CHAPTER 7

Text Manipulation

KEY TERMS

character vectors	white space characters	trailing blanks
string arrays	string scalars	delimiter
substring	substring	token
control characters	leading blanks	

CONTENTS

Text can be represented in the MATLAB® software using ***character vectors***, or using ***string arrays***, which were introduced in R2016b.

MATLAB has many built-in functions that are written specifically to manipulate strings and character vectors. Many functions that were created to manipulate character vectors also work on the new **string** type. Additionally, when **string** was introduced in R2016b, many new string-manipulating functions were also introduced. In some cases, strings contain numbers, and it is useful to convert from strings to numbers and vice versa; MATLAB has functions to do this as well.

There are many applications for text data, even in fields that are predominantly numerical. For example, when data files consist of combinations of numbers and characters, it is often necessary to read each line from the file as a string, break the string into pieces, and convert the parts that contain numbers to number variables that can be used in computations. In this chapter, the string manipulation techniques necessary for this will be introduced, and applications in file input/output will be demonstrated in Chapter 9.

7.1 CHARACTERS, CHARACTER VECTORS, AND STRING ARRAYS

Individual characters are stored in single quotation marks, are displayed using single quotes, and are the type **char**. Characters include letters of the alphabet, digits, punctuation marks, white space, and control characters.

MATLAB®. https://doi.org/10.1016/B978-0-12-815479-3.00007-6

Control characters are characters that cannot be printed, but accomplish a task (e.g., a backspace or tab). ***White space characters*** include the space, tab, newline (which moves the cursor down to the next line), and carriage return (which moves the cursor to the beginning of the current line).

```
>> letter = 'x'
letter =
     'x'
>> class(letter)
ans =
     'char'
>> size(letter)
ans =
      1     1
```

In R2016b, a function **newline** was introduced which returns a newline character:

```
>> var = newline
var =
    '
     '
```

Groups of characters, such as words, can be stored in character vectors or in ***string scalars***. Prior to R2016b, the word "*string*" was used when referring to character vectors. However, in R2016b a new **string** type was introduced and, as a result, in MATLAB there is now a distinction between character vectors and strings.

A character vector consists of any number of characters (including, possibly, none), is contained in and displayed using single quotes, and has the type **char**. These are all examples of character vectors:

```
''
' '
'x'
'cat'
'Hello there'
'123'
```

Character vectors are vectors in which every element is a single character, which means that many of the vector operations and functions that we have already seen work with these character vectors.

```
>> myword = 'Hello';
>> class(myword)
ans =
    'char'
```

```
>> size(myword)
ans =
     1     5
>> length(myword)
ans =
     5
>> myword'     % Note transpose
ans =
  5×1 char array
    'H'
    'e'
    'l'
    'l'
    'o'
>> myword(1)
ans =
    'H'
```

A ***string scalar*** can also be used to store a group of characters such as words. String scalars (which means a single string) can be created using the **string** function, or using double quotes (this was introduced in R2017a). String scalars are displayed using double quotes.

```
>> mystr = "Awesome"
mystr =
    "Awesome"
>> mystr = string('Awesome')
mystr =
    "Awesome"
>> class(mystr)
ans =
    'string'
>> size(mystr)
ans =
     1     1
```

Note
Since this is a scalar, the dimensions are 1×1.

Therefore, the **length** of the string is 1. To find the number of characters in a string scalar, the **strlength** function is used:

```
>> strlength(mystr)
ans =
     7
```

Since this is a scalar, the first element is the string itself. Using parentheses to index will show this. However, using curly braces to index will return the

character vector that is contained in the string scalar; this can be used to extract individual characters.

```
>> mystr(1)
ans =
    "Awesome"
>> mystr{1}
ans =
    'Awesome'
>> mystr{1}(2)
ans =
    'w'
```

Groups of strings can be stored in string arrays or character arrays (or, as we will see in Chapter 8, cell arrays).

The new string arrays are the preferred method for storing groups of strings. As with other arrays, string arrays can be created using square brackets. The following creates a row vector of strings.

```
>> majors = ["English", "History", "Engineering"]
majors =
  1×3 string array
    "English"    "History"    "Engineering"
>> class(majors)
ans =
    'string'
>> majors(1)
ans =
    "English"
>> majors{1}
ans =
    'English'
```

The **char** function can be used to create an array of character vectors, e.g.,

```
>> majmat = char('English', 'History', 'Engineering')
majmat =
  3×11 char array
    'English    '
    'History    '
    'Engineering'
```

This is a column vector of strings, which means that it is really a matrix in which every element is a single character. Since every row in a matrix must have the same number of columns, this means that shorter words are padded with extra blank spaces so that they all have the same length. This is one reason that this is not a preferred method for storing groups of strings.

There are several terms that can be used for either strings or character vectors. A ***substring*** is a subset or part of a string. For example, "there" is a substring within the string "Hello there". ***Leading blanks*** are blank spaces at the beginning of a string, for example, " hello", and ***trailing blanks*** are blank spaces at the end of a string.

7.2 OPERATIONS ON TEXT

MATLAB has many built-in functions that work with strings and character vectors. Most of these functions, including those that were present in earlier versions as well as the new functions introduced with the new **string** type, can operate on either strings or character vectors. A few work with either strings or character vectors, but not both. Some of the text manipulation functions that perform the most common operations will be described here.

