# Strings; Text Files

Pedro Barahona
DI/FCT/UNL
Métodos Computacionais
1st Semestre 2017/2018

# 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, ך, א, غ
- 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 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 apostrophs (‘). 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
  - `sent = ‘He said “Enough” and left.’`

  … although escape sequences can be used
  - `nome = ‘Rui d’’Almeida’` ; or
  - `Sent = “He said\“Enough\” and left.”`

  … and these are sometimes unescapable
  - `complete_name = “Rui d’Almeida said \“Enough\” and left.”`
  - `complete_sent = ‘Rui d’’Almeida said “Enough” and left.’`
- Note: In MATLAB special characters (e.g. **ç**, **ã**) should be avoided in strings, as the support is limited and cumbersome (e.g. ç is represented by \303.)

# 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 MATLAB (wih “ delimiters).

| | | |
|---|---|---|
| `\\` | **back slash”** | **(\)** |
| `\”` | **quotation** | **(”)** |
| `\’` | **apostrophe** | **(‘)** |
| `\0` | nil | (control-@ (code 0) |
| `\a` | alert | (control-g with code 7) |
| `\b` | back | (control-h with code 8) |
| `\f` | new page | (control-l with code 12). |
| `\n` | **new line** | **(control-j with code 10).** |
| `\r` | return | (control-m with code 13). |
| `\t` | **horizontal tab** | **(control-i with code 9).** |
| `\v` | vertical tab | (control-k with code 11). |

# String Operations

- Strings are encoded as uni-dimensional arrays (vectors) of characters, so the usual operations on vectors can be used to compose and decompose strings.

**Concatenation**

- Strings can be concatenated either with array operations or with the predefined functions **strcat** and **cstrcat** (the first function trims the leading and trailing spaces)

```
>> name = "rui"
name = rui
>> surname = "santos"
surname = santos
>> fullname1 = [name, " ", surname]
fullname = rui santos
>> fullname2 = strcat(name, " ", surname)
fullname = ruisantos
>> fullname3 = cstrcat(name, " ", surname)
fullname = 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,

```
>> text = "This is a string."
text = This is a string.
>> pre = text(1:6).  % all chars between the 1st and 6th
pre = This i
>> pos = text(9:end) % all chars between the 9th and last
pos = a string.
```

or through the **substr** predefined function.

```
>> text = "This is a string."
>> fix = substr(text,6,7)   % 7 chars starting at the 6th
fix = is a st
```

# String Operations

## Substring Search

- If one is interested in finding the position(s) of a substring within a string, the findstr function can be used.

```
>> text = "This is a string."
text = This is a string.
>> findstr("string", text)
ans = 11
>> findstr("i", text)
ans = 3  6  14
>> findstr("z", text)
ans = [](0x0)
```

# String Operations

## Comparing Strings

- The previous example involves the comparison of (sub)strings. Strings can of course be compared by comparing each of their characters (accessed with projections).
- Given its relevance, the predefined Boolean function **strcmp** compares two strings

```
>> text1 = "text 1"
text1 = text 1
>> text2 = "text 2"
text2 = text 2
>> strcmp(text1,text2)
ans = 0
>> strcmp(text1, "text 1")
ans = 1
```

# String Operations

## Comparing Strings Lexicographically

- Sometimes one is interested in checking whether a string precedes (in lexicographic order) another string.
- There is no predefined function for this Boolean function, but one can define it, by comparing the characters of the strings.
- Characters can be compared lexicographically, by simply using the usual relational operators. In fact this comparison is made on the codes of the characters that define their order.
- Note that codes can be obtained by
  - function t**oascii**; or
  - multiplying the char by 1! – why?

```
>> "a" < "b"
ans = 1
>> "f" < "c"
ans = 0
>> "1" < "3"
ans = 1
>> "9" < "A"
ans = 1
>> "Z" < "a"
ans = 1
>> toascii("e")
ans = 101
>> "f"*1
ans = 102
```

# String Operations

## Comparing Strings Lexicographically

- Now the strings can be compared with function **strbef** defined as defined below.
- The characters are compared one by one until a difference is spotted (and the function immediately ends – **return** statement). Otherwise, the shorter string is the before the longer (e.g. “maria” is lexicographically before “mariana”).

