Introduction to MATLAB Programming

String, Cell, Structure

String Manipulation

A *string* in the MATLAB® software consists of any number of characters and is contained in single quotes. Actually, strings are treated as vectors in which every element is a single character, which means that many of the vector operations and functions that we have seen already work with strings. MATLAB also has many built-in functions that are written specifically to manipulate strings. 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, also.

There are many applications for using strings, 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 in Chapter 8 applications in file input/output will be demonstrated.

String variables

A string consists of any number of characters (including, possibly, none). These are examples of strings:

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

A ***substring*** is a subset or part of a string. For example, 'there' is a substring within the string 'Hello there'.

Characters include letters of the alphabet, digits, punctuation marks, white space, and control characters. ***Control characters*** are characters that cannot be printed, but accomplish a task (such as a backspace or tab). ***Whitespace 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). ***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.

Creating string variables

There are several ways that string variables can be created.

One is using assign ment statements:

```
>> word = 'cat';
```

Another method is to read into a string variable. Recall that to read into a string variable using the **input** function, the second argument 's' must be included:

```
>> strvar = input('Enter a string: ', 's')
Enter a string: xyzabc
strvar =
xyzabc
```

If leading or trailing blanks are typed by the user, these will be stored in the string. For example, in the following the user entered four blanks and then 'xyz':

```
>> s = input('Enter a string: ','s')
Enter a string:     xyz
s =
    xyz
```

Strings as vectors (1)

Strings are treated as ***vectors of characters***—or in other words, a vector in which every element is a single character—so many vector operations can be performed. For example, the number of characters in a string can be found using the **length** function:

Expressions can refer to an individual element (a character within the string), or a subset of a string or a transpose of a string:

```
>> mystr = 'Hi';
>> mystr(1)
ans =
H
>> mystr'
ans =
H
i
>> sent = 'Hello there';
>> length(sent)
ans =
    11
>> sent(4:8)
ans =
lo th
```

```
>> length('cat')
ans =
    3
>> length(' ')
ans =
    1
>> length('')
ans =
    0
```

Strings as vectors (2)

A matrix can be created, which consists of strings in each row. So, essentially it is created as a column vector of strings, but the end result is that this would be treated as a matrix in which every element is a character:

```
>> wordmat = ['Hello';'Howdy']
wordmat =
Hello
Howdy
>> size(wordmat)
ans =
     2     5
```

This created a 2×5 matrix of characters.

With a character matrix, we can refer to an individual element, which is a character, or an individual row, which is one of the strings:

```
>> wordmat(2,4)
ans =
d
>> wordmat(1,:)
ans =
Hello
```

Since rows within a matrix must always be the same length, the shorter strings must be padded with blanks so that all strings have the same length, otherwise an error will occur.

```
>> greetmat = ['Hello'; 'Goodbye']
??? Error using ==> vertcat
CAT arguments dimensions are not consistent.
>> greetmat = ['Hello  '; 'Goodbye']
greetmat =
Hello
Goodbye
>> size(greetmat)
ans =
2     7
```

Exercise

Prompt the user for a string. Print the length of the string and also the last character in the string. Make sure that this works regardless of what the user enters.

Operations on strings (1)

Concatenation

String concatenation means to join strings together. Of course, since strings are just vectors of characters, the method of concatenating vectors works for strings, also. For example, to create one long string from two strings, it is possible to join them by putting them in square brackets:

```
>> first = 'Bird';
>> last = 'house';
>> [first last]
ans =
Birdhouse
```

The function **strcat** does this also horizontally, meaning that it creates one longer string from the inputs.

```
>> str1 = 'xxx   ';
>> str2 = '   yyy';
>> [str1 str2]
ans =
xxx      yyy
>> length(ans)
ans =
     12
```

The **strcat** function, however, will remove trailing blanks (but not leading blanks) from strings before concatenating. Notice that in these examples, the trailing blanks from *str1* are removed, but the leading blanks from *str2* are not:

```
>> strcat(str1,str2)
ans =
xxx   yyy
>> length(ans)
ans =
    9
>> strcat(str2,str1)
ans =
   yyyxxx
>> length(ans)
ans =
    9
```

Operations on strings (2)

Concatenation

The function **strvcat** will concatenate vertically, meaning that it will create a column vector of strings.

```
>> strvcat(first,last)
ans =
Bird
house
>> size(ans)
ans =
     2     5
```

Note that **strvcat** will pad with extra blanks automatically, in this case to make both strings have a length of 5.

Exercise

Create the following string variables:

```
v1 = 'Mechanical';
v2 = 'Engineering';
```

Then, get the length of each string. Create a new variable, *v3*, which is a substring of *v2* that stores just 'Engineer'. Create a matrix consisting of the values of *v1* and *v2* in separate rows.

Creating customized strings (1)

Creating Customized Strings

There are several built-in functions that create customized strings, including **char**, **blanks**, and **sprintf**.

We have seen already that the **char** function can be used to convert from an ASCII code to a character, for example:

```
>> char(97)
ans =
a
```

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



>>
```

This is useful in a script or function to create space in output, and is essentially equivalent to printing the newline character four times.

ASCII Punctuation & Symbols	U+003A	:	58	072	Colon	0027
	U+003B	;	59	073	Semicolon	0028
	U+003C	<	60	074	Less-than sign	0029
	U+003D	=	61	075	Equal sign	0030
	U+003E	>	62	076	Greater-than sign	0031
	U+003F	?	63	077	Question mark	0032
	U+0040	@	64	0100	At sign	0033
Latin Alphabet: Uppercase	U+0041	A	65	0101	Latin Capital letter A	0034
	U+0042	B	66	0102	Latin Capital letter B	0035
	U+0043	C	67	0103	Latin Capital letter C	0036
	U+0044	D	68	0104	Latin Capital letter D	0037
	U+0045	E	69	0105	Latin Capital letter E	0038
	U+0046	F	70	0106	Latin Capital letter F	0039
	U+0047	G	71	0107	Latin Capital letter G	0040
	U+0048	H	72	0110	Latin Capital letter H	0041
	U+0049	I	73	0111	Latin Capital letter I	0042
	U+004A	J	74	0112	Latin Capital letter J	0043
	U+004B	K	75	0113	Latin Capital letter K	0044
	U+004C	L	76	0114	Latin Capital letter L	0045
	U+004D	M	77	0115	Latin Capital letter M	0046
	U+004E	N	78	0116	Latin Capital letter N	0047
	U+004F	O	79	0117	Latin Capital letter O	0048
	U+0050	P	80	0120	Latin Capital letter P	0049
	U+0051	Q	81	0121	Latin Capital letter Q	0050
	U+0052	R	82	0122	Latin Capital letter R	0051
	U+0053	S	83	0123	Latin Capital letter S	0052
	U+0054	T	84	0124	Latin Capital letter T	0053
	U+0055	U	85	0125	Latin Capital letter U	0054
	U+0056	V	86	0126	Latin Capital letter V	0055
	U+0057	W	87	0127	Latin Capital letter W	0056
	U+0058	X	88	0130	Latin Capital letter X	0057
	U+0059	Y	89	0131	Latin Capital letter Y	0058

https://en.wikipedia.org/wiki/List_of_Unicode_characters

Creating customized strings (2)

Creating Customized Strings

The **char** function can also be used to create a matrix of characters. When using the **char** function to create a matrix, it will automatically pad the strings within the rows with blanks as necessary so that they are all the same length, just like **strvcat**.

```
>> clear greetmat
>> greetmat = char('Hello','Goodbye')
greetmat =
Hello
Goodbye
>> size(greetmat)
ans =
     2     7
```

The **blanks** function will create a string consisting of n blank characters—which are kind of hard to see here! However, in MATLAB if the mouse is moved to highlight the result in *ans*, the blanks can be seen.

