Importing Text Data Files

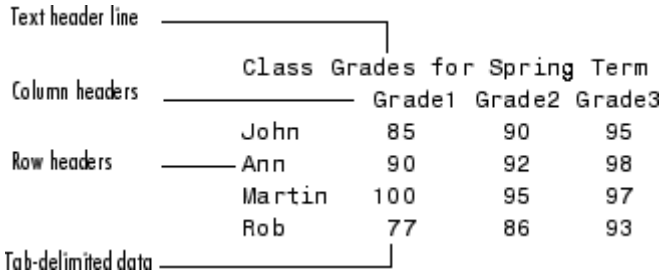
The easiest way to import data from an ASCII file is to use the Import Wizard, a graphical user interface. To start the Import Wizard, select **File** > **Import Data**.

To import without invoking a graphical interface, use `importdata`.

For most files, the Import Wizard and `importdata` automatically detect:

- Row and column headers.
- Field delimiters (characters between data items, such as commas, spaces, tabs, or semicolons).
- MATLAB comments (lines that begin with a percent sign, `'%'`).

For example, you can easily read ASCII data in the following form (see Importing Numeric ASCII Data with Headers):



Requirements for the Import Wizard or importdata

The data in your file must be:

- Rectangular, like a matrix, with the same number of data fields in each row.
- Numeric. Formatted dates and times (such as `'01/01/01'` or `'12:30:45'`) are *not* numeric. However, you can import formatted dates and times as headers.

If your data file does not meet these requirements, consider using `textscan`. For more information, see:

- Importing Nonnumeric ASCII Data
- Importing Nonrectangular ASCII Data

To import files with more complex formats, see Importing Text Data Files with Low-Level I/O.

Importing a Subset of Your Data

The Import Wizard and `importdata` import all rows and columns of your data file. To import only part of your data, use `dlmread` or `textscan`, where:

- `dlmread` requires rectangular, numeric data, but is easy to use. For more information, see Selecting a Range of Numeric Data.
- `textscan` imports a wider variety of file formats, and tracks your position in the file.

For more information, see Importing Large ASCII Data Sets.

Back to Top

Importing Numeric ASCII Data

You can import any ASCII data file with numeric fields easily using the Import Wizard or `importdata`. For example, consider a comma-delimited ASCII data file named `ph.dat`:

```
7.2, 8.5, 6.2, 6.6
5.4, 9.2, 8.1, 7.2
```

Use `importdata` to import the data. Call `whos` to learn the class of the data returned, and type the name of the output variable (in this case, `'ph'`) to see its contents:

```
ph = importdata('ph.dat');

whos ph

Name              Size              Bytes  Class     Attributes
  ph              2x4                  64  double

ph

ph =
    7.2000    8.5000    6.2000    6.6000
    5.4000    9.2000    8.1000    7.2000
```

> **Note** As an alternative to `importdata`, you can import data like `ph.dat` with `load`, `dlmread`, or the Import Wizard. All four approaches return identical 2-by-4 `double` arrays for `ph`.

Selecting a Range of Numeric Data

To select specific rows and columns to import, use `dlmread`. For example, to read the first two columns from `ph.dat`:

```
ph_partial = dlmread('ph.dat', ',', 'A1..B2')

ph_partial =
    7.2000    8.5000
    5.4000    9.2000
```

Importing Formatted Dates and Times

Formatted dates and times (such as `'01/01/01'` or `'12:30:45'`) are not numeric fields. How you import them depends on their location in the file. If the dates and times are:

- In the initial columns, like row headers, use `importdata` or the Import Wizard. For more information, see Importing Numeric ASCII Data with Headers.

- In other columns, use `textscan`. For more information, see Importing Nonnumeric ASCII Data.

Back to Top

Importing Numeric ASCII Data with Headers

You can import any ASCII data file with numeric fields and text headers easily using the Import Wizard or `importdata`.

