

# Selection Statements

KEY TERMS

|                      |                    |                   |
|----------------------|--------------------|-------------------|
| selection statements | temporary variable | cascading if-else |
| branching statements | error-checking     | “is” functions    |
| condition            | throwing an error  |                   |
| action               | nesting statements |                   |

CONTENTS

- 4.1. The if Statement .. 125
- 4.2. The if-else Statement .. 130
- 4.3. Nested if-else Statements 132
- 4.4. The Switch Statement . 138
- 4.5. The “is” Functions in MATLAB .... 142
- Summary ..... 145
- Common Pitfalls ..... 145
- Programming Style Guidelines ..... 147

In the scripts and functions we’ve seen thus far, every statement was executed in sequence. This is not always desirable, and in this chapter, we’ll see how to make choices as to whether statements are executed or not, and how to choose between or among statements. The statements that accomplish this are called *selection* or *branching* statements.

The MATLAB® software has two basic statements that allow us to make choices: the **if** statement and the **switch** statement. The **if** statement has optional **else** and **elseif** clauses for branching. The **if** statement uses expressions that are logically **true** or **false**. These expressions use relational and logical operators. MATLAB also has “is” functions that test whether an attribute is **true** or not; these can be used with the selection statements.

## 4.1 THE IF STATEMENT

The **if** statement chooses whether another statement, or group of statements, is executed or not. The general form of the **if** statement is:

```
if condition
    action
end
```

A **condition** is a relational expression that is conceptually, or logically, **true** or **false**. The **action** is a statement, or a group of statements, that will be executed if the condition is **true**. When the **if** statement is executed, first the condition is

evaluated. If the value of the condition is **true**, the action will be executed; if not, the action will not be executed. The action can be any number of statements until the reserved word **end**; the action is naturally bracketed by the reserved words **if** and **end**. (Note that this is different from the **end** that is used as an index into a vector or matrix.) The action is usually indented to make it easier to see.

For example, the following **if** statement checks to see whether the value of a variable is negative. If it is, the value is changed to a zero; otherwise, nothing is changed.

```
if num < 0
    num = 0
end
```

**If** statements can be entered in the Command Window, although they generally make more sense in scripts or functions. In the Command Window, the **if** line would be entered, followed by the Enter key, the action, the Enter key, and finally **end** and Enter. The results will follow immediately. For example, the preceding **if** statement is shown twice here.

```
>> num = -4;
>> if num < 0
    num = 0
end
num =
    0

>> num = 5;
>> if num < 0
    num = 0
end
>>
```

Note that the output from the assignment is not suppressed, so the result of the action will be shown if the action is executed. The first time the value of the variable is negative, so the action is executed and the variable is modified, but, in the second case, the variable is positive so the action is skipped.

This may be used, for example, to make sure that the square root function is not used on a negative number. The following script prompts the user for a number and prints the square root. If the user enters a negative number, the **if** statement changes it to zero before taking the square root.

```

sqrtifexamp.m

% Prompt the user for a number and print its sqrt

num = input('Please enter a number: ');

% If the user entered a negative number, change it
if num < 0
    num = 0;
end
fprintf('The sqrt of %.1f is %.1f\n', num, sqrt(num))

```

Here are two examples of running this script:

```

>> sqrtifexamp
Please enter a number: -4.2
The sqrt of 0.0 is 0.0

>> sqrtifexamp
Please enter a number: 1.44
The sqrt of 1.4 is 1.2

```

Note that in the script the output from the assignment statement is suppressed. In this case, the action of the **if** statement was a single assignment statement. The action can be any number of valid statements. For example, we may wish to print a note to the user to say that the number entered was being changed. Also, instead of changing it to zero, we will use the absolute value of the negative number entered by the user.

```

sqrtifexampii.m

% Prompt the user for a number and print its sqrt

num = input('Please enter a number: ');

% If the user entered a negative number, tell
% the user and change it
if num < 0
    disp('OK, we'll use the absolute value')
    num = abs(num);
end
fprintf('The sqrt of %.1f is %.1f\n', num, sqrt(num))

```

```

>> sqrtifexampii
Please enter a number: -25
OK, we'll use the absolute value
The sqrt of 25.0 is 5.0

```

Note that as seen in this example, two single quotes in the **disp** statement are used to print one single quote.

## PRACTICE 4.1

Write an `if` statement that would print "Hey, you get overtime!" if the value of a variable *hours* is greater than 40. Test the `if` statement for values of *hours* less than, equal to, and greater than 40. Will it be easier to do this in the Command Window or in a script?

## QUICK QUESTION!

Assume that we want to create a vector of increasing integer values from *mymin* to *mymax*. We will write a function *createvec* that receives two input arguments, *mymin* and *mymax*, and returns a vector with values from *mymin* to *mymax* in steps of one. First, we would make sure that the value of *mymin* is less than the value of *mymax*. If not, we would need to exchange their values before creating the vector. How would we accomplish this?

