

Chapter 2

Handling Textual Data

This chapter introduces the main variable classes that are used in the MATLAB[®] programming environment for representing and handling text. First, in [Sect. 2.1](#), the basic variable type for representing text, which is the character array, is described. Then, in [Sects. 2.2](#) and [2.3](#), cell arrays and structures, which are the most commonly used variable classes for handling and operating with text, are described, respectively. Finally, in [Sect. 2.4](#), a brief overview is provided on the specific MATLAB[®] built-in functions for operating with text, as well as other useful functions worth to be known.

2.1 Characters and Character Arrays

The basic variable type for representing text in the MATLAB[®] programming environment is the character. A character is a variable used to represent symbols in some predefined encoding system. Depending on the encoding scheme being used, a character can be represented with one, two or more bytes. By default, when writing and reading text files from your system, the MATLAB[®] environment uses the default encoding of the operating system. However, in the case of MATLAB[®] data files, the *unicode* encoding system is used. This guarantees the portability of data files across systems. The specific encoding of a given text can be changed at any moment by using MATLAB[®] functions `native2unicode` and `unicode2native`. More details on the encoding scheme issue will be given in [Chap. 5](#), where we will focus our attention in the problem of writing and reading text files.

A text string is represented as an array (or matrix) of characters. For defining a variable as a character array, it is required that its value is provided within apostrophes. Try the following example in the command line:

```
>> string = 'Hello, this is a string!'
string =
Hello, this is a string!
```

(2.1)

Such a command defines and initializes the variable `string` as a character array. By using MATLAB[®] command `whos` we can list all current variables in the workspace along with their corresponding sizes and classes:

```
>> whos
Name      Size      Bytes  Class  Attributes
string    1x24         48   char
```

(2.2)

You can get the numerical codes assigned to each character in the array by casting the variable `string` from character to integer as it is shown in the following example:

```
>> codes = cast(string,'int16')
codes =
Columns 1 through 10
    72    101    108    108    111    44    32    116    104    105
Columns 11 through 20
   115     32    105    115     32    97    32    115    116    114
Columns 21 through 24
   105    110    103     33
```

(2.3a)

```
>> whos
Name      Size      Bytes  Class  Attributes
codes     1x24         48  int16
string    1x24         48   char
```

(2.3b)

Similarly, you can recover the original variable `string` from variable `codes` by either casting it back from integer to character or, alternatively, by using the function `char`:

```
>> cast(codes,'char')
ans =
Hello, this is a string!
```

(2.4a)

```
>> char(codes)
ans =
Hello, this is a string!
```

(2.4b)

In the same way any numerical variable is handled by default as a matrix, characters are also handled as matrices. Indeed, as already seen in example (2.2), a string of n characters is actually represented in the MATLAB® workspace as a $1 \times n$ matrix of characters. According to this, any list or array of strings can be represented by means of a matrix. Take a look at the following example:

```
>> list = ['This is the first string ','This is the second string']
list =
This is the first string
This is the second string
```

(2.5a)

```
>> whos list
```

Name	Size	Bytes	Class	Attributes
list	2x25	100	char	

(2.5b)

The only problem with this particular way of representing string arrays is that all the strings in the array (rows in the matrix) are required to have the same number of characters. This is why we added a white space at the end of the first string in example (2.5a)!

If you try to reproduce the same example without including the trailing white space at the end of the first string, you will get the following error message:

```
>> list = ['This is the first string'; 'This is the second string']
??? Error using ==> vertcat
CAT arguments dimensions are not consistent.
```

(2.6)

So, it is very important to remember that representing string arrays by means of matrices will always require padding with blanks all the strings up to the length of the largest one. In this sense, and with the exception of some very specific cases, we do not recommend using matrices for representing string arrays. Indeed, as we will see in the following two sections, handling text by using other classes of data types such as cell arrays and structures is much more convenient.

