Chapter 4

String Handling

Strings in Python

A string represent a group of characters. Strings are important most of the data that we use in daily life will be in the form of strings. For example, the names of persons, their addresses, company name, their debit card numbers etc. are all string. Almost every application involves working with strings, and Python's provide a string class to represent all.

Since every string comprises several characters, Python handles string and characters almost in the same manner. There is no separate datatype to represent individual characters in Python.

Creating Strings

We can create a string in Python by assigning a group of character to a variable. The groups of characters in python are surrounded by either in single, double or triple quotation mark as:

'graphic' is the same as "graphic" or '"graphic"' or """graphic"""

```
s1 = 'Graphic Era Hill University'
s2 = "Graphic Era Hill University"
s3 = '"Graphic Era Hill University"'
s4 = """Graphic Era Hill University"""
```

There is no difference between single, double and triple quotes while creating the string. These quotation marks are useful when we want to represent a string that occupies several lines as:

```
s =  "Graphic"+
        "Era" +
        "Hill"+
        "University"
```

Triple single or double quotes are useful to create strings which span into several lines.

It is also possible to display quotation marks to sub string in a string. In that case, we should use one type quotes for outer string and another type of quotes for inner string as:

```
s = "Welcome to 'GEHU' Dehradun"
print(s)
```

The preceding line of code will display the following output:

Welcome to 'GEHU' Dehradun

Escape Sequence

It is possible to use escape character like \n or \t inside the string. For example:

s = "Graphic \tEra \n Hill \n University"

The preceding line of code will display the following output:

```
Graphic    Era
Hill
University
```

An escape character gets interpreted; in a single quoted as well as double quoted strings.

Following table is a list of escape or non-printable characters that can be represented with backslash notation.

Backslash notation	Hexadecimal character	Description
\a	0x07	Bell or alert
\b	0x08	Backspace

\n	0x0a	Newline
\r	0x0d	Carriage return
\s	0x20	Space
\t	0x09	Tab
\x		Character x

To avoid the effect of escape character, we can create the string as a 'raw' string as:

```
s = r"Graphic \tEra \n Hill \n University"
print(s)
```

The preceding line of code will display the following output:

```
Graphic \tEra \n Hill \n University
```

This is not showing the effect of \t or \n. It means we could not see the horizontal tab space or new line. Raw string treat escape characters as ordinary characters in a string and hence display them as they are.

To create a string with Unicode characters, we should add 'u' at the beginning of the string. Each Unicode character contains 4 digits preceded by a \u. The following statement displays 'Core Python' in Hindi using Unicode characters. There are 8 Unicode characters used for this purpose.

```
s = u"\u0915\u094b\u0930 \u092a\u0948\u0925\u0964\u0928"
print(s)
```

कोर पैथ।न

String Special Operators

Assume string variable a holds 'Hello' and variable b holds 'Python', then –

Operator	Description	Example
+	Concatenation - Adds values on either side of the operator	a + b will give HelloPython
*	Repetition - Creates new strings, concatenating multiple copies of the same string	a*2 will give - HelloHello
[]	Slice - Gives the character from the given index	a[1] will give e
[:]	Range Slice - Gives the characters from the given range	a[1:4] will give ell
In	Membership - Returns true if a character exists in the given string	H in a will give 1
not in	Membership - Returns true if a character does not exist in the given string	M not in a will give 1
r/R	Raw String - Suppresses actual meaning of Escape characters. The syntax for raw strings is exactly the same as for normal strings with the exception of the raw string operator, the letter "r," which precedes the quotation marks. The "r" can be lowercase (r) or uppercase (R) and must be	print r'\n' prints \n and print R'\n'prints \n

	placed immediately preceding the first quote mark.	
%	Format - Performs String formatting	See at next section

Python strings are sequences of individual characters, and share their basic methods of access with those other Python sequences – lists and tuples. The simplest way of extracting single characters from strings (and individual members from any sequence) is to unpack them into corresponding variables.

```
>>> s = 'Era'
>>> s
'Era'
>>> a, b, c = s # Unpack into variables
>>> a
'E'
>>> b
'r'
>>> c
'a'
```

