings.
- *Mutability* — Strings are not mutable, while lists are. You cannot change individual elements of a string in place, but Lists allow you to do so. That is, following statement is fully valid for Lists (though not for strings) :

```
L[i] = <element>
```

For example, consider the same **vowels** list created above and have a look at following code :

```
>>> vowels[0] = 'A'
>>> vowels
['A', 'e', 'i', 'o', 'u']
>>> vowels[-4] = 'E'
>>> vowels
['A', 'E', 'i', 'o', 'u']
```

Notice, it changed the element in place; ('a' changed to 'A' and 'e' changed to 'E') no new list created – because Lists are MUTABLE.

NOTE

Lists are similar to strings in many ways like indexing, slicing and accessing individual elements but they are different in the sense that Lists are mutable while strings are not.

2.3.3 List Operations

In this section, we shall talk about most common list operations, briefly.

2.3.3A Traversing a List

Traversing a list means accessing and processing each element of it. The *for loop* makes it easy to traverse or loop over the items in a list, as per following syntax :

```
for <item> in <List>:
    process each item here
```

For example, following loop shows each item of a list L in separate lines :

```
L = ['P', 'y', 't', 'h', 'o', 'n']
for a in L :
    print(a)
```

The above loop will produce result as :

```
P
y
t
h
o
n
```

2.3.3B Joining Lists

The concatenation operator +, when used with two lists, joins two lists and returns the concatenated list. Consider the example given below :

```
>>> lst1 = [ 1, 4, 9]
>>> lst2 = [6, 12, 20 ]
>>> lst1 + lst2
[1, 4, 9, 6, 12, 20]
```

The + operator when used with lists requires that both the operands must be of list types.

2.3.3C Repeating or Replicating Lists

Like strings, you can use * operator to replicate a list specified number of times, *e.g.,* (considering the same list lst1 = [1, 3, 5])

```
>>> lst1 * 3
[1, 4, 9, 1, 4, 9, 1, 4, 9]
```

Like strings, you can only use an integer with a * operator when trying to replicate a list.

2.3.3D Slicing the Lists

List slices, like string slices are the sub part of a list extracted out. You can use indexes of list elements to create list slices as per following format :

```
seq = L[start:stop]
```

Consider the following example :

```
>>> lst =[10, 12, 14, 20, 22, 24, 30, 32, 34]
>>> seq = lst [ 3: -3]
>>> seq
[20, 22, 24]
>>> lst =[10, 12, 14, 20, 22, 24, 30, 32, 34]
>>> lst [3:30]
[20, 22, 24, 30, 32, 34]
>>> lst [-15 :7]
[10, 12, 14, 20, 22, 24, 30]
```

Giving upper limit way beyond the size of the list, but Python return elements from list falling in range 3 onwards <30

Giving lower limit much lower, but Python returns elements from list falling in range –15 onwards <7

Lists also support slice steps too. That is, if you want to extract, not consecutive but every other element of the list, there is a way out – the *slice steps*. The *slice steps* are used as per following format :

```
seq = L[start:stop:step]
```

Consider some examples to understand this.

```
>>> lst
[10, 12, 14, 20, 22, 24, 30, 32, 34]
>>> lst[0 : 10 : 2]
[10, 14, 22, 30, 34]
>>> lst[2 : 10 : 3]
[14, 24, 34]
```

Include every 2nd element, i.e., skip 1 element in between. Check resulting list slice

Include every 3rd element, i.e., skip 2 elements in between

Using Slices for List Modification

You can use slices to overwrite one or more list elements with one or more other elements. Following examples will make it clear to you :

```
>>> L = ["one", "two", "THREE"]
>>> L[0:2] = [ 0, 1 ]
>>> L
[0, 1, "THREE"]

>>> L =["one", "two", "THREE"]
>>> L[0:2] = "a"
>>> L
["a", "THREE"]
```

NOTE

Like strings, in list slices, you can give *start* and *stop* beyond limits of list and it won't raise IndexError, rather it will return the elements falling between specified boundaries.

2.3.4 List Manipulation

You can perform various operations on lists like : *appending, updating, deleting* etc.

Appending Elements to a List

You can also add items to an existing sequence. The **append( )** method **adds a single item to the end of the list**. It can be done as per following format :

```
L.append(item)
```

Consider some examples :

```
>>> lst1 = [10, 12, 14]
>>> lst1.append(16)
>>> lst1
[10, 12, 14, 16]
```

The element specified as argument to append( ) is added at the end of existing list

Updating Elements to a List

To update or change an element of the list in place, you just have to assign new value to the element's index in list as per syntax :

```
L[index] = <new value>
```

Consider following example :

```
>>> lst1 = [10, 12, 14, 16]
>>> lst1[2] = 24
>>> lst1
[10, 12, 24, 16]
```

Statement updating an element (3rd element – having index 2) in the list.
*Display the list to see the **updated list**.*

Deleting Elements from a List

You can also remove items from lists. The **del** statement can be used to remove an individual item, or to remove all items identified by a slice.

It is to be used as per syntax given below :

```
del List [ <index>]          # to remove element at index
del List [<start> : <stop> ] # to remove elements in list slice
```

e.g.,

```
>>> del lst[10:15]
>>> lst
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 17, 18, 19, 20]
```

Delete all elements between indexes 10 to 15 in list namely lst. Compare the result displayed below

If you use **del <lstname>** only *e.g.*, **del lst**, it will delete all the elements and the list object too. After this, no object by the name *lst* would be existing.

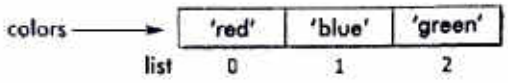
You can also use **pop( )** method to remove single element, not list slices.

The *pop( ) method* is covered in a later section, *List Functions*.

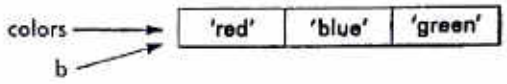
2.3.5 Making True Copy of a List

Assignment with an assignment operator (=) on lists does not make a copy. Instead, assignment makes the two variables point to the one list in memory (called *shallow copy*).

```
colors = ['red', 'blue', 'green']
```



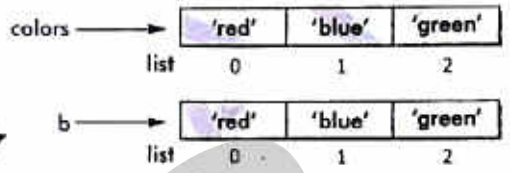

```
b = colors      ## Does not copy the list
```



So, if you make changes in one list, the other list will also report those changes because these two list-names are labels referring to same list because '=' copied the reference not the actual list.

To make **b** true copy of list **colors** *i.e.*, an independent list identical to list **colors** you should create copy of list as follows :

```
b = list(colors)
```



Now *colors* and *b* are separate lists (*deep copy*).

2.3.6 List Functions

Python also offers many built-in functions and methods for list manipulation. These can be applied to list as per following syntax :

```
<listObject>.<method name>()
```

1. The index method

This function returns the index of first matched item from the list.

```
List.index (<item>)
```

For example, for a list L1 = [13, 18, 11, 16, 18, 14],

```
>>> L1.index (18)
1
```

returns the index of first value 18, even if there is another value 18 at index 4.

However, if the given item is not in the list, it raises exception value Error

2. The append method

The *append( )* method **adds an item to the end of the list.** It works as per following syntax :

```
List.append(<item>)
```

– *Takes exactly one element and returns no value*

For example, to add a new item "yellow" to a list containing colours, you may write :

```
>>> colours = [ 'red', 'green', 'blue']
>>> colours.append('yellow')
>>> colours
['red', 'green', 'blue', 'yellow']
```

See the item got added at the end of the list

The ***append( )* does not return the new list**, just modifies the original.

3. The extend method

The *extend( )* method is also used for adding multiple elements (given in the form of a list) to a list.

The *extend( )* function works as per following format :

```
List.extend(<list>)
```

– *Takes exactly one element (a list type) and returns no value*

That is *extend( )* takes a list as an argument and appends all of the elements of the argument list to the list object on which *extend( )* is applied. Consider the following example :

```
>>> t1 = ['a', 'b', 'c']
>>> t2 = ['d', 'e']
>>> t1.extend(t2)
>>> t1
['a', 'b', 'c', 'd', 'e']
>>> t2
['d', 'e']
```

*Extend the list **t1**, by adding all elements of **t2***

*See the elements of list **t2** are added at the end of list **t1***

*But list **t2** remains unchanged.*

Difference between append( ) and extend( ) methods

While append( ) function adds one element to a list, *extend( )* can add multiple elements from a list supplied to it as argument.

