MATLAB Function Reference

textscan

Read data from text file, convert, and write to cell array

Syntax

```
C = textscan(fid, 'format')
C = textscan(fid, 'format', N)
C = textscan(fid, 'format', param, value, ...)
C = textscan(fid, 'format', N, param, value, ...)
C = textscan(str, ...)
[C, position] = textscan(...)
```

Description

Before reading a file with `textscan`, you must open the file with the `fopen` function. `fopen` supplies the `fid` input required by `textscan`. When you are finished reading from the file, you should close the file by calling `fclose(fid)`.

`C = textscan(fid, 'format')` reads data from an open text file identified by file identifier `fid` into cell array `C`. MATLAB parses the data into fields and converts it according to the conversion specifiers in the `format` string. These conversion specifiers determine the type of each cell in the output cell array. The number of specifiers determines the number of cells in the cell array.

`C = textscan(fid, 'format', N)` reads data from the file, reusing the `format` conversion specifier `N` times, where `N` is a positive integer. You can resume reading from the file after `N` cycles by calling `textscan` again using the original `fid`.

`C = textscan(fid, 'format', param, value, ...)` reads data from the file using nondefault parameter settings specified by one or more pairs of `param` and `value` arguments. The section User Configurable Options lists all valid parameter strings, value descriptions, and defaults.

`C = textscan(fid, 'format', N, param, value, ...)` reads data from the file, reusing the `format` conversion specifier `N` times, and using nondefault parameter settings specified by pairs of `param` and `value` arguments.

`C = textscan(str, ...)` reads data from string `str` in exactly the same way as it does when reading from a file. You can use the `format`, `N`, and parameter/value arguments described above with this syntax. Unlike when reading from a file, if you call `textscan` more than once on the same string, it does not resume reading where the last call left off but instead reads from the beginning of the string each time.

`[C, position] = textscan(...)` returns the location of the file or string position as the second output argument. For a file, this is exactly equivalent to calling `ftell(fid)` after making the call to `TEXTSCAN`. For a string, it indicates how many characters were read.

The Difference Between the textscan and textread Functions

The textscan function differs from textread in the following ways:

- The textscan function offers better performance than textread, making it a better choice when reading large files.
- With textscan, you can start reading at any point in the file. Once the file is open, (textscan requires that you open the file first), you can seek to any position in the file and begin the textscan at that point. The textread function requires that you start reading from the beginning of the file.
- Subsequent textscans start reading the file at the point where the last textscan left off. The textread function always begins at the start of the file, regardless of any prior textread.
- textscan returns a single cell array regardless of how many fields you read. With textscan, you don't need to match the number of output arguments to the number of fields being read as you would with textread.
- textscan offers more choices in how the data being read is converted.
- textscan offers more user-configurable options.

Field Delimiters

The textscan function regards a text file as consisting of blocks. Each block consists of a number of internally consistent fields. Each field consists of a group of characters delimited by a field delimiter character. Fields can span a number of rows. Each row is delimited by an end-of-line (EOL) character sequence.

The default field delimiter is the white-space character, (i.e., any character that returns true from a call to the isspace function). You can set the delimiter to a different character by specifying a 'delimiter' parameter in the textscan command (see User Configurable Options). If a nondefault delimiter is specified, repeated delimiter characters are treated as separate delimiters. When using the default delimiter, repeated white-space characters are treated as a single delimiter.

The default end-of-line character sequence depends on which operating system you are using. You can set end-of-line to a different character sequence by specifying an 'endofline' parameter in the textscan command (see User Configurable Options). If you set the delimiter parameter to 'EOL' (using the third syntax shown above), textscan reads complete rows.

Conversion Specifiers

This table shows the conversion type specifiers supported by textscan.

Specifier	Description
%n	Read a number and convert to double.
%d	Read a number and convert to int32.

`%d8`	Read a number and convert to `int8`.
`%d16`	Read a number and convert to `int16`.
`%d32`	Read a number and convert to `int32`.
`%d64`	Read a number and convert to `int64`.
`%u`	Read a number and convert to `uint32`.
`%u8`	Read a number and convert to `uint8`.
`%u16`	Read a number and convert to `uint16`.
`%u32`	Read a number and convert to `uint32`.
`%u64`	Read a number and convert to `uint64`.
`%f`	Read a number and convert to `double`.
`%f32`	Read a number and convert to `single`.
`%f64`	Read a number and convert to `double`.
`%s`	Read a string.
`%q`	Read a (possibly double-quoted) string.
`%c`	Read one character, including white space.
`%[...]`	Read characters that match characters between the brackets. Stop reading at the first nonmatching character or white-space. Use `%[]...]` to include `]` in the set.
`%[^...]`	Read characters that do not match characters between the brackets. Stop reading at the first matching character or white-space. Use `%[^]...]` to exclude `]` from the set.