Nevertheless, one of the main advantages of the matrix representation discussed here is that any of the conventional MATLAB® matrix indexing strategies can be used with it. For instance, try reproducing the following sequence of examples in the command line:

```
>> dataset = ['name age'; 'mark 35'; 'beth 26'; 'peter 39']
dataset =
name age
mark 35
beth 26
peter 39
```

(2.7a)

```
>> names = dataset(2:end,1:end-4)
names =
mark
beth
peter
```

(2.7b)

```
>> ages = dataset(2:end,end-1:end)
ages =
35
26
39
```

(2.7c)

```
>> people_in_their_30s = dataset(dataset(:,end-1)=='3',1:end-4)
people_in_their_30s =
mark
peter
```

(2.7d)

In the examples presented in (2.7a), first a list of names and ages was defined and written into a variable called `dataset`, which is just a two dimensional array of characters. As seen from (2.7a), the defined array of characters looks like a table, the first row containing the headers for each column in the table: name and age, and the subsequent rows containing the corresponding data entries, one sample per row.

By just using matrix indexing operations, several different tasks can be accomplished. For instance, in (2.7b), the names in `dataset` were extracted by retaining columns 1 to 5 from rows two and onwards. Similarly, in (2.7c), the ages were extracted by retaining the last two columns from rows two and onwards. Finally, (2.7d) shows a more elaborated example, in which only the names for those people whose age's first digit was equal to 3 were extracted.

2.2 Handling Text with Cell Arrays

The most convenient way of representing strings in the MATLAB[®] programming environment is by using a special class of variables denominated cell arrays. A cell array is a special class of structure that allows organizing variables into a matrix form regardless their size and type.

Consider the following example, in which a 2×2 cell array is defined:

```
>> cell_array = {'Hello World',eye(3);5+j,dataset}
cell_array =
    'Hello World'      [3x3 double]
    [5.0000 + 1.0000i] [4x9 char ]
```

(2.8)

In this example, the variable `cell_array` constitutes a 2×2 matrix of elements, where element $\{1,1\}$ is a text string, element $\{1,2\}$ is by itself a 3×3 numerical matrix, element $\{2,1\}$ is a complex number, and element $\{2,2\}$ is the same 4×9 matrix of characters created in (2.7a).

Different from the case of conventional arrays and matrices (whose elements are retrieved by indicating their indexes within parentheses), the elements within a cell array must be retrieved by using braces. So, the four elements of the 2×2 cell array defined in (2.8) can be retrieved as follows:

```
>> cell_array{1,1}
ans =
Hello World
```

(2.9a)

```
>> cell_array{1,2}
ans =
     1     0     0
     0     1     0
     0     0     1
```

(2.9b)

```
>> cell_array{2,1}
ans =
5.0000 + 1.0000i
```

(2.9c)

```
>> cell_array{2,2}
ans =
name age
mark 35
beth 26
peter 39
```

(2.9d)

According to this, the best way to represent any set of strings is by using cell arrays. In this kind of representation, each individual string is represented by means of a character array, and a vector (or matrix) of strings can be constructed by means of a cell array.

Consider the following example, in which a list of strings is constructed by using the cell array data class:

```
>> list = {'This is the 1st string','This is the 2nd one','And the 3rd'}
list =
    'This is the 1st string'
    'This is the 2nd one'
    'And the 3rd'
```

(2.10)

Notice that with this kind of representation strings are not required to have the same number of characters. And, like in any cell array, each individual string can be retrieved by using braces as follows:

```
>> list{1} % retrieves the first string in list
ans =
```

(2.11a)

```
This is the 1st string
```

```
>> list{2} % retrieves the second string
ans =
```

(2.11b)

```
This is the 2nd one
```

```
>> list{3} % retrieves the third string
ans =
```

(2.11c)

```
And the 3rd
```

Additionally, each character or substring within the strings can be retrieved in the same way it is done in the case of character arrays:

```
>> % retrieves the first four characters in the first string
>> list{1}(1:4)
ans =
```

