In Python, a **string** is an immutable sequence of Unicode characters. Each character has a unique numeric value as per the UNICODE standard. But, the sequence as a whole, doesn't have any numeric value even if all the characters are digits. To differentiate the string from numbers and other identifiers, the sequence of characters is included within single, double or triple quotes in its literal representation. Hence, 1234 is a number (integer) but '1234' is a string.

Creating Python Strings

As long as the same sequence of characters is enclosed, **single** or **double** or **triple** quotes don't matter. Hence, following string representations are equivalent.

Example

```
>>> 'Welcome To TutorialsPoint'
'Welcome To TutorialsPoint'
>>> "Welcome To TutorialsPoint"
'Welcome To TutorialsPoint'
>>> '''Welcome To TutorialsPoint'''
'Welcome To TutorialsPoint'
>>> """Welcome To TutorialsPoint"""
'Welcome To TutorialsPoint'
```

Looking at the above statements, it is clear that, internally Python stores strings as included in single quotes.

In older versions strings are stored internally as 8-bit ASCII, hence it is required to attach 'u' to make it Unicode. Since Python 3, all strings are represented in Unicode. Therefore, It is no longer necessary now to add 'u' after the string.

Accessing Values in Strings

Python does not support a character type; these are treated as strings of length one, thus also considered a substring.

To access substrings, use the square brackets for slicing along with the index or indices to obtain your substring. For example −

Open Compiler

```
var1 = 'Hello World!'
var2 = "Python Programming"

print ("var1[0]: ", var1[0])
print ("var2[1:5]: ", var2[1:5])
```

When the above code is executed, it produces the following result −

```
var1[0]:  H
var2[1:5]:  ytho
```

Updating Strings

You can "update" an existing string by (re)assigning a variable to another string. The new value can be related to its previous value or to a completely different string altogether. For example −

Open Compiler

```
var1 = 'Hello World!'
print ("Updated String :- ", var1[:6] + 'Python')
```

When the above code is executed, it produces the following result −

```
Updated String :-  Hello Python
```

Visit our Python - Modify Strings tutorial to know more about updating/modifying strings.

Escape Characters

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

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

Backslash notation	Hexadecimal character	Description
\a	0x07	Bell or alert
\b	0x08	Backspace
\cx		Control-x
\C-x		Control-x
\e	0x1b	Escape
\f	0x0c	Formfeed
\M-\C-x		Meta-Control-x
\n	0x0a	Newline
\nnn		Octal notation, where n is in the range 0.7
\r	0x0d	Carriage return
\s	0x20	Space
\t	0x09	Tab
\v	0x0b	Vertical tab
\x		Character x
\xnn		Hexadecimal notation, where n is in the range 0.9, a.f, or A.F

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 placed immediately preceding the first quote mark.	print r'\n' prints \n and print R'\n'prints \n
%	Format - Performs String formatting	See at next section

String Formatting Operator

One of Python's coolest features is the string format operator %. This operator is unique to strings and makes up for the pack of having functions from C's printf() family. Following is a simple example –

Open Compiler

```
print ("My name is %s and weight is %d kg!" % ('Zara', 21))
```

When the above code is executed, it produces the following result –

```
My name is Zara and weight is 21 kg!
```

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

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.)

Visit our Python - String Formatting tutorial to learn about various ways to format strings.

Double Quotes in Python Strings

You want to embed some text in double quotes as a part of string, the string itself should be put in single quotes. To embed a single quoted text, string should be written in double quotes.

Example

Open Compiler

```
var = 'Welcome to "Python Tutorial" from TutorialsPoint'
print ("var:", var)

var = "Welcome to 'Python Tutorial' from TutorialsPoint"
print ("var:", var)
```

It will produce the following **output** –

```
var: Welcome to "Python Tutorial" from TutorialsPoint
var: Welcome to 'Python Tutorial' from TutorialsPoint
```

Triple Quotes

To form a string with triple quotes, you may use triple single quotes, or triple double quotes – both versions are similar.

Example

Open Compiler

```
var = '''Welcome to TutorialsPoint'''
print ("var:", var)
```

```
var = """Welcome to TutorialsPoint"""
print ("var:", var)
```

It will produce the following **output** –

```
var: Welcome to TutorialsPoint
var: Welcome to TutorialsPoint
```

Python Multiline Strings

Triple quoted string is useful to form a multi-line string.

Example

Open Compiler

```
var = '''
Welcome To
Python Tutorial
from TutorialsPoint
'''
print ("var:", var)
```

It will produce the following **output** –

```
var:
Welcome To
Python Tutorial
from TutorialsPoint
```

Arithmetic Operators with Strings

A string is a non-numeric data type. Obviously, we cannot use arithmetic operators with string operands. Python raises TypeError in such a case.

