**Chapter 7 Text Processing**

We have seen data structures such as strings, lists, and tuples. We have also seen that every data type has associated with it special functions called methods.

In this chapter, we will cover more methods that can be used with strings, and also the string module. Additionally, we will introduce reading from and writing to files, using text files.

**7.1 String Processing Methods**

Text processing can be accomplished in Python using string variables, and the operators and methods associated with the type **str**. We have seen that strings can be indexed and sliced using the colon operator, and concatenated using the plus operator. Additionally, we have seen some of the string methods, including methods that manipulate strings such as **upper**, and methods that determine characteristics of strings, such as **isupper**. Since strings are a sequence type, the methods **index** and **count** can be used for strings. Additionally, there are many more methods that can be used to manipulate strings, and to determine the nature of strings. Since strings are an immutable type, methods that modify strings always return a new string; strings cannot be modified in place.

Case conversion methods

We have seen the **upper** method, which converts all letters of the alphabet in a string to upper case, and the **lower** method, which converts all letters of the alphabet in a string to lower case.

The **capitalize** method capitalizes just the first word in a string (so just the first word in a sentence, for example).

*>>> question = "how are you?"*

*>>> question.capitalize()*

'How are you?'

The value of the variable *question* has not been changed, however. To actually change the string variable, the result from the **capitalize** method would have to be assigned to *question*. This will create a new string that the variable *question* will now point to.

*>>> question = question.capitalize()*

Another useful method, **title**, will capitalize all of the words in a string.

*>>> urname = 'monty python'*

*>>> urname.title()*

'Monty Python'

The **swapcase** method converts upper to lower case, and lower to upper case.

*>>> wacky = 'hI tHERE'*

*>>> wacky.swapcase()*

'Hi There'

Spacing methods

There are methods that will delete extra spaces from strings. The **strip** method deletes both leading and trailing blanks from strings (but not from the middle of the string).

*>>> mystr = " hello there "*

*>>> len(mystr)*

18

*>>> newstr = mystr.strip()*

*>>> newstr*

'hello there'

The **lstrip** method deletes only leading blanks, and the **rstrip** method deletes only trailing blanks (the “l” and “r” in the method names stand for left and right).

*>>> newstr = mystr.lstrip()*

*>>> newstr*

'hello there '

*>>> newstr = mystr.rstrip()*

*>>> newstr*

' hello there'

There are several methods that will pad strings with extra leading and/or trailing fill characters, in order to center, left justify, or right justify the string within a specified width. These are named **center**, **ljust**, and **rjust**, respectively. By default, the fill character is a blank space. If the number of fill characters needed to center a string is odd, there will be one more trailing character than leading.

*>>> firstname = 'monty'*

*>>> firstname.center(8)*

' monty '

The fill character can be specified, e.g.

*>>> firstname.rjust(8,'$')*

'$$$monty'

The **zfill** method pads a string with leading zero (‘0’) characters in order to fill a specified width. This will, of course, make the most sense if the string contains a number.

*>>> '123.45'.zfill(10)*

'0000123.45'

Recall that an f-string can be used to fill in decimal places with 0’s.

*>>> mymoney = 123.45*

*>>> fmym = f'{mymoney:.3f}'*

*>>> fmym*

'123.450'

*>>> fmym.zfill(10)*

'000123.450'

Methods for Combining and Splitting Strings

The **join** method can be used to concatenate together all of the strings in an iterable (for example, a list). The method is called using a string which is used as the separator between the strings. In this example, the **join** method is called using a single space so each word is separated by a single space.

*>>> strlist = ['hello','hi','ciao']*

*>>> sepstr = ' '.join(strlist)*

*>>> sepstr*

'hello hi ciao'

The **split** method will take a string of words, and will split it into separate words and return a list of those words. The string to be used as the ***delimiter*** (the string between the words) can be passed to the method; otherwise, the default is whitespace. It is also possible to specify how many splits to do using the **maxsplit** argument; if this is omitted, all possible splits are made. If the delimiter is the space, all spaces will be ignored.

*>>> sent = "How are you"*

*>>> snwords = sent.split()*

*>>> snwords*

*['How', 'are', 'you']*

*>>> sent = "How are you"*

*>>> snwords = sent.split(maxsplit=1)*

*>>> snwords*

['How', 'are you']

If the delimiter is not whitespace, it is assumed that consecutive delimiters in the string are there to signify empty strings.

*>>> sent = "How\_are\_\_\_you" # one underscore, then three*

*>>> snwords = sent.split('\_')*

*>>> snwords*

['How', 'are', '', '', 'you']

The **rsplit** method works exactly like the **split** method, except that it splits from the right, not the left. This is important only if the **maxsplit** argument is specified.

*>>> sent = "How are you doing"*

*>>> snrspl = sent.rsplit(maxsplit=2)*

*>>> snrspl*

['How are', 'you', 'doing']

The **splitlines** method will split lines in a string, using the newline character as the delimiter, and return a list consisting of the individual lines as strings. Note that the \n is not included in the resulting strings.