4. The insert method

The *insert( )* function inserts an item at a given position. It is used as per following syntax :

```
List.insert( <pos>, <item>)
```

– *Takes two arguments and returns no value.*

The first argument **<pos> is the index of the element before which the second argument <item> is to be added.** Consider the following example:

```
>>> t1 = [ 'a', 'e', 'u']
>>> t1.insert(2, 'i')              # inset element 'i' at index 2.
>>> t1
['a', 'e', 'i', 'u']
```

See element 'i' inserted at index 2

5. The pop method

The *pop( )* is used to remove the item from the list. It is used as per following syntax :

```
List.pop(<index>)                 # <index is optional argument
```

– *Takes one optional argument and returns a value – the item being deleted*

Thus, pop( ) removes an element from the given position in the list, and return it. If no index is specified, pop( ) removes and returns the last item in the list. Consider some examples :

```
>>> t1
['k', 'a', 'e', 'i', 'p', 'q', 'u']
>>> ele1 = t1.pop(0)
>>> ele1
'k'
```

*Remove element at index 0 i.e., first element and store it in **ele1***

The removed element

```
>>> t1
['a', 'e', 'i', 'p', 'q', 'u']
>>> ele2 = t1.pop()
>>> ele2
'u'
>>> t1
['a', 'e', 'i', 'p', 'q']
```

List after removing first element

No index specified, it will remove the last element

The pop( ) method raises an exception (runtime error) if the list is already empty.

6. The remove method

The **remove( )** method removes the first occurrence of given item from the list. It is used as per following format :

```
List.remove ( <value> )
```

– *Takes one essential argument and **does not return anything***

The remove( ) will report an error if there is no such item in the list. Consider some examples :

```
>>> t1 = ['a', 'e', 'i', 'p', 'q', 'a', 'q', 'p']
>>> t1.remove('a')
>>> t1
['e', 'i', 'p', 'q', 'a', 'q', 'p']
>>> t1.remove('p')
>>> t1
['e', 'i', 'q', 'a', 'q', 'p']
```

First occurrence of 'a' is removed from the list

First occurrence of 'p' is removed from the list

7. The clear method

This method removes all the items from the list and the list becomes empty list after this function. This function returns nothing. It is used as per following format.

```
List.clear( )
```

For instance, if you have a list L1 as

```
>>> L1 = [2, 3, 4, 5]
>>> L1.clear( )
>>> L1
[ ]
```

it will remove all the items from list L1.

Now the L1 is an empty list.

Unlike **del <lstname>** statement, **clear( )** removes only the elements and not the list element. After **clear( )**, the list object still exists as an empty list.

8. The count method

This function returns the count of the item that you passed as argument. If the given item is not in the list, it returns zero.

It is used as per following format :

```
List.count(<item> )
```

For instance, for a list L1 = [13, 18, 20, 10, 18, 23]

```
>>> L1.count(18)
2
>>> L1.count(28)
0
```

returns 2 as there are two items with value 18 in the list.

No item with value 28 in the list, hence it returned 0 (zero)

9. The reverse method

The *reverse( )* reverses the items of the list. This is done **"in place"**, *i.e.*, it does not create a new list.

The syntax to use reverse method is :

```
List.reverse()
```

– Takes no argument, returns no list ; reverses the list 'in place' and does not return anything.

For example,

```
>>> t1 = ['e', 'i', 'q', 'a', 'q', 'p']
>>> t1.reverse()
>>> t1
['p', 'q', 'a', 'q', 'i', 'e']
```

The reversed list

10. The sort method

The *sort( )* function sorts the items of the list, by default in increasing order. This is done "**in place**", *i.e.*, it does not create a new list.

It is used as per following syntax :

```
List.sort()
```

For example,

```
>>> t1 = ['e', 'i', 'q', 'a', 'q', 'p']
>>> t1.sort()
>>> t1
['a', 'e', 'i', 'p', 'q', 'q']
```

Sorted list in default ascending order

Like reverse( ), sort( ) also performs its function and does not return anything.

To sort a list in decreasing order using sort( ), you can write :

```
>>> List.sort(reverse = True)
```

2.4 TUPLES IN PYTHON

The Tuples are depicted through parentheses *i.e.*, round brackets, *e.g.*, following are some tuples in Python :

```
( )                     # tuple with no member, empty tuple
(7,)                    # tuple with one member
(1, 2, 3)               # tuple of integers
(1, 2.5, 3.7, 9)        # tuple of numbers (integers and floating point)
('a', 'b', 'c')         # tuple of characters
('a', 1, 'b', 3.5, 'zero')   # tuple of mixed value types
('One', 'Two', 'Three')      # tuple of strings
```

Tuples are **immutable sequences** *i.e.*, you cannot change elements of a tuple in place.

2.4.1 Creating Tuples

To create a tuple, put a number of expressions, separated by commas in parentheses. That is, to create a tuple you can write in the form given below :

```
T = ()
T = (value, ...)
```

This construct is known as a **tuple display construct**.

Creating Empty Tuple

The empty tuple is ( ). You can also create an empty tuple as :

```
T = tuple()
```

Creating Single Element Tuple

Making a tuple with a single element is tricky because if you just give a single element in round brackets, Python considers it a value only, *e.g.*,

```
>>> t = (1)
>>> t
1
```

(1) was treated as an integer expression, hence t stores an integer 1, not a tuple

To construct a tuple with one element just add a comma after the single element as shown below :

```
>>> t = 3,
>>> t
(3,)
```

To create a one-element tuple, make sure to add comma at the end

Now t stores a tuple, not integer.

Creating Tuples from Existing Sequences

You can also use the built-in tuple type object (tuple( )) to create tuples from sequences as per the syntax given below :

```
T = tuple(<sequence>)
```

where *<sequence>* can be any kind of sequence object including *strings*, *lists* and *tuples*.

Consider following examples :

```
>>> t1 = tuple('hello')          #creating tuple from a string
>>> t1
('h', 'e', 'l', 'l', 'o')
>>> L = ['w', 'e', 'r', 't', 'y']
>>> t2 = tuple(L)                # creating tuple from a list
>>> t2
('w', 'e', 'r', 't', 'y')
```

Creating Tuple from Keyboard Input

You can use this method of creating tuples of single characters or single digits via keyboard input.

Consider the code below :

```
t1 = tuple(input('Enter tuple elements:'))
Enter tuple elements : 234567
>>> t1
('2', '3', '4', '5', '6', '7')
```

But most commonly used method to input tuples is eval(input( )) as shown below :

```
tuple = eval(input("Enter tuple to be added:"))
print("Tuple you entered :", tuple)
```

when you execute it, it will work somewhat like :

```
Enter tuple to be added: (2, 4, "a", "hjkjl, "[3, 4])
Tuple you entered : (2, 4, "a", "hjkjl", [3, 4])
```

2.4.2 Tuples *vs.* Lists

Tuples and lists are very similar yet different. This section is going to talk about the same.

2.4.2A Similarity between Tuples and Lists

Tuples are similar to lists in following ways :

- *Length* — Function len(T) returns the number of items (count) in the tuple T.
- *Indexing and Slicing*

  $T[i]$ returns the item at index i (the first item has index 0), and $T[i:j]$ returns a new tuple, containing the objects between i and j.
- *Membership operators*

  Both **'in'** and **'not in'** operators work on Tuples also. That is, **in** tells if an element is present in the tuple or not and **not in** does the opposite.
- *Concatenation and Replication operators + and **

  The + operator adds one tuple to the end of another. The * operator repeats a tuple.

Accessing Individual Elements

The individual elements of a tuple are accessed through their indexes given in square brackets. Consider the following examples :

```
>>> vowels = ('a', 'e', 'i', 'o', 'u')
>>> vowels[4]
'u'
>>> vowels[-1]
'u'
```

2.4.2B Difference between Tuples and Lists

Tuples are not mutable, while *lists* are. You cannot change individual elements of a *tuple* in place, but *lists* allow you to do so. That is, following statement is fully valid for lists (BUT not for tuples).

If we have a list L and a tuple T, then

```
L[i] = element          # is valid
```

is VALID for Lists. BUT

```
T[i] = element          # is invalid
```

is INVALID for tuples as you cannot perform item-assignment in immutable types.

2.4.3 Tuple Operations

In this section, we shall talk about most common tuple operations, briefly.

