

# Using the GNU text utilities

Skill Level: Introductory

[David Mertz \(mertz@gnosis.cx\)](mailto:mertz@gnosis.cx)

Developer

Gnosis Software, Inc.

09 Mar 2004

This introductory- to intermediate-level tutorial introduces the GNU text utilities and shows how to use them for processing log files, documentation, structured text databases, and other textual sources of data or content.

## Section 1. Before you start

### About this tutorial

This tutorial shows you how to use the GNU text utilities collection to process log files, documentation, structured text databases, and other textual sources of data or content. The utilities in this collection have proven their usefulness over decades of refinement by UNIX®/Linux® developers, and should be your first choice for general text-processing tasks.

This tutorial is written for UNIX/Linux programmers and system administrators, at a beginning to intermediate level.

### Prerequisites

For this tutorial, you should be generally familiar with some UNIX-like environment, and especially with a command-line shell. You need not be a programmer *per se*; in fact, the techniques described will be most useful to system administrators and users who process ad hoc reports, log files, project documentation, and the like (and less so for formal programming code processing). While working through this

tutorial, it is a good idea to keep a shell open and try the examples shown as well as variants on them.

Basic concepts are reviewed in the [Introduction: The UNIX philosophy](#) , where you can brush up on the basics of piping, streams, grep, and scripting.

---

## Section 2. Introduction: The UNIX philosophy

### Small utilities combine to do large tasks

In UNIX-inspired operating systems like Linux, FreeBSD, Mac OS X, Solaris, AIX, and so on, a common philosophy underlies the development environment, and even the shells and working environment. The gist of this philosophy is using small component utilities to do each small task *well* (and no other thing badly), then combining utilities to perform compound tasks. Most of what has been produced by the GNU project supports this component philosophy -- and indeed the specific GNU implementations have been ported to many platforms, even ones not traditionally thought of as UNIX-like. The Linux kernel, however, is necessarily a more monolithic bit of software -- though even there, kernel modules, filesystems, video drivers, and so on, are largely componentized.

For this tutorial, you should be generally familiar with some UNIX-like environment, and especially with a command-line shell. You need not be a programmer *per se*; in fact, the techniques described will be most useful to system administrators and users who process ad hoc reports, log files, project documentation, and the like (and less so for formal programming code processing). While working through this tutorial, it is a good idea to keep a shell open, and try the examples shown as well as variants on them.

### Files and streams

If the *UNIX philosophy* has a deontological aspect in advocating minimal modular components and cooperation, it also has an ontological aspect: "everything is a file." Abstractly, a *file* is simply an object that supports a few operations; firstly reading and writing bytes, but also some supporting operations like indicating its current position and knowing when it has reached its end. The UNIX permission model is also oriented around its idea of file.

Concretely, a file might be an actual region on a recordable media (with appropriate

tagging of its name, size, position on disk, and so on, supplied by the filesystem). But a file might also be a virtual device in the `/dev/` hierarchy, or a remote stream coming over a TCP/IP socket or via a higher-level protocol like NFS. Importantly, the special files `STDIN`, `STDOUT`, and `STDERR` can be used to read or write to the user console and/or to pass data between utilities. These special files can be indicated by virtual filenames, along with special syntax:

- `STDIN` is `/dev/stdin` and/or `/dev/fd/0`
- `STDOUT` is `/dev/stdout` and/or `/dev/fd/1`
- `STDERR` is `/dev/stderr` and/or `/dev/fd/2`

The advantage of UNIX' file ontology is that most of the utilities discussed here will handle various data sources uniformly and neutrally, regardless of what storage or transmission mechanisms actually underlie the delivery of bytes.

## Redirection and piping

The way that UNIX/Linux utilities are typically combined is via piping and redirection. Many utilities either automatically or optionally take their input from `STDIN`, and send their output to `STDOUT` (with special messages sent to `STDERR`). A pipe sends the `STDOUT` of one utility to the `STDIN` of another utility (or to a new invocation of the same utility). A redirect either reads the content of a file as `STDIN`, or sends the `STDOUT` and/or `STDERR` output to a named file. Redirects are often used to save data for later or repeated processing (with the later utility runs using `STDIN` redirection).

In almost all shells, piping is performed with the vertical-bar `|` symbol, and redirection with the greater-than and less-than symbols: `>` and `<`. To redirect `STDERR`, use `2>`, or `&>` to redirect both `STDOUT` and `STDERR` to the same place. You may also use a doubled greater-than (`>>`) to append to the end of an existing file. For example:

```
$ foo fname | bar - > myout 2> myerr
```

Here, the utility `foo` probably processes the file named `fname`, and outputs to `STDOUT`. The utility `bar` uses a common convention of specifying a dash when output is to be taken from `STDIN` rather than from a named file (some other utilities take only `STDIN`). The `STDOUT` from `bar` is saved in `myout`, and its `STDERR` in `myerr`.

## What are the text utilities?

The GNU text utilities is a collection of some of the tools for processing and manipulating text files and streams that have proven most useful, and been refined, over the evolution of UNIX-like operating systems. Most of them have been part of UNIX from the earliest implementations, though many have grown additional options over time.

The suite of utilities collected in the archive **textutils-2.1** includes twenty-seven tools; however, the GNU project maintainers have more recently decided to bundle these tools instead as part of the larger collection **coreutils-5.0** (and presumably likewise for later versions). On systems derived from BSD rather than GNU tools, the same utilities might be bundled a bit differently, but most of the same utilities will still be present.

