Strings in Python

Python is very well suited to manipulating text. The basic datatype to record text information is a string. As in many other programming languages, a string is written as a sequence of characters surrounded by quotes.

In [4]:
```
x = "hello"
```

In [5]:
```
x
```

Out[5]:
```
'hello'
```

Note that Python reports the value as 'hello', though we used double quotes, "hello". In fact, Python allows both types of quotes and there is no difference in the values.

In [7]:
```
y = 'hello'
```

In [8]:
```
x == y
```

Out[8]:
```
True
```

One reason to allow both kinds of quotes is to make it easy to include quotes in the string. If the string contains a ', use " to delimit the string, and vice versa.

In [9]:
```
x = "Isn't"
```

In [10]:
```
x
```

Out[10]:
```
"Isn't"
```

If you need more complex strings, such as strings containing line breaks, or strings embedding both single and double quotes, use a triple quote to delimit your string.

In [11]:
```
title='''The boy's favourite
movie is "Batman"'''
```

In [12]:
```
title
```

Out[12]:
```
'The boy\'s favourite\nmovie is "Batman"'
```

Extracting parts of a string

A string is a sequence of characters with positions starting with 0. s[i] refers the character at position i in string s.

In [13]:
```
x
```

Out[13]:
```
"Isn't"
```

In [14]:
```
x[3]
```

Out[14]:
```
"'"
```

Python does not have separate datatype for single characters. There is no distinction between a single character and a string of length 1.

In [15]:
```
"'" == x[3]
```

Out[15]:
```
True
```

Use len() to extract the length of a string.

In [16]:
```
len(x)
```

Out[16]:
```
5
```

Positions in a string s run from 0 to len(s)-1. For convenience, one can also number positions in reverse: -1 is the last character, and -len(s) is the first character. Accessing a position that is not between 0 and len(s)-1 or between -1 and -len(s) is an error.

In [17]:
```
x='hello'
```

In [18]:
```
x[4]
```

Out[18]:
```
'o'
```

In [19]:
```
x[-4]
```

Out[19]:
```
'e'
```

In [20]:
```
x[5]
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
<ipython-input-20-df3e65441209> in <module>
----> 1 x[5]

IndexError: string index out of range
```

Slices

A substring from position i to position j is called a **slice**. For a string s, s[i:j] denotes the substring from s[i] to s[j-1]. Note that the right endpoint is not included.

In [21]:
```
x[1:4]
```

Out[21]:
```
'ell'
```

In [22]:
```
x[2:2]
```

Out[22]:
```
''
```

In [23]:
```
x[-4:4]
```

Out[23]:
```
'ell'
```

When extracting a single position s[i], i must be a valid index. When using slices, invalid indices default to sensible values. If the right endpoint is too large, it defaults to len(s). If the left endpoint is too small it defaults to 0 = -len(s).

In [24]:
```
x[1:10]
```

Out[24]:
```
'ello'
```

In [25]:
```
x[-6:4]
```

Out[25]:
```
'hell'
```

In [26]:
```
x[-7:17]
```

Out[26]:
```
'hello'
```

Since there is no separate character type, a slice of length 1 is the same as a single character.

In [27]:
```
x[1:2]
```

Out[27]:
```
'e'
```

In [28]:
```
x[1]
```

Out[28]:
```
'e'
```

In [29]:
```
x[1] == x[1:2]
```

Out[29]:
```
True
```

Like range(start,end,step), a slice specification can take an optional third argument, indicating the step length.

In [30]:
```
x[0:5:2]
```

Out[30]:
```
'hlo'
```

In [31]:
```
x[5:0:-2]
```

Out[31]:
```
'ol'
```

In [32]:
```
x[5:-6:-2]
```

Out[32]:
```
'olh'
```

Concatenation

String concatenation is denoted by +.

In [33]:
```
z = x + " there"
```

In [34]:
```
z
```

Out[34]:
```
'hello there'
```

Updating a string

Strings are **immutable**. You cannot update a letter in place.

In [35]:
```
x
```

Out[35]:
```
'hello'
```

In [36]:
```
x[3] = 'p'
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-36-23fcb3a9652c> in <module>
----> 1 x[3] = 'p'

TypeError: 'str' object does not support item assignment
```

If you want to modify a string, you must create a new string and then reassign it.