2.4.3A Traversing a Tuple

Traversing a tuple means accessing and processing each element of it. The *for loop* makes it easy to traverse or loop over the items in a tuple, as per following syntax :

```
for <item> in <Tuple>:
    process each item here
```

For example, following loop shows each item of a tuple *T* in separate lines :

```
T = ('P', 'u', 'r', 'e')
for a in T :
    print(T[a])
```

The above loop will produce result as :

```
P
u
r
e
```

2.4.3B Joining Tuples

The + operator, the concatenation operator, when used with two tuples, joins two tuples. Consider the example given below :

```
>>> tpl1 = (1, 3, 5)
>>> tpl2 = (6, 7, 8)
>>> tpl1 + tpl2
(1, 3, 5, 6, 7, 8)
```

IMPORTANT

Sometimes you need to concatenate a tuple (say *tpl*) with another tuple containing only one element. In that case, if you write statement like :

```
>>> tpl + (3)
```

Python will return an error like :

```
TypeError : can only concatenate tuple (not "int") to tuple
```

The reason for above error is : a single value in ( ) is treated as single value not as tuple. That is, expressions **(3)** and **('a')** are integer and string respectively but **(3,)** and **('a',)** are single element tuples. Thus, following expression won't give any error :

```
>>> tpl + (3, )
```

2.4.3C Repeating or Replicating Tuples

Like strings and lists, you can use * operator to replicate a tuple specified number of times, *e.g.,*

```
>>> tpl1 * 3
(1, 3, 5, 1, 3, 5, 1, 3, 5)
```

Like strings and lists, you can only use an integer with a * operator when trying to replicate a tuple.

2.4.3D Slicing the Tuples

Tuple slices, like list-slices or string slices are the sub part of the tuple extracted out. You can use indexes of tuple elements to create tuple slices as per following format :

```
seq = T[start:stop]
```

Recall that index on last limit is not included in the tuple slice. Consider the following example:

```
>>> tpl = (10, 12, 14, 20, 22, 24, 30, 32, 34)
>>> seq = tpl [ 3:-3]
>>> seq
(20, 22, 24)
```

If you want to extract, not consecutive but every other element of the tuple, there is a way out – **the slice steps**. The slice steps are used as per following format :

```
seq = T[start:stop:step]
```

Consider some examples to understand this.

```
>>> tpl
(10, 12, 14, 20, 22, 24, 30, 32, 34)
>>> tpl[0 : 10 : 2]
(10, 14, 22, 30, 34)
```

Include every 2nd element, i.e., skip 1 element in between. Check resulting tuple slice

```
>>> tpl[2 : 10 : 3]
(14, 24, 34)
```

Include every 3rd element, i.e., skip 2 element in between

```
>>> tpl[::3]
(10, 20, 30)
```

No start and stop given. Only step is given as 3. That is, from the entire tuple, pick every 3rd element for the tuple

2.4.3E Unpacking Tuples

Creating a tuple from a set of values is called ***packing*** and its reverse, *i.e.*, creating individual values from a tuple's elements is called ***unpacking***.

Unpacking is done as per syntax :

```
<variable1>, <variable2>, <variable3>, ... = t
```

where the number of variables in the left side of assignment must match the number of elements in the tuple.

For example, if we have a tuple as :

```
t = (1, 2, 'A', 'B')
```

The length of above tuple *t* is 4 as there are four elements in it. Now to unpack it, we can write :

```
w, x, y, z = t
print(w, "-", x, "-", y, "-", z)
```

The output will be :

```
1-2-A-B
```

2.4.4 Tuple Functions and Methods

1. The len( ) method

This method returns length of the tuple, *i.e.*, the count of elements in the *tuple*.

Syntax : `len(<tuple>)`

```
>>> employee = ('John', 10000, 24, 'Sales')
>>> len(employee)
4
```

The len( ) returns the count of elements in the tuple

2. The max( ) method

This method returns the element from the tuple having **maximum value**.

Syntax : `max(<tuple>)`

```
>>> tpl = (10, 12, 14, 20, 22, 24, 30, 32, 34)
>>> max(tpl)
34
```

*Maximum value from tuple **tpl** is returned*

3. The min( ) method

This method returns the element from the tuple having **minimum value**.

Syntax : `min(<tuple>)`

```
>>> tpl = (10, 12, 14, 20, 22, 24, 30, 32, 34)
>>> min(tpl)
10
```

*Maximum value from tuple **tpl** is returned*

NOTE

Like max( ), for min( ) to work, the elements of tuple should be of same type.

4. The index( ) method

It returns the index of an existing element of a tuple.

Syntax : `<tuplename> . index (<item>)`

```
>>> t1 = [3, 4, 5, 6.0]
>>> t1.index(5)
2
```

But if the given item does not exist in tuple, it raises **ValueError** exception.

5. The count( ) function

The count( ) method returns the count of a member element/object in a given sequence (list/tuple).

Syntax : `<sequence name>.count(<object>)`

```
>>> t1 = (2, 4, 2, 5, 7, 4, 8, 9, 9, 11, 7, 2)
>>> t1.count(2)
3
```

There are 3 occurrences of element 2 in given tuple, hence count() return 3 here

NOTE

With tuple( ), the argument must be a sequence type *i.e.*, a string or a list or a dictionary.

6. The tuple( ) method

This method is actually constructor method that can be used to create tuples from different types of values.

Syntax : `tuple(<sequence>)`

- *Creating empty tuple*

```
>>> tuple()
()
```

- *Creating tuple from a string*

```
>>> t = tuple("abc")
>>> t
('a', 'b', 'c')
```

- *Creating a tuple from a list*

```
>>> t = tuple([1,2,3])
>>> t
(1, 2, 3)
```

- *Creating a tuple from keys of a dictionary*

```
>>> t1 = tuple ( {1:"A", 2:"B"})
>>> t1
(1, 2)
```


This 'PriP' session is aimed at revising various concepts you learnt in Class XI.

Please check the practical component-book – Progress in Computer Science with Python and fill it there in PriP 2.1 under Chapter 2 after practically doing it on the computer.

>>>❖<<<

2.5 DICTIONARIES IN PYTHON

Python dictionaries are a collection of some *key-value* pairs. Dictionaries are mutable, unordered collections with elements in the form of a **key : value** pairs that associate keys to values.

2.5.1 Creating a Dictionary

To create a dictionary, you need to include the **key : value** pairs in curly braces as per following syntax :

```
<dictionary-name> = {<key>:<value>, <key>:<value>...}
```

Following is an example dictionary by the name teachers that stores the names of teachers as keys and the subjects being taught by them as values of respective keys.

```
teachers = { "Dimple" : "Computer Science", "Karen" : "Sociology",
             "Harpreet" : "Mathematics", "Sabah" : "Legal Studies" }
```

Notice that

- the curly brackets mark the beginning and end of the dictionary,
- each entry (*Key : Value*) consists of a pair separated by a colon – the key and corresponding value is given by writing colon (:) between them,
- the key-value pairs are separated by commas (,).

Internally, dictionaries are indexed (*i.e.*, arranged) on the basis of keys.

2.5.2 Accessing Elements of a Dictionary

In dictionaries, the elements are accessed through the *keys* defined in the **key:value** pairs, as per the syntax shown below :

```
<dictionary-name> [ <key>]
```

Thus to access the value for key defined as "Karen" in above declared teachers dictionary, you will write :

```
>>> teachers["Karen"]
```

and Python will return

```
Sociology
```

Attempting to access a key that doesn't exist causes an error. Consider the following statement that is trying to access a non-existent key (13) from dictionary teachers.

```
>>> teachers["Kushal"]
teachers KeyError : 13
```

In Python dictionaries, the elements (key : value pairs) are unordered ; one cannot access elements as per specific order.

Accessing Keys or Values Simultaneously

To see all the keys in a dictionary in one go, you may write **<dictionary>.keys( )** and to see all values in one go, you may write **<dictionary>.values( )**, as shown below :

```
>>> d = {"Vowel1" : "a", "Vowel2" : "e", "Vowel3" : "i", "Vowel4" : "o", "Vowel5" : "u"}
>>> d.keys()
['Vowel5', 'Vowel4', 'Vowel3', 'Vowel2', 'Vowel1']
>>> d.values()
['u', 'o', 'i', 'e', 'a']
```

Python lists keys in an arbitrary order.

2.5.3 Characteristics of a Dictionary

Dictionaries like lists are mutable and that is the only similarity they have with lists. Otherwise, dictionaries are different type of data structures with following characteristics :

(*a*) A dictionary is a unordered set of **key : value** pairs.

(*b*) Unlike the string, list and tuple, a dictionary is not a sequence because it is unordered set of elements.

(*c*) Dictionaries are indexed by keys and its keys must be of any non-mutable type.

(*d*) Each of the keys within a dictionary must be unique.

