

Unix

Comandos de shell complejos con ejemplos

| | |
|--------------|-----------|
| Cut | 1 |
| Find | 3 |
| Grep | 9 |
| Join | 13 |
| Paste | 21 |
| Sort | 24 |
| Tr | 39 |
| Uniq | 44 |

<http://unix.about.com/library/misc/blpowercmds.htm>

Power Commands: Cut

Description

```
cut [-b | -c | -f] list [options] [File ...]
```

The *cut* command selects columns from a file and prints them to [standard output](#). If no file is specified *cut* reads from [standard input](#). Columns can be specified as bytes, characters or delimited fields. For example,

```
$ cut -c 1-10 file1 file2
```

prints the first ten characters from every line in the file *file1* to the screen then prints the first ten characters from every line in the file *file2* to the screen.

The command line options for *cut* are described below.

| Option | Description |
|----------------|--|
| -b list | Columns are specified by byte positions. |
| -c list | Columns are specified by character. For example, <i>-c 1-72</i> cuts the first 72 characters in each line of a file. |
| -f list | Columns are specified by fields. Fields should be separated by a delimiter character. The delimiter can be set with the <i>-d</i> option. The default delimiter is a TAB . For example, <i>-f 2,5</i> selects the second and fifth fields of each line in a file with columns separated by TAB s. If a line does not contain any delimiters, <i>cut</i> will print that line to standard output unless the <i>-s</i> option is used. |
| -d c | Specifies the field delimiter when the <i>-f</i> option is used. |
| -s | Used with the <i>-f</i> option. If a line does not contain any delimiters, the <i>-s</i> options stops <i>cut</i> from printing that line. |

Exactly one of *-b*, *-c* or *-f* must be specified followed by a list. The list should be a list of integer numbers in increasing order separated by commas. A hyphen separator can be used to indicate an entire range. The following table shows some examples of lists.

| List | Meaning |
|-----------------|---|
| <i>n1,n2,n3</i> | Cut <i>n1</i> , <i>n2</i> and <i>n3</i> . |
| <i>n1-n2</i> | Cut <i>n1</i> through <i>n2</i> . |
| <i>n1-n2,n3</i> | Cut <i>n1</i> through <i>n2</i> and <i>n3</i> . |
| <i>-n1,n2</i> | Cut from 1 through <i>n1</i> and <i>n2</i> . |
| <i>n1,n2-</i> | Cut <i>n1</i> and from <i>n2</i> to the end of the line . |

Examples

1. In the file *dataset1*
2. Pine 906 26 1.0 211
3. Beech 933 26 2.3 160
4. Fur 1246 27 2.44 162
5. Palm 671 25 3.8 888

cut the second entry which is stored in columns 13 to 17.

```
$ cut -c 13-17 dataset1
```

6. Cut columns 1 through 72 from the file *prog1.f* and [redirect](#) the output from the screen to the *code.f* file.
7. `$ cut -c -72 prog1.f > code.f`
8. Cut all characters stored after the 72nd column in file *prog1.f* and save the results in a file called *comments*.
9. `$ cut -c 73- prog1.f > comments`
10. In the file *dataset2*
11. Pine 906 26 020079 130.0 80.3 17.1 211
12. Beech 933 26 030079 48.0 85.2 22.7 160
13. Fur 1246 27 070079 31.0 86.5 6.9 162
14. Palm 671 25 100077 41.0 87.3 15.0 888

which has eight fields separated by spaces cut the 2nd through 4th and then the 7th fields.

```
$ cut -f 2-4,7 -d " " dataset2
```

15. In the file *dataset3*
16. Trees of the Forest
17. Pine,906,26,020079,130.0,80.3,17.1,211
18. Beech,933,26,030079,48.0,85.2,22.7,160
19. Fur,1246,27,070079,31.0,86.5,6.9,162
20. Palm,671,25,100077,41.0,87.3,15.0,888

cut fields 1, 3, 4, 5, 6 and 8.

```
$ cut -f 1,3-5,6,8 -d , dataset3
```

This will display the line *Trees of the Forest* because it has no delimiter characters. To cut the desired fields without including the line *Trees of the Forest*,

```
$ cut -f 1,3-4,6,8 -d , -s dataset3
```

21. List the first 8 character of every file in the current directory.
22. `$ ls -l | cut -c 1-8`

The [ls -l](#) command lists all files in the current directory in one column. The output from the *ls* command is [piped](#) to the *cut* command, which selects the first eight characters of the file names.

Advanced Examples

1. List login names off all users logged in.
2. `$ who | cut -f 1 -d " "`

The [who](#) command lists all users logged in. The first column contains the user's login name and subsequent columns contain more information. The output from the *who* command is [piped](#) to the *cut* command, which selects only the first column of output.

3. Display columns one and five from the */etc/passwd* file, the userid and real name.
4. `# cut -f 1,5 -d : /etc/passwd`

Note that the pound sign (#) is used for the Unix prompt when the example assumes you are the [super user](#).

Power Commands: Find

The *find* command searches entire Unix directory structures for files.

Description

```
find pathnames search-expressions action-expressions
```

The *find* command searches entire directory structures beginning with *pathnames* and performs actions specified by the *action-expressions* on all files with attributes matching the *search-expressions*. For example,

```
$ find . -name '*.f' -print
./progl.f
./stat/mean.f
./stat/var.f
./math/matrix.f
```

searches the entire directory structure beginning with the current directory (specified by a dot, *.*) for files with filenames ending in *.f* and prints the filename of each file found. In this example, *-name '*.f'* is a search expression and *-print* is an action expression. Any number of search and action expressions can be used with one *find* command. For example,

```
$ find . -name '*.f' -mtime 1 -print -cpio /dev/rmt1
```

searches the entire directory structure beginning with the current directory for filenames ending in *.f* that were modified one day ago. The *-name* search expression matches files with filenames ending in *.f* and the *-mtime* search expression matches files that were modified one day ago. The action expression *-print* prints the filenames to the screen and the action expression *-cpio* writes the files to the */dev/rmt1* tape device. The most useful search expressions and action expressions are shown below.

Search Expressions

| | |
|---------------------------|---|
| -name 'pattern' | Find files with filenames matching <i>pattern</i> . The pattern may include metacharacters . The pattern should be surrounded by single quotes to keep the shell from interpreting the special characters . |
| -size [+ -]n[c] | Find files that are at least (+ <i>n</i>) exactly (<i>n</i>) or less than (- <i>n</i>) <i>n</i> blocks large. On most systems one block is 512 bytes or half a kilobyte. If <i>c</i> is appended, sizes are specified by characters (i.e. bytes). |
| -atime +n n -n | Find files that were last accessed more than (+ <i>n</i>) exactly (<i>n</i>) or less than (- <i>n</i>) <i>n</i> days ago. A file is accessed any time a command is executed on it including the <i>find</i> command. |
| -mtime +n n -n | Find files that were last modified more than (+ <i>n</i>) exactly (<i>n</i>) or less than (- <i>n</i>) <i>n</i> days ago. |
| -ctime +n n -n | Find files that were created more than (+ <i>n</i>) exactly (<i>n</i>) or less than (- <i>n</i>) <i>n</i> days ago. |
| -newer filename | Find files modified more recently than the file specified by <i>filename</i> . |
| -type c | Find files of type <i>c</i> . The most useful values of <i>c</i> for most users are d to specify a directory, f to specify a plain or ordinary file and l to specify a symbolic link. Other values are b for a block special file, c for a character special file and p for a fifo or named pipe. |
| -perm [-]perm-list | Find files with permissions exactly matching those specified by the <i>perm-list</i> . |

| | |
|--|--|
| | If perm-list is preceded by a hyphen (-), find files with at least the permissions specified. Read the Unix File Permissions article for more information. |
|--|--|

Supported Metacharacters

The *find* command supports several metacharacters or [wildcards](#) when searching by name (i.e. using the *-name 'pattern'* search-expression).

| | |
|--------|--|
| * | Match zero or more occurrences of any character. |
| ? | Match any single character. |
| [...] | Match one from a set of characters. |
| [n-m] | Match any characters in the range expressed by <i>n-m</i> . |
| [^...] | Match any character <i>not</i> enclosed in brackets. |
| \ \ | Preceding any meta character by a backslash (\) turns off the meaning. |

Action Expressions

| | |
|-------------------------|---|
| -print | Prints the path and filename to each file found. Paths are specified relative to the search path. |
| -exec <i>cmd</i> | Execute <i>cmd</i> for each file found. In <i>cmd</i> the current file is specified with <code>\{ \}</code> (Backslash, Open Curly Bracket, Backslash, Closed Curly Bracket). The <i>cmd</i> must end with a <code>\;</code> (Backslash, semi-colon). Note that the <i>exec</i> expressions returns true if the commands was successfully completed and false otherwise. In this manner <i>exec</i> functions as a search expression as well as an action expression. |
| -ok <i>cmd</i> | Query user before executing <i>cmd</i> on each file found. <i>Ok</i> works just like <i>exec</i> except that the user is queried before each command is executed. The command about to be executed will be printed followed by a question mark (?). Typing y will cause the execution to continue. Typing n will skip execution of <i>cmd</i> and continue to the next file. |
| -depth | Causes actions to be taken on files within a directory before the directory itself. |
| -prune | Skip the directory most recently matched. |

Operators

Search expressions may be combined into compound expressions using operators. Operators allow for complicated searches. They are listed below in the order they are evaluated.

| | |
|---|---|
| <code>\(<i>expression</i> \)</code> | True if the expression in the parentheses is true. Expressions in parentheses are evaluated first. The parentheses are preceded by a backslash to keep the shell from interpreting them as special characters. This is required in the Bourne, Korn, c-shell and their derivatives. |
| <code>! <i>expression</i></code> | The exclamation point is a <i>NOT</i> operator. It evaluates to true if <i>expression</i> is false. |
| <code><i>expression</i> -a <i>expression</i> <i>expression</i> <i>expression</i></code> | The <i>and</i> operator evaluates to true if both expressions are true. The <i>a</i> does not have to be specified. It is implied by using more than one search expression. The second expression will not be evaluated if the first expression is false. |
| <code><i>expression</i> -o <i>expression</i></code> | The <i>or</i> operator, <i>-o</i> , evaluates to true if either one of the two expressions are true. The second expression will not be evaluated if the first expression is true. |