This tutorial focuses on the twenty-seven utilities traditionally included in **textutils**, with some occasional mention and use of related tools that are generally available on UNIX-like systems. However, I will skip the utility `ptx` (permuted indexes), because it is both too narrow in purpose and too difficult to understand for inclusion here.

## grep (Generalized Regular Expression Processor)

One tool that is not *per se* part of **textutils** still deserves special mention. The utility `grep` is one of the most widely used UNIX utilities, and will very often be used in pipes to or from the text utilities.

What `grep` does is in one sense very simple, in another sense quite complex to understand. Basically, `grep` identifies lines in a file that match a regular expression. Some switches let you massage the output in various ways, such as printing surrounding context lines, numbering the matching lines, or identifying only the files in which the matches occur rather than individual lines. But at heart, `grep` is just a (very powerful) filter on the lines in a file.

The complex part of `grep` is the regular expressions you can specify to describe matches of interest. But that's another tutorial (see the link to the *developerWorks* tutorial "Using regular expressions" in [Resources](#)). A number of other utilities also support regular expression patterns, but `grep` is the most general such tool, and hence it is often easier to put `grep` in your pipeline than to use the weaker filters other tools provide. A quick `grep` example:

```
$ grep -c [Ss]ystem$ * 2> /dev/null | grep :[^0]$
INSTALL:1
aclocal.m4:2
config.log:1
configure:1
```

The example lists files that contain lines ending with word "system", perhaps with initial cap, at the end of lines; and also shows the number of such occurrences (that is, if non-zero). (Actually, the example does not handle counts greater than 9 properly).

## Shell scripting

While the text utilities are designed to produce outputs in various useful formats -- often modified by command-line switches -- there are still times when being able to explicitly branch and loop is useful. Shells like `bash` let you combine utilities with flow control to perform more complex chores. Shell scripts are especially useful to encapsulate compound tasks that you will perform multiple times, especially those involving some parameterization of the task.

Explaining `bash` scripting is certainly outside the scope of this tutorial. See [Resources](#) for an introduction to `bash` in the *developerWorks* "Bash by example" series. Once you understand the text utilities, it is fairly simple to combine them into saved shell scripts. Just for illustration, here is a quick (albeit somewhat contrived) example of flow control with `bash`:

```
[~/bacchus/articles/scratch/tu]$ cat flow
#!/bin/bash
for fname in `ls $1`; do
  if (grep $2 $fname > /dev/null); then
    echo "Creating: $fname.new" ;
    tr "abc" "ABC" < $fname > $fname.new
  fi
done
[~/bacchus/articles/scratch/tu]$ ./flow '*' bash
Creating: flow.new
Creating: test1.new
[~/bacchus/articles/scratch/tu]$ cat flow.new
#!/Bin/BAsh
for fnAme in `ls $1`; do
  if (grep $2 $fnAme > /dev/null); then
    eCho "CreAtIng: $fnAme.new" ;
    tr "ABC" "ABC" < $fnAme > $fnAme.new
  fi
done
```

This script capitalizes the letters a, b, and c in those files named by wildcard, and saves the changed version under new names.

---

## Section 3. Stream-oriented filtering

## cat and tac

The simplest text utilities simply output the exact contents of a file or stream -- or perhaps a portion or simple rearrangement of those contents -- to STDOUT.

The utility `cat` (conCATenate) begins with the first line and ends with the last line. The utility `tac` ("cat" backwards) outputs lines in reverse. Both utilities read every file specified as an argument, but default to STDIN if none is specified. As with many utilities, you may explicitly specify STDIN using the special name `-`. Some examples:

```
$ cat test2
Alice
Bob
Carol
$ tac < test3
Zeke
Yolanda
Xavier
$ cat test2 test3
Alice
Bob
Carol
Xavier
Yolanda
Zeke
$ cat test2 | tac - test3
Carol
Bob
Alice
Zeke
Yolanda
Xavier
```

## head and tail

The utilities `head` and `tail` output only an initial or final portion of a file or stream, respectively. The GNU version of both utilities supports the switch `-c` to output a number of bytes; most often both utilities are used in their line-oriented mode, which outputs a number of lines (whatever the actual line lengths). Both `head` and `tail` default to outputting ten lines. As with `cat` or `tac`, `head` and `tail` default to STDIN if files are not specified. Some examples:

```
$ head -c 8 test2 && echo # push prompt to new line
Alice
Bo
$ /usr/local/bin/head -2 test2
Alice
Bob
$ cat test3 | tail -n 2
```

```

Yolanda
Zeke
$ tail -r -n 2 test3 # reverse
Zeke
Yolanda

```

By the way, the GNU versions of these utilities (and many others) have more flexible switches than do the BSD versions.

The `tail` utility has a special mode -- indicated with the switches `-f` and `-F` -- that continues to display new lines written to the end of a "followed" file. The capitalized switch watches for truncations and renaming of the file as well as the simple appends the lower case switch monitors. Follow mode is particularly useful for watching changes to a log file that another process might perform periodically.