(*e*) Like lists, dictionaries are also mutable. We can change the value of a certain key "in place" using the assignment statement as per syntax :

```
<dictionary>[<key>] = <value>
```

2.5.4 Dictionary Operations

In this section, we shall briefly talk about various operations possible on Python dictionaries.

2.5.4A Traversing a Dictionary

Traversal of a collection means accessing and processing each element of it. The *for loop* makes it easy to traverse or loop over the items in a dictionary, as per following syntax :

```
for <item> in <Dictionary> :
    process each item here
```

Consider following example that will illustrate this process. A dictionary namely *d1* is defined with three keys – a number, a string, a tuple of integers.

```
d1 = { 5 : "number", \
    "a" : "string", \
    (1,2) : "tuple" }
```

To traverse the above dictionary, you can write for loop as :

```
for key in d1 :
    print(key, ":", d1[key])
```

The above loop will produce the output as shown below :

```
a : string
(1, 2) : tuple
5 : number
```

2.5.4B Adding Elements to Dictionary

You can add new elements (**key : value** pair) to a dictionary using assignment as per the following syntax. BUT the *key* being added must not exist in dictionary and must be unique. If the *key* already exists, then this statement will change the value of existing *key* and no new entry will be added to dictionary.

```
<dictionary>[<key>] = <value>
```

Consider the following example :

```
>>> Employee = {'name' : 'John', 'salary' : 10000, 'age' : 24}
>>> Employee['dept'] = 'Sales'
>>> Employee
{'salary' : 10000, 'dept' : 'Sales', 'age' : 24, 'name' : 'John'}
```

Using this method, you can create new dictionary, if you are adding elements to an empty dictionary.

2.5.4C Updating Existing Elements in a Dictionary

Updating an element is similar to what we did just now. That is, you can change value of an existing key using assignment as per following syntax :

```
dictionary>[<key>] = <value>
```

Consider the following example :

```
>>> Employee = {'name' : 'John' , 'salary' : 10000, 'age' : 24}
>>> Employee['salary'] = 20000
>>> Employee
{'salary' : 20000, 'age' : 24, 'name' : 'John'}
```

But make sure that the *key* must exist in the dictionary, otherwise new entry will be added to the dictionary.

Using this technique of adding **key : value** pairs, you can create dictionaries interactively at runtime by accepting input from user.

Program 2.2 *Write a program to create a dictionary containing names of competition winner students as keys and number of their wins as values.*

```
n = int(input("How many students ?"))
CompWinners = { }
for a in range(n) :
    key = input("Name of the student :")
    value = int(input("Number of competitions won :"))
    CompWinners[key] = value
print("The dictionary now is :")
print(CompWinners)
```

```
How many students ? 5
Name of the student : Naval
Number of competitions won : 5
Name of the student : Zainab
Number of competitions won : 3
Name of the student : Nita
Number of competitions won : 3
Name of the student : Rosy
Number of competitions won : 1
Name of the student : Jamshed
Number of competitions won : 5
The dictionary now is :
{'Nita' : 3, 'Naval' : 5, 'Zainab' : 3, 'Rosy' : 1, 'Jamshed' : 5}
```

2.5.4D Deleting Elements from a Dictionary

There are two methods for deleting elements from a dictionary.

(*i*) To delete a dictionary element or a dictionary entry, *i.e.*, a **key:value pair**, you can use **del** command. The syntax for doing so is as given below :

```
del <dictionary>[ <key>]
```

Consider the following example :

```
>>> empl3
{'salary' : 10000, 'age' : 24, 'name' : 'John'}
>>> del empl3['age']
>>> empl3
{'salary' : 10000, 'name' : 'John'}
```

But with **del** statement, the key that you are giving to delete must exist in the dictionary, otherwise Python will return an error. See below :

```
>>> del empl3['new']
del empl3['new']
KeyError : 'new'
```

(*ii*) Another method to delete elements from a dictionary is by using pop( ) method as per following syntax :

```
<dictionary>.pop(<key>)
```

The pop( ) method will not only delete the **key:value pair** for mentioned *key* but also return the corresponding value.

Consider the following code example

```
>>> employee
{'salary' : 10000, 'age' : 24, 'name' : 'John'}
>>> employee.pop('age')
24
>>> employee
{'salary' : 10000, 'name' : 'John'}
```

If you try to delete a *key* which does not exist, the Python returns error. See below :

```
>>> employee.pop('new')
employee.pop('new')
KeyError : 'new'
```

However, pop( ) method allows you to specify what to display when the given *key* does not exist, as per following syntax :

```
<dictionary>.pop(<key>, <in-case-of-error-show-me>)
```

For example :

```
>>> employee.pop('new', "Not Found")
'Not Found'
```

2.5.4E Checking for Existence of a Key

Usual membership operators in and not in work with dictionaries as well. But they can check for the existence of keys only. You may use them as per syntax given below :

```
<key> in <dictionary>
<key> not in <dictionary>
```

- The **in** operator will return *True* if the given **key** is present in the dictionary, otherwise *False*.
- The **not in** operator will return *True* if the given **key** is not present in the dictionary, otherwise *False*.

Consider the following examples :

```
>>> empl = {'salary' : 10000, 'age' : 24, 'name' : 'John'}
>>>'age' in empl
True
>>>'John' in empl
False
>>>'John' not in empl
True
>>>'age' not in empl
False
```

2.5.5 Dictionary Functions and Methods

1. The len( ) method

This method returns length of the *dictionary* , *i.e.,* the count of elements (*key:value* pairs) in the *dictionary*. The syntax to use this method is given below :

```
len(<dictionary>)
```

e.g.,

```
>>> employee = {'name' : 'John', 'salary' : 10000, 'age' : 24}
>>> len(employee)
3
```

2. The clear( ) method

This method removes all items from the *dictionary* and the dictionary becomes empty dictionary post this method.

```
<dictionary>.clear()
```

e.g.,

```
>>> Employee = {'name' : 'John', 'salary' : 10000, 'age' : 24}
>>> Employee.clear()
>>> Employee
{ }
```

See, now the dictionary is empty

NOTE

The clear( ) removes all the elements of a dictionary and makes it empty dictionary while **del** statement removes the complete dictionary as an object. After **del** statement with a dictionary name, that dictionary object no more exists, not even empty dictionary.

3. The get( ) method

With this method, you can get the item with the given key, similar to **dictionary [key]**. If the **key** is not present, Python will give error.

```
<dictionary>.get( key , [ default ])
```

e.g.,

```
>>> empl1
{'salary' : 10000, 'dept' : 'Sales', 'age': 24, 'name' : 'John'}
>>> empl1.get('dept')
'Sales'
```

4. The items( ) method

This method returns all of the items in the *dictionary* as a sequence of (key, value) tuples. Note that these are returned in no particular order.

```
<dictionary>.items()
```

e.g.,

```
employee = {'name' : 'John', 'salary' : 10000, 'age' : 24}
myList = employee.items()
for x in myList :
    print(x)
```

The adjacent code gives output as:

```
('salary', 10000)
('age', 24)
('name', 'John')
```

5. The keys( ) method

This method returns all of the *keys* in the *dictionary* as a sequence of *keys* (in form of a list). Note that these are returned in no particular order.

```
<dictionary>.keys()
```

e.g.,

```
>>> employee
{'salary' : 10000, 'dept' : 'Sales', 'age' : 24, 'name' : 'John'}
>>> employee.keys()
['salary', 'dept', 'age', 'name']
```

6. The values( ) method

This method returns all the values from the *dictionary* as a sequence (a list). Note that these are returned in no particular order.

```
<dictionary>.values()
```

e.g.,

```
>>> employee
{'salary' : 10000, 'dept' : 'Sales', 'age' : 24, 'name' : 'John'}
>>> employee.values()
[10000, 'Sales', 24, 'John']
```

7. The update( ) method

This method merges *key : value* pairs from the new *dictionary* into the original *dictionary*, adding or replacing as needed. The items in the new dictionary are added to the old one and override any items already there with the same keys.

