Strings; Text Files

Pedro Barahona
DI/FCT/UNL
Métodos Computacionais
1st Semestre 2020/2021

Text Processing

- Much useful information is not numeric and takes the form of text (e.g. names, documents, ...). Hence the need to represent text and to subsequently process it.
- All programming languages support text data types, namely
  - **Characters**; and
  - **Strings** (sequences of characters).
- Basic 128 characters, include letters, digits, punctuation and control characters, and are usually represented by their **ASCII** (American Standard Code for Information Interchange) codes.
- Notice that 128 different characters require 7 bits to be represented (128 = 27).
- With an 8th bit (initially meant for parity checking), the extended ASCII code allows the representation of 128 more characters used in several languages (other than English).

Text Processing

- The characters represented in 7bit ASCII code are:
  - Letters (52), uppercase (26) e lowercase (26)
  - Digits (10)
  - Space and other punctuation “visible” characters (34)
    - ‘ “ ( ) [ ] { } , . : ; = < > + - * \ | / ^ ~ ´ ` # $ % & _ ! ? @
  - Control (invisible) characters (32)
    - horizontal tab (\t), new line (\n), alert (\a), ...
- With an 8th bit, other 128 characters can be represented, such as
  - ç, ã, ñ, š , ø , ∞, ← φ, Σ, Ш, غغ ,א ,ך
- The representation of other alphabets (Chinese, Arab, Indian, ...) require 16 bits (a total of 216 = 65536 characters) and is supported in Unicode (widely adopted in the Internet).
- Unicode (UTF) subsumes the ASCII code (the initial 256 characters are the same).

Strings

- Strings are sequences of characters, and text can be regarded as a “big” string.
- To assign a variable with a string, the text must be delimited by quotation marks (") or single quotes ('). For example,
  - `x = "this is a string"`
- Having two delimiters is quite handy, when the text includes one of them, as in
  - `name = "Rui d' Almeida"` ; or
  - `next = 'He said "Enough" and left.'`

  … although escape sequences can be used
  - `name = 'Rui d\' Almeida'` **`; or`**
  - `next = "He said \"Enough\" and left."`

  … and these are sometimes unescapable
  - `sentence = "Rui d' Almeida said \"Enough\" and left."`
  - `sentence = 'Rui d\' Almeida said "Enough" and left.'`

Escape Sequences

- The following escape sequences are useful for referring special non visible characters, namely control characters.
- There are some differences in the handling of the delimiters and escape characters, and the "" delimiter should be preferred. The following escape sequences are accepted in Python (e.g. in a print statement).

`\\`	**back slash**	**(\)**	
`\"`	**quotation**	**(")**	
`\'`	**single quote**	**(')**	
`\0`	nil	(code 0)	
`\a`	alert	(code 7)	
`\b`	back	(code 8)	– overwrites previous character
`\f`	new page	(code 12).	
`\n`	**new line**	**(code 10).**	
`\r`	return	(code 13)	– overwrites previous line
`\t`	**horizontal tab**	**(code 9).**	
`\v`	vertical tab	(code 11).	

String Operations

- Strings are encoded as lists of characters of characters, so the usual operations on vectors can be used to compose and decompose strings.

Concatenation

- Strings can be concatenated with the + operator, as with lists.

```
In : v1 = [1,2,3]
In : v2 = [4,5,6]
In : v1 + v2
Out: [1,2,3,4,5,6]
In : name = "Rui"
In : surname = "Santos"
In : full = name + surname
In : full
Out: "RuiSantos"
In : full = name + " " + surname
Out: "Rui Santos"
```

String Operations

Projection (Extraction) of Substrings

- Projection of strings to some of their substrings (or characters) can be obtained through the usual vector operations

```
In : text = "This is a string."
In : text
Out: 'This is a string.'
In : text[0:4].          # all chars between the 1st and 5th
Out: 'This'
In : text[-7:-1]
Out: ‘string'
```