## od and hexdump

The utilities `od` and `hexdump` output octal, hex, or otherwise encoded bytes from a file or stream. These are useful for access to or visual examination of characters in a file that are not directly displayable on your terminal. For example, `cat` or `tail` do not directly disambiguate between tabs, spaces, or other whitespace -- you can check which characters are used with `hexdump`. Depending on your system type, either or both of these two utilities will be available -- BSD systems deprecate `od` for `hexdump`; GNU systems do the reverse. The two utilities, however, have exactly the same purpose, just slightly different switches.

```

$ od test3 # default output format
0000000 054141 073151 062562 005131 067554 060556 062141 005132
0000020 062553 062412
0000024
$ od -w8 -x test3 # 8 hex digits per line
0000000 5861 7669 6572 0a59
0000010 6f6c 616e 6461 0a5a
0000020 656b 650a
0000024
$ od -c test3 # 5 escaped ASCII chars per line
0000000 X a v i e r \n Y o l a n d a \n z
0000020 e k e \n
0000024

```

As with other utilities, `od` and `hexdump` accept input from STDIN or from one or more named files. Also, the `od` switches `-j` and `-N` let you skip initial bytes and limit the number read, respectively. You may customize output formats even further than with the standard switches using `fprintf()` -like formatting specifiers.

## HERE documents

A special kind of redirection is worth noting in this tutorial. While HERE documents are, strictly speaking, a feature of shells like `bash` rather than anything to do with the text utilities, they provide a useful way of sending ad hoc data to the text utilities (or to other applications).

Direction with a double less-than can be used to take pseudo-file contents from the terminal. A HERE document must specify a terminating delimiter immediately after its `<<`. For example:

```
$ od -c <<END
> Alice
> Bob
> END
0000000  A  l  i  c  e  \n  B  o  b  \n
0000012
```

Any string may be used as a delimiter; input is terminated when the string occurs on a line by itself. This gives us a quick way to create a persistent file:

```
$ cat > myfile <<EOF
> Dave
> Edna
> EOF
$ hexdump -C myfile
00000000  44  61  76  65  0a  45  64  6e  61  0a          |Dave.Edna.|
0000000a
```

---

## Section 4. Line-oriented filtering

### Lines as records

Many Linux utilities view files as a line-oriented collection of records or data. This provides a convenient way of aggregating data collections that is both readable to people and easy to process with tools. The simple trick is to treat each newline as a delimiter between records, where each record has a similar format.

As a practical matter, line-oriented records should usually have a relatively limited length, perhaps up through a few hundred characters. While none of the text utilities has such a limit built in to it, human eyes have trouble working with extremely long lines, even if auto-wrapping or horizontal scrolling is used. Either a more complex structured data format might be used in such cases, or records might be broken into

multiple lines (perhaps flagged for type in a way that `grep` can sort out). As a simple example, you might preserve a hierarchical multi-line data format using prefix characters:

```

$ cat multiline
A Alice Aaronson
B System Administrator
C 99 9th Street
A Bob Babuk
B Programmer
C 7 77th Avenue
$ grep '^A ' multiline # names
only
A Alice Aaronson
A Bob Babuk
$ grep '^C ' multiline # address
only
C 99 9th Street
C 7 77th Avenue

```

The output from one of these `grep` filters is a usable newline-delimited collection of partial records with the field(s) of interest.

## cut

The utility `cut` writes fields from a file to the standard output, where each line is treated as a delimited collection of fields. The default delimiting character is a tab, but this can be changed with the short form option `-d <DELIM>` or the long form option `--delimiter=<DELIM>`.

You may select one or more fields with the `-f` switch. The `-c` switch selects specific character positions from each line instead. Either switch accepts comma-separated numbers or ranges as parameters (including open ranges). For example, the file `employees` is tab-delimited:

```

$ cat employees
Alice Aaronson System Administrator 99 9th Street
Bob Babuk Programmer 7 77th Avenue
Carol Cavo Manager 111 West 1st Blvd.
$ hexdump -n 50 -c employees
00000000 A l i c e A a r o n s o n \t S
00000010 y s t e m A d m i n i s t r a
00000020 t o r \t 9 9 9 t h S t r e e
00000030 t \n
00000032
$ cut -f 1,3 employees
Alice Aaronson 99 9th Street
Bob Babuk 7 77th Avenue
Carol Cavo 111 West 1st Blvd.
$ cut -c 1-3,20,25- employees
Alieministrator 99 9th Street
Bobr7th Avenue
Carlest 1st Blvd.

```

Later examples will use custom delimiters other than tabs.

## expand and unexpand

The utilities `expand` and `unexpand` convert tabs to spaces and vice versa. A tab is considered to align at specific columns, by default every eight columns, so the specific number of spaces that correspond to a tab depends on where those spaces or tab occur. Unless you specify the `-a` option, `unexpand` will only entab the initial whitespace (this default is useful for reformatting source code).

Continuing with the `employees` file above, we can perform some substitutions. Notice that after you run `unexpand`, tabs in the output may be followed by some spaces in order to produce the needed overall alignment.

```
$ cat -T employees # show tabs explicitly
Alice Aaronson^ISystem Administrator^I99 9th Street
Bob Babuk^IProgrammer^I7 77th Avenue
Carol Cavo^IManager^I111 West 1st Blvd.
$ expand -25 employees
Alice Aaronson           System Administrator       99 9th Street
Bob Babuk                Programmer                 7 77th Avenue
Carol Cavo               Manager                    111 West 1st Blvd.
$ expand -25 employees | unexpand -a | hexdump -n 50 -c
00000000 A l i c e A a r o n s o n \t \t
00000010 S y s t e m A d m i n i s t
00000020 r a t o r \t 9 9 9 t h S
00000030 t r
00000032
```

