5 String Manipulation

In This Chapter

5.1 INTRODUCTION

You all have basic knowledge about Python strings. You know that Python strings are characters enclosed in quotes of any type – *single quotation marks, double quotation marks* and *triple quotation marks*. You have also learnt things like – an empty string is a string that has 0 characters (*i.e.,* it is just a pair of quotation marks) and that Python strings are immutable. You have used strings in earlier chapters to store text type of data.

You know by now that strings are sequence of characters, where each character has a unique position-id/index. The indexes of a string begin from *0 to (length – 1)* in forward direction and –1, –2, –3, ..., – length in backward direction.

In this chapter, you are going to learn about many more string manipulation techniques offered by Python like operators, methods etc.

5.2 TRAVERSING A STRING

You know that individual characters of a string are accessible through the unique index of each character. Using the indexes, you can traverse a string character by character. Traversing refers to iterating through the elements of a string, one character at a time. You have already traversed through strings, though unknowingly, when we talked about sequences along with *for loops*. To traverse through a string, you can write a loop like :

```
name = "superb"
for ch in name :
    print (ch, '-', end = ' ')
```

*This loop will traverse through string **name** character by character.*

NOTE

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

The above code will print :

```
s-u-p-e-r-b-
```

The information that you have learnt till now is sufficient to create wonderful programs to manipulate strings. Consider the following programs that use the Python string indexing to display strings in multiple ways.

Program 5.1 *Program to read a string and display it in reverse order - display one character per line. Do not create a reverse string, just display in reverse order.*

```
string1 = input( "Enter a string :" )
print ("The", string1, "in reverse order is:")
length = len(string1)
for a in range(-1, (-length - 1), -1) :
    print (string1[a] )
```

Since the range() excludes the number mentioned as upper limit, we have taken care of that by increasing the limit accordingly.

Sample run of above program is :

```
Enter a string : python
The python in reverse order is:
n
o
h
t
y
p
```

Program 5.2 *Program to read a string and display it in the form :*

first character *last character*
second character *second last character*
⋮ ⋮

For example, string "try" should print as :

```
t   y
r   r
y   t
```

```
string1 = input( "Enter a string :" )
length = len(string1)
i = 0
for a in range(-1, (-length-1), -1) :
    print (string1[i], "\t", string1[a])
    i += 1
```

> **NOTE**
>
> When you give a negative index, Python adds **length** of string to get its forward index *e.g.*, for a 5-lettered string **5**, **S[–1]** will give ***S* [–1 + 5]** *i.e.*, ***S* [4]** letter ; for **S[–5]**, it will give you ***S* [– 5 + 5]** *i.e.*, ***S* [0]**

Sample run of above program is :

```
Enter a string : python
p       n
y       o
t       h
h       t
o       y
n       p
```

5.3 STRING OPERATORS

In this section, you'll be learning to work with various operators that can be used to manipulate strings in multiple ways. We'll be talking about basic operators + and *, membership operators **in** and **not in** and comparison operators (all relational operators) for strings.

5.3.1 Basic Operators

The two basic operators of strings are : + and *. You have used these operators as *arithmetic operators* before for *addition* and *multiplication* respectively. But when used with strings, + operator performs ***concatenation*** rather than addition and * operator performs ***replication*** rather than multiplication. Let us see, how.

Also, before we proceed, recall that strings are immutable *i.e.*, un-modifiable. Thus every time you perform something on a string that changes it, Python will internally create a new string rather than modifying the old string in place.

String Concatenation Operator +

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

```
"tea" + "pot"
```

will result into

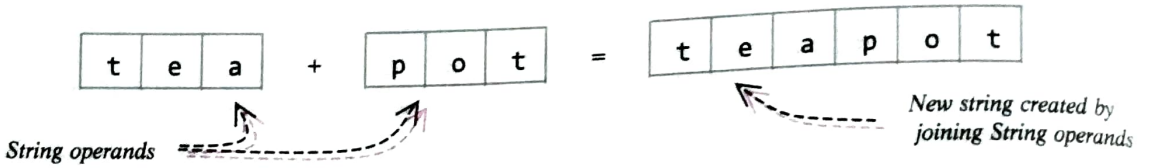
'teapot'

Two input strings joined (concatenated) to form a new string

Consider some more examples :

Expression	will result into
'1' + '1'	'11'
"a" + "0"	'a0'
'123' + 'abc'	'123abc'

Let us see how concatenation takes place internally. Python creates a new string in the mem by storing the individual characters of first string operand followed by the individ characters of second string operand. (see below)



Original strings are not modified as strings are immutable ; new strings can be crea existing strings cannot be modified.

Caution!

Another important thing that you need to know about + operator is that this operator can w with *numbers* and *strings* separately for *addition* and *concatenation* respectively, but in the sa expression, you cannot combine *numbers* and *strings* as operands with a + operator.

