Chapter 7
File Access

Chapter Table of Contents

Chapter 7
File Access

Overview

In this chapter you learn about external files and how to refer to an external file, whether it is a text file or a binary file. You learn how to read data from a file using the INFILE and INPUT statements and how to write data to an external file using the FILE and PUT statements.

With external files, you must know the format in which the data are stored or to be written. This is in contrast to SAS data sets, which are specialized files with a structure that is already known to the SAS System.

The Interactive Matrix Language statements used to access files are very similar to the corresponding statements in the SAS DATA step. The following table summarizes the IML statements and their functions.

Statement	**Function**
CLOSEFILE	closes an external file
FILE	opens an external file for output
INFILE	opens an external file for input
INPUT	reads from the current input file
PUT	writes to the current output file
SHOW: *FILES*	shows all open files, their attributes, and their status (current input and output files)

Referring to an External File

Suppose that you have data for students in a class. You have recorded the values for the variables NAME, SEX, AGE, HEIGHT, and WEIGHT for each student and have stored the data in an external text file named USER.TEXT.CLASS. If you want to read this data into IML variables, you need to indicate where the data are stored. In other words, you need to name the input file. If you want to write data from matrices to a file, you also need to name an output file.

There are two ways to refer to an input or output file: a *filepath* and a *filename*. A *filepath* is the name of the file as it is known to the operating system. A *filename* is an indirect SAS reference to the file made using the FILENAME statement. You can identify a file in either way using the FILE and INFILE statements.

For example, you can refer to the input file where the class data are stored using a literal filepath, that is, a quoted string. The statement

```
infile 'user.text.class';
```

opens the file USER.TEXT.CLASS for input. Similarly, if you want to output data to the file USER.TEXT.NEWCLASS, you need to reference the output file with the statement

```
file 'user.text.newclass';
```

You can also refer to external files using a *filename*. When using a filename as the operand, simply give the name. The name must be one already associated with a filepath by a previously issued FILENAME statement.

For example, suppose you want to reference the file with the class data using a FILENAME statement. First, you must associate the filepath with an alias (called a *fileref*), say INCLASS. Then you can refer to USER.TEXT.CLASS with the fileref INCLASS.

The following statements accomplish the same thing as the previous INFILE statement with the quoted filepath:

```
filename inclass 'user.text.class';
infile inclass;
```

You can use the same technique for output files. The following statements have the same effect as the previous file statement:

```
filename outclass 'user.text.newclass';
file outclass;
```

Three filenames have special meaning to IML: CARDS, LOG, and PRINT. These refer to the standard input and output streams for all SAS sessions, as described below:

CARDS is a special filename for instream input data.

LOG is a special filename for log output.

PRINT is a special filename for standard print output.

When the filepath is specified, there is a limit of 64 characters to the operand.

Types of External Files

Most files that you work with are *text files*, which means that they can be edited and displayed without any special program. Text files under most host environments have special characters, called carriage-control characters or end-of-line characters, to separate one record from the next.

If your file does not adhere to these conventions, it is called a *binary file*. Typically, binary files do not have the usual record separators, and they may use any binary codes, including unprintable control characters. If you want to read a binary file, you must specify RECFM=N in the INFILE statement and use the byte operand (<) in the INPUT statement to specify the length of each item you want read. Treating a file as

binary enables you to have direct access to a file position by byte-address using the byte operand (>) in the INPUT or PUT statement.

You write data to an external file using the FILE and PUT statements. The output file can be text or binary. If your output file is binary, you must specify RECFM=N in the FILE statement. One difference between binary and text files in output is that the PUT statement does not put the record-separator characters on the end of each record written for binary files.

Reading from an External File

After you have chosen a method to refer to the external file you want to read, you need an INFILE statement to open it for input and an INPUT statement to tell IML how to read the data.

The next several sections cover how to use an INFILE statement and how to specify an INPUT statement so that you can input data from an external file.