**Answer:** To exchange values, a third variable, a temporary variable, is required. For example, let's say that we have two variables, *a* and *b*, storing the values:

```
a = 3;
b = 5;
```

To exchange values, we could *not* just assign the value of *b* to *a*, as follows:

```
a = b;
```

If that were done, then the value of *a* [the 3] is lost! Instead, we need to assign the value of *a* first to a **temporary variable** so that the value is not lost. The algorithm would be:

- assign the value of *a* to *temp*
- assign the value of *b* to *a*
- assign the value of *temp* to *b*

```
>> temp = a;
>> a = b
a =
    5
>> b = temp
b =
    3
```

Now, for the function. An `if` statement is used to determine whether or not the exchange is necessary.

createvec.m

```
function outvec = createvec(mymin, mymax)
% createvec creates a vector that iterates from a
% specified minimum to a maximum
% Format of call: createvec(minimum, maximum)
% Returns a vector

% If the "minimum" isn't smaller than the "maximum",
% exchange the values using a temporary variable
if mymin > mymax
    temp = mymin;
    mymin = mymax;
    mymax = temp;
end

% Use the colon operator to create the vector
outvec = mymin:mymax;
end
```

## QUICK QUESTION!—CONT'D

Examples of calling the function are:

```
>> createvec(4,6)
ans =
    4    5    6

>> createvec(7,3)
ans =
    3    4    5    6    7
```

### 4.1.1 Representing Logical True and False

It has been stated that conceptually true expressions have the **logical** value of 1, and expressions that are conceptually false have the **logical** value of 0. Representing the concepts of **logical true** and **false** in MATLAB is slightly different: the concept of false is represented by the value of 0, but the concept of true can be represented by *any nonzero value* (not just 1). This can lead to some strange **logical** expressions. For example:

```
>> all(1:3)
ans =
    1
```

Also, consider the following **if** statement:

```
>> if 5
    disp('Yes, this is true!')
end
Yes, this is true!
```

As 5 is a nonzero value, the condition is **true**. Therefore, when this **logical** expression is evaluated, it will be **true**, so the **disp** function will be executed and “Yes, this is true” is displayed. Of course, this is a pretty bizarre **if** statement, one that hopefully would never be encountered!

However, a simple mistake in an expression can lead to a similar result. For example, let’s say that the user is prompted for a choice of ‘Y’ or ‘N’ for a yes/no question.

```
letter = input('Choice (Y/N): ', 's');
```

In a script we might want to execute a particular action if the user responded with ‘Y’. Most scripts would allow the user to enter either lowercase or uppercase; for example, either ‘y’ or ‘Y’ to indicate “yes”. The proper expression that would return **true** if the value of *letter* was ‘y’ or ‘Y’ would be

```
letter == 'y' || letter == 'Y'
```

However, if by mistake this was written as:

```
letter == 'y' || 'Y'           %Note: incorrect!!
```

this expression would ALWAYS be **true**, regardless of the value of the variable *letter*. This is because 'Y' is a nonzero value, so it is a **true** expression. The first part of the expression, on the left side of the or operator, may be **false**, but as the second expression (on the right side of the or operator; the 'Y') is **true** the entire expression would be **true**, regardless of the value of the variable *letter*.

## 4.2 THE IF-ELSE STATEMENT

The **if** statement chooses whether or not an action is executed. Choosing between two actions, or choosing from among several actions, is accomplished using **if-else**, nested **if-else**, and **switch** statements.

The **if-else** statement is used to choose between two statements, or sets of statements. The general form is:

```
if condition
    action1
else
    action2
end
```

First, the condition is evaluated. If it is **true**, then the set of statements designated as “action1” is executed, and that is the end of the **if-else** statement. If, instead, the condition is **false**, the second set of statements designated as “action2” is executed, and that is the end of the **if-else** statement. The first set of statements (“action1”) is called the action of the **if** clause; it is what will be executed if the expression is **true**. The second set of statements (“action2”) is called the action of the **else** clause; it is what will be executed if the expression is **false**. One of these actions, and only one, will be executed—which one depends on the value of the condition.

For example, to determine and print whether or not a random number in the range from 0 to 1 is less than 0.5, an **if-else** statement could be used:

```
if rand < 0.5
    disp('It was less than .5!')
else
    disp('It was not less than .5!')
end
```

## PRACTICE 4.2

Write a script *printsindegorrad* that will:

- prompt the user for an angle
- prompt the user for (r)adians or (d)egrees, with radians as the default
- if the user enters 'd', the **sind** function will be used to get the sine of the angle in degrees; otherwise, the **sin** function will be used. Which sine function to use will be based solely on whether the user entered a 'd' or not ('d' means degrees, so **sind** is used; otherwise, for any other character the default of radians is assumed, so **sin** is used)
- print the result.

Here are examples of running the script:

```
>> printsindegorrad
Enter the angle: 45
(r)adians (the default) or (d)egrees: d
The sin is 0.71

>> printsindegorrad
Enter the angle: pi
(r)adians (the default) or (d)egrees: r
The sin is 0.00
```

One application of an **if-else** statement is to check for errors in the inputs to a script (this is called **error-checking**). For example, an earlier script prompted the user for a radius, and then used that to calculate the area of a circle. However, it did not check to make sure that the radius was valid (e.g., a positive number). Here is a modified script that checks the radius:

```
checkradius.m

% This script calculates the area of a circle
% It error-checks the user's radius
radius = input('Please enter the radius: ');
if radius <= 0
    fprintf('Sorry; %.2f is not a valid radius\n', radius)
else
    area = calcrea(radius);
    fprintf('For a circle with a radius of %.2f, ', radius)
    fprintf(' the area is %.2f\n', area)
end
```

Examples of running this script when the user enters invalid and then valid radii are shown as follows:

```
>> checkradius
Please enter the radius: -4
Sorry; -4.00 is not a valid radius
```

```
>> checkradius
Please enter the radius: 5.5
For a circle with a radius of 5.50, the area is 95.03
```

The **if-else** statement in this example chooses between two actions: printing an error message, or using the radius to calculate the area and then printing out the result. Note that the action of the **if** clause is a single statement, whereas the action of the **else** clause is a group of three statements.