For example,

```
2 + 3 = 5                    # addition - VALID
'2' + '3' = '23'             # concatenation - VALID
```

But the expression

```
'2' + 3
```

is invalid. It will produce an error like :

```
>>> '2' + 3
Traceback (most recent call last):
  File "<pyshell#2>", line 1, in <module>
    '2' + 3
TypeError: cannot concatenate 'str' and 'int' objects
```

NOTE

The + operator has to have bot operands of the same typ either of number types (fo addition) or of string types (fo multiplication). It cannot wor with one operand as string and one as a number.

Thus we can summarize + operator as follows :

Table 5.1 *Working of Python + operator*

Operands' data type	*Operation performed by +*	*Example*
numbers	addition	9 + 9 = 18
string	concatenation	"9" + "9" = "99"

String Replication Operator *

The * operator when used with numbers (*i.e.*, when both operands are numbers), it perform *multiplication* and returns the *product* of the two number operands.

To use a * operator with strings, you need two types of operands – *a string* and *a number*, *i.e.*, 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 * "go!"
```

will return

```
'go!go!go!'
```

Input strings repeated specified number of times to form a new string.

NOTE

For replication operator *, Python creates a new string that is a number of repetitions of the input string.

Consider some more examples :

Expression	will result into
"abc" * 2	"abcabc"
5 * "@"	"@@@@@"
":-" * 4	":-:-:-:-"
"1" * 2	"11"

Caution!

Another important thing that you need to know about * operator is that this operator can work with numbers as both operands for *multiplication* and with a string and a number for *replication* respectively, but in the same expression, you cannot have *strings* as both the operands with a * operator.

For example,

```
2 * 3 = 6                # multiplication - VALID
"2" * 3  = "222"         # replication - VALID
```

But the expression

```
"2" *  "3"
```

is invalid. It will produce an error like :

```
>>> "2" * "3"
Traceback (most recent call last):
  File "<pyshell#0>", line 1, in <module>
    "2" * "3"
TypeError: can't multiply sequence by non-int of type 'str'
```

NOTE

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.

Thus we can summarize + operator as follows :

Table 5.2 *Working of Python * operator*

Operands' data type	Operation performed by *	Example
numbers	multiplication	9 * 9 = 18
string, number	replication	"#" * 3 = "###"
number , string	replication	3 * "#" = "###"

5.3.2 Membership Operators

There are two membership operators for strings (in fact for all sequence types). These are **in** and **not in**. We have talked about them in previous chapter, briefly. Let us learn about these operators in context of strings.

Recall that :

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>
```

e.g.,

```
"12" in "xyz"
"12" not in "xyz"
```

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

`"a" in "heya"`	will give	True
`"jap" in "heya"`	will give	False
`"jap" in "japan"`	will give	True
`"jap" in "Japan"`	will give	False because j letter's cases are different; hence "jap" is not contained in "Japan"
`"jap" not in "Japan"`	will give	True because string "jap" is not contained in string "Japan"
`"123" not in "hello"`	will give	True because string "123" is not contained in string "hello"
`"123" not in "12345"`	will give	False because "123" is contained in string "12345"

The **in** and **not in** operators can also work with string variables. Consider this :

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

5.3.3 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 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)

Equality and non-equality in strings are easier to determine because it goes for exact character matching for individual letters including the case (upper-case or lower-case) of the letter. But for other comparisons like *less than* (<) or *greater than* (>), you should know the following piece of useful information.

As internally Python compares using Unicode values (called ordinal value), let us know about some most common characters and their ordinal values. For most common characters, the ASCII values and Unicode values are the same.

The most common characters and their ordinal values are :

Table 5.3 *Common Characters and their Ordinal Values*

Characters	Ordinal Values
'0' to '9'	48 to 57
'A' to 'Z'	65 to 90
'a' to 'z'	97 to 122

Thus upper-case letters are considered smaller than the lower-case letters. For instance,

`'a' < 'A'`	will give	*False* because the Unicode value of lower-case letters is higher than upper case letters ; hence 'a' is greater than 'A', not lesser.
`'ABC' > 'AB'`	will give	*True* for obvious reasons.
`'abc' <= 'ABCD'`	will give	*False* because letters of 'abc' have higher ASCII values compared to 'ABCD'.
`'abcd' > 'abcD'`	will give	*True* because strings 'abcd' and 'abcD' are same till first three letters but the last letter of 'abcD' has lower ASCII value than last letter of string 'abcd'.

Thus, you can say that Python compares two strings through relational operators using character-by-character comparison of their Unicode values.

Determining Ordinal/Unicode Value of a Single Character

Python offers a built-in function ***ord( )*** that takes a single character and returns the cc ponding ordinal Unicode value. It is used as per following format :

```
ord(<single-character>)
```

Let us see how, with the help of some examples :

- To know the ordinal value of letter 'A', you'll write **ord('A')** and Python will return th corresponding ordinal value (see below) :

```
>>> ord( 'A' )
65
```

But you need to keep in mind that ***ord( )*** function requires single character string only. You m even write an escape sequence enclosed in quotes for *ord( )* function.