7.2.1 Operations on Character Vectors

Character vectors are created using single quotes, as we have seen. The **input** function is another method of creating a character vector:

```
>> phrase = input('Enter something: ', 's')
Enter something: hello there
phrase =
     'hello there'
```

Another function that creates only character vectors is the **blanks** function, which creates a character vector consisting of *n* blank characters.

```
>> b = blanks(4)
b =
    '    '
```

Displaying the transpose of the result from the **blanks** function can also be used to move the cursor down. In the Command Window, it would look like this:

```
>> disp(blanks(4)')



>>
```

Another example is to insert blank spaces into a character vector:

```
>> ['Space' blanks(10) 'Cowboy']
ans =
     'Space          Cowboy'
```

The **char** function creates a character array, which is a matrix of individual characters. As has been mentioned, however, it is better to use a string array.

PRACTICE 7.1

Prompt the user for a character vector. Print the length of the character vector and also its first and last characters. Make sure that this works regardless of what the user enters.

7.2.2 Operations on Strings

String scalars and string arrays can be created using double quotes, as we have seen. The **string** function is another method of creating a string from a character vector.

```
>> shout = string('Awesome')
shout =
     "Awesome"
```

Without any arguments, the **string** function creates a string scalar that contains no characters. However, since it is a scalar, it is not technically empty. The **strlength** function should be used to determine whether a string contains any characters, not the **isempty** function.

```
>> es = string
es =
     ""
>> isempty(es)
ans =
     0
>> strlength(es) == 0
ans =
     1
```

The **plus** function or operator can join, or concatenate, two strings together:

```
>> "hello" + " goodbye"
ans =
     "hello goodbye"
```

PRACTICE 7.2

Prompt the user for a character vector. Use the **string** function to convert it to a string. Print the length of the string and also its first and last characters. Concatenate "!!" to the end of your string using the plus operator.

7.2.3 Operations on Strings or Character Vectors

Most functions can have either strings or character vectors as input arguments. Unless specified otherwise, for text-manipulating functions, if the argument is a

character vector, the result will be a character vector, and if the argument is a string, the result will be a string.

7.2.3.1 Creating and Concatenating

We have already seen several methods of creating and concatenating both strings and character vectors, including putting them in square brackets. The **strcat** function can be used to concatenate text horizontally, meaning it results in one longer piece of text. One difference is that it will remove trailing blanks (but not leading blanks) for character vectors, whereas it will not remove either from strings.

Note

In some explanations, the word "string" will be used generically to mean either a MATLAB **string**, or a character vector.

```
>> strcat('Hello', ' there')
ans =
    'Hello there'
>> strcat('Hello ', 'there')
ans =
    'Hellothere'
>> strcat('Hello', ' ', 'there')
ans =
    'Hellothere'
>> strcat("Hello", "there")
ans =
    "Hellothere"
>> strcat("Hello", " ", "there")
ans =
    "Hello there"
```

The **sprintf** function can be used to create customized strings or character vectors. The **sprintf** function works exactly like the **fprintf** function, *but instead of printing it creates a string (or character vector)*. Here are several examples in which the output is not suppressed, so the value of the resulting variable is shown:

```
>> sent1 = sprintf('The value of pi is %.2f', pi)
sent1 =
    'The value of pi is 3.14'

>> sent2 = sprintf("Some numbers: %5d, %2d", 33, 6)
sent2 =
    "Some numbers:    33,    6"
>> strlength(sent2)
ans =
    23
```

Note

In the first example, the format specifier used a character vector, so the result was a character vector; whereas the second example used a string for the format specifier, so the result was a string.

All of the formatting options that can be used in the **fprintf** function can also be used in the **sprintf** function.

One very useful application of the **sprintf** function is to create customized text, including formatting and/or numbers that are not known ahead of time (e.g., entered by the user or calculated). This customized text can then be passed to other functions, for example, for plot titles or axis labels. For example, assume that a file "expnoanddata.dat" stores an experiment number, followed by the experiment data. In this case, the experiment number is "123", and then the rest of the file consists of the actual data.

```
123    4.4    5.6    2.5    7.2    4.6    5.3
```

The following script would load these data and plot them with a title that includes the experiment number.

plotexpno.m

```
% This script loads a file that stores an experiment number
% followed by the actual data. It plots the data and puts
% the experiment # in the plot title

load expnoanddata.dat
experNo = expnoanddata(1);
data = expnoanddata(2:end);
plot(data,'ko')
xlabel('Sample #')
ylabel('Weight')
title(sprintf('Data from experiment %d', experNo))
```

The script loads all numbers from the file into a row vector. It then separates the vector; it stores the first element, which is the experiment number, in a variable *experNo*, and the rest of the vector in a variable *data* (the rest being from the second element to the end). It then plots the data, using **sprintf** to create the title, which includes the experiment number as seen in Figure 7.1.

PRACTICE 7.3

In a loop, create and print strings with file names "file1.dat", "file2.dat", and so on for file numbers 1 through 5.

Another way of accomplishing this (in a script or function) would be:

```
fprintf('%s, Enter your id #: ',username);
id_no = input('');
```

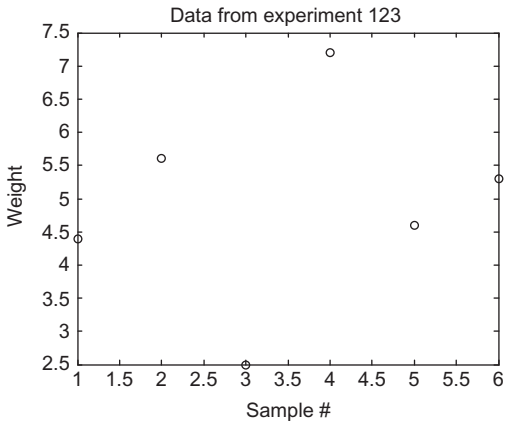

FIGURE 7.1
Customized title in plot using **sprintf**.

QUICK QUESTION!

How could we use the **sprintf** function to customize prompts for the **input** function?