Unfortunately, it's not often that we have the luxury of knowing in advance how many variables we are going to need in order to store every character in the string. And if the number of variables we supply doesn't match with the number of characters in the string, Python will give us an error.

```
s = 'Graphic Era'
a, b, c = s
Traceback (most recent call last):
File "", line 1, in
ValueError: too many values to unpack
```

Accessing characters in strings by index in Python

Typically it's more useful to access the individual characters of a string by using Python's array-like indexing syntax. Index represents the position number written using square braces []. By specifying the position number through an index, we can refer to the individual elements (or characters) of a

string. For Example, s[0] refers to the 0th element of the string and s[1] refers to the 1st element of the string.

We can use index as a negative number in python, it refers to elements in reverse order. Thus s[-1] refers to the last element and s[-2] refers to second element from the last.

S = “Graphic Era”

G	R	A	P	H	I	C		E	R	A	
0	1	2	3	4	5	6	7	8	9	10	11
G	R	A	P	H	I	C		E	R	A	
-12	-11	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1

```
>>> s = 'Graphic Era'
>>> s[4] # Get the 5th character
'i'
```

If you want to start counting from the end of the string, instead of the beginning, use a negative index. For example, an index of -1 refers to the right-most character of the string.

```
>>> s[-1]
'a'
>>> s[-7]
'h'
```

Python strings are immutable, which is just a fancy way of saying that once they've been created, you can't change them. Attempting to do so triggers an error.

```
>>> s[6]
'c'
>>> s[6] = 'x'
Traceback (most recent call last):
File "", line 1, in
TypeError: 'str' object does not support item assignment
```

We will take an ambiguous example. In this example, we are creating two strings 's1' and 's2' as:

s1 = "Good"

s2 = "Morning"

Now, we are modifying the content of the string 's2' by storing the content of 's1' into it as:

s2 = s1 #store reference of s1 into s2

if we display, the 's2' string, we can see the same content of 's1'.

print(s1) # display Good

print(s2) # display Good

Identity number of an object internally refers to the memory address of the object and is given by id() function. If we display identity numbers using id() function, we can find that the identity number of 's2' and 's1' are same since they refer to one and the same object.

```
s1 = "Good"
s2 = "Morning"
print(id(s1) # display 30484852
print(id(s2) # display 50784890
s2 = s1
print(id(s1) # display 30484852
print(id(s2) # display 30484852
```

If you want to modify a string, you have to create it as a totally new string.

Program1: A Python program to access each element of a string in forward and reverse orders using while loop.

```
#indexing on strings
str = "Core Python"
n = len(str)
i=0
#acces each character using while loop
while i<n:
    print(str[i], end = '')
    i+=1
print()
```

```
#access in reverse order
i =-1
while(i>=-n):
    print(str[i], end='')
    i-=1
print()

#reverse order using negative index
i=1
n = len(str)
while i<=n:
    print(str[-i],end='')
    i+=1
```

Output:

```
Core Python
nohtyP eroC
nohtyP eroC
```

Slicing the Strings

Slice represents a part or piece of a string. Before that, what if you want to extract a piece of more than one character, with known position and size? That's fairly easy and intuitive. We extend the square-bracket syntax a little, so that we can specify not only the starting position of the piece we want, but also where it ends. The format of slicing is:

Stringname[start: stop: stepsize]

If 'start and 'stop' are not specified, then slicing is done from 0th to n-1th elements. If 'stepsize' is not written, then it is taken to be 1.If 'start' and 'stop' are not specified, then it is taken from 0th to n-1th elements. If 'stepsize' is not written, then it is taken to be 1.