The opposite of **ord( )** function is **chr( )**, *i.e.*, while **ord( )** returns the ordinal value of a charact the **chr( )** takes the ordinal value in integer form and returns the character corresponding to th ordinal value. The general syntax of **chr( )** function is :

```
chr(<int>)                    # the ordinal value is given in integer
```

Have a look at some examples (compare with the Table 5.3 given above) :

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

5.4 STRING SLICES

As an English term, you know the meaning of word 'slice', which means – 'a part of'. In the same way, in Python, the term '*string slice*' refers to a part of the string, where strings are sliced using a range of indices.

STRING SLICE

Part of a string containing some contiguous characters from the string.

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 $m-1$. Let us understand this with the help of examples. Say we have a string namely ***word*** storing a string 'amazing' *i.e.*,

	0	1	2	3	4	5	6
word	**a**	m	a	z	i	**n**	g
	–7	–6	–5	–4	–3	–2	–1

Then,

`word[ 0 : 7 ]`	will give	`'amazing'`	(the letters starting from index 0 going up till 7 – 1 *i.e.*, 6 : from indices 0 to 6, both inclusive)
`word[ 0 : 3 ]`	will give	`'ama'`	(letters from index 0 to 3 – 1 *i.e.*, 0 to 2)

`word[ 2 : 5 ]`	will give	`'azi'`	(letters from index 2 to 4 (*i.e.*, 5 – 1))
`word[-7 : -3]`	will give	`'amaz'`	(letters from indices –7, –6, –5, –4 excluding index –3)
`word[-5 : -1]`	will give	`'azin'`	(letters from indices –5, –4, –3, –2 excluding –1)

From above examples, one thing must be clear to you :

- In a string slice, the character at last index (the one following colon (:)) is not included in the result.

In a string slice, you give the slicing range in the form *[<begin-index> :<last >]*. If, however, you skip either of the ***begin-index*** or ***last***, Python will consider the limits of the string *i.e.*, for missing ***begin-index***, it will consider **0** (the first index) and for missing ***last*** value, it will consider **length of the string.**

Consider following examples to understand this :

`word[:7]`	will give	`'amazing'`	(missing index before colon is taken as 0 (zero))
`word[:5]`	will give	`'amazi'`	(-do-)
`word[3:]`	will give	`'zing'`	(missing index after colon is taken as 7 (the length of the string))
`word[5:]`	will give	`'ng'`	(-do-)

The string slice refers to a part of the string **s[start:end]** that is the elements beginning at **start** and extending up to but not including **end.**

Following figure (Fig. 5.1) shows some string slices :

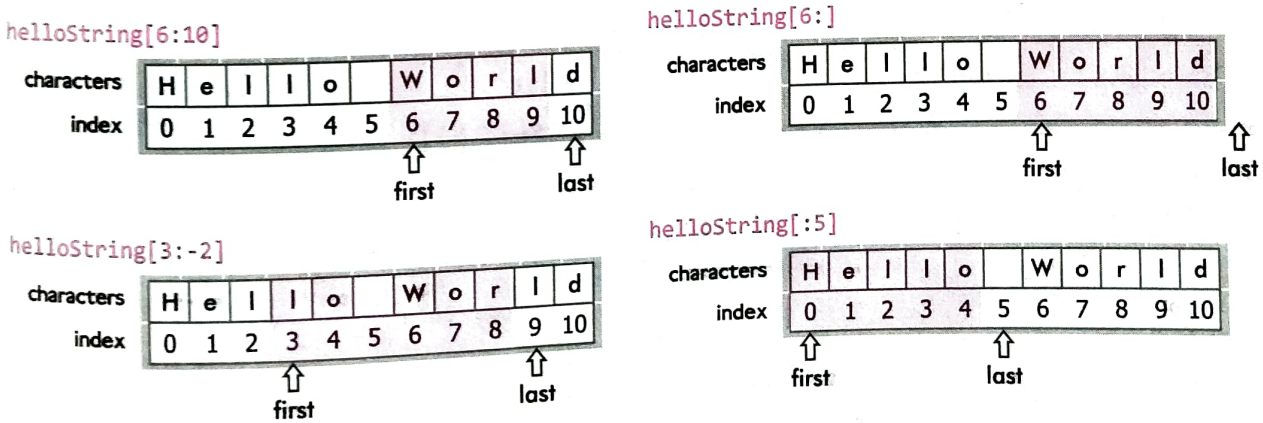


Figure 5.1 String Slicing in Python.

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.

Let us prove this with an example. Consider the same 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'
```

TIP

String [: : –1] is an easy way to reverse a string.

You can give a third (optional) index (say *n*) in string slice too. With that every *n*th element will be taken as part of slice *e.g.,* for word = 'amazing', look at following examples.

```
>>> word [1:6:2]
'mzn'
```
It will take every 2nd character starting from index = 1 till index < 6.

```
>>> word [–7:–3:3]
'az'
```
It will take every 3rd character starting from index = –7 to index < –3.

```
>>> word [: : –2]
'giaa'
```
Every 2nd character taken backwards.

