字符串的基本操作

我们使用单引号或者双引号指定字符串

Python也使用\作为转义符

**>>> circus = 'Monty Python\'s Flying Circus'**[3] **>>> circus  
"Monty Python's Flying Circus"**

当一行写不完时，我们也使用\表示下一行继续。

**>>> couplet = "Shall I compare thee to a Summer's day?"\  
... "Thou are more lovely and more temperate:"**[1] **>>> print(couplet)  
Shall I compare thee to a Summer's day?Thou are more lovely and more temperate:**

括号也可起到同样效果

**couplet = ("Rough winds do shake the darling buds of May,"  
... "And Summer's lease hath all too short a date:")**[2] **>>> print(couplet)  
Rough winds do shake the darling buds of May,And Summer's lease hath all too short a date:**

转行要用三重引号来表示

**>>> couplet = """Shall I compare thee to a Summer's day?  
... Thou are more lovely and more temperate:"""  
>>> print(couplet)  
Shall I compare thee to a Summer's day?  
Thou are more lovely and more temperate:  
>>> couplet = '''Rough winds do shake the darling buds of May,  
... And Summer's lease hath all too short a date:'''  
>>> print(couplet)  
Rough winds do shake the darling buds of May,  
And Summer's lease hath all too short a date:**

我们可以看一下关于字符串都有哪些操作：

**>>> help(str)**

**Help on class str in module \_\_builtin\_\_:**

**class str(basestring)**

**| str(object='') -> string**

**|**

**| Return a nice string representation of the object.**

**| If the argument is a string, the return value is the same object.**

**|**

**| Method resolution order:**

**| str**

**| basestring**

**| object**

**|**

**| Methods defined here:**

**|**

**| \_\_add\_\_(...)**

**| x.\_\_add\_\_(y) <==> x+y**

**|**

**| \_\_contains\_\_(...)**

**| x.\_\_contains\_\_(y) <==> y in x**

**|**

**| \_\_eq\_\_(...)**

**| x.\_\_eq\_\_(y) <==> x==y**

**|**

**| \_\_format\_\_(...)**

**| S.\_\_format\_\_(format\_spec) -> string**

**|**

**| Return a formatted version of S as described by format\_spec.**

**|**

**| \_\_ge\_\_(...)**

**| x.\_\_ge\_\_(y) <==> x>=y**

**|**

**| \_\_getattribute\_\_(...)**

**| x.\_\_getattribute\_\_('name') <==> x.name**

**|**

**| \_\_getitem\_\_(...)**

**| x.\_\_getitem\_\_(y) <==> x[y]**

**|**

**| \_\_getnewargs\_\_(...)**

**|**

**| \_\_getslice\_\_(...)**

**| x.\_\_getslice\_\_(i, j) <==> x[i:j]**

**|**

**| Use of negative indices is not supported.**

**|**

**| \_\_gt\_\_(...)**

**| x.\_\_gt\_\_(y) <==> x>y**

**|**

**| \_\_hash\_\_(...)**

**| x.\_\_hash\_\_() <==> hash(x)**

**|**

**| \_\_le\_\_(...)**

**| x.\_\_le\_\_(y) <==> x<=y**

**|**

**| \_\_len\_\_(...)**

**| x.\_\_len\_\_() <==> len(x)**

**|**

**| \_\_lt\_\_(...)**

**| x.\_\_lt\_\_(y) <==> x<y**

**|**

**| \_\_mod\_\_(...)**

**| x.\_\_mod\_\_(y) <==> x%y**

**|**

**| \_\_mul\_\_(...)**

**| x.\_\_mul\_\_(n) <==> x\*n**

**|**

**| \_\_ne\_\_(...)**

**| x.\_\_ne\_\_(y) <==> x!=y**

**|**

**| \_\_repr\_\_(...)**

**| x.\_\_repr\_\_() <==> repr(x)**

**|**

**| \_\_rmod\_\_(...)**

**| x.\_\_rmod\_\_(y) <==> y%x**

**|**

**| \_\_rmul\_\_(...)**

**| x.\_\_rmul\_\_(n) <==> n\*x**

**|**

**| \_\_sizeof\_\_(...)**

**| S.\_\_sizeof\_\_() -> size of S in memory, in bytes**

**|**

**| \_\_str\_\_(...)**

**| x.\_\_str\_\_() <==> str(x)**

**|**

**| capitalize(...)**

**| S.capitalize() -> string**

**|**

**| Return a copy of the string S with only its first character**

**| capitalized.**

**|**

**| center(...)**

**| S.center(width[, fillchar]) -> string**

**|**

**| Return S centered in a string of length width. Padding is**

**| done using the specified fill character (default is a space)**

**|**

**| count(...)**

**| S.count(sub[, start[, end]]) -> int**

**|**

**| Return the number of non-overlapping occurrences of substring sub in**

**| string S[start:end]. Optional arguments start and end are interpreted**

**| as in slice notation.**

**|**

**| decode(...)**

**| S.decode([encoding[,errors]]) -> object**

**|**

**| Decodes S using the codec registered for encoding. encoding defaults**

**| to the default encoding. errors may be given to set a different error**

**| handling scheme. Default is 'strict' meaning that encoding errors raise**

**| a UnicodeDecodeError. Other possible values are 'ignore' and 'replace'**

**| as well as any other name registered with codecs.register\_error that is**

**| able to handle UnicodeDecodeErrors.**

**|**

**| encode(...)