Advanced Expressions

These expressions are useful for advanced users or system administrators.

| | |
|----------------------------|---|
| -inum <i>inode</i> | Find files whose <u>inode number</u> is <i>inode</i> . |
| -links <i>n</i> | Find files with <i>n</i> <u>links</u> . |
| -group <i>gname</i> | Find files that belong to the group specified in <i>gname</i> . <i>Gname</i> can be a group name or numeric group id. |
| -nogroup | Find files that belong to a group <i>not</i> in <i>/etc/group</i> . |
| -user <i>uname</i> | Find files that belong to the user specified by <i>uname</i> . <i>Uname</i> can be a group name or numeric user id. |
| -nouser | Find files that belong to a user <i>not</i> in <i>/etc/passwd</i> . |
| -cpio <i>device</i> | Writes each file found onto <i>device</i> using the <u>cpio</u> format. For most users <i>device</i> is the physical name of a tape drive. |
| -xdev | Do not follow a search onto a different filesystem. Search only for file that reside on the same file system as <i>pathname</i> . The <i>-xdev</i> expression is implemented even if it follows a false search expression. Note that on older System V based Unix flavors this expression is <i>-mount</i> instead of <i>-xdev</i> . <i>Xdev is not implemented on all Unix flavors</i> . |
| -follow | Follow symbolic links and track the directories visited. This should not be used with the <i>-type l</i> expression. <i>Follow is not implemented on all Unix flavors</i> . |

Examples

1. Search the entire home directory including all subdirectories for a file named *lostfile* and print the path to *lostfile* to the screen.

```
$ find ~ -name 'lostfile' -print
```

Note that the tilde (~) specifies your home directory.

2. Starting from the home directory, recursively find all files with filenames ending in *.cpp* and print the results to the screen.

```
$ find ~ -name '*.cpp' -print
```

3. Find all files starting from the current directory whose filenames begin with *doc* and have one extra character.

```
$ find . -name 'doc?' -print
```

This would find files named *doc1*, *docd* and *docs*, but not *doc* or *doc12*. Note that the period (.) specifies the current directory.

4. Find all files starting with the current directory whose filenames begin with a capital letter and end with a number.

```
$ find . -name '[A-Z]*[0-9]' -print
```

5. Find all files starting with the directory */usr/local/install* that are named *readme* or *Readme*.

```
$ find /usr/local/install -name '[R,r]eadme' -print
```

This can also be done with the *or* compound expression.

```
$ find /usr/local/install \( -name 'readme' -o -name 'Readme' \)
-print
```

6. Find all files starting with the current directory whose filenames do not end in *.bak*.

```
$ find . -name '*[^.bak]' -print
```

This can also be done with the *NOT* logical operator.

```
$ find . ! -name '*.bak' -print
```

7. Find all files starting from the home directory named *Spec*ial*.

```
$ find ~ -name 'Spec\*ial' -print
```

Note that the backslash (\) tells *find* not to treat the star (*) as a meta character.

8. Find all files starting from the home directory created in the last five days.

```
$ find ~ -ctime -5 -print
```

9. Create a listing of all files and subdirectories contained in the directories *~/ccode* and *~/fortran* and save it in the file *programlist*.

```
$ find ~/ccode ~/fortran -print > programlist
```

Note the greater than sign (>) [redirects](#) the *-print* output from the screen to the *programlist* file.

10. Create a listing of all directories starting with the home directory and save it in the file *dirlist*.

```
$ find ~ -type d -print > dirlist
```

11. Find all files starting in the home directory that have not been accessed in the last 30 days.

```
$ find ~ -atime +30 -print
```

Note that if you run this command a second time there will be no files found because the *find* command accesses each file in the home directory when it runs.

12. Find all files starting in the home directory whose filenames end in *.f* that were modified in the last day.

```
$ find ~ -name '*.f' -mtime -1 -print
```

13. Find all files starting in the home directory newer than the file *~/newfiles/outline*.

```
$ find ~ -newer ~/newfiles/outline -print
```

14. Find all files starting in the home directory newer than the file *~/newfiles/outline* and also named *outline*.

```
$ find ~ -newer ~/newfiles/outline -name 'outline' -print
```

15. Find all files starting in the home directory newer than the file *~/newfiles/outline*, named *outline* and copy them to the current directory.

```
$ find ~ -newer ~/newfiles/outline -name 'outline' -exec cp {\}
. \;
```

16. Remove all files and subdirectories starting with the directory *olddir*. Query the user before executing the remove command.

```
$ find olddir -depth -ok rm {\} \;
```

The *-depth* option is required. Otherwise, *find* would attempt to remove directories before they were empty and this would fail.

17. Find all files starting with your home directory with read and write permission for the user and read permission only for the group and others.

```
$ find ~ -perm 644 -print
```

In the above example the permission list is specified with [octal numbers](#). This method should work under most Unix flavors. Modern Unix flavors support a [symbolic mode](#) to specify the permission list. For example,

```
$ find ~ -perm u=rw,go=r -print
```

does the same search as the above example. You can learn more about searching for files based on their file permission in the [Unix 101: File Permissions, Part 2](#) feature article.

18. Find all files in my home directory where the group or others have write permission and use the [chmod](#) command to remove that permission.

```
$ find ~ \( -perm -020 -o -perm -002 \) -exec chmod go-w {\} \;
```

```
$ find ~ \( -perm -g=w -o -perm -o=w \) -exec chmod go-w {\} \;
```

The first example uses octal numbers and the second symbolic mode to specify the permission list in *find*.

19. Find all files starting with the current directory larger than 1000 blocks (around 500 kilobytes on most systems).

```
$ find . -size +1000 -print
```

20. Find all regular files starting with the current directory larger than 1000 blocks whose filenames do not end with *.Z* and query the user before compressing them with the [compress](#) command.

```
$ find . ! \( -name '*.Z' \) -type f -size +1000 -ok compress
{\} \;
```

21. Find all files starting in the current directory whose filenames end in *.ssd01* or *.sct01* copy them to the directory *~/saslib* and query me before remove them *only if* the *cp* command was successful.

```
$ find . \( -name '*.ssd01' -o -name '*.sct01' \) -exec cp {\}
~/saslib \; -ok rm {\} \;
```

22. Starting from the home directory, find all files whose filenames end with *.bak* but skip the *backups* directory.

```
find ~ \( -name '*.bak' -o \( -name 'backups' -prune \) \) -type
f -print
```

Advanced Examples

1. Find a file in the current directory with inode number 1428846 and query before renaming the file to *newname*.

```
$ find . -inum 1428846 -ok mv {\} newname \;
```

This is useful for [renaming files with special characters in their filenames](#). Note: To find the inode number of a file use the *-i* option with the *ls* command.

2. Find all files starting in the */usr/home* directory that do not belong to a group in the */etc/groups* file.

```
# find /usr/home -nogroup -print
```

Note that the pound sign (#) is used for the Unix prompt when the example assumes you are the super user.

3. Find all files starting in the */usr/bin* directory that have exactly five links.

```
# find /usr/bin -links 5 -print
```

4. Find all files starting in the */usr/home* directory that belong to the user *bobq* and change the owner of the file to *root* using the *chown* command.

```
# find /usr/home -user bobq -exec chown root {\} \;
```

5. Find all regular files in the */usr* directory that have been modified in the last 5 days and copy them to a tape device. Follow symbolic links but do not find any files on another filesystem.

```
# find /usr -follow -xdev -mtime -5 -cpio /dev/rmt1
```

Power Commands: Grep

The *grep* command is a flexible and powerful tool that searches for text strings in files.

Description

```
grep [options] 'pattern' [file ...]
```

The *grep* command searches one or more files for a text pattern and prints all lines that contain that pattern. If no files are specified, *grep* reads from [standard input](#). If more than one file is specified, the filename is printed before matching lines. For example,

```
$ grep 'Howdy' *
TexasMemo:Howdy Ya'll
letter:Howdy Mark,
letter:I was just writting to say Howdy.
```

prints three lines that contain the string *Howdy* from two files in the current directory, *TexasMemo* and *letter*.

The most common command line options for *grep* are shown below.

| Option | Description |
|-----------|---|
| -i | Ignore upper/lower case distinction. |
| -n | Print matched lines and their line numbers. |
| -c | Print only a count of the matching lines. |
| -l | Print names of files with matching lines but not the lines. |
| -h | Print matching lines but not the filenames. |
| -v | Print all lines that <i>don't</i> match <i>pattern</i> . |
| -s | Suppress error messages for non-existent or unreadable files. |

Patterns in *grep* are based on limited [regular expressions](#). Regular expressions provide flexible character matching abilities including the use of wildcards, matching on ranges of characters and searching for the beginning or end of lines. For example, the caret (^) symbol indicates the beginning of a line, so

```
$ grep '^Howdy' *
TexasMemo:Howdy Ya'll
letter:Howdy Mark,
```

lists all lines that begin with *Howdy*.

Some useful features of regular expressions are shown in the table below.

| Symbol | Meaning |
|---------------|---|
| ^ | Match the beginning of a line. |
| \$ | Match the end of a line. |
| [...] | Match one from a set of characters. |
| [^...] | Match any character <i>not</i> enclosed in brackets. |
| [n-m] | Match any characters in the range expressed by <i>n-m</i> . |
| . | Match any single character except a newline. |
| c* | Match any number of the preceding character, <i>c</i> . |

| | |
|---------|---|
| .* | Match zero or more occurrences of any character. |
| \{n\} | Match exactly <i>n</i> occurrences of the preceding character or regular expression. |
| \{n,\} | Match at least <i>n</i> occurrences of the preceding character or regular expression. |
| \{n,m\} | Match any number between <i>n</i> and <i>m</i> of the preceding character or regular expression. Note: <i>n</i> and <i>m</i> must be between 0 and 256 inclusively. |
| \ | Preceding any special character by a backslash (\) turns off the meaning. |

Regular expressions should be surrounded by single quotes to [prevent the shell from interpreting the special characters](#).