```
>>> word [: : –1]
'gnizama'
```
Every character taken backwards.

Another interesting inference is :

- 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. The reason behind this is that when you use an index, you are accessing a constituent character of the string, thus the index must be valid and out of bounds index causes error as there is no character to return from the given index. But slicing always returns a subsequence and empty sequence is a valid sequence. Thus when you slice a string outside the bounds, it still can return empty subsequence and hence Python gives no error and returns empty subsequence.

Truly amazing ;). Isn't it ?.

Check Point 5.1

1. How are strings internally stored ?
2. For a string **s** storing 'Goldy', what would s[0] and s[-1] return ?
3. The last character of a string s is at index len(s) - 1. True / False ?
4. For strings, + means (1) ; * means (2) . Suggest words for positions (1) and (2).
5. Given that

   ```
   s1 = "spam "
   s2 = "ni!"
   ```

   What is the output produced by following expressions ?
   (*a*) " The Knights who say, " + s2
   (*b*) 3 * s1 + 2 * s2
   (*c*) s1[1]
6. What are membership operators? What do they basically do ?
7. On what principles, the strings are compared in Python ?
8. What will be the result of following expressions ?
   (*a*) "Wow Python" [1]
   (*b*) "Strings are fun." [5]
   (*c*) len("awesome")
   (*d*) "Mystery" [:4]
   (*e*) "apple" > "pineapple"
   (*f*) "pineapple" < "Peach"
   (*g*) "cad" in "abracadabra"
   (*h*) "apple" in "Pineapple"
   (*i*) "pine" in "Pineapple"
9. What do you understand by string slices ?
10. Considering the same strings s1 and s2 of question 5 above, evaluate the following expressions :
    (*a*) s1[1:3]
    (*b*) s1[2] + s2[:2]
    (*c*) s1 + s2[-1]
    (*d*) s1[:3] + s1[3:]
    (*e*) s2[:-1] + s2[-1:]

Program 5.3 *Program that prints the following pattern without using any nested loop*

```
#
##
###
####
#####
```

```
string = '#'
pattern = ""          # empty string
for a in range(5) :
    pattern += string
    print (pattern)
```

Sample run of the program is as shown below :

```
#
##
###
####
#####
```

5.5 STRING FUNCTIONS AND METHODS

Python also offers many built-in functions and methods for string manipulation. You have already worked with one such method **len( )** in earlier chapters. In this chapter, you will learn about many other built-in powerful string methods of Python used for string manipulation.

Every string object that you create in Python is actually an instance of String class (you need not do anything specific for this ; Python does it for you – you know built-in). The string manipulation methods that are being discussed below can be applied to string as per following syntax :

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

In the following table we are referring to <stringObject> as ***string*** only (no angle brackets) but the meaning is intact *i.e.,* you have to replace ***string*** with a legal string (*i.e.,* either a string literal or a string variable that holds a string value).

Let us now have a look at some useful built-in string manipulation methods.

Table 5.4 *Python's Built-in String Manipulation Methods*

`string.capitalize()`	Returns a copy of the ***string*** with its first character capitalized.
	Example
	`>>> 'true'.capitalize()`
	`'True'`
	`>>> ' 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)`
	`13`
	`>>> string.find(sub, 15, 22)`
	`–1`
	`>>> string.find(sub, 15, 25)`
	`19`

`string.isalnum()`	Returns **True** if the characters in the ***string*** are alphanumeric (alphabets or numbers) and there is at least one character, **False** otherwise.
	Example
	`>>> string = "abc123"`
	`>>> string2 = 'hello'`
	`>>> string3 = '12345'`
	`>>> string4 = ' '`
	`>>> string.isalnum()`
	`True`
	`>>> string2.isalnum()`
	`True`
	`>>> string3.isalnum()`
	`True`
	`>>> string4.isalnum()`
	`False`
	`>>>`

`string.isalpha()`	Returns **True** if all characters in the ***string*** are alphabetic and there is at least one character, **False** otherwise.
	Example
	(considering the same string values as used in example of previous function - **isalnum**)
	`>>> string.isalpha()`
	`False`
	`>>> string2.isalpha()`
	`True`
	`>>> string3.isalpha()`
	`False`
	`>>> string4.isalpha()`
	`False`

`string.isdigit()`	Returns **True** if all the characters in the ***string*** are digits. There must be at least one character, otherwise it returns **False**.
	Example
	(considering the same string values as used in example of previous function - **isalnum**)
	`>>> string.isdigit()`
	`False`
	`>>> string2.isdigit()`
	`False`
	`>>> string3.isdigit()`
	`True`
	`>>> string4.isdigit()`
	`False`

string.islower()	Returns **True** if all cased characters in the ***string*** are lowercase. There must be at least one cased character. It returns **False** otherwise.
	Example
	`>>> string = 'hello'`
	`>>> string2 = 'THERE'`
	`>>> string3 = 'Goldy'`
	`>>> string.islower()`
	`True`
	`>>> string2.islower()`
	`False`
	`>>> string3.islower()`
	`False`

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.isupper()	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.
	Example
	`>>> string = "HELLO"`
	`>>> string2 ="There"`
	`>>> string3 = "goldy"`
	`>>> string4 = "U123"  # character in uppercase`
	`>>> string5 = "123f"  # character in lowercase`
	`>>> 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 (considering the same string values as used in example of previous function - **isupper**)
	`>>> string.lower()`
	`'hello'`
	`>>> string2.lower()`
	`'there'`
	`>>> string3.lower()`
	`'goldy'`
	`>>> string4.lower()`
	`'u123'`
	`>>> string5.lower()`
	`'123f'`

`string.upper()`	Returns a copy of the ***string*** converted to uppercase.
	Example (considering the same string values as used in example of previous function - **isupper**)
	`>>> string.upper()`
	`'HELLO'`
	`>>> string2.upper()`
	`'THERE'`
	`>>> string3.upper()`
	`'GOLDY'`
	`>>> string4.upper()`
	`'U123'`
	`>>> string5.upper()`
	`'123F'`

`string.lstrip([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*** : **Example** ▲ The example for lstrip( ) requires a detailed discussion, hence we are giving example of lstrip( ) at the end of this table.
`string.rstrip([chars])`	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 : **Example** ▲ The example for rstrip( ) requires a detailed discussion, hence we are giving example of rstrip( ) at the end of this table.

Example for lstrip( ) function