The syntax to use this method is given below :

Dictionary to be updated → `<dictionary>.update ( <other-dictionary>)` ← *This dictionary's items will be taken for updating other dictionary.*

e.g.,

See, the elements of dictionary ***employee2*** *have overridden the elements of dictionary* ***employee1*** *having the same keys i e., of keys* ***'name'*** *and "salary"*

```
>>> employee1 = {'name' : 'John', 'salary' : 10000, 'age' : 24}
>>> employee2 = {'name' : 'Diya' , 'salary' : 54000, 'dept' : 'Sales'}
>>> employee1.update(employee2)
>>> employee1
{'salary' : 54000, 'dept' : 'Sales', 'name' : 'Diya', 'age' : 24}
>>> employee2
{'salary' : 54000, 'dept' : 'Sales', 'name' : 'Diya'}
```

Program 2.3 *Given three lists as list1 = ['a','b','c'] , list2 = ['h','i','t'] and list3 = ['0','1','2']. Write a program that adds* ***lists 2 and 3 to list1*** *as single element each. The resultant list should be in the order of* ***list3, elements of list1, list2.***

```
list1 =['a','b','c']
list2 = ['h','i','t']
list3 = ['0','1', '2']
print("Originally :")
print("List1 = ", list1)
print("List2 = ", list2)
print("List3 = ", list3)

# adding list2 as single element at the end of list1
list1.append(list2)        #list2 gets added as one element at the end of list1
```

```
# adding list3 as single element at the start of list1
list1.insert(0,list3) #list3 gets added as one element at the beginning of list1
print("After adding two lists as individual elements, list now is :")
print(list1)
```

The output produced by above program is as follows :

```
Originally :
List1 = ['a', 'b', 'c']
List2 = ['h', 'i', 't']
List3 = ['0', '1', '2']
After adding two lists as individual elements, list now is :
[['0', '1', '2'], 'a', 'b', 'c', ['h', 'i', 't']]
```

Program 2.4 *Given three lists as list1 = ['a','b','c'] , list2 = ['h','i','t'] and list3 = ['0','1','2']. Write a program that adds individual elements of lists 2 and 3 to list1. The resultant list should be in the order of elements of list3, elements of list1, elements of list2.*

```
list1 =['a','b','c']
list2 = ['h','i','t']
list3 = ['0','1', '2']
print("Originally :")
print("List1 = ", list1)
print("List2 = ", list2)
print("List3 = ", list3)

# adding elements of list1 at the end of list3
list3.extend(list1)

# adding elements of list2 at the end of list3
list3.extend(list2)
print("After adding elements of two lists individually, list now is :")
print(list3)
```

The output produced by above program is as follows :

```
Originally :
List1 = ['a', 'b', 'c']
List2 = ['h', 'i', 'y']
List3 = ['0', '1', '2']
After adding elements of two lists individually, list now is :
['0', '1', '2', 'a', 'b', 'c', 'h', 'i', 't']
```

Program 2.5 Write a program that finds an element's index/position in a tuple WITHOUT using index().

```
tuple1 = ('a','p','p','l','e',)
char = input("Enter a single letter without quotes : ")
```

```
        if char in tuple1:
            count = 0
            for a in tuple1:
                if a != char:
                    count += 1
                else:
                    break
            print(char, "is at index", count, "in", tuple1)
        else:
            print(char, "is NOT in", tuple1)
```

```
Enter a single letter without quotes : l
l is at position 3 in ('a', 'p', 'p', 'l', 'e')
================================================
Enter a single letter : p
p is at position 1 in ('a', 'p', 'p', 'l', 'e')
```

Program 2.6 *Write a program that checks for presence of a value inside a dictionary and prints its key.*

```
info = {'Riya':'CSc.', 'Mark':'Eco', 'Ishpreet':'Eng', 'Kamaal':'Env.Sc'}
inp = input("Enter value to be searched for :")
if inp in info.values():
    for a in info:
        if info[a] == inp :
            print("The key of given value is", a)
            break
    else:
        print("Given value does not exist in dictionary")
```

```
Enter value to be searched for : Env.Sc
The key of given value is Kamaal
=================================
Enter value to be searched for : eng
Given value does not exist in dictionary
```

Program 2.7 *The code of previous will not work if the cases of the given value and value inside dictionary are different. That is, the result (of previous program) will be like :*

```
Enter value to be searched for : eng
Given value does not exist in dictionary
```

Make changes in above program so that the program returns the key, even if the cases differ, i.e., match the two values ignoring their cases

```
info = {'Riya':'CSc.', 'Mark':'Eco', 'Ishpreet':'Eng', 'Kamaal':'Env.Sc'}
inp = input("Enter value to be searched for :")
```

```
for a in info:
    if info[a].upper() == inp.upper() :
        print("The key of given value is", a)
        break
else:
    print("Given value does not exist in dictionary")
```

The sample run of above program is as given below :

```
Enter value to be searched for : eng
The key of given value is Ishpreet
```

2.6 SORTING TECHNIQUES

SORTING

Sorting, in computer terms, refers to arranging elements in a specific order — ascending or descending.

Sorting in computer terms means arranging elements in a specific order — ascending or increasing order or descending or decreasing order.

There are multiple ways or techniques or algorithms that you can apply to sort a group of elements such as *selection sort, insertion sort, bubble sort, heap sort, quick sort* etc.

We shall cover *Bubble sort* and *insertion sort*, as recommended by syllabus.

2.6.1 Bubble Sort

The basic idea of bubble sort is to compare two adjoining values and exchange them if they are not in proper order. To understand this, have a look at figure 2.3 that visually explains the process of *bubble sort*.

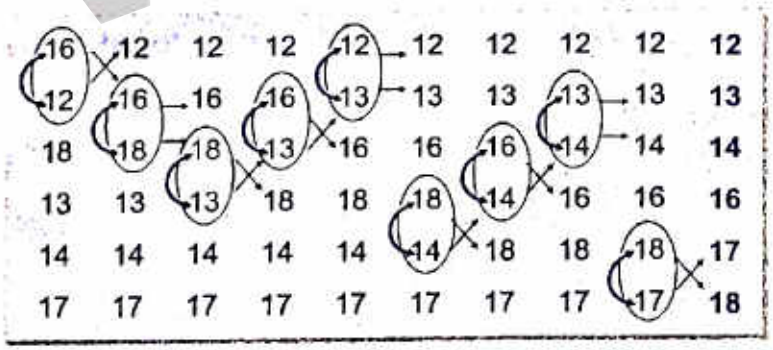


Figure 2.3

Program 2.8 *Program to sort a list using Bubble sort.*

```
aList = [ 15, 6, 13, 22, 3, 52, 2]
print ("Original list is :", aList)
n = len(aList)
# Traverse through all list elements
for i in range(n):
    # Last i elements are already in place
```

We have taken a pre-initialized list. You can even input a list prior to sorting it.

```
    for j in range(0, n-i-1):
        # traverse the list from 0 to n-i-1
        # Swap if the element found is greater
        # than the next element
        if aList[j] > aList[j + 1] :
            aList[j], aList[j + 1] = aList[j + 1], aList[j]
print ("List after sorting :", aList)
```

This expression will ensure that we do not compare the heavier elements that have already settled at correct position.

The output produced by above code is like :

```
Original list is : [15, 6, 13, 22, 3, 52, 2]
List after sorting : [2, 3, 6, 13, 15, 22, 52]
```

INSERTION SORT

Insertion sort is a sorting algorithm that builds a sorted list one element at a time from the unsorted list by inserting the element at its correct position in sorted list.

2.6.2 Insertion Sort

Insertion sort is a sorting algorithm that builds a sorted list one element at a time from the unsorted list by inserting the element at its correct position in sorted list.

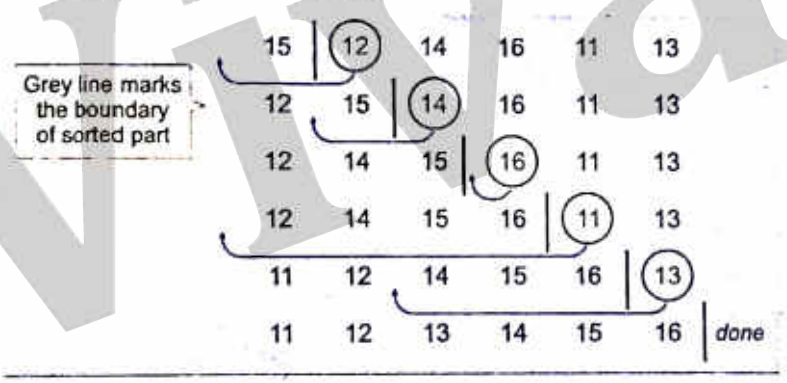


Figure 2.4

Program 2.9 Program to sort a sequence using insertion sort.

```
aList = [15, 6, 13, 22, 3, 52, 2]
print ("Original list is :", aList)
for i in range(1, len(aList)) :
    key = aList[i]
    j = i - 1
    while j >= 0 and key < aList[j] :
        aList[j + 1] = aList[j]    #shift elements to right to make room for key
        j = j - 1
    else:
        aList[j + 1] = key
print("List after sorting :", aList)
```

```
Original list is : [15, 6, 13, 22, 3, 52, 2]
List after sorting : [2, 3, 6, 13, 15, 22, 52]
```

This 'PriP' session is aimed at revising various concepts you learnt in Class XI.