Examples

1. Search for the string *finished* in the file *working*. Each line of *working* which contains the string *finished* is printed to the screen.

```
$ grep 'finished' working
```

2. Search for *finished* in all files in the current directory.

```
$ grep 'finished' *
```

3. List the names of all files in the current directory that contain the string *finished*. This will list just the filenames, not the individual lines containing *finished*.

```
$ grep -l 'finished' *
```

4. Search for the string "*my dog has fleas*" in all files in the current directory with filenames ending in *.txt*. Ignore the upper/lower case distinction.

```
$ grep -i 'my dog has fleas' *.txt
```

5. Search for the string "*end of sentence. Begin*" in *myfile*.

```
$ grep 'end of sentence\. Begin' myfile
```

The backslash (\) before the period (.) tells *grep* to ignore the special meaning of the period.

6. Search for *toyota* in all files in the current directory that have filenames beginning with *car*. Ignore the upper/lower case distinction. Print the matching lines but do not print the filenames.

```
$ grep -i -h 'toyota' car*
```

7. Search for the string *failing* in the file *students*. Print the matching lines and their line numbers to the screen.

```
$ grep -n 'failing' students
```

8. List all lines in the file *students* that do not contain the string *failing*.

```
$ grep -v 'failing' students
```

9. Count the number of lines in the file *students* that contain the string *failing*.

```
$ grep -c 'failing' students
```

10. Count the number of lines in the file *students* that do not contain the string *failing*.

```
$ grep -c -v 'failing' students
```
11. List lines containing any of the strings *suess*, *sucess*, *success*, *successs* etc., but not *succes*.

```
$ grep 'suc*ess' afile
```
12. List lines containing any of the strings *bid*, *bud*, *bed*, etc., but not *bd*, *band* or *lid*.

```
$ grep 'b.d' afile
```
13. List lines containing any of the strings *bd*, *bid*, *bud*, *band*, etc. but not *bank*.

```
$ grep 'b.*d' afile
```
14. List all lines that begin with *#include* in all files with filenames ending in *.c*.

```
$ grep '^#include' *.c
```
15. List all lines that end with *pork* in the file *dinner*.

```
$ grep 'pork$' dinner
```
16. Search for all lines containing *Urgent* or *urgent* in the file *TODO*. Display the line number with each matching line.

```
$ grep -n '[uU]rgent' TODO
```
17. List all lines in the file *myfile* that include *bad*, *bed*, *bid*, or *bud* but not *bod* or *bend*.

```
$ grep 'b[aeiu]d' myfile
```
18. Search for all lines in the file *TODO* that include a digit.

```
$ grep '[0-9]' TODO
```
19. Search for all lines in the file *myfile* that include an upper case letter.

```
$ grep '[A-Z]' myfile
```
20. List all lines that contain the string *bed*, *bud*, *bid*, etc but not *bd*, *bid* or *bond*.

```
$ grep 'b[^i]d' afile
```
21. List all lines in the file *myfile* that begin with an upper or lower case character.

```
$ grep '^[A-Za-z]' myfile.
```
22. List lines containing the strings *sucess* or *success* but not *suess* or *successs*.

```
$ grep 'suc\{1,2\}ess' afile
```
23. List all lines that contain a phone number of the form (nnn) nnn-nnnn.
24.

```
$ grep '([0-9]\{3\}) [0-9]\{3\}-[0-9]\{4\}' afile
```

Advanced Examples

1. Save all lines from the file *log* that begin with *error* or *dump* in a new file named *problems*.

```
$ grep '^error' log > problems
$ grep '^dump' log >> problems
```

The first *grep* command lists lines that begin with *error* and [redirects](#) the output to the file *problems*. The second *grep* command lists lines that begin with *dump* and appends the output to the *problems* file.

2. Search for all files in the current directory with filenames ending in *.sas* that contain the string "*cms filedef*" at the beginning of a line. Ignore the upper/lower case distinction and print only the name of files with matching lines.

```
$ grep -i -l '^cms filedef' *.sas
```

Use the [find](#) command to do the same search on all files in your entire directory tree beginning with your home directory.

```
$ find ~ -name '*.sas' -exec grep -i -l '^cms filedef' \{\} \;
```

3. List all files in the current directory that do not contain the string *error*.

```
$ grep -c 'error' * | grep ':0$'
```

The first *grep* command lists each file in the directory followed by a colon (:) and the number of times *error* appears in the file. The output is [piped](#) to the second *grep* command which lists all lines that end in :0 (i.e. did not contain *error*).

4. Find all files in the current directory with filenames ending in *.c* by piping the output from a long file listing to the *grep* command.

```
$ ls -l | grep '\.c$'
```

The [ls -l](#) command lists the files in the current directory in one column. The *\$* in the *grep* pattern specifies the end of the line while the backslash (\), keeps *grep* from interpreting the period (.).

5. List all directories that have execute permission for "other" users.

```
$ ls -l | grep 'd.....x'
```

The *ls -l* command does a long listing including the [file permissions](#). The pattern in *grep* searches for a string that starts with *d* has exactly eight unspecified characters and then an *x*. This will find permission lists that begin with *d*, specifying a directory, and end in *x*, specifying execute permission for others.

6. Count the number of users who use the Korn shell.

```
$ grep -c /bin/ksh /etc/passwd
```

Power Commands: Join

The Unix *join* command merges corresponding lines of two sorted files based on a common column of data. Learn to use *join* with this example-laden tutorial.

Description

```
join [options] file1 file2
```

Join merges corresponding lines of two files, *file1* and *file2*, containing columns of data (often called fields) that have been sorted using the same sort rule (see the [sort command](#)). If a dash (-) is used in place of *file1* or *file2*, *join* reads from [standard input](#). The results are written to [standard output](#). *Join* merges the files by comparing the entries in a *common field*. By default, the common field is the first field of each file. For all matching entries, *join* writes one occurrence of the common field, then all other fields from *file1* followed by all other fields from *file2*. For example,

```
$ cat test1
andrea 92 A
bobby 87 B+
zach 90 A-
$ cat test2
andrea 89 B+
bobby 94 A
zach 84 B
$ join test1 test2
andrea 92 A 89 B+
bobby 87 B+ 94 A
zach 90 A- 84 B
```

Join options are shown below. In these options, *f* can be 1 or 2 indicating *file1* or *file2*.

| Option | Description |
|------------------|---|
| -tc | <p>Specifies the character, <i>c</i>, that separates fields. Used for input and output. For example, "-t," indicates that commas separate fields. Each occurrence of <i>c</i> is significant so <i>cc</i> represents an empty field. For example, if the separator character is a comma then in "a, ,d" field one is "a", field two is empty and field three is "d".</p> <p>When <i>-t</i> is not used, any whitespace is considered a separator. In this case, multiple occurrences of whitespace are not significant. For both "a<space>b" and "a<space><space>b", field one is "a" and field two is "b".</p> |
| -jf n | <p>Specifies the common fields that are used for merging. Joining is done on the <i>n</i>th field of file <i>f</i>. For example, "-j1 2 -j2 4" joins by comparing the second field of <i>file1</i> to the fourth field of <i>file2</i>.</p> <p>If <i>f</i> is omitted, join on the <i>n</i>th field of both files. For example, "-j 2" joins by comparing the second field of <i>file1</i> to the second field of <i>file2</i>.</p> <p>By default, <i>join</i> merges on the first field of both files.</p> <p>Note: You can only specify one field for each file. For example, "-j1 2 -j 3" specifies field two for <i>file1</i> then field three for <i>file1</i> and <i>file2</i>. In this case, only the last specification, "-j 3", is used.</p> |
| -o f.n... | <p>Specifies an output order. Outputs field <i>n</i> from file <i>f</i>. For example, "-o 1.2 2.1 1.3" outputs field two of <i>file1</i> followed by field one of <i>file2</i> then field three of <i>file1</i>. Note: When the <i>-o</i> option is used, the common field is not automatically output. It must be specified like any other field.</p> |

| | |
|---------------|---|
| | If <i>-o</i> is not used, <i>join</i> outputs one occurrence of the common field, then all other fields from <i>file1</i> followed by all other fields from <i>file2</i> |
| -af | Output unpaired lines from file <i>f</i> . For example, " -a1 -a2 " will output unpaired lines from both files. By default, unpaired lines are not output. On some systems, if <i>f</i> is omitted, unpaired lines from both files are output. |
| -e str | Replace empty fields with the string <i>str</i> . Should be used with the <i>-o</i> option. |
| -v f | Instead of the usual output, print only the unpaired lines in file <i>f</i> . For example, " -v 1 -v 2 " outputs the unpaired lines from both files. |

Examples

The following examples use files *test1* and *test2*, which contain a student's name and test grade.

```
$ cat test1
zach 79
karen 83
bobby 92
suzie 85
$ cat test2
karen 91
bobby 84
zach 95
andy 87
```

1. Join *test1* and *test2* by pairing student names. The first step is to sort both files by field one (student name).
2. **\$ sort -k 1 test1 > test1s**
3. **\$ sort -k 1 test2 > test2s**

The files *test1s* and *test2s* contain the data from *test1* and *test2* sorted in alphabetical order by student name. If you are unfamiliar with using *>* to [redirect](#) output to a file, see [Unix 101: Input/Output Redirection](#). For more information on the *sort* command, see [Power Commands: Sort](#).

```
$ join test1s test2s
bobby 92 84
karen 83 91
zach 79 95
```

Notice that *join* does not output unpaired lines. Students who were absent during one of the test days are not part of the output data.

4. Join *test1* and *test2* by student name include unpaired lines from both files.
5. **\$ join -a1 -a2 test1s test2s**
6. andy 87
7. bobby 92 84
8. karen 83 91
9. suzie 85
10. zach 79 95

The "-a1" option includes unpaired lines from file1 (*test1s*) and the "-a2" option includes unpaired lines from file2 (*test2s*).

