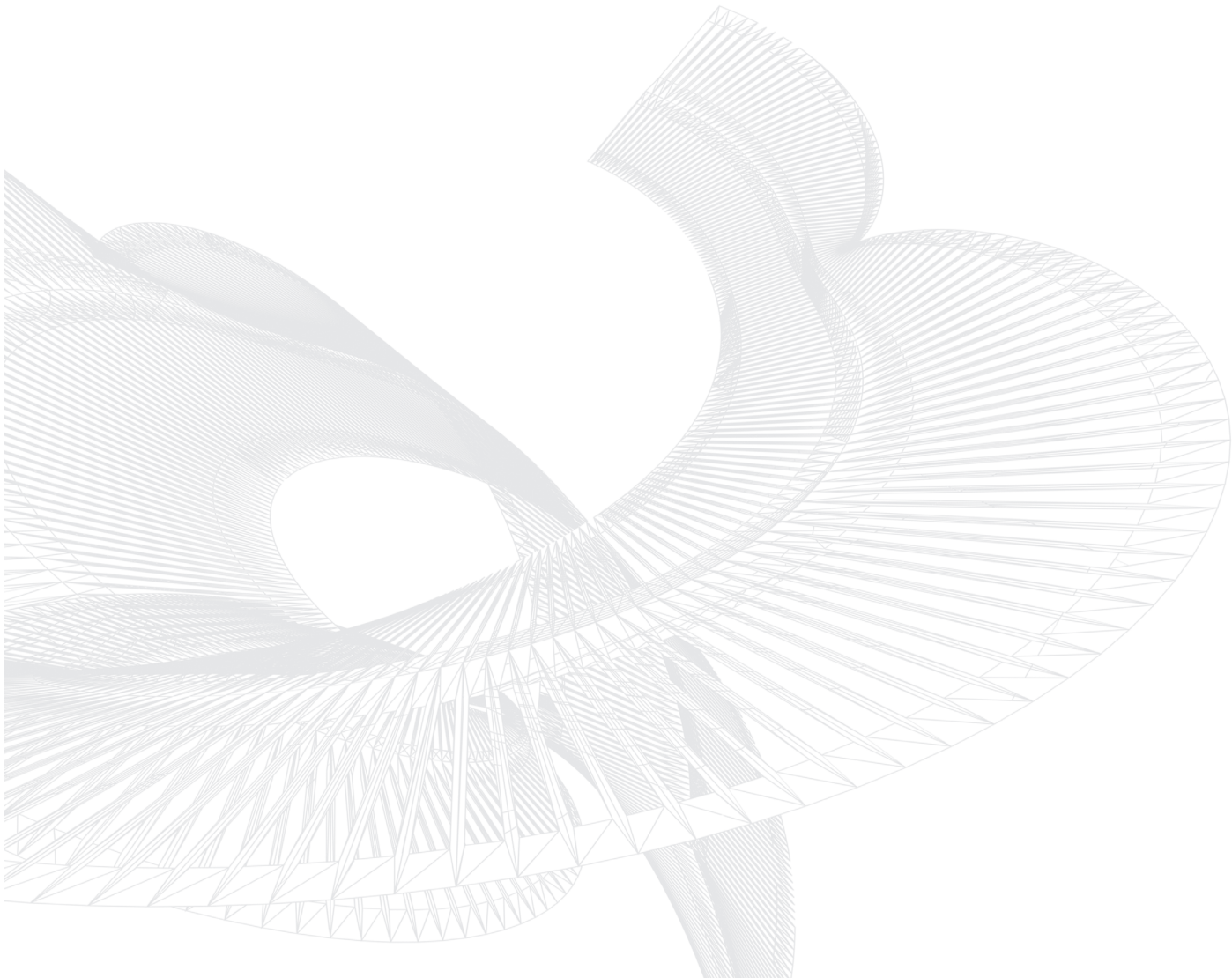


Wolfram *Mathematica*® Tutorial Collection

# DATA MANIPULATION



For use with Wolfram *Mathematica*<sup>®</sup> 7.0 and later.

**For the latest updates and corrections to this manual:**

visit [reference.wolfram.com](http://reference.wolfram.com)

**For information on additional copies of this documentation:**

visit the Customer Service website at [www.wolfram.com/services/customerservice](http://www.wolfram.com/services/customerservice)  
or email Customer Service at [info@wolfram.com](mailto:info@wolfram.com)

**Comments on this manual are welcomed at:**

[comments@wolfram.com](mailto:comments@wolfram.com)

Printed in the United States of America.

15 14 13 12 11 10 9 8 7 6 5 4 3 2

---

©2008 Wolfram Research, Inc.

All rights reserved. No part of this document may be reproduced or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording or otherwise, without the prior written permission of the copyright holder.

Wolfram Research is the holder of the copyright to the Wolfram *Mathematica* software system ("Software") described in this document, including without limitation such aspects of the system as its code, structure, sequence, organization, "look and feel," programming language, and compilation of command names. Use of the Software unless pursuant to the terms of a license granted by Wolfram Research or as otherwise authorized by law is an infringement of the copyright.

**Wolfram Research, Inc. and Wolfram Media, Inc. ("Wolfram") make no representations, express, statutory, or implied, with respect to the Software (or any aspect thereof), including, without limitation, any implied warranties of merchantability, interoperability, or fitness for a particular purpose, all of which are expressly disclaimed. Wolfram does not warrant that the functions of the Software will meet your requirements or that the operation of the Software will be uninterrupted or error free. As such, Wolfram does not recommend the use of the software described in this document for applications in which errors or omissions could threaten life, injury or significant loss.**

*Mathematica*, *MathLink*, and *MathSource* are registered trademarks of Wolfram Research, Inc. *J/Link*, *MathLM*, *.NET/Link*, and *webMathematica* are trademarks of Wolfram Research, Inc. Windows is a registered trademark of Microsoft Corporation in the United States and other countries. Macintosh is a registered trademark of Apple Computer, Inc. All other trademarks used herein are the property of their respective owners. *Mathematica* is not associated with Mathematica Policy Research, Inc.

# Contents

---

## Files, Streams, and External Operations

Reading and Writing <i>Mathematica</i> Files .....	1
External Programs .....	8
Streams and Low-Level Input and Output .....	12
Naming and Finding Files .....	18
Files for Packages .....	26
Manipulating Files and Directories .....	27
Reading Textual Data .....	28
Searching Files .....	36
Searching and Reading Strings .....	41
Binary Files .....	44
Generating C and Fortran Expressions .....	47
Splicing <i>Mathematica</i> Output into External Files .....	48

## Importing and Exporting

Importing and Exporting Data .....	50
Importing and Exporting Files .....	51
Exporting Graphics and Sounds .....	54
Generating and Importing TeX .....	57
Exchanging Material with the Web .....	58

## Image Processing

Image Creation and Representation .....	60
Basic Image Manipulation .....	63
Image Processing by Point Operations .....	66
Image Processing by Area Operations .....	71



# Files, Streams, and External Operations

## Reading and Writing *Mathematica* Files

### *Storing Mathematica Expressions in External Files*

You can use files on your computer system to store definitions and results from *Mathematica*. The most general approach is to store everything as plain text that is appropriate for input to *Mathematica*. With this approach, a version of *Mathematica* running on one computer system produces files that can be read by a version running on any computer system. In addition, such files can be manipulated by other standard programs, such as text editors.

<code>&lt;&lt;file</code> or <code>Get["file"]</code>	read in a file of <i>Mathematica</i> input, and return the last expression in the file
<code>FilePrint["file"]</code>	display the contents of a file
<code>expr&gt;&gt;file</code> or <code>Put[expr,"file"]</code>	write an expression to a file
<code>expr&gt;&gt;&gt;file</code> or <code>PutAppend[expr,"file"]</code>	append an expression to a file

Reading and writing files.

This expands  $(x + y)^3$ , and outputs the result to a file called `tmp`.

```
In[1]:= Expand[(x + y)^3] >> tmp
```

Here are the contents of `tmp`. They can be used directly as input for *Mathematica*.

```
In[2]:= FilePrint["tmp"]
```

```
x^3 + 3*x^2*y + 3*x*y^2 + y^3
```

This reads in `tmp`, evaluating the *Mathematica* input it contains.

```
In[3]:= << tmp
```

```
Out[3]= x^3 + 3 x^2 y + 3 x y^2 + y^3
```

This shows the contents of the file `factors`.

```
In[1]:= FilePrint["ExampleData/factors"]

(* Factors of x^20 - 1 *)
(-1 + x)*(1 + x)*(1 + x^2)*(1 - x + x^2 - x^3 + x^4)*
(1 + x + x^2 + x^3 + x^4)*(1 - x^2 + x^4 - x^6 + x^8)
```

This reads in the file, and returns the last expression in it.

```
In[2]:= << ExampleData/factors
Out[2]= (-1 + x) (1 + x) (1 + x^2) (1 - x + x^2 - x^3 + x^4) (1 + x + x^2 + x^3 + x^4) (1 - x^2 + x^4 - x^6 + x^8)
```

If *Mathematica* cannot find the file you ask it to read, it prints a message, then returns the symbol `$Failed`.

```
In[19]:= << factors

Get::noopen: Cannot open factors. >>

Out[19]= $Failed
```

When you read in a file with `<< file`, *Mathematica* returns the last expression it evaluates in the file. You can avoid getting any visible result from reading a file by ending the last expression in the file with a semicolon, or by explicitly adding `Null` after that expression.

If *Mathematica* encounters a syntax error while reading a file, it reports the error, skips the remainder of the file, then returns `$Failed`. If the syntax error occurs in the middle of a package which uses `BeginPackage` and other context manipulation functions, then *Mathematica* tries to restore the context to what it was before the package was read.

## ***Saving Multiple Mathematica Expressions***

*Mathematica* input files can contain any number of expressions. Each expression, however, must start on a new line. The expressions may continue for as many lines as necessary. Just as in a standard interactive *Mathematica* session, the expressions are processed as soon as they are complete. Note, that in a file, unlike an interactive session, you can insert a blank line at any point without effect.

When you use `expr >>> file`, *Mathematica* appends each new expression you give to the end of your file. If you use `expr >> file`, however, then *Mathematica* instead wipes out anything that was in the file before, and then puts `expr` into the file.

This writes an expression to the file `tmp`.

```
In[4]:= Factor[x^6 - 1] >> tmp
```

Here are the contents of the file.

```
In[5]:= FilePrint["tmp"]
(-1 + x)*(1 + x)*(1 - x + x^2)*(1 + x + x^2)
```

This appends another expression to the same file.

```
In[6]:= Factor[x^8 - 1] >>> tmp
```

Both expressions are now in the file.

```
In[7]:= FilePrint["tmp"]
(-1 + x)*(1 + x)*(1 - x + x^2)*(1 + x + x^2)
(-1 + x)*(1 + x)*(1 + x^2)*(1 + x^4)
```

If you are familiar with command-line operating systems, you will recognize the *Mathematica* redirection operators `>>`, `>>>` and `<<` as being analogous to the command-line operators `>`, `>>` and `<`.

## ***Saving Mathematica Expressions in Different Formats***

When you use either `>>` or `>>>` to write expressions to files, the expressions are usually given in *Mathematica* input format, so that you can read them back into *Mathematica*. Sometimes, however, you may want to save expressions in other formats. You can do this by explicitly wrapping a format directive such as `OutputForm` around the expression you write out.

This writes an expression to the file `tmp` in output format.

```
In[8]:= OutputForm[Factor[x^6 - 1]] >> tmp
```

The expression in `tmp` is now in output format.

```
In[9]:= FilePrint["tmp"]
(-1 + x) (1 + x) (1 - x + x2) (1 + x + x2)
```

## Saving Definitions of Mathematica Objects

One of the most common reasons for using files is to save definitions of *Mathematica* objects, to be able to read them in again in a subsequent *Mathematica* session. The operators `>>` and `>>>` allow you to save *Mathematica* expressions in files. You can use the function `save` to save complete definitions of *Mathematica* objects, in a form suitable for execution in subsequent *Mathematica* sessions.

<code>Save["file", symbol]</code>	save the complete definitions for a symbol in a file
<code>Save["file", "form"]</code>	save definitions for symbols whose names match the string pattern <i>form</i>
<code>Save["file", "context`"]</code>	save definitions for all symbols in the specified context
<code>Save["file", {object<sub>1</sub>, object<sub>2</sub>, ...}]</code>	save definitions for several objects

Saving definitions in plain text files.

This assigns a value to the symbol `a`.

```
In[51]:= a = 2 - x^2
```

```
Out[51]= 2 - x^2
```

You can use `Save` to write the definition of `a` to a file.

```
In[52]:= Save["afile", a]
```

Here is the definition of `a` that was saved in the file.

```
In[53]:= FilePrint["afile"]
```

```
a = 2 - x^2
```

This defines a function `f` which depends on the symbol `a` previously defined.

```
In[54]:= f[z_] := a^2 - 2
```

This saves the complete definition of `f` in a file.

```
In[55]:= Save["ffile", f]
```



The file contains not only the definition of  $f$  itself, but also the definition of the symbol  $a$  on which  $f$  depends.

```
In[56]:= FilePrint["ffile"]

      f[z_] := a^2 - 2

      a = 2 - x^2
```

This clears the definitions of  $f$  and  $a$ .

```
In[57]:= Clear[f, a]
```

You can reinstate the definitions you saved simply by reading in the file `ffile`.

```
In[58]:= << ffile

Out[58]= 2 - x^2
```

The function `save` makes use of the output forms `Definition` and `FullDefinition`, which print as definitions of *Mathematica* symbols. In some cases, you may find it convenient to use these output forms directly.

The output form `Definition[f]` prints as the sequence of definitions that have been made for  $f$ .

```
In[59]:= Definition[f]

Out[59]= f[z_] := a^2 - 2
```

`FullDefinition[f]` includes definitions of the objects on which  $f$  depends.

```
In[60]:= FullDefinition[f]

      f[z_] := a^2 - 2

Out[60]= a = 2 - x^2
```

When you define a new object in *Mathematica*, your definition will often depend on other objects that you defined before. If you are going to be able to reconstruct the definition of your new object in a subsequent *Mathematica* session, it is important that you store not only its own definition, but also the definitions of other objects on which it depends. The function `save` looks

through the definitions of the objects you ask it to save, and automatically also saves all definitions of other objects on which it can see that these depend. However, in order to avoid saving a large amount of unnecessary material, `save` never includes definitions for symbols that have the attribute `Protected`. It assumes that the definitions for these symbols are also built in. Nevertheless, with such definitions taken care of, it should always be the case that reading the output generated by `save` back into a new *Mathematica* session will set up the definitions of your objects exactly as you had them before.

## ***Saving Mathematica Definitions in Encoded Form***

When you create files for input to *Mathematica*, you usually want them to contain only “plain text”, which can be read or modified directly. Sometimes, however, you may want the contents of a file to be “encoded” so that they cannot be read or modified directly as plain text, but can be loaded into *Mathematica*. You can create encoded files using the *Mathematica* function `Encode`.

<code>Encode [ "source" , "dest" ]</code>	write an encoded version of the file <i>source</i> to the file <i>dest</i>
<code>&lt;&lt;dest</code>	read in an encoded file
<code>Encode [ "source" , "dest" , "key" ]</code>	encode with the specified key
<code>Get [ "dest" , "key" ]</code>	read in a file that was encoded with a key
<code>Encode [ "source" , "dest" , MachineID-&gt;"ID" ]</code>	create an encoded file which can only be read on a machine with a particular ID

Creating and reading encoded files.