Using the INFILE Statement

An INFILE statement identifies an external file containing data that you want to read. It opens the file for input or, if the file is already open, makes it the current input file. This means that subsequent INPUT statements are read from this file until another file is made the current input file.

The following options can be used with the INFILE statement:

FLOWOVER
enables the INPUT statement to go to the next record to obtain values for the variables.

LENGTH=*variable*
names a variable containing the length of the current record, where the value is set to the number of bytes used after each INPUT statement.

MISSOVER
prevents reading from the next input record when an INPUT statement reaches the end of the current record without finding values for all variables. It assigns missing values to all values that are expected but not found.

RECFM=N
specifies that the file is to be read in as a pure binary file rather than as a file with record-separator characters. You must use the byte operands (< and >) to get new records rather than separate INPUT statements or the new line operator (/).

STOPOVER
stops reading when an INPUT statement reaches the end of the current record without finding values for all variables in the statement. It treats going past the end of a record as an error condition, triggering an end-of-file condition. The STOPOVER option is the default.

The FLOWOVER, MISSOVER, and STOPOVER options control how the INPUT statement works when you try to read past the end of a record. You can specify only one of these options. Read these options carefully so that you understand them completely.

Below is an example using the INFILE statement with a FILENAME statement to read the class data file. The MISSOVER option is used to prevent reading from the next record if values for all variables in the INPUT statement are not found.

```
filename inclass 'user.text.class';
infile inclass missover;
```

You can specify the filepath with a quoted literal also. The preceding statements could be written as

```
infile 'user.text.class' missover;
```

Using the INPUT Statement

Once you have referenced the data file containing your data with an INFILE statement, you need to tell IML exactly how the data are arranged:

- the number of variables and their names
- each variable's type, either numeric or character
- the format of each variable's values
- the columns that correspond to each variable

In other words, you must tell IML how to read the data.

The INPUT statement describes the arrangement of values in an input record. The INPUT statement reads records from a file specified in the previously executed INFILE statement, reading the values into IML variables.

There are two ways to describe a record's values in an IML INPUT statement:

- list (or scanning) input
- formatted input

Here are several examples of valid INPUT statements for the class data file, depending, of course, on how the data are stored.

If the data are stored with a blank or a comma between fields, then list input can be used. For example, the INPUT statement for the class data file might look as follows:

```
infile inclass;
input name $ sex $ age height weight;
```

These statements tell IML the following:

- There are five variables: NAME, SEX, AGE, HEIGHT and WEIGHT.
- Data fields are separated by commas or blanks.
- NAME and SEX are character variables, as indicated by the dollar sign ($).
- AGE, HEIGHT, and WEIGHT are numeric variables, the default.

The data must be stored in the same order in which the variables are listed in the INPUT statement. Otherwise, you can use formatted input, which is column specific. Formatted input is the most flexible and can handle any data file. Your INPUT statement for the class data file might look as follows:

```
infile inclass;
input @1 name $char8. @10 sex $char1. @15 age 2.0
   @20 height 4.1 @25 weight 5.1;
```

These statements tell IML the following:

- NAME is a character variable; its value begins in column 1 (indicated by @1) and occupies eight columns ($char8.).
- SEX is a character variable; its value is in column 10 ($char1.).
- AGE is a numeric variable; its value is found in columns 15 and 16 and has no decimal places (2.0).
- HEIGHT is a numeric variable found in columns 20 through 23 with one decimal place implied (4.1).
- WEIGHT is a numeric variable found in columns 25 through 29 with one decimal place implied (5.1).

The next sections discuss these two modes of input.

List Input

If your data are recorded with a comma or one or more blanks between data fields, you can use list input to read your data. If you have missing values, that is, unknown values, they must be represented by a period (.) rather than a blank field.

When IML looks for a value, it skips past blanks and tab characters. Then it scans for a delimiter to the value. The delimiter is a blank, a comma, or the end of the record. When the ampersand (&) format modifier is used, IML looks for two blanks, a comma, or the end of the record.