In [37]:
```
x = x[0:3]+"p!"
```

In [38]:
```
x
```

Out[38]:
```
'help!'
```

Built-in string functions

Python has a number of built-in functions to operate on strings. Here are a few.

find() and index()

s.find(c) locates the first position in s where c occurs, and returns -1 if c is not found in s.

In [39]:
```
x.find("h")
```

Out[39]:
```
0
```

In [40]:
```
x.find("l")
```

Out[40]:
```
2
```

In [41]:
```
x.find("z")
```

Out[41]:
```
-1
```

An optional second argument to find() gives the index from which to start searching for the string.

```
In [42]: x.find("l",3)
Out[42]: -1
```

One way to find all occurrences of a pattern is to repeatedly apply find(), moving beyond the latest index each time, till the return value is -1.

```
In [43]: y="abracadabra"

In [44]: y.find("a")
Out[44]: 0

In [45]: y.find("a",1)
Out[45]: 3

In [46]: y.find("a",4)
Out[46]: 5

In [47]: y.find("a",6)
Out[47]: 7

In [48]: y.find("a",8)
Out[48]: 10

In [49]: y.find("a",11)
Out[49]: -1

In [50]: y.find("a",9,11)
Out[50]: 10

In [51]: y.find("a",9,10)
Out[51]: -1
```

index() is like find() except it returns an error rather than -1 if the value is not found.

In [52]:
```
y.index("a")
```

Out[52]:
```
0
```

In [53]:
```
y.index("a",8)
```

Out[53]:
```
10
```

In [54]:
```
y.index("a",8,10)
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-54-8ff918f87f70> in <module>
----> 1 y.index("a",8,10)

ValueError: substring not found
```

Stripping whitespace

One common task with strings is to remove leading and trailing whitespace. Python has built-in functions strip(), lstrip() and rstrip() for this.

In [55]:
```
name = "  Madhavan  "
```

In [56]:
```
name.strip()
```

Out[56]:
```
'Madhavan'
```

In [57]:
```
name.rstrip()
```

Out[57]:
```
'  Madhavan'
```

In [58]:
```
name.lstrip()
```

Out[58]:
```
'Madhavan  '
```

Splitting and joining strings

One frequent use-case for working with strings is parsing csv (comma separated value) files exported from spreadsheets. split() splits a string into a list of substrings based on the separator provided.

In [59]:
```
line="ABC DEF,15-01-2016,M"
```

```
In [60]: line.split(",")
Out[60]: ['ABC DEF', '15-01-2016', 'M']

In [61]: columns = line.split(",")

In [62]: columns
Out[62]: ['ABC DEF', '15-01-2016', 'M']

In [63]: columns[0]
Out[63]: 'ABC DEF'

In [64]: columns[1]
Out[64]: '15-01-2016'

In [65]: columns[2]
Out[65]: 'M'

In [66]: columns[1].split('-')
Out[66]: ['15', '01', '2016']

In [67]: columns[1].split('-')[2]
Out[67]: '2016'

In [68]: (line.split(',')[1]).split('-')[2]
Out[68]: '2016'

In [69]: dobparts = columns[1].split('-')

In [70]: dobparts
Out[70]: ['15', '01', '2016']

In [71]: columns
Out[71]: ['ABC DEF', '15-01-2016', 'M']
```

The converse operation is to combine a list of strings into a single string, separated by a given delimiter. This function is called join. The syntax is slightly unusual: to combine a list l of strings with a delimiter d,

one writes d.join(l), not l.join(d).

```
In [72]: ",".join(columns)
```

```
Out[72]: 'ABC DEF,15-01-2016,M'
```

```
In [73]: "***".join(columns)
```

```
Out[73]: 'ABC DEF***15-01-2016***M'
```

```
In [74]: '-'.join(dobparts)
```

```
Out[74]: '15-01-2016'
```

If we don't specify the delimiter, split() breaks up the string at all whitespace. A sequence of whitespace is treated as a single delimiter.

```
In [75]: sentence="The quick brown  fox     jumps over the lazy dog"
```

```
In [76]: sentence.split()
```

```
Out[76]: ['The', 'quick', 'brown', 'fox', 'jumps', 'over', 'the', 'lazy', 'dog']
```

If we explicitly use ' ' as the delimiter, each occurrence of ' ' is treated as a separate delimiter.

```
In [77]: sentence.split(' ')
```

```
Out[77]: ['The',
          'quick',
          'brown',
          '',
          'fox',
          '',
          '',
          '',
          '',
          'jumps',
          'over',
          'the',
          'lazy',
          'dog']
```

Formatting a string

One common use of strings is to generate outputs from your code, through a print() statement.