For example, consider the file `grades.dat`:

```
Class Grades for Spring Term
        Grade1 Grade2 Grade3
John     85      90     95
Ann      90      92     98
Martin  100      95     97
Rob      77      86     93
```

A call to `importdata` of the form

```
grades_imp = importdata('grades.dat');
```

Returns the same results as a call to the Import Wizard:

```
grades_imp = uiimport('grades.dat');
```

You can also start the Import Wizard by selecting **File** > **Import Data**.

Because the data includes both row and column headers, `importdata` or the Import Wizard returns the structure `grades_imp` as follows:

```
grades_imp =
        data: [4x3 double]
    textdata: {6x1 cell}

grades_imp.data =
    85    90    95
    90    92    98
   100    95    97
    77    86    93

grades_imp.textdata =
    'Class Grades for Spring Term'
    '        Grade1 Grade2 Grade3'
    'John'
    'Ann'
    'Martin'
    'Rob'
```

Additional Variables and Fields

If your data file includes either column headers or a single column of row headers, but not both:

- You can create vectors based on the rows or columns in your file with the Import Wizard. For more information, see Creating Column or Row Vectors from Text Files or Spreadsheets.
- `importdata` and the Import Wizard store the row or column headers in `rowheaders` or `colheaders` fields of the output structure. For example, if `grades_col.dat` includes only column headers:

```
Grade1 Grade2 Grade3
 85      90     95
 90      92     98
 100     95     97
 77      86     93
```

A call to `importdata` of the form

```
grades_col = importdata('grades_col.dat');
```

Or a call to the Import Wizard, using the default settings:

```
grades_col = uiimport('grades_col.dat');
```

returns

```
grades_col =
           data: [4x3 double]
       textdata: {'Grade1'  'Grade2'  'Grade3'}
    colheaders: {'Grade1'  'Grade2'  'Grade3'}
```

Restrictions

If your file includes:

- Multiple column headers, `colheaders` contains only the lowest row of header text. However, `textdata` contains all text.
- Nonnumeric characters that are not part of row or column headers, including formatted dates or times, use `textscan` to import the file. For more information, see Importing Nonnumeric ASCII Data.

Back to Top

Importing Nonnumeric ASCII Data

To import an ASCII data file with fields that contain nonnumeric characters, use `textscan`.

For example, you can use `textscan` to import a file called `mydata.dat`:

```
Sally    09/12/2005 12.34 45 Yes
Larry    10/12/2005 34.56 54 Yes
```

```
Tommy    11/12/2005 67.89 23 No
```

Open the File

Preface any calls to `textscan` with a call to `fopen` to open the file for reading, and, when finished, close the file with `fclose`.

Describe Your Data

The `textscan` function is flexible, but requires that you specify more information about your file. Describe each field using format specifiers, such as `'%s'` for a string, `'%d'` for an integer, or `'%f'` for a floating-point number. (For a complete list of format specifiers, see the `textscan` reference page.)

Import into a Cell Array

Send `textscan` the file identifier and the format specifiers to describe the five fields in each row of `mydata.dat`. `textscan` returns a cell array with five cells:

```
fid = fopen('mydata.dat');
mydata = textscan(fid, '%s %s %f %d %s');
fclose(fid);

whos mydata
  Name         Size             Bytes  Class    Attributes

  mydata       1x5                952  cell

mydata =
  {3x1 cell}  {3x1 cell}  [3x1 double]  [3x1 int32]  {3x1 cell}
```

where

```
mydata{1} = {'Sally'; 'Larry'; 'Tommy'}
mydata{2} = {'09/12/2005'; '10/12/2005'; '11/12/2005'}
mydata{3} = [12.3400; 34.5600; 67.8900]
mydata{4} = [45; 54; 23]
mydata{5} = {'Yes'; 'Yes'; 'No'}
```

Back to Top

Importing Nonrectangular ASCII Data

Most of the ASCII data import functions require that your data is *rectangular*, that is, in a regular pattern of columns and rows. The `textscan` function relaxes this restriction, although it requires that your data is in a repeated pattern.