```
function before = strbef(st1, st2)
% this function checks whether st1 is before st2
  i = 1;  % start comparison in char 1
  while length(st1) >= i && length(st2) >= i
    if st1(i) == st2(i)
      i = i + 1;
    else
      before = (st1(i) < st2(i));
      return;
    end
  end
  before = (length(st1) < length(st2));
end
```

# String Operations

## Comparing Strings Lexicographically

- Since comparisons are made between the **codes** of the characters it is often important to guarantee that characters have the same **case**, so as to avoid that "Mariana" is considered before "maria".
- The string operations **toupper/1** and **tolower/1** allow the conversion of all letters to the same case (upper and lower respectively) so that they can be properly compared.

```
>> strbef("Mariana", "maria")
ans = 1
>> strbef(toupper("Mariana"), toupper("maria"))
ans = 0
>> tolower("Mariana")
ans = mariana
```

# String Operations

## Comparing Strings Lexicographically

- When comparing strings it is often necessary to remove leading and trailing spaces, since they are not usually significant. This can be done with functions
  - **strtrim** – removes removes both leading and trailing spaces
  - **deblank** – only removes the trailing spaces

```
>> text = "  Spaces: 2 leading, 4 trailing    ";
>> length(text)
ans = 35
>> length(strtrim(text))
ans = 29
>> length(deblank(text))
ans = 31
```

- In longer strings, not adequately formatted, it is also convenient to convert **newlines** into spaces and remove duplicate spaces. This is left as an exercise.

# String Operations

## Strings and Numbers

- Another commonly used conversion is between text that represents numerical information, and the numbers it represents.
- A string simply encodes the digits of a number, not the number itself, and this has to be taken into account for handling this information.

```
>> st1 = "15";
>> st2 = "426";
>> st3 = "158";
>> st1 + st2
error: mx_el_lt: nonconformant arguments (op1 is 1x3, op2 is 1x2)
>> st2 + st3
ans = 101 106 110
>> st2 * 1
ans = 52 53 54
```

- Two functions, **str2num** and **num2str** allow the conversion between these two representations

# String Operations

## Strings and Numbers

- To obtain the expected results, conversion to the appropriate data types is needed.

```
>> st1 = "15";
>> st2 = "426";
>> st3 = "158"
>> x = str2num(st1) + str2num(st2)
x = 441
>> y = str2num(st2) + str2num(st3)
y = 584
>> v1 = [x,y]
v1 = 441 584
>> v2 = [num2str(x),num2str(y)]
v2 = 441584
>> = 1*v2
ans = 52 52 49 53 56 52
>> = 1*str2num(v2)
ans = 441584
```

# String Operations

## Information Boolean Functions about Types

- In addition to the conversion functions a number of information Boolean functions is available in MATLAB to obtain the type of the character (or string) being used
  - **isalpha(s)** 1 if s is alphabetic (a letter)
    - upper or lower case
  - **islower(s)** 1 if s is lower case letter
  - **isupper(s)** 1 if s is upper case letter
  - **isdigit(s)** 1 if s is a digit
  - **isalnum(s)** 1 if s is alphanumeric
    - a digit or alphabetic
  - **isspace(s)** 1 if s is space
  - **ispunct(s)** 1 if s is a punctuation char
  - **iscntrl(s)** 1 if s is a control character