Please check the practical component-book – Progress in Computer Science with Python and fill it there in PriP 2.2 under Chapter 2 after practically doing it on the computer.

>>>❖<<<

LET US REVISE

- *Python strings are stored in memory by storing individual characters in contiguous memory locations.*
- *The index (also called subscript sometimes) is the numbered position of a letter in the string.*
- *In Python, indices begin **0** onwards in the forward direction up to **length-1** and **1, -2, ...** up to – **length** in the backward direction. This is called **two-way indexing**.*
- *The string slice refers to a part of the string **s[start:end]** is the element beginning at start and extending up to but not including end.*
- *Lists are mutable sequences of Python i.e., you can change elements of a list in place.*
- *Lists index their elements just like strings, i.e., two way indexing.*
- *Lists are similar to strings in many ways like **indexing**, **slicing** and **accessing individual elements** but they are different in the sense that Lists are mutable while strings are not.*
- *Membership operator **in** tells if an element is present in the sequence or not and **not in** does the opposite.*
- *List slice is an extracted part of a list; list slice is a list in itself.*
- ***L[start:stop]** creates a list slice out of list L with elements falling between indexes start and stop , not including stop.*
- *Tuples are immutable sequences of Python i.e., you cannot change elements of a tuple in place.*
- *To create a tuple, put a number of comma-separated expressions in round brackets. The empty round brackets i.e., ( ) indicate an empty tuple.*
- *Tuples index their elements just like strings or lists, i.e., **two way indexing**.*
- *Tuples are stored in memory exactly like strings, except that because some of their objects are larger than others, they store a reference at each index instead of single character as in strings.*
- *Tuple slice is an extracted part of tuple; tuple slice is a tuple in itself.*
- *T[start:stop] creates a tuple slice out of tuple T with elements falling between indexes start and stop, not including stop.*
- *Dictionaries are mutable with elements in the form of a **key:value** pair that associate keys to values.*
- *The keys of a dictionary must be of immutable types.*
- *In Python dictionaries, the elements (key:value pairs) are unordered ; one cannot access element as per specific order.*

- *Keys of a dictionary must be unique.*
- *In Dictionaries, the updation and addition of elements are similar in syntax. But for addition, the key must not exist in the dictionary and for updation, the key must exist in the dictionary.*
- *Sorting of an array means arranging the array elements in a specified order.*
- *In bubble sort, the adjoining values are compared and exchanged if they are not in proper order. This process is repeated until the entire array is sorted.*
- *In insertion sort, each successive element is picked & inserted at an appropriate position in the previously sorted array.*

Solved Problems

1. *What is indexing in context to Python strings ? Why is it also called two-way indexing ?*

   Solution. In Python strings, each individual character is given a location number, called index and this process is called indexing.

   Python allocates indices in two directions :

   - in forward direction, the indexes are numbered as 0, 1, 2,.... length-1.
   - in backward direction, the indexes are numbered as –1, –2, –3.... length.

   This is known as two-way indexing.

2. *What is a string slice ? How is it useful ?*

   Solution. A sub-part or a slice of a string, say **s**, can be obtained using **s [n : m]** where *n* and *m* are integers. Python returns all the characters at indices *n*, *n*+1, *n*+2...*m*–1 *e.g.*,

   ```
   'Well done'[1 : 4]    will give    'ell'
   ```

3. *Figure out the problem with following code fragment. Correct the code and then print the output.*

   ```
   1.  s1 = 'must'
   2.  s2 = 'try'
   3.  n1 = 10
   4.  n2 = 3
   5.  print(s1 + s2)
   6.  print(s2 * n2)
   7.  print(s1 + n1)
   8.  print(s2 * s1)
   ```

   Solution. The problem is with lines 7 and 8.

   - **Line 7 – print(s1 + n1)** will cause error because **s1** being a string cannot be concatenated with a number **n1**.

     This problem can be solved either by changing the operator or operand *e.g.*, all the following statements will work :

     (*a*) `print (s1 * n1)`

     (*b*) `print (s1 + str(n1))`

     (*c*) `print (s1 + s2)`

- **Line 8** – print(s2 * s1) will cause error because two strings cannot be used for replication. The corrected statement will be :

```
print(s2 + s1)
```

If we replace the **Line 7** with its suggested solution (*b*), the output will be :

```
must try
try try try
must 10
try must
```

4. *Consider the following code :*

```
string = input( "Enter a string :" )
count = 3
while True :
    if string[0] == 'a' :
        string = string[2 :]
    elif string[-1] == 'b' :
        string = string [ : 2]
    else :
        count += 1
        break
print(string)
print(count)
```

What will be the output produced, if the input is : (i) aabbcc (ii) aaccbb (iii) abcc

Solution.

(*a*) *bbcc* (*b*) *cc* (*c*) *cc*
4 4 4

5. *Consider the following code :*

```
Inp = input("Please enter a string :" )
while len(Inp) <= 4 :
    if Inp[-1] == 'z' :                    #condition 1
        Inp = Inp [0 : 3] + 'c'
    elif 'a' in Inp :                      #condition 2
        Inp = Inp[0] + 'bb'
    elif not int(Inp[0]) :                 #condition 3
        Inp = '1' + Inp[1 :] + 'z'
    else :
        Inp = Inp + '*'
print(Inp)
```

What will be the output produced if the input is (i) 1bzz, (ii) '1a' (iii) 'abc' (iv) '0xy', (v) 'xyz'.

Solution.

(*i*) *1bzc** (*ii*) *1bb***

(*iii*) endless loop because '*a*' will always remain at index 0 and condition 3 will be repeated endlessly.

(*iv*) *1× yc** (*v*) Raises an error as *Inp[0]* cannot be converted to int.

6. *Write a program that takes a string with multiple words and then capitalizes the first letter of each word and forms a new string out of it.*

Solution.

```
string = input( "Enter a string :" )
length = len(string)
a = 0
end = length
string2 =''        #empty string
while a < length :
    if a == 0 :
        string2 += string[0].upper()
        a += 1
    elif (string[a] ==' ' and string[a+1] !=' ') :
        string2 += string[a]
        string2 += string[a+1].upper()
        a +=2
    else :
        string2 += string[a]
        a += 1
print("Original String :", string)
print("Captilized words String", string2)
```

7. *Write a program that reads a string and checks whether it is a palindrone string or not.*

Solution.

```
string = input("Enter a string :")
length = len(string)
mid = length/2
rev = -1
for a in range(mid) :
    if string[a] == string[rev] :
        a += 1
        rev -= 1
    else :
        print(string, "is not a palindrome")
        break
else :        #loop else
    print(string, "is a palindrome")
```

8. *How are lists different from strings when both are sequences ?*

Solution. The lists and strings are different in following ways :

(*i*) The lists are mutable sequences while strings are immutable.

(*ii*) In consecutive locations, a string stores the individual characters while a list stores the references of its elements.

(*iii*) Strings store single type of elements – all characters while lists can store elements belonging to different types.

9. *What are nested lists ?*

Solution. When a list is contained in another list as a member-element, it is called nested list, *e.g.*,

$a = [2, 3, [4, 5]]$

The above list a has three elements – an integer 2, an integer 3 and a list [4, 5], hence it is nested list.

10. *What does each of the following expressions evaluate to?*

Solution. Suppose that *L* is the list

["These", ["are", "a"], ["few", "words"], "that", "we", "will", "use"].

(*a*) `L[3:4] + L[1:2]`

(*b*) `"few" in L[2:3]`

(*c*) `"few" in L[2]`

(*d*) `L[2][1:]`

(*e*) `L[1] + L[2]`

Solution.

(*a*)
```
L[3:4] = ['that']
L[1:2] = [['are', 'a']]
L[3:4] + L[1:2] = ['that', ['are', 'a']]
```

(*b*) **False.** The string "few" is not an element of this range. *L*[2 : 3] returns a list of elements from *L*–> [['few', 'words']], this is a list with one element, a list.

(*c*) **True.** *L*[2] returns the list ['few', 'words'], "few" is an element of this list.