The general form of the INPUT statement for list input is

INPUT *variable* < *$* > < *&* > < . . . *variable* < *$* > < *&* > > **;**

where

variable names the variable to be read by the INPUT statement.

$ indicates that the preceding variable is character.

& indicates that a character value may have a single embedded blank. Because a blank normally indicates the end of a data value, use the ampersand format modifier to indicate the end of the value with at least two blanks or a comma.

With list input, IML scans the input lines for values. Consider using list input when

- blanks or commas separate input values
- periods rather than blanks represent missing values

List input is the default in several situations. Descriptions of these situations and the behavior of IML follow:

- If no input format is specified for a variable, IML scans for a number.
- If a single dollar sign or ampersand format modifier is specified, IML scans for a character value. The ampersand format modifier allows single embedded blanks to occur.
- If a format is given with width unspecified or 0, IML scans for the first blank or comma.

If the end of a record is encountered before IML finds a value, then the behavior is as described by the record overflow options in the INFILE statement discussed in the section "Using the INFILE Statement."

When you read with list input, the order of the variables listed in the INPUT statement must agree with the order of the values in the data file. For example, consider the following data:

```
Alice    f    10   61   97
Beth     f    11   64   105
Bill     m    12   63   110
```

You can use list input to read this data by specifying the following INPUT statement:

```
input name $ sex $ age height weight;
```

Note: This statement implies that the variables are stored in the order given. That is, each line of data contains a student's NAME, SEX, AGE, HEIGHT, and WEIGHT in that order and separated by at least one blank or by a comma.

Formatted Input

The alternative to list input is formatted input. An INPUT statement reading formatted input must have a SAS informat after each variable. An *informat* gives the data

type and field width of an input value. Formatted input may be used with pointer controls and format modifiers. Note, however, that neither pointer controls nor format modifiers are necessary for formatted input.

Pointer control features

Pointer controls reset the pointer's column and line positions and tell the INPUT statement where to go to read the data value. You use pointer controls to specify the columns and lines from which you want to read:

- *Column pointer controls* move the pointer to the column you specify.
- *Line pointer controls* move the pointer to the next line.
- *Line hold controls* keep the pointer on the current input line.
- *Binary file indicator controls* indicate that the input line is from a binary file.

Column pointer controls

Column pointer controls indicate in which column an input value starts. Column pointer controls begin with either an at sign (@) or a plus sign (+).

@*n*	moves the pointer to column *n*.
@*point-variable*	moves the pointer to the column given by the current value of *point-variable*.
@(*expression*)	moves the pointer to the column given by the value of the *expression*. The *expression* must evaluate to a positive integer.
+*n*	moves the pointer *n* columns.
+*point-variable*	moves the pointer the number of columns given by the value of *point-variable*.
+(*expression*)	moves the pointer the number of columns given by the value of *expression*. The value of *expression* can be positive or negative.

Here are some examples using column pointer controls:

Example	Meaning
@12	go to column 12
@N	go to the column given by the value of N
@(N−1)	go to the column given by the value of N−1
+5	skip 5 spaces
+N	skip N spaces
+(N+1)	skip N+1 spaces

In the earlier example using formatted input, you used several pointer controls:

```
infile inclass;
input @1 name $char8. @10 sex $char1. @15 age 2.0
      @20 height 4.1 @25 weight 5.1;
```

The @1 moves the pointer to column 1, the @10 moves it to column 10, and so on. You move the pointer to the column where the data field begins and then supply an informat specifying how many columns the variable occupies. The INPUT statement could also be written as

```
input @1 name $char8. +1 sex $char1. +4 age 2. +3 height 4.1
      +1 weight 5.1;
```

In this form, you move the pointer to column 1 (@1) and read eight columns. The pointer is now at column 9. Now, move the pointer +1 columns to column 10 to read SEX. The $char1. informat says to read a character variable occupying one column. After you read the value for SEX, the pointer is at column 11, so move it to column 15 with +4 and read AGE in columns 15 and 16 (the 2. informat). The pointer is now at column 17, so move +3 columns and read HEIGHT. The same idea applies for reading WEIGHT.