See the following example

S = "Graphic Era Hill University"

G	R	A	P	H	I	C		E	R	A	

0	1	2	3	4	5	6	7	8	9	10	11
G	R	A	P	H	I	C		E	R	A	
-12	-11	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1

```
s = "Graphic Era"
print(s[::])
print(s[::-1])
print(s[0:7:1])
print(s[0:7:])
print(s[0::])
print(s[0:7:-1]) # nothing will be print
print(s[-3:-1:1])
print(s[-1:-3:-1])
print(s[-1:-3:1]) # nothing will be print
print(s[-1:-3:-1])
print(s[-3::])
```

Output:

```
Graphic Era
arE cihparG
Graphic
Graphic
Graphic Era

Er
ar

ar
Era
```

Repeating the String

The repetition operator is denoted by '*' symbol and is useful to repeat the string for several times. For example:

```
s = "Graphic Era"
print(s*2)
```

The preceding line of code will display the following output:

```
Graphic EraGraphicEra
```

Similarly , it is possible to repeat a part of the string obtained by slicing as:

```
s = s[8::]*3
print(s)
```

display **EraEraEra**

Concatenation of Strings

The + operator concatenates strings. It returns a string consisting of the operands joined together, as shown here:

```
s1 = "Graphic "
s2 = "Era"
s3 = s1 + s2
print(s3)  # display Graphic Era
```

Comparing String

We can use relational operators like >,>=,<, <=, ==, != operator to compare two strings. They return Boolean value, i.e. either True or False depending on the strings being compared.

```
s1 = "Graphic"
s2 = "Graphik"
if(s1==s2):
    print('Both are same')
else:
    print('Not same')
```

This code returns 'Not same' as the strings are not same. While comparing Python string interpreter compares them by taking them in English dictionary order. The string which comes first in the dictionary order will have a low value than the string which comes next.

Removing spaces from a String

A space is also considered as a character inside a string. Sometimes, the unwanted spaces in a string will lead to wrong results. For example, a person types his institution name 'GEHU ' (observer single space at the end of the string) instead of typing 'GEHU'. If we compare these two strings using '==' operator as

```
If "Gehu " =="GEHU" :
        print('Welcome')
else:
        print(' Institute Not Found')
```

The Output will be 'Institute Not Found'. In this way, spaces may lead to wrong results. Hence such space must be removed from the strings before they are compared. This is possible using rstrip(),lstrip() and strip() functions.

The rstrip() method removes the spaces which are at the right side of the string. The The lstrip() method removes the spaces which are at the left side of the string. The strip() function removes spaces from both sides of the strings. For example:

```
s = "    Graphic Era   "

print(s.rstrip())  # remove spaces at right

print(s.lstrip())  # remove spaces at left
```

Now if we write:

```
print(s.strip())  # remove spaces from both side but do not remove

                  #spaces which are in the middle of the string
```

Replacing a String with another String

The method replace() returns a copy of the string in which the occurrences of old have been replaced with new, optionally restricting the number of replacements to max.

Syntax

Following is the syntax for replace() method −

```
str.replace(old, new[, max])
```

Parameters

- old − This is old substring to be replaced.
- new − This is new substring, which would replace old substring.
- max − If this optional argument max is given, only the first count occurrences are replaced.

Example

```
str = "this is string example....wow!!! this is really string"

print str.replace("is", "was")

print str.replace("is", "was", 3)
```

Output:

```
thwas was string example....wow!!! thwas was really string
thwas was string example....wow!!! thwas is really string
```

Finding Sub Strings

The find(), rfind(), index() and rindex() methods are useful to locate sub string in a string. These methods return the location of the first occurrence of the sub string in the main string. The find() and index() methods search for the sub string from beginning of the main string. The rfind() and rindex() methods search for the sub string from right to left.

The find() method returns -1 if the sub string is not found in the main string. The index() method returns 'ValueError' exception if the sub string is not found. The format of find() method is:

mainstring.find(str, beg,end)

Parameters

- **str** – This specifies the string to be searched.
- **beg** – This is the starting index, by default its 0.
- **end** – This is the ending index, by default its equal to the length of the string.

The same format is used for other method also.