```
>>>string = "hello"
>>>string.lstrip()
'hello'
>>>string2 = 'There'
'There'
>>>string2.lstrip( 'the' )
'There'
>>>string2.lstrip( 'The' )
're'
>>>string2.lstrip('he')
'There'
>>>string2.lstrip( 'Te')
'here'
>>>string2.lstrip('Teh')
're'
>>>string2.lstrip('heT')
're'
```

No argument supplied to lstrip() hence it removed leading whitespaces of the string

All possible substrings of given argument 'the' are matched with left of the string2 and if found, removed. That is,
'the', 'th', 'he', 'te' , 't', 'h', 'e' and their reversed strings are matched, if any of these is found, it is removed from the left of the string

'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

No match in the left of string found for 'he', 'eh', 'h', 'e' ; thus same string returned.

'heT', '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

On the same lines, you'll be able to justify that

```
>>>"saregamapadhanisa".lstrip( "tears" )
'gamapadhanisa'
>>>"saregamapadhanisa".lstrip( "races" )
'gamapadhanisa'
```

Example for rstrip( ) function

The *rstrip( )* works on identical principle as that of *lstrip( )*, the only difference is that it matches from right direction. Consider following examples. We know, you'll be able to understand if you have carefully read the reasons for *lstrip( )* examples.

```
>>>string
'hello'
>>>string.rstrip()
'hello'
>>>string2
'There'
>>>string2.rstrip('ere')
```

Check Point 5.2

1. What is the role of these functions ?
   (*i*) isalpha( ) (*ii*) isalnum( )
   (*iii*) isdigit( ) (*iv*) isspace( )
2. Name the case related string manipulation functions.
3. How is islower( ) function different from lower( ) function ?
4. What is the utility of find( ) function ?
5. How is capitalize( ) function different from upper( ) function ?
6. How do lstrip( ) and rstrip( ) functions work ?

```
'Th'
>>> string2.rstrip('care')
'Th'
>>> string2.rstrip('car')
'There'
>>>
>>> "saregamapadhanisa".rstrip( "tears" )
'saregamapadhani'
>>> "saregamapadhani".rstrip( "races" )
'saregamapadhani'
```

One more function that you have used with strings is **len( )** function which gives you the length of the string as the count of characters contained in it. Recall that you use it as :

```
len(<string>)
```

For example,

```
>>> string = 'hello'
>>> len(string)
5
```

Consider following program that applies some of the string manipulation functions that you have learnt so far.

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

Program

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)
```

Sample run of the program is :

```
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
```

Program 5.5 *Program that reads a line and a substring. It should then display the number of occurrences of the given substring in the line.*

```
line = input( "Enter line :" )
sub = input( "Enter substring :" )
length = len(line)
lensub = len(sub)
start = count = 0
end = length
while True :
    pos = line.find(sub, start, end)
    if pos != -1 :
        count += 1
        start = pos + lensub
    else :
        break
    if start >= length :
        break
print ("No. of occurrences of", sub, ':', count)
```

Sample runs of above program is :

```
Enter line : jingle bells jingle bells jingle all the way
Enter substring : jingle
No. of occurrences of jingle : 3
========== RESTART ==========
Enter line : jingle bells jingle bells jingle all the way
Enter substring : bells
No. of occurrences of bells : 2
```

PriP STRING MANIPULATION — Progress In Python 5.1

This *'Progress in Python'* session works on the objective of practicing String manipulation operators and functions.