(2.12a)

```
This
```

```
>> % retrieves the last three characters in the second string
>> list{2}(end-2:end)
ans =
```

(2.12b)

```
one
```

```
>> % retrieves all non-white-space characters in the third string
>> list{3}(not(list{3}==' '))
ans =
```

(2.12c)

```
Andthe3rd
```

It should be known that cell arrays also admit the use of parentheses for retrieving their contents. However, it is important to understand the difference between using parentheses or braces for accessing cell array elements. When using parentheses, the retrieved elements are the cells of the cell array (i.e. retrieved elements are also of the class `cell` array), while when using braces the retrieved elements are the contents within the cells (i.e. whichever class is contained in each cell).

The following example better clarifies the difference between using parentheses or braces for accessing cell array contents:

```
>> with_braces = list{1}
with_braces =
This is the 1st string
```

(2.13a)

```
>> with_parentheses = list(1)
with_parentheses =
    'This is the 1st string'
```

(2.13b)

```
>> whos list with_braces with_parentheses
```

Name	Size	Bytes	Class	Attributes
list	3x1	440	cell	
with_braces	1x22	44	char	
with_parentheses	1x1	156	cell	

(2.13c)

As can be seen from (2.13c), while the variable `with_braces` is a character array of size 1×22 , the variable `with_parentheses` is a cell array of size 1×1 . In this latter case, we have retrieved cell 1 from cell array `list`, while in the first case, we have retrieved the string within the cell. This difference might not seem important at this point, but it will definitively be very important later on.

2.3 Handling Text with Structures

Another alternative for handling text is using structures. As many other programming languages, the MATLAB[®] environment supports the definition and use of structures. A structure is actually a set of variables that are indexed as fields of a common main variable. This kind of representation is especially useful for importing and exporting data from and to databases or *XML* formats. Consider, for instance, the following example in which a structure with 4 fields is created:

```
>> structure = struct('f1','Hello World','f2',eye(3),'f3',5+j);
```

(2.14a)

```
>> structure.f4.names = ['mark ','beth ','peter'];
>> structure.f4.ages = [35;26;39];
```

(2.14b)

Notice from (2.14b) that the fourth field of `structure` is defined as a structure itself, which is composed of subfields `names` and `ages`. Structures, as well as cell arrays, admit nested constructs.

For accessing structure contents, each field in a structure can be retrieved by appending a dot to the name of the main variable followed by the name of the field:

```
>> structure.f1 % retrieves field f1
ans =
Hello World
```

(2.15a)

```
>> structure.f2 % retrieves field f2
ans =
    1     0     0
    0     1     0
    0     0     1
```

(2.15b)

```
>> structure.f3 % retrieves field f3
ans =
5.0000 + 1.0000i
```

(2.15c)

```
>> structure.f4 % retrieves field f4
ans =
    names: [3x5 char]
    ages: [3x1 double]
```

(2.15d)

In the particular case of nested structures, such as the one in `structure.f4`, each element can be retrieved by concatenating with dots the names of the corresponding sequence of fields:

```
>> structure.f4.ages' % retrieves (and transposes) subfield ages of f4
ans =
    35     26     39
```

(2.16a)

```
>> structure.f4.names % retrieves subfield names of f4
ans =
mark
beth
peter
```

(2.16b)

```
>> % retrieves the second row of subfield names of f4
>> structure.f4.names(2,:)
ans =
beth
```

(2.16c)

Finally, very important and useful resources for converting structure-based text representations into cell-array-based text representations, and vice versa, are available through MATLAB[®] built-in functions `struct2cell`, `fieldnames` and `cell2struct`. In the remaining of this section, we illustrate the use of these three functions in detail.

