String Manipulation

chapter 10

In This Chapter

10.1 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.

10.2 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 :

To see String Traversal in action

Scan QR Code

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

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

The above code will print :

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

NOTE

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

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

Program 10.1 *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
```

NOTE

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

Program 10.2 *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
```

Sample run of above program ...

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

10.3 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.

10.3.1 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

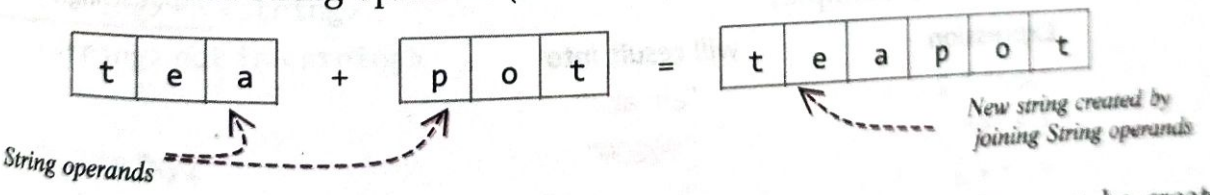
```
'teapot'
```

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 memory by storing the individual characters of first string operand followed by the individual characters of second string operand. (see below)



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

Caution !

Another important thing that you need to know about + operator is that this operator can work with *numbers* and *strings* separately for *addition* and *concatenation* respectively, but in the same 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
```

Thus we can summarize + operator as follows :

NOTE

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.

Table 10.1 *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 performs *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., 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 * "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 10.2 *Working of Python * operator*

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

10.3.2 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
```

Program 10.3 Write a program to input an integer and check if it contains any 0 in it.

```
n = int(input("Enter a number :"))
s = str(n)          # convert n to string
if '0' in s:
    print("There's a 0 in", n)
else:
    print("No 0 in", n)
```

```
Enter a number : 6708
There's a 0 in 6708
====================
Enter a number : 7281
No 0 in 7281
```

10.3.3 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 shown in Table 10.3.

Table 10.3 *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

`'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 corresponding 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 the corresponding ordinal value (*see* below) :

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

But you need to keep in mind that ***ord( )*** function requires single character string only. You may 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 character, the **chr( )** takes the ordinal value in integer form and returns the character corresponding to that 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 10.3 given above) :

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

Program 10.4 An easy way to print a blank line is **print( )**. However, to print ten blank lines, would you write *print( )* 10 times ? Or is there a smart alternative ? If yes, write code to print a line with 30 '=' characters followed by 10 blank line and another line with 30 '=' characters.

```
print('=' * 30)
print('\n'* 9)
print('=' * 30)
```

```
==============================


==============================
```

Program 10.5 Write a program that asks a user for a *username* and a *code*. Ensure that the user doesn't use their username as part of their code.

```
uname = input("Enter user name :")
code = input("Enter code :")
if uname in code :
    print("Your code should not \
    contain your user name.")
print("Thank you")
```

```
Enter user name : kobe
Enter code : mykobeworld
Your code should not contain your user
name.
Thank you
```

Program 10.6 Write a program that asks a user for a *username* and a *pcode*. Make sure that *pcode* does not contain username and matches *'Trident111'*.

```
uname = input ("Enter user name :")
pcode = input ("Enter code :")
if uname not in pcode and pcode == 'Trident111' :
    print("Your code is Valid to proceed.")
else :
    print("Your code is Not Valid.")
```

```
Enter user name : Raina
Enter code : Xerxes111
Your code is Not Valid.
Thank you
==========
Enter user name : Raina
Enter code : Raina123
Your code is Not Valid.
==========
Enter user name : Raina
Enter code : Trident111
Your code is Valid to proceed.
```

Program 10.7 *Program that prints the following pattern without using any nested loop.*

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

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

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

10.4 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.

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.*,

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

	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.