This writes an expression in plain text to the file `tmp`.

```
In[61]:= Factor[x^2 - 1] >> tmp
```

This writes an encoded version of the file `tmp` to the file `tmp.x`.

```
In[62]:= Encode["tmp" , "tmp.x"]
```

Here are the contents of the encoded file. The only recognizable part is the special *Mathematica* comment at the beginning.

```
In[63]:= FilePrint["tmp.x"]

(*!1N!* )mcm
_QZ9tcI1cfre*Wo8:) P
```

Even though the file is encoded, you can still read it into *Mathematica* using the << operator.

```
In[64]:= << tmp.x
Out[64]= (-1 + x) (1 + x)
```

DumpSave ["file.mx", symbol]	save definitions for a symbol in internal <i>Mathematica</i> format
DumpSave ["file.mx", "context^ "]	save definitions for all symbols in a context
DumpSave ["file.mx", {object <sub>1</sub> , object <sub>2</sub> , ...}]	save definitions for several symbols or contexts
DumpSave ["package^ ", objects]	save definitions in a file with a specially chosen name

Saving definitions in internal *Mathematica* format.

If you have to read in very large or complicated definitions, you will often find it more efficient to store these definitions in internal *Mathematica* format, rather than as text. You can do this using `DumpSave`.

This saves the definition for `f` in internal *Mathematica* format.

```
In[22]:= DumpSave["ffile.mx", f]
Out[22]= {f}
```

You can still use << to read the definition in.

```
In[23]:= << ffile.mx
```

<< recognizes when a file contains definitions in internal *Mathematica* format, and operates accordingly. One subtlety is that the internal *Mathematica* format differs from one computer system to another. As a result, .mx files created on one computer cannot typically be read on another.

If you use `DumpSave["package`", ...]` then *Mathematica* will write out definitions to a file with a name like `package.mx / system / package.mx`, where *system* identifies your type of computer system.

This creates a file with a name that reflects the name of the computer system being used.

```
In[24]:= DumpSave["gffile`", f]
```

```
Out[24]= {f}
```

<< automatically picks out the file with the appropriate name for your computer system.

```
In[25]:= << gffile`
```

## External Programs

On most computer systems, you can execute external programs or commands from within *Mathematica*. Often you will want to take expressions you have generated in *Mathematica*, and send them to an external program, or take results from external programs, and read them into *Mathematica*.

*Mathematica* supports two basic forms of communication with external programs: *structured* and *unstructured*.

Structured communication	use <i>MathLink</i> to exchange expressions with <i>MathLink</i> -compatible external programs
Unstructured communication	use file reading and writing operations to exchange ordinary text

Two kinds of communication with external programs in *Mathematica*.

The idea of structured communication is to exchange complete *Mathematica* expressions to external programs which are specially set up to handle such objects. The basis for structured communication is the *MathLink* system, discussed in "*MathLink* and External Program Communication".

Unstructured communication consists in sending and receiving ordinary text from external programs. The basic idea is to treat an external program very much like a file, and to support the same kinds of reading and writing operations.

<code>&lt;&lt;file</code>	read in a file
<code>&lt;&lt;"!command"</code>	run an external command, and read in the output it produces
<code>expr&gt;&gt;"!command"</code>	feed the textual form of <i>expr</i> to an external command
<code>ReadList["!command", Number]</code>	run an external command, and read in a list of the numbers it produces

Some ways to communicate with external programs.

In general, wherever you might use an ordinary file name, *Mathematica* allows you instead to give a *pipe*, written as an external command, prefaced by an exclamation point. When you use the pipe, *Mathematica* will execute the external command, and send or receive text from it.

This sends the result from `FactorInteger` to the external program `lpr`. On many Unix systems, this program generates a printout.

```
In[1]:= FactorInteger[2^31 - 1] >> !lpr
```

This executes the external command `echo $TERM`, then reads the result as *Mathematica* input.

```
In[2]:= << "!echo $TERM"
```

```
Out[2]= xterm
```

With a text-based interface, putting `!` at the beginning of a line causes the remainder of the line to be executed as an external command. `squares` is an external program which prints numbers and their squares.

```
In[1]:= !squares 4
```

```
1 1
2 4
3 9
4 16
```

This runs the external command `squares 4`, then reads numbers from the output it produces.

```
In[3]:= ReadList["!squares 4", Number, RecordLists -> True]
```

```
Out[3]= {{1, 1}, {2, 4}, {3, 9}, {4, 16}}
```

One point to notice is that you can get away with dropping the double quotes around the name of a pipe on the right-hand side of `<<` or `>>` if the name does not contain any spaces or other special characters.

Pipes in *Mathematica* provide a very general mechanism for unstructured communication with external programs. On many computer systems, *Mathematica* pipes are implemented using pipe mechanisms in the underlying operating system; in some cases, however, other interprocess communication mechanisms are used. One restriction of unstructured communication in *Mathematica* is that a given pipe can only be used for input or for output, and not for both at the same time. In order to do genuine two-way communication, you need to use *MathLink*.

Even with unstructured communication, you can nevertheless set up somewhat more complicated arrangements by using "temporary files". The basic idea is to write data to a file, then to read it as needed.

<code>OpenWrite[]</code>	open a new file with a unique name in the default area for temporary files on your computer system
--------------------------	--

Opening a "temporary file".

Particularly when you work with temporary files, you may find it useful to be able to execute external commands which do not explicitly send or receive data from *Mathematica*. You can do this using the *Mathematica* function `Run`.

<code>Run["command", arg<sub>1</sub>, ...]</code>	run an external command from within <i>Mathematica</i>
---	--

Running external commands without input or output.

This executes the external Unix command `date`. The returned value is an "exit code" from the operating system.

```
In[4]:= Run["date"]
```

```
Out[4]= 0
```

Note that when you use `Run`, you must not preface commands with exclamation points. `Run` simply takes the textual forms of the arguments you specify, then joins them together with spaces in between, and executes the resulting string as an external shell command.

It is important to realize that `Run` never "captures" any of the output from an external command. As a result, where this output goes is purely determined by your operating system. Similarly, `Run` does not supply input to external commands. This means that the commands can get input through any mechanism provided by your operating system. Sometimes external commands may be able to access the same input and output streams that are used by *Mathematica* itself. In some cases, this may be what you want. But particularly if you are using *Mathematica* with a front end, this can cause considerable trouble.

<code>RunThrough [ "command" , expr ]</code>	run <i>command</i> , using <i>expr</i> as input, and reading the output back into <i>Mathematica</i>
--	--

Running *Mathematica* expressions through external programs.

As discussed above, `<<` and `>>` cannot be used to both send and receive data from an external program at the same time. Nevertheless, by using temporary files, you can effectively both send and receive data from an external program while still using unstructured communication.

The function `RunThrough` writes the text of an expression to a temporary file, then feeds this file as input to an external program, and captures the output as input to *Mathematica*. Note that in `RunThrough`, like `Run`, you should not preface the names of external commands with exclamation points.

This feeds the expression 789 to the external program `cat`, which in this case simply echoes the text of the expression. The output from `cat` is then read back into *Mathematica*.

```
In[5]:= RunThrough["cat", 789]
Out[5]= 789
```

<code>SystemOpen [ "target" ]</code>	opens the specified file, URL or other target with the associated program on your computer system
--------------------------------------	---

Opening files with external programs.

This opens the URL using your system's preferred web browser.

```
In[6]:= SystemOpen["http://www.wolfram.com"]
```

`SystemOpen` uses settings in your operating system to determine how to open a URI or file. When opening files, it typically uses the same program that would be used if you double-clicked the file's icon.

## Streams and Low-Level Input and Output

Files and pipes are both examples of general *Mathematica* objects known as *streams*. A stream in *Mathematica* is a source of input or output. There are many operations that you can perform on streams.

You can think of `>>` and `<<` as "high-level" *Mathematica* input-output functions. They are based on a set of lower-level input-output primitives that work directly with streams. By using these primitives, you can exercise more control over exactly how *Mathematica* does input and output. You will often need to do this, for example, if you write *Mathematica* programs which store and retrieve intermediate data from files or pipes.

The basic low-level scheme for writing output to a stream in *Mathematica* is as follows. First, you call `OpenWrite` or `OpenAppend` to "open the stream", telling *Mathematica* that you want to write output to a particular file or external program, and in what form the output should be written. Having opened a stream, you can then call `Write` or `WriteString` to write a sequence of expressions or strings to the stream. When you have finished, you call `Close` to "close the stream".

<code>"name"</code>	a file, specified by name
<code>"!name"</code>	a command, specified by name
<code>InputStream["name", n]</code>	an input stream
<code>OutputStream["name", n]</code>	an output stream

Streams in *Mathematica*.

When you open a file or a pipe, *Mathematica* creates a "stream object" that specifies the open stream associated with the file or pipe. In general, the stream object contains the name of the file or the external command used in a pipe, together with a unique number.

The reason that the stream object needs to include a unique number is that in general you can have several streams connected to the same file or external program at the same time. For example, you may start several different instances of the same external program, each connected to a different stream.

Nevertheless, when you have opened a stream, you can still refer to it using a simple file name or external command name so long as there is only one stream associated with this object.



This opens an output stream to the file `tmp`.

```
In[1]:= stmp = OpenWrite["tmp"]
```

```
Out[1]= OutputStream[tmp, 36]
```

This writes a sequence of expressions to the file.

```
In[2]:= Write[stmp, a, b, c]
```

Since you only have one stream associated with file `tmp`, you can refer to it simply by giving the name of the file.

```
In[3]:= Write["tmp", x]
```

This closes the stream.

```
In[4]:= Close[stmp]
```

```
Out[4]= tmp
```

Here is what was written to the file.

```
In[5]:= FilePrint["tmp"]
```

```
abc
x
```

<code>OpenWrite["file"]</code>	open an output stream to a file, wiping out the previous contents of the file
<code>OpenWrite[]</code>	open an output stream to a new temporary file
<code>OpenAppend["file"]</code>	open an output stream to a file, appending to what was already in the file
<code>OpenWrite["!command"]</code>	open an output stream to an external command
<code>Write[stream, expr<sub>1</sub>, expr<sub>2</sub>, ...]</code>	write a sequence of expressions to a stream, ending the output with a newline (line feed)
<code>WriteString[stream, str<sub>1</sub>, str<sub>2</sub>, ...]</code>	write a sequence of character strings to a stream, with no extra newlines
<code>Close[stream]</code>	tell <i>Mathematica</i> that you are finished with a stream

Low-level output functions.

When you call `Write[stream, expr]`, it writes an expression to the specified stream. The default is to write the expression in *Mathematica* input form. If you call `Write` with a sequence of expressions, it will write these expressions one after another to the stream. In general, it leaves no space between the successive expressions. However, when it has finished writing all the expressions, `Write` always ends its output with a newline.

This reopens the file tmp.

```
In[6]:= stmp = OpenWrite["tmp"]
Out[6]= OutputStream[tmp, 37]
```

This writes a sequence of expressions to the file, then closes the file.

```
In[7]:= Write[stmp, a^2, 1 + b^2]; Write[stmp, c^3]; Close[stmp]
Out[7]= tmp
```

All the expressions are written in input form. The expressions from a single `write` are put on the same line.

```
In[8]:= FilePrint["tmp"]
      a^21 + b^2
      c^3
```

`write` provides a way of writing out complete *Mathematica* expressions. Sometimes, however, you may want to write out less structured data. `writeString` allows you to write out any character string. Unlike `write`, `writeString` adds no newlines or other characters.

This opens the stream.

```
In[9]:= stmp = OpenWrite["tmp"]
Out[9]= OutputStream[tmp, 38]
```

This writes two strings to the stream.

```
In[10]:= WriteString[stmp, "Arbitrary output.\n", "More output."]
```

This writes another string, then closes the stream.

```
In[11]:= WriteString[stmp, " Second line.\n"]; Close[stmp]
Out[11]= tmp
```

Here are the contents of the file. The strings were written exactly as specified, including only the newlines that were explicitly given.

```
In[12]:= FilePrint["tmp"]
      Arbitrary output.
      More output. Second line.
```

<code>Write [ {stream<sub>1</sub>, stream<sub>2</sub>}, expr<sub>1</sub>, ...]</code>	write expressions to a list of streams
<code>WriteString [ {stream<sub>1</sub>, stream<sub>2</sub>}, str<sub>1</sub>, ...]</code>	write strings to a list of streams

Writing output to lists of streams.

An important feature of the functions `Write` and `WriteString` is that they allow you to write output not just to a single stream, but also to a list of streams.

In using *Mathematica*, it is often convenient to define a *channel* which consists of a list of streams. You can then simply tell *Mathematica* to write to the channel, and have it automatically write the same object to several streams.

In a standard interactive *Mathematica* session, there are several output channels that are usually defined. These specify where particular kinds of output should be sent. Thus, for example, `$Output` specifies where standard output should go, while `$Messages` specifies where messages should go. The function `Print` then works essentially by calling `Write` with the `$Output` channel. `Message` works in the same way by calling `Write` with the `$Messages` channel. "The Main Loop" lists the channels used in a typical *Mathematica* session.

Note that when you run *Mathematica* through *MathLink*, a different approach is usually used. All output is typically written to a single *MathLink* link, but each piece of output appears in a "packet" which indicates what type it is.

In most cases, the names of files or external commands that you use in *Mathematica* correspond exactly with those used by your computer's operating system. On some systems, however, *Mathematica* supports various streams with special names.

<code>"stdout"</code>	standard output
<code>"stderr"</code>	standard error

Special streams used on some computer systems.

The special stream `"stdout"` allows you to give output to the "standard output" provided by the operating system. Note however that you can use this stream only with simple text-based interfaces to *Mathematica*. If your interaction with *Mathematica* is more complicated, then this stream will not work, and trying to use it may cause considerable trouble.

<i>option name</i>	<i>default value</i>	
FormatType	InputForm	the default output format to use
PageWidth	78	the width of the page in characters
NumberMarks	\$NumberMarks	whether to include $\sim$ marks in approximate numbers
CharacterEncoding	\$CharacterEncoding	encoding to be used for special characters

Some options for output streams.

You can associate a number of options with output streams. You can specify these options when you first open a stream using `OpenWrite` or `OpenAppend`.

This opens a stream, specifying that the default output format used should be `OutputForm`.

```
In[13]:= stmp = OpenWrite["tmp", FormatType -> OutputForm]
Out[13]= OutputStream[tmp, 39]
```

This writes expressions to the stream, then closes the stream.

```
In[14]:= Write[stmp, x^2 + y^2, " ", z^2]; Close[stmp]
Out[14]= tmp
```

The expressions were written to the stream in `OutputForm`.

```
In[15]:= FilePrint["tmp"]
      2    2    2
      x  + y  z
```

Note that you can always override the output format specified for a particular stream by wrapping a particular expression you write to the stream with an explicit *Mathematica* format directive, such as `OutputForm` or `TeXForm`.

The option `PageWidth` gives the width of the page available for textual output from *Mathematica*. All lines of output are broken so that they fit in this width. If you do not want any lines to

be broken, you can set `PageWidth -> Infinity`. Usually, however, you will want to set `PageWidth` to the value appropriate for your particular output device. On many systems, you will have to run an external program to find out what this value is. Using `SetOptions`, you can make the default rule for `PageWidth` be, for example, `PageWidth := << "!devicewidth"`, so that an external program is run automatically to find the value of the option.