Let us consider again the sample dataset defined in (2.7a), which contains information about the names and ages of three people. In (2.7a), this dataset was represented by using of a two dimensional array of characters. Now, let us consider a more appropriate representation by means of a structure array composed of the two fields `name` and `age`. In the following example, such a structure array is created and filled in with the corresponding three data entries:

```
>> data = struct('name',{'mark','beth','peter'},'age',{'35','26','39'})
data =
3x1 struct array with fields:
    name
    age
```

(2.17)

As seen from (2.17), our sample dataset has been stored into a structure array of 3×1 elements (one element per data entry), and each element of the structure array is composed of two fields: `name` and `age`.

Now, let us consider the procedure for converting the structure-based representation of (2.17) into a cell-array-based representation. For this, the two functions `struct2cell` and `fieldnames` must be used. While the former maps the fields of the structure array into a cell array, the latter retrieves the names of the fields into another cell array. We proceed as follows:

```
>> datacell = struct2cell(data)
datacell =
    'mark'    'beth'    'peter'
    '35'      '26'      '39'
```

(2.18a)

```
>> datafields = fieldnames(data)
datafields =
    'name'
    'age'
```

(2.18b)

Notice from (2.18a) how the resulting cell array `datacell` contains the same data entries as the structure array `data`. The mapping has been done such that elements and fields of the structure array correspond to columns and rows of the cell array, respectively; i.e. the 3×1 structure array has been mapped into a $2 \times 3 \times 1$ cell array. In general, `struct2cell` will map any $N \times M$ structure array of K fields into a $K \times N \times M$ cell array.

In addition to `struct2cell`, as seen from (2.18b), function `fieldnames` should be used if we are also interested in retrieving the field names in the structure. In this case, the structure's field names are retrieved within a one-dimensional cell array of strings.

Let us now consider the problem of going back from the cell-array-based representation obtained in (2.18) to the original structure-based representation in

(2.17). In this case, the function `cell2struct` must be used as it is illustrated in the following example:

```
>> databack = cell2struct(datacell,datafields,1)
databack =
3x1 struct array with fields:
    name
    age
```

(2.19a)

```
>> databack(1)
ans =
    name: 'mark'
    age: '35'
```

(2.19b)

```
>> databack(2)
ans =
    name: 'beth'
    age: '26'
```

(2.19c)

```
>> databack(3)
ans =
    name: 'peter'
    age: '39'
```

(2.19d)

As seen from (2.19a), function `cell2struct` requires three input parameters: the cell array to be mapped into the structure, a cell array of strings containing the names of the fields to be used in the structure definition, and the dimension along which the cell array is to be folded into the structure fields (as the original fields were mapped into rows in (2.18a); in this case, this parameter must be 1).

2.4 Some Useful Functions

After having seen the most fundamental issues related to handling textual data, and before moving forward to more advanced procedures and techniques, we will devote this section to present an overview of the most common MATLAB[®] built-in functions for handling and operating with text strings, as well as some other general functions that are worth to be known. More than a comprehensive revision, this section presents a general overview, which is actually intended to be a quick reference. In this sense, a very basic description of the functions is presented here. Most of these functions will be studied and described in more detail in the following two chapters, while other functions are left for self-studying in some of the exercises proposed in Sect. 2.6.

Table 2.1 summarizes the main functions that are available in the MATLAB[®] programming environment for handling and operating with strings, along with their corresponding categories and descriptions (this same information can be listed in the MATLAB[®] command window by typing: `help strfun`).

There are 41 functions in total in Table 2.1, which are distributed into six different categories. From these six categories, *string operations* constitutes the most extensive one. Most of the functions within this category will be described and studied in detail in Chaps. 3 and 4, while others are covered by some proposed exercises within those chapters. More specifically:

- Chapter 3 is fully devoted to function `regexp` (and its case insensitive version `regexpi`) which is the MATLAB[®] implementation for matching regular expressions.
- Chapter 4 devotes special attention to string functions that are specifically suited for: search and comparing (`strfind`, `strcmp`, `strcmpi`, `strncmp` and `strncmpi`), replacement and insertion (`regexprep` and `strrep`), and segmentation and concatenation (`strtok`, `strcat` and `strvcat`).
- The rest of functions within the category *string operations* are covered by proposed exercises in the exercise sections of Chaps. 3 and 4.