Please check the practical component-book – Progress in Computer Science with Python and fill it there in PriP 5.1 under Chapter 5 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 memory locations of string characters are given indexes or indices.*
- *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.*
- *For strings, + means concatenation, * means replication.*
- ***The in*** *and* ***not in*** *are membership operators that test for a substring's presence in the string.*
- *Comparison in Python strings takes place in dictionary order by applying character-by-character comparison rules for ASCII or Unicode.*
- *The ord( ) function returns the ASCII value of given character.*
- *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****.*
- *The string slice with syntex* ***s[start : end : n]*** *is the element beginning at* ***start*** *and extending up to but not including* ***end****, taking every* ***nth*** *character.*
- *Python also provides some built-in string manipulation methods like : capitalize( ), find( ), isalnum( ), isalpha( ), isdigit( ), islower( ), isupper( ), lower( ), upper( ), lstrip( ), rstrip( ) etc.*

Solved Problems

1. *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'
   ```

2. *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
```

3. *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

4. *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*) 1*bzc**
(*ii*) 1*bb***

(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.

5. *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 ("Capitalized words String", string2)
```

6. *Write a program that reads a string and checks whether it is a palindrome 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 :
        print (string, "is a palindrome")
```

7. *Write a program that reads a string and displays the longest substring of the given string having just the consonants.*

Solution.

```
string = input( "Enter a string :" )
length = len(string)
```

```
sub = ''                    # empty string
lensub = 0                  # empty string
for a in range(length):
    if string[a] in 'aeiou' or string[a] in 'AEIOU' :
        if lensub > maxlength :
            maxsub = sub
            maxlength = lensub
            sub = ''
            lensub = 0
    else :
        sub += string[a]
        lensub = len(sub)
    a += 1
print ("Maximum length consonant substring is :"  , maxsub, end = ' ')
print ("with" , maxlength,"characters")
```

8. *Write a program that reads a string and then prints a string that capitalizes every other letter in the string e.g., passion becomes pAsSiOn.*

Solution.

```
string = input( "Enter a string :" )
length = len(string)
print ("Original string :", string)
string2 = " "                          # empty string
for a in range(0, length, 2) :
    string2 += string[a]
    if a < (length-1) :
        string2 += string[a + 1].upper()
print ("Alternatively capitalized string :"  , string2)
```

9. *Write a program that reads email-id of a person in the form of a string and ensures that it belongs to domain @edupillar.com. (Assumption : No invalid characters are there in email-id).*

Solution.

```
email = input( "Enter your email id :" )
domain = '@edupillar.com'
ledo = len(domain)              # ledo – length of domain
lema = len(email)               # lema – length of email
sub = email[lema-ledo :]
if sub == domain :
    if ledo != lema :
        print ("It is valid email id")
    else :
        print ("This is invalid email id - contains just the domain name.")
else:
    print ("This email-id is either not valid or belongs to some other domain.")
```

Index	A variable or value used to select a member character from a string,
String slice	A part of a string (substring) specified by a range of indices.
Subscript	Index.
Traversal	Iterating through a sequence such as a string, member by member.

Assignments

Type A : Short Answer Questions/Conceptual Questions

1. Write a Python script that traverses through an input string and prints its characters in different lines – two characters per line.
2. Out of the following operators, which ones can be used with strings ?

   =, –, *, /, //, %, >, <>, in, not in, <=
3. What is the result of following statement, if the input is 'Fun' ?

```
print ( input(" ... ") + "trial" + "Ooty" * 3)
```

4. Which of the following is not a Python legal string operation ?

   (*a*) 'abc' + 'abc' (*b*) 'abc' * 3

   (*c*) 'abc' + 3 (*d*) 'abc'.lower()
5. Can you say strings are character lists ? Why ? Why not ?
6. Given a string **S** = "CARPE DIEM". If **n** is **length/2** (*length* is the length of the given string), then what would following return ?

   (*a*) S[: n] (*b*) S[n :] (*c*) S[n : n] (*d*) S[1 : n] (*e*) S[n : length – 1]
7. From the string S = "CARPE DIEM", which ranges return "DIE" and "CAR" ?
8. What would following expression return ?

   (*a*) "Hello World".upper( ).lower( ) (*b*) "Hello World".lower( ).upper( )

   (*c*) "Hello World".find("Wor", 1, 6) (*d*) "Hello World".find("Wor")

   (*e*) "Hello World".find("wor") (*f*) "Hello World".isalpha( )

   (*g*) "Hello World".isalnum( ) (*h*) "1234".isdigit( )

   (*i*) "123FGH".isdigit( )
9. Which functions would you choose to use to remove leading and trailing white spaces from a given string ?
10. Try to find out if for any case, the string functions **isalnum**( ) and **isalpha**( ) return the same result.
11. Suggest appropriate functions for the following tasks :

    (*i*) To check whether the string contains digits

    (*ii*) To find for the occurrence a string within another string

    (*iii*) To convert the first letter of a string to upper case

    (*iv*) to capitalize all the letters of the string

    (*v*) to check whether all letters of the string are in capital letters

    (*vi*) to remove from right of a string all string- combinations from a given set of letters