This opens a stream, specifying that the page width is 20 characters.

```
In[16]:= stmp = OpenWrite["tmp", PageWidth -> 20]
Out[16]= OutputStream[tmp, 40]
```

This writes out an expression, then closes the stream.

```
In[17]:= Write[stmp, Expand[(1 + x)^5]]; Close[stmp]
Out[17]= tmp
```

The lines in the expression written out are all broken so as to be at most 20 characters long.

```
In[18]:= FilePrint["tmp"]

1 + 5*x + 10*x^2 +
 10*x^3 + 5*x^4 +
 x^5
```

The option `CharacterEncoding` allows you to specify a character encoding that will be used for all strings which are sent to a particular output stream, whether by `write` or `writeString`. You will typically need to use `CharacterEncoding` if you want to modify an international character set, or prevent a particular output device from receiving characters that it cannot handle.

<code>Options</code> [ <i>stream</i> ]	find the options that have been set for a stream
<code>SetOptions</code> [ <i>stream</i> , <i>opt</i> <sub>1</sub> -> <i>val</i> <sub>1</sub> , ...]	reset options for an open stream

Manipulating options of streams.

This opens a stream with the default settings for options.

```
In[19]:= stmp = OpenWrite["tmp"]
Out[19]= OutputStream[tmp, 41]
```

This changes the `FormatType` option for the open stream.

```
In[20]:= SetOptions[stmp, FormatType -> TeXForm];
```

`Options` shows the options you have set for the open stream.

```
In[21]:= Options[stmp]
```

```
Out[21]= {BinaryFormat -> False, FormatType -> TeXForm, PageWidth -> 78, PageHeight -> 22,
  TotalWidth -> ∞, TotalHeight -> ∞, CharacterEncoding -> Automatic, NumberMarks -> $NumberMarks}
```

This closes the stream again.

```
In[22]:= Close[stmp]
```

```
Out[22]= tmp
```

<code>Options[\$Output]</code>	find the options set for all streams in the channel <code>\$Output</code>
<code>SetOptions[\$Output, opt<sub>1</sub>-&gt;val<sub>1</sub>, ...]</code>	set options for all streams in the channel <code>\$Output</code>

Manipulating options for the standard output channel.

At every point in your session, *Mathematica* maintains a list `streams[]` of all the input and output streams that are currently open, together with their options. In some cases, you may find it useful to look at this list directly. *Mathematica* will not, however, allow you to modify the list, except indirectly through `OpenRead` and so on.

## Naming and Finding Files

### Directory Operations

The precise details of the naming of files differ from one computer system to another. Nevertheless, *Mathematica* provides some fairly general mechanisms that work on all systems.

*Mathematica* assumes that all your files are arranged in a hierarchy of *directories*. To find a particular file, *Mathematica* must know both what the name of the file is, and what sequence of directories it is in.

At any given time, however, you have a *current working directory*, and you can refer to files or other directories by specifying where they are relative to this directory. Typically you can refer to files or directories that are actually *in* this directory simply by giving their names, with no directory information.

<code>Directory[]</code>	your current working directory
<code>SetDirectory["dir"]</code>	set your current working directory
<code>ResetDirectory[]</code>	revert to your previous working directory

Manipulating directories.

This gives a string representing your current working directory.

```
In[1]:= Directory[]
Out[1]= /users/sw
```

This sets your current working directory to be the `Examples` subdirectory.

```
In[2]:= SetDirectory["Examples"]
Out[2]= /users/sw/Examples
```

Now your current working directory is different.

```
In[3]:= Directory[]
Out[3]= /users/sw/Examples
```

This reverts to your previous working directory.

```
In[4]:= ResetDirectory[]
Out[4]= /users/sw
```

When you call `SetDirectory`, you can give any directory name that is recognized by your operating system. Thus, for example, on Unix-based systems, you can specify a directory one level up in the directory hierarchy using the notation `..`, and you can specify your "home" directory as `~`.

Whenever you go to a new directory using `SetDirectory`, *Mathematica* always remembers what the previous directory was. You can return to this previous directory using `ResetDirectory`. In general, *Mathematica* maintains a stack of directories, given by `DirectoryStack[]`. Every time you call `SetDirectory`, it adds a new directory to the stack, and every time you call `ResetDirectory` it removes a directory from the stack.

<code>ParentDirectory[]</code>	the parent of your current working directory
<code>\$InitialDirectory</code>	the initial directory when <i>Mathematica</i> was started
<code>\$HomeDirectory</code>	your home directory, if this is defined
<code>\$BaseDirectory</code>	the base directory for systemwide files to be loaded by <i>Mathematica</i>
<code>\$UserBaseDirectory</code>	the base directory for user-specific files to be loaded by <i>Mathematica</i>
<code>\$InstallationDirectory</code>	the top-level directory in which your <i>Mathematica</i> installation resides

Special directories.

## Finding a File

Whenever you ask for a particular file, *Mathematica* in general goes through several steps to try and find the file you want. The first step is to use whatever standard mechanisms exist in your operating system or shell.

*Mathematica* scans the full name you give for a file, and looks to see whether it contains any of the "metacharacters" `*`, `$`, `~`, `?`, `[`, `"`, `\` and `'`. If it finds such characters, then it passes the full name to your operating system or shell for interpretation. This means that if you are using a Unix-based system, then constructions like `name *` and `$VAR` will be expanded at this point. But in general, *Mathematica* takes whatever was returned by your operating system or shell, and treats this as the full file name.

For output files, this is the end of the processing that *Mathematica* does. If *Mathematica* cannot find a unique file with the name you specified, then it will proceed to create the file.

If you are trying to get input from a file, however, then there is another round of processing that *Mathematica* does. What happens is that *Mathematica* looks at the value of the `Path` option for the function you are using to determine the names of directories relative to which it should search for the file. The default setting for the `Path` option is the global variable `$Path`.

<code>Get["file", Path-&gt;{"dir1", "dir2", ...}]</code>	get a file, searching for it relative to the directories <code>dir<sub>i</sub></code>
<code>\$Path</code>	default list of directories relative to which to search for input files

Search path for files.



In general, the global variable `$Path` is defined to be a list of strings, with each string representing a directory. Every time you ask for an input file, what *Mathematica* effectively does is temporarily to make each of these directories in turn your current working directory, and then from that directory to try and find the file you have requested.

Here is a typical setting for `$Path`. The current directory (`.`) and your home directory (`~`) are listed first.

```
In[5]:= $Path
Out[5]= {., ~, /users/math/bin, /users/math/Packages}
```

You can also use `FindFile` to locate a file.

<code>FindFile["name"]</code>	find the file with the specified name that would be loaded by <code>Get</code> and related functions
<code>FileExistsQ["name"]</code>	determine whether the file exists

Finding a file on the `$Path`.

`FindFile` searches all directories in `$Path` and returns the absolute name of the file that would be loaded by `Get`, `Needs`, and other functions. `FileExistsQ` tests whether the file with the given name exists.

```
In[5]:= FindFile["init.m"]
Out[5]= "C:\\Documents and Settings\\sw\\Application
Data\\Mathematica\\Kernel\\init.m"
```

`FindFile` applied to a package name returns the absolute name of the `init.m` file from that package.

```
In[5]:= FindFile["Combinatorica`"]
Out[5]= "C:\\Program Files\\Wolfram
Research\\Mathematica\\7.0\\AddOns\\Packages\\Combinatorica\\Kernel\\init.m"
```

## Listing Contents of Directories

<code>FileNames []</code>	list all files in your current working directory
<code>FileNames ["form"]</code>	list all files in your current working directory whose names match the string pattern <i>form</i>
<code>FileNames [{"form<sub>1</sub>", "form<sub>2</sub>", ...}]</code>	list all files whose names match any of the <i>form<sub>i</sub></i>
<code>FileNames [forms, {"dir<sub>1</sub>", "dir<sub>2</sub>", ...}]</code>	give the full names of all files whose names match <i>forms</i> in any of the directories <i>dir<sub>i</sub></i>
<code>FileNames [forms, dirs, n]</code>	include files that are in subdirectories up to <i>n</i> levels down
<code>FileNames [forms, dirs, Infinity]</code>	include files in all subdirectories
<code>FileNames [forms, \$Path, Infinity]</code>	give all files whose names match <i>forms</i> in any subdirectory of the directories in <i>\$Path</i>

Getting lists of files in particular directories.

`FileNames` returns a list of strings corresponding to file names. When it returns a file that is not in your current directory, it gives the name of the file relative to the current directory. Note that all names are given in the format appropriate for the particular computer system on which they were generated.

Here is a list of all files in the current working directory whose names end with `.m`.

```
In[6]:= FileNames["*.m"]
Out[6]= {alpha.m, control.m, signals.m, test.m}
```

This lists files whose names start with `a` in the current directory, and in subdirectories with names that start with `P`.

```
In[7]:= FileNames["a*", {".", "P*"}]
Out[7]= {alpha.m, Packages/astrodata, Packages/astro.m, Previous/atmp}
```

The file name form you give to `FileNames` can use any of *Mathematica*'s string pattern objects, typically combined with the `~~` operator.

This gives a list of all files in your current working directory whose names match the form `Test*.m`.

```
In[3]:= FileNames["Test*.m"]
Out[3]= {Test1.m, Test2.m, TestFinal.m}
```

This lists only those files with names of the form `Test d.m`, where `d` is a sequence of one or more digits.

```
In[3]:= FileNames["Test" ~~ DigitCharacter .. ~~ ".m"]
Out[3]= {Test1.m, Test2.m}
```

## Composing a Filename

<code>DirectoryName["file"]</code>	extract the directory name from a file name
<code>ToFileName["directory", "name"]</code>	assemble a full file name from a directory name and a file name
<code>ParentDirectory["directory"]</code>	give the parent of a directory
<code>ToFileName[{"dir1", "dir2", ...}, "name"]</code>	assemble a full file name from a hierarchy of directory names
<code>ToFileName[{"dir1", "dir2", ...}]</code>	assemble a single directory name from a hierarchy of directory names

Manipulating file names.

You should realize that different computer systems may give file names in different ways. Thus, for example, Windows systems typically give names in the form `dir : \ dir \ dir \ name` and Unix systems give names in the form `dir / dir / name`. The function `ToFileName` assembles file names in the appropriate way for the particular computer system you are using.

This gives the directory portion of the file name.

```
In[8]:= DirectoryName["Packages/Math/test.m"]
Out[8]= Packages/Math/
```

This constructs the full name of another file in the same directory as `test.m`.

```
In[9]:= ToFileName[%, "abc.m"]
Out[9]= Packages/Math/abc.m
```

<code>FileNameSplit["name"]</code>	split the file name into a list of directory and file names
<code>FileNameJoin[{dir<sub>1</sub>, ...}]</code>	combine a list of directory and file names into the file name
<code>FileNameTake["name", ...]</code>	extract part of the file name
<code>FileNameDrop["name", ...]</code>	drop parts of the file name
<code>FileNameDepth["name"]</code>	get the number of path elements in the file name
<code>\$PathnameSeparator</code>	path name separator used in your operating system

Manipulating file names.

Functions like `FileNameSplit` and `FileNameJoin` provide additional operations on file names. They respect the file name separator used by your operating system and will split the file name appropriately. `FileNameJoin` will by default use the `$PathnameSeparator` to produce the name in a canonical form suitable for your operating system.

If you want to set up a collection of related files, it is often convenient to be able to refer to one file when you are reading another one. The global variable `$Input` gives the name of the file from which input is currently being taken. Using `DirectoryName` and `ToFileName` you can then conveniently specify the names of other related files.

<code>\$Input</code>	the name of the file or stream from which input is currently being taken
----------------------	--

Finding out how to refer to a file currently being read by *Mathematica*.

One issue in handling files in *Mathematica* is that the form of file and directory names varies between computer systems. This means for example that names of files which contain standard *Mathematica* packages may be quite different on different systems. Through a sequence of conventions, it is however possible to read in a standard *Mathematica* package with the same command on all systems. The way this works is that each package defines a so-called *Mathematica* context, of the form `name`name``. On each system, all files are named in correspondence with the contexts they define. Then when you use the command `<< name`name` Mathematica` automatically translates the context name into the file name appropriate for your particular computer system.

## Standard Filename Extensions

<code>file.m</code>	<i>Mathematica</i> expression file in plain text format
<code>file.nb</code>	<i>Mathematica</i> notebook file
<code>file.mx</code>	<i>Mathematica</i> definitions in DumpSave format

Typical names of *Mathematica* files.

If you use a notebook interface to *Mathematica*, then the *Mathematica* front end allows you to save complete notebooks, including not only *Mathematica* input and output, but also text, graphics and other material.

It is conventional to give *Mathematica* notebook files names that end in `.nb`, and most versions of *Mathematica* enforce this convention.

<code>FileName</code> ["name"]	the parent of your current working directory
<code>FileExtension</code> ["name"]	the initial directory when <i>Mathematica</i> was started

File name and extension.

You can use `FileName` and `FileExtension` to extract the name of the file and its extension.

When you open a notebook in the *Mathematica* front end, *Mathematica* will immediately display the contents of the notebook, but it will not normally send any of these contents to the kernel for evaluation until you explicitly request this to be done.

Within a *Mathematica* notebook, however, you can use the **Cell** menu in the front end to identify certain cells as *initialization cells*, and if you do this, then the contents of these cells will automatically be evaluated whenever you open the notebook.

The **I** in the cell bracket indicates that the second cell is an initialization cell that will be evaluated whenever the notebook is opened.

**Implementation** ]

```
f[x_] := Log[x] + Log[1 - x]
```

It is sometimes convenient to maintain *Mathematica* material both in a notebook which contains explanatory text, and in a package which contains only raw *Mathematica* definitions. You can do this by putting the *Mathematica* definitions into initialization cells in the notebook. Every time you save the notebook, the front end will then allow you to save an associated `.m` file which contains only the raw *Mathematica* definitions.

## Files for Packages

When you create or use *Mathematica* packages, you will often want to refer to files in a system-independent way. You can use contexts to do this.

The basic idea is that on every computer system there is a convention about how files corresponding to *Mathematica* contexts should be named. Then, when you refer to a file using a context, the particular version of *Mathematica* you are using converts the context name to the file name appropriate for the computer system you are on.

<code>&lt;&lt;context`</code>	read in the file corresponding to the specified context
-------------------------------	---

Using contexts to specify files.

This reads in one of the standard packages that come with *Mathematica*.

```
In[1]:= << VectorAnalysis`
```

<code>name.mx</code>	file in DumpSave format
<code>name.mx/\$SystemID/name.mx</code>	file in DumpSave format for your computer system
<code>name.m</code>	file in <i>Mathematica</i> source format
<code>name/init.m</code>	initialization file for a particular directory
<code>dir/...</code>	files in other directories specified by <code>\$Path</code>

The typical sequence of files looked for by `<< name``.

*Mathematica* is set up so that `<< name`` will automatically try to load the appropriate version of a file. It will first try to load a `name.mx` file that is optimized for your particular computer system. If it finds no such file, then it will try to load a `name.m` file containing ordinary system-independent *Mathematica* input.

If *name* is a directory, then *Mathematica* will try to load the initialization file `init.m` in that directory. The purpose of the `init.m` file is to provide a convenient way to set up *Mathematica* packages that involve many separate files. The idea is to allow you to give just the command `<< name``, but then to load `init.m` to initialize the whole package, reading in whatever other files are necessary.

