

Operations on String

Hands-on Guide

edureka!

edureka!

© 2014 Brain4ce Education Solutions Pvt. Ltd.

Operations on String

Hands-on Guide

Table of Contents

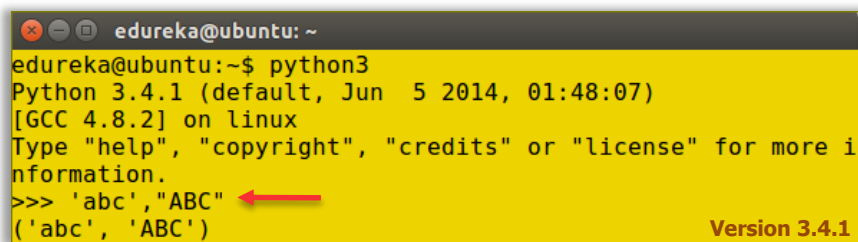
| | |
|---|----|
| 1. Most Commonly Used Operations on String..... | 2 |
| a. Concatenation and repetition | 3 |
| b. Indexing and slicing Operation..... | 3 |
| c. Membership Checking..... | 5 |
| d. String Formatting Operator: %..... | 6 |
| e. Escape Sequences and Line continuation | 6 |
| 2. Extensive List of Operations on String..... | 7 |
| a. Accessing Values in Strings..... | 7 |
| b. Updating Strings | 7 |
| c. Escape Characters | 8 |
| d. String Special Operators..... | 9 |
| e. String Formatting Operator..... | 10 |
| f. Triple Quotes..... | 11 |
| g. Unicode String | 13 |

Most Commonly Used Operations on String

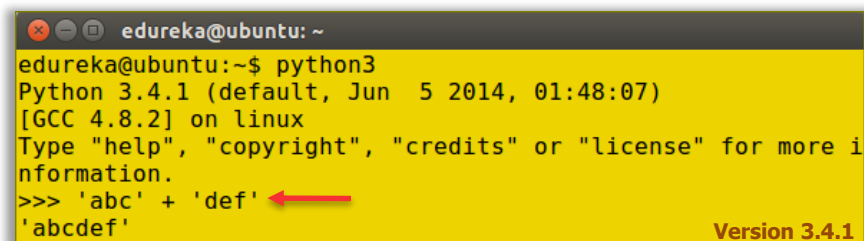
Python does not support a character type; these are treated as strings of length one, thus also considered a substring.

- Strings are Immutable.
- Strings are ordered blocks of text
- Strings are enclosed in single or double quotation marks.
- Double quotation marks allow the user to extend strings over multiple lines without backslashes, which usually signal the continuation of an expression.

Examples: 'abc', "ABC".



```
edureka@ubuntu: ~  
edureka@ubuntu:~$ python3  
Python 3.4.1 (default, Jun  5 2014, 01:48:07)  
[GCC 4.8.2] on linux  
Type "help", "copyright", "credits" or "license" for more i  
nformation.  
>>> 'abc', "ABC" ←  
('abc', 'ABC') Version 3.4.1
```



```
edureka@ubuntu: ~  
edureka@ubuntu:~$ python3  
Python 3.4.1 (default, Jun  5 2014, 01:48:07)  
[GCC 4.8.2] on linux  
Type "help", "copyright", "credits" or "license" for more i  
nformation.  
>>> 'abc' + 'def' ←  
'abcdef' Version 3.4.1
```

Concatenation and repetition

- Strings are concatenated with the + sign:

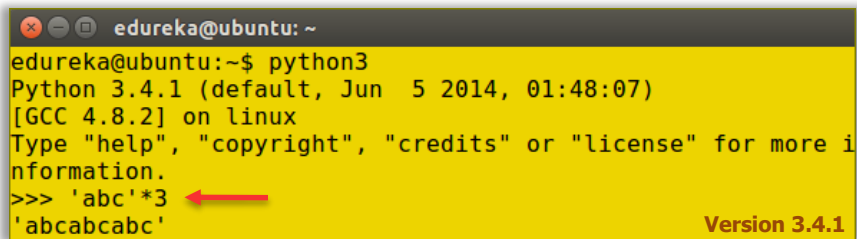
```
>>> 'abc'+'def'
```

```
'abcdef'
```

- Strings are repeated with the * sign:

```
>>> 'abc'*3
```

```
'abccabccabc'
```



```
edureka@ubuntu: ~  
edureka@ubuntu:~$ python3  
Python 3.4.1 (default, Jun  5 2014, 01:48:07)  
[GCC 4.8.2] on linux  
Type "help", "copyright", "credits" or "license" for more information.  
>>> 'abc'*3  
'abccabccabc'
```

Indexing and slicing Operation

- Python starts indexing at 0.
- A string *s* will have indexes running from 0 to *len(s)-1* (where *len(s)* is the length of *s*) in integer quantities.
- *s[i]* fetches the *i*th element in *s*.