Program2: A Python program to find the first occurrence of sub string in a main string

```
#to find first occurrence of sub string in a main string
mstr = input("Enter main string: ")
substr = input("Enter sub string: ")
n = mstr.find(substr,0,len(mstr))
if n==-1:
    print("Sub string not find")
else:
    print("Sub String found at position:", n+1)
```

Output:

```
Enter main string: This is a book
Enter sub string: is
Sub String found at position: 3
```

Same program can be rewritten using index() method. If the substring is not found index() method returns 'ValueError' exception.

```
#to find first occurrence of sub string in a main string
mstr = input("Enter main string: ")
substr = input("Enter sub string: ")
try:
        n = mstr.find(substr,0,len(mstr))
except ValueError:
        print("Sub string not find")
else:
    print("Sub String found at position:", n+1)
```

Output:

```
Enter main string: This is a book
Enter sub string: is
Sub String found at position: 3
```

The find () method and index() methods return only the first occurrence of the sub string. When the sub string occurs several times in the main string, they cannot return all those occurrences. Is there any way that we can find out all the occurrence of sub string. For this purpose, we should develop additional logic.

Initially, searching should start from 0^{th} character in the main 'mstr' up to the last character 'n' which is given by len(mstr). So 'I' value will start initially at 0. When the find() method finds the position of the sub string, we should display it. Suppose the sub string is found at 2^{nd} position, then we need not again search for the sub string up to the 2^{nd} position. This time, we should continue searching from 3^{rd} character onwards. This logic is used in **program 3**:

Program3: A Python program to find the first occurrence of sub string in a main string

```
#to find first occurrence of sub string in a main string
mstr = input("Enter main string: ")
substr = input("Enter sub string: ")
i=0
flag = False
n = len(mstr)
while i<n:
    pos = mstr.find(substr,i,n)
    if pos !=-1:
        print("Found at position: ", pos+1)
        i=pos+1
        flag=True
    else:
        i=i+1
if flag == False:
    print("Sub String not found")
```

Output:

```
Enter main string: This is a book
Enter sub string: is
Found at position:  3
Found at position:  6
```

Replacing a String with another String

The method **replace()** returns a copy of the string in which the occurrences of *old* have been replaced with *new*, optionally restricting the number of replacements to *max*.

Syntax

Following is the syntax for **replace()** method –

```
str.replace(old, new[, max])
```

arameters

- **old** – This is old substring to be replaced.
- **new** – This is new substring, which would replace old substring.
- **max** – If this optional argument max is given, only the first count occurrences are replaced.

Example

```
str = "this is string example....wow!!! this is really string"

print str.replace("is", "was")

print str.replace("is", "was", 3)
```

Output:

```
thwas was string example....wow!!! thwas was really string
thwas was string example....wow!!! thwas is really string
```

Splitting and Joining String

The split() method is used to brake a string into pieces. These pieces are returned as a list. For example,

```
a = "this is a string"
a = a.split(" ") # a is converted to a list of strings.
print a
```

output:
['this', 'is', 'a', 'string']

Joining a string is simple:

```
a = "-".join(a)
print a
```

this-is-a-string

Example

```
#split of string
str1=input("Enter first String with space :: ")
print(str1.split())    #splits at space

str2=input("Enter second String with (,) :: ")
print(str2.split(','))    #splits at ','

str3=input("Enter third String with (:) :: ")
print(str3.split(':'))    #splits at ':'
```

```
str4=input("Enter fourth String with (;) :: ")
print(str4.split(';'))    #splits at ';'


str5=input("Enter fifth String without space :: ")
print([str5[i:i+2]for i in range(0,len(str5),2)])    #splits at position 2
```

Output

```
Enter first String with space :: python program
['python', 'program']
Enter second String with (,) :: python, program
['python', 'program']
Enter third String with (:) :: python: program
['python', 'program']
Enter fourth String with (;) :: python; program
['python', 'program']
Enter fifth String without space :: python program
['py', 'th', 'on', 'pr', 'og', 'ra', 'm']
```