## Manipulating Files and Directories

<code>CopyFile ["file<sub>1</sub>", "file<sub>2</sub>"]</code>	copy <i>file<sub>1</sub></i> to <i>file<sub>2</sub></i>
<code>RenameFile ["file<sub>1</sub>", "file<sub>2</sub>"]</code>	give <i>file<sub>1</sub></i> the name <i>file<sub>2</sub></i>
<code>DeleteFile ["file"]</code>	delete a file
<code>FileByteCount ["file"]</code>	give the number of bytes in a file
<code>FileDate ["file"]</code>	give the modification date for a file
<code>SetFileDate ["file"]</code>	set the modification date for a file to be the current date
<code>FileType ["file"]</code>	give the type of a file as File, Directory or None

Functions for manipulating files.

Different operating systems have different commands for manipulating files. *Mathematica* provides a simple set of file manipulation functions, intended to work in the same way under all operating systems.

Notice that `CopyFile` and `RenameFile` give the final file the same modification date as the original one. `FileDate` returns modification dates in the `{year, month, day, hour, minute, second}` format used by `DateList`.

<code>CreateDirectory ["name"]</code>	create a new directory
<code>DeleteDirectory ["name"]</code>	delete an empty directory
<code>DeleteDirectory ["name", DeleteContents-&gt;True]</code>	delete a directory and all files and directories it contains
<code>RenameDirectory ["name<sub>1</sub>", "name<sub>2</sub>"]</code>	rename a directory
<code>CopyDirectory ["name<sub>1</sub>", "name<sub>2</sub>"]</code>	copy a directory and all the files in it

Functions for manipulating directories.

## Reading Textual Data

With `<<`, you can read files which contain *Mathematica* expressions given in input form. Sometimes, however, you may instead need to read files of *data* in other formats. For example, you may have data generated by an external program which consists of a sequence of numbers separated by spaces. This data cannot be read directly as *Mathematica* input. However, the function `ReadList` can take such data from a file or input stream, and convert it to a *Mathematica* list.

<code>ReadList["file", Number]</code>	read a sequence of numbers from a file, and put them in a <i>Mathematica</i> list
---------------------------------------	---

Reading numbers from a file.

Here is a file of numbers.

```
In[1]:= FilePrint["ExampleData/numbers"]
```

```
11.1  22.2  33.3
44.4  55.5  66.6
```

This reads all the numbers in the file, and returns a list of them.

```
In[2]:= ReadList["ExampleData/numbers", Number]
```

```
Out[2]= {11.1, 22.2, 33.3, 44.4, 55.5, 66.6}
```

<code>ReadList["file", {Number, Number}]</code>	read numbers from a file, putting each successive pair into a separate list
---	---

<code>ReadList["file", Table[Number, {n}]]</code>	put each successive block of <i>n</i> numbers in a separate list
---	--

<code>ReadList["file", Number, RecordLists-&gt;True]</code>	put all the numbers on each line of the file into a separate list
---	---

Reading blocks of numbers.

This puts each successive pair of numbers from the file into a separate list.

```
In[3]:= ReadList["ExampleData/numbers", {Number, Number}]
```

```
Out[3]= {{11.1, 22.2}, {33.3, 44.4}, {55.5, 66.6}}
```



This makes each line in the file into a separate list.

```
In[4]:= ReadList["ExampleData/numbers", Number, RecordLists -> True]
Out[4]= {{11.1, 22.2, 33.3}, {44.4, 55.5, 66.6}}
```

`ReadList` can handle numbers which are given in Fortran-like "E" notation. Thus, for example, `ReadList` will read `2.5 E + 5` as  $2.5 \times 10^5$ . Note that `ReadList` can handle numbers with any number of digits of precision.

Here is a file containing numbers in Fortran-like "E" notation.

```
In[5]:= FilePrint["ExampleData/bignum"]
      4.5E-5      7.8E4
      2.5E2      -8.9
```

`ReadList` can handle numbers in this form.

```
In[6]:= ReadList["ExampleData/bignum", Number]
Out[6]= {0.000045, 78000., 250., -8.9}
```

<code>ReadList["file", type]</code>	read a sequence of objects of a particular type
<code>ReadList["file", type, n]</code>	read at most <i>n</i> objects

Reading objects of various types.

`ReadList` can read not only numbers, but also a variety of other types of object. Each type of object is specified by a symbol such as `Number`.

Here is a file containing text.

```
In[7]:= FilePrint["ExampleData/strings"]
      Here is text.
      And more text.
```

This produces a list of the characters in the file, each given as a one-character string.

```
In[8]:= ReadList["ExampleData/strings", Character]
Out[8]= {H, e, r, e, , i, s, , t, e, x, t, ., ., ,
      , A, n, d, , m, o, r, e, , t, e, x, t, ., .,
      }
```

Here are the integer codes corresponding to each of the bytes in the file.

```
In[9]:= ReadList["ExampleData/strings", Byte]
Out[9]= {72, 101, 114, 101, 32, 105, 115, 32, 116, 101, 120, 116, 46, 32,
      10, 65, 110, 100, 32, 109, 111, 114, 101, 32, 116, 101, 120, 116, 46, 10}
```

This puts the data from each line in the file into a separate list.

```
In[10]:= ReadList["ExampleData/strings", Byte, RecordLists -> True]
Out[10]= {{72, 101, 114, 101, 32, 105, 115, 32, 116, 101, 120, 116, 46, 32},
          {65, 110, 100, 32, 109, 111, 114, 101, 32, 116, 101, 120, 116, 46}}
```

Byte	single byte of data, returned as an integer
Character	single character, returned as a one-character string
Real	approximate number in Fortran-like notation
Number	exact or approximate number in Fortran-like notation
Word	sequence of characters delimited by word separators
Record	sequence of characters delimited by record separators
String	string terminated by a newline
Expression	complete <i>Mathematica</i> expression
Hold[Expression]	complete <i>Mathematica</i> expression, returned inside Hold

Types of objects to read.

This returns a list of the "words" in the file strings.

```
In[11]:= ReadList["ExampleData/strings", Word]
Out[11]= {Here, is, text., And, more, text.}
```

`ReadList` allows you to read "words" from a file. It considers a "word" to be any sequence of characters delimited by word separators. You can set the option `wordSeparators` to specify the strings you want to treat as word separators. The default is to include spaces and tabs, but not to include, for example, standard punctuation characters. Note that in all cases successive words can be separated by any number of word separators. These separators are never taken to be part of the actual words returned by `ReadList`.

<i>option name</i>	<i>default value</i>	
<code>RecordLists</code>	<code>False</code>	whether to make a separate list for the objects in each record
<code>RecordSeparators</code>	<code>{"\r\n", "\n", "\r"}</code>	separators for records
<code>WordSeparators</code>	<code>{" ", "\t"}</code>	separators for words
<code>NullRecords</code>	<code>False</code>	whether to keep zero-length records
<code>NullWords</code>	<code>False</code>	whether to keep zero-length words
<code>TokenWords</code>	<code>{}</code>	words to take as tokens

Options for `ReadList`.

This reads the text in the file `strings` as a sequence of words, using the letter `e` and `.` as word separators.

```
In[12]:= ReadList["ExampleData/strings", Word, WordSeparators -> {"e", "."}]
Out[12]= {H, r, is t, xt, , And mor, t, xt}
```

*Mathematica* considers any data file to consist of a sequence of *records*. By default, each line is considered to be a separate record. In general, you can set the option `RecordSeparators` to give a list of separators for records. Note that words can never cross record separators. As with word separators, any number of record separators can exist between successive records, and these separators are not considered to be part of the records themselves.

By default, each line of the file is considered to be a record.

```
In[13]:= ReadList["ExampleData/strings", Record] // InputForm
Out[13]//InputForm= {"Here is text. ", "And more text."}
```

Here is a file containing three "sentences" ending with periods.

```
In[14]:= FilePrint["ExampleData/sentences"]

Here is text. And more.
And a second line.
```

This allows both periods and newlines as record separators.

```
In[15]:= ReadList["ExampleData/sentences", Record, RecordSeparators -> {".", "\n"}]
Out[15]= {Here is text, And more, And a second line}
```

This puts the words in each "sentence" into a separate list.

```
In[16]:= ReadList["ExampleData/sentences", Word,
  RecordLists -> True, RecordSeparators -> {".", "\n"}]
Out[16]= {{Here, is, text}, {And, more}, {And, a, second, line}}
```

```
ReadList["file", Record, RecordSeparators -> {}]
```

read the whole of a file as a single string

```
ReadList["file", Record, RecordSeparators -> {{ "lsep1", ... }, { "rsep1", ... } }
```

make a list of those parts of a file which lie between the *lsep<sub>i</sub>* and the *rsep<sub>i</sub>*

Settings for the `RecordSeparators` option.

Here is a file containing some text.

```
In[17]:= FilePrint["ExampleData/source"]

      f[x] (: function f :)
      g[x] (: function g :)
```

This reads all the text in the file source, and returns it as a single string.

```
In[18]:= InputForm[ReadList["ExampleData/source", Record, RecordSeparators -> {}]]
Out[18]//InputForm= {"f[x] (: function f :)\ng[x] (: function g :)\n"}
```

This gives a list of the parts of the file that lie between ( : and : ) separators.

```
In[19]:= ReadList["ExampleData/source", Record, RecordSeparators -> {{": "}, {" :"}}]
Out[19]= {function f, function g}
```

By choosing appropriate separators, you can pick out specific parts of files.

```
In[20]:= ReadList["ExampleData/source", Record,
      RecordSeparators -> {{": function ", "[", {" :)", "}]"}]}
Out[20]= {x, f, x, g}
```

*Mathematica* usually allows any number of appropriate separators to appear between successive records or words. Sometimes, however, when several separators are present, you may want to assume that a “null record” or “null word” appears between each pair of adjacent separators. You can do this by setting the options `NullRecords -> True` or `NullWords -> True`.

Here is a file containing “words” separated by colons.

```
In[21]:= FilePrint["ExampleData/words"]

      first:second::fourth:::seventh
```

Here the repeated colons are treated as single separators.

```
In[22]:= ReadList["ExampleData/words", Word, WordSeparators -> {":."}]
Out[22]= {first, second, fourth, seventh}
```

Now repeated colons are taken to have null words in between.

```
In[23]:= ReadList["ExampleData/words", Word, WordSeparators -> {":."}, NullWords -> True]
Out[23]= {first, second, , fourth, , , seventh}
```

In most cases, you want words to be delimited by separators which are not themselves considered as words. Sometimes, however, it is convenient to allow words to be delimited by special “token words”, which are themselves words. You can give a list of such token words as a setting for the option `TokenWords`.

Here is some text.

```
In[24]:= FilePrint["ExampleData/language"]
      22*a*b+56*c+13*a*d
```

This reads the text, using the specified token words to delimit words in the text.

```
In[25]:= ReadList["ExampleData/language", Word, TokenWords -> {"+", "*"}]
Out[25]= {22, *, a, *, b, +, 56, *, c, +, 13, *, a, *, d}
```

You can use `ReadList` to read *Mathematica* expressions from files. In general, each expression must end with a newline, although a single expression may go on for several lines.

Here is a file containing text that can be used as *Mathematica* input.

```
In[26]:= FilePrint["ExampleData/exprs"]
      x + y +
      z
      2^8
```

This reads the text in `exprs` as *Mathematica* expressions.

```
In[27]:= ReadList["ExampleData/exprs", Expression]
Out[27]= {x + y + z, 256}
```

This prevents the expressions from being evaluated.

```
In[28]:= ReadList["ExampleData/exprs", Hold[Expression]]
Out[28]= {Hold[x + y + z], Hold[2^8]}
```

`ReadList` can insert the objects it reads into any *Mathematica* expression. The second argument to `ReadList` can consist of any expression containing symbols such as `Number` and `Word` specifying objects to read. Thus, for example, `ReadList["file", {Number, Number}]` inserts successive pairs of numbers that it reads into lists. Similarly, `ReadList["file", Hold[Expression]]` puts expressions that it reads inside `Hold`.

If `ReadList` reaches the end of your file before it has finished reading a particular set of objects you have asked for, then it inserts the special symbol `EndOfFile` in place of the objects it has not yet read.

Here is a file of numbers.

```
In[29]:= FilePrint["ExampleData/numbers"]
```

```
11.1  22.2  33.3
44.4  55.5  66.6
```

The symbol `EndOfFile` appears in place of numbers that were needed after the end of the file was reached.

```
In[30]:= ReadList["ExampleData/numbers", {Number, Number, Number, Number}]
```

```
Out[30]= {{11.1, 22.2, 33.3, 44.4}, {55.5, 66.6, EndOfFile, EndOfFile}}
```

<code>ReadList["!command", type]</code>	execute a command, and read its output
<code>ReadList[stream, type]</code>	read any input stream

Reading from commands and streams.

This executes the Unix command `date`, and reads its output as a string.

```
In[31]:= ReadList["!date", String]
```

```
Out[31]= {Thu Mar 31 19:20:36 CST 2005}
```

<code>OpenRead["file"]</code>	open a file for reading
<code>OpenRead["!command"]</code>	open a pipe for reading
<code>Read[stream, type]</code>	read an object of the specified type from a stream
<code>Skip[stream, type]</code>	skip over an object of the specified type in an input stream
<code>Skip[stream, type, n]</code>	skip over <i>n</i> objects of the specified type in an input stream
<code>Close[stream]</code>	close an input stream

Functions for reading from input streams.

`ReadList` allows you to read *all* the data in a particular file or input stream. Sometimes, however, you want to get data a piece at a time, perhaps doing tests to find out what kind of data to expect next.

When you read individual pieces of data from a file, *Mathematica* always remembers the “current point” that you are at in the file. When you call `OpenRead`, *Mathematica* sets up an input stream from a file, and makes your current point the beginning of the file. Every time you read an object from the file using `Read`, *Mathematica* sets your current point to be just after the object you have read. Using `Skip`, you can advance the current point past a sequence of objects without actually reading the objects.

Here is a file of numbers.

```
In[32]:= FilePrint["ExampleData/numbers"]
      11.1    22.2    33.3
      44.4    55.5    66.6
```

This opens an input stream from the file.

```
In[33]:= snum = OpenRead["ExampleData/numbers"]
Out[33]= InputStream[ExampleData/numbers, 66]
```

This reads the first number from the file.

```
In[34]:= Read[snum, Number]
Out[34]= 11.1
```

This reads the second pair of numbers.

```
In[35]:= Read[snum, {Number, Number}]
Out[35]= {22.2, 33.3}
```

This skips the next number.

```
In[36]:= Skip[snum, Number]
```

And this reads the remaining numbers.

```
In[37]:= ReadList[snum, Number]
Out[37]= {55.5, 66.6}
```

This closes the input stream.

```
In[38]:= Close[snum]
Out[38]= ExampleData/numbers
```

You can use the options `WordSeparators` and `RecordSeparators` in `Read` and `Skip` just as you do in `ReadList`.

Note that if you try to read past the end of file, `Read` returns the symbol `EndOfFile`.

## Searching Files

<code>FindList["file", "text"]</code>	get a list of all the lines in the file that contain the specified text
<code>FindList["file", "text", n]</code>	get a list of the first $n$ lines that contain the specified text
<code>FindList["file", {"text<sub>1</sub>", "text<sub>2</sub>", ...}]</code>	get lines that contain any of the $text_i$

Finding lines that contain specified text.