Most of the functions belonging to the three conversion categories (*character set conversion*, *string to number conversion* and *base number conversion*) are left to you for self-studying. Basic descriptions for these functions can be obtained by using the `help` command. The functions belonging to the two remaining categories *general* and *string test* are partially covered by exercises in Sect. 2.6. Special attention is paid to functions `char`, `cellstr` and `sprintf` in Sect. 4.3.

Figure 2.1 presents a two-dimensional map of the basic string functions listed in Table 2.1. In the map, each function is represented by a marker and each of the six function categories are identified with a different marker. Function names are printed next to the markers for further reference.

In addition to the string related functions presented in Table 2.1, there are other MATLAB[®] functions of more general scope that are worth to be known too. These functions are summarized in Table 2.2 and described thereafter.

Next, let us illustrate the use of the functions presented in Table 2.2. Consider, for instance, the following example in which we illustrate the use of `lookfor` and `help` for searching functions and displaying their descriptions¹:

```
>> lookfor blanks % searches the string 'blanks' in all M-files
blanks                - String of blanks.
deblank               - Remove trailing blanks.
```

(2.20a)

¹ Reprinted with permission from The MathWorks, Inc.

Table 2.1 Basic string functions and their corresponding categories and descriptions. (Reprinted with permission from The MathWorks, Inc.)

Function	Category	Description
char	General	Create character array (string)
strings	General	Help for strings
cellstr	General	Create cell array of strings from character array
blanks	General	String of blanks
deblank	General	Remove trailing blanks
iscellstr	String tests	True for cell array of strings
ischar	String tests	True for character array (string)
isspace	String tests	True for white space characters
isstrprop	String tests	Check if string elements are of a specified category
regex	String operations	Match regular expression
regexpi	String operations	Match regular expression, ignoring case
regexprep	String operations	Replace string using regular expression
strcat	String operations	Concatenate strings
strcmp	String operations	Compare strings
strncmp	String operations	Compare first N characters of strings
strcmpi	String operations	Compare strings ignoring case
strncmpi	String operations	Compare first N characters of strings ignoring case
strfind	String operations	Find one string within another
strjust	String operations	Justify character array
strrep	String operations	Replace string with another
strtok	String operations	Find token in string
strtrim	String operations	Remove insignificant whitespace
upper	String operations	Convert string to uppercase
lower	String operations	Convert string to lowercase
native2unicode	Char set conversion	Convert bytes to Unicode characters
unicode2native	Char set conversion	Convert Unicode characters to bytes
num2str	String to number conversion	Convert numbers to a string
int2str	String to number conversion	Convert integer to string
mat2str	String to number conversion	Convert a 2-D matrix to a string in MATLAB syntax
str2double	String to number conversion	Convert string to double precision value
str2num	String to number conversion	Convert string matrix to numeric array
sprintf	String to number conversion	Write formatted data to string
sscanf	String to number conversion	Read string under format control
hex2num	Base number conversion	Convert hexadecimal string to double precision num.
hex2dec	Base number conversion	Convert hexadecimal string to decimal integer
dec2hex	Base number conversion	Convert decimal integer to hexadecimal string
bin2dec	Base number conversion	Convert binary string to decimal integer
dec2bin	Base number conversion	Convert decimal integer to a binary string
base2dec	Base number conversion	Convert base B string to decimal integer
dec2base	Base number conversion	Convert decimal integer to base B string
num2hex	Base number conversion	Convert singles and doubles to hexadecimal strings


```
>> iskeyword('start') % valid variable name
ans =
    0
```

 (2.21a)

```
>> iskeyword('end') % not valid name as it is a MATLAB keyword
ans =
    1
```