To see String Slices in action

Scan QR Code

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. 10.1) shows some string slices :

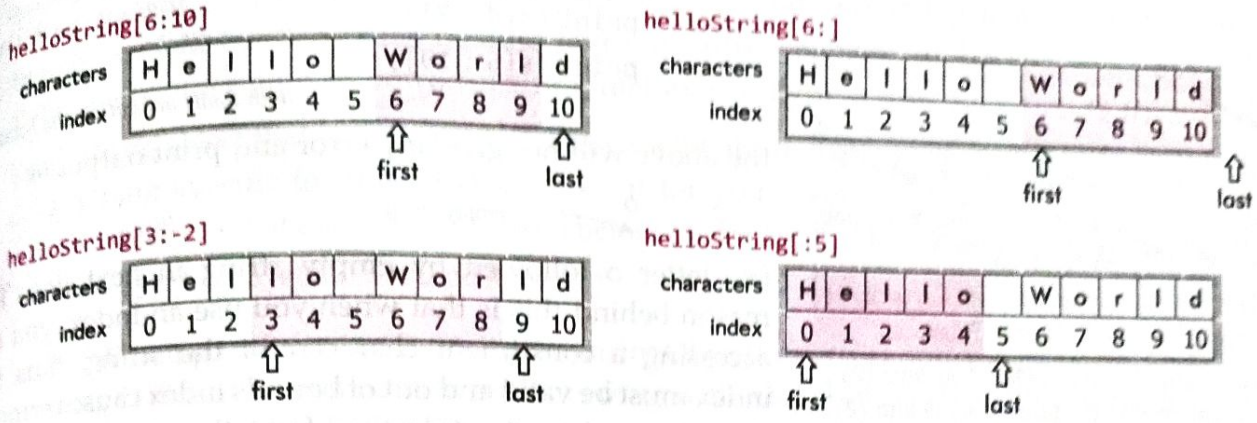


Figure 10.1 String Slicing in Python.

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 10.1

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 10.8 Write a program to input two strings. If string1 is contained in string2, then create a third string with first four characters of string2' added with word 'Restore'.

```
s1 = input ("Enter string 1 :")
s2 = input ("Enter string 2 :")
print("Original strings :", s1, s2)
if s1 in s2 :
    s3 = s2[0:4] + "Restore"
print("Final strings :", s1, s3)
```

```
Enter string 1 : rin
Enter string 2 : shagrin
Original strings : rin shagrin
Final strings : rin shagRestore
```

Program 10.9 Write a program to input a string and check if it is a palindrome string using a string slice.

```
s = input("Enter a string :")
if (s == s[::-1]) :
    print(s, "is a Palindrome.")
else :
    print(s, "is NOT a Palindrome.")
```

```
Enter a string : Mira
Mira is NOT a Palindrome.
==========
Enter a string : mirarim
mirarim is a Palindrome.
```

10.5 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.

Python's Built-in String Manipulation Functions and Methods

Let us now discuss some most commonly used string functions and methods of Python.

1. The len( ) function

It returns the length of its argument string, *i.e.*, it returns the count of characters in the passed string. The ***len( )*** function is a Python standard library function.

It is used as per the syntax :

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

Examples

```
>>> len("hello ")
6
>>> name = "Maria"
>>> len(name)
5
```

2. The capitalize( ) method

It returns a copy of the string with its first character capitalized.

It is used as per the syntax :

```
<string>.capitalize( )
```

Example

```
>>> 'true'.capitalize()
True
>>> 'i love my India'.capitalize()
I love my india
```

3. The count ( ) method

It returns the number of occurrences of the substring **sub** in ***string*** (or ***string [start:end]*** if these arguments are given).

It is used as per the syntax :

```
<string>.count (sub[, start[, end] ] )
```

Examples

To count the occurrences of *'ab'* in – *'abracadabra'* in the whole string, 4-8 characters and 6th character onwards.

```
>>> 'abracadabra'.count('ab')
2
>>> 'abracadabra'.count('ab', 4, 8)
0
>>> 'abracadabra'.count('ab', 6)
1
```

Count the occurrences of 'ab' in the whole string

Count the occurrences of 'ab' in string's 4th to 8th characters.

Count the occurrences of 'ab' in string's 6th character onwards

4. The find( ) method

It 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.