MATLAB also has an **error** function that can be used to display an error message; the terminology is that this is *throwing an error*. In the previous script, the **if** clause could be modified to use the **error** function rather than **fprintf**; the result will be displayed in red as with the error messages generated by MATLAB. Also, very importantly, when an error message is thrown, the script stops executing. This is illustrated by the following modified script:

checkraderror.m

```
radius = input('Please enter the radius: ');
if radius <= 0
    error('Sorry; %.2f is not a valid radius\n', radius)
else
    area = pi * radius.^2;
    fprintf('For a circle with a radius of %.2f, ', radius)
    fprintf(' the area is %.2f\n', area)
end
disp("And that is it!")
```

```
>> checkraderror
Please enter the radius: -5
Error using checkraderror (line 3)
Sorry; -5.00 is not a valid radius

>> checkraderror
Please enter the radius: 4.5
For a circle with a radius of 4.50, the area is 63.62
And that is it!
```

If the entered radius is not valid, an error message is thrown and nothing else is executed. However, if the radius is valid, it is used to calculate and print the error; also, the **disp** statement is executed after the **if-else** statement ends.

### 4.3 NESTED IF-ELSE STATEMENTS

The **if-else** statement is used to choose between two actions. To choose from more than two actions, the **if-else** statements can be *nested*, meaning one



statement inside another. For example, consider implementing the following continuous mathematical function  $y = f(x)$ :

$$\begin{aligned} y &= 1 && \text{if } x < -1 \\ y &= x^2 && \text{if } -1 \leq x \leq 2 \\ y &= 4 && \text{if } x > 2 \end{aligned}$$

The value of  $y$  is based on the value of  $x$ , which could be in one of three possible ranges. Choosing which range could be accomplished with three separate **if** statements, as follows:

```
if x < -1
    y = 1;
end
if x >= -1 && x <= 2
    y = x^2;
end
if x > 2
    y = 4;
end
```

Note that the **&&** in the expression of the second if statement is necessary. Writing the expression as  $-1 \leq x \leq 2$  would be incorrect; recall from [Chapter 1](#) that that expression would always be **true**, regardless of the value of the variable  $x$ .

As the three possibilities are mutually exclusive, the value of  $y$  can be determined by using three separate **if** statements. However, this is not very efficient code: all three **logical** expressions must be evaluated, regardless of the range in which  $x$  falls. For example, if  $x$  is less than  $-1$ , the first expression is **true** and 1 would be assigned to  $y$ . However, the two expressions in the next two **if** statements are still evaluated. Instead of writing it this way, the statements can be nested so that the entire **if-else** statement ends when an expression is found to be **true**:

```
if x < -1
    y = 1;
else
    % If we are here, x must be >= -1
    % Use an if-else statement to choose
    % between the two remaining ranges
    if x <= 2
        y = x^2;
    else
        % No need to check
        % If we are here, x must be > 2
        y = 4;
    end
end
end
```

By using a nested **if-else** to choose from among the three possibilities, not all conditions must be tested as they were in the previous example. In this case, if  $x$  is less than  $-1$ , the statement to assign 1 to  $y$  is executed and the **if-else** statement is completed, so no other conditions are tested. If, however,  $x$  is not less than  $-1$ , the **else** clause is executed. If the **else** clause is executed, then we already know that  $x$  is greater than or equal to  $-1$ , so that part does not need to be tested.

Instead, there are only two remaining possibilities: either  $x$  is less than or equal to 2, or it is greater than 2. An **if-else** statement is used to choose between those two possibilities. So, the action of the **else** clause was another **if-else** statement. Although it is long, all of the above code is one **if-else** statement, a nested **if-else** statement. The actions are indented to show the structure of the statement. Nesting **if-else** statements in this way can be used to choose from among 3, 4, 5, 6, ...; the possibilities are practically endless!

This is actually an example of a particular kind of nested **if-else** called a *cascading if-else* statement. This is a type of nested **if-else** statement in which the conditions and actions cascade in a stair-like pattern.

Not all nested **if-else** statements are cascading. For example, consider the following (which assumes that a variable  $x$  has been initialized):

```
if x >= 0
    if x < 4
        disp('a')
    else
        disp('b')
    end
else
    disp('c')
end
```

### 4.3.1 The **elseif** Clause

#### THE PROGRAMMING CONCEPT

In some programming languages, choosing from multiple options means using nested **if-else** statements. However, MATLAB has another method of accomplishing this using the **elseif** clause.

#### THE EFFICIENT METHOD

To choose from among more than two actions, the **elseif** clause is used. For example, if there are  $n$  choices (where  $n > 3$  in this example), the following general form would be used:

## THE EFFICIENT METHOD—CONT'D

```

if condition1
    action1
elseif condition2
    action2
elseif condition3
    action3
% etc: there can be many of these
else
    actionn    % the nth action
end

```

The actions of the if, elseif, and else clauses are naturally bracketed by the reserved words if, elseif, else, and end.