Specifying Field Length

To read a certain number of characters or digits from a field, specify that number directly following the percent sign. For example, if the file you are reading contains the string

```
'Blackbird singing in the dead of night'
```

then the following command returns only five characters of the first field:

```
C = textscan(fid, '%5s', 1);
C{:}
ans =
   'Black'
```

If you continue reading from the file, `textscan` resumes the operation at the point in the string where you left off. It applies the next format specifier to that portion of the field. For example, execute this command on the same file:

```
C = textscan(fid, '%s %s', 1);
```

> **Note** Spaces between the conversion specifiers are shown only to make the example easier to read. They are not required.

`textscan` reads starting from where it left off and continues to the next whitespace, returning `'bird'`. The second `%s` reads the word `'singing'`.

The results are

```
C{:}
ans =
    'bird'
ans =
    'singing'
```

Skipping Fields

To skip any field, put an asterisk directly after the percent sign. MATLAB does not create an output cell for any fields that are skipped.

Refer to the example from the last section, where the file you are reading contains the string

```
'Blackbird singing in the dead of night'
```

Seek to the beginning of the file and then reread the line, this time skipping the second, fifth, and sixth fields:

```
fseek(fid, 0, -1);
C = textscan(fid, '%s %*s %s %s %*s %*s %s', 1);
```

C is a cell array of cell arrays, each containing a string. Piece together the string and display it:

```
str = '';
for k = 1:length(C)
   str = [str char(C{k}) ' '];
   if k == 4,  disp(str),  end
end

Blackbird in the night
```

Skipping Literal Strings

In addition to skipping entire fields, you can have `textscan` skip leading literal characters in a string. Reading a file containing the following data,

```
Sally    Level1  12.34
Joe      Level2  23.54
Bill     Level3  34.90
```

this command removes the substring `'Level'` from the output and converts the level number to a `uint8`:

```
C = textscan(fid, '%s Level%u8 %f');
```

This returns a cell array `C` with the second cell containing only the unsigned

integers:

```
C{1} = {'Sally'; 'Joe'; 'Bill'}       class cell
C{2} = [1; 2; 3]                      class uint8
C{3} = [12.34; 23.54; 34.90]          class double
```

Specifying Numeric Field Length and Decimal Digits

With numeric fields, you can specify the number of digits to read in the same manner described for strings in the section Specifying Field Length. The next example uses a file containing the line

```
'405.36801 551.94387 298.00752 141.90663'
```

This command returns the starting 7 digits of each number in the line. Note that the decimal point counts as a digit.

```
C = textscan(fid, '%7f32 %*n');
C{:} =
   [405.368; 551.943; 298.007; 141.906]
```

You can also control the number of digits that are read to the right of the decimal point for any numeric field of type `%f`, `%f32`, or `%f64`. The format specifier in this command uses a `%9.1` prefix to cause `textscan` to read the first 9 digits of each number, but only include 1 digit of the decimal value in the number it returns:

```
C = textscan(fid, '%9.1f32 %*n');
C{:} =
   [405.3; 551.9; 298.0; 141.9]
```

Conversion of Numeric Fields

This table shows how `textscan` interprets the numeric field specifiers.

Format Specifier	Action Taken
`%n`, `%d`, `%u`, `%f`, and variants thereof	Read to the first delimiter. Example: `%n` reads `'473.238 '` as `473.238`.
`%Nn`, `%Nd`, `%Nu`, `%Nf`, and variants thereof	Read `N` digits (counting a decimal point as a digit), or up to the first delimiter, whichever comes first. Example: `%5f32` reads `'473.238 '` as `473.2`.
Specifiers that start with `%N.Df`	Read `N` digits (counting a decimal point as a digit), or up to the first delimiter, whichever comes first. Return `D` decimal digits in the output. Example: `%7.2f` reads `'473.238 '` as `473.23`.