*>>> longstr = "Welcome!\nHow are you?\nI am fine, thank you."*

*>>> linelist = longstr.splitlines()*

*>>> linelist*

['Welcome!', 'How are you?', 'I am fine, thank you.']

The **partition** method splits a string at the first occurrence of a specified separator and returns 3 values as a tuple: the part of the string to the left of the separator, the separator, and the part of the string to the right of the separator. The separator must be passed; there is no default for it.

*>>> sent = "How are you"*

*>>> left, sep, right = sent.partition(' ')*

*>>> print(repr(left), repr(sep), repr(right))*

'How' ' ' 'are you'

The **rpartition** method also splits a string into a tuple of 3 values, but it splits the string at the last occurrence of the specified separator, rather than the first.

*>>> sent = "How are you"*

*>>> left, sep, right = sent.rpartition(' ')*

*>>> print(repr(left), repr(sep), repr(right))*

'How are ' ' ' 'you'

Methods for Finding and Replacing

We have seen the **index** method, which returns the index of the first occurrence of the beginning of a substring within a string. If the substring is not found, the **index** method throws an error message. To protect from this, the **in** operator can be used in an **if-else** statement to check to see whether or not the substring is in the string.

*urname = 'monty python'*

*if 'x' in urname:*

*where = urname.index('x')*

*else:*

*print('x is not in the string')*

The **find** method also returns the index of the first occurrence of the beginning of a substring within a string, but if the substring is not found, it returns -1.

*>>> urname = 'monty python'*

*>>> urname.find('x')*

-1

There are similar methods, **rindex** and **rfind**, that return the index of the last occurrence of the beginning of a substring within a string. If the substring is not found, **rindex** throws an error message, whereas **rfind** returns -1.

*>>> urname = 'monty python'*

*>>> urname.rfind('o')*

10

*>>> urname = 'monty python'*

*>>> urname.rfind('x')*

-1

The **replace** method finds occurrences of a substring within a string, and replaces them with another substring. By default, all occurrences are replaced, but it is also possible to pass to the **replace** method a count of the number of occurrences to replace.

*>>> origstr = 'xxHelloxxxtherex'*

*>>> newstr = origstr.replace('x',' ')*

*>>> newstr*

' Hello there '

*>>> origstr = 'xxHelloxxxtherex'*

*>>> newstr = origstr.replace('x',' ',3)*

*>>> newstr*

' Hello xxtherex'

Boolean Methods

There are methods that ask questions about strings, and return **True**/**False** answers. We have seen the start **startswith** method, which determines whether or not a string begins with a substring, and the **endswith** method, which determines whether or not a string ends with a substring. We have also seen the **isupper** method, which determines whether or not all letters of the alphabet in a string are upper case, and the **islower** method, which determines whether or not all letters of the alphabet in a string are lower case. There are quite a few other “is” methods that ask questions and return True/False answers.

The **isalpha** method returns **True** if all of the characters in a non-empty string are alphabetic, or **False** if not. If the string is empty, it returns **False**.

*>>> testalnum = 'abcde123'*

*>>> testalnum.isalpha()*

False

*>>> testal = 'xyz'*

*>>> testal.isalpha()*

True

*>>> ''.isalpha()*

False

The **isalnum** method returns **True** if all characters in a non-empty string are alphanumeric (alphabetic or numeric), or **False** if not.

*>>> testalnum.isalnum()*

True

*>>> 'abc12!?'.isalnum()*

False

The **isdigit** method returns **True** if all of the characters in a non-empty string are digits, or **False** if not.

*>>> '45.67'.isdigit()*

False

*>>> '45678'.isdigit()*

True

String Module

There is a **string** module that contains several useful string constants. The constant **ascii\_lowercase** is a string with all of the lower case letters of the alphabet, in sequence.

*>>> import string*

*>>> string.ascii\_lowercase*

'abcdefghijklmnopqrstuvwxyz'

Similarly, the constant **ascii\_uppercase** is a string containing all of the upper case letters of the alphabet, and **ascii\_letters** is a string constant that contains all of the letters, first lower and then upper case.

*>>> string.ascii\_letters*

'abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ'

The constant **digits** contains all of the digits.

*>>> from string import digits*

*>>> digits*

'0123456789'

The following example checks each character in a string to determine whether it is a letter of the alphabet, a digit, or something else.

*import string*

*mychars = "Hi, 33!"*

*for onechar in mychars:*

*if onechar in string.ascii\_letters:*

*print(onechar, 'is a letter!')*

*elif onechar in string.digits:*

*print(onechar, 'is a digit!')*

*else:*

*print(onechar, 'is something else.')*

H is a letter!

i is a letter!

, is something else.

is something else.

3 is a digit!

3 is a digit!

! is something else.