## fold

The `fold` utility simply forces lines in a file to wrap. By default, wrapping is to 80 columns, but you may specify other widths. You get a limited sort of word-wrap formatting with `fold`, but it will not fully rewrap paragraphs. The option `-s` is useful for at least forcing new line breaks to occur on whitespace. Using a recent article of mine as a source (and clipping an example portion using tools we've seen earlier):

```
$ tail -4 rexx.txt | cut -c 3-
David Mertz' fondness for IBM dates back embarrassingly many decades.
David may be reached at mertz@gnosis.cx; his life pored over at
http://gnosis.cx/publish/. And buy his book: Text Processing in
Python (http://gnosis.cx/TPiP/).
$ tail -4 rexx.txt | cut -c 3- | fold -w 50
David Mertz' fondness for IBM dates back embarrass
ingly many decades.
David may be reached at mertz@gnosis.cx; his life
pored over at
```

```

http://gnosis.cx/publish/. And buy his book: _Text
  Processing in
Python_ (http://gnosis.cx/TPiP/).
$ tail -4 rexx.txt | cut -c 3- | fold -w 50 -s
David Mertz' fondness for IBM dates back
embarrassingly many decades.
David may be reached at mertz@gnosis.cx; his life
pored over at
http://gnosis.cx/publish/. And buy his book:
_Text Processing in
Python_ (http://gnosis.cx/TPiP/).

```

## fmt

For most purposes, `fmt` is a more useful tool for wrapping lines than is `fold`. The utility `fmt` will wrap lines, while both preserving initial indentation and aggregating lines for paragraph balance (as needed). `fmt` is useful for formatting documents such as e-mail messages before transmission or final storage.

```

$ tail -4 rexx.txt | fmt -40 -w50 # goal 40, max 50
David Mertz' fondness for IBM dates back
embarrassingly many decades. David may be
reached at mertz@gnosis.cx; his life pored
over at http://gnosis.cx/publish/. And
buy his book: _Text Processing in Python_
(http://gnosis.cx/TPiP/).
$ tail -4 rexx.txt | fold -40
David Mertz' fondness for IBM dates ba
ck embarrassingly many decades.
David may be reached at mertz@gnosis.c
x; his life pored over at
http://gnosis.cx/publish/. And buy his
book: _Text Processing in
Python_ (http://gnosis.cx/TPiP/).

```

The GNU version of `fmt` provides several options for indentation of first and subsequent lines. A useful option is `-u`, which normalizes word and sentence spaces (redundant runs of whitespace are removed).

## nl (and cat)

The utility `nl` numbers the lines in a file, with a variety of options for how numbers appear. `cat` contains the line numbering options you will need for most purposes -- choose the more general tool, `cat`, when it does what you need. Only in special cases such as controlling display of leading zeros is `nl` needed (historically, `cat` did not always include line numbering).

```

$ nl -w4 -nrz -ba rexx.txt | head -6 # width 4, zero padded
0001 LINUX ZONE FEATURE: Regina and NetRexx

```

```

0002   Scripting with Free Software Rexx implementations
0003
0004   David Mertz, Ph.D.
0005   Text Processor, Gnosis Software, Inc.
0006   January, 2004
$ cat -b rexx.txt | head -6 # don't number bare lines
  1  LINUX ZONE FEATURE: Regina and NetRexx
  2  Scripting with Free Software Rexx implementations

  3  David Mertz, Ph.D.
  4  Text Processor, Gnosis Software, Inc.
  5  January, 2004

```

Aside from making it easier to discuss lines by number, line numbers potentially provide sort or filter criteria for downstream processes.

## tr

The utility `tr` is a powerful tool for transforming the characters that occur within a file -- or rather, within STDIN, since `tr` operates exclusively on STDIN and writes exclusively to STDOUT (redirection and piping is allowed, of course).

`tr` has more limited capability than its big sibling `sed`, which is not included in the text utilities (nor in this tutorial) but is still almost always available on UNIX-like systems. Where `sed` can perform general replacements of regular expressions, `tr` is limited to replacing and deleting single characters (it has no real concept of context). At its most basic, `tr` replaces the characters of STDIN that are contained in a source string with those in a target string.

A simple example helps illustrate `tr`. We might have a file with variable numbers of tabs and spaces, and wish to normalize these separators, and replace them with a new delimiter. The trick is to use the `-s` (squeeze) flag to eliminate runs of the same character:

```

$ expand -26 employees | unexpand -a > empl.multitab
$ cat -T empl.multitab
Alice Aaronson^I^I System Administrator^I   99 9th Street
Bob Babuk^I^I Programmer^I^I   7 77th Avenue
Carol Cavo^I^I Manager^I^I   111 West 1st Blvd.
$ tr -s "\t " | " < empl.multitab | /usr/local/bin/cat -T
Alice Aaronson| System Administrator| 99 9th Street
Bob Babuk| Programmer| 7 77th Avenue
Carol Cavo| Manager| 111 West 1st Blvd.

```

In addition to translating explicitly listed characters, `tr` supports ranges and several named character classes. For example, to translate lower-case characters to upper-case, you may use either of:

```
$ tr "a-z" "A-Z" < employees
ALICE AARONSON  SYSTEM ADMINISTRATOR    99 9TH STREET
BOB BABUK      PROGRAMMER                7 77TH AVENUE
CAROL CAVO     MANAGER 111 WEST 1ST BLVD.
```

```
$ tr [:lower:] [:upper:] < employees
ALICE AARONSON  SYSTEM ADMINISTRATOR    99 9TH STREET
BOB BABUK      PROGRAMMER                7 77TH AVENUE
CAROL CAVO     MANAGER 111 WEST 1ST BLVD.
```

If the second range is not as long as the first, the second is padded with occurrences of its last character:

```
$ tr [:upper:] "a-l#" < employees
alice aaronson #system administrator    99 9th #treet
bob babuk      #rogrammer                7 77th avenue
carol cavo     #anager 111 #est 1st blvd.
```

Here, the script replaced all of the upper-case letters from the second half of the alphabet with "#."