Example :

```
>>> s = 'string'
```

```
>>> s[1] # note that Python considers 't' the first element.
```

```
't'      # of our string s
```

- *s[i:j]* fetches elements *i* (inclusive) through *j* (not inclusive).

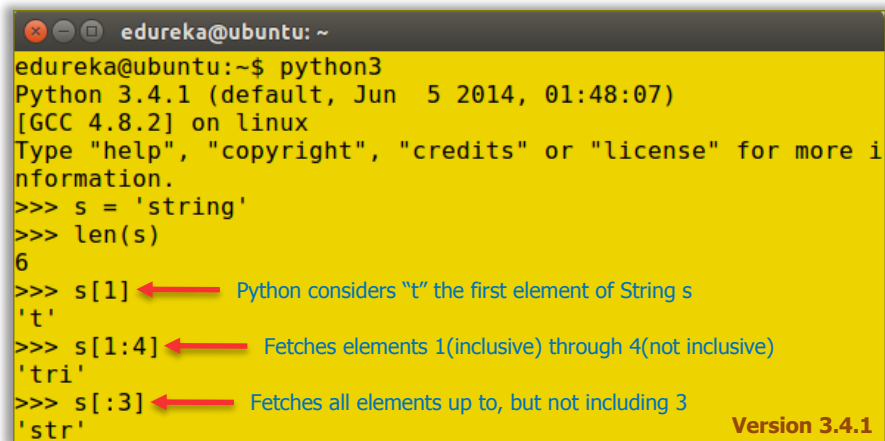
```
>>> s[1:4]
```

'tri'

- `s[:j]` fetches all elements up to, but not including `j`.

```
>>> s[:3]
```

'str'



```
edureka@ubuntu: ~  
edureka@ubuntu:~$ python3  
Python 3.4.1 (default, Jun  5 2014, 01:48:07)  
[GCC 4.8.2] on linux  
Type "help", "copyright", "credits" or "license" for more i  
nformation.  
>>> s = 'string'  
>>> len(s)  
6  
>>> s[1] ← Python considers "t" the first element of String s  
't'  
>>> s[1:4] ← Fetches elements 1(inclusive) through 4(not inclusive)  
'tri'  
>>> s[:3] ← Fetches all elements up to, but not including 3  
'str'
```

Version 3.4.1

-
- `s[i:]` fetches all elements from `i` onward (inclusive).

```
>>> s[2:]
```

'ring'

- `s[i:j:k]` extracts every `k`th element starting with index `i` (inclusive) and ending with index `j` (not inclusive).

```
>>> s[0:5:2]
```

'srn'

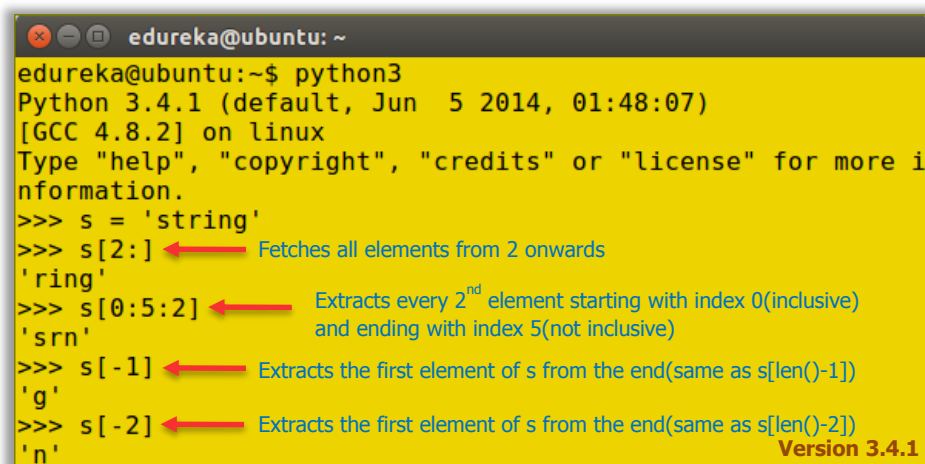
Python also supports negative indexes. For example, `s[-1]` means extract the first element of `s` from the end (same as `s[len(s)-1]`).

```
>>> s[-1]
```

```
'g'
```

```
>>> s[-2]
```

```
'n'
```



```
edureka@ubuntu: ~
edureka@ubuntu:~$ python3
Python 3.4.1 (default, Jun  5 2014, 01:48:07)
[GCC 4.8.2] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> s = 'string'
>>> s[2:] ← Fetches all elements from 2 onwards
'ring'
>>> s[0:5:2] ← Extracts every 2nd element starting with index 0(inclusive) and ending with index 5(not inclusive)
'srn'
>>> s[-1] ← Extracts the first element of s from the end(same as s[len()-1])
'g'
>>> s[-2] ← Extracts the first element of s from the end(same as s[len()-2])
'n'
Version 3.4.1
```