For example, you can use `textscan` to import a file called `nonrect.dat`:

```
begin
v1=12.67
v2=3.14
```

```
v3=6.778
end
begin
v1=21.78
v2=5.24
v3=9.838
end
```

Describe Your Data

To use `textscan`, describe the pattern of the data using format specifiers and delimiter parameters. Typical format specifiers include `'%s'` for a string, `'%d'` for an integer, or `'%f'` for a floating-point number. (For a complete list of format specifiers and parameters, see the `textscan` reference page.)

To import `nonrect.dat`, use the format specifier `'%*s'` to tell `textscan` to skip the strings `'begin'` and `'end'`. Include the literals `'v1='`, `'v2='`, and `'v3='` as part of the format specifiers, so that `textscan` ignores those strings as well.

Since each field is on a new line, the `delimiter` is a newline character (`'\n'`). To combine all the floating-point data into a single array, set the `CollectOutput` parameter to `true`. The final call to `textscan` is:

```
fid = fopen('nonrect.dat');

c = textscan(fid, ...
                '%*s v1=%f v2=%f v3=%f %*s', ...
                'Delimiter', '\n', ...
                'CollectOutput', true);

fclose(fid);

whos c
Name      Size            Bytes  Class    Attributes

  c         1x1               108  cell

c{1} =
   12.6700    3.1400    6.7780
   21.7800    5.2400    9.8380
```

Back to Top

Importing Large ASCII Data Sets

To import large data files, consider using `textscan` to read the file in segments, which reduces the amount of memory required.

For example, suppose you want to process the file `largefile.dat` with the user-defined `process_data` function. This example assumes that the `process_data` function

processes any number of lines of data, including zero.

```
clear segarray;
block_size = 10000;

% describe the format of the data
% for more information, see the  textscan reference page
format = '%s %n %s %8.2f %8.2f %8.2f %8.2f %u8';

file_id = fopen('largefile.dat');

while ~feof(file_id)
   segarray = textscan(file_id, format, block_size);
   process_data(segarray);
end

fclose(file_id);
```

The `fopen` function positions a pointer at the beginning of the file, and each read operation adjusts the location of that pointer. You can also use low-level file I/O functions such as `fseek` and `frewind` to reposition the pointer within the file. For more information, see Moving within a File.

Back to Top

Importing Text Data Files with Low-Level I/O

Low-level file I/O functions allow the most control over reading or writing data to a file. However, these functions require that you specify more detailed information about your file than the easier-to-use *high-level functions*, such as `importdata`. For more information on the high-level functions that read text files, see Importing Text Data Files.

If the high-level functions cannot import your data, use one of the following:

- `fscanf`, which reads formatted data in a text or ASCII file; that is, a file you can view in a text editor. For more information, see Reading Data in a Formatted Pattern.
- `fgetl` and `fgets`, which read one line of a file at a time, where a newline character separates each line. For more information, see Reading Data Line-by-Line.
- `fread`, which reads a stream of data at the byte or bit level. For more information, see Importing Binary Data with Low-Level I/O.

For additional information, see:

- Testing for End of File (EOF)
- Opening Files with Different Character Encodings

> **Note** The low-level file I/O functions are based on functions in the ANSI® Standard C Library. However, MATLAB includes *vectorized* versions of the functions, to read and write data in an array with minimal control loops.

Reading Data in a Formatted Pattern

To import text files that `importdata` and `textscan` cannot read, consider using `fscanf`. The `fscanf` function requires that you describe the format of your file, but includes many options for this format description.

For example, create a text file `mymeas.dat` as shown. The data in `mymeas.dat` includes repeated sets of times, dates, and measurements. The header text includes the number of sets of measurements, `N`:

```
Measurement Data
N=3

12:00:00
01-Jan-1977
4.21  6.55  6.78  6.55
9.15  0.35  7.57  NaN
7.92  8.49  7.43  7.06
9.59  9.33  3.92  0.31
09:10:02
23-Aug-1990
2.76  6.94  4.38  1.86
0.46  3.17  NaN   4.89
0.97  9.50  7.65  4.45
8.23  0.34  7.95  6.46
15:03:40
15-Apr-2003
7.09  6.55  9.59  7.51
7.54  1.62  3.40  2.55
NaN   1.19  5.85  5.05
6.79  4.98  2.23  6.99
```

Opening the File. As with any of the low-level I/O functions, before reading, open the file with `fopen`, and obtain a file identifier. By default, `fopen` opens files for read access, with a permission of `'r'`.