    (*vii*) to remove all white spaces from the beginning of a string

1. What is the result of the following expressions ?

(a)
```
print (" " "
1
 2
  3
" " " )
```

(c)
```
text = "Test.\nNext line."
print (text)
```

(c)
```
print ('One', ' Two ' * 2)
print ('One ' + 'Two' * 2)
print (len('0123456789'))
```

(d)
```
s = '0123456789'
print (s[3], ", ", s[0 : 3], " - ", s[2 : 5])
print (s[:3], " - ", s[3:], " , ", s[3:100])
print (s[20:], s[2:1], s[1:1])
```

(e)
```
s = '987654321'
print (s[-1], s[-3]
print (s[-3:], s[:-3])
print (s[-100:-3], s[-100:3])
```

2. 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, ' : ', end = ' ')
```

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

3. Carefully go through the code given below and answer the questions based on it :

```
theStr = " This is a test "
inputStr = input(" Enter integer: ")
inputInt = int(inputStr)
testStr = theStr
while inputInt >= 0 :
    testStr = testStr[1:-1]
    inputInt = inputInt - 1
testBool = ' t ' in testStr
print (theStr)              # Line 1
print (testStr)             # Line 2
print (inputInt)            # Line 3
print (testBool)            # Line 4
```

(i) Given the input integer 3, what output is produced by Line 1 ?

(a) This is a test (b) This is a (c) is a test (d) is a (e) None of these

(ii) Given the input integer 3, what output is produced by Line 2 ?

(a) This is a test (b) This is a (c) is a test (d) is a (e) None of these

(iii) Given the input integer 2, what output is produced by Line 3 ?

(a) 0 (b) 1 (c) 2 (d) 3 (e) None of these

(iv) Given the input integer 2, what output is produced by Line 4 ?

(a) False (b) True (c) 0 (d) 1 (e) None of these

4. Carefully go through the code given below and answer the questions based on it :

```
testStr = "abcdefghi"
inputStr = input ("Enter integer:")
inputInt = int(inputStr)
count = 2
newStr = ''
while count <= inputInt :
    newStr = newStr + testStr[0 : count]
    testStr = testStr[2:]          #Line 1
    count = count + 1
print (newStr)                     # Line 2
print (testStr)                    # Line 3
print (count)                      # Line 4
print (inputInt)                   # Line 5
```

(*i*) Given the input integer 4, what output is produced by Line 2 ?

(*a*) abcdefg (*b*) aabbccddeeffgg (*c*) abcdeefgh (*d*) ghi (*e*) None of these

(*ii*) Given the input integer 4, what output is produced by Line 3 ?

(*a*) abcdefg (*b*) aabbccddeeffgg (*c*) abcdeefgh (*d*) ghi (*e*) None of these

(*iii*) Given the input integer 3, what output is produced by Line 4 ?

(*a*) 0 (*b*) 1 (*c*) 2 (*d*) 3 (*e*) None of these

(*iv*) Given the input integer 3, what output is produced by Line 5 ?

(*a*) 0 (*b*) 1 (*c*) 2 (*d*) 3 (*e*) None of these

(*v*) Which statement is equivalent to the statement found in Line 1 ?

(*a*) `testStr = testStr[2:0]` (*b*) `testStr = testStr[2:-1]`

(*c*) `testStr = testStr[2:-2]` (*d*) `testStr = testStr - 2`

(*e*) None of these

5. Carefully go through the code given below and answer the questions based on it :

```
inputStr = input(" Give me a string:")
bigInt = 0
littleInt = 0
otherInt = 0
for ele in inputStr:
    if ele >= 'a' and ele <= 'm':          # Line 1
        littleInt = littleInt + 1
    elif ele > 'm' and ele <= 'z':
        bigInt = bigInt + 1
    else :
        otherInt = otherInt + 1
print (bigInt)                             # Line 2
print (littleInt)                          # Line 3
print (otherInt)                           # Line 4
print (inputStr.isdigitQ)                  # Line 5
```

(*i*) Given the input abcd what output is produced by Line 2 ?

(*a*) 0 (*b*) 1 (*c*) 2 (*d*) 3 (*e*) None of these

(ii) Given the input *Hi Mom* what output is produced by Line 3 ?
(a) 0 (b) 1 (c) 2 (d) 3 (e) None of these

(iii) Given the input *Hi Mom* what output is produced by Line 4 ?
(a) 0 (b) 1 (c) 2 (d) 3 (e) None of these

(iv) Given the input 1+2 =3 what output is produced by Line 5 ?
(a) 0 (b) 1 (c) True (d) False (e) None of these

(v) Give the input *Hi Mom*, what changes result from modifying Line 1 from
if ele >= 'a' and ele <= 'm' to the expression
if ele >= 'a' and ele < 'm' ?
(a) No change (b) otherInt would be larger (c) littleInt would be larger
(d) bigInt would be larger (e) None of these

6. Carefully go through the code given below and answer the questions based on it :

