Using strings in files

Introduction to Programming

Rae Harbird, r.harbird@ucl.ac.uk

Overview

- Python programs can take input from files and send output to files.
- Python can deal with two types of files:
  - **Text files** are files containing (roughly speaking) a sequence of characters. Text files can be opened in a text editor such as Notepad.
  - **Binary files** are files containing (roughly speaking) a sequence of binary digits. Binary files are good for storing e.g. images or audio or (structures of) objects.
- **Note:** We will only cover text files in this course.

Useful string functions

As we are dealing with text files in this lecture, we will be using more of the string handling features in Python.

Every line in a text file is a string and Python has a lot of string processing methods that we can use to process the contents, for example:

Method name	Description
`string.strip()`	remove the character or characters given as a parameter from both sides of the string, default whitespace
`string.rstrip()`	remove the character or characters given as a parameter from the right hand end of a string
`string.split()`	split the string into elements using the character given as a parameter, default is whitespace
`string.count()`	return the number of occurrences of the string given as a parameter
`string.index()`	return the start posibtion of the string given as a parameter

Stripping whitespace from strings

Take this string, there's a lot of whitespace either side of the text:

```
>>> sentence = '\t \n   This is a test string. \t\t \n\n\n'
>>> print(sentence)

    This is a test string.
```

Removing leading whitespace

```
>>> print(sentence.lstrip())
This is a test string.
```

Removing trailing whitespace

```
print(sentence.rstrip())

    This is a test string.
```

Stripping whitespace from strings (continued)

Removing both leading and trailing whitespace

```
print(sentence.strip())
This is a test string.
```

Searching for substrings

Count the number of occurrences

```
sentence = "Be nice to yu turkeys dis christmas\n\
Cos' turkeys just wanna hav fun\nTurkeys are cool,\
turkeys are wicked\nAn every turkey has a Mum."

>>> sentence.count('turkey')
4
```

Find the position of a string from the lefthand side

```
>>> sentence.index('Turkey')
68
```

Search for position in slice

```
>>> len(sentence)
131
>>> sentence.count('turkey', 68, 131)
2
>>> sentence.count('turkey', 68)
2
```

Substring in a string?

Position

```
>>> sentence.index('cool')
80
```

Determining whether a string contains a substring

```
>>> 'cool' in sentence
True

>>> 'stuffing' in sentence
False

>>> 'turkey' not in sentence
False
```

Replacing substrings

```
>>> sentence.replace('turkey', 'chestnut roast')

"Be nice to yu chestnut roasts dis christmas\nCos' chestnut roasts
just wanna hav fun\nTurkeys are cool, chestnut roasts are wicked\n
An every chestnut roast has a Mum."
```

Splitting substrings

Splitting strings is especially important in file processing. Given a string containing words separated by one or more words:

```
>>> sentence = "Talking Turkeys\nBe nice to yu turkeys\
dis christmas,\n        Cos' turkeys just wanna hav fun."
>>> sentence_list = sentence.split()
>>> sentence_list

['Be', 'nice', 'to', 'yu', 'turkeys', 'dis', 'christmas,', "Cos'",
  'turkeys', 'just', 'wanna', 'hav', 'fun.']
```

There's still a problem with the comma and full-stop here and more processing may be necessary.

Joining substrings

Joining the elements of a list is similar:

```
>>> '<--->'.join(sentence_list)
'Be<--->nice<--->to<--->yu<--->turkeys<--->dis<--->
christmas<--->Cos\'<--->turkeys<--->just<--->wanna
<--->hav<--->fun<--->Turkeys<--->are<--->cool,<--->
turkeys<--->are<--->wicked<--->An<--->every<--->turkey
<--->has<--->a<--->Mum.'
```

Comparison operators (review)

Example	Result
`'Orange' == 'orange'`	False
`'Orange' != 'orange'`	True
`'Orange' < 'orange'`	True
`'Orange' <= 'orange'`	True
`'Orange' > 'orange'`	False
`'Orange' >= 'orange'`	False

What is going on here?

In programming languages, including Python, each character is represented internally by a number. We can see this when using the functions `chr()` and `ord()`.

`ord()` returns the numeric representation of a unicode character.

```
>>> ord(`o`)
111

>>> ord(`O`)
79
```

`chr()` does the reverse.

```
>>> chr(79)
'O'
>>> chr(111)
'o'
```

Character testing methods

We will see that when processing a file, we often need to check that the content is as we expect. We might need to check that, for example, the characters we have read in are digits. Python has several functions that can help with this. Note: the table does not contain a complete list.

Character testing functions	Description
`isalnum()`	Returns `True` if the string only contains alphanumeric characters.
`isalpha()`	Returns `True` if the string only contains alphabetic characters.
`isdigit()`	Returns `True` if the string only contains digits ('0', '2', etc).
`isnumeric()`	Returns `True` if the string only contains a number without +, - or a decimal place.
`isdigit()`	Returns `True` if the string only contains digits.