Answer: For example, if it is desired to have the contents of a string variable printed in a prompt, **sprintf** can be used:

```
>> username = input('Please enter your name: ', 's');
Please enter your name: Bart
>> prompt = sprintf('%s, Enter your id #: ',username);
>> id_no = input(prompt)
Bart, Enter your id #: 177
id_no =
    177
```

Note that the calls to the **sprintf** and **fprintf** functions are identical except that the **fprintf** prints (so there is no need for a prompt in the **input** function), whereas the **sprintf** creates a string that can then be displayed by the **input** function. In this case, using **sprintf** seems cleaner than using **fprintf** and then having an empty string for the prompt in **input**.

As another example, the following program prompts the user for endpoints (x_1, y_1) and (x_2, y_2) of a line segment and calculates the midpoint of the line segment, which is the point (x_m, y_m). The coordinates of the midpoint are found by:

$$x_m = \frac{1}{2}(x_1 + x_2) \quad y_m = \frac{1}{2}(y_1 + y_2)$$

The script *midpoint* calls a function *entercoords* to separately prompt the user for the x and y coordinates of the two endpoints, calls a function *findmid* twice to calculate separately the x and y coordinates of the midpoint, and then prints this midpoint. When the program is executed, the output looks like this:

```
>> midpoint
Enter the x coord of the first endpoint: 2
Enter the y coord of the first endpoint: 4
Enter the x coord of the second endpoint: 3
Enter the y coord of the second endpoint: 8
The midpoint is (2.5, 6.0)
```

In this example, the word 'first' or 'second' is passed to the *entercoords* function so that it can use whichever word is passed in the prompt. The prompt is customized using **sprintf**.

midpoint.m

```
% This program finds the midpoint of a line segment

[x1, y1] = entercoords('first');
[x2, y2] = entercoords('second');

midx = findmid(x1,x2);
midy = findmid(y1,y2);

fprintf('The midpoint is (%.1f, %.1f)\n',midx,midy)
```

entercoords.m

```
function [xpt, ypt] = entercoords(word)
% entercoords reads in & returns the coordinates of
%    the specified endpoint of a line segment
% Format: entercoords(word) where word is 'first'
%         or 'second'

prompt = sprintf('Enter the x coord of the %s endpoint: ', ...
       word);
xpt = input(prompt);

prompt = sprintf('Enter the y coord of the %s endpoint: ', ...
       word);
ypt = input(prompt);
end
```

findmid.m

```
function mid = findmid(pt1,pt2)
% findmid calculates a coordinate (x or y) of the
%  midpoint of a line segment
% Format: findmid(coord1, coord2)

mid = 0.5 * (pt1 + pt2);
end
```

7.2.3.2 Removing Characters

MATLAB has functions that will remove trailing and/or leading blanks from strings and character vectors and also will delete specified characters and substrings.

The **deblank** function will remove trailing blank spaces from the end of text (but it does not remove leading blanks).

```
>> deblank("  Hello ")
ans =
    "  Hello"
```

The **strtrim** function will remove both leading and trailing blanks from text, but not blanks in the middle. In the following example, the three blanks in the beginning and four blanks in the end are removed, but not the two blanks in the middle.

```
>> strtrim("   Hello  there    ")
ans =
    "Hello  there"
>> strlength(ans)
ans =
    12
```

The **strip** function can be used to remove leading and/or trailing characters, either whitespace or other specified characters. One simple method of calling it follows:

```
>> strip("xxxHello there!x", "x")
ans =
    "Hello there!"
```

The **erase** function removes all occurrences of a substring within a string (or character vector).

```
>> erase("xxabcxdefgxhijxxx","x")
ans =
    "abcdefghij"
```

7.2.3.3 Changing Case

MATLAB has two functions that convert text to all uppercase letters, or lowercase, called **upper** and **lower**.

```
>> mystring = "AbCDEfgh";
>> lower(mystring)
ans =
    "abcdefgh"
>> upper('Char vec')
ans =
    'CHAR VEC'
```

PRACTICE 7.4

Assume that these expressions are typed sequentially in the Command Window. Think about it, write down what you think the results will be, and then verify your answers by actually typing them.

```
lnstr = '1234567890';
mystr = '  abc    xy';
newstr = strtrim(mystr)
length(newstr)
upper(newstr(1:3))
numstr = sprintf("Number is %4.1f", 3.3)
erase(numstr,"  ") % Note 2 spaces
```

7.2.3.4 Comparing Strings

There are several functions that compare strings or character vectors and return **logical true** if they are equivalent, or **logical false** if not. The function **strcmp** compares text, character by character. It returns **logical true** if the strings (or character vectors) are completely identical (which infers that they must also be of the same length), or **logical false** if they are not of the same length or any corresponding characters are not identical. Note that for character vectors, these functions are used to determine whether two character vectors are equal to each other or not, not the equality operator ==. Here are some examples of these comparisons:

```
>> word1 = 'cat';
>> word2 = 'car';
>> word3 = 'cathedral';
>> word4 = 'CAR';
>> strcmp(word1,word3)
ans =
     0
>> strcmp(word1,word1)
```

```
ans =
     1
>> strcmp(word2,word4)
ans =
     0
```

The function **strncmp** compares only the first *n* characters in strings and ignores the rest. The first two arguments are the strings to compare and the third argument is the number of characters to compare (the value of *n*).

```
>> strncmp(word1,word3,3)
ans =
     1
```

QUICK QUESTION!

How can we compare strings (or character vectors), ignoring whether the characters are uppercase or lowercase?

Answer: See the following Programming Concept and Efficient Method.