Splitting on a Specific Substring

By providing an optional parameter, .split('x') can be used to split a string on a specific substring 'x'. Without 'x' specified, .split() simply splits on all whitespace, as seen above.

```
>>> mary = 'Mary had a little lamb'
>>> mary.split('a')                # splits on 'a'
['M', 'ry h', 'd ', ' little l', 'mb']
>>> hi = 'Hello mother,\nHello father.'
>>> print(hi)
Hello mother,
Hello father.
>>> hi.split()              # no parameter given: splits on whitespace
['Hello', 'mother,', 'Hello', 'father.']
>>> hi.split('\n')              # splits on '\n' only
['Hello mother,', 'Hello father.']
```

Joining a List of Strings: .join()

If you have a list of words, how do you put them back together into a single string? .join() is the method to use. Called on a "separator" string 'x', 'x'.join(y) joins every element in the list y separated by 'x'. Below, words in mwords are joined back into the sentence string with a space in between:

```
>>> mwords
['Mary', 'had', 'a', 'little', 'lamb']
>>> ' '.join(mwords)
'Mary had a little lamb'
```

Joining can be done on any separator string. Below, '--' and the tab character '\t' are used.

```
>>> '--'.join(mwords)
'Mary--had--a--little--lamb'
>>> '\t'.join(mwords)
'Mary\thad\ta\tlittle\tlamb'
>>> print('\t'.join(mwords))
Mary    had     a       little  lamb
```

The method can also be called on the empty string '' as the separator. The effect is the elements in the list joined together with nothing in between. Below, a list of characters is put back together into the original string:

```
>>> hi = 'hello world'
>>> hichars = list(hi)
>>>print( hichars)
['h', 'e', 'l', 'l', 'o', ' ', 'w', 'o', 'r', 'l', 'd']
>>> ''.join(hichars)
'hello world'
```

Checking starting and Ending of a String

The startwith() method is useful to know whether a string is starting with a sub string or not.

When the sub string is found in the main string 'str' , this method returns True. If the sub string is not found, it returns false. For example:

```
>>> 'school'.startswith('s')
True
>>> 'school'.startswith('scho')
True
>>> 'school'.startswith('k')
False
```

On the other side

```
>>> 'school'.endswith('ol')
True
>>> 'school'.endswith('l')
True
>>> 'school'.endswith('ll')
False
```

What about substrings in the middle? Well, strings do not have a designated method, but the versatile in operator can be used here. Because in is not a string METHOD, it uses a different syntax of y in x:

```
>>> 'oo' in 'school'
True
>>> 'h' in 'school'
True
>>> 'scho' in 'school'          # matches string-initially
True
>>> 'school' in 'school'        # whole substring
True
>>> 'k' in 'school'
False
>>> 'ee' in 'school'
False
```

Counting Substring

.count() returns the number of occurrences of a given substring. The return value is an integer:

```
>>> 'colorless green ideas sleep furiously'.count('s')
5
>>> 'colorless green ideas sleep furiously'.count('ee')
2
```

Case Manipulators

.upper() is an uppercasing method. It returns a new string where every character is uppercased. .lower() does the opposite: it returns a new string where every character is lowercased.

```
>>> 'Do not enter'.upper()
'DO NOT ENTER'
>>> 'DO NOT ENTER'.lower()
'do not enter'
```

.capitalize() turns only the very first character in the entire string into an uppercase letter, while .title() returns a string where every word starts with an uppercase letter:

```
>>> 'colorless green ideas'.capitalize()
'Colorless green ideas'
>>> 'colorless green ideas'.title()
'Colorless Green Ideas'
```

Transforming a String

.replace() takes two parameters. When called like x.replace(y,z), a new string is returned where all instances of y in x are replaced by z.

```
>>> mary = 'Mary had a little lamb'
>>> mary.replace('a', 'xx')
'Mxxry hxxd xx little lxxmb'
```

.replace() can be used to delete substrings by specifying the empty string '' as the second parameter:

```
>>> chom = 'Colorless green ideas sleep furiously'
>>> chom.replace('ee', '')
'Colorless grn ideas slp furiously'
>>> chom.replace(' ', '')
'Colorlessgreenideassleepfuriously'
```