```
>> blanks(4)
ans =

>> length(ans)
ans =
     4
```

Usually this function is most useful when concatenating strings, and you want a number of blank spaces in between. For example, this will insert five blank spaces in between the words:

```
>> [first blanks(5) last]
ans =
Bird     house
```

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

Creating customized strings (3)

This is useful in a script or function to create space in output, and is essentially equivalent to printing the newline character four times.

The **sprintf** function works exactly like the **fprintf** function, but instead of printing it creates a string. Here are several examples in which the output is not suppressed so the value of the string 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
>> length(sent2)
ans =
    23
```

In the following example, on the other hand, the output of the assignment is suppressed so the string is created including a random integer and stored in the string variable. Then, some exclamation points are concatenated to that string.

```
>> phrase = sprintf('A random integer is %d', ...
        randint(1,1,[5,10]));
>> strcat(phrase, '!!!')
ans =
A random integer is 7!!!
```

All the conversion specifiers that can be used in the **fprintf** function can also be used in the **sprintf** function.

Applications of customized strings: prompts, labels, arguments to functions

One very useful application of this is to include numbers in strings, which are used for plot titles and 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
```

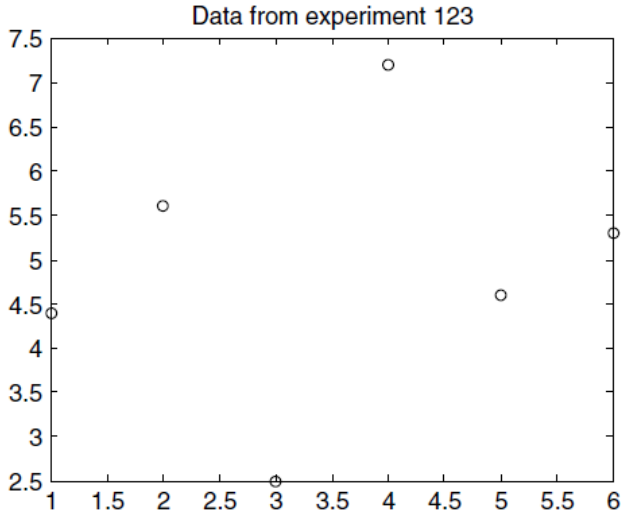
The following script would load this data and plot it with a title that includes the experiment number.

plotexpno.m

```
% This script loads a file that stores an experiment #
% followed by the actual data. It plots the data and puts
% the experiment # in the plot title
load expnoanddata.dat
exper_no = expnoanddata(1);
data = expnoanddata(2:end);
plot(data,'ko')
title(sprintf('Data from experiment %d', exper_no))
```

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 *exper_no*, 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 that includes the experiment number as seen in Figure 6.1.

FIGURE 6.1
Customized title in plot using **sprintf.**



Exercise

How could we use the **sprintf** function in customizing prompts for the **input** function?

Answer: For example, if you want 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
```

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

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

Notice 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**.

fprintf vs. sprintf

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 **sprintf** function generates this customized prompt.

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)
% Prompts the user for the coordinates of an endpoint

% Two different methods are used to customize the
% prompt to show the difference
fprintf('Enter the x coord of the %s endpoint: ', word)
xpt = input('');
prompt = sprintf('Enter the y coord of the %s endpoint: ', ...
    word);
ypt = input(prompt);
```

findmid.m

```
function mid = findmid(pt1,pt2)
% Calculate a coordinate (x or y) of the
% midpoint of a line segment
mid = 0.5 * (pt1 + pt2);
```

Removing whitespace characters

MATLAB has functions that will remove trailing blanks from the end of a string and/or leading blanks from the beginning of a string.

The **deblank** function will remove blank spaces from the end of a string. For example, if some strings are padded in a string matrix so that all are the same length, it is frequently preferred to then remove those extra blank spaces in order to actually use the string.

```
>> names = char('Sue', 'Cathy','Xavier')
names =
Sue
Cathy
Xavier
>> name1 = names(1,:)
name1 =
Sue
>> length(name1)
ans =
     6
>> name1 = deblank(name1);
>> length(name1)
ans =
     3
```

Note: The **deblank** function removes only trailing blanks from a string, not leading blanks.

The **strtrim** function will remove both leading and trailing blanks from a string, but not blanks in the middle of the string. 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. Selecting the result in MATLAB with the mouse would show the blank spaces.

```
>> strvar = [blanks(3) 'xx' blanks(2) 'yy' blanks(4)]
strvar =
   xx  yy
>> length(strvar)
ans =
    13
>> strtrim(strvar)
ans =
xx  yy
>> length(ans)
ans =
     6
```

Changing case

Changing Case

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

```
>> mystring = 'AbCDEfgh';
>> lower(mystring)
ans =
abcdefgh
>> upper(ans)
ans =
ABCDEFGH
```

Comparing Strings

There are several functions that compare strings and return logical true if they are equivalent, or logical false if not. The function **strcmp** compares strings, character by character. It returns logical true if the strings are completely identical (which infers that they must be of the same length, also) or logical false if the strings are not the same length or any corresponding characters are not identical. Here are some examples of these comparisons:

```
>> word1 = 'cat';
>> word2 = 'car';
>> word3 = 'cathedral';
>> word4 = 'CAR';
>> strcmp(word1,word2)
ans =
    0
>> 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
>> strncmp(word1,word3,4)
ans =
    0
```

Exercise

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.

```
wxyzstring = ...
'123456789012345';
longstring = '  abc  de f  '
length(longstring)
shortstring =
strtrim(longstring)
length(shortstring)
upper(shortstring)
news = sprintf('The first part is %s', ...
   shortstring(1:3))
```

Exercise

How can we compare strings, ignoring whether the characters in the string are uppercase or lowercase?

Answer: See the Programming Concept and Efficient Method below.

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** that compares *n* characters, ignoring the case.

Finding, replacing, and separating strings (1)

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

The function **findstr** receives two strings as input arguments. It finds all occurrences of the shorter string within the longer, and returns the subscripts of the beginning of the occurrences. The order of the strings does not matter with **findstr**; it will always find the shorter string within the longer, whichever that is. The shorter string can consist of one character, or any number of characters. If there is more than one occurrence of the shorter string within the longer one, **findstr** returns a vector with all indices. Note that what is returned is the index of the beginning of the shorter string.

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

The function **strfind** does essentially the same thing, except that the order of the arguments does make a difference. The general form is **strfind(string, substring)**; it finds all occurrences of the substring within the string, and returns the subscripts.

```
>> strfind('abcdeabcde','e')
ans =
    5     10
```

For both **strfind** and **findstr**, if there are no occurrences, the empty vector is returned.

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

Exercise

How can you find how many blanks there are in a string (e.g., 'how are you')?

Answer: The **strfind** function will return an index for every occurrence of a substring within a string, so the result is a vector of indices. The **length** of this vector of indices would be the number of occurrences. For example, the following finds the number of blank spaces in *phrase*.

```
>> phrase = 'Hello, and how are you doing?';
>> length(strfind(phrase,' '))
ans =
    5
```

If you want to get rid of any leading and trailing blanks first (in case there are any), the **strtrim** function would be used first.