Conversion specifiers `%n`, `%d`, `%u`, `%f`, or any variant thereof (e.g., `%d16`) return a `K`-by-1 MATLAB numeric vector of the type indicated by the conversion specifier, where `K` is the number of times that specifier was found in the file. `textscan` converts the numeric fields from the field content to the output type according to the conversion specifier and MATLAB rules regarding overflow

and truncation. `NaN`, `Inf`, and `-Inf` are converted according to applicable MATLAB rules.

`textscan` imports any complex number as a whole into a complex numeric field, converting the real and imaginary parts to the specified numeric type. Valid forms for a complex number are

Form	Example
`±<real>±<imag>i|j`	`5.7-3.1i`
`±<imag>i|j`	`-7j`

Embedded white-space in a complex number is invalid and is regarded as a field delimiter.

Conversion of Strings

This table shows how `textscan` interprets the string field specifiers.

Format Specifier	Action Taken
`%s` or `%q`	Read to the first delimiter. Example: %s reads 'summer ' as 'summer'.
`%Ns` or `%Nq`	Read N characters, or to the first delimiter, whichever comes first. Example: %3s reads 'summer ' as 'sum'.
`%[abc]`	Read up to the first character not specified within the brackets (i.e., read up to the first character that is not an `a`, `b`, or `c`). Example: `%[mus]` reads `'summer '` as `'summ'`.
`%N[abc]`	Read N characters, or up to the first character not specified within the brackets, whichever comes first. Example: %2[mus] reads 'summer ' as 'su'.
`%[^abc]`	Read up to the first character that is specified within the brackets, (i.e., read up to the first occurrence of an `a`, `b`, or `c`). Example: `%[^xrg]` reads `'summer '` as `'summe'`.
`%N[^abc]`	Read `N` characters, or up to the first character that is specified within the brackets, whichever comes first. Example: `%2[^xrg]` reads `'summer '` as `'su'`.

Conversion specifiers `%s`, `%q`, `%[...]`, and `%[^...]` return a `K`-by-1 MATLAB cell vector of strings, where `K` is the number of times that specifier was found in the file. If you set the `delimiter` parameter to a non-white-space character, or set the `whitespace` parameter to `''`, `textscan` returns all characters in the string field, including white-space. Otherwise each string terminates at the beginning of white-space.

Conversion of Characters

This table shows how `textscan` interprets the character field specifiers.

Format Specifier	Action Taken
`%c`	Read one character. Example: `%c` reads `'Let's go!'` as `'L'`.
`%Nc`	Read `N` characters, including delimiter characters. Example: `%9c` reads `'Let's go!'` as `'Let's go!'`.

Conversion specifier `%Nc` returns a `K`-by-`N` MATLAB character array, where `K` is the number of times that specifier was found in the file. `textscan` returns all characters, including white-space but excluding the delimiter.

Conversion of Empty Fields

An empty field in the text file is defined by two adjacent delimiters indicating an empty set of characters, or, in all cases except `%c`, white-space. The empty field is returned as `NaN` by default, but is user definable. In addition, you may specify custom strings to be used as empty values, in *numeric fields only*. `textscan` does not examine nonnumeric fields for custom empty values. See User Configurable Options.

> **Note** MATLAB represents integer `NaN` as zero. If `textscan` reads an empty field that is assigned an integer format specifier (one that starts with `%d` or `%u`), it returns the empty value as zero rather than as `NaN`. (See the value returned in C{5} in "Example 6 -- Using a Nondefault Empty Value".

User Configurable Options

This table shows the valid `param-value` options and their default values.

Parameter	Value	Default
`bufSize`	Maximum string length in bytes	`4095`
`commentStyle`	Symbol(s) designating text to be ignored (see Values for commentStyle, below)	None
`delimiter`	Delimiter characters	None
`emptyValue`	Empty cell value in delimited files	`NaN`
`endOfLine`	End-of-line character	Determined from the file
`expChars`	Exponent characters	`'eEdD'`
`headerLines`	Number of lines at beginning of file to skip	0

`multipleDelimsAsOne`	If set to 1, `textread` treats consecutive delimiters as a single delimiter. If set to 0, `textread` treats them as separate delimiters. Only valid if the `delimiter` option is specified.	`0`
`returnOnError`	Behavior on failing to read or convert (1=`true` or 0)	`1`
`treatAsEmpty`	String(s) to be treated as an empty value. A single string or cell array of strings can be used.	None
`whitespace`	White-space characters	`' \b\t'`