Stacked Application

When a string method returns a string type, another string method can be tacked on. Below, .replace() method is called repeatedly in order to remove all "vowel" characters:

```
>>> chom = 'Colorless green ideas sleep furiously'
>>> chom.replace('a', '')
'Colorless green ides sleep furiously'
>>> chom.replace('a', '').replace('e', '').replace('i','').replace('o', '').replace('u', '')
'Clrlss grn ds slp frsly'
```

Formatting the String

Formatting a string presenting the string in a clearly understandable manner.

Python uses two ways to format a string.

First C-style string formatting to create new, formatted strings. The "%" operator is used to format a set of variables enclosed in a "tuple" (a fixed size list), together with a format string, which contains normal text together with "argument specifiers", special symbols like "%s" and "%d".

Let's say you have a variable called "name" with your user name in it, and you would then like to print(out a greeting to that user.)

This prints out "Hello, Amit!"

```
name = "Amit"
print("Hello, %s!" % name)


# This prints out "Amit is 20 years old."
name = "Amit"
age = 20
print("%s is %d years old." % (name, age))
```

Any object which is not a string can be formatted using the %s operator as well. The string which returns from the "repr" method of that object is formatted as the string. For example:

```
# This prints out: A list: [1, 2, 3]
mylist = [1,2,3]
print("A list: %s" % mylist)
```

Here is the list of complete set of symbols which can be used along with % –

Sr.No.	Format Symbol & Conversion
1	**%c** character
2	**%s** string conversion via str() prior to formatting
3	**%i** signed decimal integer
4	**%d** signed decimal integer
5	**%u** unsigned decimal integer

6	**%o** octal integer
7	**%x** hexadecimal integer (lowercase letters)
8	**%X** hexadecimal integer (UPPERcase letters)
9	**%e** exponential notation (with lowercase 'e')
10	**%E** exponential notation (with UPPERcase 'E')
11	**%f** floating point real number
12	**%g** the shorter of %f and %e
13	**%G** the shorter of %f and %E

Format() Method

The format() is another way to format string. This method is used as:

.'format string with replacement fields' . format(values)

For example create a string by name ,str' to display 3 values. These 3 values or variables should be mentioned inside the format() method as :

Id =18

name = “Amit”

Sal =59000.75

str = ‘{ } , { }, { }’. format(id, name,sal)

print(str) # display 18, Amit,59000.75

str = ‘{ } - { }- { }’. format(id, name,sal)

print(str) # display 18- Amit-59000.75

We can also display message string before every value, which can be done as:

str = ‘Id={ }\n Name= { }\n Salary= { }’. format(id, name,sal)

print(str)

Output:

Id = 18

Name = Amit

Salary=59000.75

String Testing Methods

There are several methods to test the nature of characters in a string. These method either return True or False. Below is a list of similar methods and examples. They all return True/False.

Method	Is every character ...?	Example
.isalpha()	alphabetic (A-Z, a-z)	'iPad'.isalpha() True >>> 'co-operate'.isalpha() False

.isalnum()	alpha-numeric (A-Z, a-z, 0-9)	>>> 'Homework3'.isalnum()
		True
		>>> 'ice cream'.isalnum()
		False

.isdigit()	numeric (0-9)	>>> '2014'.isdigit()
		True
		>>> '9:30'.isdigit()
		False

.isspace()	whitespace character:
	' ', '\n', '\t', '\r'
		>>> '\n\t \r \n'.isspace()
		True
		>>> ' A b c 12 '.isspace()
		False

.islower()	lowercase (a-z)	>>> 'apple'.islower()
		True
		>>> 'iPad'.islower()
		False

.isupper()	uppercase (A-Z)	>>> 'HELLO'.isupper()
		True
		>>> 'Hello'.isupper()
		False

Sorting Strings

We can sort a group of string into alphabetical order using sort() method and sorted method() function. The sort() method is used in the following way:

str.sort()