11. Display the student's who were absent for the first and/or second test.
12. **\$ join -v 1 test1s test2s**
13. suzie 85

The "-v 1" option displays the unpaired lines from file1 (*test1s*). These are students that took test one but missed test two. Likewise, the following command displays students that took test two but missed test one.

```
$ join -v 2 test1s test2s  
andy 87
```

Use the "-v 1" and "-v 2" options together to output students that missed either test one or test two.

```
$ join -v 1 -v 2 test1s test2s  
andy 87  
suzie 85
```

The following examples use the files *employID* and *pay*. The *employID* file contains an employee-id number, first name and last name. The *pay* file contains an employee-id number, salary and year-end bonus. Fields are separated by a colon.

```
$ cat employID  
1001:Jane:Doe  
1002:Micheal:Smith  
1003:Monica:Sandburg  
1004:Angel:Gonzalez  
1005:Blair:Hammond  
$ cat pay  
1001:40,000:400  
1002:45,000:450  
1003:35,000:350  
1004:22,000:220  
1005:39,000:390
```

-
14. Join *employID* and *pay* on the employee-id field.
 15. **\$ join -t: employID pay**
 16. 1001:Jane Doe:40,000:400
 17. 1002:Micheal:Smith:45,000:450
 18. 1003:Monica:Sandburg:35,000:350
 19. 1004:Angel:Gonzalez:22,000:220
 20. 1005:Blair:Hammond:39,000:390

The "-t:" option tells *join* that a colon separates fields. Notice that the output order is the common field (employee-id) followed by all other fields from file1 (*employID*) then all other fields from file2 (*pay*).

21. Join *employID* and *pay* on the *employee-id* field, output only the employee last name and salary.
22. **\$ join -t: -o 1.3 2.2 employID pay**
23. Doe:40,000
24. Smith:45,000
25. Sandburg:35,000
26. Gonzalez:22,000
27. Hammond:39,000

The "-o 1.3 2.2" option outputs the third field of file1 (last name from *employID*) followed by the second field of file2 (yearly salary from *pay*).

The following example uses files *grades01* and *grades02*, which contain the date, student's name and test score.

```
$ cat grades01
Dec 30 2000 Aaron Zach 79
Dec 30 2000 Jones Karen 83
Dec 30 2000 Smith Bob 92
$ cat grades02
Feb 4 2001 Aaron Zach 91
Feb 4 2001 Jones Karen 72
Feb 4 2001 Smith Bob 84
```

28. Join *grades01* and *grades02* so that the output contains last name, first name, Dec 30 2000 test score and Feb 4 2001 test score.
29. **\$ join -j 4 -o 1.4 1.5 1.6 2.6 grades01 grades02**
30. Aaron Zach 79 91
31. Jones Karen 83 72
32. Smith Bob 92 84

The option "-j 4" joins on field four of both files. The option "-o 1.4 1.5 1.6 2.6" outputs fields four, five and six from *grades01* followed by field six from *grades02*.

Advanced Examples

Substitution Example

The following example uses the file *mf*, which contains a list of names and an M or F for male or female.

```
$ cat mf
andy M
jane F
jim M
michelle F
john M
```

```
sue F
sharon F
```

1. Replace M with the number one and F with the number two. First create a file *trans* which contains the following text
2. F 2
3. M 1

Next, sort *mf* by the second field.

```
$ sort -k 2 mf > mfs
$ cat mfs
jane F
michelle F
sharon F
sue F
andy M
jim M
john M
```

Now join field two of *mfs* with field one of *trans* and output only the name and number.

```
$ join -j1 2 -j2 1 -o 1.1 2.2 mfs trans
jane 2
michelle 2
sharon 2
sue 2
andy 1
jim 1
john 1
```

This can be done without creating the file *mfs*.

```
$ sort -k 2 mf | join -j1 2 -j2 1 -o 1.1 2.2 - trans
```

The [pipe](#) uses the [standard output](#) from the *sort* command as [standard input](#) to the *join* command. The - tells *join* to use standard input as file1.

You might also want to resort the output by name.

```
$ sort -k 2 mf | join -j1 2 -j2 1 -o 1.1 2.2 - trans \
| sort -k 1
```

Note: The backslash (\) is a [quoting character](#) used to [spread one command over several lines](#).

Different Separators

The following example uses the files *f1* and *f2*.

```
$ cat f1
aa 1
bb 2
cc 3
$ cat f2
aa,4,7
bb,5,8
```

1. Join *f1* and *f2* using field one. *Join* requires that both input files use the same field separator so one of the files must be edited. The following example uses [sed](#) to replace every instance of a comma in *f2* with a space.
2. `$ sed 's/,/ /g' f2 > f2out`
3. `$ join f1 f2out`
4. aa 1 4 7
5. bb 2 5 8
6. cc 3 6 9

Formatting Output

The following example uses the files *employID* and *pay*. The *employID* file contains an employee-id number, first name and last name. The *pay* file contains an employee-id number, salary and year-end bonus.

```
$ cat employID
1001 Jane Doe
1002 Micheal Smith
1003 Monica Sandburg
$ cat pay
1001 40,000 400
1002 145,000 1450
1003 35,000 99
```

1. Join *employID* and *pay* on the employee-id field.
2. `$ join employID pay`
3. 1001 Jane Doe 40,000 400
4. 1002 Micheal Smith 145,000 1450
5. 1003 Monica Sandburg 35,000 99

No matter how whitespace is used in the input files, *join* will use only one space to separate fields in the output. The following example uses [awk](#) to make the output look nicer.

```
$ join employID pay | \
awk '{printf("%-5s %-8s %-10s %8s %7s\n", \
$1, $2, $3, $4, $5)}'
```

| | | | | |
|------|---------|----------|---------|------|
| 1001 | Jane | Doe | 40,000 | 400 |
| 1002 | Micheal | Smith | 145,000 | 1450 |
| 1003 | Monica | Sandburg | 35,000 | 99 |

Formatting Unpaired Output

The following examples use files *hair1* and *eye2*.

```
$ cat hair1
andrea brown
bob red
jane black
zach blond
$ cat eye2
bob blue
cindy green
```

```
jane brown
zach blue
```

1. Join files *hair1* and *eye2* including unpaired lines from both in output.
2. **\$ join -a1 -a2 hair1 eye2**
3. andrea brown
4. bob red blue
5. cindy green
6. jane black brown
7. zach blond blue

Because the output includes unpaired lines, hair and eye color are not in distinct fields. Cindy's eye color, green, is placed in field two because she has no hair color.

8. Use the *-o* option with the *-e* option to put *NA* in fields with missing data so that hair and eye color are in the correct columns.
9. **\$ join -a1 -a2 -o 1.1 1.2 2.2 -e NA hair1 eye2**
10. andrea brown NA
11. bob red blue
12. NA NA green
13. jane black brown
14. zach blond blue

This output is not ideal. The name *cindy* is replaced with *NA* because the name field is read from *hair1* and *cindy* does not have an entry in *hair1*.

15. To fix the output in the above example, we need to use the name from field one of *hair1* when an unpaired line exists in the *hair1* file and the name from field one of *eye2* when an unpaired line exists in the *eye2* file. This is tricky but not impossible. First execute
16. **\$ join -a1 -o 1.1 1.2 2.2 -e NA hair1 eye2 > temp**
17. **\$ cat temp**
18. andrea brown NA
19. bob red blue
20. jane black brown
21. zach blond blue

The *join* command prints all paired lines plus unpaired lines from *hair1* in the order name (as read from *hair1*), hair color, eye color and replaces any missing data with *NA*. Output is [redirected to](#) (saved in) the file *temp*. Now execute

```
$ join -v 2 -o 2.1 1.2 2.2 -e NA hair1 eye2 >> temp
$ cat temp
andrea brown NA
bob red blue
jane black brown
zach blond blue
cindy NA green
```

The *join* command prints all unpaired lines from *eye2* in the order name (as read from *eye2*), hair color, eye color and replaces any missing data with *NA*. Note: Because we only output unpaired lines from *eye2*, hair color is always missing. Output is appended to the *temp* file, which now

contains name, hair color and eye color in the correct columns; however, *temp* is no longer sorted by name. To resort *temp*:

```
$ sort -k 1,1 temp
andrea brown NA
bob red blue
cindy NA green
jane black brown
zach blond blue
```

22. The above example can be executed without using a temporary file.

```
23. $ ( join -a1 -o 1.1 1.2 2.2 -e NA hair1 eye2 ; \
24. join -v 2 -o 2.1 1.2 2.2 -e NA hair1 eye2 ) \
25. | sort -k 1,1
26. andrea brown NA
27. bob red blue
28. cindy NA green
29. jane black brown
30. zach blond blue
```

How does this work? The semicolon is used to [string two commands together](#). The parentheses are used to run both commands in a [subshell](#) so that the output can be [simultaneously redirected](#) to the *sort* command. The backslash is a [quoting character](#) used to [spread one command over several lines](#).

31. And now for some real fun...

```
32. $ ( echo NAME HAIR EYES ; \
33. ( join -a1 -o 1.1 1.2 2.2 -e NA hair1 eye2 ; \
34. join -v 2 -o 2.1 1.2 2.2 -e NA hair1 eye2 ) \
35. | sort -k 1,1 ) | awk \
36. '{printf("%-10s %-10s %-10s\n", $1, $2, $3)}'
37. NAME      HAIR      EYES
38. andrea    brown    NA
39. bob       red      blue
40. cindy     NA      green
41. jane     black   brown
    zach     blond   blue
```

Power Commands: Paste

The *paste* command merges corresponding lines of files into vertical columns and prints the results to the screen.