```
>> st = "47 is Prime.";
>> d = isdigit(st)
d = 1 1 0 0 0 0 0 0 0 0 0 0
>> l = islower(st)
l = 0 0 0 1 1 0 0 1 1 1 1 0
>> u = isupper(st)
u = 0 0 0 0 0 0 1 0 0 0 0 0
>> s = isspace(st)
s = 0 0 1 0 0 1 0 0 0 0 0 0
>> p = ispunct(st)
p = 0 0 0 0 0 0 0 0 0 0 0 1
```

# 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).
- The OCTAVE IDE 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
  - **dir** – returns a string denoting directories and files of 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 MATLAB, opening a file is done with instruction
  - `fopen(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
- The function returns a positive integer (the channel number) 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 -1.
- Once used, the file should be closed with instruction
  - `fclose(fid)`

  where
  - **fid** is the channel number that was obtained when the file was opened.
  - **Note:** This function returns 0 if the file was properly closed or -1 otherwise.

# 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, the following MATLAB instruction may be used
  - **`fprintf(fid, template, par1, par2, …, parn)`**

  where
  - **fid** is the channel number that was obtained when the file was opened.
  - **template** is the string that is written, where parameters $\mathbf{par_i}$ replace the "place holders" (in the sequence they are specified), which can take the following types
    - **%[n]i** an integer parameter , with optional **n** characters (leading spaces)
    - **%[m.n]f** a real number with optional **m** characters, **n** after the decimal dot.
    - **%[n]s** a string with (optional) **n** characters (padded with leading spaces)

```
>> s1 = "integer";
>> s2 = "decimal"
>> printf("An%10s:%4i and %s %8.3.", s1, 17, s2, 4.12)
"An   integer:  17 and decimal    4.12."
```

# File Output

**Some Notes:**

1. The following variants of the **fprintf** instruction can be used with the exact same formatting rules, but omitting the file id to
   - **`printf(template, parameters)`**
     - writes the string to the terminal
   - **`sprintf(template, parameters)`**
     - returns the string (e.g. the string can be assigned to a variable)

2. Since the place holders of the parameters are specified with a % sign, if the string to be written includes a "%", then it is specified by the escape sequence "%%".

3. Tabs and newlines (change of line) can be specified in the template by means of the escape sequences
   - **`\t`** **for a tab**
   - **`\n`** **for a newline**

# File Input

- The most general form to read text from a (text) file is to read each and every character from the file, which can be done with instruction
  - `fscanf(fid, %c, "C")`
    - Returns the next character from the file being read.
- The format of what is being read can change. Rather than a single **character**, one might be interested in reading one **word** at a time, and this can be done changing the reading template to
  - `fscanf(fid, %s, "C")`
    - Returns the next word from the file being read.
    - Notice that words are delimited by spaces and new lines, but punctuation characters are considered in the words.
- Finally, the next command allows reading a file, line by line
  - `fgetl(fid)`
    - Returns a string with the line with the current position of the cursor, i.e. all the characters that start in the cursor and up to the next **newline**.
    - It returns -1 if attempting to read **beyond** the end of the file.

# File Input

- Of course, when reading a file one might not know its length, so the end of file should be tested before attempting to read anything. This can be done with function
  - `feof(fid)`
    - A Boolean function that returns whether the end of the file has been reached.
- Of course, every line must be "parsed" to extract its content that may consist of several data items.
- Typically, when data items are separated by some character (e.g. space or comma) the parsing might be done by finding the positions of the separators (with **findstr**) and then extracting the data between the separators.
- In other cases, specially when the lines / files contain numerical information, the function **str2num/1** makes a direct conversion of arrays and matrices.

# File Input / Output

- To read text from the following function can be defined

```
function txt = read_text_file(filename)
   fid = fopen(filename, "r");
   txt = "";
   while ! feof(fid)
      ch = fscanf(fid, "%c", "C");
      if ischar(ch)
         txt = cstrcat(txt,ch);
      end
   end
   fclose(fid);
end
```

- **Note**: Beware that strings with new lines are not shown adequately in the Octave console.

# File Input / Output

- With the function above an array or matrix can be read “indirectly” as shown below.
- Assume the following text file in your working directory

**Matrix.txt**

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

- Then the following interaction illustrates the reading of the matrix

```
>> str = read_text_file("matrix.txt");
>> mat = str2num(str)
mat = 12 20 30 89
      34 50 98 13
      25 47 26 56
>>
```