THE PROGRAMMING CONCEPT

Ignoring the case when comparing strings can be done by changing all characters in the strings to either upper or lowercase, for example, in MATLAB using the **upper** or **lower** function:

```
>> strcmp(upper(word2),upper(word4))
ans =
     1
```

THE EFFICIENT METHOD

The function **strcmpi** compares the strings but ignores the case of the characters.

```
>> strcmpi(word2,word4)
ans =
     1
```

There is also a function **strncmpi,** which compares *n* characters, ignoring the case.

Note

For character vectors, the equality operator will compare character by character, or it will throw an error message if the vectors are not of the same length.

```
>> 'hello' == 'hello'
ans =
  1×5 logical array
    1   1   1   1   1
>> 'hello' == 'help'
Matrix dimensions must agree.
```

For strings, however, the equality operator will return simply 1 for true if the two strings are exactly the same, or 0 for false if not.

```
>> "hello" == "hello"
ans =
     1
>> "hello" == "help"
ans =
     0
```

7.2.3.5 Finding, Replacing, and Separating Strings

There are functions that find and replace strings, or parts of strings, within other strings and functions that separate strings into substrings.

The function **strfind** receives two strings as input arguments. The general form is **strfind(string, substring)**; it finds all occurrences of the substring within the string and returns the subscripts of the beginning of the occurrences. The substring can consist of one character, or any number of characters. If there is more than one occurrence of the substring within the string, **strfind** returns a vector with all indices. Note that what is returned is the index of the beginning of the substring.

```
>> strfind('abcde', 'd')
ans =
     4
>> strfind("abcde", "bc")
ans =
     2
>> strfind('abcdeabcdedd', 'd')
ans =
     4     9     11     12
```

Note that the arguments can be character vectors or strings. If there are no occurrences, the empty vector is returned.

```
>> strfind('abcdeabcde','ef')
ans =
     []
```

The function **strrep** finds all occurrences of a substring within a string and replaces them with a new substring. The order of the arguments matters. The format is

```
strrep(string, oldsubstring, newsubstring)
```

The following replaces all occurrences of the substring 'e' with the substring 'x':

```
>> strrep('abcdeabcde','e','x')
ans =
    'abcdxabcdx'
```

All strings can be of any length, and the lengths of the old and new substrings do not have to be the same. If the old substring is not found, nothing is changed in the original string.

The function **count** counts the number of occurrences of a substring within a string (or character vector).

```
>> count('xxhelloxxxhix','x')
ans =
     6
>> count("hello everyone", " ")
ans =
     1
>> count("hello everyone", "el")
ans =
     1
```

Note: the empty string (or character vector) is considered to be a substring within every string (or character vector). In fact, there is an empty string at the beginning of every string, at the end of every string, and in between every two characters within the string!

```
>> count("hello", "")
ans =
     6
>> count('abc', '')
ans =
     4
```

In addition to the string functions that find and replace, there is a function that separates a string into two substrings. The **strtok** function breaks a string into two pieces; it can be called several ways. The function receives one string as an input argument. It looks for the first ***delimiter***, which is a character or set of characters that act as a separator within the string.

By default, the delimiter is any white space character. The function returns a ***token*** that is the beginning of the string, up to (but not including) the first delimiter. It also returns the rest of the string, which includes the delimiter. Assigning the returned values to a vector of two variables will capture both of these. The format is

```
[token, rest] = strtok(string)
```

where *token* and *rest* are variable names. For example,

```
>> sentence1 = "Hello there";
>> [word, rest] = strtok(sentence1)
word =
     "Hello"
rest =
     " there"
>> strlength(word)
ans =
      5
>> strlength(rest)
ans =
      6
```

Note that the rest of the string includes the blank space delimiter.

Alternate delimiters can be defined. The format

```
[token, rest] = strtok(string, delimeters)
```

returns a token that is the beginning of the string, up to the first character contained within the delimiters string, and also the rest of the string. In the following example, the delimiter is the character 'l'.

```
>> [word, rest] = strtok(sentence1,'l')
word =
     "He"
rest =
     "llo there"
```

Leading delimiter characters are ignored, whether it is the default white space or a specified delimiter. For example, the leading blanks are ignored here:

```
>> [firstpart, lastpart] = strtok(' materials science')
firstpart =
     'materials'
lastpart =
     ' science'
```

QUICK QUESTION!

What do you think **strtok** returns if the delimiter is not in the string?

Answer: The first result returned will be the entire input argument. The second result depends on whether the input argument is a string or character vector. If it is a string, the second result will be a string scalar that contains no characters, whereas if it is a character vector the result will be an empty character vector.

```
>> [first, rest] = strtok("ABCDE")
first =
    "ABCDE"
rest =
    ""
>> [first, rest] = strtok('ABCDE')
first =
    'ABCDE'
rest =
  0×0 empty char array
```

PRACTICE 7.5

Think about what would be returned by the following sequence of expressions and statements, and then type them into MATLAB to verify your results.

```
dept = "Electrical";
strfind(dept,'e')

strfind(lower(dept),'e')

phone_no = '703-987-1234';
[area_code, rest] = strtok(phone_no,'-')

rest = rest(2:end)

strcmpi('Hi','HI')
```

QUICK QUESTION!

The function **date** returns the current date as a string (e.g., '10-Dec-2017'). How could we write a function to return the day, month, and year as separate output arguments?