Here 'str' represents an array that contains a group of strings. When sort() method is used, it will sort the original array, i.e. 'str'. So the original order of the strings will be lost and we will have only one sorted array. To retain the original array even after sorting we can use sorted() function as:

str1 = sorted(str)

where:

str= orginal array

str1= sorted array

Program 4: A program to sort a group of strings into alphabetical order.

```
# sorting a group of strings
str = []
n = int(input("How many strings? "))
for i in range(n):
    print("Enter string: ", end='')
    str.append(input())

str.sort()
str1 = sorted(str)
print("Sorted List: ")
for i in str1:
    print(i)
```

Output:

```
How many strings? 5
Enter string: Delhi
Enter string: Chandigarh
Enter string: Kolkatta
Enter string: Chennai
Enter string: Dehradun
Sorted List:
Chandigarh
Chennai
Dehradun
Delhi
Kolkatta
```

Searching in the Strings

```
for i in range(len(str)):
        If s ==str[i]:
```

Program 5: To search for the position of a string in a given group of strings

```
# Searching for a string in a group of strings
str = []
n = int(input("How many strings? "))
```

```
for i in range(n):
    print("Enter string: ", end='')
    str.append(input())
s = input("Enter String to search: ")
flag = False
for i in range(len(str)):
    if s== str[i]:
        print("Found at: ", i+1)
        flag = True
if flag == False:
    print("Not Found")
```

Output:

```
How many strings? 5
Enter string: Amit
Enter string: Suraj
Enter string: Sushant
Enter string: Zafar
Enter string: Navin
Enter String to search: Zafar
Found at:  4
```

Other supported symbols and functionality are listed in the following table −

Sr.No.	Symbol & Functionality
1	* argument specifies width or precision
2	- left justification

3	**+** display the sign
4	**<sp>** leave a blank space before a positive number
5	**#** add the octal leading zero ('0') or hexadecimal leading '0x' or '0X', depending on whether 'x' or 'X' were used.
6	**0** pad from left with zeros (instead of spaces)
7	**%** '%%' leaves you with a single literal '%'
8	**(var)** mapping variable (dictionary arguments)
9	**m.n.** m is the minimum total width and n is the number of digits to display after the decimal point (if appl.)

Some Other Built-in String Methods

Python includes the following built-in methods to manipulate strings −

Sr.No.	Methods & Description

1	**capitalize()** Capitalizes first letter of string
2	**center(width, fillchar)** Returns a string padded with *fillchar* with the original string centered to a total of *width* columns.
3	**count(str, beg = 0,end = len(string))** Counts how many times str occurs in string or in a substring of string if starting index beg and ending index end are given.
4	**decode(encoding = 'UTF-8',errors = 'strict')** Decodes the string using the codec registered for encoding. encoding defaults to the default string encoding.
5	**encode(encoding = 'UTF-8',errors = 'strict')** Returns encoded string version of string; on error, default is to raise a ValueError unless errors is given with 'ignore' or 'replace'.
6	**endswith(suffix, beg = 0, end = len(string))** Determines if string or a substring of string (if starting index beg and ending index end are given) ends with suffix; returns true if so and false otherwise.
7	**expandtabs(tabsize = 8)** Expands tabs in string to multiple spaces; defaults to 8 spaces per tab if tabsize not provided.
8	**find(str, beg = 0 end = len(string))** Determine if str occurs in string or in a substring of string if starting index beg and ending index end are

	given returns index if found and -1 otherwise.
9	**index(str, beg = 0, end = len(string))** Same as find(), but raises an exception if str not found.
10	**isalnum()** Returns true if string has at least 1 character and all characters are alphanumeric and false otherwise.
11	**isalpha()** Returns true if string has at least 1 character and all characters are alphabetic and false otherwise.
12	**isdigit()** Returns true if string contains only digits and false otherwise.
13	**islower()** Returns true if string has at least 1 cased character and all cased characters are in lowercase and false otherwise.
14	**isnumeric()** Returns true if a unicode string contains only numeric characters and false otherwise.
15	**isspace()** Returns true if string contains only whitespace characters and false otherwise.
16	**istitle()** Returns true if string is properly "titlecased" and false otherwise.
17	**isupper()**