You may also delete characters from the STDIN stream. Typically you might delete special characters like formfeeds or high-bit characters you want to filter. But for this, let's continue with the prior example:

```
$ tr -d [:lower:] < employees
A A    S A    99 9 S
B B    P     7 77 A
C C    M     111 W 1 B.
```

---

## Section 5. File-oriented filtering

### Working with line collections

The tools we have seen so far operate on each line individually. Another subset of the text utilities treats files as collections of lines, and performs some kind of global manipulation on those lines.

Pipes under UNIX-like operating systems can operate very efficiently in terms of

memory and latency. When a process earlier in a pipe produces a line to STDOUT, that line is immediately available to the next stage. However, the utilities below will not produce output until they have (mostly) completed their processing. For large files, some of these utilities can take a while to complete (but they are nonetheless all well optimized for the tasks they perform).

## sort

The utility `sort` does just what the name suggests: it sorts the lines within a file or files. A variety of options exist to allow sorting on fields or character positions within the file, and to modify the comparison operation (numeric, date, case-insensitive, etc).

A common use of `sort` is in combining multiple files. Building on our earlier example:

```
$ cat employees2
Doug Dobrovsky Accountant 333 Tri-State Road
Adam Aman Technician 4 Fourth Street
$ sort employees employees2
Adam Aman Technician 4 Fourth Street
Alice Aaronson System Administrator 99 9th Street
Bob Babuk Programmer 7 77th Avenue
Carol Cavo Manager 111 West 1st Blvd.
Doug Dobrovsky Accountant 333 Tri-State Road
```

Field, and character position within fields, may be specified as sort criteria, and you can also sort numerically:

```
$ cat namenums
Alice 123
Bob 45
Carol 6
$ sort -k 2.1 -n namenums
Carol 6
Bob 45
Alice 123
```

## uniq

The utility `uniq` removes adjacent lines that are identical to each other -- or if some switches are used, close enough to count as identical (you may skip fields, character positions, or compare as case-insensitive). Most often, the input to `uniq` is the output from `sort`, though GNU `sort` itself contains a limited ability to eliminate duplicate lines with the `-u` switch.

The most typical use of `uniq` is in the expression `sort list_of_things | uniq`, producing a list with just one of each item (one per line). But some fancier uses let you analyze duplicates or use different duplication criteria:

```
$ uniq -d test5 # identify duplicates
Bob
$ uniq -c test5 # count occurrences
1 Alice
2 Bob
1 Carol
$ cat test4
1 Alice
2 Bob
3 Bob
4 Carol
$ uniq -f 1 test4 # skip first field in comparisons
1 Alice
2 Bob
4 Carol
```

## tsort

The utility `tsort` is a bit of an oddity in the text utilities collection. The utility itself is quite useful in a limited context, but what it does is not something you would centrally think of as text processing -- `tsort` performs a *topological* sort on a directed graph. Don't panic just yet if this concept is not familiar to you: in simple terms, `tsort` is good for finding a suitable order among dependencies. For example, installing packages might need to occur with certain order constraints, or some system daemons might need to be initialized before others.

Using `tsort` is quite simple, really. Just create a file (or stream) that lists each known dependency (space separated). The utility will produce a suitable (not necessarily uniquely so) order for the whole collection. For example:

```
$ cat dependencies # not necessarily exhaustive, but realistic
libpng XFree86
FreeType XFree86
Fontconfig XFree86
FreeType Fontconfig
expat Fontconfig
Zlib libpng
Binutils Zlib
Coreutils Zlib
GCC Zlib
Glibc Zlib
Sed Zlibc
$ tsort dependencies
Sed
Glibc
GCC
Coreutils
Binutils
Zlib
expat
```

```
FreeType
libpng
Zlibc
Fontconfig
XFree86
```

## pr

The `pr` utility is a general page formatter for text files that provides facilities such as page headers, linefeeds, columns of source texts, indentation margins, and configurable page and line width. However, `pr` does not itself rewrap paragraphs, and so might often be used in conjunction with `fmt`.

```
$ tail -5 rexx.txt | pr -w 60 -f | head
2004-01-31 03:22                                     Page 1

{Picture of Author: http://gnosis.cx/cgi-bin/img_dqm.cgi}
David Mertz' fondness for IBM dates back embarrassingly many decades.
David may be reached at mertz@gnosis.cx; his life pored over at
http://gnosis.cx/publish/. And buy his book: _Text Processing in
Python_ (http://gnosis.cx/TPiP/).
```

And now as two columns:

```
$ tail -5 rexx.txt | fmt -30 > blurb.txt
$ pr blurb.txt -2 -w 65 -f | head
2004-01-31 03:24                                     blurb.txt                                     Page 1

{Picture of Author:                                     at mertz@gnosis.cx; his life
http://gnosis.cx/cgi-bin/img_d                         pored over at http://gnosis.cx
David Mertz' fondness for IBM                         And buy his book: _Text
dates back embarrassingly many                        Processing in Python_
decades. David may be reached                         (http://gnosis.cx/TPiP/).
```

---

## Section 6. Combining and splitting multiple files

### comm

The utility `comm` is used to compare the contents of already (alphabetically) sorted files. This is useful when the lines of files are considered as unordered collections of

*items*. The `diff` utility, though not included in the text utilities, is a more general way of comparing files that might have isolated modifications -- but that are treated in an ordered manner (such as source code files or documents). On the other hand, files that are considered as fields of records do not have any inherent order, and sorting does not change the information content.