Answer: We could use **strrep** to replace the '-' characters with blanks and then use **strtok** with the blank as the default delimiter to break up the string (twice) or, more simply, we could just use **strtok** and specify the '-' character as the delimiter.

separatedate.m

```
function [todayday, todaymo, todayyr] = separatedate
% separatedate separates the current date into day,
% month, and year
% Format: separatedate or separatedate()

[todayday, rest] = strtok(date,'-');
[todaymo, todayyr] = strtok(rest,'-');
todayyr = todayyr(2:end);
end
```

Continued

QUICK QUESTION!—CONT'D

As we need to separate the string into three parts, we need to use the **strtok** function twice. The first time the string is separated into '10' and '-Dec-2017' using **strtok**. Then, the second string is separated into 'Dec' and '-2017' using **strtok**. (As leading delimiters are ignored the second '-' is found as the delimiter in '-Dec-2017'.) Finally, we need to remove the '-' from the string '-2017'; this can be done by just indexing from the second character to the end of the string.

An example of calling this function follows:

```
>> [d, m, y] = separatedate
d =
10
m =
Dec
y =
2017
```

Note that no input arguments are passed to the *separatedate* function; instead, the **date** function returns the current date as a string. Also, note that all three output arguments are strings.

7.2.4 Operations on String Arrays

So far, we have focused mostly on individual "strings," stored in either string scalars or character vectors. In this section, we will see that there are operations and functions that can be applied to all string scalars that are stored in string arrays.

A string array can be preallocated using the **strings** function, e.g.,

```
>> sa = strings(2,4)
sa =
  2×4 string array
    ""    ""    ""    ""
    ""    ""    ""    ""
```

Strings could then be stored in the individual elements by indexing into the array.

The lengths of all strings in a string array can be found using the **strlength** function.

```
>> majors = ["English", "History", "Engineering"];
>> strlength(majors)
ans =
     7     7    11
```

In fact, many string functions can have a string array as an input argument and will return the function of each element in the array. For example, we could convert all to uppercase:

```
>> upmaj = upper(majors)
upmaj =
     "ENGLISH"    "HISTORY"    "ENGINEERING"
```

The plus operator can be used to concatenate the same string to all strings, or a subset of strings determined by indexing, in a string array.

```
>> "BA in " + majors(1:2)
ans =
     "BA in English"     "BA in History"
```

Two string arrays can also be concatenated, as long as they have the same length (note: this means the same number of elements in the string arrays, not the lengths of individual strings within them).

```
>> degrees = ["BA" "BA" "BS"];
>> (degrees + " in " + majors)'   % Note transpose
ans =
  3×1 string array
     "BA in English"
     "BA in History"
     "BS in Engineering"
```

A function that joins strings together (but not character vectors) is **strjoin**, which will concatenate strings in a string array together (putting spaces in between them); **strsplit** does the reverse:

```
>> majlist = strjoin(majors)
majlist =
     "English History Engineering"
>> strsplit(majlist)
ans =
  1×3 string array
     "English"    "History"    "Engineering"
```

The function **join** will concatenate strings in corresponding elements of columns in a string array, e.g.:

```
>> newsa = [degrees; majors]'
newsa =
  3×2 string array
     "BA"    "English"
     "BA"    "History"
     "BS"    "Engineering"
>> join(newsa)
ans =
  3×1 string array
     "BA English"
     "BA History"
     "BS Engineering"
```

7.3 THE "IS" FUNCTIONS FOR TEXT

There are several "is" functions for strings and character vectors, which return **logical true** or **false**. The function **isletter** returns **logical true** for every character in a character vector if the character is a letter of the alphabet or **false** if not. The function **isspace** returns **logical true** for every character in a character vector that is a white space character.

```
>> isletter('EK125')
ans =
     1     1     0     0     0

>> isspace('a b')
ans =
     0     1     0
```

The **ischar** function will return **logical true** if the vector argument is a character vector, or **logical false** if not.

```
>> vec = 'EK125';
>> ischar(vec)
ans =
     1
>> vec = 3:5;
>> ischar(vec)
ans =
     0
>> ischar("EK125")
ans =
     0
```

The **isstring** function will return **logical true** if the vector argument is a string, or **logical false** if not.

```
>> isstring("EK125")
ans =
     1
>> isstring('hello')
ans =
     0
```

The **isStringScalar** will return **logical true** if the vector argument is a string scalar (a string array with only one element), or **logical false** if not.

```
>> isStringScalar("hello")
ans =
     1
>> isStringScalar(["hello" "hi"])
ans =
     0
```

The **isstrprop** function determines whether the characters in a string are in a category specified by a second argument. For example, the following tests to see whether or not the characters are alphanumeric; all are except for the dot '.'.

```
>> isstrprop('AB123.4','alphanum')
ans =
     1     1     1     1     1     0     1
```

There are several other true/false functions that do not start with "is". The **contains** function will return **logical true** if a specified substring is within a string (or character vector), or **logical false** if not.

```
>> contains("hello", "ll")
ans =
     1
>> contains("hello", "x")
ans =
     0
>> majors = ["English", "History", "Engineering"];
>> contains(majors, "Eng")
ans =
  1×3 logical array
     1    0    1
```

The **endsWith** and **startsWith** functions will return **logical true** if a string ends with (or starts with, respectively) a specified string, or **logical false** if not.

```
>> endsWith("filename.dat", ".dat")
ans =
      1
>> startsWith('abcde', 'b')
ans =
     0
```

Recall that every string starts with and ends with the empty string.

```
>> endsWith("abc", "")
ans =
     1
```

7.4 CONVERTING BETWEEN TEXT AND NUMBER TYPES

MATLAB has several functions that convert numbers to strings or character vectors and vice versa. (Note that these are different from the functions such as **char** and **double** that convert characters to ASCII equivalents and vice versa.)

To convert numbers to character vectors, MATLAB has the functions **int2str** for integers and **num2str** for real numbers (which also works with integers). The function **int2str** would convert, for example, the integer 38 to the character vector '38'.

```
>> num = 38;
>> cv1 = int2str(num)
cv1 =
    '38'
>> length(num)
ans =
     1
>> length(cv1)
ans =
     2
>> vec = 2:5;
>> result = int2str(vec)
result =
    '2  3  4  5'
```

The variable *num* is a scalar that stores one number, whereas *cv1* is a character vector that stores two characters, '3' and '8'.