Description

```
paste [-s] [-d char] [Files...]
```

The *paste* command merges corresponding lines of one or more files into vertical columns and prints the results to the screen. For example,

```
$ cat height
5'4"
6'2"
$ cat weight
124lb
180lb
$ paste height weight
5'4" 124lb
6'2" 180lb
```

If one file has fewer lines than another, *paste* will concatenate lines from the longer files with an empty line. For example,

```
$ cat height
5'4"
$ cat weight
124lb
180lb
$ paste height weight
5'4" 124lb
      180lb
```

The command line options for *paste* are shown below.

| Option | Description |
|-----------------------|--|
| -d <i>char</i> | By default, merged lines are <i>delimited</i> or separated by the TAB character. The <i>-d</i> option tells <i>paste</i> to separate columns with the character specified by <i>char</i> . <i>Char</i> can be a regular character or one of the following escape sequences. \n Newline \t Tab \0 (Backslash followed by Zero) Empty string. \\ Backslash Escape sequences should be surrounded by quotes to keep the shell from interpreting them . You can separate columns with different characters by specifying more than one value for <i>char</i> . For example, -d '-*' would separate the first and second columns with a dash (-) and the second and third column with an asterisk (*). If more columns exist, the <i>paste</i> command would alternate between using a dash and an asterisk as a delimiter. |
| -s | Merge all lines from each input file into one line. Each newline in a file, except the last, is replaced with a TAB or a delimiter specified by the -d option. If multiple input files are specified then there will be one line per file printed in the order they are listed on the command line. |

-

If a minus sign (-) is specified as an input file then [standard input](#) is used.

Examples

1. `$ paste file1 file2 file3 > newfile`

Create a new file, *newfile*, with three columns from the files *file1*, *file2* and *file3*. The results of the *paste* command are [redirected](#) from the screen and saved in the file *newfile*.

2. `$ ls | paste -`

List all files in the current directory in one column by [piping](#) the results of the *ls* command to the *paste -* command. The dash (-) specifies that standard input should be used as an input file. This is equivalent to using the *ls* command with the *-l* option.

3. `$ ls | paste - - -`

List all files in the current directory in three columns.

The following examples will use the files *students* and *grades*.

```
$ cat students
```

```
Jenny
```

```
Bobby
```

```
Susan
```

```
Leo
```

```
$ cat grades
```

```
100
```

```
92
```

```
88
```

```
97
```

4. Merge corresponding lines from the files *students* and *grades*.

```
$ paste students grades
```

```
Jenny 100
```

```
Bobby 92
```

```
Susan 88
```

```
Leo 97
```

A **TAB**, the default delimiter, separates the columns.

5. Merge corresponding lines from the files *students* and *grades* and save the results in the file *stugrades*.

```
$ paste students grades > stugrades
```

6. Merge corresponding lines from the files *students* and *grades* separating columns with a single space.

```
$ paste -d ' ' students grades
```

```
Jenny 100
```

```
Bobby 92
Susan 88
Leo 97
```

7. Merge all lines from the file *students* into one line.

```
$ paste -s students
Jenny Bobby Susan Leo
```

The lines become columns and are separated by a **TAB**, the default delimiter.

8. Merge all lines from the file *students* into one line using an asterisk (*) as the delimiter.

```
$ paste -s -d '*' students
Jenny*Bobby*Susan*Leo
```

9. Merge all lines from the file *students* into one line alternating between using an asterisk (*) and an exclamation point (!) as the delimiter.

```
$ paste -s -d '*!' students
Jenny*Bobby!Susan*Leo
```

10. Merge all lines from the file *students* into one line using a **newline** as a delimiter.

```
$ paste -s -d '\n' students
Jenny
Bobby
Susan
Leo
```

This has no effect other than to print the *students* file to the screen because the *-s* option tells *paste* to replace every **newline** with the character specified by the *-d* option, in this case a **newline** character.

11. Merge every two lines in the file *students* into one line.

```
$ paste -s -d '\t\n' students
Jenny Bobby
Susan Leo
```

The *-s* option merges all lines in the file into one while the *-d '\t\n'* option alternates between using a **TAB** and a **newline** as the delimiter character.

12. Merge all lines from the *students* and *grades* files into one line.

```
$ paste -s students grades
Jenny Bobby Susan Leo
100 92 88 97
```

13. Create a file, *stugrades*, containing the first two characters of the names in the *students* file in column one and the numbers from the *grades* file in column two.

```
$ cut -c 1-2 students | paste - grades > stugrades
```

Power Commands: Sort

The Unix *sort* command sorts the lines or columns in a file in alphabetical, numeric or reverse order. Learn to use *sort* with this example-laden tutorial. Sort is a versatile and powerful Unix command; however, it can be a little difficult to learn. To make it easier, I'm going to divide *sort*'s abilities into three categories and cover each one separately.

1. **Simple sort.** Sort the lines of one file in alphabetical, numeric or reverse order.
2. **Column sort.** Sort using one or more fields separated into columns. The sort order of each column can be individually specified.
3. **Merge files.** Files (presorted and unsorted) can be merged by sort order.

Simple Sorts

```
sort [options] [files...]
```

The *sort* command sorts one or more files in alphabetical, numeric or reverse order. The default is alphabetical order. For example,

```
$ cat myfile
Susan
Elizabeth
John
Michael
$ sort myfile
Elizabeth
John
Michael
Susan
```

If no input file is specified, *sort* reads from [standard input](#). The command line options you need to know for simple sorts are shown below.

General Options

| Option | Description |
|--------------------|--|
| -o filename | Saves output in the file, <i>filename</i> . If no file is specified, output is written to standard output . |
| -u | (unique) Identical input lines are output only once. |
| -c | Checks to see if files are already sorted. If so, <i>sort</i> produces no output. If not, <i>sort</i> sends an error message to standard error . |

Sort Order Options

| Option | Description |
|-----------|---|
| -d | Sort in <i>dictionary</i> order. Ignore all characters except letters, digits and blanks when determining sort order. |
| -n | Sort in numerical order (For example: -2.5, -1, 0, 0.54, 3, 18). Numerical order ignores leading spaces when determining the sort order and interprets a leading minus sign (-) as a negative number. Numbers may include a comma to separate thousands (e.g. 1,000 or 10,000). Non-numeric entries are sorted in alphabetical order between zero and positive numbers. Blank lines are sorted between negative numbers and zero. <i>Sort</i> does not interpret a leading plus sign (+) as a positive number, but as the start of a non-numeric entry. |
| -f | Ignore the uppercase/lowercase distinction. (a is the same as A). |
| -M | Order the first three characters as months. (e.g. jan < feb < mar ...). Uppercase letters precede lowercase letters for the same month (e.g. JAN < Jan < jan < FEB) Invalid names are sorted in alphabetical order before valid names. (e.g. misspelled < nomonth < jan). |

| | |
|-----------|---|
| -i | Ignore non-printing characters. Non-printing characters include control characters such as tab, form feed, carriage return, etc. Non-printing characters are those outside the ASCII range 040-176. |
| -r | Reverse the sort order. |

Sort in Alphabetical and Dictionary Order

The next several examples will use the file *fl*.

```
$ cat fl
.this line begins with a period
a line that begins with lowercase a.
This is a line.
abracadabra
1234
Where will this line sort?
A line that begins with uppercase a.
```

1. Sort lines of *fl* in alphabetical order.
2. **\$ sort fl**
3. .this line begins with a period
4. 1234
5. A line that begins with uppercase a.
6. This is a line.
7. Where will this line sort?
8. a line that begins with lowercase a.
9. abracadabra

Notice that spaces and punctuation marks are ordered before numbers followed by the uppercase letters A to Z then lowercase letters a to z.

10. Sort *fl* in alphabetical order and save the output in the file *sfl*.

```
$ sort -o sfl fl
```

11. Sort *fl* in reverse alphabetical order.
12. **\$ sort -r fl**
13. abracadabra
14. a line that begins with lowercase a.
15. Where will this line sort?
16. This is a line.
17. A line that begins with uppercase a.
18. 1234
19. .this line begins with a period
20. Sort *fl* in alphabetical order ignoring the uppercase/lowercase distinction.
21. **\$ sort -f fl**
22. .this line begins with a period
23. 1234
24. a line that begins with lowercase a.
25. A line that begins with uppercase a.
26. abracadabra
27. This is a line.
28. Where will this line sort?
29. Sort *fl* in dictionary order.
30. **\$ sort -d fl**

31. 1234
32. A line that begins with uppercase a.
33. This is a line.
34. Where will this line sort?
35. a line that begins with lowercase a.
36. abracadabra
37. .this line begins with a period

Dictionary order ignores all characters except number, letters and blanks so ".this line begins with a period" is sorted as if it were "this line begins with a period".

38. Sort *fl* in dictionary order, ignore the uppercase/lowercase distinction.
39. **\$ sort -df fl**
40. 1234
41. a line that begins with lowercase a.
42. A line that begins with uppercase a.
43. abracadabra
44. This is a line.
45. .this line begins with a period
46. Where will this line sort?
47. Sort *fl* in reverse dictionary order ignoring the uppercase/lowercase distinction.
48. **\$ sort -dfr fl**
49. Where will this line sort?
50. .this line begins with a period
51. This is a line.
52. abracadabra
53. A line that begins with uppercase a.
54. a line that begins with lowercase a.
55. 1234

This is the exact opposite of using the "*sort -dfl*" command.

Sort in Numeric Order

The next two examples will use the file *n1*.

```
$ cat n1
-18
 18
  0
-1.4
 0.54
 0.0
  3
0.1
```

1. Sort *n1* in numeric order.
2. **\$ sort -n n1**
3. -18
4. -1.4
5. 0
6. 0.0
7. 0.1
8. 0.54
9. 3
10. 18

11. Sort *nl* in alphabetical order.
12. `$ sort nl`
13. 0
14. 0.0
15. 0.54
16. 3
17. -1.4
18. 18
19. -18
20. 0.1

Notice that this is not *mathematically* sorted.

21. For each file in the current directory, list the number of lines in the file. Sort files from those with the most lines to those with the least.
22. `$ wc -l * | sort -rn`

The `wc` command prints the number of lines in a file. Output from `wc` is [piped](#) to the `sort` command where the `-n` option orders numbers from smallest to largest, but the `-r` option reverses the sort order, ordering numbers from largest to smallest.