```
>> phrase = '   Well, hello there!   ';
>> length(strfind(strtrim(phrase),' '))
ans =
    2
```

Finding, replacing, and separating strings (2)

Let's expand this, and write a script that creates a vector of strings that are phrases. The output is not suppressed so that the strings can be seen when the script is executed. It loops through this vector and passes each string to a function **countblanks**. This function counts the number of blank spaces in the string, not including any leading or trailing blanks.

phraseblanks.m

```
% This script creates a column vector of phrases
% It loops to call a function to count the number
% of blanks in each one and prints that
phrasemat = char('Hello and how are you?', ...
  'Hi there everyone!', 'How is it going?', 'Whazzup?')
[r c] = size(phrasemat);
for i = 1:r
  % Pass each row (each string) to countblanks function
  howmany = countblanks(phrasemat(i,:));
  fprintf('Phrase %d had %d blanks\n',i,howmany)
end
```

countblanks.m

```
function num = countblanks(phrase)
% Counts the number of blanks in a trimmed string
num = length(strfind(strtrim(phrase), ' '));
```

For example, running this script would result in:

```
>> phraseblanks
phrasemat =
Hello and how are you?
Hi there everyone!
How is it going?
Whazzup?
Phrase 1 had 4 blanks
Phrase 2 had 2 blanks
Phrase 3 had 3 blanks
Phrase 4 had 0 blanks
```

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 example replaces all occurrences of the substring 'e' with the substring 'x':

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

All strings can be any length, and the lengths of the old and new substrings do not have to be the same.

Finding, replacing, and separating strings (3)

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 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 whitespace character. The function returns a ***token***, which 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'
sentence1 =
Hello there
>> [word rest] = strtok(sentence1)
word =
Hello

rest =
 there
>> length(word)
ans =
     5
>> length(rest)
ans =
     6
```

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

By default, the delimiter for the token is a whitespace character (meaning that the token is defined as everything up to the blank space), but 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 whitespace or a specified delimiter. For example, the leading blanks are ignored here:

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

Exercise

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

Answer: The first result returned will be the entire string, and the second will be the empty string.

```
>> [first rest] = strtok ('ABCDE')
first =
ABCDE
rest =
  Empty string: 1-by-0
```

Exercise

The function **date** returns the current date as a string; for example, '07-Feb-2008'. 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.

Here is an example of calling this function:

```
>> [d m y] = separatedate()
d =
07
m =
Feb
y =
2008
```

separatedate.m

```
function [todayday, todaymo, todayyr] = separatedate()
% This function separates the current date into day,
% month, and year
[todayday rest] = strtok(date,'-');
[todaymo todayyr] = strtok(rest,'-');
todayyr = todayyr(2:end);
```

Since 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 '07' and '-Feb-2008' using **strtok**. Then, the second string is separated into 'Feb' and '-2008' using **strtok**. (Since leading delimiters are ignored the second '-' is found as the delimiter in '-Feb-2008'.) Finally, we need to remove the '-' from the string '-2008'; this can be done by just indexing from the second character to the end of the string.

Notice that no input arguments are passed to the function; instead, the **date** function returns the current date as a string.

Evaluating a string

The function **eval** is used to evaluate a string as a function. For example, in the following, the string 'plot(x)' is interpreted to be a call to the **plot** function, and it produces the plot shown in Figure 6.2.

```
>> x = [2 6 8 3];
>> eval('plot(x)')
```

This would be useful if the user entered the name of the type of plot to use. In this example, the string that the user enters (in this case 'bar') is concatenated with the string '(x)' to create the string 'bar(x)'; this is then evaluated as a call to the **bar** function as seen in Figure 6.3. The name of the plot type is also used in the title.

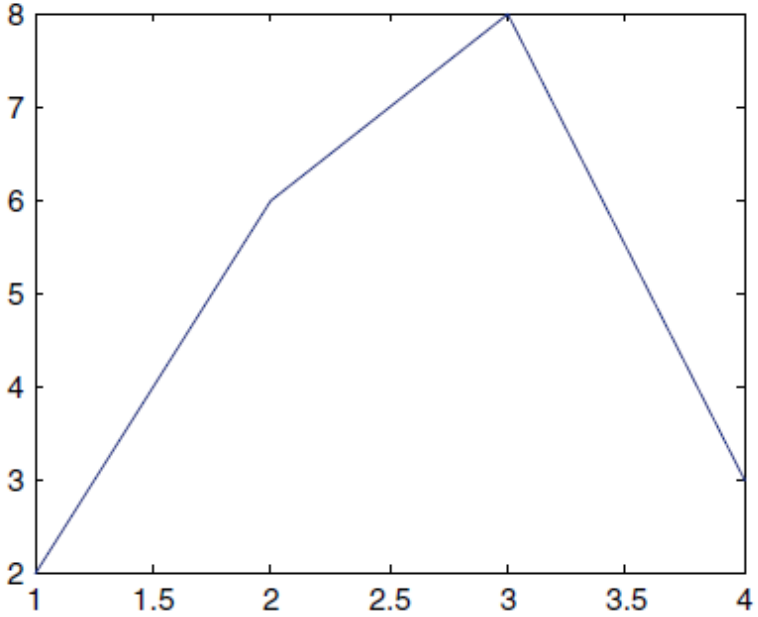


FIGURE 6.2

Plot type passed to the **eval** *function.*

Exercise

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

```
strcmp('hello', 'height')
strncmp('hello', 'height',2)
strncmpi('yes', 'YES', 1)
name = 'Smith, Carly';
ind = findstr(name,',')
first = name(1:ind-1)
last = name(ind+2:end)
[f rest] = strtok(name, ',')
l = rest(3:end)
```

Exercise

Create an *x* vector. Prompt the user for 'sin', 'cos', or 'tan' and create a string with that function of *x* (e.g., 'sin(x)' or 'cos(x)'). Use **eval** to create a *y* vector using the specified function.

Functions for strings

There are several is functions for strings, which return logical true or false. The function **isletter** returns logical true if the character is a letter of the alphabet. The function **isspace** returns logical true if the character is a whitespace character. If strings are passed to these functions, they will return logical true or false for every element, or, in other words, every character.

```
>> isletter('a')
ans =
     1
>> isletter('EK127')
ans =
     1     1     0     0     0
>> isspace('a b')
ans =
     0     1     0
```

The **ischar** function will return logical true if an array is a character array, or logical false if not.

```
>> vec = 'EK127';
>> ischar(vec)
ans =
     1
>> vec = 3:5;
>> ischar(vec)
ans =
     0
```

Converting between string and number

MATLAB has several functions that convert numbers to strings in which each character element is a separate digit, and vice versa. (Note: these are different from the functions **char**, **double**, etc., that convert characters to ASCII equivalents and vice versa.)

To convert numbers to strings, 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 4 to the string '4'.

```
>> rani = randint(1,1,50)
rani =
    38
>> s1 = int2str(rani)
s1 =
38
>> length(rani)
ans =
    1
>> length(s1)
ans =
    2
```

The variable *rani* is a scalar that stores one number, whereas *s1* is a string that stores two characters, '3' and '8'.

Even though the result of the first two assignments is 38, notice that the indentation in the Command Window is different for the number and the string.

The **num2str** function, which converts real numbers, can be called in several ways. If only the real number is passed to the **num2str** function, it will create a string 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 strings can also be passed, as shown:

```
>> str2 = num2str(3.456789)
3.4568
>> length(str2)
ans =
     6
>> str3 = num2str(3.456789,3)
str3 =
3.46
>> str = num2str(3.456789,'%6.2f')
str =
3.46
```

Note that in the last example, MATLAB removed the leading blanks from the string.

The function **str2num** does the reverse; it takes a string in which a number is stored and converts it to the type **double**:

```
>> num = str2num('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,

```
>> mystr = '66 2 111';
>> numvec = str2num(mystr)
numvec =
    66     2     111
>> sum(numvec)
ans =
    179
```

Exercise

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

```
isletter ('?')
isspace('Oh no!')
str = '12 33';
ischar(str)
v = str2num(str)
ischar(v)
sum(v)
num = 234;
size(num)
snum = int2str(num);
size(snum)
```

Exercise

Let's say that we have a string that consists of an angle followed by either 'd' for degrees or 'r' for radians. For example, it may be a string entered by the user:

```
degrad = input('Enter angle and d/r: ', 's');
Enter angle and d/r: 54r
```

How could we separate the string into the angle and the character, and then get the sine of that angle using either **sin** or **sind**, as appropriate (**sin** for radians or **sind** for degrees)?