The **num2str** function, which converts real numbers, can be called in several ways. If only one real number is passed to the **num2str** function, it will create a character vector that has four decimal places, which is the default in MATLAB for displaying real numbers. The precision can also be specified (which is the number of digits), and format specifiers can also be passed, as shown in the following:

```
>> cv2 = num2str(3.456789)
cv2 =
    '3.4568'
>> length(cv2)
ans =
     6
>> cv3 = num2str(3.456789, 3)
cv3 =
    '3.46'
>> cv4 = num2str(3.456789, '%6.2f')
cv4 =
    '3.46'
```

Note

In the last example, MATLAB removed the leading blanks from the character vector.

The functions **str2double** and **str2num** do the reverse; they take a character vector in which number(s) are stored and convert them to the type **double**:

```
>> num = str2double('123.456')
num =
  123.4560
```

If there is a string in which there are numbers separated by blanks, the **str2num** function will convert this to a vector of numbers (of the default type **double**). For example,

```
>> mycv = '66 2 111';
>> numvec = str2num(mycv)
numvec =
    66    2    111
>> size(numvec)
ans =
     1    3
```

The **str2double** function is a better function to use in general than **str2num**, but it can only be used when a scalar is passed; it would not work, for example, for the variable *mycv* above.

The **str2double** and **str2num** functions perform the same operations on strings. To convert numbers to strings, the function **string** can be used.

```
>> num = 38;
>> st1 = string(num)
st1 =
    "38"
>> vec = 2:5;
>> starr = string(vec)
starr =
  1×4 string array
    "2"    "3"    "4"    "5"
```

PRACTICE 7.6

Think about what would be returned by the following sequence of expressions and statements, and then type them into MATLAB to verify your results.

```
vec = 'yes or no';
isspace(vec)

all(isletter(vec) ~= isspace(vec))

ischar(vec)

nums = [33 1.5];
num2str(nums)

nv = num2str(nums)

sum(nums)

string([11 33])
```

Explore Other Interesting Features

In many of the search and replace functions, search patterns can be specified which use ***regular expressions***. Use **help** to find out about these patterns.

Explore the **replace** and **replaceBetween** functions, which find and replace.

Explore the **split** and **splitlines** functions, which split text.

Explore the **extractAfter**, **extractBefore**, and **extractBetween** functions, which extract substrings.

Explore the **insertAfter** and **insertBefore** functions, which insert text.

Explore the **sscanf** function, which reads data from a string.

Explore the **strjust** function, which justifies a string or character vector.

Explore the **mat2str** function, to convert from a matrix to a character vector. ■

SUMMARY

COMMON PITFALLS

- Putting arguments to **strfind** in incorrect order.
- Trying to use == to compare character vectors for equality, instead of the **strcmp** function (or its variations)
- Confusing **sprintf** and **fprintf**. The syntax is the same, but **sprintf** creates a string whereas **fprintf** prints.
- Trying to create a vector of strings with varying lengths (one way is to use **char** which will pad with extra blanks automatically; a better way is to use a string array).
- Forgetting that when using **strtok**, the second argument returned (the "rest") contains the delimiter.
- When breaking a string into pieces, forgetting to convert the numbers in the strings to actual numbers that can then be used in calculations.

PROGRAMMING STYLE GUIDELINES

- Make sure the correct string comparison function is used, for example, **strcmpi** if ignoring case is desired.

MATLAB Functions and Commands			
string	upper	strings	endsWith
newline	lower	strjoin	startsWith
strlength	strcmp	strsplit	int2str
blanks	strncmp	join	num2str
plus	strcmpi	isletter	str2double
strcat	strncmpi	isspace	str2num
sprintf	strfind	ischar	
deblank	strrep	isstring	
strtrim	count	isStringScalar	
strip	strtok	isstrprop	
erase	date	contains	

Exercises

1. A file name is supposed to be in the form *filename.ext*. Write a function that will determine whether text is in the form of a name followed by a dot followed by a three-character extension, or not. The function should return 1 for **logical true** if it is in that form, or 0 for **false** if not. The function should work with both character vector and string arguments.
2. The following script calls a function *getstr* that prompts the user for a string, error-checking until the user enters something (the error would occur if the user just hits the Enter key without characters other than white space characters first). The script then prints the length of the string. Write the *getstr* function.

```
thestring = getstr();
fprintf('Thank you, your string is %d characters long\n', ...
       length(thestring))
```

3. Write a script that will, in a loop, prompt the user for four course numbers. Each will be a character vector of length 5 of the form 'CS101'. These strings are to be stored in a character matrix.
4. Write a function that will generate two random integers, each in the inclusive range from 10 to 30. It will then return a character vector consisting of the two integers joined together, e.g., if the random integers are 11 and 29, the character vector that is returned will be '1129'.
5. Modify the function from Exercise 4 to return a string instead.
6. Write a script that will create x and y vectors. Then, it will ask the user for a color ('red', 'blue', or 'green') and for a plot style (circle or star). It will then create a character vector *pstr* that contains the color and plot style, so that the call to the **plot** function would be: **plot(x,y,pstr)**. For example, if the user enters 'blue' and '*', the variable *pstr* would contain 'b*'.

7. Assume that you have the following function and that it has not yet been called.

strfunc.m

```
function strfunc(instr)
persistent mystr
if isempty(mystr)
    mystr = '';
end
mystr = strcat(instr,mystr);
fprintf('The string is %s\n',mystr)
end
```

What would be the result of the following sequential expressions?

```
strfunc('hi')
strfunc("hello")
```

Note that the argument can be either a character vector or string.