Open Compiler

```
print ("Hello"-"World")
```

On executing the above program it will generate the following error –

```
>>> "Hello"-"World"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for -: 'str' and 'str'
```

Getting Type of Python Strings

A string in Python is an object of str class. It can be verified with type() function.

Example

Open Compiler

```
var = "Welcome To TutorialsPoint"
print (type(var))
```

It will produce the following **output** –

```
<class 'str'>
```

Built-in String Methods

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

Sr.No.	Methods with Description
1	**capitalize()** Capitalizes first letter of string.
2	**casefold()** Converts all uppercase letters in string to lowercase. Similar to lower(), but works on UNICODE characters alos.

3	**center(width, fillchar)** Returns a space-padded string with the original string centered to a total of width columns.
4	**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.
5	**decode(encoding='UTF-8',errors='strict')** Decodes the string using the codec registered for encoding. encoding defaults to the default string encoding.
6	**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'.
7	**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.
8	**expandtabs(tabsize=8)** Expands tabs in string to multiple spaces; defaults to 8 spaces per tab if tabsize not provided.
9	**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.
10	**format(*args, **kwargs)** This method is used to format the current string value.
11	**format_map(mapping)** This method is also use to format the current string the only difference is it uses a mapping object.
12	**index(str, beg=0, end=len(string))** Same as find(), but raises an exception if str not found.

13	**isalnum()** Returns true if string has at least 1 character and all characters are alphanumeric and false otherwise.
14	**isalpha()** Returns true if string has at least 1 character and all characters are alphabetic and false otherwise.
15	**isascii()** Returns True is all the characters in the string are from the ASCII character set.
16	**isdecimal()** Returns true if a unicode string contains only decimal characters and false otherwise.
17	**isdigit()** Returns true if string contains only digits and false otherwise.
18	**isidentifier()** Checks whether the string is a valid Python identifier.
19	**islower()** Returns true if string has at least 1 cased character and all cased characters are in lowercase and false otherwise.
20	**isnumeric()** Returns true if a unicode string contains only numeric characters and false otherwise.
21	**isprintable()** Checks whether all the characters in the string are printable.
22	**isspace()** Returns true if string contains only whitespace characters and false otherwise.
23	**istitle()** Returns true if string is properly "titlecased" and false otherwise.

24	**isupper()** Returns true if string has at least one cased character and all cased characters are in uppercase and false otherwise.
25	**join(seq)** Merges (concatenates) the string representations of elements in sequence seq into a string, with separator string.
26	**ljust(width[, fillchar])** Returns a space-padded string with the original string left-justified to a total of width columns.
27	**lower()** Converts all uppercase letters in string to lowercase.
28	**lstrip()** Removes all leading white space in string.
29	**maketrans()** Returns a translation table to be used in translate function.
30	**partition()** Splits the string in three string tuple at the first occurrence of separator.
31	**removeprefix()** Returns a string after removing the prefix string.
32	**removesuffix()** Returns a string after removing the suffix string.
33	**replace(old, new [, max])** Replaces all occurrences of old in string with new or at most max occurrences if max given.
34	**rfind(str, beg=0,end=len(string))** Same as find(), but search backwards in string.
35	**rindex(str, beg=0, end=len(string))** Same as index(), but search backwards in string.

36	**rjust(width,[, fillchar])** Returns a space-padded string with the original string right-justified to a total of width columns.
37	**rpartition()** Splits the string in three string tuple at the ladt occurrence of separator.
38	**rsplit()** Splits the string from the end and returns a list of substrings.
39	**rstrip()** Removes all trailing whitespace of string.
40	**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.
41	**splitlines(num=string.count('\n'))** Splits string at all (or num) NEWLINEs and returns a list of each line with NEWLINEs removed.
42	**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.
43	**strip([chars])** Performs both lstrip() and rstrip() on string.
44	**swapcase()** Inverts case for all letters in string.
45	**title()** Returns "titlecased" version of string, that is, all words begin with uppercase and the rest are lowercase.

46	**translate(table, deletechars="")** Translates string according to translation table str(256 chars), removing those in the del string.
47	**upper()** Converts lowercase letters in string to uppercase.
48	**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).

Built-in Functions with Strings

Following are the built-in functions we can use with strings –

Sr.No.	Function with Description
1	**len(list)** Returns the length of the string.
2	**max(list)** Returns the max alphabetical character from the string str.
3	**min(list)** Returns the min alphabetical character from the string str.