- Several methods are defined in the class string (cf. the dir function)

```
In : dir(text)
Out:
['__add__’,
 ...
'zfill']
```

String Operations

Substring Search

- If one is interested in finding the (first) position(s) where a substring occurs within a string, the **find** and **rfind** methods can be used.

```
In : text = 'This is a string.'
In : text.find('string')
Out: 10
In : text.find('i')
Out: 3
In : text.rfind('i')
Out: 13
In : text.find('z')
Out: -1   # not found
```

String Operations

Splitting Strings

- In many cases we are interested in splitting a string by some character(s) that is used as a separator (for example a semi-colon (;), a tab (‘\t) or a space.
- Method split() returns a list of strings, without the separators

```
In : line = 'abd; def; 123'
In : line.split(';')
Out: ['abd', ' def', ' 123']
In : line = ‘12\t24\t45.8\n’
In : line.split(‘\t’)
Out: ['12', '24', '45.8\n']
```

- Note: Beware of spaces and “end of line” (‘\n’) characters that might be maintained in the individual strings.

String Operations

"Cleaning" Strings

- In many cases we are not interested in leading and trailing spaces, as well as white characters such as tabs and end-of-lines (e.g. when they are read from files).
- They can be eliminated with methods **strip**.

```
In : line = "   This is a line.  \n"
In : len(line)
Out: 21
In : line.strip()
Out: 'This is a line.'
In : len(line.strip())
Out: 15
```

String Operations

Comparing Strings

- Strings may also be compared lexicographically (i.e. alphabetically).
- Notice that lower and upper cases are different (in ASCII, upper cases are before lower cases).

```
In : "abc" == "abc"
Out: True
In : "abc" > "abd"
Out: False
In : "A" < "a"
Out: True
In : "A" < "5"
Out: False
In : "5" < 5
TypeError: '<' not supported between instances of 'str' and 'int'
```

String Types

Strings and Numbers

- Strings are different from numbers, and different operations apply to these types.
- But converting strings to numbers and vice-versa is possible (but beware of different types of numbers).

```
In : '45'+'12'
Out: '4512'
In : '45'*'12'
TypeError: can't multiply sequence by non-int of type 'str'
In : int('45')
Out: 45
In : str(34)
Out: '34'
In : float('45.7')
Out: 45.7
In : int('45.7')
ValueError: invalid literal for int() with base 10: '45.7'
```

String Type Information

Information Functions about Types

- In addition to the conversion functions a number of methods are available to strings to obtain the types of characters, namely

- isalnum - string composed of alphanumeric characters
- isalpha - string composed of alphabetic characters
- isascii - string composed of ASCII characters (7 bits, no special characters)
- isdigit - string where all characters are digits
- isidentifier - string is a valid identifier
- islower - string where all characters are lower case letters
- isprintable - string where all characters are printable (spaces, tabs, eol)
- isspace - string where all characters are non printable (spaces, tabs, eol)
- istitle - string starting with an upper case letter followed by lower case
- isupper - string where all characters are upper case letters

String Type Information

Some examples

```
In : 'ab5dc'.isalnum()
Out: True
In : 'ação'.isascii()
Out: False
In : '3456'.isdigit()
Out: True
In : '_45'.isidentifier()
Out: True
In : ‘a45'.isidentifier()
Out: True
In : '56 67'.isprintable()
Out: True
In : '\t   \n'.isprintable()
Out: False
In :'Doutor'.istitle()
Out: True
```

```
In : 'ab5dc'.isalpha ()
Out: False
In : ‘facto'.isascii()
Out: True
In : ‘34a56'.isdigit()
Out: False
In : 'a.45'.isidentifier()
Out: False
In : '45a'.isidentifier()
Out: False
In : '56 67’.isspace()
Out: False
In : '\t   \n’.isspace()
Out: False
In :'DR.'.istitle()
Out: True
```