8. Create a string array that contains pet types, e.g.,

```
pets = ["cat" "dog" "gerbil"];
```

Show the difference in the following methods of indexing into the first two strings:

```
pets(1:2)
pets{1:2}
[p1 p2] = pets{1:2}
```

9. Show the difference between assigning an empty vector to an element in a string array, by using parentheses and by using curly braces to index into the element.
10. Explain in words what the following function accomplishes (not step-by-step, but what the end result is).

dostr.m

```
function out = dostr(inp)
persistent str
[w, r] = strtok(inp);
str = strcat(str,w);
out = str;
end
```

11. Write a function that will receive a name and department as separate character vectors and will create and return a code consisting of the first two letters of the name and the last two letters of the department. The code should be uppercase letters. For example,

```
>> namedept('Robert','Mechanical')
ans =

       'ROAL'
```

12. Modify the function from Exercise 11 to receive string inputs and return a string instead of a character vector.
13. Write a function "createUniqueName" that will create a series of unique names. When the function is called, a string or character vector is passed as an input argument. The functions adds an integer to the end of the input and returns the resulting text. Every time the function is called, the integer that it adds is incremented. Here are some examples of calling the function:

```
>> createUniqueName("myvar")
ans =
 "myvar1"
>> createUniqueName('filename')
ans =
'filename2'
```

14. What does the **blanks** function return when a 0 is passed to it? A negative number? Write a function *myblanks* that does exactly the same thing as the **blanks** function, using the programming method. Here are some examples of calling it:

```
>> fprintf('Here is the result:%s!\n', myblanks(0))
Here is the result:!

>> fprintf('Here is the result:%s!\n', myblanks(7))
Here is the result:    !
```

15. Write a function that will prompt the user separately for a filename and extension and will create and return a string with the form 'filename.ext'.
16. Write a function that will receive one input argument, which is an integer *n*. The function will prompt the user for a number in the range from 1 to *n* (the actual value of *n* should be printed in the prompt) and return the user's input. The function should error-check to make sure that the user's input is in the correct range.
17. Write a script that will generate a random integer, ask the user for a field width, and print the random integer with the specified field width. The script will use **sprintf** to create a string such as "The # is %4d\n" (if, for example, the user entered 4 for the field width), which is then passed to the **fprintf** function. To print (or create a string using **sprintf**) either the % or \ character, there must be two of them in a row.
18. Most of a modular program has been written that will calculate and print the volume of a cone. The script "coneprog" calls a function that prompts for the

radius and the height, error-checking for each, another function to calculate the volume, and then a third function that prints the results. The script and the "conevol" function have been written for you. A function stub has been written for the "printvol" function; you do not need to modify that. You are to fill in the "get_vals" function, which must use a local function "readitin".

coneprog.m

```
[rad ht] = get_vals('radius', 'height');
vol = conevol(rad, ht);
printvol(rad, ht, vol)
```

```
function vol = conevol(rad, ht)
vol = (pi/3) * rad .^2 .* ht;
end
```

```
function printvol(rad, ht, vol)
disp(rad)
disp(ht)
disp(vol)
end
```

get_vals.m

```
function [out1, out2] = get_vals(first, second)
out1 = readitin(first);
out2 = readitin(second);
end
function out = readitin(word)
```

19. We are writing a program that will call a function to prompt the user for an input vector *x*, call a function to calculate a *y* vector from *x*, and then call a function to print the *x* and *y* vectors, and plot them with the length in the title (see Figure 7.2). You are given the script *custtitle* and stubs for the *getthex* and *calclyfromx* functions. You are to write the *dispandplot* function.

custtitle.m

```
x = getthex();
y = calcyfromx(x);
dispandplot(x,y)
```

```
function x = getthex
x = -pi:0.9:pi;
end
```

```
function y = calcyfromx(x)
y = sin(x);
end
```

Here is an example of running the script:

```
>> custtitle
```

```
The x vector is:

   -3.1416   -2.2416   -1.3416   -0.4416    0.4584    1.3584    2.2584

The y vector is:

   -0.0000   -0.7833   -0.9738   -0.4274    0.4425    0.9775    0.7728
```

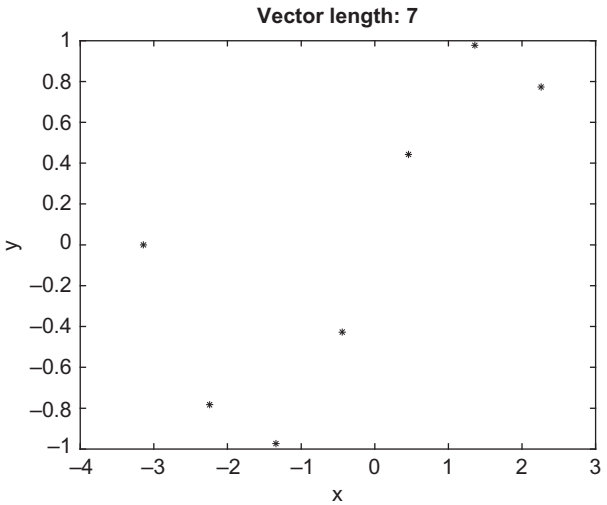

FIGURE 7.2
Plot of x and y vectors.