Let's look at the difference between two sorted lists of names; the columns displayed are those in first file only, those in the second only, and those in common:

```
$ comm test2b test2c
      Alice
Betsy
      Bob
Brian
      Cal
      Carol
```

Introducing an out-of-order name, we see that `diff` compares happily, while `comm` fails to identify overlaps anymore:

```
$ cat test2d
Alice
Zack
Betsy
Bob
Carol
$ diff -U 2 test2d test2c
--- test2d      Sun Feb  1 18:18:26 2004
+++ test2c      Sun Feb  1 18:01:49 2004
@@ -1,5 +1,4 @@
 Alice
-Zack
-Betsy
 Bob
+Cal
 Carol
$ comm test2d test2c
      Alice
      Bob
      Cal
      Carol

Zack
Betsy
Bob
Carol
```

## join

The utility `join` is quite interesting; it performs some basic relational calculus (as will be familiar to readers who know relational database theory). In short, `join` lets you find records that share fields between (sorted) record collections. For example, you might be interested in which IP addresses have visited both your Web site and your

FTP site, along with information on these visits (resources requested, times, etc., which will be in your logs).

To present a simple example, suppose you issue color-coded access badges to various people: vendors, partners, employees. You'd like information on which badge types have been issued to employees. Notice that names are the first field in employees, but second in badges, all tab separated:

```
$ cat employees
Alice Aaronson System Administrator 99 9th Street
Bob Babuk Programmer 7 77th Avenue
Carol Cavo Manager 111 West 1st Blvd.
$ cat badges
Red Alice Aaronson
Green Alice Aaronson
Red David Decker
Blue Ernestine Eckel
Green Francis Fu
$ join -1 2 -2 1 -t $'\t' badges employees
Alice Aaronson Red System Administrator 99 9th Street
Alice Aaronson Green System Administrator 99 9th Street
```

## paste

The utility `paste` is approximately the reverse operation of that performed by `cut`. That is, `paste` combines multiple files into columns, such as fields. By default, the corresponding lines between files are tab separated, but you may use a different delimiter by specifying a `-d` option.

While `paste` can combine unrelated files (leaving empty fields if one input is longer), it generally makes the most sense to `paste` synchronized data sources. One example of this is in reorganizing the fields of an existing data file:

```
$ cut -f 1 employees > names
$ cut -f 2 employees > titles
$ paste -d "," titles names
System Administrator,Alice Aaronson
Programmer,Bob Babuk
Manager,Carol Cavo
```

The flag `-s` lets you reverse the use of rows and columns, which amounts to converting successive lines in a file into delimited fields:

```
$ paste -s titles | cat -T
System Administrator^IProgrammer^IManager
```

## split

The utility `split` simply divides a file into multiple parts, each one of a specified number of lines or bytes (the last one perhaps smaller). The parts are written to files whose names are sequenced with two suffix letters (by default `xaa`, `xab`, ... `xzz`).

While `split` can be useful just in managing the size of large files or data sets, it is more interesting in processing more structured data. For example, in [Lines as records](#), we saw an example of splitting fields across lines -- what if we want to assemble those back into `employees`-style tab-separated fields, one per line. Here is a way to do it:

```
$ cut -b 3- multiline | split -l 3 - employee
$ cat employeeab
Bob Babuk
Programmer
7 77th Avenue
$ paste -s employeea*
Alice Aaronson System Administrator 99 9th Street
Bob Babuk Programmer 7 77th Avenue
```

## csplit

The utility `csplit` is similar to `split`, but it divides files based on context lines within them, rather than on simple line/byte counts. You may divide on one or more different criteria within a command, and may repeat each criterion however many times you wish. The most interesting criterion-type is regular expressions to match against lines. For example, as an odd cut-up of `multiline`:

```
$ csplit multiline -zq 2 /99/ /Progr/ # line 2, find 99, find Progr
$ cat xx00
A Alice Aaronson
$ cat xx01
B System Administrator
$ cat xx02
C 99 9th Street
A Bob Babuk
$ cat xx03
B Programmer
C 7 77th Avenue
```

The above division is a bit perverse in that it does not correspond with the data structure. A more usual approach might be to arrange to have delimiter *lines*, and split on those throughout:

```
$ head -5 multiline2
```

```

Alice Aaronson
System Administrator
99 9th Street
-----
Bob Babuk
$ csplit -zq multiline2 /-----/+1 {*} # incl dashes at end, per chunk
$ cat xx01
Bob Babuk
Programmer
7 77th Avenue
-----

```

---

## Section 7. Summarizing and identifying files

### The simplest summary: `wc`

Most of the tools we have seen so far produce output that is largely reversible to create the original form -- or at the least, each line of input contributes in some straightforward way to the output. A number of tools in the GNU text utilities can instead be best described as producing a summary of a file. Specifically, the output of these utilities is generally much shorter than the inputs, and the utilities all discard most of the information in their input (technically, you could describe them as *one-way functions*).