In [78]:
```
print("The sentence is",sentence)
```

```
The sentence is The quick brown  fox     jumps over the lazy dog
```

In [79]:
```
x
```

Out[79]:
```
'help!'
```

In [80]:
```
print(x)
```

```
help!
```

Often, we want to intersperse fixed text with values of names.

In [81]:
```
a = 10
b = 7
```

In [82]:
```
print("the value of ",a,"+",b," is ",a+b)
```

```
the value of  10 + 7  is  17
```

A better way to do this is to use the format statement. In its basic form, {} denotes a position to be replaced by an argument to format. If there are three values to be substituted, use {} in three places and supply arguments to format in the same order.

In [83]:
```
print("the value of {} + {} is {}".format(a,b,a+b))
```

```
the value of 10 + 7 is 17
```

We can explicitly indicate the positions in the placeholders and use the same argument multiple times.

In [84]:
```
print("the left value of {0} and {1} is {0}".format(a,b))
```

```
the left value of 10 and 7 is 10
```

If we name the arguments, the order is not important.

In [85]:
```
print("{first} and {second} is {first}".format(first=a,second=b))
```

```
10 and 7 is 10
```

In [86]:
```
print("{first} and {second} is {first}".format(second=b,first=a))
```

```
10 and 7 is 10
```

The format() statement also allows to control the way the value is displayed, and how much space it takes.

In [87]:
```
print("a = {0}".format(a))
```

```
a = 10
```

Print a as a floating point number.

In [88]:
```
print("a = {0:f}".format(a))
```

```
a = 10.000000
```

In [89]:
```
print("a = {0:d}".format(a))
```

```
a = 10
```

Print a as a decimal (integer) so that it takes up 4 spaces.

In [90]:
```
print("a = {0:4d}".format(a))
```

```
a =   10
```

Coding with strings

If we just want to check if a pattern appears in a string or not, the notation "p in s" is simpler than using find()

In [91]:
```
x
```

Out[91]:
```
'help!'
```

In [92]:
```
"l" in x
```

Out[92]:
```
True
```

In [93]:
```
"ll" in x
```

Out[93]:
```
False
```

In [94]:
```
"lll" in x
```

Out[94]:
```
False
```

A string is a sequence. We can run through all characters in a string as follows.

In [95]:
```
for c in x:
    print(c)
```

```
h
e
l
p
!
```

Examples

Reversing a string

In [96]:
```
x[len(x)::-1]
```

Out[96]:
```
'!pleh'
```

In [97]:
```
def reverse(s):
    return(s[len(s)::-1])
```

In [98]:
```
reverse(x)
```

Out[98]:
```
'!pleh'
```

Checking if a string is a palindrome

In [99]:
```
def palindrome(s):
    return(s == reverse(s))
```

In [100]:
```
palindrome("malayalam")
```

Out[100]:
```
True
```

In [101]:
```
palindrome("tamil")
```

Out[101]:
```
False
```

Remove all occurrences of a character from a string

In [102]:
```
def exclude(s,p):  # remove all p's from s
    outputs = ""
    for c in s:
        if c != p:
            outputs = outputs + c
    return(outputs)
```

In [103]:
```
exclude(x,"l")
```

Out[103]:
```
'hep!'
```

In [104]:
```
exclude(x,"p")
```

Out[104]:
```
'hel!'
```

Remove all occurrences of a set of characters from a string

In [105]:
```
def excludeset(s,pset):  # remove all p's in pset from s
    outputs = ""
    for c in s:
        if not(c in pset):
            outputs = outputs + c
    return(outputs)
```

In [106]:
```
excludeset(x,"aeiou")
```

Out[106]:
```
'hlp!'
```

In [107]:
```
excludeset(x,"aeiouaei")
```

Out[107]:
```
'hlp!'
```

In [108]:
```
w = "2017abc334xyz"
```

In [109]:
```
excludeset(w,"0123456789")
```

Out[109]:
```
'abcxyz'
```

Using built-in functions

Removing all occurrences of a character is same as replacing them by the empty string.

In [110]:
```
x.replace("l","")
```

Out[110]:
```
'hep!'
```

Use replace() to update part of a string

In [111]:
```
x.replace("lo","p!")
```

Out[111]:
```
'help!'
```