When you finish processing the file, close it with `fclose(fid)`.

Describing the Data. Describe the data in the file with format specifiers, such as `'%s'` for a string, `'%d'` for an integer, or `'%f'` for a floating-point number. (For a complete list of specifiers, see the `fscanf` reference page.)

To skip literal characters in the file, include them in the format description. To skip a data field, use an asterisk (`'*'`) in the specifier.

For example, consider the header lines of `mymeas.dat`:

```
Measurement Data   % skip 2 strings, go to next line:  %*s %*s\n
N=3                % ignore 'N=', read integer:  N=%d\n
                   % go to next line:  \n
12:00:00
01-Jan-1977
```

```
4.21   6.55   6.78   6.55
...
```

To read the headers and return the single value for `N`:

```
N = fscanf(fid, '%*s %*s\nN=%d\n\n', 1);
```

Specifying the Number of Values to Read. By default, `fscanf` reapplies your format description until it cannot match the description to the data, or it reaches the end of the file.

Optionally, specify the number of values to read, so that `fscanf` does not attempt to read the entire file. For example, in `mymeas.dat`, each set of measurements includes a fixed number of rows and columns:

```
measrows = 4;
meascols = 4;
meas  = fscanf(fid, '%f', [measrows, meascols])';
```

Creating Variables in the Workspace. There are several ways to store `mymeas.dat` in the MATLAB workspace. In this case, read the values into a structure. Each element of the structure has three fields: `mtime`, `mdate`, and `meas`.

> **Note** `fscanf` fills arrays with numeric values in column order. To make the output array match the orientation of numeric data in a file, transpose the array.

```
filename = 'mymeas.dat';
measrows = 4;
meascols = 4;

% open the file
fid = fopen(filename);

% read the file headers, find N (one value)
N = fscanf(fid, '%*s %*s\nN=%d\n\n', 1);

% read each set of measurements
for n = 1:N
    mystruct(n).mtime = fscanf(fid, '%s', 1);
    mystruct(n).mdate = fscanf(fid, '%s', 1);

    % fscanf fills the array in column order,
    % so transpose the results
    mystruct(n).meas  = ...
      fscanf(fid, '%f', [measrows, meascols])';
end

% close the file
fclose(fid);
```

Reading Data Line-by-Line

MATLAB provides two functions that read lines from files and store them in string vectors: `fgetl` and `fgets`. The `fgets` function copies the newline character to the output string, but `fgetl` does not.

The following example uses `fgetl` to read an entire file one line at a time. The function `litcount` determines whether an input literal string (`literal`) appears in each line. If it does, the function prints the entire line preceded by the number of times the literal string appears on the line.

```
function y = litcount(filename, literal)
% Search for number of string matches per line.

fid = fopen(filename);
y = 0;
tline = fgetl(fid);
while ischar(tline)
   matches = strfind(tline, literal);
   num = length(matches);
   if num > 0
      y = y + num;
      fprintf(1,'%d:%s\n',num,tline);
   end
   tline = fgetl(fid);
end
fclose(fid);
```

Create an input data file called `badpoem`:

```
Oranges and lemons,
Pineapples and tea.
Orangutans and monkeys,
Dragonflys or fleas.
```

To find out how many times the string `'an'` appears in this file, call `litcount`:

```
litcount('badpoem','an')
```

This returns:

```
2: Oranges and lemons,
1: Pineapples and tea.
3: Orangutans and monkeys,
ans =
     6
```

Testing for End of File (EOF)

When you read a portion of your data at a time, you can use `feof` to check whether you have reached the end of the file. `feof` returns a value of `1` when the file pointer is at the end of the file. Otherwise, it returns `0`.