 (2.21b)

The function `pause` allows for momentarily halting the execution either for some specified amount of time or until the user hits any key in the keyboard:

```
>> pause(5) % halts the execution for 5 seconds
```

 (2.22a)

```
>> pause % halts the execution until a key is hit
```

 (2.22b)

The function `keyboard` also halts the execution of the current program, but additionally it gives access to the command window. The execution is resumed when the user enters the command `return` and hits the *enter* key:

```
>> keyboard % halts the program and gives access to the command window
K>> a = 5+1;
K>> return
```

 (2.23)

The function `input` allows the user for entering either a numeric value or a string directly from the command window:

```
>> r = input('How old are you? ') % prompts for a numeric variable
How old are you? 25
r =
    25
```

 (2.24a)

```
>> s = input('What is your name? ','s') % prompts for an input string
What is your name? John
s =
John
```

 (2.24b)

The function `disp` allows for displaying variables in the command window:

```
>> disp(dataset) % displays the variable dataset defined in (2.7a)
name  age
mark  35
beth  26
peter 39
```

(2.25)

Finally, functions `inputdlg`, `questdlg` and `msgbox` allow for collecting inputs and displaying information by means of interactive dialog and message boxes. In the case of `inputdlg`, the entered information is returned into a cell array; while in the case of `questdlg`, the selected option is returned in a string.

The function `msgbox`, by default, does not halt the execution of the current program. If the execution is to be halted when displaying a message, the function `msgbox` should be used along with function `uiwait` and the option 'modal' must be included in the function call.

The following examples illustrate the use of these last three functions. The resulting dialog and message boxes are depicted in Fig. 2.2.

```
>> boxname = 'Date'; q1 = 'What day is it?'; q2 = 'What time is it?';
>> today = inputdlg({q1,q2},boxname,2);
```

(2.26a)

```
>> boxname = 'Fruit'; q1 = 'What is your favorite fruit?';
>> fruit = questdlg(q1,boxname,'apple','banana','mango','mango');
```

(2.26b)

```
>> boxname = 'Notice'; msg = 'This is the end of section 2.4.';
>> uiwait(msgbox(msg,boxname,'modal'));
```

(2.26c)

2.5 Further Reading

For more detailed information on the *unicode* character set you must refer to The Unicode Consortium (2011). Some details about some other character encodings are also provided in Sect. 5.1 when describing the function `fopen`.

For a more comprehensive description on string handling, as well as all string related functions described in this chapter you should refer to the MATLAB® online Product Documentation (The Mathworks 2011a). Similarly, more detailed information on dialog boxes and other user interface functions are available from (The Mathworks 2011b).

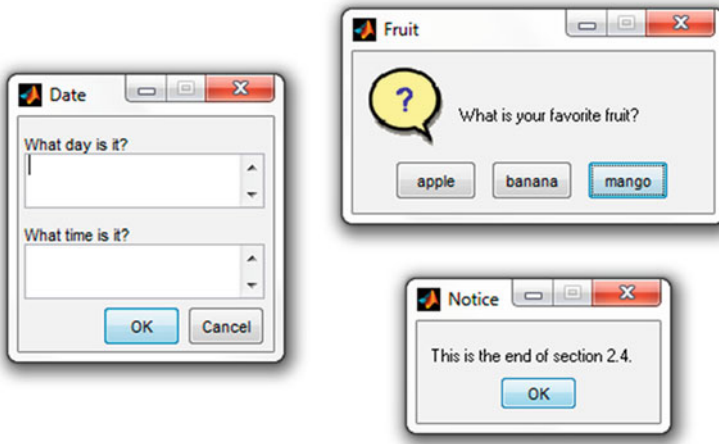


Fig. 2.2 Dialog and message boxes resulting from examples in (2.26). (Reprinted with permission from The MathWorks, Inc.)