Line pointer control

The line pointer control (/) directs IML to skip to the next line of input. You need a line pointer control when a record of data takes more than one line. You use the new line pointer control (/) to skip to the next line and continue reading data. In the example reading the class data, you do not need to skip a line because each line of data contains all the variables for a student.

Line hold control

The trailing at sign (@), when at the end of an INPUT statement, directs IML to hold the pointer on the current record so that you can read more data with subsequent INPUT statements. You can use it to read several records from a single line of data. Sometimes, when a record is very short, say ten columns or so, you can save space in your external file by coding several records on the same line.

Binary file indicator controls

When the external file you want to read is a binary file (RECFM=N is specified in the INFILE statement), you must tell IML how to read the values using the following binary file indicator controls:

>n	start reading the next record at the byte position n in the file.
>*point-variable*	start reading the next record at the byte position in the file given by *point-variable*.
>(*expression*)	start reading the next record at the byte position in the file given by *expression*.
<n	read the number of bytes indicated by the value of n.
<*point-variable*	read the number of bytes indicated by the value of *point-variable*.
<(*expression*)	read the number of bytes indicated by the value of *expression*.

Pattern Searching

You can have the input mechanism search for patterns of text by using the at sign (@) positional with a character operand. IML starts searching at the current position,

advances until it finds the pattern, and leaves the pointer at the position immediately after the found pattern in the input record. For example, the statement

```
input @ 'NAME=' name $;
```

searches for the pattern **NAME=** and then uses list input to read the value after the found pattern.

If the pattern is not found, then the pointer is left past the end of the record, and the rest of the INPUT statement follows the conventions based on the options MISSOVER, STOPOVER, and FLOWOVER described in the section "Using the INFILE Statement" earlier in this chapter. If you use pattern searching, you usually specify the MISSOVER option so that you can control for the occurrences of the pattern not being found.

Notice that the MISSOVER feature enables you to search for a variety of items on the same record, even if some of them are not found. For example, the statements

```
infile in1 missover;
input @1 @ "NAME=" name $
      @1 @ "ADDR=" addr &
      @1 @ "PHONE=" phone $;
```

are able to read in the ADDR variable even if **NAME=** is not found (in which case, NAME is unvalued).

The pattern operand can use any characters except for the following:

% $ [ ] { } < > – ? * # @ ^ ` (backquote)

Record Directives

Each INPUT statement goes to a new record except for the following special cases:

- An at sign (@) at the end of an INPUT statement specifies that the record is to be held for future INPUT statements.
- Binary files (RECFM=N) always hold their records until the > directive.

As discussed in the syntax of the INPUT statement, the line pointer operator (/) instructs the input mechanism to go immediately to the next record. For binary (RECFM=N) files, the > directive is used instead of the /.

Blanks

For character values, the informat determines the way blanks are interpreted. For example, the $CHAR$w$. format reads blanks as part of the whole value, while the BZw. format turns blanks into 0s. Refer to *SAS Language Reference: Dictionary* for more information on informats.

Missing Values

Missing values in formatted input are represented by blanks or a single period for a numeric value and by blanks for a character value.

Matrix Use

Data values are either character or numeric. Input variables always result in scalar (one row by one column) values with type (character or numeric) and length determined by the input format.

End-of-File Condition

End of file is the condition of trying to read a record when there are no more records to read from the file. The consequences of an end-of-file condition are described as follows.

- All the variables in the INPUT statement that encountered end of file are freed of their values. You can use the NROW or NCOL function to test if this has happened.
- If end of file occurs while inside a DO DATA loop, execution is passed to the statement after the END statement in the loop.

For text files, the end of file is encountered first as the end of the last record. The next time input is attempted, the end-of-file condition is raised.