Here is a file containing some text.

```
In[1]:= FilePrint["ExampleData/textfile"]
```

```
Here is the first line of text.
And the second.
And the third. Here is the end.
```

This returns a list of all the lines in the file containing the text `is`.

```
In[2]:= FindList["ExampleData/textfile", "is"]
```

```
Out[2]= {Here is the first line of text., And the third. Here is the end.}
```

The text `fourth` appears nowhere in the file.

```
In[3]:= FindList["ExampleData/textfile", "fourth"]
```

```
Out[3]= {}
```

By default, `FindList` scans successive lines of a file, and returns those lines which contain the text you specify. In general, however, you can get `FindList` to scan successive *records*, and return complete records which contain specified text. As in `ReadList`, the option `RecordSeparators` allows you to tell *Mathematica* what strings you want to consider as record separators. Note that by giving a pair of lists as the setting for `RecordSeparators`, you can specify different left and right separators. By doing this, you can make `FindList` search only for text which is between specific pairs of separators.



This finds all "sentences" ending with a period which contain And.

```
In[4]:= FindList["ExampleData/textfile", "And", RecordSeparators -> {".."}]
Out[4]= {
  And the second,
  And the third}
```

<i>option name</i>	<i>default value</i>	
RecordSeparators	{"\n"}	separators for records
AnchoredSearch	False	whether to require the text searched for to be at the beginning of a record
WordSeparators	{" ", "\t"}	separators for words
WordSearch	False	whether to require that the text searched for appear as a word
IgnoreCase	False	whether to treat lowercase and uppercase letters as equivalent

Options for FindList.

This finds only the occurrence of Here which is at the beginning of a line in the file.

```
In[5]:= FindList["ExampleData/textfile", "Here", AnchoredSearch -> True]
Out[5]= {Here is the first line of text.}
```

In general, FindList finds text that appears anywhere inside a record. By setting the option WordSearch -> True, however, you can tell FindList to require that the text it is looking for appears as a separate *word* in the record. The option WordSeparators specifies the list of separators for words.

The text th does appear in the file, but not as a word. As a result, the FindList fails.

```
In[6]:= FindList["ExampleData/textfile", "th", WordSearch -> True]
Out[6]= {}
```

```
FindList [{"file1", "file2", ...}, "text"]
```

search for occurrences of the text in any of the *file<sub>i</sub>*

Searching in multiple files.

This searches for third in two copies of textfile.

```
In[7]:= FindList[{"ExampleData/textfile", "ExampleData/textfile"}, "third"]
Out[7]= {And the third. Here is the end., And the third. Here is the end.}
```

It is often useful to call `FindList` on lists of files generated by functions such as `FileNames`.

<code>FindList ["!command", ...]</code>	run an external command, and find text in its output
---	--

Finding text in the output from an external program.

This runs the external Unix command `date` in a text-based interface.

```
In[8]:= ! date
```

```
Thu Mar 31 19:20:36 CST 2006
```

```
Out[8]= 0
```

This finds the time-of-day field in the date.

```
In[9]:= FindList["!date", ":", RecordSeparators -> {" "}]
```

```
Out[9]= {19:20:36}
```

<code>OpenRead ["file"]</code>	open a file for reading
<code>OpenRead ["!command"]</code>	open a pipe for reading
<code>Find [stream, text]</code>	find the next occurrence of <i>text</i>
<code>Close [stream]</code>	close an input stream

Finding successive occurrences of text.

`FindList` works by making one pass through a particular file, looking for occurrences of the text you specify. Sometimes, however, you may want to search incrementally for successive occurrences of a piece of text. You can do this using `Find`.

In order to use `Find`, you first explicitly have to open an input stream using `OpenRead`. Then, every time you call `Find` on this stream, it will search for the text you specify, and make the current point in the file be just after the record it finds. As a result, you can call `Find` several times to find successive pieces of text.

This opens an input stream for `textfile`.

```
In[10]:= stext = OpenRead["ExampleData/textfile"]
```

```
Out[10]= InputStream[ExampleData/textfile, 76]
```

This finds the first line containing `And`.

```
In[11]:= Find[stext, "And"]
```

```
Out[11]= And the second.
```

Calling `Find` again gives you the next line containing `And`.

```
In[12]:= Find[stext, "And"]
Out[12]= And the third. Here is the end.
```

This closes the input stream.

```
In[13]:= Close[stext]
Out[13]= ExampleData/textfile
```

Once you have an input stream, you can mix calls to `Find`, `Skip` and `Read`. If you ever call `FindList` or `ReadList`, *Mathematica* will immediately read to the end of the input stream.

This opens the input stream.

```
In[14]:= stext = OpenRead["ExampleData/textfile"]
Out[14]= InputStream[ExampleData/textfile, 77]
```

This finds the first line which contains `second`, and leaves the current point in the file at the beginning of the next line.

```
In[15]:= Find[stext, "second"]
Out[15]= And the second.
```

`Read` can then read the word that appears at the beginning of the line.

```
In[16]:= Read[stext, Word]
Out[16]= And
```

This skips over the next three words.

```
In[17]:= Skip[stext, Word, 3]
```

*Mathematica* finds `is` in the remaining text, and prints the entire record as output.

```
In[18]:= Find[stext, "is"]
Out[18]= And the third. Here is the end.
```

This closes the input stream.

```
In[19]:= Close[stext]
Out[19]= ExampleData/textfile
```

<code>StreamPosition[stream]</code>	find the position of the current point in an open stream
<code>SetStreamPosition[stream,n]</code>	set the position of the current point
<code>SetStreamPosition[stream,0]</code>	set the current point to the beginning of a stream
<code>SetStreamPosition[stream,Infinity]</code>	set the current point to the end of a stream

Finding and setting the current point in a stream.

Functions like `Read`, `Skip` and `Find` usually operate on streams in an entirely sequential fashion. Each time one of the functions is called, the current point in the stream moves on.

Sometimes, you may need to know where the current point in a stream is, and be able to reset it. On most computer systems, `StreamPosition` returns the position of the current point as an integer giving the number of bytes from the beginning of the stream.

This opens the stream.

```
In[20]:= stext = OpenRead["ExampleData/textfile"]
Out[20]= InputStream[ExampleData/textfile, 78]
```

When you first open the file, the current point is at the beginning, and `StreamPosition` returns 0.

```
In[21]:= StreamPosition[stext]
Out[21]= 0
```

This reads the first line in the file.

```
In[22]:= Read[stext, Record]
Out[22]= Here is the first line of text.
```

Now the current point has advanced.

```
In[23]:= StreamPosition[stext]
Out[23]= 31
```

This sets the stream position back.

```
In[24]:= SetStreamPosition[stext, 5]
Out[24]= 5
```

Now `Read` returns the remainder of the first line.

```
In[25]:= Read[stext, Record]
Out[25]= is the first line of text.
```

This closes the stream.

```
In[26]:= Close[stext]
Out[26]= ExampleData/textfile
```

## Searching and Reading Strings

Functions like `Read` and `Find` are most often used for processing text and data from external files. In some cases, however, you may find it convenient to use these same functions to process strings within *Mathematica*. You can do this by using the function `StringToStream`, which opens an input stream that takes characters not from an external file, but instead from a *Mathematica* string.

<code>StringToStream ["string"]</code>	open an input stream for reading from a string
<code>Close [stream]</code>	close an input stream

Treating strings as input streams.

This opens an input stream for reading from the string.

```
In[1]:= str = StringToStream["A string of words."]
Out[1]= InputStream[String, 27]
```

This reads the first "word" from the string.

```
In[2]:= Read[str, Word]
Out[2]= A
```

This reads the remaining words from the string.

```
In[3]:= ReadList[str, Word]
Out[3]= {string, of, words.}
```

This closes the input stream.

```
In[4]:= Close[str]
Out[4]= String
```

Input streams associated with strings work just like those with files. At any given time, there is a current position in the stream, which advances when you use functions like `Read`. The current position is given as the number of characters from the beginning of the string by the function `StreamPosition[stream]`. You can explicitly set the current position using `SetStreamPosition[stream, n]`.

Here is an input stream associated with a string.

```
In[5]:= str = StringToStream["123 456 789"]
Out[5]= InputStream[String, 28]
```

The current position is initially 0 characters from the beginning of the string.

```
In[6]:= StreamPosition[str]
Out[6]= 0
```

This reads a number from the stream.

```
In[7]:= Read[str, Number]
Out[7]= 123
```

The current position is now 3 characters from the beginning of the string.

```
In[8]:= StreamPosition[str]
Out[8]= 3
```

This sets the current position to be 1 character from the beginning of the string.

```
In[9]:= SetStreamPosition[str, 1]
Out[9]= 1
```

If you now read a number from the string, you get the 23 part of 123.

```
In[10]:= Read[str, Number]
Out[10]= 23
```

This sets the current position to the end of the string.

```
In[11]:= SetStreamPosition[str, Infinity]
Out[11]= 11
```

If you now try to read from the stream, you will always get EndOfFile.

```
In[12]:= Read[str, Number]
Out[12]= EndOfFile
```

This closes the stream.

```
In[13]:= Close[str]
Out[13]= String
```

Particularly when you are processing large volumes of textual data, it is common to read fairly long strings into *Mathematica*, then to use `StringToStream` to allow further processing of these strings within *Mathematica*. Once you have created an input stream using `StringToStream`, you can read and search the string using any of the functions discussed for files.

This puts the whole contents of `textfile` into a string.

```
In[14]:= s = First[ReadList["ExampleData/textfile", Record, RecordSeparators -> {}]]
Out[14]= Here is the first line of text.
        And the second.
        And the third. Here is the end.
```

This opens an input stream for the string.

```
In[15]:= str = StringToStream[s]
Out[15]= InputStream[String, 30]
```

This gives the lines of text in the string that contain `is`.

```
In[16]:= FindList[str, "is"]
Out[16]= {Here is the first line of text., And the third. Here is the end.}
```

This resets the current position back to the beginning of the string.

```
In[17]:= SetStreamPosition[str, 0]
Out[17]= 0
```

This finds the first occurrence of `the` in the string, and leaves the current point just after it.

```
In[18]:= Find[str, "the", RecordSeparators -> {" "}]
Out[18]= the
```

This reads the "word" which appears immediately after `the`.

```
In[19]:= Read[str, Word]
Out[19]= first
```

This closes the input stream.

```
In[20]:= Close[str]
Out[20]= String
```

## Binary Files

Functions like `Read` and `Write` handle ordinary printable text. But in dealing with external data files or devices it is sometimes necessary to go to a lower level, and work directly with raw binary data. You can do this using `BinaryRead` and `BinaryWrite`.

<code>BinaryRead [stream]</code>	read one byte
<code>BinaryRead [stream, type]</code>	read an object of the specified type
<code>BinaryRead [stream, {type<sub>1</sub>, type<sub>2</sub>, ...}]</code>	read a list of objects
<hr/>	
<code>BinaryWrite [stream, b]</code>	write one byte
<code>BinaryWrite [stream, {b<sub>1</sub>, b<sub>2</sub>, ...}]</code>	write a sequence of bytes
<code>BinaryWrite [stream, "string"]</code>	write the characters in a string
<code>BinaryWrite [stream, x, type]</code>	write an object of the specified type
<code>BinaryWrite [stream, {x<sub>1</sub>, x<sub>2</sub>, ...}, type]</code>	write a sequence of objects
<code>BinaryWrite [stream, {x<sub>1</sub>, x<sub>2</sub>, ...}, {type<sub>1</sub>, type<sub>2</sub>, ...}]</code>	write objects of different types

Reading and writing binary data.



"Byte"	8-bit unsigned integer
"Character8"	8-bit character
"Character16"	16-bit character
"Complex64"	IEEE single-precision complex number
"Complex128"	IEEE double-precision complex number
"Complex256"	IEEE quad-precision complex number
"Integer8"	8-bit signed integer
"Integer16"	16-bit signed integer
"Integer32"	32-bit signed integer
"Integer64"	64-bit signed integer
"Integer128"	128-bit signed integer
"Real32"	IEEE single-precision real number
"Real64"	IEEE double-precision real number
"Real128"	IEEE quad-precision real number
"TerminatedString"	null-terminated string of 8-bit characters
"UnsignedInteger8"	8-bit unsigned integer
"UnsignedInteger16"	16-bit unsigned integer
"UnsignedInteger32"	32-bit unsigned integer
"UnsignedInteger64"	64-bit unsigned integer
"UnsignedInteger128"	128-bit unsigned integer

Types supported in `BinaryRead` and `BinaryWrite`.

This writes a sequence of bytes to a file.

```
In[1]:= BinaryWrite["tmp", {97, 98, 99, 100, 101}]
Out[1]= tmp
```

`BinaryWrite` automatically opens a stream for the file. This closes it.

```
In[2]:= Close["tmp"];
```

This reads the first byte from the file, returning it as an integer.

```
In[3]:= BinaryRead["tmp"]
Out[3]= 97
```

This reads the second 8 bits in the file as a character.

```
In[4]:= BinaryRead["tmp", "Character8"]
Out[4]= b
```

This reads the next 32 bits as a 32-bit integer.

```
In[5]:= BinaryRead["tmp", "Integer32"]
Out[5]= EndOfFile
```

Like `Read` and `Write`, `BinaryRead` and `BinaryWrite` work with streams. But if you give a file name, they automatically open the specified file as a stream. To create a stream directly you can use `OpenRead` or `OpenWrite`. On some computer systems, the option setting `BinaryFormat -> True` is required for any stream to be used with `BinaryRead` and `BinaryWrite`, in order to prevent possible corruption from such issues as newline translation.

In using *Mathematica* you are normally completely insulated from the raw representation of data inside your computer. But with `BinaryRead` and `BinaryWrite` this is no longer so. One of the subtleties that then arises is that different computers may take the bytes that make up numbers to be in different orders, as specified by their setting for `$ByteOrdering`.

This writes a 32-bit integer to a file.

```
In[6]:= BinaryWrite["tmp2", 45 671, "Integer32"]
Out[6]= tmp2
```

This closes the file.

```
In[7]:= Close["tmp2"];
```

This reads the integer back, but assumes an opposite byte ordering.

```
In[8]:= BinaryRead["tmp2", "Integer32", ByteOrdering -> -$ByteOrdering]
Out[8]= 1 739 718 656
```

<code>BinaryReadList["file"]</code>	read all the bytes in a file
<code>BinaryReadList["file", type]</code>	read all the data, treating it as objects of a certain type
<code>BinaryReadList["file", {type<sub>1</sub>, type<sub>2</sub>, ...}]</code>	treat the data as objects of a sequence of types
<code>BinaryReadList["file", types, n]</code>	read only the first <i>n</i> objects

Reading complete binary files.

This writes out a 128-bit real number.

```
In[9]:= BinaryWrite["tmp3", 5.67891, "Real128"]
Out[9]= tmp3
```

This reads back the bytes in the number.

```
In[10]:= BinaryReadList["tmp3", "Byte"]
Out[10]= {0, 0, 0, 0, 0, 0, 0, 0, 224, 89, 187, 237, 66, 115, 107, 1, 64}
```

This reads back the bytes as a sequence of 32-bit real numbers.

```
In[11]:= BinaryReadList["tmp3", "Real32"]
Out[11]= {0., -3.68935×1019, 118.866, 2.02218}
```

This treats the data as pairs containing a byte and a 32-bit real.

```
In[12]:= BinaryReadList["tmp3", {"Byte", "Real32"}]
Out[12]= {{0, 0.}, {0, -0.00332451}, {237, 4.32454×10-38}, {64, EndOfFile}}
```

`BinaryRead` and `BinaryWrite` allow complete flexibility in reading and writing raw binary data. But in many practical applications one instead wants to work only with particular predefined formats. You can do this using `Import` and `Export`.

In addition to many complex formats, `Import` and `Export` support files containing sequences of identical data elements, of the same types as in `BinaryRead` and `BinaryWrite`. They also support the "Bit" format, consisting of individual binary bits, represented as 0 or 1.