Values for commentStyle

Possible values for the `commentStyle` parameter are

Value	Description	Example
Single string, `s`	Ignore any characters that follow string `s` and are on the same line.	`'%', '//'`
Cell array of two strings, `c`	Ignore any characters that lie between the opening and closing strings in `c`.	`{'/*', '*/'}, {'/%', '%/'}`

Resuming a Text Scan

If `textscan` fails to convert a data field, it stops reading and returns all fields read before the failure. When reading from a file, you can resume reading from the same file by calling `textscan` again using the same file identifier, `fid`. When reading from a string, the two-output argument syntax enables you to resume reading from the string at the point where the last read terminated. The following command is an example of how you can do this:

```
textscan(str(position+1:end), ...)
```

Remarks

For information on how to use `textscan` to import large data sets, see Large Data Sets in the MATLAB Programming documentation.

Examples

Example 1-- Reading Different Types of Data

Text file `scan1.dat` contains data in the following form:

```
Sally  Level1 12.34 45 1.23e10 inf NaN Yes
Joe    Level2 23.54 60 9e19 -inf 0.001 No
```

```
Bill   Level3 34.90 12 2e5 10 100 No
```

Read each column into a variable:

```
fid = fopen('scan1.dat');
C = textscan(fid, '%s %s %f32 %d8 %u %f %f %s');
fclose(fid);
```

> **Note** Spaces between the conversion specifiers are shown only to make the example easier to read. They are not required.

`textscan` returns a 1-by-8 cell array `C` with the following cells:

```
C{1} = {'Sally'; 'Joe'; 'Bill'}          class cell
C{2} = {'Level1'; 'Level2'; 'Level3'}    class cell
C{3} = [12.34; 23.54; 34.9]              class single
C{4} = [45; 60; 12]                      class int8
C{5} = [4294967295; 4294967295; 200000]  class uint32
C{6} = [Inf; -Inf; 10]                   class double
C{7} = [NaN; 0.001; 100]                 class double
C{8} = {'Yes'; 'No'; 'No'}               class cell
```

The first two elements of `C{5}` are the maximum values for a 32-bit unsigned integer, or `intmax('uint32')`.

Example 2 -- Reading All But One Field

Read the file as a fixed-format file, skipping the third field:

```
fid = fopen('scan1.dat');
C = textscan(fid, '%7c %6s %*f %d8 %u %f %f %s');
fclose(fid);
```

`textscan` returns a 1-by-8 cell array `C` with the following cells:

```
C{1} = ['Sally  '; 'Joe    '; 'Bill   ']   class char
C{2} = {'Level1'; 'Level2'; 'Level3'}      class cell
C{3} = [45; 60; 12]                        class int8
C{4} = [4294967295; 4294967295; 200000]    class uint32
C{5} = [Inf; -Inf; 10]                     class double
C{6} = [NaN; 0.001; 100]                   class double
C{7} = {'Yes'; 'No'; 'No'}                 class cell
```

Example 3 -- Reading Only the First Field

Read the first column into a cell array, skipping the rest of the line:

```
fid = fopen('scan1.dat');
names = textscan(fid, '%s%*[^\n]');
fclose(fid);
```

`textscan` returns a 1-by-1 cell array `names`:

```
size(names)
ans =
     1     1
```

The one cell contains

```
names{1} = {'Sally'; 'Joe'; 'Bill'}       class cell
```

Example 4 -- Removing a Literal String in the Output

The second `format` specifier in this example, `%sLevel`, tells `textscan` to read the second field from a line in the file, but to ignore the initial string `'Level'` within that field. All that is left of the field is a numeric digit. `textscan` assigns the next specifier, `%f`, to that digit, converting it to a `double`.