For example, a previous example could be written using the elseif clause, rather than nesting if-else statements:

```

if x < -1
    y = 1;
elseif x <= 2
    y = x^2;
else
    y = 4;
end

```

Note that in this example, we only need one end. So, there are three ways of accomplishing the original task: using three separate if statements, using nested if-else statements, and using an if statement with elseif clauses, which is the simplest.

This could be implemented in a function that receives a value of  $x$  and returns the corresponding value of  $y$ :

```

calcy.m

function y = calcy(x)
% calcy calculates y as a function of x
% Format of call: calcy(x)
% y = 1      if    x < -1
% y = x^2    if    -1 <= x <= 2
% y = 4      if    x > 2

if x < -1
    y = 1;
elseif x <= 2
    y = x^2;
else
    y = 4;
end
end

```

```
>> x = 1.1;
>> y = calcy(x)
y =
    1.2100
```

## QUICK QUESTION!

How could you write a function to determine whether an input argument is a scalar, a vector, or a matrix?

**Answer:** To do this, the **size** function can be used to find the dimensions of the input argument. If both the number of rows and columns is equal to 1, then the input argument is a scalar. If, however, only one dimension is 1, the input argument is a vector (either a row or column vector). If neither dimension is 1, the input argument is a matrix. These three options can be tested using a nested **if-else** statement. In this example, the word 'scalar', 'vector', or 'matrix' is returned from the function.

findargtype.m

```
function outtype = findargtype(inputarg)
% findargtype determines whether the input
% argument is a scalar, vector, or matrix
% Format of call: findargtype (inputArgument)
% Returns a string

[r c] = size(inputarg);
if r == 1 && c == 1
    outtype = 'scalar';
elseif r == 1 || c == 1
    outtype = 'vector';
else
    outtype = 'matrix';
end
end
```

Note that there is no need to check for the last case: if the input argument isn't a scalar or a vector, it must be a matrix! Examples of calling this function are:

```
>> findargtype(33)
ans =
    'scalar'

>> disp(findargtype(2:5))
vector

>> findargtype(zeros(2,3))
ans =
    'matrix'
```

---

## PRACTICE 4.3

Modify the function *findargtype* to return either 'scalar', 'row vector', 'column vector', or 'matrix', depending on the input argument.

---

---

## PRACTICE 4.4

Modify the original function *findargtype* to use three separate **if** statements instead of a nested **if-else** statement.

---

Another example demonstrates choosing from more than just a few options. The following function receives an integer quiz grade, which should be in the range from 0 to 10. The function then returns a corresponding letter grade, according to the following scheme: a 9 or 10 is an 'A', an 8 is a 'B', a 7 is a 'C', a 6 is a 'D', and anything below that is an 'F'. As the possibilities are mutually exclusive, we could implement the grading scheme using separate **if** statements. However, it is more efficient to have one **if-else** statement with multiple **elseif** clauses. Also, the function returns the letter 'X' if the quiz grade is not valid. The function assumes that the input is an integer.

```
letgrade.m

function grade = letgrade(quiz)
% letgrade returns the letter grade corresponding
% to the integer quiz grade argument
% Format of call: letgrade(integerQuiz)
% Returns a character

% First, error-check
if quiz < 0 || quiz > 10
    grade = 'X';

% If here, it is valid so figure out the
% corresponding letter grade
elseif quiz == 9 || quiz == 10
    grade = 'A';
elseif quiz == 8
    grade = 'B';
elseif quiz == 7
    grade = 'C';
elseif quiz == 6
    grade = 'D';
else
    grade = 'F';
end
end
```

Three examples of calling this function are:

```
>> quiz = 8;
>> lettergrade = letgrade(quiz)
lettergrade =
    'B'

>> quiz = 4;
>> letgrade(quiz)
ans =
    'F'

>> lg = letgrade(22)
lg =
    'X'
```

In the part of this **if** statement that chooses the appropriate letter grade to return, all of the **logical** expressions are testing the value of the variable *quiz* to see if it is equal to several possible values, in sequence (first 9 or 10, then 8, then 7, etc.). This part can be replaced by a **switch** statement.

## 4.4 THE SWITCH STATEMENT

A **switch** statement can often be used in place of a nested **if-else** or an **if** statement with many **elseif** clauses. **Switch** statements are used when an expression is tested to see whether it is *equal to* one of several possible values.

The general form of the **switch** statement is:

```
switch switch_expression
case caseexp1
    action1
case caseexp2
    action2
case caseexp3
    action3
% etc: there can be many of these
otherwise
    actionn
end
```

The **switch** statement starts with the reserved word **switch** and ends with the reserved word **end**. The *switch\_expression* is compared, in sequence, to the **case** expressions (*caseexp1*, *caseexp2*, etc.). If the value of the *switch\_expression* matches *caseexp1*, for example, then *action1* is executed and the **switch** statement ends. If the value matches *caseexp3*, then *action3* is executed and, in

general, if the value matches *case<sub>i</sub>* where *i* can be any integer from 1 to *n*, then *action<sub>i</sub>* is executed. If the value of the *switch\_expression* does not match any of the *case* expressions, the action after the word *otherwise* is executed (the *n*th action, *action<sub>n</sub>*) if there is an *otherwise* (if not, no action is executed). It is not necessary to have an *otherwise* clause, although it is frequently useful. The *switch\_expression* must be either a scalar or a character vector.