```
>> angle_d_or_r
Enter angle and d/r: 3.1r
The sine of 3.1 radians is 0.042.
>> angle_d_or_r
Enter angle and d/r: 53t
Error! Enter d or r with the angle.
Enter angle and d/r: 53d
The sine of 53.0 degrees is 0.799.
```

angle_d_or_r.m

```
% Prompt the user for angle and 'd' for degrees
% or 'r' for radians; print the sine of the angle

% Read in the response as a string and then
% separate the angle and character
degrad = input('Enter angle and d/r: ', 's');
angle = degrad(1:length(degrad)-1);
dorr = degrad(end);
% Error-check to make sure user enters 'd' or 'r'
while dorr ~= 'd' & dorr ~= 'r'
    disp('Error! Enter d or r with the angle.')
    degrad = input('Enter angle and d/r: ', 's');
    angle = degrad(1:length(degrad)-1);
    dorr = degrad(end);
end
% Convert angle to number
anglenum = str2num(angle);
fprintf('The sine of %.1f ', anglenum)
% Choose sin or sind function
if dorr == 'd'
    fprintf('degrees is %.3f.\n', sind(anglenum))
else
    fprintf('radians is %.3f.\n', sin(anglenum))
end
```

(Continued)

Summary

- Putting arguments to **strfind** in incorrect order (the order matters for **strfind** but not for **findstr**).
- 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 (the easiest way is to use **strvcat** or **char**, which will pad with extra blanks automatically).
- Forgetting that when using **strtok**, the second argument returned (the rest of the string) 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

- Trim trailing blanks from strings stored in matrices before using.
- Make sure the correct string comparison function is used; for example, **strcmpi** if ignoring case is desired.

MATLAB Functions and Commands			
strcat	upper	findstr	isletter
strvcat	lower	strfind	isspace
blanks	strcmp	strtok	ischar
sprintf	strncmp	strrep	int2str
deblank	strcmpi	date	num2str
strtrim	strncmpi	eval	str2num

Data structures: Cell Arrays and Structures

Data structures are variables that store more than one value. In order for it to make sense to store more than one value in a variable, the values should somehow be logically related. There are many different kinds of data structures. We have already been working with one kind, arrays (e.g., vectors and matrices). An array is a data structure in which all the values are logically related in that they are of the same type, and represent in some sense the same thing. So far, that has been true for the vectors and matrices that we have used.

A ***cell array*** is a kind of data structure that stores values of different types. Cell arrays can be vectors or matrices; the different values are stored in the elements of the array. One very common use of a cell array is to store strings of different lengths.

Structures are data structures that group together values that are logically related, but are not the same thing and not necessarily the same type. The different values are stored in separate ***fields*** of the structure.

Cell arrays（单元数组）

One type of data structure that MATLAB has but is not found in many programming languages is a ***cell array***. A cell array in MATLAB is an array, but unlike the vectors and matrices we have used so far, elements in cell arrays can store different types of values.

Creating Cell Arrays

There are several ways to create cell arrays. For example, we will create a cell array in which one element will store an integer, one element will store a character, one element will store a vector, and one element will store a string. Just as with the arrays we have seen so far, this could be a 1×4 row vector, a 4×1 column vector, or a 2×2 matrix. The syntax for creating vectors and matrices is the same as before. Values within rows are separated by spaces or commas, and rows are separated by semicolons. However, for cell arrays, curly braces are used rather than square brackets. For example, the following creates a row vector cell array with the four different types of values:

```
>> cellrowvec = {23, 'a', 1:2:9, 'hello'}
cellrowvec =
    [23]    'a'    [1x5 double]    'hello'
```

To create a column vector cell array, the values are instead separated by semicolons:

```
>> cellcolvec = {23; 'a'; 1:2:9; 'hello'}
cellcolvec =
    [     23]
    'a'
    [1x5 double]
    'hello'
```

This method creates a 2×2 cell array matrix:

```
>> cellmat = {23 'a'; 1:2:9 'hello'}
cellmat =
    [     23]      'a'
    [1x5 double]   'hello'
```

Cell arrays

Creating Cell Arrays

Another method of creating a cell array is simply to assign values to specific array elements and build it up element by element. However, as explained before, extending an array element by element is a very inefficient and time-consuming method. It is much more efficient, if the size is known ahead of time, to preallocate the array. For cell arrays, this is done with the **cell** function. For example, to preallocate a variable *mycellmat* to be a 2×2 cell array, the **cell** function would be called as follows:

```
>> mycellmat = cell(2,2)
mycellmat =
    []    []
    []    []
```

Note that this is a function call so the arguments to the function are in parentheses. This creates a matrix in which all the elements are empty vectors. Then, each element can be replaced by the desired value. How to refer to each element in order to accomplish this will be explained next.

Referring to and displaying cell array elements and attributes

Just as with the other vectors we have seen so far, we can refer to individual elements of cell arrays. The only difference is that curly braces are used for the subscripts. For example, this refers to the second element of the cell array *cellrowvec*:

```
> cellrowvec{2}
ans =
a
```

Row and column indices are used to refer to an element in a matrix (again using curly braces), for example,

```
>> cellmat{1,1}
ans =
    23
```

Values can be assigned to cell array elements. For example, after preallocating the variable *mycellmat* in the previous section, the elements can be initialized:

```
>> mycellmat{1,1} = 23
mycellmat =
    [23]   []
      []   []
```

When an element of a cell array is itself a data structure, only the type of the element is displayed when the cell array contents are shown. For example, in the cell arrays just created, the vector is shown just as 1×5 **double**. Referring to that element specifically would display its contents, for example,

```
>> cellmat{2,1}
ans =
     1     3     5     7     9
```

Referring to and displaying cell array elements and attributes (1)

Since this element is a vector, parentheses are used to refer to its elements. For example, the fourth element of the preceding vector is:

```
>> cellmat{2,1}(4)
ans =
     7
```

Notice that the index into the cell array is given in curly braces, and then parentheses are used to refer to an element of the vector.

We can also refer to subsets of cell arrays, for example,

```
>> cellcolvec{2:3}
ans =
a
ans =
     1     3     5     7     9
```

Notice, however, that MATLAB stored *cellcolvec*{2} in the default variable *ans*, and then replaced that with the value of *cellcolvec*{3}. This is because the two values are different types, and therefore cannot be stored together in *ans*. However, they could be stored in two separate variables by having a vector of variables on the left-hand side of an assignment.

```
>> [c1 c2] = cellcolvec{2:3}
c1 =
a
c2 =
     1     3     5     7     9
```

There are several methods of displaying cell arrays. The celldisp function displays all elements of the cell array:

```
>> celldisp(cellrowvec)
cellrowvec{1} =
     23
cellrowvec{2} =
a
cellrowvec{3} =
     1     3     5     7     9
cellrowvec{4} =
hello
```

The function cellplot puts a graphical display of the cell array in a Figure Window; however, it is a high-level view and basically just displays the same information as typing the name of the variable (e.g., it wouldn't show the contents of the vector in the previous example).

Many of the functions and operations on arrays that we have already seen also work with cell arrays. For example, here are some related to dimensioning:

Referring to and displaying cell array elements and attributes (2)

```
>> length(cellrowvec)
ans =
      4
>> size(cellcolvec)
ans =
      4    1
>> cellrowvec{end}
ans =
hello
```

It is not possible to delete an individual element from a cell array. For example, assigning an empty vector to a cell array element does not delete the element, it just replaces it with the empty vector:

```
>> cellrowvec
mycell =
      [23]    'a'     [1x5 double]    'hello'
>> length(cellrowvec)
ans =
      4
>> cellrowvec{2} = []
mycell =
      [23]     []     [1x5 double]    'hello'
>> length(cellrowvec)
ans =
      4
```

However, it is possible to delete an entire row or column from a cell array by assigning the empty vector (Note: use parentheses rather than curly braces to refer to the row or column):

```
>> cellmat
mycellmat =
      [     23]         'a'
      [1x5 double]      'hello'
>> cellmat(1,:) = []
mycellmat =
      [1x5 double]      'hello'
```

Storing strings in cell arrays

One good application of a cell array is to store strings of different lengths. Since cell arrays can store different types of values in the elements, that means strings of different lengths can be stored in the elements.