File Input / Output

- When the amount of data is large, it is not practical/feasible to enter data and read program results from the terminal. In most cases, we use files to have permanent access to this data (here we will only consider text files – that can be read by any text processor, such as notepad).
- Files are managed by a file system (part of the operation system – Windows, Linux, MacOS) and files are organised in a (inverted) tree.
- At the top there is a root directory that recursively contains other directories (the branches of the tree) and possibly files (the leafs of the tree).
- Spyder supports some typical file system instructions, that can be used either in a program or at the terminal. Among the most useful
  - **pwd** – returns a string representing the current directory
  - **ls** – shows the files and folders in the current directory
  - **cd name** – changes the current directory to the directory with name
  - **cd ..** – changes the current directory to its parent directory
  - **cd //** – makes the root as the current directory

File Input / Output

- To read to or write from a file, it is necessary a) to **open** it, and after handling its data (reading from / writing into), the file should be **closed**.
- In **Python**, opening a file is done with instruction
  - `open(fileName, mode)`

  where
  - **fileName** is the name of the file (as seen from the current directory)
  - **mode** is either “r” for read or “w” for write

```
fid = open('file.txt', 'r')
```

- The function returns an object (the file handler) that should be subsequently used to read/write data and finally to close the file.

File Input / Output

- The function returns an object (the file handler) that should be subsequently used to read/write data and finally to close the file.
  - **Note:** If the file could not be opened, the function returns an error. To avoid aborting the computation this error should be handled by an IO exception

```
try:
    fid = open('file.txt', 'r')
except IOError:
    print(Error: no such file’)
```

- Once used, the file should be closed with method
  - `fid.close()`

where

- **fid** is the file handler that was obtained when the file was opened.

File Output

- The access to an open file is **sequential**, i.e. data items are read/written one after the other with no going back or direct access to some k^{th} item of the file.
- To write (text) data in a file, previously opened the method write should be used on the fid object.

```
In : fid = open('example.txt', 'w')
In : fid.write('This is the first line;\nand this is the second.\n')
Out: 48
In : fid.write('Fim\n')
Out: 4
In : fid.close()
```

example.txt

```
This is the first line;
and this is the second.
Fim.
```

- Note the explicit use of the new line (\n) character.
  - there is no writeln method in Python

File Input

read()

- To read a file, the method read may be used.
- This method reads the whole file (from the current position to the end) and retuns a string with all characters that were read, including the new lines.
- Reading beyond the end of file returns an empty string.

readlines()

- Quite often it is more useful to read the text file line by line, so as to process the information in each line
- The method readlines() returns a list with all the file lines.

readline()

- To read incrementally the file, the method readline() reads a single line (from the current position of the cursor).
  - It returns an empty string if attempting to read **beyond** the end of the file.

File Input

read()

- To read a file, the method read may be used.
- This method reads the whole file (from the current position to the end) and retuns a string with all characters that were read, including the new lines.
- Reading beyond the end of file returns an empty string.

readlines()

- Quite often it is more useful to read the text file line by line, so as to process the information in each line
- The method readlines() returns a list with all the file lines.

readline()

- To read incrementally the file, the method readline() reads a single line (from the current position of the cursor).
  - It returns an empty string if attempting to read **beyond** the end of the file.

File Input / Output

- Example: Read the file with a matrix and return (it as a lists of lists)

```
def read_matrix(fname):
    """returns a matrix stored in file"""
    fid = open(fname, 'r');
    mat = []
    lines = fid.readlines();
    fid.close()
    for line in lines:
        row = []
        numbers = line.strip().split(' ');
        for number in numbers:
            row.append(int(number))
        mat.append(row)
    return mat
```

matrix.txt

```
12 20 30 89
34 50 98 13
25 47 26 56
```

```
In : mm = read_matrix('matrix.txt')
In : mm
Out: [[12, 20, 30, 89], [34, 50, 98, 13], [25, 47, 26, 56]]
```