For the previous example, the *switch* statement can be used as follows:

```
switchletgrade.m

function grade = switchletgrade(quiz)
% switchletgrade returns the letter grade corresponding
%   to the integer quiz grade argument using switch
% Format of call: switchletgrade(integerQuiz)
% Returns a character

% First, error-check
if quiz < 0 || quiz > 10
    grade = 'X';
else
    % If here, it is valid so figure out the
    %   corresponding letter grade using a switch
    switch quiz
        case 10
            grade = 'A';
        case 9
            grade = 'A';
        case 8
            grade = 'B';
        case 7
            grade = 'C';
        case 6
            grade = 'D';
        otherwise
            grade = 'F';
    end
end
end
```

Here are two examples of calling this function:

```
>> quiz = 22;
>> lg = switchletgrade(quiz)
lg =
    'X'

>> switchletgrade(9)
ans =
    'A'
```

#### Note

that it is assumed that the user will enter an integer value. If the user does not, either an error message will be printed or an incorrect result will be returned. Methods for remedying this will be discussed in [Chapter 5](#).

As the same action of printing 'A' is desired for more than one grade, these can be combined as follows:

```
switch quiz
    case {10,9}
        grade = 'A';
    case 8
        grade = 'B';
    % etc.
```

The curly braces around the case expressions 10 and 9 are necessary.

In this example, we error-checked first using an if-else statement. Then, if the grade was in the valid range, a switch statement was used to find the corresponding letter grade.

Sometimes the otherwise clause is used for the error message rather than first using an if-else statement. For example, if the user is supposed to enter only a 1, 3, or 5, the script might be organized as follows:

```
switcherror.m

% Example of otherwise for error message

choice = input('Enter a 1, 3, or 5: ');

switch choice
    case 1
        disp('It''s a one!!')
    case 3
        disp('It''s a three!!')
    case 5
        disp('It''s a five!!')
    otherwise
        disp('Follow directions next time!!')
end
```

In this example, actions are taken if the user correctly enters one of the valid options. If the user does not, the otherwise clause handles printing an error message. Note the use of two single quotes within the string to print one quote.

```
>> switcherror
Enter a 1, 3, or 5: 4
Follow directions next time!!
```

Note that the order of the case expressions does not matter, except that this is the order in which they will be evaluated.

MATLAB has a built-in function called **menu** that will display a Figure Window with pushbuttons for the options. A script that uses this **menu** function would

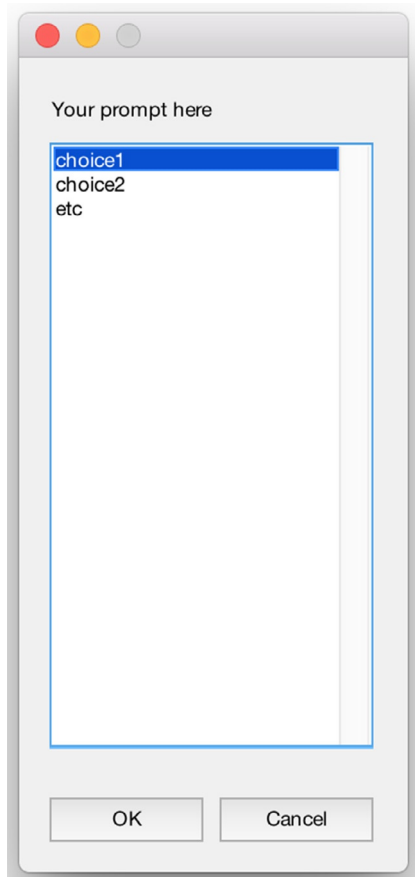


then use either an **if-else** statement or a **switch** statement to take an appropriate action based on the button pushed. As of Version R2015b, however, the **menu** function is no longer recommended. An alternative, which is not quite as simple, is the **listdlg** function. In a call to **listdlg** that allows the user to make only one choice, the format is:

```
>> var = listdlg('SelectionMode', 'Single', ...  
                'PromptString', 'Your prompt here', ...  
                'ListString', {'choice1', 'choice2', 'etc'});
```

This will bring up a Figure Window, as seen in [Figure 4.1](#).

There are three choices. If the user highlights 'choice1' and presses the 'OK' button, the value of *var* will be 1; if the user highlights 'choice2' and presses 'OK',



**FIGURE 4.1**  
Generic listdlg Figure Window.

the value of *var* will be 2, and if the user chooses 'etc' and presses 'OK', the value of *var* will be 3. On the other hand, if the user presses the red button or 'Cancel', the value of *var* will be the empty vector. The inputs to the **listdlg** function are in what are called Parameter/Value pairs. The first pair, 'SelectionMode', 'Single', should always be used. For the 'PromptString' parameter, you enter your own prompt, and for the 'ListString' parameter, you list your options in curly braces.