**

**| S.encode([encoding[,errors]]) -> object**

**|**

**| Encodes S using the codec registered for encoding. encoding defaults**

**| to the default encoding. errors may be given to set a different error**

**| handling scheme. Default is 'strict' meaning that encoding errors raise**

**| a UnicodeEncodeError. Other possible values are 'ignore', 'replace' and**

**| 'xmlcharrefreplace' as well as any other name registered with**

**| codecs.register\_error that is able to handle UnicodeEncodeErrors.**

**|**

**| endswith(...)**

**| S.endswith(suffix[, start[, end]]) -> bool**

**|**

**| Return True if S ends with the specified suffix, False otherwise.**

**| With optional start, test S beginning at that position.**

**| With optional end, stop comparing S at that position.**

**| suffix can also be a tuple of strings to try.**

**|**

**| expandtabs(...)**

**| S.expandtabs([tabsize]) -> string**

**|**

**| Return a copy of S where all tab characters are expanded using spaces.**

**| If tabsize is not given, a tab size of 8 characters is assumed.**

**|**

**| find(...)**

**| S.find(sub [,start [,end]]) -> int**

**|**

**| Return the lowest index in S where substring sub is found,**

**| such that sub is contained within S[start:end]. Optional**

**| arguments start and end are interpreted as in slice notation.**

**|**

**| Return -1 on failure.**

**|**

**| format(...)**

**| S.format(\*args, \*\*kwargs) -> string**

**|**

**| Return a formatted version of S, using substitutions from args and kwargs.**

**| The substitutions are identified by braces ('{' and '}').**

**|**

**| index(...)**

**| S.index(sub [,start [,end]]) -> int**

**|**

**| Like S.find() but raise ValueError when the substring is not found.**

**|**

**| isalnum(...)**

**| S.isalnum() -> bool**

**|**

**| Return True if all characters in S are alphanumeric**

**| and there is at least one character in S, False otherwise.**

**|**

**| isalpha(...)**

**| S.isalpha() -> bool**

**|**

**| Return True if all characters in S are alphabetic**

**| and there is at least one character in S, False otherwise.**

**|**

**| isdigit(...)**

**| S.isdigit() -> bool**

**|**

**| Return True if all characters in S are digits**

**| and there is at least one character in S, False otherwise.**

**|**

**| islower(...)**

**| S.islower() -> bool**

**|**

**| Return True if all cased characters in S are lowercase and there is**

**| at least one cased character in S, False otherwise.**

**|**

**| isspace(...)**

**| S.isspace() -> bool**

**|**

**| Return True if all characters in S are whitespace**

**| and there is at least one character in S, False otherwise.**

**|**

**| istitle(...)**

**| S.istitle() -> bool**

**|**

**| Return True if S is a titlecased string and there is at least one**

**| character in S, i.e. uppercase characters may only follow uncased**

**| characters and lowercase characters only cased ones. Return False**

**| otherwise.**

**|**

**| isupper(...)**

**| S.isupper() -> bool**

**|**

**| Return True if all cased characters in S are uppercase and there is**

**| at least one cased character in S, False otherwise.**

**|**

**| join(...)**

**| S.join(iterable) -> string**

**|**

**| Return a string which is the concatenation of the strings in the**

**| iterable. The separator between elements is S.**

**|**

**| ljust(...)**

**| S.ljust(width[, fillchar]) -> string**

**|**

**| Return S left-justified in a string of length width. Padding is**

**| done using the specified fill character (default is a space).**

**|**

**| lower(...)**

**| S.lower() -> string**

**|**

**| Return a copy of the string S converted to lowercase.**

**|**

**| lstrip(...)**

**| S.lstrip([chars]) -> string or unicode**

**|**

**| Return a copy of the string S with leading whitespace removed.**

**| If chars is given and not None, remove characters in chars instead.**

**| If chars is unicode, S will be converted to unicode before stripping**

**|**

**| partition(...)**

**| S.partition(sep) -> (head, sep, tail)**

**|**

**| Search for the separator sep in S, and return the part before it,**

**| the separator itself, and the part after it. If the separator is not**

**| found, return S and two empty strings.**

**|**

**| replace(...)**

**| S.replace(old, new[, count]) -> string**

**|**

**| Return a copy of string S with all occurrences of substring**

**| old replaced by new. If the optional argument count is**

**| given, only the first count occurrences are replaced.