```
>> names = {'Sue', 'Cathy', 'Xavier'}
names =
    'Sue'    'Cathy'    'Xavier'
```

This is extremely useful, because unlike vectors of strings created using char or strvcat, these strings do not have extra trailing blanks.

The length of each string can be displayed using a for loop to loop through the elements of the cell array:

```
>> for i = 1:length(names)
    disp(length(names{i}))
   end
    3
    5
    6
```

It is possible to convert from a cell array of strings to a character array, and vice versa. MATLAB has several functions that facilitate this. For example, the function cellstr converts from a character array padded with blanks to a cell array in which the trailing blanks have been removed.

```
>> greetmat = char('Hello','Goodbye');
>> cellgreets = cellstr(greetmat)
cellgreets =
    'Hello'
    'Goodbye'
```

The char function can convert from a cell array to a character matrix:

```
>> names = {'Sue', 'Cathy', 'Xavier'};
>> cnames = char(names)
cnames =
Sue
Cathy
Xavier
>> size(cnames)
ans =
     3     6
```

The function iscellstr will return logical true if a cell array is a cell array of all strings, or logical false if not.

```
>> iscellstr(names)
ans =
     1
>> iscellstr(cellcolvec)
ans =
     0
```

We will see several examples of cell arrays containing strings of varying lengths in the coming chapters, including advanced file input functions and customizing plots.

Exercise

Write an expression that would display a random element from a cell array (without assuming that the number of elements in the cell array is known). Create two different cell arrays and try the expression on them to make sure that it is correct.

Structures

Structures are data structures that group together values that are logically related in what are called *fields* of the structure. An advantage of structures is that the fields are named, which helps to make it clear what the values are that are stored in the structure. However, structure variables are not arrays. They do not have elements, so it is not possible to loop through the values in a structure.

Creating and Modifying Structure Variables

Creating structure variables can be accomplished by simply storing values in fields using assignment statements, or by using the struct function.

The first example that will be used is that the local Computer Super Mart wants to store information on the software packages that it sells. For every one, they will store:

- The item number
- The cost to the store
- The price to the customer
- A code indicating the type of software

An individual structure variable for one software package might look like this:

package			
item_no	cost	price	code
123	19.99	39.95	'g'

The name of the structure variable is *package*; it has four fields called item_no, cost, price, and code.

One way to initialize a structure variable is to use the struct function, which preallocates the structure. The field names are passed to the struct in quotes, following each one with the value for that field:

```
>> package = struct('item_no',123,'cost',19.99,...
     'price',39.95,'code','g')
package =
    item_no: 123
       cost: 19.9900
      price: 39.9500
       code: 'g'
```

Structures

Creating and Modifying Structure Variables

Typing the name of the structure variable will display the names and contents of all fields:

```
>> package
package =
    item_no: 123
       cost: 19.9900
      price: 39.9500
       code: 'g'
```

Notice that in the Workspace Window, the variable *package* is listed as a 1 × 1 struct. MATLAB, since it is written to work with arrays, assumes the array format. Just as a single number is treated as a 1 × 1 double, a single structure is treated as a 1 × 1 struct. Later in this chapter we will see how to work more generally with vectors of structs.

An alternative method of creating this structure, which is not as efficient, involves using the *dot operator* to refer to fields within the structure. The name of the structure variable is followed by a dot, or period, and then the name of the field within that structure. Assignment statements can be used to assign values to the fields.

By using the dot operator in the first assignment statement, a structure variable is created with the field *item_no*. The next three assignment statements add more fields to the structure variable.

Adding a field to a structure later is accomplished as shown earlier, by using an assignment statement.

An entire structure variable can be assigned to another. This would make sense, for example, if the two structures had some values in common. Here, for example, the values from one structure are copied into another and then two fields are selectively changed.

```
>> newpack = package;
>> newpack.item_no = 111;
>> newpack.price = 34.95
newpack =
    item_no: 111
       cost: 19.9900
      price: 34.9500
       code: 'g'
```

Structures

Creating and Modifying Structure Variables

To print from a structure, the disp function will display either the entire structure or a field.

```
>> disp(package)
    item_no: 123
       cost: 19.9900
      price: 39.9500
       code: 'g'
>> disp(package.cost)
    19.9900
```

However, using fprintf, only individual fields can be printed; the entire structure cannot be printed.

```
>> fprintf('%d %c\n', package.item_no, package.code)
123 g
```

The function rmfield removes a field from a structure. It returns a new structure with the field removed, but does not modify the original structure (unless the returned structure is assigned to that variable). For example, the following would remove the *code* field from the *newpack* structure, but store the resulting structure in the default variable *ans*. The value of *newpack* remains unchanged.

```
>> rmfield(newpack, 'code')
ans =
    item_no: 111
       cost: 19.9900
      price: 34.9500
>> newpack
newpack =
    item_no: 111
       cost: 19.9000
      price: 34.9500
       code: 'g'
```

To change the value of *newpack*, the structure that results from calling rmfield must be assigned to *newpack*.

```
>> newpack = rmfield(newpack, 'code')
newpack =
    item_no: 111
       cost: 19.9000
      price: 34.9500
```

Structures

Creating and Modifying Structure Variables

To print from a structure, the disp function will display either the entire structure or a field.

```
>> disp(package)
    item_no: 123
       cost: 19.9900
      price: 39.9500
       code: 'g'
>> disp(package.cost)
    19.9900
```

However, using fprintf, only individual fields can be printed; the entire structure cannot be printed.

```
>> fprintf('%d %c\n', package.item_no, package.code)
123 g
```

The function rmfield removes a field from a structure. It returns a new structure with the field removed, but does not modify the original structure (unless the returned structure is assigned to that variable). For example, the following would remove the *code* field from the *newpack* structure, but store the resulting structure in the default variable *ans*. The value of *newpack* remains unchanged.

```
>> rmfield(newpack, 'code')
ans =
    item_no: 111
       cost: 19.9900
      price: 34.9500
>> newpack
newpack =
    item_no: 111
       cost: 19.9000
      price: 34.9500
       code: 'g'
```

To change the value of *newpack*, the structure that results from calling rmfield must be assigned to *newpack*.

```
>> newpack = rmfield(newpack, 'code')
newpack =
    item_no: 111
       cost: 19.9000
      price: 34.9500
```

Structures

Creating and Modifying Structure Variables

two different versions of a function that calculates the profit on a software package. The profit is defined as the price minus the cost.

In the first version, the entire structure variable is passed to the function, so the function must use the dot operator to refer to the *price* and *cost* fields of the input argument.

calcprof.m

```
function profit = calcprof(packstruct)
% Calculates the profit for a software package
% The entire structure is passed to the function
profit = packstruct.price - packstruct.cost;
```

```
>> calcprof(package)
ans =
    19.9600
```

In the second version, just the *price* and *cost* fields are passed to the function using the dot operator in the function call. These are passed to two scalar input arguments in the function header, so there is no reference to a structure variable in the function itself, and the dot operator is not needed in the function.

calcprof2.m

```
function profit = calcprof2(oneprice, onecost)
% Calculates the profit for a software package
% The individual fields are passed to the function
profit = oneprice - onecost;
```

```
>> calcprof2(package.price, package.cost)
ans =
    19.9600
```

It is important, as always with functions, to make sure that the arguments in the function call correspond one-to-one with the input arguments in the function header. In the case of *calcprof*, a structure variable is passed to an input argument, which is a structure. For the second function *calcprof2*, two individual fields, which are double values, are passed to two double arguments.

Exercise

A silicon wafer manufacturer stores, for every part in their inventory, a part number, how many are in the factory, and the cost for each.

onepart		
part_no	quantity	costper
123	4	33

Create this structure variable using **struct**. Print the cost in the form $xx.xx.

Structures

Related Structure Functions

There are several functions that can be used with structures in MATLAB. The function isstruct will return 1 for logical true if the variable argument is a structure variable, or 0 if not. The isfield function returns logical true if a fieldname (as a string) is a field in the structure argument, or logical false if not.