## 4.5 THE “IS” FUNCTIONS IN MATLAB

There are a lot of functions that are built into MATLAB that test whether or not something is **true**; these functions have names that begin with the word “is”. For example, we have already seen the use of the **isequal** function to compare arrays for equality. As another example, the function called **isletter** returns **logical 1** if the character argument is a letter of the alphabet, or 0 if it is not:

```
>> isletter('h')
ans =
    1
>> isletter('4')
ans =
    0
```

The **isletter** function will return **logical true** or **false**, so it can be used in a condition in an **if** statement. For example, here is code that would prompt the user for a character, and then print whether or not it is a letter:

```
mychar = input('Please enter a char: ','s');
if isletter(mychar)
    disp('Is a letter')
else
    disp('Not a letter')
end
```

When used in an **if** statement, it is not necessary to test the value to see whether the result from **isletter** is equal to 1 or 0; this is redundant. In other words, in the condition of the **if** statement,

```
isletter(mychar)
```

and

```
isletter(mychar) == 1
```

would produce the same results.

## QUICK QUESTION!

How can we write our own function *myisletter* to accomplish the same result as **isletter**?

**Answer:** The function would compare the character's position within the character encoding.

```
myisletter.m

function outlog = myisletter(inchar)
% myisletter returns true if the input argument
% is a letter of the alphabet or false if not
% Format of call: myisletter(inputCharacter)
% Returns logical 1 or 0

outlog = inchar >= 'a' && inchar <= 'z' ...
        || inchar >= 'A' && inchar <= 'Z';
end
```

Note that it is necessary to check for both lowercase and uppercase letters.

The function **isempty** returns **logical true** if a variable is empty, **logical false** if it has a value, or an error message if the variable does not exist. Therefore, it can be used to determine whether a variable has a value yet or not. For example,

```
>> clear
>> isempty(evec)
Unrecognized function or variable 'evec'.

>> evec = [];
>> isempty(evec)
ans =
    1

>> evec = [evec 5];
>> isempty(evec)
ans =
    0
```

The **isempty** function will also determine whether or not a string or character vector variable is empty. This can be used to determine whether the user entered a string in an **input** function. In the following example, when prompted, the user simply hit the Return key.

```
>> istr = input('Please enter a string: ','s');
Please enter a string:
>> isempty(istr)
ans =
    1
```

The following example script will allow the user to make a pizza choice. The **listdlg** function is used to present choices. If the result is empty, an error message is printed; otherwise, a **switch** statement will choose what to print, based on the user's choice.

```
pickpizza.m

mypick = listdlg('SelectionMode', 'Single', ...
                'PromptString', 'Pick a pizza', ...
                'ListString', {'Cheese', 'Shroom', 'Sausage'});

if isempty(mypick)
    disp('No pizza for you today!')
else
    switch mypick
        case 1
            disp('Order a cheese pizza')
        case 2
            disp('Order a mushroom pizza')
        case 3
            disp('Order a sausage pizza')
    end
end
```

## PRACTICE 4.5

Prompt the user for a string, and then print either the string that the user entered or an error message if the user did not enter anything.

The **isa** function can be used to determine whether the first argument is a particular type.

```
>> num = 11;
>> isa(num, 'int16')
ans =
    0
>> isa(num, 'double')
ans =
    1
```

The function **iskeyword** will determine whether or not a string is the name of a keyword in MATLAB, and therefore something that cannot be used as an identifier name. By itself (with no arguments), it will return the list of all keywords. Note that the names of functions like “sin” are not keywords, so their values can be overwritten if used as an identifier name.

```
>> iskeyword('sin')
ans =
    0
>> iskeyword('switch')
ans =
    1
>> iskeyword
ans =
    'break'
    'case'
    'catch'
    % etc.
```

There are many other “is” functions; the complete list can be found in the Help browser.

## ■ Explore Other Interesting Features

If the **menu** function exists in your version of MATLAB, learn how to use it and practice using a switch statement to choose actions based on the user’s choice.

There are many other “is” functions. As more concepts are covered in the book, more and more of these functions will be introduced. Others you may want to explore now include **isvarname** and functions that will tell you whether an argument is a particular type or not (**ischar**, **isfloat**, **isinteger**, **islogical**, **isnumeric**, **isstr**, **isreal**).

There are “is” functions to determine the type of an array: **isvector**, **isrow**, **iscolumn**.

The **try/catch** functions are a particular type of **if-else** used to find and avoid potential errors. They may be a bit complicated to understand at this point, but keep them in mind for the future! ■

## SUMMARY

## COMMON PITFALLS

- Using `=` instead of `==` for equality in conditions
- Putting a space in the keyword **elseif**
- Not using quotes when comparing a string variable to a string, such as

```
letter == y
instead of
letter == 'y'
```

- Not spelling out an entire **logical** expression. An example is typing

```
radius || height <= 0
```

instead of

```
radius <= 0 || height <= 0
```

or typing

```
letter == 'y' || 'Y'
```

instead of

```
letter == 'y' || letter == 'Y'
```

Note that these are logically incorrect, but would not result in error messages. Note also that the expression “`letter == 'y' || 'Y'`” will ALWAYS be **true**, regardless of the value of the variable *letter*, as ‘Y’ is a nonzero value and therefore a **true** expression.

- Writing conditions that are more complicated than necessary, such as

```
if (x < 5) == 1
```

instead of just

```
if (x < 5)
```

(The “`==1`” is redundant.)