20. If the strings (or character vectors) passed to **strfind** are of the same length, what are the only two possible results that could be returned?
21. Vectorize this for a character vector *mystrn*:

```
while mystrn(end) == ' '  % Note one space in quotes
     mystrn = mystrn(1:end-1);
end
```

22. Vectorize this for a character vector sentence:

```
loc = findstr(sentence, ' ');
where = loc(1);
first = sentence(1:where-1);
last = sentence(where:end);
```

23. Vectorize this:

```
vec = [];
for i = 1:8
    vec = [ vec ' ']; % one blank space
end
vec  % just for display
```

24. Vectorize this for character vectors str1 and str2:

```
if length(str1) ~= length(str2)
    outlog = false;
else
    outlog = true;
    for i = 1:length(str1)
        if str1(i) ~= str2(i)
            outlog = false;
        end
    end
end
outlog % Just to display the value
```

25. Write a function *nchars* that will create a character vector of n characters, without using any loops or selection statements.

```
>> nchars('*', 6)
ans =
    '******'
```

26. Write a function *rid_multiple_blanks* that will receive a string or character vector as an input argument. The text input contains a sentence that has multiple blank spaces in between some of the words. The function will return the text (as a character vector or a string, depending on what type the input argument was) with only one blank in between words. For example,

```
>> mystr = 'Hello  and how  are  you?';
>> rid_multiple_blanks(mystr)
ans =
'Hello and how are you? '
```

27. Two variables store character vectors that consist of a letter of the alphabet, a blank space, and a number (in the form 'R 14.3'). Write a script that would initialize two such variables. Then, use functions to extract the numbers from the character vectors and add them together.
28. In very simple cryptography, the intended message sometimes consists of the first letter of every word in a string. Write a function *crypt* that will receive

a string or character vector with the message and return the encrypted message.

```
>> estring = 'The early songbird tweets';
>> m = crypt(estring)
m =
'Test'
```

29. Using the functions **char** and **double**, one can shift words. For example, one can convert from lowercase to uppercase by subtracting 32 from the character codes:

```
>> orig = 'ape';
>> new = char(double(orig)-32)
new =
'APE'
>> char(double(new)+32)
ans =
'ape'
```

We've "encrypted" a character vector by altering the character codes. Figure out the original character vector. Try adding and subtracting different values (do this in a loop) until you decipher it:

```
Jmkyvih$mx$syx$}ixC
```

30. Load files named *file1.dat*, *file2.dat*, and so on in a loop. To test this, create just 2 files with these names in your Current Folder first.
31. Create the following three variables:

```
>> var1 = 123;
>> var2 = '123';
>> var3 = "123"
```

Then, add 1 to each of the variables. What is the difference?

32. Use **help isstrprop** to find out what properties can be tested; try some of them on a string variable.
33. Write a script that will first initialize a character vector variable that will store x and y coordinates of a point in the form 'x 3.1 y 6.4'. Then, use text-manipulating functions to extract the coordinates and plot them.
34. Write a function *wordscramble* that will receive a word in a character vector as an input argument. It will then randomly scramble the letters and return the result. Here is an example of calling the function:

```
>> wordscramble('fantastic')
ans =
'safntcait'
```

35. A file called *avehighs.dat* stores for three locations the average high temperatures for each month for a year (rounded to integers). There are three lines in the file; each stores the location number followed by the 12 temperatures (this format may be assumed). For example, the file might store:

```
432  33 37 42 45 53 72 82 79 66 55 46 41
777  29 33 41 46 52 66 77 88 68 55 48 39
567  55 62 68 72 75 79 83 89 85 80 77 65
```

Write a script that will read these data in and plot the temperatures for the three locations separately in one Figure Window. A **for** loop must be used to accomplish this. For example, if the data are as shown in the previous data block, the Figure Window would appear as Figure 7.3. The axis labels and titles should be as shown.

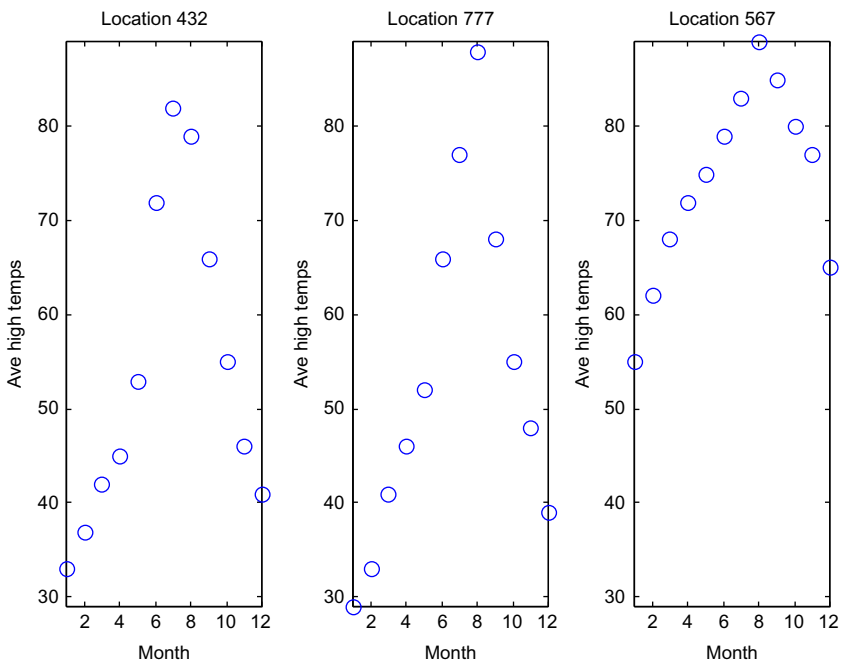

FIGURE 7.3
Subplot to display data from file using a **for** loop.

If you have version R2016a or later, write the script as a live script.

36. Investigate the use of the **date** function. Replace the dashes in the result with spaces.
37. Investigate the use of the **datestr** function, for example, with **datestr(now)**. Extract the date and time separately.