```
>> isstruct(package)
ans =
     1
>> isfield(package,'cost')
ans =
     1
```

The fieldnames function will return the names of the fields that are contained in a structure variable.

```
>> pack_fields = fieldnames(package)
pack_fields =
    'item_no'
    'cost'
    'price'
    'code'
```

Since the names of the fields are of varying lengths, the fieldnames function returns a cell array with the names of the fields.

Curly braces are used to refer to the elements, since pack_fields is a cell array. For example, we can refer to the length of one of the strings:

```
>> length(pack_fields{2})
ans =
     4
```

Exercise

How can we ask the user for a field in a structure and either print its value or an error if it is not actually a field?

Answer: The **isfield** function can be used to determine whether or not it is a field of the structure. Then, by concatenating that field name to the structure variable and dot, and then passing the entire string to **eval**, the expression would be evaluated to the actual field in the structure. The following code

```
inputfield = input('Which field would you like to see: ','s');
if isfield(package, inputfield)
  fprintf('The value of the %s field is: %s\n', ...
    inputfield, eval(['package.' inputfield]))
else
  fprintf('Error: %s is not a valid field\n', inputfield)
end
```

would produce this output (assuming the *package* variable was initialized as shown earlier):

```
Which field would you like to see: code
The value of the code field is: g
```

Vectors of structures (1)

In many applications, including database applications, information normally would be stored in a *vector of structures*, rather than in individual structure variables. For example, if the Computer Super Mart is storing information on all the software packages that it sells, it would likely be in a vector of structures, for example,

	packages			
	item_no	cost	price	code
1	123	19.99	39.95	'g'
2	456	5.99	49.99	'l'
3	587	11.11	33.33	'w'

In this example, *packages* is a vector that has three elements. It is shown as a column vector. Each element is a structure consisting of four fields, item_no, cost, price, and code. It may look like a matrix with rows and columns, but it is instead a vector of structures.

This can be created several ways. One method is to create a structure variable, as shown earlier, to store information on one software package. This can then be expanded to be a vector of structures.

```
>> packages = struct('item_no',123,'cost',19.99,...
     'price',39.95,'code','g');
>> packages(2) = struct('item_no',456,'cost', 5.99,...
     'price',49.99,'code','l');
>> packages(3) = struct('item_no',587,'cost',11.11,...
     'price',33.33,'code','w');
```

The first assignment statement shown here creates the first structure in the structure vector, the next one creates the second structure, and so on. This actually creates a 1×3 row vector.

Alternatively, the first structure could be treated as a vector to begin with, for example,

```
>> packages(1) = struct('item_no',123,'cost',19.99,...
     'price',39.95,'code','g');
>> packages(2) = struct('item_no',456,'cost', 5.99,...
     'price',49.99,'code','l');
>> packages(3) = struct('item_no',587,'cost',11.11,...
     'price',33.33,'code','w');
```

Vectors of structures (2)

```
>> packages(3) = struct('item_no',587,'cost',11.11,...
     'price',33.33,'code','w');
>> packages(1) = struct('item_no',123,'cost',19.99,...
     'price',39.95,'code','g');
>> packages(2) = struct('item_no',456,'cost', 5.99,...
     'price',49.99,'code','l');
```

Another method is to create one element with the values from one structure, and use repmat to replicate it to the desired size. Then, the remaining elements can be modified. The following creates one structure and then replicates this into a 1×3 matrix.

```
>> packages = repmat(struct('item_no',587,'cost',...
       11.11, 'price',33.33,'code','w'), 1,3);
>> packages(2) = struct('item_no',456,'cost', 5.99,...
       'price',49.99,'code','l');
>> packages(3) = struct('item_no',587,'cost',11.11,...
       'price',33.33,'code','w');
```

Typing the name of the variable will display only the size of the structure vector and the names of the fields:

```
>> packages
packages =
1x3 struct array with fields:
    item_no
    cost
    price
    code
```

The variable *packages* is now a vector of structures, so each element in the vector is a structure. To display one element in the vector (one structure), an index into the vector would be specified. For example, to refer to the second element:

```
>> packages(2)
ans =
    item_no: 456
       cost: 5.9900
      price: 49.9900
       code: 'l'
```

Vectors of structures (3)

So, there are essentially three levels to this data structure. The variable *packages* is the highest level, which is a vector of structures. Each of its elements is an individual structure. The fields within these individual structures are the lowest level. The following loop displays each element in the *packages* vector.

```
>> for i = 1:length(packages)
     disp(packages(i))
   end
    item_no: 123
       cost: 19.9900
      price: 39.9500
       code: 'g'

    item_no: 456
       cost: 5.9900
      price: 49.9900
       code: 'l'

    item_no: 587
       cost: 11.1100
      price: 33.3300
       code: 'w'
```

To refer to a particular field for all structures, in most programming languages it would be necessary to loop through all elements in the vector and use the dot operator to refer to the field for each element. However, this is not the case in MATLAB.

The Programming Concept

For example, to print all of the costs, a **for** loop could be used:

```
>> for i=1:3
     fprintf('%f\n',packages(i).cost)
end
19.990000
5.990000
11.110000
```

The Efficient Method

However, **fprintf** would do this automatically in MATLAB:

```
>> fprintf('%f\n',packages.cost)
19.990000
5.990000
11.110000
```

Vectors of structures (4)

Using the dot operator in this manner to refer to all values of a field would result in the values being stored successively in the default variable *ans*:

```
>> packages.cost
ans =
   19.9900
ans =
   5.9900
ans =
   11.1100
```

However, the values can all be stored in a vector:

```
>> pc = [packages.cost]
pc =
   19.9900    5.9900    11.1100
```

Using this method, MATLAB allows the use of functions on all the same fields within a vector of structures. For example, to sum all three cost fields, the vector of cost fields is passed to the sum function:

```
>> sum([packages.cost])
ans =
   37.0900
```

For vectors of structures, the entire vector (e.g., *packages*) could be passed to a function, or just one element (e.g., *packages(1)*), which would be a structure, or a field within one of the structures (e.g., *packages(2).price*).

Here is an example of a function that receives the entire vector of structures as an input argument, and prints all of it in a nice table format.

printpackages.m

```
function printpackages(packstruct)
% This function prints a table showing all
% values from a vector of packages structures
fprintf('\nItem # Cost Price Code\n\n')
no_packs = length(packstruct);
for i = 1:no_packs
    fprintf('%6d %6.2f %6.2f %3c\n', ...
        packstruct(i).item_no, ...
        packstruct(i).cost, ...
        packstruct(i).price, ...
        packstruct(i).code)
end
```

The function loops through all the elements of the vector, each of which is a structure, and uses the dot operator to refer to and print each field. Here is an example of calling the function:

```
>> printpackages(packages)
Item #       Cost       Price      Code
123          19.99      39.95      g
456          5.99       49.99      l
587          11.11      33.33      w
```

Exercise

A silicon wafer manufacturer stores, for every part in their inventory, a part number, how many are in the factory, and the cost for each. First, create a vector of structs called *parts* so that when displayed it has the following values:

```
>> parts
parts =
1x3 struct array with fields:
    partno
    quantity
    costper
>> parts(1)
ans =
     partno: 123
    quantity: 4
     costper: 33
>> parts(2)
ans =
     partno: 142
    quantity: 1
     costper: 150
>> parts(3)
ans =
     partno: 106
    quantity: 20
     costper: 7.5000
```

Next, write general code that will, for any values and any number of structures in the variable *parts*, print the part number and the total cost (quantity of parts multiplied by the cost of each) in a column format. For example, if the variable *parts* stores the values shown, the result would be:

```
    123   132.00
    142   150.00
    106   150.00
```

Vectors of structures (5)

The previous example involved a vector of structs. In the next example, a somewhat more complicated data structure will be introduced: a vector of structs in which some fields are vectors themselves. The example is a database of information that a professor might store for his or her class. This will be implemented as a vector of structures. The vector will store all the class information. Every element in the vector will be a structure, representing all information about one particular student. For every student, the professor wants to store (for now, this would be expanded later):

- Name (a string)
- University ID number
- Quiz scores (a vector of 4 quiz scores)

The vector variable, called *student*, might look like this:

	student					
	name	id_no	quiz			
			1	2	3	4
1	C, Joe	999	10.0	9.5	0.0	10.0
2	Hernandez, Pete	784	10.0	10.0	9.0	10.0
3	Brownnose, Violet	332	7.5	6.0	8.5	7.5

Each element in the vector is a struct with three fields (*name, id_no, quiz*). The *quiz* field is a vector of quiz grades. The *name* field is a string.

This data structure could be defined as follows.