Sort Months

The next example will use the file *months*.

```
$ cat months  
FEB  
misspelled  
mar  
MAY  
january  
may  
nomonth  
jan  
May
```

Use the `-M` option to sort *months* in chronological order.

```
$ sort -M months  
misspelled  
nomonth  
jan  
january  
FEB  
mar  
MAY  
May  
may
```

Notice that non-months are sorted first and uppercase letters precede lowercase letters for identical months.

Sorting with the Unique Option

The next example will use the file *errlog*.

```
$ cat errlog  
error 01: /tmp directory not found  
error 17: out of memory  
error 01: /tmp directory not found
```

```
error 22: low disk space
error 01: /tmp directory not found
```

1. Sort the file *errlog* in alphabetical order. Identical input lines are output only once.
2. **\$ sort -u errlog**
3. error 01: /tmp directory not found
4. error 17: out of memory
5. error 22: low disk space

Sorting with the Check Option

1. Sorting a large file can be slow. Ironically, it is slower to run *sort* on a presorted file than on an unsorted file. The *-c* option checks to see if a file is already sorted in a specific order. If so, *sort* does nothing. If not, *sort* prints an error message to [standard error](#). For example, assume that the file *alphasorted* is currently in alphabetical order.
2. **\$ sort -c alphasorted**
3. **\$**

This is much quicker than sorting a file that is already in the correct order.

4. Assume that the file *notsorted* is not in alphabetical order.
5. **\$ sort -c notsorted**
6. sort: disorder on notsorted
7. Now try the command.
8. **\$ sort -fc alphasorted**
9. sort: disorder on alphasorted

The *-f* option tells *sort* to ignore the uppercase/lowercase distinction. The *alphasorted* file is sorted in regular alphabetic order with the uppercase/lowercase distinction in place. Therefore, the *-c* option reports disorder.

Column Sorts

Sort can order files by columns (also called fields). For example, the file *f1* has two fields, first name and last name.

```
$ cat f1
Susan Jones
Jill Zane
John Smith
Andrew Carter
```

The following command, sorts *f1* by the second field.

```
$ sort -k 2 f1
Andrew Carter
Susan Jones
John Smith
Jill Zane
```

Command line options you need to know to sort by columns are shown below. These options should be used after [general](#) and [sort order](#) options.

Column Sorting Options

| Option | Description |
|------------|--|
| -tc | Specifies the character, <i>c</i> , that separates fields. For example, " <i>-t ,</i> " indicates that |

| | |
|--------------------------------------|--|
| | commas separate fields. Each occurrence of <i>c</i> is significant so <i>cc</i> represents an empty field. For example, if the separator character is a comma then in "a , , d" field one is "a", field two is empty and field three is "d". The default separator is any whitespace. |
| -b | Ignore leading whitespace (spaces and tabs) when determining the starting character of columns. When whitespace is used to separate columns, the <i>-b</i> option overrides the significance of multiple column separators. For example, "c" is the first character of the second field of "ab<space>cd", "ab<space><space>cd", and "ab<space><space><space>cd". |
| -k <i>Start</i>[,<i>End</i>] | Defines a <i>sort key</i> or a section of each line used for ordering. The sort key will begin with the field <i>Start</i> and end with the field <i>End</i> . If <i>End</i> is not specified, the key begins with <i>Start</i> and continues to the end of the line. More details on specifying sort keys below. |

Sort Key Specification for -k Option

Start and *End* are specified using the format *FNum*[.*CNum*][*type*] where *FNum* is the field number, starting from 1, and *CNum*, if present, is the character within the field. [The *type* modifier is described later.](#) For example,

| | |
|------------|---|
| -k 1 | Sort starting with the first character of the first field and continuing through the end of the line. This is the same as a simple sort. |
| -k 1,1 | Sorts by the first field only. The ordering of lines with identical first fields is unspecified (random). Notice that this is different than the above example. |
| -k 1,3 | Sort starting with the first character of the first field and ending with the final character of the third field. |
| -k 1.2 | Sorts starting from the second character in the first field and continuing through the end of the line. |
| -k 1.3,3.3 | Sort key starts with the 3 rd character in the first field and ending with the 3 rd character in the 3 rd field. More on specifying <i>CNum</i> below. |

Any number of field specifications can be used in a sort. For example,

| | |
|----------------------|---|
| -k 3,5 -k 2,2 | Sorts by field three through five then two. |
| -k 1,1 -k 2,2 -k 3,3 | Sorts by field one. If field one is identical, sort by field two. If fields one and two are identical, sort by field three. |
| -k 1,3 | Sorts by field one through three. Note that this is different than the above example. |

Type Modifiers

A type modifier can be added to *Start* or *End* to change the default sort order for the sort key. Type modifiers are one or more of the following letters: d, f, i, M, n, or r. The effect is the same as corresponding [sort order option](#) (*-d*, *-f*, etc.) except that only the ordering of sort key specified after *-k* is affected. These type modifiers can be applied to *Start*, *End* or both. The effect is the same. For example,

| | |
|-------------------|--|
| -k 1n | Sort by the entire line (field one until the end) using numerical sort order. |
| -n -k 1 | Identical to the above example. |
| -n -k 3,3 -k 1,1 | Sorts by the third field. If the third field is identical, sort by the first field. Both sorts are in numerical order. |
| -k 3,3n -k 1,1n | Identical to the above example. |
| -k 3n,3 -k 1n,1 | Identical to the above example. |
| -k 3n,3n -k 1n,1n | Identical to the above example. |

| | |
|-----------------------------|---|
| <code>-k 3,3n -k 1,1</code> | Sorts by the third field using numerical ordering. If the third field is identical, sorts by the first field using the default alphabetical ordering. |
|-----------------------------|---|

Once a type modifier has been attached to a sort key specification, other simple sort order options are ignored for that sort key. For example,

| | |
|---------------------------------|--|
| <code>-df -k 2,2</code> | Sort by field two in dictionary order ignoring the uppercase/lowercase distinction. |
| <code>-k 2,2df</code> | Same as above example. |
| <code>-f -k 2,2d</code> | Sort by field two in dictionary order, but do not apply the <i>-f</i> option to the <i>-k 2,2d</i> sort key. Case will be relevant when ordering by field two. |
| <code>-f -k 2,2df -k 3,3</code> | Sort by field two in dictionary order ignoring the uppercase/lowercase distinction. If field two is identical, sort by field three ignoring the uppercase/lowercase distinction, but using the default alphabetical rather than dictionary ordering. |

The *b* type modifier, like the *-b* option, causes *sort* to ignore blank characters when determining field and character positions. Unlike other type modifiers, the *b* modifier affects *Start* and *End* separately.

| | |
|------------------------|--|
| <code>-b -k 2,3</code> | Sort by field two through three. Ignore leading whitespace when determining the starting character of field two and field three. |
| <code>-k 2,3bd</code> | Sort by field two through three both in dictionary order. Leading whitespace will be ignored when determining the starting character of field three but not field two. You probably don't want to do this. |
| <code>-k 2b,3bd</code> | Sort by field two then three both in dictionary order. Ignore leading whitespace when determining the starting character for field two and three. |

More on Specifying *CNum*

When counting field characters, *sort* is sensitive the number and type of separation characters used between fields. Generally, it will begin counting characters in a field *after* it reaches the first separation character specified by the *-t* option. This makes sense if you use a field separator like a comma. For example,

```
collrow1,12345678
col2row2,abcdefgh
```

| | |
|-----------------------------|---|
| <code>-t, -k 2.2,2.4</code> | Sorts beginning with the character "2" in row one and "b" in row two and ending with the character "4" in row one and "d" in row two. |
|-----------------------------|---|

In the following example, there is a space between the field separator, a comma, and the useful data in field two.

```
collrow1, 12345678
col2row2, abcdefgh
```

| | |
|--------------------------|--|
| <code>-t, -k 2.3</code> | Sorts beginning with the characters "2" and "b". The first characters are the leading spaces after the comma, the second characters are "1" and "a". |
| <code>-t, -k 2.2b</code> | Identical to the above example. The <i>b</i> modifier ignores leading spaces so the first characters are "1" and "a". |

It is more confusing when whitespace is used to separate columns. When no explicit field separator is specified with the *-t* option, *sort* will use any whitespace as a field separator. It will also count this whitespace as a character in the next field. For example, say you have a file with two columns separated by one space.

```
collrow1 12345678
collrow2 abcdefgh
```

| | |
|---------------|---|
| -t " " -k 2.1 | Sorts beginning with the characters "l" and "a". Because the space was explicitly specified as the field separator, <i>sort</i> begins counting field characters after it. |
| -k 2.2 | Sorts beginning with the characters "l" and "a". Because the space was not explicitly specified as a field separator, <i>sort</i> counts the space separating fields one and two as the first character of field two, even though it is a field separator by default and won't affect the sort order. |
| -k 2.1b | Sorts beginning with the characters "l" and "a". As we saw in the last example, <i>sort</i> would normally count the separator space as a field character; however, the b type modifier tells it not to include leading whitespace when counting characters. |

Examples: Column Sorts

The next several examples will use the file *grades*, which contains the date (month, day, year), student's first name, last name and test grade.

```
$ cat grades
Dec 30 2000 Smith Bob 92
Dec 30 2000 Jones Karen 83
Dec 30 2000 Smith John 78
Dec 30 2000 Sandburg Sara 85
Feb 4 2001 Smith Bob 84
Feb 4 2001 Smith John 92
Feb 4 2001 Sandburg Sara 91
Feb 4 2001 Jones Karen 72
```

1. Sort *grades* by putting the student's last name (4th field) in alphabetical order.
2. **`$ sort -k 4 grades`**
3. Feb 4 2001 Jones Karen 72
4. Dec 30 2000 Jones Karen 83
5. Dec 30 2000 Sandburg Sara 85
6. Feb 4 2001 Sandburg Sara 91
7. Feb 4 2001 Smith Bob 84
8. Dec 30 2000 Smith Bob 92
9. Dec 30 2000 Smith John 78
10. Feb 4 2001 Smith John 92

Because no ending field was specified, the file was sorted starting with the 4th column and ending with the final column. Therefore, the first name and grade are included in the sort. If we use the following command

```
$ sort -k 4,4 grades
Dec 30 2000 Jones Karen 83
Feb 4 2001 Jones Karen 72
Dec 30 2000 Sandburg Sara 85
Feb 4 2001 Sandburg Sara 91
Dec 30 2000 Smith Bob 92
Dec 30 2000 Smith John 78
Feb 4 2001 Smith Bob 84
Feb 4 2001 Smith John 92
```

only the 4th column is used in the sort. The output is *not* sorted by first name or grade.

