Python - String

A string object is one of the sequence data types in Python. It is an immutable sequence of Unicode characters. Strings are objects of Python's built-in class 'str'. String literals are written by enclosing a sequence of characters in single quotes ('hello'), double quotes ("hello") or triple quotes ('''hello''' or """hello""").

```
>>> str1='hello'
>>> str1
'hello'
>>> str2="hello"
>>> str2
'hello'
>>> str3='''hello'''
>>> str3
'hello'
>>> str4="""hello"""
>>> str4
'hello'
```

Note that the value of all the strings, as displayed by the Python interpreter, is the same ('hello'), irrespective of whether single, double or triple quotes were used for string formation. If it is required to embed double quotes as part of a string, the string itself should be put in single quotes. On the other hand, if a single-quoted text is to be embedded, the string should be written in double quotes.

```
>>> str1='Welcome to "Python Tutorial" from Altran'
>>> str1
'Welcome to "Python Tutorial" from Altran'
>>> str2="Welcome to 'Python Tutorial' from Altran"
>>> str2
"Welcome to 'Python Tutorial' from Altran"
```

A sequence is defined as an ordered collection of items. Hence, a string is an ordered collection of characters. The sequence uses an index (starting with zero) to fetch a certain item (a character in case of a string) from it.

```
>>> myString='hello'
>>> myString[0]
'h'
>>> myString[1]
'e'
>>> myString[4]
'o'
```

The string is an immutable object. Hence, it is not possible to modify it. The attempt to assign different characters at a certain index results in errors.

```
>>> myString[1]='a'
TypeError: 'str' object does not support item assignment
```

Triple Quoted String

The triple quoted string is useful when a multi-line text is to be defined as a string literal.

```
>>> myString="""Welcome to
Python Tutorial
from Altran"""
>>> myString
'Welcome to "Python Tutorial" from Altran'
```

Escape Sequences

The escape character is used to invoke an alternative implementation of the subsequent character in a sequence. In Python backslash \ is used as an escape character. Here is a list of escape sequences and their purpose.

Escape sequence	Description	Example	Result
\a	Bell or alert	"\a"	Bell sound
\b	Backspace	"ab\bc"	ac
\f	Formfeed	"hello\fworld"	hello world
\n	Newline	"hello\nworld"	Hello
\nnn	Octal notation, where n is in the range 0-7	'\101'	A
\t	Tab	'Hello\tPython'	Hello Python
\xnn	Hexadecimal notation, where n is in the range 0-9, a-f, or A-F	'\x41'	A

String Operators

Obviously, arithmetic operators don't operate on strings. However, there are special operators for string processing.

Operator	Description	Example

+	Appends the second string to the first	>>> a='hello'
		>>> b='world'
		>>> a+b
		'helloworld'

*	Concatenates multiple copies of the same string	>>> a='hello'
		>>> a*3
		'hellohellohello'

[]	Returns the character at the given index	>>> a = 'Python Tutorials'
		>>> a[7]
		T

[:]	Fetches the characters in the range specified by two index operands separated by the : symbol	>>> a = 'Python Tutorials'
		>>> a[7:15]
		'Tutorial'

in	Returns *true* if a character exists in the given string	>>> a = 'Python Tutorials'
		>>> 'X' in a
		False
		>>> 'Python' in a
		True
		>>> 'python' in a
		False

not in	Returns *true* if a character does not exist in the given string	>>> a = 'Python Tutorials'
		>>> 'X' not in a
		True
		>>> 'Python' not in a
		False

Converting to String

Python has an in-built function `str()` which returns a printable string representation of any object. In the previous chapter we have used `int()`, `float()` and `complex()` functions. They convert the string representation into integer, float and complex numbers, respectively. The `str()` function converts any number to a string object.

```
>>> str(12)
'12'
>>> str(6+5j)
'(6+5j)'
>>> str(1.11)
'1.11'
```

String Formatting

Interpolation of objects of different types at placeholders inside a string is called string formatting. The % operator (otherwise an arithmetic operator used to return the remainder of

division) is used to perform string formatting also. Format specification symbols (%d, %c, %f, %s, etc) used in C language are utilized as placeholders in a string.

In the following example, `name` is a string and age is an integer variable. Their values are inserted in the string with %s and %d format specification symbols, respectively. These symbols are interpolated to values in a tuple with the % operator in front.

```
>>> name="Bond"
>>> "My name is %s." % name
'My name is Bond.'
```

You can have multiple parameters too.

```
>>> name="Bond"
>>> age=30
>>> "My name is %s and age is %d years." % (name, age)
'My name is Bond and age is 30 years.'
```

All C style format specification symbols are permitted.