For binary files, the end of file can result in the input mechanism returning a record that is shorter than the requested length. In this case IML still attempts to process the record, using the rules described in the section "Using the INFILE Statement," earlier in this chapter.

The DO DATA mechanism provides a convenient mechanism for handling end of file. For example, to read the class data from the external file USER.TEXT.CLASS into a SAS data set, you need to perform the following steps:

1. Establish a *fileref* referencing the data file.
2. Use an INFILE statement to open the file for input.
3. Initialize any character variables by setting the length.
4. Create a new SAS data set with a CREATE statement. You want to list the variables you plan to input in a VAR clause.
5. Use a DO DATA loop to read the data one line at a time.
6. Write an INPUT statement telling IML how to read the data.
7. Use an APPEND statement to add the new data line to the end of the new SAS data set.
8. End the DO DATA loop.
9. Close the new data set.
10. Close the external file with a CLOSEFILE statement.

Your code would look as follows.

```
filename inclass 'user.text.class';
infile inclass missover;
name="12345678";
sex="1";
create class  var{name sex age height weight};
do data;
   input name $ sex $ age height weight;
   append;
end;
close class;
closefile inclass;
```

Note that the APPEND statement is not executed if the INPUT statement reads past the end of file since IML escapes the loop immediately when the condition is encountered.

Differences with the SAS DATA Step

If you are familiar with the SAS DATA step, you will notice that the following features are supported differently or are not supported in IML:

- The pound sign (#) directive supporting multiple current records is not supported.
- Grouping parentheses are not supported.
- The colon (:) format modifier is not supported.
- The byte operands (< and >) are new features supporting binary files.
- The ampersand (&) format modifier causes IML to stop reading data if a comma is encountered. Use of the ampersand format modifier is valid with list input only.
- The RECFM=F option is not supported.

Writing to an External File

If you have data in matrices and you want to write this data to an external file, you need to reference, or point to, the file (as discussed in the section "Referring to an External File"). The FILE statement opens the file for output so that you can write data to it. You need to specify a PUT statement to direct how the data is output. These two statements are discussed in the following sections.

Using the FILE Statement

The FILE statement is used to refer to an external file. If you have values stored in matrices, you can write these values to a file. Just as with the INFILE statement, you need a fileref to point to the file you want to write to. You use a FILE statement to indicate that you want to write to rather than read from a file. For example, if you want to output to the file USER.TEXT.NEWCLASS, you can specify the file with a quoted literal filepath.

```
> file 'user.text.newclass';
```

Otherwise, you can first establish a fileref and then refer to the file by its fileref:

```
> filename outclass 'user.text.class';
> file outclass;
```

There are two options you can use in the FILE statement:

RECFM=N specifies that the file is to be written as a pure binary file without record-separator characters.

LRECL=*operand* specifies the size of the buffer to hold the records.

The FILE statement opens a file for output or, if the file is already open, makes it the current output file so that subsequent PUT statements write to the file. The FILE statement is similar in syntax and operation to the INFILE statement.

Using the PUT Statement

The PUT statement writes lines to the SAS log, to the SAS output file, or to any external file specified in a FILE statement. The file associated with the most recently executed FILE statement is the *current output file*.

You can use the following arguments with the PUT statement:

variable names the IML variable with a value that is put to the current pointer position in the record. The variable must be scalar valued. The put variable can be followed immediately by an output format.

literal gives a literal to be put to the current pointer position in the record. The literal can be followed immediately by an output format.

(expression) must produce a scalar-valued result. The expression can be immediately followed by an output format.

format names the output formats for the values.

pointer-control moves the output pointer to a line or column.

Pointer Control Features

Most PUT statements need the added flexibility obtained with pointer controls. IML keeps track of its position on each output line with a pointer. With specifications in the PUT statement, you can control pointer movement from column to column and line to line. The pointer controls available are discussed in the section "Using the INPUT statement".