```
in1Str = input(" Enter string of digits: ")
in2Str = input(" Enter string of digits: ")

if len(in1Str)>len(in2Str):
    small = in2Str
    large = in1Str
else :
    small = in1Str
    large = in2Str
newStr = ''
for element in small:
    result = int(element) + int(large[0])
    newStr = newStr + str(result)
    large = large[1:]
print (len(newStr)                    # Line 1
print (newStr)                        # Line 2
print (large)                         # Line 3
print (small)                         # Line 4
```

(i) Given a first input of 12345 and a second input of 246, what result is produced by Line 1 ?
(a) 1 (b) 3 (c) 5 (d) 0 (e) None of these

(ii) Given a first input of 12345 and a second input of 246, what result is produced by Line 2 ?
(a) 369 (b) 246 (c) 234 (d) 345
(e) None of these

(iii) Given a first input of 123 and a second input of 4567, what result is produced by Line 3 ?
(a) 3 (b) 7 (c) 12 (d) 45 (e) None of these

(iv) Given a first input of 123 and a second input of 4567, what result is produced by Line 4 ?
(a) 123 (b) 4567 (c) 7 (d) 3 (e) None of these

7. Find the output :

(a)
```
S = input("Enter String :")
RS = " "
for ch in S :
    RS = ch + RS
print(S + RS)
```

(b)
```
S = input("Enter string :")
RS = " "
for ch in S :
    RS = ch + 2 + RS
print(RS + S)
```

8. Find the errors

(a)
```
S = "PURA VIDA"
print(S[9] + S[9 : 15])
```

(b)
```
S = "PURA VIDA"
S1 = S[: 10] + S[10 :]
S2 = S[10] + S[-10]
```

(c)
```
S = "PURA VIDA"
S1 = S * 2
S2 = S1[-19] + S1[-20]
S3 = S1[-19 :]
```

(d)
```
S = "PURA VIDA"
S1 = S[: 5]
S2 = S[5 :]
S3 = S1 * S2
S4 = S2 + '3'
S5 = S1 + 3
```

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 :
   - prompt the user for a string
   - extract all the digits from the string.
   - If there are digits :
     - sum the collected digits together
     - print out :
       - the original string
       - the digits
       - the sum of the digits
   - If there are no digits:
     - print the original string and a message "has no digits"

   Sample
   - given the input : abc123
     prints abc123 has the digits 123 which sum to 6
   - given the input : abcd
     prints abcd has no digits

3. 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 is a word.

4. Write a Python program as per specifications given below :
   - Repeatedly prompt for a sentence (string) or for 'q' to quit.
   - Upon input of a sentence s, print the string produced from s by converting each lower case letter to upper case and each upper case letter to lower case.
   - All other characters are left unchanged.

For example,

```
========== RESTART ==========
Please enter a sentence, or 'q' to quit : This is the Bomb!
tHIS IS THE bOMB!
Please enter a sentence, or 'q ' to quit : What's up Doc ???
wHAT'S UP dOC ???
Please enter a sentence, or 'q' to quit : q
```

5. Write a program that does the following :
   - takes two inputs : the first, an integer and the second, a string
   - from the input string extract all the digits, in the order they occurred, from the string.
     - if no digits occur, set the extracted digits to 0
   - add the integer input and the digits extracted from the string together as integers
   - print a string of the form :
     "integer_input + string_digits = sum"

For example :

```
For inputs 12, ' abc123 ' → ' 12 + 123 = 135 '
For inputs 20, ' a5b6c7 ' → ' 20 + 567 = 587 '
For inputs 100, ' hi mom ' → ' 100 + 0 = 100 '
```

6. On what principles does Python compare two strings ? Write a program that takes two strings from the user and displays the smaller string in single line and the larger string as per this format :

```
1st letter                    last letter
   2nd letter              2nd last letter
      3rd letter        3rd last letter
                    :
```

For example,

if the two strings entered are Python and PANDA then the output of the program should be :

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

7. Write a program to convert a given number into equivalent Roman number (store its value as a string). You can use following guidelines to develop solution for it :
   - From the given number, pick successive digits, using %10 and /10 to gather the digits from right to left.
   - The rules for Roman Numerals involve using four pairs of symbols for ones and five, tens and fifties, hundreds and five hundreds. An additional symbol for thousands covers all the relevant bases.
   - When a number is followed by the same or smaller number, it means addition. "II" is two 1's = 2. "VI" is 5 + 1 = 6.
   - When one number is followed by a larger number, it means subtraction. "IX" is 1 before 10 = 9. "IIX" isn't allowed, this would be "VIII". For numbers from 1 to 9, the symbols are "I" and "V", and the coding works like this. "I" , "II", "III", "IV", "V", "VI", "VII", "VIII", "IX".
   - The same rules work for numbers from 10 to 90, using "X" and "L". For numbers from 100 to 900, using the sumbols "C" and "D". For numbers between 1000 and 4000, using "M".

Here are some examples. 1994 = MCMXCIV, 1956 = MCMLVI, 3888= MMMDCCCLXXXVIII