## Generating C and Fortran Expressions

If you have special-purpose programs written in C or Fortran, you may want to take formulas you have generated in *Mathematica* and insert them into the source code of your programs. *Mathematica* allows you to convert mathematical expressions into C and Fortran expressions.

<code>CForm [expr]</code>	write out <i>expr</i> so it can be used in a C program
<code>FortranForm [expr]</code>	write out <i>expr</i> for Fortran

*Mathematica* output for programming languages.

Here is an expression, written out in standard *Mathematica* form.

```
In[1]:= Expand[(1 + x + y)^2]
Out[1]= 1 + 2 x + x^2 + 2 y + 2 x y + y^2
```

Here is the expression in Fortran form.

```
In[2]:= FortranForm[%]
Out[2]//FortranForm= 1 + 2*x + x**2 + 2*y + 2*x*y + y**2
```

Here is the same expression in C form. Macros for objects like `Power` are defined in the C header file `mdefs.h` that comes with most versions of *Mathematica*.

```
In[3]:= CForm[%]
Out[3]//CForm= 1 + 2*x + Power(x,2) + 2*y + 2*x*y + Power(y,2)
```

You should realize that there are many differences between *Mathematica* and C or Fortran. As a result, expressions you translate may not work exactly the same as they do in *Mathematica*. In addition, there are so many differences in programming constructs that no attempt is made to translate these automatically.

`Compile[x, expr]`

compile an expression into efficient internal code

A way to compile *Mathematica* expressions.

One of the common motivations for converting *Mathematica* expressions into C or Fortran is to try to make them faster to evaluate numerically. But the single most important reason that C and Fortran can potentially be more efficient than *Mathematica* is that in these languages one always specifies up front what type each variable one uses will be—integer, real number, array, and so on.

The *Mathematica* function `Compile` makes such assumptions within *Mathematica*, and generates highly efficient internal code. Usually this code runs not much if at all slower than custom C or Fortran.

## Splicing *Mathematica* Output into External Files

If you want to make use of *Mathematica* output in an external file such as a program or document, you will often find it useful to “splice” the output automatically into the file.

<code>Splice["file.mx"]</code>	splice <i>Mathematica</i> output into an external file named <i>file.mx</i> , putting the results in the file <i>file.x</i>
<code>Splice["infile", "outfile"]</code>	splice <i>Mathematica</i> output into <i>infile</i> , sending the output to <i>outfile</i>

Splicing *Mathematica* output into files.

The basic idea is to set up the definitions you need in a particular *Mathematica* session, then run `splice` to use the definitions you have made to produce the appropriate output to insert into the external files.

```
#include "mdefs.h"

double f(x)
double x;
{
double y;

y = <* Integrate[Sin[x]^5, x] *> ;

return(2*y - 1) ;
}
```

A simple C program containing a *Mathematica* formula.

```
#include "mdefs.h"

double f(x)
double x;
{
double y;

y = -5*Cos(x)/8 + 5*Cos(3*x)/48 - Cos(5*x)/80 ;

return(2*y - 1) ;
}
```

The C program after processing with `Splice`.

# Importing and Exporting

## Importing and Exporting Data

<code>Import["file", "Table"]</code>	import a table of data from a file
<code>Export["file", list, "Table"]</code>	export <i>list</i> to a file as a table of data

Importing and exporting tabular data.

This exports an array of numbers to the file `out.dat`.

```
In[1]:= Export["out.dat", {{5.7, 4.3}, {-1.2, 7.8}}]
Out[1]= out.dat
```

Here are the contents of the file `out.dat`.

```
In[2]:= FilePrint["out.dat"]

5.7    4.3
-1.2   7.8
```

This imports the contents of `out.dat` as a table of data.

```
In[3]:= Import["out.dat", "Table"]
Out[3]= {{5.7, 4.3}, {-1.2, 7.8}}
```

`Import["file", "Table"]` will handle many kinds of tabular data, automatically deducing the details of the format whenever possible. `Export["file", list, "Table"]` writes out data separated by spaces, with numbers given in C or Fortran-like form, as in `2.3 E5` and so on.

<code>Import["name.ext"]</code>	import data assuming a format deduced from the file name
<code>Export["name.ext", expr]</code>	export data in a format deduced from the file name

Importing and exporting general data.

table formats	"CSV", "TSV", "XLS"
matrix formats	"HarwellBoeing", "MAT", "MTX"
specialized data formats	"DIF", "FITS", "HDF5", "MPS", "SDTS", etc.

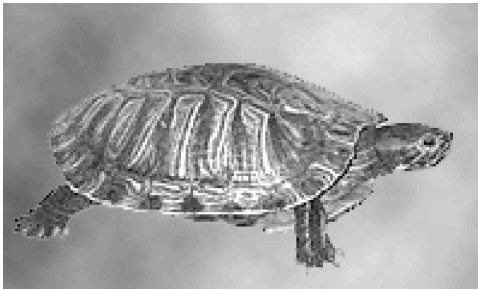
Some common formats for tabular data.

`Import` and `Export` can handle not only tabular data, but also data corresponding to graphics, sounds, expressions and even whole documents. `Import` and `Export` can often deduce the appropriate format for data simply by looking at the extension of the file name for the file in which the data is being stored. "Exporting Graphics and Sounds" and "Importing and Exporting Files" discuss in more detail how `Import` and `Export` work. Note that you can also use `Import` and `Export` to manipulate raw files of binary data.

This imports a graphic in JPEG format.

```
In[4]:= Import["ExampleData/turtle.jpg"]
```

Out[4]=



<code>\$ImportFormats</code>	import formats supported on your system
<code>\$ExportFormats</code>	export formats supported on your system

Finding the complete list of supported import and export formats.

## Importing and Exporting Files

<code>Import["file", "List"]</code>	import a one-dimensional list of data from a file
<code>Export["file", list, "List"]</code>	export <i>list</i> to a file as a one-dimensional list of data
<code>Import["file", "Table"]</code>	import a two-dimensional table of data from a file
<code>Export["file", list, "Table"]</code>	export <i>list</i> to a file as a two-dimensional table of data
<code>Import["file", "CSV"]</code>	import data in comma-separated format
<code>Export["file", list, "CSV"]</code>	export data in comma-separated format

Importing and exporting lists and tables of data.

This exports a list of data to the file `out1`.

```
In[1]:= Export["out1", {6.7, 8.5, -5.3}, "List"]
```

```
Out[1]= out1
```

Here are the contents of the file.

```
In[2]:= FilePrint["out1"]

      6.7
      8.5
     -5.3
```

This imports the contents back into *Mathematica*.

```
In[3]:= Import["out1", "List"]
Out[3]= {6.7, 8.5, -5.3}
```

If you want to use data purely within *Mathematica*, then the best way to keep it in a file is usually as a complete *Mathematica* expression, with all its structure preserved, as discussed in "Reading and Writing *Mathematica* Files: Files and Streams". But if you want to exchange data with other programs, it is often more convenient to have the data in a simple list or table format.

This exports a two-dimensional array of data.

```
In[4]:= Export["out2.dat", {{5.6 × 1012, 7.2 × 1012}, {3, 5}}, "Table"]
Out[4]= out2.dat
```

When necessary, numbers are written in C or Fortran-like "E" notation.

```
In[5]:= FilePrint["out2.dat"]

      5.6e12  7.2e12
      3      5
```

This imports the array back into *Mathematica*.

```
In[6]:= Import["out2.dat", "Table"]
Out[6]= {{5.6 × 1012, 7.2 × 1012}, {3, 5}}
```

If you have a file in which each line consists of a single number, then you can use `Import["file", "List"]` to import the contents of the file as a list of numbers. If each line consists of a sequence of numbers separated by tabs or spaces, then `Import["file", "Table"]` will yield a list of lists of numbers. If the file contains items that are not numbers, then these are returned as *Mathematica* strings.



This exports a mixture of textual and numerical data.

```
In[7]:= Export["out3.dat", {"first", 3.4}, {"second", 7.8}]
Out[7]= out3.dat
```

Here is the exported data.

```
In[8]:= FilePrint["out3.dat"]

      first    3.4
      second   7.8
```

This imports the data back into *Mathematica*.

```
In[9]:= Import["out3.dat", "Table"]
Out[9]= {{first, 3.4}, {second, 7.8}}
```

With `InputForm`, you can explicitly see the strings.

```
In[10]:= InputForm[%]
Out[10]//InputForm= {"first", 3.4}, {"second", 7.8}
```

<code>Import["file", "List"]</code>	treat each line as a separate numerical or other data item
<code>Import["file", "Table"]</code>	treat each element on each line as a separate numerical or other data item
<code>Import["file", "String"]</code>	treat the whole file as a single character string
<code>Import["file", "Text"]</code>	treat the whole file as a single string of text
<code>Import["file", {"Text", "Lines"}]</code>	treat each line as a string of text
<code>Import["file", {"Text", "Words"}]</code>	treat each separated word as a string of text

Importing files in different formats.

This creates a file with two lines of text.

```
In[11]:= Export["out4.txt", {"The first line.", "The second line."}, {"Text", "Lines"}]
Out[11]= out4.txt
```

Here are the contents of the file.

```
In[12]:= FilePrint["out4.txt"]

      The first line.
      The second line.
```

This imports the whole file as a single string.

```
In[13]:= Import["out4.txt", "Text"] // InputForm
Out[13]//InputForm= "The first line.\nThe second line."
```

This imports the file as a list of lines of text.

```
In[14]:= Import["out4.txt", {"Text", "Lines"}] // InputForm
Out[14]//InputForm= {"The first line.", "The second line."}
```

This imports the file as a list of words separated by white space.

```
In[15]:= Import["out4.txt", {"Text", "Words"}] // InputForm
Out[15]//InputForm= {"The", "first", "line.", "The", "second", "line."}
```

## Exporting Graphics and Sounds

*Mathematica* allows you to export graphics and sounds in a wide variety of formats. If you use the notebook front end for *Mathematica*, then you can typically just copy and paste graphics and sounds directly into other programs using the standard mechanism available on your computer system.

<code>Export["name.ext", graphics]</code>	export graphics to a file in a format deduced from the file name
<code>Export["file", graphics, "format"]</code>	export graphics in the specified format
<code>Export["!command", graphics, "format"]</code>	export graphics to an external command
<code>Export["file", {g<sub>1</sub>, g<sub>2</sub>, ...}, ...]</code>	export a sequence of graphics for an animation
<code>ExportString[graphics, "format"]</code>	generate a string representation of exported graphics

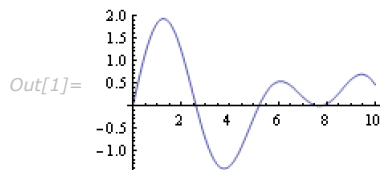
Exporting *Mathematica* graphics and sounds.

"EPS"	Encapsulated PostScript (.eps)
"PDF"	Adobe Acrobat portable document format (.pdf)
"SVG"	Scalable Vector Graphics (.svg)
"PICT"	Macintosh PICT
"WMF"	Windows metafile format (.wmf)
<hr/>	
"TIFF"	TIFF (.tif, .tiff)
"GIF"	GIF and animated GIF (.gif)
"JPEG"	JPEG (.jpg, .jpeg)
"PNG"	PNG format (.png)
"BMP"	Microsoft bitmap format (.bmp)
"PCX"	PCX format (.pcx)
"XBM"	X window system bitmap (.xbm)
"PBM"	portable bitmap format (.pbm)
"PPM"	portable pixmap format (.ppm)
"PGM"	portable graymap format (.pgm)
"PNM"	portable anymap format (.pnm)
"DICOM"	DICOM medical imaging format (.dcm, .dic)
"AVI"	Audio Video Interleave format (.avi)

Typical graphics formats supported by *Mathematica*. Formats in the first group are resolution independent.

This generates a plot.

```
In[1]:= Plot[Sin[x] + Sin[Sqrt[2] x], {x, 0, 10}]
```



This exports the plot to a file in Encapsulated PostScript format.

```
In[2]:= Export["sinplot.eps", %]
```

```
Out[2]= sinplot.eps
```

When you export a graphic outside of *Mathematica*, you usually have to specify the absolute size at which the graphic should be rendered. You can do this using the `ImageSize` option to `Export`.

`ImageSize -> x` makes the width of the graphic be  $x$  printer's points; `ImageSize -> 72 xi` thus makes the width  $xi$  inches. The default is to produce an image that is four inches wide. `ImageSize -> {x, y}` scales the graphic so that it fits in an  $x \times y$  region.

<code>ImageSize</code>	<code>Automatic</code>	absolute image size in printer's points
<code>"ImageTopOrientation"</code>	<code>Top</code>	how the image is oriented in the file
<code>ImageResolution</code>	<code>Automatic</code>	resolution in dpi for the image

Options for `Export`.

Within *Mathematica*, graphics are manipulated in a way that is completely independent of the resolution of the computer screen or other output device on which the graphics will eventually be rendered.

Many programs and devices accept graphics in resolution-independent formats such as Encapsulated PostScript (EPS). But some require that the graphics be converted to rasters or bitmaps with a specific resolution. The `ImageResolution` option for `Export` allows you to determine what resolution in dots per inch (dpi) should be used. The lower you set this resolution, the lower the quality of the image you will get, but also the less memory the image will take to store. For screen display, typical resolutions are 72 dpi and above; for printers, 300 dpi and above.

<code>"DXF"</code>	AutoCAD drawing interchange format ( <code>.dxf</code> )
<code>"STL"</code>	STL stereolithography format ( <code>.stl</code> )

Typical 3D geometry formats supported by *Mathematica*.

<code>"WAV"</code>	Microsoft wave format ( <code>.wav</code> )
<code>"AU"</code>	$\mu$ law encoding ( <code>.au</code> )
<code>"SND"</code>	sound file format ( <code>.snd</code> )
<code>"AIFF"</code>	AIFF format ( <code>.aif</code> , <code>.aiff</code> )

Typical sound formats supported by *Mathematica*.

## Generating and Importing TeX

*Mathematica* notebooks provide a sophisticated environment for creating technical documents. But particularly if you want to merge your work with existing material in TeX, you may find it convenient to use `TeXForm` to convert expressions in *Mathematica* into a form suitable for input to TeX.

<code>TeXForm [expr]</code>	print <i>expr</i> in TeX input form
-----------------------------	-------------------------------------

*Mathematica* output for TeX.

Here is an expression, printed in standard *Mathematica* form.

```
In[1]:= (x + y) ^ 2 / Sqrt[x y]
```

```
Out[1]= 
$$\frac{(x + y)^2}{\sqrt{xy}}$$

```

Here is the expression in TeX input form.

```
In[2]:= TeXForm[%]
```

```
Out[2]//TeXForm= \frac{(x+y)^2}{\sqrt{x y}}
```

<code>ToExpression ["input", TeXForm]</code>	convert TeX input to <i>Mathematica</i>
--	---

Converting TeX strings to *Mathematica*.

This converts a TeX string to *Mathematica*. Note the double backslashes needed in the string.

```
In[3]:= ToExpression["\\sqrt{x y}", TeXForm]
```

```
Out[3]= 
$$\sqrt{xy}$$

```

In addition to being able to convert individual expressions to TeX, *Mathematica* also provides capabilities for translating complete notebooks. These capabilities can usually be accessed from the **File ► Save As...** menu in the notebook front end.

## Exchanging Material with the Web

`Export["file.html", nb]` save the notebook *nb* in HTML form

Converting notebooks to HTML.