About the simplest one-way function on an input file is to count its lines, words, and/or bytes, which is what `wc` (word count) does. These are interesting things to know about a file, but are clearly non-unique among distinct files. For example:

```

chars, name    $ wc rexx.txt # lines, words,
              402      2585   18231 rexx.txt
count         $ wc -w < rexx.txt # bare word
              2585
name          $ wc -lc rexx.txt # lines, chars,
              402      18231 rexx.txt

```

Put to practical use, I could determine which *developerWorks* articles I have written are the wordiest. I might use (note inclusion of total, another pipe to `tail` could remove that):

```

$ wc -w *.txt | sort -nr | head -4
55190 total

```

```
3905 quantum_computer.txt
3785 filtering-spam.txt
3098 linuxppc.txt
```

## cksum and sum

The utilities `cksum` and `sum` produce checksums and block counts of files. The latter exists for historical reasons only, and implements a less robust method. Either utility produces a calculated value that is unlikely to be the same between randomly chosen files. In particular, a checksum lets you establish to a reasonable degree of certainty that a file has not become corrupted in transmission or accidentally modified. `cksum` implements four successively more robust techniques, where `-o 1` is the behavior of `sum`, and the default (no switch) is best.

```
$ cksum rexx.txt
937454632 18231 rexx.txt
$ cksum -o 3 < rexx.txt
4101119105 18231
$ cat rexx.txt | cksum -o 2
47555 36
$ cksum -o 1 rexx.txt
10915 18 rexx.txt
```

## md5sum and sha1sum

The utilities `md5sum` and `sha1sum` are similar in concept to that of `cksum`. Note, by the way, that in BSD-derived systems, the former command goes by the name `md5`. However, `md5sum` and `sha1sum` produce 128-bit and 160-bit checksums, respectively, rather than the 16-bit or 32-bit outputs of `cksum`. Checksums are also called hashes.

The difference in checksum lengths gives a hint as to a difference in purpose. In truth, comparing a 32-bit hash value is quite unlikely to falsely indicate that a file was transmitted correctly and left unchanged. But protection against accidents is a much weaker standard than protection against malicious tamperers. And MD5 or SHA hash is a value that is computationally infeasible to spoof. The hash length of a *cryptographic* hash like MD5 or SHA is necessary for its strength, but a lot more than just the length went into their design.

Imagine this scenario: you are sent a file on an insecure channel. In order to make sure that you receive the real data rather than some malicious substitute, the sender publishes (through a different channel) an MD5 or SHA hash for the file. An adversary cannot create a false file with the published MD5/SHA hash -- the checksum, for practical purposes, uniquely identifies the desired file. While `sha1sum`

is actually a bit better cryptographically, for historical reasons, `md5sum` is in more widespread use.

```
$ md5sum rexx.txt
2cbdbc5bc401b6eb70a0d127609d8772  rexx.txt
$ cat md5s
2cbdbc5bc401b6eb70a0d127609d8772  rexx.txt
c8d19740349f9cd9776827a0135444d5  metaclass.txt
$ md5sum -cv md5s
rexx.txt      OK
metaclass.txt FAILED
md5sum: 1 of 2 file(s) failed MD5 check
```

---

## Section 8. Working with log files

### The structure of a weblog

A weblog file provides a good data source for demonstrating a variety of real-world uses of the text utilities. Standard Apache log files contain a variety of space-separated fields per line, with each line describing one access to a Web resource. Unfortunately for us, spaces also occur at times inside quoted fields, so processing is often not quite as simple as we might hope (or as it might be if the delimiter were excluded from the fields). Oh well, we must work with what we are given.

Let's take a look at a line from one of my weblogs before we perform some tasks with it.

We can see that the original file is pretty large: 24,422 records. Wrapping the fields with `fmt` does not always wrap on field boundaries, but the quotes let you see what the fields are.

```
$ wc access-log
 24422  448497 5075558 access-log
$ head -1 access-log | fmt -25
62.3.46.183 - -
[28/Dec/2003:00:00:16 -0600]
"GET /TPiP/cover-small.jpg
HTTP/1.1" 200 10146
"http://gnosis.cx/publish/programming/regular_expressions.html "
"Mozilla/4.0 (compatible;
MSIE 6.0; Windows NT 5.1)"
```

## Extracting the IP addresses of Web site visitors

A very simple task to perform on a weblog file is to extract all the IP addresses of visitors to the site. This combines a few of our utilities in a common pipe pattern (we'll look at the first few):

```
$ cut -f 1 -d " " access-log | sort | uniq | head -5
12.0.36.77
12.110.136.90
12.110.238.249
12.111.153.49
12.13.161.243
```

We might wonder, also, just how many such distinct visitors have visited in total:

```
$ cut -f 1 -d " " access-log | sort | uniq | wc -l
2820
```

## Counting occurrences

We determined how many visitors our Web site got, but perhaps we are also interested in how much each of those 2820 visitors contribute to the overall 24,422 hits. Or specifically, who are the most frequent visitors. In one line we can run:

```
$ cut -f 1 -d " " access-log | sort | uniq -c | sort -nr | head -5
1264 131.111.210.195
524 213.76.135.14
307 200.164.28.3
285 160.79.236.146
284 128.248.170.115
```

While this approach works, it might be nice to pull out the histogram part into a reusable shell script:

```
$ cat histogram
#!/bin/sh
sort | uniq -c | sort -nr | head -n $1
$ cut -f 1 -d " " access-log | ./histogram 3
1264 131.111.210.195
524 213.76.135.14
307 200.164.28.3
```

Now we can pipe any line-oriented list of items to our histogram shell script. The

number of most frequent items we want to display is a parameter passed to the script.