Membership Checking

`in` - Returns true if a character exists in the given string.

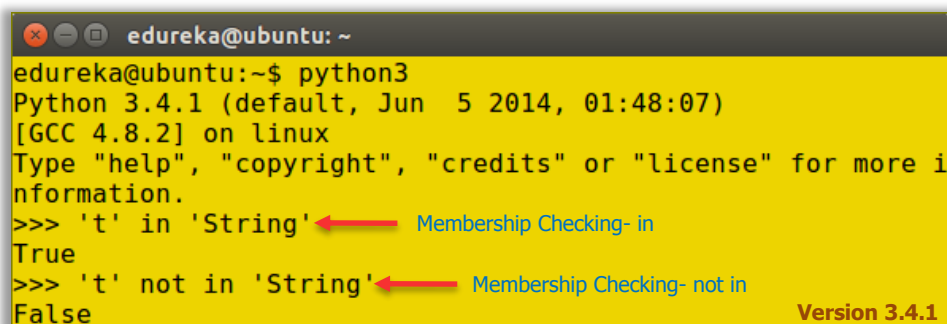
```
>>> 't' in 'String'
```

```
True
```

`not in` - Returns true if a character does not exist in the given string.

```
>>> 't' not in 'String'
```

```
False
```



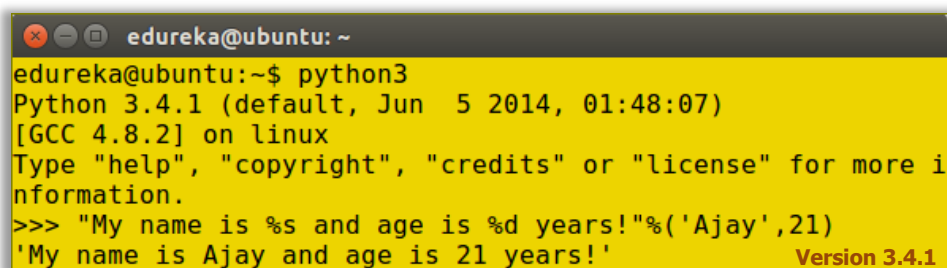
```
edureka@ubuntu: ~
edureka@ubuntu:~$ python3
Python 3.4.1 (default, Jun  5 2014, 01:48:07)
[GCC 4.8.2] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> 't' in 'String' ← Membership Checking- in
True
>>> 't' not in 'String' ← Membership Checking- not in
False
Version 3.4.1
```

String Formatting Operator: %

This operator is unique to strings and makes up for the pack of having functions from C's printf() family.

```
>>> print "My name is %s and age is %d years!" % ('Ajay', 21)
```

My name is Ajay and age is 21 years!



```
edureka@ubuntu: ~  
edureka@ubuntu:~$ python3  
Python 3.4.1 (default, Jun  5 2014, 01:48:07)  
[GCC 4.8.2] on linux  
Type "help", "copyright", "credits" or "license" for more information.  
>>> "My name is %s and age is %d years!"%('Ajay',21)  
'My name is Ajay and age is 21 years!' Version 3.4.1
```