	Returns true if string has at least one cased character and all cased characters are in uppercase and false otherwise.
18	**join(seq)** Merges (concatenates) the string representations of elements in sequence seq into a string, with separator string.
19	**len(string)** Returns the length of the string
20	**ljust(width[, fillchar])** Returns a space-padded string with the original string left-justified to a total of width columns.
21	**lower()** Converts all uppercase letters in string to lowercase.
22	**lstrip()** Removes all leading whitespace in string.
23	**maketrans()** Returns a translation table to be used in translate function.
24	**max(str)** Returns the max alphabetical character from the string str.
25	**min(str)** Returns the min alphabetical character from the string str.
26	**replace(old, new [, max])** Replaces all occurrences of old in string with new or

	at most max occurrences if max given.
27	**rfind(str, beg = 0,end = len(string))** Same as find(), but search backwards in string.
28	**rindex(str, beg = 0, end = len(string))** Same as index(), but search backwards in string.
29	**rjust(width,[, fillchar])** Returns a space-padded string with the original string right-justified to a total of width columns.
30	**rstrip()** Removes all trailing whitespace of string.
31	**split(str="", num=string.count(str))** Splits string according to delimiter str (space if not provided) and returns list of substrings; split into at most num substrings if given.
32	**splitlines(num=string.count('\n'))** Splits string at all (or num) NEWLINEs and returns a list of each line with NEWLINEs removed.
33	**startswith(str, beg=0,end=len(string))** Determines if string or a substring of string (if starting index beg and ending index end are given) starts with substring str; returns true if so and false otherwise.
34	**strip([chars])** Performs both lstrip() and rstrip() on string
35	**swapcase()** Inverts case for all letters in string.

36	**title()** Returns "titlecased" version of string, that is, all words begin with uppercase and the rest are lowercase.
37	**translate(table, deletechars="")** Translates string according to translation table str(256 chars), removing those in the del string.
38	**upper()** Converts lowercase letters in string to uppercase.
39	**zfill (width)** Returns original string leftpadded with zeros to a total of width characters; intended for numbers, zfill() retains any sign given (less one zero).
40	**isdecimal()** Returns true if a unicode string contains only decimal characters and false otherwise.

Some Practice Questions

Python | Extract numbers from string

Many times, while working with strings we come across this issue in which we need to get all the numeric occurrences. This type of problem generally occurs in competitive programming and also in web development. Let's discuss certain ways in which this problem can be solved.

Method #1 : Using List comprehension + isdigit() + split()

This problem can be solved by using split function to convert string to list and then the list comprehension which can help us iterating through the list and isdigit function helps to get the digit out of a string.

```
# Python3 code to demonstrate
# getting numbers from string
# using List comprehension + isdigit() +split()

# initializing string
test_string = "There are 2 apples for 4 persons"

# printing original string
print("The original string : " + test_string)

# using List comprehension + isdigit() +split()
# getting numbers from string
res = [int(i) for i in test_string.split() if i.isdigit()]

# print result
print("The numbers list is : " + str(res))
```

Output :

```
The original string : There are 2 apples for 4 persons

The numbers list is : [2, 4]
```

Method #2 : Using re.findall()

This particular problem can also be solved using python regex, we can use the findall function to check for the numeric occurrences using matching regex string.

```
# Python3 code to demonstrate
# getting numbers from string
# using re.findall()
```

```
import re

# initializing string
test_string = "There are 2 apples for 4 persons"

# printing original string
print("The original string : " + test_string)

# using re.findall()
# getting numbers from string
temp = re.findall(r'\d+', test_string)
res = list(map(int, temp))

# print result
print("The numbers list is : " + str(res))
```

Output :

```
The original string : There are 2 apples for 4 persons

The numbers list is : [2, 4]
```