It is used as per the syntax :

```
<string>.find(sub[, start[, end]])
```

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
```

5. The index( ) method

It returns the *lowest index* where the specified substring is found. If the substring is not found then an exception, **ValueError**, is raised. It works like **find( )**, but **find( )** returns –1 if the ***sub*** is not found, BUT **index( )** raises an exception, if ***sub*** is not found in the string.

It is used as per the syntax :

```
<string>.index(sub[, start[, end]])
```

Examples

To find the index of the 1st occurrence of 'ab' in 'abracadabra' – in the whole string, 4-8 characters – and 6th character onwards.

```
>>> 'abracadabra'.index('ab')
0
```

Returned the index of 1st occurrence of 'ab' in the whole string

```
>>> 'abracadabra'.index('ab', 6)
7
```

Returned the index of 1st occurrence of 'ab' in the string's 6th character onwards

```
>>> 'abracadabra'.index('ab', 4, 8)
Traceback (most recent call last):
  File"<pyshell#7>", line 1, in <module>
    'abracadabra'.index('ab', 4, 8)
ValueError: substring not found
```

*Raised **ValueError** exception, when 'ab' could not be found in given range of the string*

NOTE

Use the **index( )** and **find( )** functions only when you want to know the index position of the substring. For checking the presence of a substring, use the **in** operator instead.

6. The isalnum( ) method

It returns **True** if the characters in the *string* are alphanumeric (alphabets or numbers) and there is at least one character, **False** otherwise. Please note that the space (' ') is not treated as alphanumeric.

It is used as per the syntax :

```
<string>.isalnum( )
```

Example

```
>>> string = "abc123"
>>> string2 = 'hello'
>>> string3 = '12345'
>>> string4 = ' '
>>> string.isalnum()
True
>>> string2.isalnum()
True
>>> string3.isalnum()
True
>>> string4.isalnum()
False
```

NOTE

To check if *c* is a space or a special character, you can use **c.isalnum( ) != True**

7. The isalpha( ) method

It returns **True** if all characters in the *string* are alphabetic and there is at least one character, **False** otherwise.

It is used as per the syntax :

```
<string>.isalpha( )
```

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
```

8. The isdigit( ) method

It returns **True** if all the characters in the ***string*** are digits. There must be at least one character, otherwise it returns **False**.

It is used as per the syntax :

```
<string>.isdigit( )
```

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
```

9. The islower( ) method

It returns **True** if all cased characters in the ***string*** are lowercase. There must be at least one cased character. It returns **False** otherwise.

It is used as per the syntax :

```
<string>.islower( )
```

Example

```
>>> string = 'hello'
>>> string2 = 'THERE'
>>> string3 = 'Goldy'
>>> string.islower()
True
>>> string2.islower()
False
>>> string3.islower()
False
```

10. The isspace( ) method

It returns **True** if there are only whitespace characters in the ***string***. There must be at least one character. It returns **False** otherwise.

It is used as per the syntax :

```
<string>.isspace( )
```

Example

```
>>> string = "   "      # stores three spaces
>>> string2 = ""        # an empty string
>>> string.isspace()
True
>>> string2.isspace()
False
```

11. The isupper( ) method

It tests whether all cased characters in the ***string*** are uppercase and requires that there be at least one cased character. Returns **True** if so, **False** otherwise. It is used as per the syntax :

```
<string>.isupper( )
```

Example

```
>>> string = "HELLO"
>>> string2 ="There"
>>> string3 = "goldy"
>>> string4 = "U123"      # in uppercase
>>> string5 = "123f"      # in lowercase
>>> string.isupper()
True
>>> string2.isupper()
False
>>> string3.isupper()
False
>>> string4.isupper()
True
>>> string5.isupper()
False
```

12. The lower( ) method

It returns a copy of the ***string*** converted to lowercase. It is used as per the syntax :

```
<string>.lower( )
```

Example (considering the same string values as used in the example of previous function - **isupper**)