See `C{2}` in the results:

```
fid = fopen('scan1.dat');
C = textscan(fid, '%s Level%u8 %f32 %d8 %u %f %f %s');
fclose(fid);
```

`textscan` returns a 1-by-8 cell array, `C`, with cells

```
C{1} = {'Sally'; 'Joe'; 'Bill'}          class cell
C{2} = [1; 2; 3]                         class uint8
C{3} = [12.34; 23.54; 34.90]             class single
C{4} = [45; 60; 12]                      class int8
C{5} = [4294967295; 4294967295; 200000]  class uint32
C{6} = [Inf; -Inf; 10]                   class double
C{7} = [NaN; 0.001; 100]                 class double
C{8} = {'Yes'; 'No'; 'No'}               class cell
```

Example 5 -- Using a Nondefault Delimiter and White-Space

Read the M-file into a cell array of strings:

```
fid = fopen('fft.m');
file = textscan(fid, '%s', 'delimiter', '\n', 'whitespace', '');
fclose(fid);
```

`textscan` returns a 1-by-1 cell array, `file`, that contains a 37-by-1 cell array:

```
file =
    {37x1 cell}
```

Show the first three lines of the file:

```
lines = file{1};
lines{1:3, :}
ans =
    'function [varargout] = fft(varargin)'
ans =
    '%FFT Discrete Fourier transform.'
ans =
    '%   FFT(X) is the discrete Fourier transform (DFT) of vector
X.  For'
```

Example 6 -- Using a Nondefault Empty Value

Read files with empty cells, setting the `emptyvalue` parameter. The file `data.csv` contains

```
1,  2,  3,  4,   ,  6
7,  8,  9,   , 11, 12
```

Read the file as shown here, using `-Inf` in empty cells:

```
fid = fopen('data.csv');
C = textscan(fid, '%f%f%f%f%u32%f', 'delimiter', ',', ...
             'emptyValue', -Inf);
fclose(fid);
```

`textscan` returns a 1-by-6 cell array `C` with the following cells:

```
C{1} = [1; 7]            class double
C{2} = [2; 8]            class double
C{3} = [3; 9]            class double
C{4} = [4; NaN]          class double
C{5} = [-Inf; 11]        class uint32 (-Inf converted to 0)
C{6} = [6; 12]           class double
```

Example 7 -- Using Custom Empty Values and Comments

You have a file `data.csv` that contains the lines

```
abc, 2, NA, 3, 4
// Comment Here
def, na, 5, 6, 7
```

Designate what should be treated as empty values and as comments. Read in all other values from the file:

```
fid = fopen('data5.csv');
C = textscan(fid, '%s%n%n%n%n', 'delimiter', ',', ...
             'treatAsEmpty', {'NA', 'na'}, ...
             'commentStyle', '//');
fclose(fid);
```

This returns the following data in cell array `C`:

```
C{:}
ans =
    'abc'
    'def'
ans =
     2
   NaN
ans =
   NaN
     5
ans =
     3
     6
ans =
     4
     7
```

Example 8 -- Reading From a String

Read in a string (quoted from Albert Einstein) using `textscan`:

```
str = ...
  ['Do not worry about your difficulties in Mathematics. ' ...
   'I can assure you mine are still greater.'];

s = textscan(str, '%s', 'delimiter', '.');
```

```
s{:}
ans =
    'Do not worry about your difficulties in Mathematics'
    'I can assure you mine are still greater'
```

Example 9 -- Handling Multiple Delimiters

This example takes a comma-separated list of names, the test pilots known as the Mercury Seven, and uses `textscan` to return a list of their names in a cell array. When some names are removed from the input list, leaving multiple sequential delimiters, `textscan`, by default, accounts for this. If you override that default by calling `textscan` with the `multipleDelimsAsOne` option, `textscan` ignores the missing names.

Here is the full list of the astronauts:

```
Mercury7 = ...
   'Shepard,Grissom,Glenn,Carpenter,Schirra,Cooper,Slayton';
```

Remove the names Grissom and Cooper from the input string, and `textscan`, by default, does not treat the multiple delimiters as one, and returns an empty string for each missing name:

```
Mercury7 = 'Shepard,,Glenn,Carpenter,Schirra,,Slayton';
names = textscan(Mercury7, '%s', 'delimiter', ',');
names{:}'
ans =
  'Shepard'  ''  'Glenn'  'Carpenter'  'Schirra'  ''  'Slayton'
```

Using the same input string, but this time setting the `multipleDelimsAsOne` switch, `textscan` ignores the multiple delimiters:

```
names = textscan(Mercury7, '%s', 'delimiter', ',', ...
                 'multipledelimsasone', 1);
names{:}'
ans =
  'Shepard'  'Glenn'  'Carpenter'  'Schirra'  'Slayton'
```

See Also

dlmread, dlmwrite, xlswrite, fopen, importdata

textread textwrap

© 1994-2005 The MathWorks, Inc.