`Export` has many options applying to HTML export that allow you to specify how notebooks should be converted for web browsers with different capabilities.

`MathMLForm[expr]` print *expr* in MathML form  
`MathMLForm[StandardForm[expr]]` use `StandardForm` rather than traditional mathematical notation  
`ToExpression["string", MathMLForm]` interpret a string of MathML as *Mathematica* input

Converting to and from MathML.

Here is an expression printed in MathML form.

```
In[1]:= MathMLForm[x^2 / z]
Out[1]//MathMLForm= <math>
  <mfrac>
    <msup>
      <mi>x</mi>
      <mn>2</mn>
    </msup>
    <mi>z</mi>
  </mfrac>
</math>
```

If you paste MathML into a *Mathematica* notebook, *Mathematica* will automatically try to convert it to *Mathematica* input. You can copy an expression from a notebook as MathML using the **Copy As** menu in the notebook front end.

`Export["file.xml", expr]` export in XML format  
`Import["file.xml"]` import from XML  
`ImportString["string", "XML"]` import data from a string of XML

XML importing and exporting.

Somewhat like *Mathematica* expressions, XML is a general format for representing data. *Mathematica* automatically converts certain types of expressions to and from specific types of XML. MathML is one example. Another example is SVG for graphics.

If you ask *Mathematica* to import a generic piece of XML, it will produce a *SymbolicXML* expression. Each XML element of the form `< elem attr = ' val ' > data < / elem >` is translated to a *Mathematica* *SymbolicXML* expression of the form `XMLElement["elem", {"attr" -> "val"}, {data}]`. Once you have imported a piece of XML as *SymbolicXML*, you can use *Mathematica*'s powerful symbolic programming capabilities to manipulate the expression you get. You can then use `Export` to export the result in XML form.

This generates a *SymbolicXML* expression, with an `XMLElement` representing the `a` element in the XML string.

```
In[2]:= ImportString["<a aa='va'>s</a>", "XML"]
Out[2]= XMLObject[Document][{}, XMLElement[a, {aa -> va}, {s}], {}]
```

There are now two nested levels in the *SymbolicXML*.

```
In[3]:= ImportString["<a><b bb='1'>ss</b><b bb='2'>ss</b></a>", "XML"]
Out[3]= XMLObject[Document][{}, XMLElement[a, {}, {XMLElement[b, {bb -> 1}, {ss}], XMLElement[b, {bb -> 2}, {ss}]}], {}]
```

This does a simple transformation on the *SymbolicXML*.

```
In[4]:= % /. "ss" -> XMLElement["c", {}, {"xx"}]
Out[4]= XMLObject[Document][{}, XMLElement[a, {}, {XMLElement[b, {bb -> 1}, {XMLElement[c, {}, {xx}]}]}, XMLElement[b, {bb -> 2}, {XMLElement[c, {}, {xx}]}]], {}]
```

This shows the result as an XML string.

```
In[5]:= ExportString[%, "XML"]
Out[5]= <a>
  <b bb='1'>
    <c>xx</c>
  </b>
  <b bb='2'>
    <c>xx</c>
  </b>
</a>
```

<code>Import["http://url", ...]</code>	import a file from any accessible URL
<code>Import["ftp://url", ...]</code>	import a file from an FTP server

Importing data from web sources.

This imports a picture from a website.

```
In[6]:= Import["http://reference.wolfram.com/mathematica/ExampleData/ocelot.jpg"]
```

# Image Processing

## Image Processing

*Mathematica* now provides built-in support for both programmatic and interactive image processing—fully integrated with *Mathematica*'s powerful mathematical and algorithmic capabilities. You can create and import images, manipulate them with built-in functions, apply linear and nonlinear filters to them, and visualize them in any number of ways.

### ***Image Creation and Representation***

Images can be created from numerical arrays, from *Mathematica* graphics via cut-and-paste methods, and from external sources via `Import`.


<code>Image [data]</code>	raster image with pixel values given by <i>data</i>
<code>Import ["file"]</code>	import data from a file

Image creation functions.

The simplest method for creating an image object is to wrap `Image` around a matrix of real values ranging from 0 to 1.

Here is a one-channel image created from a matrix of numbers.

```
In[1]:= Image[{{0., 1., 0.}, {1., 0., 1.}, {0., 1., 0.}}
```

```
Out[1]= 
```

You can also copy and paste or drag and drop an image from other applications. You can use `Import` to obtain an image from a file on the local file system or any accessible remote location.



This imports an image from the *Mathematica* documentation directory `ExampleData`.

```
In[10]:= i = Import["ExampleData/ocelot.jpg"]
```



Out[10]=

Useful properties of an image can be obtained by calling the following functions.

<code>ImageDimensions [image]</code>	give the pixel dimensions of the raster associated with <i>image</i>
<code>ImageChannels [image]</code>	give the number of channels present in the data for <i>image</i>
<code>ImageType [image]</code>	give the type of values used for each pixel element in <i>image</i>
<code>ImageQ [image]</code>	give True if <i>image</i> has the form of a valid Image object and False otherwise
<code>Options [symbol]</code>	give the list of default options assigned to a symbol
<code>ImageData [image]</code>	the array of pixel values in <i>image</i>

Image properties.

This returns the image dimensions.

```
In[11]:= ImageDimensions[i]
```

```
Out[11]= {200, 200}
```

Here is the setting of the `ColorSpace` option.

```
In[12]:= Options[i, ColorSpace]
```

```
Out[12]= {ColorSpace -> Grayscale}
```

The image's array of pixel values can be easily extracted using the function `ImageData`. By default, the function returns real values, but you can ask for a specific type using the optional "type" argument.

This returns a fragment of the image as a matrix of real values scaled to the range 0 to 1.

```
In[14]:= ImageData[ImageTake[i, {94, 97}, {54, 59}]] // MatrixForm
```

$$\text{Out[14]//MatrixForm} = \begin{pmatrix} 0.772549 & 0.392157 & 0.0627451 & 0.203922 & 0.352941 & 0.372549 \\ 0.560784 & 0. & 0.164706 & 0.415686 & 0.415686 & 0.458824 \\ 0.278431 & 0.0352941 & 0.286275 & 0.435294 & 0.427451 & 0.368627 \\ 0.184314 & 0.0666667 & 0.32549 & 0.443137 & 0.54902 & 0.701961 \end{pmatrix}$$

Here is the same fragment as a matrix of integers in the range 0 to 255.

```
In[13]:= ImageData[ImageTake[i, {94, 97}, {54, 59}], "Byte"] // MatrixForm
```

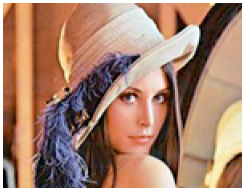
$$\text{Out[13]//MatrixForm} = \begin{pmatrix} 197 & 100 & 16 & 52 & 90 & 95 \\ 143 & 0 & 42 & 106 & 106 & 117 \\ 71 & 9 & 73 & 111 & 109 & 94 \\ 47 & 17 & 83 & 113 & 140 & 179 \end{pmatrix}$$

In the case of multichannel images, the raw pixel data is represented by a 3D array arranged in one of two possible ways as determined by the option `Interleaving`.

This imports a color image.

```
In[1]:= i = Import["ExampleData/lena.tif"]
```

Out[1]=



With the default setting `Interleaving -> True`, the data is organized as a 2D array of lists of color values, a triplet in the common case of images in RGB color space.

This shows the default data organization.

```
In[22]:= MatrixForm@ImageData[ImageTake[i, {90, 93}, {50, 53}], "Byte"]
```

$$\text{Out[22]//MatrixForm} = \begin{pmatrix} \begin{pmatrix} 30 \\ 24 \\ 30 \end{pmatrix} & \begin{pmatrix} 35 \\ 22 \\ 23 \end{pmatrix} & \begin{pmatrix} 37 \\ 24 \\ 16 \end{pmatrix} & \begin{pmatrix} 43 \\ 27 \\ 19 \end{pmatrix} \\ \begin{pmatrix} 33 \\ 28 \\ 33 \end{pmatrix} & \begin{pmatrix} 34 \\ 24 \\ 23 \end{pmatrix} & \begin{pmatrix} 40 \\ 26 \\ 18 \end{pmatrix} & \begin{pmatrix} 49 \\ 33 \\ 22 \end{pmatrix} \\ \begin{pmatrix} 31 \\ 28 \\ 32 \end{pmatrix} & \begin{pmatrix} 32 \\ 23 \\ 22 \end{pmatrix} & \begin{pmatrix} 38 \\ 24 \\ 16 \end{pmatrix} & \begin{pmatrix} 53 \\ 37 \\ 24 \end{pmatrix} \\ \begin{pmatrix} 33 \\ 31 \\ 35 \end{pmatrix} & \begin{pmatrix} 34 \\ 27 \\ 25 \end{pmatrix} & \begin{pmatrix} 35 \\ 21 \\ 13 \end{pmatrix} & \begin{pmatrix} 55 \\ 38 \\ 27 \end{pmatrix} \end{pmatrix}$$

The option setting `Interleaving -> False` can be used to store and retrieve the raw data as a list of matrices, one for each of the color channels.

Here is a fragment of the example image arranged as a list of channel matrices.

```
In[23]:= MatrixForm /@
ImageData[ImageTake[i, {90, 93}, {50, 53}], "Byte", Interleaving -> False]
```

$$\text{Out[23]} = \left\{ \begin{pmatrix} 30 & 35 & 37 & 43 \\ 33 & 34 & 40 & 49 \\ 31 & 32 & 38 & 53 \\ 33 & 34 & 35 & 55 \end{pmatrix}, \begin{pmatrix} 24 & 22 & 24 & 27 \\ 28 & 24 & 26 & 33 \\ 28 & 23 & 24 & 37 \\ 31 & 27 & 21 & 38 \end{pmatrix}, \begin{pmatrix} 30 & 23 & 16 & 19 \\ 33 & 23 & 18 & 22 \\ 32 & 22 & 16 & 24 \\ 35 & 25 & 13 & 27 \end{pmatrix} \right\}$$

A multichannel image can be split into a list of single-channel images and, conversely, a multi-channel image can be created from any number of single-channel images.

This splits the example RGB color image into three grayscale images.

```
In[2]:= ColorSeparate[i]
```



```
In[3]:= First[Options[#, "ColorSpace"] & /@%]
```

```
Out[3]= {ColorSpace -> Grayscale}
```

## Basic Image Manipulation

Consider the image manipulation operations that change the image dimensions by cropping or padding. These operations serve a variety of useful purposes. Cropping allows you to create a new image from a selected portion of a larger one, while padding is typically used to extend an image at the borders to ensure uniform treatment of the border pixels in many image processing tasks.

<code>ImageTake [image, n]</code>	give an image consisting of the first $n$ rows of <i>image</i>
<code>ImageCrop [image]</code>	crop <i>image</i> by removing borders of uniform color
<code>ImagePad [image, m]</code>	pad <i>image</i> on all sides with $m$ background pixels

Image cropping and padding operations.

This selects the first 50 rows of the example image.

```
In[24]:= ImageTake[i, 50]
```

```
Out[24]=
```

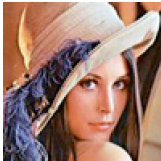


`ImageCrop` conveniently complements `ImageTake`. Instead of specifying the exact number of rows or columns to be extracted, it allows you to define the desired dimensions of the resulting image, namely, the number of rows or columns that are to be retained. By default, the cropping operation is centered, thus an equal number of rows and columns are deleted from the edges of the image.

Here a  $100 \times 100$  pixel region is extracted from the center of the example image.

```
In[27]:= ImageCrop[i, {100, 100}]
```

```
Out[27]=
```

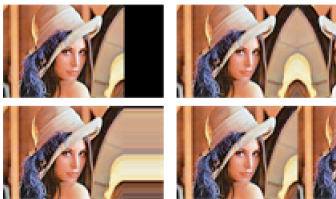


While `ImageCrop` is primarily used to reduce the dimensions of the source image, it is frequently desirable to pad an image to increase its dimensions. All the most common padding methods are supported.

This shows four different padding methods applied to the right edge of the example image.

```
In[33]:= Grid@Partition[
  ImagePad[i, {{0, 50}, {0, 0}}, #] & /@ {0, "Reflected", "Fixed", "Periodic"}, 2]
```

```
Out[33]=
```



It is frequently necessary to change the dimensions of an image by resampling or to reposition it in some manner. Functions that perform these basic geometric tasks are readily available.

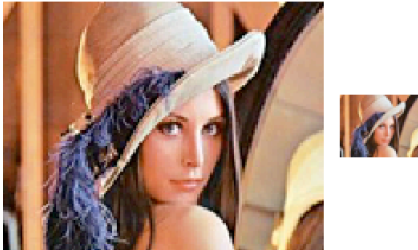
<code>ImageResize [image, w]</code>	give a resized version of <i>image</i> that is <i>w</i> pixels wide
<code>Thumbnail [image]</code>	give a thumbnail version of <i>image</i>
<code>ImageRotate [image]</code>	rotate <i>image</i> counterclockwise by 90°
<code>ImageReflect [image]</code>	reverse <i>image</i> by top-bottom mirror reflection

Spatial operations.

Here, `ImageResize` is used to increase and diminish the size of the original image, respectively.

```
In[38]:= Row@{ImageResize[i, 200], Spacer[10], ImageResize[i, 50]}
```

Out[38]=

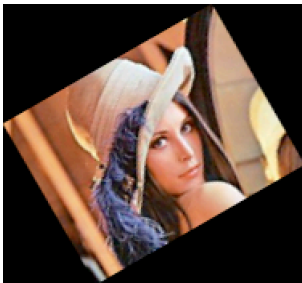


`ImageRotate` is another common spatial operation. It results in an image whose pixel positions are all rotated counter-clockwise with respect to a pivot point centered on the image.

This rotates the example image by 30 degrees.

```
In[39]:= ImageRotate[i,  $\pi / 6$ ]
```

Out[39]=




Several useful image processing tasks require nothing more than simple arithmetic operations between two images or an image and a constant. For example, you can change brightness by multiplying an image by a constant factor or by adding (subtracting) a constant to (from) an image. More interestingly, the difference of two images can be used to detect change and the product of two images can be used to hide or highlight regions in an image in a process called masking. For this purpose, three basic arithmetic functions are available.

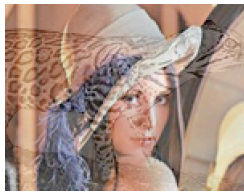
<code>ImageAdd [image, x]</code>	add an amount $x$ to each channel value in $image$
<code>ImageSubtract [image, x]</code>	subtract a constant amount $x$ from each channel value in $image$
<code>ImageMultiply [image, x]</code>	multiply each channel value in $image$ by a factor $x$

Arithmetic operations.

Here is an example of image blending using addition and multiplication.

```
In[17]:= ImageAdd[ImageMultiply[i, 2 / 3], ImageMultiply[, 1 / 3]]
```

```
Out[17]=
```



## Image Processing by Point Operations

Point operations constitute a simple but important class of image processing operations. These operations change the luminance values of an image and therefore modify how an image appears when displayed. The terminology originates from the fact that point operations take single pixels as inputs. This can be expressed as

$$g(i, j) = T[f(i, j)]$$

where  $T$  is a grayscale transformation that specifies the mapping between the input image  $f$  and the result  $g$ , and  $i, j$  denotes the row, column index of the pixel. Point operations are a one-to-one mapping between the original (input) and modified (output) images according to some function defining the transformation  $T$ .