```
>>> string.lower()
'hello'
>>> string2.lower()
'there'
>>> string3.lower()
'goldy'
>>> string4.lower()
'u123'
>>> string5.lower()
'123f'
```

13. The upper( ) method

It returns a copy of the ***string*** converted to uppercase. It is used as per the syntax :

```
<string>.upper( )
```

Example (considering the same string values as used in the example of previous function - **isupper**)

```
>>> string.upper()
'HELLO'
>>> string2.upper()
'THERE'
>>> string3.upper()
'GOLDY'
>>> string4.upper()
'U123'
>>> string5.upper()
'123F'
```

14. The lstrip( ), rstrip( ), strip( ) methods

lstrip( ) Returns a copy of the string with leading whitespaces removed, *i.e.*, whitespaces from the leftmost end are removed.

It is used as per the syntax :

```
<string>.lstrip( )
```

rstrip( ) Returns a copy of the string with trailing whitespaces removed, *i.e.*, whitespaces from the rightmost end are removed.

It is used as per the syntax :

```
<string>.rstrip( )
```

strip( ) Returns a copy of the string with leading and trailing whitespaces removed, *i.e.*, whitespaces from the leftmost and the rightmost ends are removed.

It is used as per the syntax :

```
<string>.strip( )
```

Example

```
>>> "  Sipo ".lstrip()
'Sipo '
```

*The **lstrip( )** removed only the leading whitespaces*

```
>>> "  Sipo ".rstrip()
'  Sipo'
```

*The **rstrip( )** removed only the trailing whitespaces*

```
>>> "  Sipo ".strip()
'Sipo'
```

*The **strip( )** removed both the leading and the trailing whitespaces*

15. The startswith( ), endswith( ) method

startswith( ) Returns *True* if the string starts with the substring ***sub***, otherwise returns *False*.

It is used as per the syntax :

```
<string>.startswith( )
```

endswith( ) Returns *True* if the string ends with the substring ***sub***, otherwise returns *False*. It is used as per the syntax :

```
<string>.endswith( )
```

Example

```
>>> "abcd".startswith("cd")
False
>>> "abcd".startswith("ab")
True
>>> "abcd".endswith("b")
False
>>> "abcd".endswith("cd")
True
```

16. The title( ) method

It returns a title-cased version of the string where all words start with uppercase characters and all remaining letters are in lowercase.

It is used as per the syntax :

```
<string>.title( )
```

Example

```
>>> "the sipo app".title()
'The Sipo App'
>>> "COMPUTER SCIENCE".title()
'Computer Science'
```

17. The istitle( ) method

It returns *True* if the string has the title case (*i.e.*, the first letter of each word in uppercase while rest of the letters in lowercase), *False* otherwise. It is used as per the syntax :

```
<string>.istitle( )
```

Example

```
>>> 'Computer Science'.istitle()
True
>>> "COMPUTER SCIENCE".istitle()
False
```

18. The replace( ) method

It returns a copy of the string with all occurrences of substring ***old*** replaced by ***new*** string. It is used as per the syntax :

```
<string>.replace(old, new)
```

Example

```
>>> 'abracadabra'.replace('ab', 'sp')
'spracadspra'
>>> 'I work for you'.replace('work', 'care')
'I care for you'
>>> 'you and I work for you'.replace('you', 'U')
'U and I work for U'
```

19. The join( ) method

It *joins* a string or character (*i.e.*, <str>) after each member of the *string iterator i.e., a string based sequence*. It is used as per the syntax :

```
<string>.join(<string iterable>)
```

(*i*) If the string based iterator is a string then the <str> is inserted after every character of the string, *e.g.*,

```
>>>"*".join("Hello")
'H*e*l*l*o'
```

See, a character is joined with each member of the given string to form the new string

```
>>>"***".join("TRIAL")
'T***R***I***A***L'
```

*See, a string("***" here) is joined with each member of the given string to form the new string*

(*ii*) If the string based iterator is a *list* or *tuple* of strings then, the given string/character is joined with each member of the list or tuple, BUT the tuple or list must have all member as strings otherwise Python will raise an error.