> **Note** Opening an empty file does *not* move the file position indicator to the end of the file. Read operations, and the `fseek` and `frewind` functions, move the file position indicator.

Testing for EOF with feof. When you use `textscan`, `fscanf`, or `fread` to read portions of data at a time, use `feof` to check whether you have reached the end of the file.

For example, suppose that the hypothetical file `mymeas.dat` has the following form, with no information about the number of measurement sets. Read the data into a structure with fields for `mtime`, `mdate`, and `meas`:

```
12:00:00
01-Jan-1977
4.21  6.55  6.78  6.55
9.15  0.35  7.57  NaN
7.92  8.49  7.43  7.06
9.59  9.33  3.92  0.31
09:10:02
23-Aug-1990
2.76  6.94  4.38  1.86
0.46  3.17  NaN   4.89
0.97  9.50  7.65  4.45
8.23  0.34  7.95  6.46
```

To read the file:

```
filename = 'mymeas.dat';
measrows = 4;
meascols = 4;

% open the file
fid = fopen(filename);

% make sure the file is not empty
finfo = dir(filename);
fsize = finfo.bytes;

if fsize > 0

    % read the file
    block = 1;
    while ~feof(fid)
        mystruct(block).mtime = fscanf(fid, '%s', 1);
        mystruct(block).mdate = fscanf(fid, '%s', 1);

        % fscanf fills the array in column order,
        % so transpose the results
        mystruct(block).meas  = ...
          fscanf(fid, '%f', [measrows, meascols])';
```

```
        block = block + 1;
    end

end

% close the file
fclose(fid);
```

Testing for EOF with fgetl and fgets. If you use `fgetl` or `fgets` in a control loop, `feof` is not always the best way to test for end of file. As an alternative, consider checking whether the value that `fgetl` or `fgets` returns is a character string.

For example, the function `litcount` described in Reading Data Line-by-Line includes the following `while` loop and `fgetl` calls :

```
y = 0;
tline = fgetl(fid);
while ischar(tline)
   matches = strfind(tline, literal);
   num = length(matches);
   if num > 0
      y = y + num;
      fprintf(1,'%d:%s\n',num,tline);
   end
   tline = fgetl(fid);
end
```

This approach is more robust than testing `~feof(fid)` for two reasons:

- If `fgetl` or `fgets` find data, they return a string. Otherwise, they return a number (`-1`).
- After each read operation, `fgetl` and `fgets` check the next character in the file for the end-of-file marker. Therefore, these functions sometimes set the end-of-file indicator *before* they return a value of `-1`. For example, consider the following three-line text file. Each of the first two lines ends with a newline character, and the third line contains only the end-of-file marker:

  ```
  123
  456
  ```

  Three sequential calls to `fgetl` yield the following results:

  ```
  t1 = fgetl(fid);    % t1 = '123', feof(fid) = false
  t2 = fgetl(fid);    % t2 = '456', feof(fid) = true
  t3 = fgetl(fid);    % t3 = -1,    feof(fid) = true
  ```

  This behavior does not conform to the ANSI specifications for the related C language functions.

Opening Files with Different Character Encodings

Encoding schemes support the characters required for particular alphabets, such as those for Japanese or European languages. Common encoding schemes include US-ASCII or UTF-8.

If you do not specify an encoding scheme, `fopen` opens files for processing using the default encoding for your system. To determine the default, open a file, and call `fopen` again with the syntax:

```
[filename, permission, machineformat, encoding] = fopen(fid);
```

If you specify an encoding scheme when you open a file, the following functions apply that scheme: `fscanf`, `fprintf`, `fgetl`, `fgets`, `fread`, and `fwrite`.

For a complete list of supported encoding schemes, and the syntax for specifying the encoding, see the `fopen` reference page.

Back to Top

Importing MAT-Files Importing XML Documents