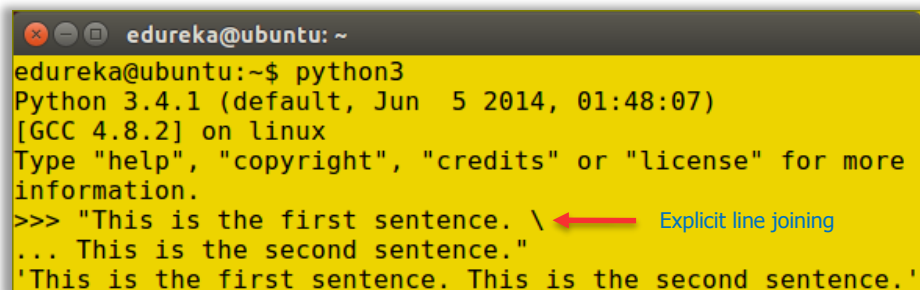
Escape Sequences and Line continuation

- \n, \t

```
>>>"This is the first sentence. \
```

This is the second sentence. "

"This is the first sentence. This is the second sentence."



```
edureka@ubuntu: ~  
edureka@ubuntu:~$ python3  
Python 3.4.1 (default, Jun  5 2014, 01:48:07)  
[GCC 4.8.2] on linux  
Type "help", "copyright", "credits" or "license" for more information.  
>>> "This is the first sentence. \ ← Explicit line joining  
... This is the second sentence."  
'This is the first sentence. This is the second sentence.'
```

Extensive List of Operations on String

Strings are amongst the most popular types in Python. We can create them simply by enclosing characters in quotes. Python treats single quotes the same as double quotes.

Creating strings is as simple as assigning a value to a variable. For example:

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

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. Following is a simple example:

```
#!/usr/bin/python  
  
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. Following is a simple example:

```
#!/usr/bin/python  
  
var1 = 'Hello World!'  
  
print "Updated String :- ", var1[:6] + 'Python'
```

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

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


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 lack of having functions from C's printf() family. Following is a simple example:

```
#!/usr/bin/python
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 %:

| Format Symbol | Conversion |
|---------------|---|
| %c | character |
| %s | string conversion via str() prior to formatting |
| %i | signed decimal integer |
| %d | signed decimal integer |
| %u | unsigned decimal integer |
| %o | octal integer |
| %x | hexadecimal integer (lowercase letters) |
| %X | hexadecimal integer (UPPERcase letters) |
| %e | exponential notation (with lowercase 'e') |
| %E | exponential notation (with UPPERcase 'E') |
| %f | floating point real number |
| %g | the shorter of %f and %e |
| %G | the shorter of %f and %E |

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

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

Triple Quotes

Python's triple quotes comes to the rescue by allowing strings to span multiple lines, including verbatim NEWLINES, TABs, and any other special characters.

The syntax for triple quotes consists of three consecutive **single or double** quotes.

```
#!/usr/bin/python

para_str = """this is a long string that is made up of
several lines and non-printable characters such as
TAB ( \t ) and they will show up that way when displayed.
NEWLINES within the string, whether explicitly given like
this within the brackets [ \n ], or just a NEWLINE within
the variable assignment will also show up.
"""
print para_str;
```

When the above code is executed, it produces the following result. Note how every single special character has been converted to its printed form, right down to the last NEWLINE at the end of the string between the "up." and closing triple quotes.

Also note that NEWLINES occur either with an explicit carriage return at the end of a line or its escape code (\n):

```
This is a long string that is made up of several lines and non-printable characters
such as TAB ( ) and they will show up that way when displayed.
NEWLINES within the string, whether explicitly given like this within the brackets
[], or just a NEWLINE within the variable assignment will also show up.
```

Raw strings don't treat the backslash as a special character at all. Every character you put into a raw string stays the way you wrote it:

```
#!/usr/bin/python
print 'C:\\nowhere'
```

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

```
C:\nowhere
```

Now let's make use of raw string. We would put expression in **r'expression'** as follows:

```
#!/usr/bin/python
print r'C:\\nowhere'
```

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

```
C:\\nowhere
```

Unicode String

Normal strings in Python are stored internally as 8-bit ASCII, while Unicode strings are stored as 16-bit Unicode. This allows for a more varied set of characters, including special characters from most languages in the world. I'll restrict my treatment of Unicode strings to the following:

```
#!/usr/bin/python  
print u'Hello, world!'
```

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

```
Hello, world!
```

As you can see, Unicode strings use the prefix `u`, just as raw strings use the prefix `r`.

Source- http://www.tutorialspoint.com/python/python_strings.htm