- Using an **if** statement instead of an **if-else** statement for error-checking; for example,

```
% Wrong method
if error occurs
    print error message
end
continue rest of code
```

instead of

```
% Correct method
if error occurs
    print error message
else
    continue rest of code
end
```

In the first example, the error message would be printed, but then the program would continue anyway.

## PROGRAMMING STYLE GUIDELINES

- Use indentation to show the structure of a script or function. In particular, the actions in an `if` statement should be indented.
- When the `else` clause isn't needed, use an `if` statement rather than an `if-else` statement. The following is an example:

```
if unit == 'i'
    len = len * 2.54;
else
    len = len; % this does nothing so skip it!
end
```

Instead, just use:

```
if unit == 'i'
    len = len * 2.54;
end
```

- Do not put unnecessary conditions on `else` or `elseif` clauses. For example, the following prints one thing if the value of a variable *number* is equal to 5, and something else if it is not.

```
if number == 5
    disp('It is a 5')
elseif number ~= 5
    disp('It is not a 5')
end
```

The second condition, however, is not necessary. Either the value is 5 or not, so just the `else` would handle this:

```
if number == 5
    disp('It is a 5')
else
    disp('It is not a 5')
end
```

| MATLAB Reserved Words |        |           |
|-----------------------|--------|-----------|
| if                    | else   | otherwise |
| switch                | elseif | case      |

| MATLAB Functions and Commands |          |           |
|-------------------------------|----------|-----------|
| error                         | isletter | isa       |
| menu                          | isempty  | iskeyword |
| listdlg                       |          |           |

## Exercises

1. Write a script that tests whether the user can follow instructions. It prompts the user to enter an 'x'. If the user enters anything other than an 'x', it prints an error message—otherwise, the script does nothing.
2. Write a function *nexthour* that receives one integer argument, which is an hour of the day, and returns the next hour. This assumes a 12-hour clock; so, for example, the next hour after 12 would be 1. Here are two examples of calling this function.
 

```
>> fprintf('The next hour will be %d.\n',nexthour(3))
The next hour will be 4.
>> fprintf('The next hour will be %d.\n',nexthour(12))
The next hour will be 1.
```
3. Write a script that will prompt the user for his or her favorite letter, and then either echo print it or print an error message if the user does not enter a letter.
4. When would you use just an **if** statement and not an **if-else**?
5. Come up with “trigger words” in a problem statement that would tell you when it would be appropriate to use **if**, **if-else**, or **switch** statements.
6. Write a statement that will store **logical true** in a variable named *isit* if the value of a variable *x* is in the range from 0 to 10, or **logical false** if not. Do this with just one assignment statement, with no **if** or **if-else** statement!
7. A year-over-year (YOY) analysis compares a statistic from one year to the same statistic a year earlier. For example, the number of books sold in June 2017 could be compared to the number of books sold in June 2016. Or, the number of manufacturing jobs in March 2017 could be compared to the number of manufacturing jobs in March 2016. If the number is higher this year than last, there is a year-over-year gain; otherwise, it is a loss (we will assume that the numbers will not be the same). Write a script that will prompt the user for employment in a town in August 2017 and also for August 2016, and print whether it was a gain or a loss.
8. The Pythagorean theorem states that for a right triangle, the relationship between the length of the hypotenuse *c* and the lengths of the other sides *a* and *b* is given by:

$$c^2 = a^2 + b^2$$

Write a script that will prompt the user for the lengths *a* and *c*, call a function *findb* to calculate and return the length of *b*, and print the result. Note that any values of *a* or *c* that are less than or equal to zero would not make sense, so the script should print an error message if the user enters any invalid value. Here is the function *findb*:



findb.m

```
function b = findb(a,c)
% Calculates b from a and c
b = sqrt(c^2 - a^2);
end
```

9. A data file “parttolerance.dat” stores, on one line, a part number, and the minimum and maximum values for the valid range that the part could weigh. Write a script “parttol” that will read these values from the file, prompt the user for a weight, and print whether or not that weight is within range. Create a sample data file; for example, the file might store the following:

```
>> type parttolerance.dat
123 44.205 44.287
```

10. Write a script that will prompt the user for a character. It will create an x-vector that has 50 numbers, equally spaced between  $-2\pi$  and  $2\pi$ , and then a y-vector which is  $\cos(x)$ . If the user entered the character ‘r’, it will plot these vectors with red \*s—otherwise, for any other character, it will plot the points with green +s.
11. Simplify this statement:

```
if number > 100
    number = 100;
else
    number = number;
end
```

12. Simplify this statement:

```
if val >= 10
    disp('Hello')
elseif val < 10
    disp('Hi')
end
```

13. The continuity equation in fluid dynamics for steady fluid flow through a stream tube equates the product of the density, velocity, and area at two points that have varying cross-sectional areas. For incompressible flow, the densities are constant, so the equation is  $A_1V_1 = A_2V_2$ . If the areas and  $V_1$  are known,  $V_2$  can be found as  $\frac{A_1}{A_2}V_1$ . Therefore, whether the velocity at the second point increases or decreases depends on the areas at the two points. Write a script that will prompt the user for the two areas in square feet and will print whether the velocity at the second point will increase, decrease, or remain the same as at the first point.
14. Given the following function:

divit.m

```
function myout = divit(myin)
myout = 1/myin;
end
```

Write a script *divornot* that will prompt the user for a number (assume the user does enter a number). The script will then print “cannot call function” if the

user enters a 0, or it will pass the user's number to the function and print the result.