```
>> student(3) = struct('name','Brownnose, Violet',...
    'id_no',332,'quiz', [7.5 6 8.5 7.5]);
>> student(1) = struct('name','C, Joe',...
  'id_no',999,'quiz', [10 9.5 0 10]);
>> student(2) = struct('name','Hernandez, Pete',...
  'id_no',784,'quiz', [10 10 9 10]);
```

Once this data structure has been initialized, in MATLAB we could refer to different parts of it. The variable *student* is the entire array; MATLAB just shows the names of the fields.

```
>> student
student =
1x3 struct array with fields:
    name
    id_no
    quiz
```

Vectors of structures (6)

To see the actual values, we would have to refer to individual structures and fields.

```
>> student(1)
ans =
    name: 'C, Joe'
    id_no: 999
    quiz: [10 9.5000 0 10]
>> student(1).quiz
ans =
    10.0000    9.5000    0    10.0000
>> student(1).quiz(2)
ans =
    9.5000
>> student(3).name(1)
ans =
B
```

With a more complicated data structure like this, it is important to be able to understand different parts of the variable. The following are examples of expressions that refer to different parts of this data structure:

- *student* is the entire data structure, which is a vector of structs
- *student(1)* is an element from the vector, which is an individual struct
- *student(1).id_no* is the *id_no* field from the structure, which is a double value
- *student(1).quiz* is the *quiz* field from the structure, which is a vector of doubles
- *student(1).quiz(1)* is an individual double quiz grade

One example of using this data structure would be to calculate and print the quiz average for each student. The following function accomplishes this. The *student* structure, as just defined, is passed to this function. The algorithm in the function is:

- Print column headings.
- Loop through the individual students. For each,
  - Sum the quiz grades
  - Calculate the average
  - Print the student's name and quiz average

With the programming method, a second (nested) loop would be required to find the running sum of the quiz grades. However, as we have seen, the sum

Example

function can be used to sum the vector of all quiz grades for each student. The function is defined as:

print_aves.m

```
function print_aves(student)
% This function prints the average quiz grade
% for each student in the vector of structs
fprintf('%-20s %-10s\n', 'Name', 'Average')
for i = 1:length(student)
    qsum = sum([student(i).quiz]);
    no_quizzes = length(student(i).quiz);
    ave = qsum / no_quizzes;
    fprintf('%-20s %.1f\n', student(i).name, ave);
end
```

Here is an example of calling the function:

```
>> print_aves(student)
Name                 Average
C, Joe               7.4
Hernandez, Pete      9.8
Brownnose, Violet    7.4
```

Nested Structures (1)

A *nested structure* is a structure in which at least one member is itself a structure.

For example, a structure for a line segment might consist of fields representing the two points at the ends of the line segment. Each of these points would be represented as a structure consisting of the x- and y-coordinates.

lineseg			
endpoint1		endpoint2	
x	y	x	y
2	4	1	6

This shows a structure variable called *lineseg* that has two fields, *endpoint1* and *endpoint2*. Each of these is a structure consisting of two fields, *x* and *y*.

One method of defining this is to nest calls to the struct function:

Nested Structures (2)

```
>> lineseg = struct('endpoint1',struct('x',2,'y',4), ...
                   'endpoint2',struct('x',1,'y',6))
```

This method is the most efficient. However, another method is to build the nested structure one field at a time. Since this is a nested structure with one structure inside of another, the dot operator must be used twice here to get to the actual x- and y-coordinates.

```
>> lineseg.endpoint1.x = 2;
>> lineseg.endpoint1.y = 4;
>> lineseg.endpoint2.x = 1;
>> lineseg.endpoint2.y = 6;
```

Once the nested structure has been created, we can refer to different parts of the variable *lineseg*. Just typing the name of the variable shows only that it is a structure consisting of two fields, *endpoint1* and *endpoint2*, each of which is a structure.

```
>> lineseg
lineseg =
  endpoint1: [1x1 struct]
  endpoint2: [1x1 struct]
```

Typing the name of one of the nested structures will display the field names and values within that structure:

```
>> lineseg.endpoint1
ans =
  x: 2
  y: 4
```

Using the dot operator twice will refer to an individual coordinate, for example,

```
>> lineseg.endpoint1.x
ans =
     2
```

Exercise

How could we write a function *strpoint* that returns a string (x,y) containing the x- and y-coordinates? For example, it might be called separately to create strings for the two endpoints and then printed as shown here:

```
>> fprintf('The line segment consists of %s and %s\n', ...
   strpoint(lineseg.endpoint1), ...
   strpoint(lineseg.endpoint2))
The line segment consists of (2, 4) and (1, 6)
```

Answer: Since an endpoint structure is passed to the function to an input argument, the dot operator is used within the function to refer to the x- and y-coordinates. The **sprintf** function is used to create the string that is returned.

strpoint.m

```
function ptstr = strpoint(ptstruct)
% This function receives the struct containing x and y
% coordinates and returns a string '(x,y)'
ptstr = sprintf('(%d, %d)', ptstruct.x, ptstruct.y);
```

A nested structure variable for a line segment could also be created by creating structure variables for the points first, and then storing these in the two fields of a line segment variable. For example:

```
>> pointone = struct('x', 5, 'y', 11);
>> pointtwo = struct('x', 7, 'y', 9);
>> myline = struct('endpoint1', pointone,...
                   'endpoint2', pointtwo)

myline =
  endpoint1: [1x1 struct]
  endpoint2: [1x1 struct]
```

Then, referring to different parts of the variable would work the same:

```
>> myline.endpoint1
ans =
  x: 5
  y: 11

>> myline.endpoint2.x
ans =
  7
```

Nested Structures (3)

A nested structure variable for a line segment could also be created by creating structure variables for the points first, and then storing these in the two fields of a line segment variable. For example:

```
>> pointone = struct('x', 5, 'y', 11);
>> pointtwo = struct('x', 7, 'y', 9);
>> myline = struct('endpoint1', pointone,...
                   'endpoint2', pointtwo)

myline =
  endpoint1: [1x1 struct]
  endpoint2: [1x1 struct]
```

Then, referring to different parts of the variable would work the same:

```
>> myline.endpoint1
ans =
  x: 5
  y: 11

>> myline.endpoint2.x
ans =
  7
```

Vectors of nested structures (1)

Combining vectors and nested structures, it is possible to have a vector of structures in which some fields are structures themselves. Here is an example in which a company manufactures cylinders from different materials for industrial use. Information on them is stored in a data structure in a program. The variable *cyls* is a vector of structures, each of which has fields *code*, *dimensions*, and *weight*. The *dimensions* field is a structure itself consisting of fields *rad* and *height* for the radius and height of each cylinder.

	cyls			
	code	dimensions		weight
		rad	height	
1	'x'	3	6	7
2	'a'	4	2	5
3	'c'	3	6	9

Vectors of nested structures (2)

Here is an example of initializing the data structure by preallocating:

```
>> cyls(3) = struct('code', 'c', 'dimensions',...
   struct('rad', 3, 'height', 6), 'weight', 9);
>> cyls(1) = struct('code', 'x', 'dimensions',...
   struct('rad', 3, 'height', 6), 'weight', 7);
>> cyls(2) = struct('code', 'a', 'dimensions',...
   struct('rad', 4, 'height', 2), 'weight', 5);
```

Alternatively, it could be initialized by using the dot operator:

```
>> cyls(3).code = 'c';
>> cyls(3).dimensions.rad = 3;
>> cyls(3).dimensions.height = 6;
>> cyls(3).weight = 9;
>> cyls(1).code = 'x';
>> cyls(1).dimensions.rad = 3;
>> cyls(1).dimensions.height = 6;
>> cyls(1).weight = 7;
>> cyls(2).code = 'a';
>> cyls(2).dimensions.rad = 4;
>> cyls(2).dimensions.height = 2;
>> cyls(2).weight = 5;
```

Here is an example of calling this function:

```
>> printcylvols(cyls)
Cylinder x has a volume of 169.6
Cylinder a has a volume of 100.5
Cylinder c has a volume of 169.6
```

There are several layers in this variable. For example,

- *cyls* is the entire data structure, which is a vector of structs
- *cyls(1)* is an individual element from the vector, which is a struct
- *cyls(2).code* is the *code* field from the struct *cyls(2);* it is a character
- *cyls(3).dimensions* is the *dimensions* field from the struct *cyls(3);* it is a struct itself
- *cyls(1).dimensions.rad* is the *rad* field from the struct *cyls(1).dimensions;* it is a double number

For these cylinders, one desired calculation may be the volume of each cylinder, which is defined as $\pi * r^2 * h$, where r is the radius and h is the height. The function *printcylvols* prints the volume of each cylinder, along with its *code* for identification purposes. It calls a subfunction to calculate each volume.

printcylvols.m

```
function printcylvols(cyls)
% This function prints the volumes of each cylinder
% It calls a subfunction to calculate each volume
for i = 1:length(cyls)
    vol = cylvol(cyls(i).dimensions);
    fprintf('Cylinder %c has a volume of %.1f\n', …
    cyls(i).code, vol);
  end

function cvol = cylvol(dims)
% Calculates the volume of a cylinder
cvol = pi * dims.rad ^ 2 * dims.height;
```

Exercise

Modify the function *cylvol* to calculate the surface area of the cylinder in addition to the volume.

Summary (1)

Common Pitfalls

- Trying to use parentheses rather than curly braces for a cell array
- Forgetting to index into a vector using parentheses or referring to a field of a structure using the dot operator

Programming Style Guidelines

- Use arrays when values are the same type and represent in some sense the same thing.
- Use cell arrays or structures when the values are logically related but not the same type or the same thing.
- Use cell arrays rather than character matrices when storing strings of different lengths.

Summary (2)

- Use cell arrays rather than structures when you want to loop through the values.
- Use structures rather than cell arrays when you want to use names for the different values rather than indices.

MATLAB Functions and Commands			
cell	cellstr	rmfield	fieldnames
celldisp	iscellstr	isstruct	
cellplot	struct	isfield	

MATLAB Operators	
cell arrays {}	dot operator for structs.

Practice problem

Write a script that will prompt the user to enter a word, and then print the first character in the word. For example, the output might look like this:

```
>> Enter a word: howdy
The word howdy starts with the letter 'h'
```

Practice problem

Write a function that will receive a name and department as separate strings 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
```

Practice problem

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 ('o', '*'). It will then create a string *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*'.

Practice problem

Words in a sentence variable (just a string variable) called *mysent* are separated by /'s instead of blank spaces. For example, mysent might have this value:

```
'This/is/not/quite/right'
```

Write a function *slashtoblank* that will receive a string in this form and will return a string in which the words are separated by blank spaces. This should be general and work regardless of the value of the argument. No loops are allowed in this function; the built-in string function(s) must be used.

```
>> mysent = 'This/is/not/quite/right';
>> newsent = slashtoblank(mysent)
newsent =
This is not quite right
```

Practice problem

Create the following two variables:

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

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

Practice problem

Using the functions **char** and **double**, you can shift words. For example, you 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 string by altering the character codes. Figure out the original string. Try adding and subtracting different values (do this in a loop) until you decipher it:

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

Practice problem

Two variables store strings 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 string manipulating functions to extract the numbers from the strings and add them together.

Practice problem

Write a script that will first initialize a string variable that will store x and y coordinates of a point in the form ‘x 3.1 y 6.4’. Then, use string manipulating functions to extract the coordinates and plot them.

Practice problem

Modify the script in the previous example to be more general: the string could store the coordinates in any order; for example, it could store ‘y 6.4 x 3.1’.

Practice problem

Create a cell array that stores phrases, for example,

```
exclaimcell = {'Bravo', 'Fantastic job'};
```

Pick a random phrase to print.

Practice problem

Create three cell array variables that store people's names, verbs, and nouns. For example,

```
names = {'Harry', 'Xavier', 'Sue'};
verbs = {'loves', 'eats'};
nouns = {'baseballs', 'rocks', 'sushi'};
```

Write a script that will initialize these cell arrays, and then print sentences using one random element from each cell array, for example, 'Xavier eats sushi'.

Practice problem

Create a 2 × 2 cell array by using the **cell** function to preallocate and then put values in the individual elements. Then, insert a row in the middle so that the cell array is now 3 × 2. Hint: Extend the cell array by adding another row and copying row 2 to row 3, and then modify row 2.

Practice problem

Write a function *rid_multiple_blanks* that will receive a string as an input argument. The string contains a sentence that has multiple blank spaces in between some of the words. The function will return the string 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?
```

MATLAB Functions and Commands			
strcat	upper	findstr	isletter
strvcat	lower	strfind	isspace
blanks	strcmp	strtok	ischar
sprintf	strncmp	strrep	int2str
deblank	strcmpi	date	num2str
strtrim	strncmpi	eval	str2num

Practice problem

The built-in **clock** function returns a vector with six elements representing the year, month, day, hours, minutes, and seconds; the first five elements are integers whereas the last is a **double** value, but calling it with **fix** will convert all to integers. The built-in **date** function returns the day, month, and year as a string. For example,

```
>> fix(clock)
ans =
   2008    4    25    14    25    49
>> date
ans =
25-Apr-2008
```

Write a script that will call both of these built-in functions, and then compare results to make sure that the year is the same. The script will have to convert one from a string to a number, or the other from a number to a string in order to compare.

Practice problem

Write the code in MATLAB that would create the following data structure, and put the following values into the variable:

	experiments					
	num	code	weights		height	
			1	2	feet	inches
1	33	'x'	200.34	202.45	5	6
2	11	't'	111.45	111.11	7	2

The variable is called *experiments*, which is a vector of structs. Each struct has four fields: *num*, *code*, *weights*, and *height*. The field *num* is an integer, *code* is a character, *weights* is a vector with two values (both of which are double values), and *height* is a struct with fields *feet* and *inches* (both of which are integers). Write the statements that would accomplish this, so that typing the following expressions in MATLAB would give the results shown:

```
>> experiments
experiments =
1x2 struct array with fields:
  num
  code
  weights
  height
```

```
>> experiments(2)

  ans =
        num: 11
       code: 't'
    weights: [111.4500 111.1100]
     height: [1x1 struct]
  >> experiments(1).height
  ans =
       feet: 5
     inches: 6
```

Practice problem

A script stores information on potential subjects for an experiment in a vector of structures called *subjects*. The following show an example of what the contents might be:

```
>> subjects
subjects =
1x3 struct array with fields:
  name
  sub_id
  height
  weight
>> subjects(1)
ans =
name: 'Joey'
  sub_id: 111
  height: 6.7000
  weight: 222.2000
```

For this particular experiment, the only subjects who are eligible are those whose height or weight is lower than the average height or weight of all subjects. The script will print the names of those who are eligible. Create a vector with sample data in a script, and then write the code to accomplish this. Don't assume that the length of the vector is known; the code should be general.