```
>>>"$$".join(["trial", "hello"])
'trial$$hello'
```

Given string ("$$") joined between the individual items of a string based list

```
>>>"###".join(("trial", "hello", "new"))
'trial###hello###new'
```

Given string ("$$") joined between the individual items of a string based tuple

```
>>>"###".join((123, "hello", "new"))
Traceback (most recent call last):
  "###".join((123, "hello", "new"))
TypeError: sequence item 0: expected str instance, int found
```

The sequence must contain all strings, else Python will raise an error.

20. The split( ) method

It *splits* a string(*i.e.*, <str>) based on given string or character (*i.e.*, <string/char>) and returns a list containing split strings as members. It is used as per the syntax :

```
<string>.split(<string/char>)
```

(*i*) If you do not provide any argument to split then by default it will split the given string considering whitespace as a separator, *e.g.*,

```
>>>"I Love Python".split()
['I', 'Love', 'Python']

>>>"I Love Python".split(" ")
['I', 'Love', 'Python']
```

With or without whitespace, the output is just the same i.e., the list containing individual words

NOTE

You can use **split( )** with a line of text to split it in words. It will be quite handy for you in some file based programs.

(*ii*) If you provide a string or a character as an argument to split( ), then the given string is divided into parts considering the given string/character as separator and separator character is not included in the split strings *e.g.*,

```
>>>"I Love Python".split("o")
['I L', 've Pyth', 'n']
```

The given string is divided from positions containing "o"

21. The partition( ) method

The **partition( )** method splits the string **at the first occurrence of separator**, and returns a tuple containing *three* items.

- The part before the separator
- The separator itself
- The part after the separator

It is used as per the syntax :

```
<string>.partition(<separator/string>)
```

Example

```
>>> txt = "I enjoy working in Python"
>>> x = txt.partition("working")
>>> print(x)
('I enjoy', 'working', 'in Python')
```

'working' is the separator string

See, it returned a tuple with 3 elements : the substring until the separator, the separator, and the substring after the separator

Table 10.4 *Difference between partition( ) vs. split( )*

The split( )'s Functioning	The partition( )'s Functioning
split( ) will split the string at any occurrence of the given argument.	partition( ) will only split the string at the first occurrence of the given argument.
It will return a **list** type containing the split substrings.	It will return a **tuple** type containing the split substrings.
The length of the list is equal to the number of words, if split on whitespaces.	It will always return a **tuple** of length 3, with the given separator as the middle value of the tuple.

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

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

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

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

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
Number of symbols : 1
```

10.11 *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
```

Program 10.12 Write a program that inputs a string that contains a decimal number and prints out the decimal part of the number. For instance, if 515.8059 is given, the program should print out 8059.

```
s = input('Enter a string (a decimal number) : ')
t = s.partition('.')
print("Given string:", s)
print("part after decimal", t[2] )
```

Sample run of the program is :

```
Enter a string (a decimal number):515.8059
Given string: 515.8059
part after decimal 8059
```

Program 10.13 Write a program that inputs individual words of your school motto and joins them to make a string. It should also input day, month and year of your school's foundation date and print the complete date.

```
print("Enter words of your school motto, one by one")
w1 = input("Enter word 1:")
w2 = input("Enter word 2:")
w3 = input("Enter word 3:")
w4 = input("Enter word 4:")
motto = " ".join( [w1, w2, w3, w4])
print("Enter your school's foundation date.")
dd = input("Enter dd:")
mm = input("Enter mm:")
yyyy = input("Enter yyyy:")
dt = "/".join( [dd, mm, yyyy] )
print("School motto :", motto )
print("School foundation date : ", dt)
```

Sample run of the program is :