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 / %X	hexadecimal integer (lowercase letters)
%e / %E	exponential notation (with lowercase 'e')
%f	floating point real number

You can specify the width of integer and float objects. Here, integers a, b and c will occupy the width of 3 characters in the formatted string. Additional spaces will be padded to the left.

```
>>> a=1
>>> b=11
>>> c=111
>>> "a=%3d b=%3d c=%3d" % (a, b, c)
'a=  1 b= 11 c=111'
```

The following specifies the width of the float variable.

```
>>> percent=55.50
>>> "%5.2f" % percent
```

```
'55.50'
>>> "%6.2f" % percent
' 55.50'
>>> "%6.3f" % percent
'55.500'
>>> "%7.3f" % percent
' 55.500'
```

In the above example, `%5.2` specifies the width of float, where 5 is for total characters and 2 is for decimals. So, the result would be '55.50'.

The width of a string can also be specified. The default alignment is right. For left alignment, give a negative sign to width.

```
>>>'%4s' % 'abc'
>>>' abc'

>>>'%6s' % 'abc'
>>>' abc'

>>>'%-6s' % 'abc'
>>>'abc '

>>>a='abc'
>>>'%-6s' % a
>>>'abc '
```

format() method

The format() method can handle complex string formatting more efficiently. This method of in-built string class provides the ability to do complex variable substitutions and value formatting. This new formatting technique is regarded as more elegant. The general syntax of the `format()` method is as follows:

```
string.format(str1, str2,...)
```

The string itself contains placeholders {}, in which the values of variables are successively inserted.

```
>>>name="Bill"
>>>age=25
>>>"My name is {} and I am {} years old.".format(name, age)
'My name is Bill and I am 25 years old.'

>>>myStr = "My name is {} and I am {} years old."
>>>myStr.format(name, age)
'my name is Bill and I am 25 years old.'
```

You can also specify formatting symbols by using : instead of %. For example, instead of %s use {:s} and instead of %d use {:d}.

```
>>> "My name is {:s} and I am {:d} years old.".format(name, age)
'My name is Bill and I am 25 years old.'
```

Precision formatting of numbers can be done accordingly.

```
>>> percent=55.50
>>> "I have scored {:6.3f} percent marks.".format(percent)
'I have scored 55.500 percent marks.'
```

String alignment is done with <, > and ^ symbols in the place holder causing left, right and center alignment, respectively. Default is left alignment.

```
>>> '{:>10}'.format('test')
'      test'
>>> '{:<10}'.format('test')
'test      '
>>> '{:^10}'.format('test')
'   test   '
```

Built-in String Methods

capitalize():

Converts the first character of a string to uppercase letters.

```
>>> mystr='python'
>>> mystr.capitalize()
'Python'
```

upper()

Replaces the lowercase characters in a string with corresponding uppercase characters.

```
>>> mystr='Python'
>>> mystr.upper()
'PYTHON'
```

lower()

Replaces the uppercase characters in a string with corresponding lowercase characters.

```
>>> mystr='PYTHON'
>>> mystr.lower()
'python'
```

title():

Returns the string with the first character of each word converted to uppercase.

```
>>> mystr='python tutorial from altran'
>>> mystr.title()
'Python Tutorial From Altran'
```

find()

The `find()` method finds the first occurrence of a substring in another string. If not found, the method returns -1.

```
>>> mystr='Python Tutorial From Altran'
>>> mystr.find('From')
16
>>> mystr.find('xyz')
-1
```

Substring 'From' first occurs at position 16 (the count starts from 0). 'xyz' is not found, hence it returns -1.

count()

The `count()` method returns the number of occurrences of a substring in the given string.

```
>>> mystr='Python Tutorial From Altran'
>>> mystr.count('Tutorial')
2
```

isalpha()

The `isalpha()` method returns true if all the characters in a string are alphabetic letters (a-z or A-Z), otherwise it returns false.

```
>>> mystr='Altran'
>>> mystr.isalpha()
True
>>> mystr=Altran'
>>> mystr.isalpha()
False
```

isdigit()

The `isdigit()` is method returns true if all the characters in string are digits (0-9), if not, it returns false.

```
>>> str1='2000'
>>> str1.isdigit()
True
>>> str2='2,000'
>>> str2.isdigit()
False
```

islower()

The `islower()` method returns true if all the characters in the string are lowercase characters, else it returns false.

```
>>> str1='python'
>>> str1.islower()
True
>>> str2='Python'
>>> str2.islower()
False
```

isupper()

The `isupper()` method returns true if all the characters in the string are uppercase characters, else it returns false.

```
>>> var='ALTRAN'
>>> var.isupper()
True
>>> var='Altran'
>>> var.isupper()
False
```