11. Sort *grades* by putting the student's last name (4th field) in alphabetical order. Save the output in the file *sgrades*.

```
$ sort -o sgrades -k 4 grades
```

12. Sort *grades* from the highest to the lowest test score.

```
13. $ sort -nr -k 6,6 grades  
14. Feb 4 2001 Smith John 92  
15. Dec 30 2000 Smith Bob 92  
16. Feb 4 2001 Sandburg Sara 91  
17. Dec 30 2000 Sandburg Sara 85  
18. Feb 4 2001 Smith Bob 84  
19. Dec 30 2000 Jones Karen 83  
20. Dec 30 2000 Smith John 78  
21. Feb 4 2001 Jones Karen 72
```

The *-k 6,6* option sorts by the 6th column. The *-n* option sorts in numerical order (lowest to highest) and the *-r* option reverse the sort (highest to lowest). The following command is equivalent.

```
$ sort -k 6,6nr grades
```

22. Sort *grades* by student name, last name and first name, and then the test date, year followed by month followed by day.

```
23. $ sort -k 4,5 -k 3,3n -k 1,1M -k 2,2n grades  
24. Dec 30 2000 Jones Karen 83  
25. Feb 4 2001 Jones Karen 72  
26. Dec 30 2000 Sandburg Sara 85  
27. Feb 4 2001 Sandburg Sara 91  
28. Dec 30 2000 Smith Bob 92  
29. Feb 4 2001 Smith Bob 84  
30. Dec 30 2000 Smith John 78  
31. Feb 4 2001 Smith John 92
```

Note that the names are sorted in alphabetical order, the year and day are sorted in numeric order and the months are sorted chronologically as months.

Save the output from the above sort in a file named *g2*.

```
$ sort -o g2 -k 4,5 -k 3,3n -k 1,1M -k 2,2n grades
```

Use the check option (*-c*) to determine if *grades* or *g2* is presorted by student name and test date.

```
$ sort -c -k 4,5 -k 3,3n -k 1,1M -k 2,2n grades  
sort: disorder on grades  
$ sort -c -k 4,5 -k 3,3n -k 1,1M -k 2,2n g2  
$
```

The next example uses the file *f1*.

```
$ cat f1  
.this line begins with a period
```

a line that begins with lowercase a.
This line begins with spaces.
abracadabra
1234
Where will this line sort?
A line that begins with uppercase a.

32. Sort *fl* in dictionary order ignoring the uppercase/lowercase distinction and leading spaces.
33. **\$ sort -dfb -k 1 fl**
34. 1234
35. a line that begins with lowercase a.
36. A line that begins with uppercase a.
37. abracadabra
38. .this line begins with a period
39. This line begins with spaces.
40. Where will this line sort?

This is a trick to do a simple sort ignoring leading blank characters required because the *-b* option only affects column ordering. However, on some systems the *-b* option will affect simple sorts as well. On these systems the following command is identical.

```
$ sort -dfb fl
```

The next few examples will use the file *nums*, which use a colon as a field separator.

```
$ cat nums  
3:18  
12:5  
3:22  
8:5  
12:5
```

41. Sort *nums* in numeric order by field one. If field one is identical, sort by field two. Use a colon as a field separator.
42. **\$ sort -n -t":" -k 1,1 -k 2,2 nums**
43. 3:18
44. 3:22
45. 8:5
46. 12:5
47. 12:5

Repeat the same sort using the unique (*-u*) option. Identical input lines are output only once.

```
$ sort -un -t":" -k 1,1 -k 2,2 nums  
3:18  
3:22  
8:5  
12:5
```

48. Try the following command.

```

49. $ sort -n -t":" -k 1,2 nums
50. 3:22
51. 3:18
52. 8: 5
53. 12:5
54. 12:5

```

Are you surprised that "3:22" comes before "3:18"? This occurs because "-k 1,2" combines fields one and two before it sorts creating two strings "3:22" and "3:18". These are not recognized as numbers so they are sorted in alphabetical order despite the -n option. Because there is a space in front of "3:22", it is order first alphabetically.

Advanced Examples: Column Sorts

The next example will use the file *jnames*. *Jnames* contains first name, last names and a middle initial justified using spaces.

```

$ cat jnames
Mike      Smith      C
TJ        Zane       R
Sampson   Elliot     T
tj        Meyers     D
Bobby     smith      A

```

- Sort *jnames* by the second through third fields, last name and middle initial. Ignore the uppercase/lowercase distinction.
- `$ sort -k 2,3f jnames`
- tj Meyers D
- TJ Zane R
- Mike Smith C
- Bobby smith A
- Sampson Elliot T

Why is *Zane* ordered before *Smith*? Because we didn't tell *sort* to ignore leading whitespace. *Sort* is actually ordering "<space><space><space><space><space>Zane" before "<space><space><space><space>Smith" because it has more leading spaces. To fix this problem use the *b* type modifier.

```

$ sort -k 2b,3bf jnames
Sampson   Elliot     T
tj        Meyers     D
Bobby     smith      A
Mike      Smith      C
TJ        Zane       R

```

The following commands are also problematic.

```

$ sort -b -k 2,3f jnames

```

because the type modifier `f` is added to the sort key specification "`-k 2,3f`", `sort` does not apply the `-b` sort order option when sorting by that key.

```
$ sort -k 2,3bf jnames
```

because the `b` type modifier will only affect field three.

The next two examples use the file `tms`, which contains a weekday specification in field one followed by a `hour:minute:second` time specification in field two. Fields are separated by a `TAB`.

```
$ cat tms
Wed      02:43:55
Tue      14:46:32
Wed      11:43:13
```

8. Sort `tms` by the minutes.
9. `$ sort -k 2.4b,2.5bn tms`
10. Wed 02:43:55
11. Wed 11:43:13
12. Tue 14:46:32

Note: I am using the `b` type modifier so that no whitespace is included when counting the character position.

13. Sort `tms` by the minutes followed by the seconds.
 14. `$ sort -k 2.4b,2.5bn -k 2.7b,2.8bn tms`
 15. Wed 11:43:13
 16. Wed 02:43:55
 17. Tue 14:46:32
-

The next two examples use the file `kids`, which contains the first name, last name and age of three children.

```
$ cat kids
First Name Last Name Age
Susan      Jones      6
Elizabeth  Zane      11
Michael    Crown     8
```

The first line of the file has column names rather than data.

18. Sort the data from the `kids` file by age. Ignore the column names.
19. `$ tail +2 kids | sort -k 3n`
20. Susan Jones 6
21. Michael Crown 8
22. Elizabeth Zane 11

The "`tail +2 kids`" command prints the contents of the `kids` file, starting with the second line, to [standard output](#). The [pipe](#) (`|`) redirects the standard output from `tail` to the `sort` command, which then sorts by field three in numerical order.

23. Create a file, `skids`, that contains the data from `kids` sorted by age. Include the column names at the top of the `skids` file but do not include them in the sort.
24. `$ (head -1 kids ; tail +2 kids | sort -k 3n) > skids`

How does this work? The semicolon is used to [type two commands on the same line](#). The first command, "`head -1 kids`", prints the first line of the `kids` file, the column names, to standard output. The second command "`tail +2 kids | sort -k 3n`" sorts the data in `kids` by age and prints the results to standard output. The parentheses are used to run both commands in a [subshell](#) so that the output can be [simultaneously redirected](#) to the `skids` file.

Merging

Files (presorted and unsorted) can be merged by sort order. For example, assume you have two files, `f1` and `f2`,

```
$ sort -o soutput f1 f2
```

merges files `f1` and `f2`, sorts them and saves the output in the file `soutput`. It is equivalent to executing the following commands.

```
$ cat f1 f2 > f3
```

```
$ sort -o soutput f3
```

```
$ rm f3
```

One command line option affects file merging.

| Option | Description |
|-----------------|---|
| <code>-m</code> | Merge only. Use on presorted input files. |

For example, if two files `sf1` and `sf2` are already sorted

```
$ sort f1 -o sf1
```

```
$ sort f2 -o sf2
```

then

```
$ sort -m -o soutput sf1 sf2
```

saves time by not resorting `sf1` and `sf2`. It just integrates them.

Examples: Merging

The following two examples use files `t1` and `t2`.

```
$ cat t1
```

```
A - from file 1
```

```
C - from file 1
```

```
E - from file 1
```

```
$ cat t2
```

```
B - from file 2
```

```
D - from file 2
```

1. Merge files `t1` and `t2` in alphabetical order.
2. `$ sort -m t1 t2`
3. A - from file 1
4. B - from file 2
5. C - from file 1
6. D - from file 2
7. E - from file 1

The *-m* (merge only) option is appropriate because *t1* and *t2* are already sorted in alphabetical order.