## Generate a new ad hoc report

Sometimes existing data files contain information we need, but not necessarily in the arrangement needed by a downstream process. As a basic example, suppose you want to pull several fields out of the weblog shown on [The structure of a weblog](#), and combine them in different order (and skipping unneeded fields):

```
$ cut -f 6 -d \" access-log > browsers
$ cut -f 1 -d \" \" access-log > ips
$ cut -f 2 -d \" \" access-log | cut -f 2 -d \" \"
  | tr "/" ":" > resources
$ paste resources browsers ips > new.report
$ head -2 new.report | tr "\t" "\n"
:TPiP:cover-small.jpg
Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1)
62.3.46.183
:publish:programming:regular_expressions.html
Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1)
62.3.46.183
```

The line that produces `resources` uses two passes through `cut`, with different delimiters. That is because what `access-log` thinks of as a REQUEST contains more information than we want as a RESOURCE.

```
$ cut -f 2 -d \" \" access-log | head -1
GET /TPiP/cover-small.jpg HTTP/1.1
```

We also decide to massage the path delimiter in the Apache log to use a colon path separator (which is the old Mac OS format, but we really just do it here to show a type of operation).

## cut\_by\_regex

This next script combines much of what we have seen in this tutorial into a rather complex pipeline. Suppose that we know we want to cut a field from a data file, but we do not know its field position. Obviously, visual examination could provide the answer, but to automate processing different types of data files, we can cut *whichever* field matches a regular expression:

```
$ cat cut_by_regex
#!/bin/sh
# USAGE: cut_by_regex <pattern> <file> <delim>
```

```
cut -d "$3" -f \  
  `head -n 1 $2 | tr "$3" "\n" | nl | \  
  egrep $1 | cut -f 1 | head -1` \  
$2
```

In practice, we might use this:

```
$ ./cut_by_regex "[0-9]+\.[0-9]{3}" access-log " " | ./histogram 3  
1264 131.111.210.195  
524 213.76.135.14  
307 200.164.28.3
```

Several parts of this could use further explanation. The backtick is a special syntax in `bash` to treat the result of a command as an argument to another command. Specifically, the pipe in the backticks produces the *first* field number that matches the regular expression given as the first argument.

How does it manage this? First we pull off only the first line of the data file; then we transform the specified delimiter to a newline (one field per line now); then we number the resulting lines/fields; then we search for a line with the desired pattern; then we cut just the field number from the line; and finally we take only the first match, even if several fields match. It takes a bit of thought to put a good pipeline together, but a lot can be done this way.

---

## Section 9. Summary

### Summary

This tutorial directly presents just a small portion of what you can achieve with the GNU text utilities. The final few examples start to suggest just how powerful the utilities can be with creative use of pipes and redirection. The key is to break an overall transformation down into useful intermediate data, either saving that intermediary to another file or piping it to a utility that deals with that data format.

I wish to thank my colleague Andrew Blais for assistance in preparation of this tutorial.

# Resources

## Learn

- Get help with any of the utilities covered in this tutorial by typing `man utility-name` or `utility-name --help` at the command line. Or view the [online man pages](#) at GNU.
- Some other common UNIX and Linux command-line tools are covered in the developerWorks tutorial "[GNU and UNIX commands](#)" (developerWorks, November 2005) and [Basic UNIX commands](#).
- Peter Seebach's "[The art of writing Linux utilities: Developing small, useful command-line tools](#)" (developerWorks, January 2004) gives guidelines and best practices for writing your own utility.
- Learn more about UNIX utilities, how to use them, and how to write your own in "[UNIX utilities](#)" (developerWorks, May 2001).
- Once you've learned to write your own utilities, learn how to turn them into a library! It's easy with the tutorial "[Building a cross-platform C library](#)" (developerWorks, June 2001).
- Get more in tune with your UNIX/Linux environment by reading the essays "[Basics of the Unix Philosophy](#)" and "[The Unix Philosophy](#)".
- Get started with bash programming with the three-part developerWorks series, starting with "[Bash by example](#)" (developerWorks, March 2000).
- David Mertz's "[Using regular expressions](#)" tutorial (developerWorks, September 2000) is a good starting point for understanding the tools like `grep` and `csplit` that use regular expressions.
- David's book *[Text Processing in Python](#)* (Addison Wesley, 2003; ISBN: 0-321-11254-7) also contains an introduction to regular expressions, as well as extensive discussion of many of the techniques in this tutorial using Python.
- David's article "[Rexx for everyone](#)" (developerWorks, February 2004) presents an alternative approach to simple text-processing tasks. The scope of the text utilities is nearly identical to the core purpose of the Rexx programming language.
- Find more [tutorials for Linux developers](#) in the [developerWorks Linux one](#).
- Stay current with [developerWorks technical events and Webcasts](#).

## Get products and technologies

- Linux users will in all likelihood already have the text utilities installed in their distribution. But if not -- or if you are running older versions of them -- you can [download the latest 27 GNU text utilities](#) from their FTP site.

- The most current utilities have been incorporated into the [GNU Core Utilities](#).
- Windows users can find these tools and many others in the [Cygwin](#) package, while Mac OS X users can try [Fink](#).
- Download [IBM trial software](#) directly from developerWorks.

### Discuss

- Read [developerWorks blogs](#), and get involved in the developerWorks community.

## About the author

### David Mertz

David Mertz has an enduring fondness for munging text. He even went so far as to write the book, *[Text Processing in Python](#)*, and frequently touches on related topics for his IBM developerWorks articles and columns. [David's Web site](#) is also a good source of related information.