```
Enter words of your school motto, one by one
Enter word 1:World
Enter word 2:is
Enter word 3:a
Enter word 4:family.
Enter your school's foundation date.
Enter dd:01
Enter mm:10
Enter yyyy:1975
School motto :  World is a family.
School foundation date :  01/10/1975
```

Check Point 10.2

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 ?

String Manipulation Functions and Methods

S.No.	Function / Method	Description
1.	`len(<string>)`	It returns the length of its argument string.
2.	`<string>.capitalize( )`	It returns a copy of the string with its first character capitalized.
3.	`<string>.count(sub[, st[, end] ] )` `st is start`	It returns the number of occurrences of the substring **sub** in string the (or string [start:end] if these arguments are given).
4.	`<string>.find(sub[, st[, end]])` `st is start`	It 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.
5.	`<string>.index(sub[, st[, end]])`	It returns the *lowest index* where the specified substring is found.
6.	`<string>.isalnum( )`	It returns **True** if the characters in the ***string*** are alphanumeric (alphabets or numbers) and there is at least one character, **False** otherwise.
7.	`<string>.isalpha( )`	It returns **True** if all characters in the ***string*** are alphabetic and there is at least one character, **False** otherwise.
8.	`<string>.isdigit( )`	It returns **True** if all the characters in the ***string*** are digits. There must be at least one character, otherwise it returns **False**.
9.	`<string>.islower( )`	It returns **True** if all cased characters in the ***string*** are lowercase. There must be at least one cased character. It returns **False** otherwise.
10.	`<string>.isspace( )`	It returns **True** if there are only whitespace characters in the ***string***. There must be at least one character. It returns **False** otherwise.
11.	`<string>.isupper( )`	It 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.
12.	`<string>.lower( )`	It returns a copy of the ***string*** converted to lowercase.
13.	`<string>.upper( )`	It returns a copy of the ***string*** converted to uppercase.
14.	`<string>.lstrip( )`	It returns a copy of the string with leading white- spaces removed.
	`<string>.rstrip( )`	It returns a copy of the string with trailing white- spaces removed.
	`<string>.strip( )`	It returns a copy of the string with leading and trailing whitespaces removed.
15.	`<string>.startswith( )`	It returns *True* if the string starts with the substring ***sub***, otherwise returns *False*.
	`<string>.endswith( )`	It returns *True* if the string ends with the substring ***sub***, otherwise returns *False*.
16.	`<string>.title( )`	It returns a title-cased version of the string where all words start with uppercase characters and all remaining letters are lowercase.
17.	`<string>.istitle( )`	It returns *True* if the string has the title case.
18.	`<string>.replace(old, new)`	It returns a copy of the string with all occurrences of substring ***old*** replaced by ***new*** string.
19.	`<string>.join(<string iterable>)`	It *joins* a string or character (*i.e.*, <str>) after each member of the *string iterator i.e.*, a *string based sequence.*
20.	`<string>.split(<string/char>)`	It *splits* a string(*i.e.*, <str>) based on the given string or character (*i.e.*, <string/char>) and returns a list containing split strings as members.
21.	`<string>.partition(<sep/string>)` `sep is seperator`	It splits the string at the first occurrence of separator, and returns a tuple containing *three* items as string till separation, separator and the string after separator.

STRING MANIPULATION

Progress In Python 10.1

PriP

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 10.1 under Chapter 10 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 : len( ), capitalize( ), find( ), isalnum( ), isalpha( ), isdigit( ), islower( ), isupper( ), lower( ), upper( ), index( ), join( ), split( ), partition( ) etc.*

OBJECTIVE TYPE QUESTIONS OTQs

MULTIPLE CHOICE QUESTIONS

1. Negative index –1 belongs to _____ of string.
   (*a*) first character (*b*) last character
   (*c*) second last character (*d*) second character
2. Which of the following is/are not legal string operators ?
   (*a*) in (*b*) + (*c*) * (*d*) /
3. Which of the following functions will return the total number of characters in a string ?
   (*a*) count( ) (*b*) index( ) (*c*) len( ) (*d*) all of these
4. Which of the following functions will return the last three characters of a string s ?
   (*a*) s[3:] (*b*) s[:3] (*c*) s[– 3:] (*d*) s[: – 3]
5. Which of the following functions will return the first three characters of a string s ?
   (*a*) s[3:] (*b*) s[:3] (*c*) s[– 3:] (*d*) s[: – 3]