(*d*)
```
L[2] = ['few', 'words']
L[2][1:] = ['words']
```

(*e*)
```
L[1] = ['are', 'a']
L[2] = ['few', 'words']
L[1] + L[2] = ['are', 'a', 'few', 'words']
```

11. *What is the output produced by the following code snippet ?*

```
aLst =[1,2,3,4,5,6,7,8,9]
print(aLst[::3])
```

Solution.

```
[1,4,7]
```

12. *What will be the output of the following code snippet ?*

```
Lst =[1,2,3,4,5,6,7,8,9]
Lst[::2]=10,20,30,40,50,60
print(Lst)
```

(*a*) *ValueError* : attempt to assign sequence of size 6 to extended slice of size 5

(*b*) [10, 2, 20, 4, 30, 6, 40, 8, 50, 60]

(*c*) [1, 2, 10, 20, 30, 40, 50, 60]

(*d*) [1, 10, 3, 20, 5, 30, 7, 40, 9, 50, 60]

Solution. (*a*)

13. *What will be the output of the following code snippet ?*

```
values = []
for i in range (1,4):
    values.append(i)
    print (values)
```

Solution.

```
[1]
[1, 2]
[1, 2, 3]
```

14. *What will be the output of the following code ?*

```
rec = {"Name" : "Python", "Age":"20"}
r = rec.copy()
print(id(r) == id(rec))
```

(*a*) True (*b*) False (*c*) 0 (*d*) 1

Solution. (*b*)

15. *What will be the output of the following code snippet?*

```
dc1 = {}
dc1[1] = 1
dc1['1'] = 2
dc1[1.0] = 4

sum = 0
for k in dc1:
    sum += dc1[k]
print (sum)
```

Solution. 6

16. *Predict the output of following code fragment :*

```
fruit = {}
fl = ['Apple','Banana', 'apple', 'Banana']
for index in fl :
    if index in fruit:
        fruit[index] += 1
    else:
        fruit[index] = 1
    print(fruit)
print (len(fruit))
```

Solution.

```
{'Apple': 1}
{'Apple': 1, 'Banana': 1}
{'Apple': 1, 'Banana': 1, 'apple': 1}
{'Apple': 1, 'Banana': 2, 'apple': 1}
3
```

17. *Find the error in following code. State the reason of the error.*

```
aLst = {'a':1,'b':2,'c':3}
print (aLst['a','b'])
```

Solution.

The above code will produce **KeyError**, the reason being that there is no key same as the list ['a', 'b'] in dictionary **aLst**.

It seems that the above code intends to print the values of two keys 'a' and 'b', thus we can modify the above code to perform this as :

```
aLst = {'a':1,'b':2,'c':3}
print (aLst['a'], aLst['b'])
```

Now it will give the result as :

```
1 2
```

18. *Find the error in the following code fragment. State the reason behind the error.*

```
box = {}
jars = {}
crates = {}
box['biscuit'] = 1
box['cake'] = 3
jars['jam'] = 4
crates['box'] = box
crates['jars'] = jars
print (crates[box])
```

Solution. The above code will produce error with **print( )** because it is trying to print the value from dictionary **crates** by specify a key which is of a mutable type dictionary(**box** is a dictionary). There can never be a key of mutable type in a dictionary, hence the error.

The above code can be corrected by changing the **print( )** as :

```
print (crates['box'])
```

19. *Write the most appropriate list method to perform the following tasks.*

(*a*) *Delete a given element from the list.* (*b*) *Delete 3rd element from the list.*

(*c*) *Add an element in the end of the list.*

(*d*) *Add an element in the beginning of the list.*

(*e*) *Add elements of a list in the end of a list.*

Solution. (*a*) remove( ) (*b*) pop( ) (*c*) append( ) (*d*) insert( ) (*e*) extend( )

20. *How are tuples different from lists when both are sequences ?*

Solution. The tuples and lists are different in following ways :

- The tuples are immutable sequences while lists are mutable.
- Lists can grow or shrink while tuples cannot.

21. *How can you say that a tuple is an ordered list of objects ?*

Solution. A tuple is an ordered list of objects. This is evidenced by the fact that the objects can be accessed through the use of an ordinal index and for a given index, same element is returned everytime.

22. *Following code is trying to create a tuple with a single item. But when we try to obtain the length of the tuple is, Python gives error. Why? What is the solution ?*

```
>>> t = (6)
>>> len(t)
Traceback (most recent call last):
  File "<pyshell#8>", line 1, in <module>
    len(t)
TypeError: object of type 'int' has no len()
```

Solution. The syntax for a tuple with a single item requires the item to be followed by a comma as shown below :

```
t = ("a",)
```

Thus, above code is not creating a tuple in t but an integer, on which *len( )* cannot be applied. To create a tuple in t with single element, the code should be modified as :

```
>>> t = (6,)
>>> len(t)
```

23. *What is the length of the tuple shown below ?*

```
t = ((('a', 1), 'b', 'c'), 'd', 2), 'e', 3)
```

Solution. The length of this tuple is 3 because there are just three elements in the given tuple. Because a careful look at the given tuple yields that tuple *t* is made up of :

```
t1 = "a", 1
t2 = t1, "b", "c"
t3 = t2, "d", 2
t = ( t3,"e", 3)
```

24. *Can tuples be nested ?*

Solution. Tuples can contain other compound objects, including lists, dictionaries, and other tuples. Hence, tuples can be nested.

25. *How are dictionaries different from lists ?*

Solution. The dictionary is similar to lists in the sense that it is also a collection of data-items just like lists BUT it is different from lists in the sense that lists are *sequential collections* (ordered) and dictionaries are *non-sequential collections* (unordered).

In lists, where there is an order associated with the data-items because they act as storage units for other objects or variables you've created. Dictionaries are different from lists and tuples because the group of objects they hold aren't in any particular order, but rather each object has its own unique name, commonly known as a key.

26. *How are objects stored in lists and dictionaries different ?*

Solution. The objects or values stored in a dictionary can basically be anything (even the nothing type defined as None), but keys can only be immutable type-objects. *e.g.*, strings, tuples, integers, etc.

27. *When are dictionaries more useful than lists ?*

Solution. Dictionaries can be much more useful than lists. *For example*, suppose we wanted to store all our friends' cell-phone numbers. We could create a list of pairs (name of friend, phone number), but once this list becomes long enough searching this list for a specific phone number will get time-consuming. Better would be if we could index the list by our friend's name. This is precisely what a dictionary does.

28. *Can sequence operations such as slicing and concatenation be applied to dictionaries ? Why ?*

Solution. No, sequence operations like slicing and concatenation cannot be applied on dictionaries. The reason being, a dictionary is not a sequence. Because it is not maintained in any specific order, operations that depend on a specific order cannot be used.

28. *Why can't Lists be used as keys ?*

Solution. Lists cannot be used as keys in a dictionary because they are mutable. And a Python dictionary can have only keys of immutable types.

30. *If the addition of a new key:value pair causes the size of the dictionary to grow beyond its original size, an error occurs. True or false ?*

Solution. **False.** There cannot occur an error because Dictionaries being the mutable types, they can grow or shrink on an as-needed basis.

31. *Consider a dictionary* **my_points** *with single-letter keys, each followed by a 2-element tuple representing the coordinates of a point in an x-y coordinate plane.*

```
my_points = { 'a' : (4, 3), 'b' : (1, 2), 'c' : (5, 1) }
```

Write a program to calculate the maximum value from within all of the values tuples at same index.

For example, maximum for 0th index will be computed from values 4, 1 and 5 – all the entries at 0th index in the value-tuple.

Print the result in following format :

```
Maximum Value at index(my_points, 0) = 5
Maximum Value at index(my_points, 1) = 3
```

Solution.

```
my_points = { 'a' : (4, 3), 'b' : (1, 2), 'c' : (5, 1) }
highest = [0, 0]
init = 0
for a in range(2) :
    init = 0
    for b in my_points.keys():
        val = my_points[b][a]

        if init == 0 :
            highest[a] = val
        init += 1

        if val > highest[a] :
            highest[a] = val
    print("Maximum Value at index(my_points, ", a, ") = ", highest[a])
```

GLOSSARY

Dictionary	A mutable, unordered collection with elements in the form of a key:value pairs that associate keys to value.
Index	An integer variable that is used to identify the position of an element and access the element.
List	A mutable sequence of Python that can store objects of any type.
Lookup	A dictionary operation that takes a key and finds the corresponding value.

Mapping	Linking of a key with a value through some internal function (hash function).
Nesting	Having an element of similar type inside another element.
Packing	Creating a tuple from individual values.
Tuple	An immutable sequence of elements.
Unpacking	Creating individual values from a tuple's elements.
Sorting	Arranging elements of a sequence in some order (ascending/descending).