15. Modify your script for Problem 14 to throw an **error** if the user enters a 0.
16. In chemistry, the pH of an aqueous solution is a measure of its acidity. The pH scale ranges from 0 to 14, inclusive. A solution with a pH of 7 is said to be *neutral*, a solution with a pH greater than 7 is *basic*, and a solution with a pH less than 7 is *acidic*. Write a script that will prompt the user for the pH of a solution and will print whether it is neutral, basic, or acidic. If the user enters an invalid pH, an error message will be printed.
17. Write a script called *donotdoit* that will tell the user not to enter anything, and then prompts the user to enter an integer. Then, the script will print whether or not the user entered anything.
18. In a script, the user is supposed to enter either a 'y' or 'n' in response to a prompt. The user's input is read into a character variable called *letter*. The script will print "OK, continuing" if the user enters either a 'y' or 'Y' or it will print "OK, halting" if the user enters an 'n' or 'N' or "Error" if the user enters anything else. Put this statement in the script first:

```
letter = input('Enter your answer: ', 's');
```

Write the script using a single nested **if-else** statement (**elseif** clause is permitted).

19. Write the script from the previous exercise using a **switch** statement instead.
20. In aerodynamics, the Mach number is a critical quantity. It is defined as the ratio of the speed of an object (e.g., an aircraft) to the speed of sound. If the Mach number is less than 1, the flow is subsonic; if the Mach number is equal to 1, the flow is transonic; and if the Mach number is greater than 1, the flow is supersonic. Write a script that will prompt the user for the speed of an aircraft and the speed of sound at the aircraft's current altitude and will print whether the condition is subsonic, transonic, or supersonic.
21. Write a script that will generate one random integer and will print whether the random integer is an even or an odd number. (Hint: an even number is divisible by 2, whereas an odd number is not; so check the remainder after dividing by 2.)
22. Whether a storm is a tropical depression, tropical storm, or hurricane is determined by the average sustained wind speed. In miles per hour, a storm is a tropical depression if the winds are less than 38 mph. It is a tropical storm if the winds are between 39 and 73 mph, and it is a hurricane if the wind speeds are  $\geq 74$  mph. Write a script that will prompt the user for the wind speed of the storm and will print which type of storm it is.
23. The Beaufort Wind Scale is used to characterize the strength of winds. The scale uses integer values and goes from a force of 0, which is no wind, up to 12, which is a hurricane. The following script first generates a random force value. Then, it prints a message regarding what type of wind that force represents, using a **switch** statement. You are to rewrite this **switch** statement as one nested

if-else statement that accomplishes exactly the same thing. You may use else and/or elseif clauses.

```
ranforce = randi([0, 12]);
switch ranforce
    case 0
        disp('There is no wind')
    case {1,2,3,4,5,6}
        disp('There is a breeze')
    case {7,8,9}
        disp('This is a gale')
    case {10,11}
        disp('It is a storm')
    case 12
        disp('Hello, Hurricane!')
end
```

24. Rewrite the following nested if-else statement as a switch statement that accomplishes exactly the same thing. Assume that *num* is an integer variable that has been initialized, and that there are functions *f1*, *f2*, *f3*, and *f4*. Do not use any if or if-else statements in the actions in the switch statement, only calls to the four functions.

```
if num < -2 || num > 4
    f1(num)
else
    if num <= 2
        if num >= 0
            f2(num)
        else
            f3(num)
        end
    else
        f4(num)
    end
end
```

25. Write a script *areaMenu* that will print a list consisting of “cylinder”, “circle”, and “rectangle”. It prompts the user to choose one, and then prompts the user for the appropriate quantities (e.g., the radius of the circle) and then prints its area. If the user enters an invalid choice, the script simply prints an error message. The script should use a nested if-else statement to accomplish this. Here are two examples of running it (units are assumed to be inches).

```
>> areaMenu
Menu
1. Cylinder
2. Circle
```

```

3. Rectangle
Please choose one: 2
Enter the radius of the circle: 4.1
The area is 52.81

>> areaMenu
Menu
1. Cylinder
2. Circle
3. Rectangle
Please choose one: 3
Enter the length: 4
Enter the width: 6
The area is 24.00

```

26. Modify the *areaMenu* script to use a **switch** statement to decide which area to calculate.
27. Modify the *areaMenu* script to use the **listdlg** function instead of printing a menu.
28. Write a script that will prompt the user for a string and then print whether it was empty or not.
29. Simplify this statement:

```

if iskeyword('else') == 1
    disp('Cannot use as a variable name')
end

```

30. Store a value in a variable and then use **isa** to test to see whether or not it is the type **double**.
31. Write a function called “makemat” that will receive two row vectors as input arguments, and from them create and return a matrix with two rows. You may not assume that the length of the vectors is known. Also, the vectors may be of different lengths. If that is the case, add 0’s to the end of one vector first to make it as long as the other. For example, a call to the function might be:

```

>> makemat(1:4, 2:7)
ans =
    1    2    3    4    0    0
    2    3    4    5    6    7

```