Differences with the SAS DATA Step

If you are familiar with the SAS DATA step, you will notice that the following features are supported differently or are not supported:

- The pound sign (#) directive supporting multiple current records is not supported.

- Grouping parentheses are not supported.
- The byte operands (< and >) are a new feature supporting binary files.

Examples

Writing a Matrix to an External File

If you have data stored in an $n \times m$ matrix and you want to output the values to an external file, you need to write out the matrix element by element.

For example, suppose that you have a matrix $\mathbf{X}$ containing data that you want written to the file USER.MATRIX. Suppose also that $\mathbf{X}$ contains 1s and 0s so that the format for output can be one column. You need to do the following:

1. Establish a fileref, for example, OUT.
2. Use a FILE statement to open the file for output.
3. Specify DO loop for the rows of the matrix.
4. Specify DO loop for the columns of the matrix.
5. Use a PUT statement to specify how to write the element value.
6. End the inner DO loop.
7. Skip a line.
8. End the outer DO loop.
9. Close the file.

Your code should look as follows:

```
filename out 'user.matrix';
file out;
   do i=1 to nrow(x);
      do j=1 to ncol(x);
         put (x[i,j]) 1.0 +2 @;
      end;
      put;
   end;
closefile out;
```

The output file contains a record for each row of the matrix. For example, if your matrix is 4×4, then the file might look as follows:

```
1  1  0  1
1  0  0  1
1  1  1  0
0  1  0  1
```

Quick Printing to the PRINT File

You can use the FILE PRINT statement to route output to the standard print file. The following statements generate data that are output to the PRINT file:

```
> file print;
> do a=0 to 6.28 by .2;
>    x=sin(a);
>    p=(x+1)#30;
>    put @1 a 6.4 +p x 8.4;
> end;
```

The result is shown below:

```
0.0000                                  0.0000
0.2000                                       0.1987
0.4000                                             0.3894
0.6000                                                   0.5646
0.8000                                                       0.7174
1.0000                                                           0.8415
1.2000                                                             0.9320
1.4000                                                               0.9854
1.6000                                                               0.9996
1.8000                                                               0.9738
2.0000                                                             0.9093
2.2000                                                         0.8085
2.4000                                                     0.6755
2.6000                                                 0.5155
2.8000                                           0.3350
3.0000                                       0.1411
3.2000                                 -0.0584
3.4000                            -0.2555
3.6000                       -0.4425
3.8000                  -0.6119
4.0000             -0.7568
4.2000         -0.8716
4.4000      -0.9516
4.6000   -0.9937
4.8000   -0.9962
5.0000    -0.9589
5.2000       -0.8835
5.4000          -0.7728
5.6000               -0.6313
5.8000                    -0.4646
6.0000                         -0.2794
6.2000                               -0.0831
```

Listing Your External Files

To list all open files and their current input or current output status, use the SHOW FILES statement.

Closing an External File

The CLOSEFILE statement closes files opened by an INFILE or a FILE statement. You specify the CLOSEFILE statement just as you do the INFILE or FILE statement. For example, the following statements open the external file USER.TEXT.CLASS for input and then close it:

```
filename in 'user.text.class';
infile in;
closefile in;
```

Summary

In this chapter, you learned how to refer to, or point to, an external file with a FILENAME statement. You can use the FILENAME statement whether you want to read from or write to an external file. The file can also be referenced by a quoted literal filepath. You also learned about the difference between a text file and a binary file.

You learned how to read data from an external file with the INFILE and INPUT statements, using either list or formatted input. You learned how to write your matrices to an external file using the FILE and PUT statements. Finally, you learned how to close your files.

The correct bibliographic citation for this manual is as follows: SAS Institute Inc., *SAS/IML User's Guide, Version 8*, Cary, NC: SAS Institute Inc., 1999. 846 pp.

SAS/IML User's Guide, Version 8

ISBN 1–58025–553–1

SAS Institute Inc., SAS Campus Drive, Cary, North Carolina 27513.

1st printing, October 1999