Assignment

Type A : Short Answer Questions/Conceptual Questions

1. What is the internal structure of Python strings ?
2. Write a Python script that traverses through an input string and prints its characters in different lines – two characters per line.
3. Discuss the utility and significance of Lists, briefly.
4. What do you understand by mutability ? What does "in place" task mean ?
5. Start with the list [8, 9, 10]. Do the following :
   (*a*) Set the second entry (index 1) to 17 (*b*) Add 4, 5 and 6 to the end of the list
   (*c*) Remove the first entry from the list (*d*) Sort the list
   (*e*) Double the list (*f*) Insert 25 at index 3
6. What's *a* [1 : 1] if a is a string of at least two characters ? And what if string is shorter ?
7. What are the two ways to add something to a list ? How are they different ?
8. What are the two ways to remove something from a list? How are they different ?
9. What is the difference between a list and a tuple ?
10. In the Python shell, do the following :
    (*i*) Define a variable named states that is an empty list. (*ii*) Add 'Delhi' to the list.
    (*iii*) Now add 'Punjab' to the end of the list.
    (*iv*) Define a variable *states2* that is initialized with 'Rajasthan', 'Gujrat', and 'Kerala'.
    (*v*) Add 'Odisha' to the beginning of the list.
    (*vi*) Add 'Tripura' so that it is the third state in the list.
    (*vii*) Add 'Haryana' to the list so that it appears before 'Gujrat'. Do this as if you DO NOT KNOW where 'Gujrat' is in the list.
    Hint. See what states2.index("Rajasthan") does. What can you conclude about what listname.index(item) does ?
    (*viii*) Remove the 5th state from the list and print that state's name.
11. Discuss the utility and significance of Tuples, briefly.
12. If a is (1, 2, 3)
    (*a*) what is the difference (if any) between a * 3 and (a, a, a) ?
    (*b*) is a * 3 equivalent to a + a + a ? (*c*) what is the meaning of a[1:1] ?
    (*d*) what's the difference between a[1:2] and a[1:1] ?
13. What is the difference between (30) and (30,) ?
14. Why is a dictionary termed as an unordered collection of objects ?

15. What type of objects can be used as keys in dictionaries ?
16. Though tuples are immutable type, yet they cannot always be used as keys in a dictionary. What is the condition to use tuples as a key in a dictionary ?
17. Dictionary is a mutable type, which means you can modify its contents ? What all is modifiable in a dictionary ? Can you modify the keys of a dictionary ?
18. How is del D and del D[<key>] different from one another if *D* is a dictionary ?
19. Create a dictionary named D with three entries, for keys 'a', 'b' and 'c'. What happens if you try to index a nonexistent key (D['d']) ? What does Python do if you try to assign to a nonexistent key d (*e.g.*, D['d']='spam') ?
20. What is sorting ? Name some popular sorting techniques.
21. Discuss Bubble sort and Insertion sort techniques.

Type B : Application Based Questions

1. What will be the output produced by following code fragments ?

(*a*)
```
y = str(123)
x = "hello" * 3
print(x, y)
x = "hello" + "world"
y = len(x)
print(y, x)
```

(*b*)
```
x = "hello" +
"to Python" +
"world"
for char in x :
    y = char
    print(y, ':',)
```

(*c*)
```
x = "hello world"
print(x[:2], x[:-2], x[-2:])
print(x[6], x[2:4])
print(x[2:-3], x[-4:-2])
```

2. Write a short Python code segment that adds up the lengths of all the words in a list and then prints the average (mean) length. Use the final list from previous question to test your program.
3. Predict the output of the following code snippet ?
```
a = [1,2,3,4,5]
print(a[3:0:-1])
```
4. Predict the output of the following code snippet?

(*a*)
```
arr = [1, 2, 3, 4, 5, 6]
for i in range(1, 6):
    arr[i - 1] = arr[i]
for i in range(0, 6):
    print(arr[i], end = " ")
```

(*b*)
```
Numbers = [9, 18, 27, 36]
for Num in Numbers :
    for N in range(1, Num%8) :
        print(N, "#", end = " ")
    print( )
```
[CBSE D 2016]

5. Find the errors. State reasons.

(*a*)
```
t = (1, "a", 9.2)
t[0] = 6
```

(*b*)
```
t = [1, "a", 9.2]
t[0] = 6
```

(*c*)
```
t = [1, "a", 9.2]
t[4] = 6
```

(*d*)
```
t = 'hello'
t[0] = "H"
```

(*e*)
```
for Name in [Amar, Shveta, Parag]
    IF Name[0] = 'S' :
        print(Name)
```
[CBSE D 2016]

6. Assuming words is a valid list of words, the program below tries to print the list in reverse. Does it have an error ? If so, why ? (Hint. There are two problems with the code.)
```
for i in range(len(words), 0, -1):
    print(words[i], end = ' ')
```

7. What would be the output of following code if ntpl = ("Hello", "Nita", "How's", "life ?") ?

```
(a, b, c, d) = ntpl
print("a is:", a)
print("b is:", b)
print("c is:", c)
print("d is:", d)
ntpl= (a, b, c, d)
print(ntpl[0][0]+ntpl[1][1], ntpl[1])
```

8. What will be the output of the following code ?

```
tuple_a = 'a', 'b'
tuple_b = ('a', 'b')
print (tuple_a == tuple_b)
```

(a) 0 (b) 1 (c) False (d) True

9. What will be the output of the following code snippet ?

```
rec = {"Name" : "Python", "Age":"20", "Addr" : "NJ", "Country" : "USA"}
id1 = id(rec)
del rec
rec = {"Name" : "Python", "Age":"20", "Addr" :"NJ", "Country" : "USA"}
id2 = id(rec)
print(id1 == id2)
```

(a) True (b) False (c) 1 (d) Exception

10. What will be the output of the following code snippet ?

```
my_dict = {}
my_dict[(1,2,4)] = 8
my_dict[(4,2,1)] = 10
my_dict[(1,2)] = 12
sum = 0
for k in my_dict:
    sum += my_dict[k]
print (sum)
print(my_dict)
```

11. Write a method in python to display the elements of list thrice if it is a number and display the element terminated with '#' if it is not a number. [CBSE D 2015]

For example, if the content of list is as follows :

```
This List = ['41', 'DROND', 'GIRIRAJ', '13', 'ZARA']
```

The output should be

```
414141
DROND#
GIRIRAJ#
131313
ZARA#
```

12. Name the function/method required to
    (*i*) check if a string contains only uppercase letters
    (*ii*) gives the total length of the list. [CBSE D 2015]

Type C : Programming Practice/Knowledge based Questions

1. Write a program that prompts for a phone number of 10 digits and two dashes, with dashes after the area code and the next three numbers. *For example*, 017-555-1212 is a legal input.
   Display if the phone number entered is valid format or not and display if the phone number is valid or not (*i.e.*, contains just the digits and dash at specific places).
2. Write a program that should prompt the user to type some sentence(s) followed by "enter". It should then print the original sentence(s) and the following statistics relating to the sentence(s) :
   - Number of words
   - Number of characters (including white-space and punctuation)
   - Percentage of characters that are alpha numeric

   Hints
   - Assume any consecutive sequence of non-blank characters in a word.
3. Write a program that takes any two lists L and M of the same size and adds their elements together to form a new list N whose elements are sums of the corresponding elements in L and M. For instance, if $L = [3, 1, 4]$ and $M = [1, 5, 9]$, then N should equal $[4, 6, 13]$.
4. Write a program that rotates the elements of a list so that the element at the first index moves to the second index, the element in the second index moves to the third index, etc., and the element in the last index moves to the first index.
5. Write a short Python code segment that prints the longest word in a list of words.
6. Write a program that creates a list of all the integers less than 100 that are multiples of 3 or 5.
7. Define two variables first and second so that first = "Jimmy" and second = "Johny". Write a short Python code segment that swaps the values assigned to these two variables and prints the results.
8. Write a Python program that creates a tuple storing first 9 terms of Fibonacci series.
9. Create a dictionary whose keys are *month names* and whose values are the *number of days* in the corresponding months.
   (*a*) Ask the user to enter a month name and use the dictionary to tell them how many days are in the month.
   (*b*) Print out all of the keys in alphabetical order.
   (*c*) Print out all of the months with 31 days.
   (*d*) Print out the (key-value) pairs sorted by the number of days in each month.
10. Write a function called addDict(dict1, dict2) which computes the union of two dictionaries. It should return a new dictionary, with all the items in both its arguments (assumed to be dictionaries). If the same key appears in both arguments, feel free to pick a value from either.
11. Write a program to sort a dictionary's keys using Bubble sort and produce the sorted keys as a list.
12. Write a program to sort a dictionary's values using Bubble sort and produce the sorted values as a list.

Viva Technologies

☎: 94257-01888,02888,03888

✉: vivatechgwl@gmail.com