### Contrast Modification

Contrast modifying point operations frequently encountered in image processing include negation (grayscale or color), gamma correction, which is a power-law transformation, and linear or nonlinear contrast stretching.

<code>Lighter [image, ...]</code>	give a lighter version of an <i>image</i>
<code>Darker [image, ...]</code>	give a darker version of an <i>image</i>
<code>ColorNegate [image]</code>	give the negative of <i>image</i> , in which all colors have been negated
<code>ImageAdjust [image]</code>	adjust the levels in <i>image</i> , rescaling them to cover the range 0 to 1
<code>ImageApply [f, image]</code>	apply <i>f</i> to the list of channel values for each pixel in <i>image</i>

Selected point operators.

One of the simplest examples of a point transformation is negation. For a grayscale image  $f$ , the transformation is defined by

$$g(i, j) = 1 - f(i, j).$$

It is applied to every pixel in the source image. In the case of multichannel images, the same transformation is applied to each color value, of every pixel.

This show the original example image and its digital negative.

```
In[6]:= GraphicsRow[{i, ColorNegate[i]}, ImageSize -> Medium]
```

Out[6]=

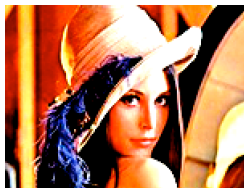


The function `ImageAdjust` can be used to perform most of the commonly needed contrast stretching and power-law transformations, while `ImageApply` enables you to realize any desired point transformation whatsoever.

This increases contrast using linear scaling.

```
In[37]:= ImageAdjust[i, 1.5]
```

Out[37]=



As an example of a nonlinear contrast stretching operation, consider the following transformation called sigma scaling. Assuming the default range of 0 to 1, the transformation is defined by

$$g(i, j) = \frac{1}{1 + e^{-\frac{f(i,j)-\mu}{\sigma}}}$$

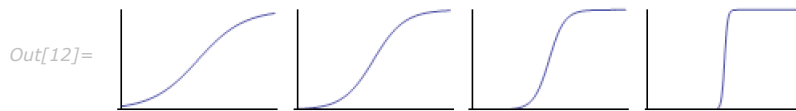
This defines the transformation.

```
In[10]:= f[x_, μ_, σ_] := 
$$\frac{1}{1 + e^{-\frac{x-\mu}{\sigma}}}$$

```

Here are several plots of the transformation for different values of the variance parameter.

```
In[12]:= GraphicsRow[Plot[f[x, 0.5, #], {x, 0, 1}, PlotRange -> {0, 1},  
  Ticks -> False, ImageSize -> Tiny] & /@ {0.15, 0.1, 0.05, 0.01}]
```



This shows the effect of the transformation on the example image.

```
In[36]:= ImageApply[f[#, 0.5, 0.1] &, i]
```



Image binarization is the operation of converting a multilevel image into a binary image. In a binary image, each pixel value is represented by a single binary digit. In its simplest form, binarization, also called thresholding, is a point-based operation that assigns the value of 0 or 1 to each pixel of an image based on a comparison with some global threshold value  $t$ .

$$g(i, j) = \begin{cases} 1, & \text{if } f(i, j) \geq t \\ 0, & \text{if } f(i, j) < t \end{cases}$$

Thresholding is an attractive early processing step because it leads to significant reduction in data storage and results in binary images that are simpler to analyze. Binary images permit the use of powerful morphological operators for shape and structure-based analysis of image content. Binarization is also a form of image segmentation, as it divides an image into distinct regions.



<code>Binarize [image]</code>	create a binary image from <i>image</i>
<code>ColorQuantize [image, n]</code>	give an approximation to <i>image</i> that uses only <i>n</i> distinct colors

Quantization functions.

Color images are first converted to grayscale prior to thresholding. If the threshold value is not explicitly given, an optimal value is calculated using one of several well-known methods.

Here is the default binarization based on Otsu's method for optimal threshold selection.

```
In[2]:= Binarize[i]
```



Here `ImageApply` is used to return a color image in which each individual channel is binarized, resulting in a maximum of 8 distinct colors.

```
In[17]:= ImageApply[UnitStep[# - 0.5] &, i]
```



## Color Conversion

Four color spaces are currently supported: RGB (red, green, and blue), CMYK (cyan, magenta, yellow, and black), HSB (hue, saturation, and brightness) and grayscale.

The RGB (red, green, blue) color scheme is the most frequently used color representation used in practice. The three so-called primary colors are combined (added) in various proportions to produce a composite, full-color image. The RGB color model is universally used in color moni-


tors and video recorders and cameras. Also, the human visual system is tuned to perceive color as a variable combination of these primary colors. The primary colors added in equal amounts produce the secondary colors of light: cyan (C), magenta (M), and yellow (Y). These are the primary pigment colors used in the printing industry and thus the relevance of the CMY color model. For image processing applications it is often useful to separate the color information from luminance. The HSB (hue, saturation, brightness) model has this property. Hue represents the dominant color as seen by an observer, saturation refers to the amount of dilution of the color with white light, and brightness defines the average luminance. The luminance component may, therefore, be processed independently of the image's color information.

<code>ColorConvert[<i>expr</i>, <i>colspace</i>]</code>	convert color specifications in <i>expr</i> to refer to the color space represented by <i>colspace</i>
---	--

Color conversion function.

This shows the conversion results from an RGB source to the remaining supported color spaces.

```
In[38]:= i = Image[{{1, 0, 0}, {0, 1, 0}, {0, 0, 1}}, ColorSpace -> "RGB"]
```

```
Out[38]= 
```

```
In[39]:= Column[InputForm[ColorConvert[i, #]] & /@ {"CMYK", "HSB", "Grayscale"}]
```

```
Image[{{0., 1., 1., 0.}, {1., 0., 1., 0.}, {1., 1., 0., 0.}},  
"Real", ColorSpace -> "CMYK", Interleaving -> True]
```

```
Out[39]= Image[{{0., 1., 1.}, {0.3333333333333333, 1., 1.}, {0.6666666666666666, 1., 1.}},  
"Real", ColorSpace -> "HSB", Interleaving -> True]
```

```
Image[{{0.299, 0.587, 0.114}}, "Real", ColorSpace -> "Grayscale", Interleaving -> None]
```

Note that the RGB -> Grayscale transformation uses the weighting coefficients recommended for U.S. broadcast TV (NTSC) and later incorporated into the CCIR 601 standard for digital video.

## Image Histogram

An important concept common to many image enhancement operations is that of a histogram, which is simply a count (or relative frequency, if normalized) of the gray levels in the image. Analysis of the histogram gives useful information about image contrast. Image histograms are important in many areas of image processing, most notably compression, segmentation, and thresholding.

<code>ImageLevels [image]</code>	give a list of pixel values and counts for each channel in <i>image</i>
<code>ImageHistogram [image]</code>	plot a histogram of the pixel levels for each channel in <i>image</i>

Image histogram functions.

This shows two different histogram visualization methods.

```
In[3]:= GraphicsRow[{ImageHistogram[i], ImageHistogram[i, Appearance -> "Separated"]},
  ImageSize -> Medium]
```



## Image Processing by Area Operations

Most useful image processing operators are area based. Area based operations calculate a new pixel value based on the values in a local, typically small, neighborhood. This is usually implemented through a linear or nonlinear filtering operation with a finite-sized operator (i.e., a filter). Without loss of generality, consider a centered and symmetric  $3 \times 3$  neighborhood of the image pixel at position  $n, m$ , with value  $f[n, m]$ . A general area-based transformation can be expressed as

$$g[i, j] = T \left[ \begin{pmatrix} f[i-1, j-1] & f[i, j-1] & f[i+1, j-1] \\ f[i-1, j] & f[i, j] & f[i+1, j] \\ f[i-1, j+1] & f[i, j+1] & f[i+1, j+1] \end{pmatrix} \right]$$

where  $g$  is the output image resulting from applying transformation  $T$  to the  $3 \times 3$  centered neighborhoods of all the pixels in input image  $f$ . It should be noted that the spatial dimensions and geometry of the neighborhood are generally determined by the needs of the application. Examples of image processing region-based operations include noise reduction, edge detection, edge sharpening, image enhancement, segmentation, and more.

## Linear and Nonlinear Filtering

Linear image filtering using convolution is one of the most common methods of processing images. To achieve a desired result you must specify an appropriate filter. Tasks such as smoothing, sharpening, edge finding, zooming, and more are typical examples of image processing tasks that have convolution-based implementations. Other tasks, noise removal for example, are better accomplished using nonlinear processing techniques.

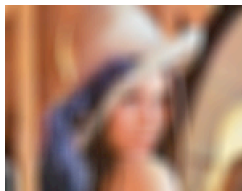
<code>ImageFilter [f, image, r]</code>	apply $f$ to the range $r$ of each pixel in each channel of $image$
<code>ImageConvolve [image, ker]</code>	give the convolution of $image$ with kernel $ker$

General filtering operators.

Here is a typical blurring operation using one of the smoothing filters.

```
In[4]:= ImageConvolve[i, BoxMatrix[5] / 121.]
```

Out[4]=

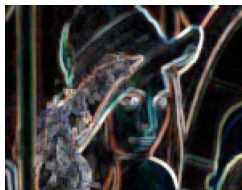


The more general (but slower) `ImageFilter` function can be used in cases when traditional linear filtering is not possible and the desired operation is not implemented by any of the built-in filtering functions.

This calculates the maximum range of values within a small neighborhood of each pixel.

```
In[5]:= ImageFilter[Max[Flatten[#]] - Min[Flatten[#]] &, i, 1]
```

Out[5]=



A large number of linear and nonlinear operators are available as built-in functions. Here is a partial listing.

<code>Blur [image]</code>	give a blurred version of <i>image</i>
<code>Sharpen [image]</code>	give a sharpened version of <i>image</i>
<code>MeanFilter [image,r]</code>	replace every value by the mean value in its range <i>r</i>
<code>GaussianFilter [image,r]</code>	convolve with a Gaussian kernel of pixel radius <i>r</i>
<code>MedianFilter [image,r]</code>	replace every value by the median in its range <i>r</i>
<code>MinFilter [image,r]</code>	replace every value by the minimum in its range <i>r</i>
<code>CommonestFilter [image,r]</code>	replace each pixel with the most common pixel value in its range <i>r</i>

Common linear and nonlinear filtering operators.

One of the more common applications of linear filtering in image processing has been in the computation of approximations of discrete derivatives and consequently edge detection. The well-known methods of Prewitt, Sobel, and Canny are all essentially based on the calculation of two orthogonal derivatives at each point in an image and the gradient magnitude.

Here are the two Sobel filters.

```
In[6]:= sobelY = {{1, 2, 1}, {0, 0, 0}, {-1, -2, -1}} / 4.;
sobelX = {{1, 0, -1}, {2, 0, -2}, {1, 0, -1}} / 4.;
```

This returns the edges of a grayscale image using Sobel filters.

```
In[7]:= Image[Sqrt[ImageData[ImageConvolve[ , sobelX]]^2 +
ImageData[ImageConvolve[ , sobelY]]^2]]
```

Out[7]=



As a second example, consider the task of removing the impulsive noise, which is called salt noise due to its visual appearance, from an image. This is a classic example contrasting the different outcomes resulting from a linear moving-average and a nonlinear moving-median calculation.

This creates a small image with impulsive noise.

```
In[13]:= Image[ReplacePart[ArrayPad[ConstantArray[160, {20, 20}], 15, 60],
  255, RandomInteger[{1, 50}, {100, 2}]], "Byte"]
```

Out[13]= 

Here is the side-by-side comparison.

```
In[14]:= Row[{MeanFilter[%, 1], Spacer[5], MedianFilter[%, 1]}]
```

Out[14]= 

Clearly, the median filter returns the better result.

## Morphological Processing

Mathematical morphology provides an approach to the processing of digital images that is based on the spatial structure of objects in a scene. In binary morphology, unlike linear and nonlinear operators discussed so far, morphological operators modify the shape of pixel groupings instead of their amplitude. However, in analogy with these operators, binary morphological operators may be implemented using convolution-like algorithms with the fundamental operations of addition and multiplication replaced by logical OR and AND.

`Dilation [image, r]`

give the dilation with respect to a range  $r$  square

`Erosion [image, r]`

give the erosion with respect to a range  $r$  square

Fundamental morphological operators.

This shows the dilation (left) and erosion (right) of the example image (center) using a 5×5 uniform structuring element.

```
In[8]:= b = Binarize[i];
GraphicsRow[{Dilation[b, 2], b, Erosion[b, 2]}, ImageSize -> Medium]
```



The definitions of binary morphology extend naturally to the domain of grayscale images with Boolean AND and OR becoming point-wise minimum and maximum operators, respectively. For a uniform, zero-valued structuring element, the dilation of an image  $f$  reduces to the following simple form:

$$g[i, j] = \text{Max} \left[ \begin{array}{ccc} f[i-1, j-1] & f[i, j-1] & f[i+1, j-1] \\ f[i-1, j] & f[i, j] & f[i+1, j] \\ f[i-1, j+1] & f[i, j+1] & f[i+1, j+1] \end{array} \right]$$

This shows the grayscale dilation (left) and erosion (right) of the example image (center) using a 5×5 uniform structuring element.

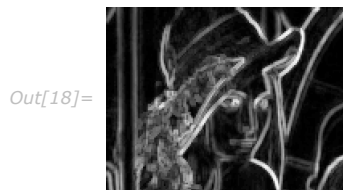
```
In[10]:= GraphicsRow[{Dilation[b, 2], b, Erosion[b, 2]}, ImageSize -> Medium]
```



These operators can be used in combinations using a single structuring element or a list of such elements to perform many useful image processing tasks. A partial listing includes thinning, thickening, edge and corner detection, and background normalization.

This uses dilation and erosion to detect edges in a grayscale image.

```
In[17]:= g = ColorConvert[i, "Grayscale"];
e = ImageSubtract[Dilation[g, 1], Erosion[g, 1]]
```



<code>GeodesicDilation[marker, mask]</code>	give the fixed point of the geodesic dilation of the image <i>marker</i> constrained by the image <i>mask</i>
<code>GeodesicErosion[marker, mask]</code>	give the fixed point of the geodesic erosion of the image <i>marker</i> constrained by the image <i>mask</i>
<code>DistanceTransform[image]</code>	give the distance transform of <i>image</i> , in which the value of each pixel is replaced by its distance to the nearest background pixel
<code>MorphologicalComponents[image]</code>	give an array in which each pixel of <i>image</i> is replaced by an integer index representing the connected foreground image component in which the pixel lies

Selected morphological functions.

An important category of morphological algorithms, called morphological reconstruction, are based on repeated application of dilation (or erosion) to a marker image, while the result of each step is constrained by a second image, the mask. The process ends when a fixed point is reached. Interestingly, many image processing tasks have a natural formulation in terms of reconstruction. Peak and valley detection, hole filling, region flooding, and hysteresis threshold are just a few examples. The latter, also known as a double threshold, is an integral part of the widely used Canny edge detector. Pixels falling below the low threshold are rejected, pixels above the high threshold are accepted, while pixels in the intermediate range are accepted only if they are "connected" to the high threshold pixels. Connectivity may be established using a variety of algorithms, but reconstruction gives an effective and very simple solution.

Here are the low, high, and double threshold images, respectively.

```
In[36]:= {mask = Binarize[e, 0.45], mark = Binarize[e, 0.8], GeodesicDilation[mask, mark]}
```



This clears all the symbols.

```
In[37]:= Clear[b, g, i, e, mask, mark];
```