**

**|**

**| rfind(...)**

**| S.rfind(sub [,start [,end]]) -> int**

**|**

**| Return the highest index in S where substring sub is found,**

**| such that sub is contained within S[start:end]. Optional**

**| arguments start and end are interpreted as in slice notation.**

**|**

**| Return -1 on failure.**

**|**

**| rindex(...)**

**| S.rindex(sub [,start [,end]]) -> int**

**|**

**| Like S.rfind() but raise ValueError when the substring is not found.**

**|**

**| rjust(...)**

**| S.rjust(width[, fillchar]) -> string**

**|**

**| Return S right-justified in a string of length width. Padding is**

**| done using the specified fill character (default is a space)**

**|**

**| rpartition(...)**

**| S.rpartition(sep) -> (head, sep, tail)**

**|**

**| Search for the separator sep in S, starting at the end of S, and return**

**| the part before it, the separator itself, and the part after it. If the**

**| separator is not found, return two empty strings and S.**

**|**

**| rsplit(...)**

**| S.rsplit([sep [,maxsplit]]) -> list of strings**

**|**

**| Return a list of the words in the string S, using sep as the**

**| delimiter string, starting at the end of the string and working**

**| to the front. If maxsplit is given, at most maxsplit splits are**

**| done. If sep is not specified or is None, any whitespace string**

**| is a separator.**

**|**

**| rstrip(...)**

**| S.rstrip([chars]) -> string or unicode**

**|**

**| Return a copy of the string S with trailing whitespace removed.**

**| If chars is given and not None, remove characters in chars instead.**

**| If chars is unicode, S will be converted to unicode before stripping**

**|**

**| split(...)**

**| S.split([sep [,maxsplit]]) -> list of strings**

**|**

**| Return a list of the words in the string S, using sep as the**

**| delimiter string. If maxsplit is given, at most maxsplit**

**| splits are done. If sep is not specified or is None, any**

**| whitespace string is a separator and empty strings are removed**

**| from the result.**

**|**

**| splitlines(...)**

**| S.splitlines(keepends=False) -> list of strings**

**|**

**| Return a list of the lines in S, breaking at line boundaries.**

**| Line breaks are not included in the resulting list unless keepends**

**| is given and true.**

**|**

**| startswith(...)**

**| S.startswith(prefix[, start[, end]]) -> bool**

**|**

**| Return True if S starts with the specified prefix, False otherwise.**

**| With optional start, test S beginning at that position.**

**| With optional end, stop comparing S at that position.**

**| prefix can also be a tuple of strings to try.**

**|**

**| strip(...)**

**| S.strip([chars]) -> string or unicode**

**|**

**| Return a copy of the string S with leading and trailing**

**| whitespace removed.**

**| If chars is given and not None, remove characters in chars instead.**

**| If chars is unicode, S will be converted to unicode before stripping**

**|**

**| swapcase(...)**

**| S.swapcase() -> string**

**|**

**| Return a copy of the string S with uppercase characters**

**| converted to lowercase and vice versa.**

**|**

**| title(...)**

**| S.title() -> string**

**|**

**| Return a titlecased version of S, i.e. words start with uppercase**

**| characters, all remaining cased characters have lowercase.**

**|**

**| translate(...)**

**| S.translate(table [,deletechars]) -> string**

**|**

**| Return a copy of the string S, where all characters occurring**

**| in the optional argument deletechars are removed, and the**

**| remaining characters have been mapped through the given**

**| translation table, which must be a string of length 256 or None.**

**| If the table argument is None, no translation is applied and**

**| the operation simply removes the characters in deletechars.**

**|**

**| upper(...)**

**| S.upper() -> string**

**|**

**| Return a copy of the string S converted to uppercase.**

**|**

**| zfill(...)**

**| S.zfill(width) -> string**

**|**

**| Pad a numeric string S with zeros on the left, to fill a field**

**| of the specified width. The string S is never truncated.**

**|**

**| ----------------------------------------------------------------------**

**| Data and other attributes defined here:**

**|**

**| \_\_new\_\_ = <built-in method \_\_new\_\_ of type object>**

**| T.\_\_new\_\_(S, ...) -> a new object with type S, a subtype of T**