2.6 Proposed Exercises

1. Create the $2 \times N$ cell array `ascii_codes` and store into it the corresponding code values and symbols for *ASCII* codes in the range from 32 to 127.
 - Store the numerical values of the codes (integers) in the first row of the cell array, i.e. `ascii_codes{1,n}`.
 - Store the corresponding symbols (characters) in the second row of the cell array, i.e. `ascii_codes{2,n}`.
 - Consider using function `char` for casting code's integer values into characters.
2. Consider a dataset of personal contacts containing the full name, affiliation, phone number and e-mail of each person in the dataset.
 - Create a script for manually entering the data (consider using the functions `inputdlg` and `questdlg`). Collect few data samples (about 10).
 - Write a script for converting the collected data into a structure array.
 - Organize the information in the following five fields: name, surname, affiliation, phone number and e-mail address.
3. Consider the dataset of personal contacts created in the previous exercise.
 - Create a script for converting the structure into a cell array.
 - Retrieve the field names and save them into another cell array.
 - Sort the collection into alphabetical order according to the contact person's surnames (consider using the function `sort`).

- Use function `disp` in combination with function `blanks` to display a header containing the five field names.
 - Just below the header line, generate a printout of the sorted dataset (notice that you will need to implement a loop for doing this).
4. Create a function to convert a list of words between character array and cell array format representations.
 - Use function `input` to manually enter 20 words. Save the words into a cell array (one word per cell).
 - Create the function `convert` to convert the cell array representation into a two-dimensional character array representation (one word per row) and vice versa.
 - Consider using function `char` for converting from cell array into character array representation, and function `cellstr` for converting from character array into cell array.
 - The function should automatically identify whether the input variable is a cell array or a character array. You might use function `iscellstr` or `ischar` to identify the type of variable.
 - Modify the function such that it is able to display an error message in a message box when an input different from a cell array and character array is given.
 5. Consider a dataset containing the following variables: name, gender, age, weight and height, for a given group of people.
 - Create a script for manually collecting the data and storing it into a structure array. Collect all variables as strings. Enter about 10 data samples.
 - Read the age, weight and height from the structure array and convert the strings into numeric values (consider using function `str2num`).
 - Store the converted values into three numerical arrays, one for age, one for weight and one for height.
 - Read the gender from the structure array and convert the strings into binary values, and store the values into a binary array.
 - Compute the basic statistics (mean, standard deviation, maximum and minimum) of age, weight and height for men, women and the whole group.
 - Convert the resulting numeric values into strings (consider using function `num2str`).
 - Generate a report of the results and display it.
 6. Create a function that receives any string and returns the same string with all white spaces “ ” replaced by underscores “_”.

- As a hint, consider the procedure illustrated in (2.12c) for identifying non-white-space characters. In this case, you should identify them and replace them with the underscore character “_”.²
- Alternatively, you can consider the *ASCII* codes for the white space (32) and the underscore (95), and replace one with another.
- Modify the function so that it can receive either a single string or a cell array of strings. In the case it receives a cell array of strings it must return a similar cell array with all white spaces in each individual string replaced by underscores.
- The function should be able to automatically detect whether the input is a single string or a cell array of strings and proceed accordingly.

References

- The MathWorks (2011a) MATLAB product documentation: strings, <http://www.mathworks.com/help/techdoc/ref/strings.html>. Accessed 18 Nov 2011
- The MathWorks (2011b) MATLAB product documentation: dialog, <http://www.mathworks.com/help/techdoc/ref/dialog.html>. Accessed 18 Nov 2011
- The Unicode Consortium (2011) Unicode 6.0.0, <http://www.unicode.org/versions/Unicode6.0.0/>. Accessed 6 Nov 2011

² More details on character replacement will be given in Sect. 4.2.



<http://www.springer.com/978-1-4614-4150-2>

Text Mining with MATLAB®

Banchs, R.E.

2013, XII, 356 p., Hardcover

ISBN: 978-1-4614-4150-2