8. Using the *-m* option with a file that is not presorted in the correct order will result in disordered output. For example, merge *t1* and *t2* in reverse alphabetical order.
9. **\$ sort -r -m t1 t2**
10. B - from file 2
11. D - from file 2
12. A - from file 1
13. C - from file 1
14. E - from file 1

You can use the *-c* option to check if files are sorted in the correct order before deciding to use *-m*.

```
$ sort -c -r t1 ; sort -c -r t2
sort: disorder on t1
sort: disorder on t2
```

Because the check option reported disorder, the files need to be sorted as well as merged.

```
$ sort -r t1 t2
E - from file 1
D - from file 2
C - from file 1
B - from file 2
A - from file 1
```

The following example uses files *f1* and *f2*.

```
$ cat f1
.this line begins with a period
a line that begins with lowercase a.
Where will this line sort?
A line that begins with uppercase a.
$ cat f2
This is a line.
abracadabra
1234
```

15. Merge and sort files *f1* and *f2* in dictionary order, ignoring the uppercase/lowercase distinction.
16. **\$ sort -fd f1 f2**
17. 1234
18. a line that begins with lowercase a.
19. A line that begins with uppercase a.
20. abracadabra
21. This is a line.
22. .this line begins with a period
23. Where will this line sort?

Two equivalent ways to merge and sort these files are:

```
$ cat f1 f2 > f3
$ sort -fd f3
```

and

```
$ sort -fd -o sf1 f1
$ sort -fd -o sf2 f2
$ sort -fdm sf1 sf2
```

24. The above examples, have only merged two files; however *sort* can merge more than two files.
25. `$ sort f1 f2 f3 f4 f5 ...`

The following two examples use files *grade1* and *grade2*.

```
$ cat grade1
Smith Bob 92
Jones Karen 83
Smith John 78
Sandburg Sara 85
$ cat grade2
Smith Bob 84
Smith John 92
Sandburg Sara 91
Jones Karen 72
```

Grade1 contains three fields, last name, first name and a grade for the first test of the year.
Grade2 contains the same data for the second test of the year.

-
26. Sort and merge *grade1* and *grade2* by the name.
27. `$ sort -k 1,2 grade1 grade2`
28. Jones Karen 72
29. Jones Karen 83
30. Sandburg Sara 85
31. Sandburg Sara 91
32. Smith Bob 84
33. Smith Bob 92
34. Smith John 78
35. Smith John 92
36. You may want to produce output that has one line per student with both grades, e.g.
37. Jones Karen 83 72
38. Sandburg Sara 85 91
39. Smith Bob 92 84
40. Smith John 78 92

This kind of merging or joining is not provided by the *sort* command. For this example, you can accomplish this task using the *sort* command along with the [*cut*](#) and [*paste*](#) commands.

```
$ sort -o sg1 -k 1,2 grade1
$ sort -o sg2 -k 1,2 grade2
$ cut -d" " -f 3 sg2 | paste sg1 -
```

The [*join*](#) command provides a more advanced means of merging by columns.

Power Commands: Tr

The Unix *tr*, translate characters, command can be used to substitute, compress or delete characters in a file. Learn to use *tr* with this example-laden tutorial.

Description

```
tr [options] string1 [string2]
```

The *tr* command copies text from [standard input](#), replaces characters that match *string1* with characters from *string2* or replaces multiple occurrences of characters in *string1* with a single character or deletes character in *string1* then prints the results to [standard output](#). For example,

```
$ tr "abc" "xyz" < infile > outfile
```

replaces the character *a* with *x*, *b* with *y* and *c* with *z* in the file *infile* and saves the results in the file *outfile*. It does not require "abc" occur together for the substitutions to occur. The string "about cats" would be translated to "xyout zxts". As in this example, the *tr* command is often used with [input/output redirection](#).

The command line options for *tr* are shown below.

| Option | Description |
|-----------|---|
| -s | Compress repeated characters in <i>string1</i> . Normally the <i>-s</i> option is not used with the <i>-d</i> option or with a second string specification (<i>string2</i>). For example, <pre>tr -s " " < infile</pre> will replace all multiple occurrences of spaces with a single space in the file <i>infile</i> . |
| -d | Delete characters in <i>string1</i> . Normally the <i>-d</i> option is not used with the <i>-s</i> option or with a second string specification (<i>string2</i>). For example, <pre>tr -d "!" < infile</pre> will delete all exclamation points in the file <i>infile</i> . |
| -c | Use the compliment of the characters in <i>string1</i> . (i.e. every character except those in <i>string1</i>). |

Specifying Strings

- When specified, *string2* should be the same length as *string1*. Each character in *string1* will be substituted with a corresponding character in *string2*.
- String specifications should be surrounded by [quotes](#) to keep the shell from interpreting special characters.
- In some older Unix flavors, all strings must be enclosed in square brackets [].

The *tr* command supports several advanced features when specifying characters for *string1* and *string2*. The following is a list of potential string specifications.

c

Any keyboard character: alphabetic, numeric or symbol.

c-c

Specifies a range of characters. For example, *a-d* includes the characters *a,b,c* and *d*.

\c

An escape sequence. Valid escape sequences include:

** backslash

\n newline

\r carriage return

\t tab

\v vertical tab

\f form feed

[*:class:*]

Specify a class of characters. Valid classes are:

alnum alphabetic or numeric characters
alpha alphabetic characters [A-Za-z]
lower lower case characters [a-z]
upper upper case characters [A-Z]
digit numeric characters [0-9]
blank tab or a space
space white-space characters including the a space, form feed, newline, carriage return, tab and vertical tab.
punct punctuation characters [~!@#\$%^&* ()_+ | { } " : < > ? ` - = \ [] ; ' / . ,]
cntrl [control characters](#) - tab, newline, form feed, carriage return, etc.
print printable character - includes the space character but no control characters

The character classes *upper* and *lower* can be used to convert from upper to lower case characters. For example

```
tr "[:upper:]" "[:lower:]"
```

Otherwise, character classes are not available to define the replacement characters in a substitution. They can only used to define *string1*.

[*c*n*]

Represents *n* repetitions of the characters *c*. Only valid when specifying *string2*. For example, "[a*3]" is equivalent to "aaa". If *n* is omitted *c* will be repeated enough times that *string2* is the same length as *string1*.

[*=equiv=*]

All characters in the same equivalency class as *equiv*. Equivalence class are sets of characters that are naturally grouped together. For example, all accented letters like ò ó ô ö based on the same base letter, in this case o. Equivalency classes can only be used when specifying *string1*. They are not available to define the replacement characters in a substitution.

Examples

1. The *tr* command reads input from [standard input](#) and sends the results of the command to [standard output](#). It is commonly used with [input/output redirection](#). For example,

```
$ tr "[]" "(" < infile > outfile
```

replaces all square brackets with parenthesis in the file *infile* and saves the results in the file *outfile*.

2. To edit a file using the *tr* command requires two steps. First translate the characters in the file and save the output in a temporary file.

```
$ tr "[]" "(" < infile > tempfl
```

Second, replace the original file with the temporary file.

```
$ mv tempfl infile
```

The remaining examples show the *tr* command without specifying the input and output files.

Examples: Compressing Characters

1.

```
$ tr -s "ab"
```

Compress all multiple occurrences of the characters *a* and *b* into one. The string "abaabbaaabbb" would be replaced with "ababab".

2. `$ tr -s "\n"`

Replace all multiple occurrences of the newline (`\n`) character. This will convert a file with double, triple (or more) line spacing to a file with single line spacing.

3. `$ tr -s " \t"`

Compress all multiple occurrences of a space or tab (`\t`) character.

4. `$ tr -s "[:blank:]"`

Compress all multiple occurrence of characters in the *blank* class. Because the *blank* class includes only the space and tab (`\t`), this is the same as the above example.

Examples: Deleting Characters

1. `$ tr -d "x"`

Delete all occurrences of the character *x*.

2. `$ tr -d "\t\f"`

Delete all tabs (`\t`) and form feed (`\f`) characters.

3. `$ tr -dc "[:print:]"`

Delete all characters that are not in the character class *print*. The `-c` option specifies the compliment and the *print* character class specifies all characters that can be printed.

4. `$ tr -dc "[:alnum:][:space:]"`

Delete all characters that are not alphabetic, numeric or white-space characters.

Examples: Substitute Characters

1. `$ tr "abc" "xyz"`

Replaces the character *a* with *x*, *b* with *y* and *c* with *z*. It does not require "abc" occur together for the substitutions to occur. The string "about cats" would be translated to "xyout zxts".

2. `$ tr "\t" " "`

Replace all tabs (`\t`) with a space.

3. `$ tr "[A-Z]" "[a-z]"`

Translate uppercase to lowercase. This can also be done using the *upper* and *lower* character class specifications.

- ```
$ tr "[:upper:]" "[:lower:]"
4. $ tr "0123456789" "dddddddddd"
```

Replace all occurrences of a digit *0* to *9* with the letter *d*. There are several equivalent ways of executing this command.

```
$ tr "[0-9]" "[d*10]"
```

The range `[0-9]` is used instead of typing out all ten digits. The specification `[d*10]` means repeat the *d* character ten times.

```
$ tr "[:digit:]" "[d*]"
```

The character class *digit* is used to specify all ten digits. The specification `[d*]` means repeat the *d* character as many times as necessary for *string2* to equal *string1* in length.

- ```
5. $ tr -c "[:space:]" "[x*]"
```

Replace any character which is not in the character class *space* with the letter *x*.

- ```
6. tr "[=o=]" o
```

Substitute the letter *o* for all characters that are in the *o* equivalent class. This can be used to remove any diacritical marks.

7. Substitution and compression can be done with one command.

```
$ tr -s "ab" "xy"
```

Replace *a* with *x* and *b* with *y*. Then compress all multiple occurrences of *x* and *y*. This would translate the string *aaabb* to *xy*. The above command is equivalent to the following two commands.

- ```
$ tr "ab" "xy"  
$ tr -s "xy"  
8. $ tr -cs "[:alnum:]" "[\n*]"
```

Replace all characters that are not alphabetic or numeric with a newline characters. Compress all multiple newline characters into one newline characters. This will print one word per line.

Advanced Examples

- ```
1. $ echo $PATH | tr ":" "\n"
```

Print each directory in your path on one line.

2. Suppose you have a group files in the current directory that you want to execute the same *tr* command on. For example, you want to compress all multiple occurrence of white-space in every file whose filename ends with *.txt*. This can be done using a shell loop. The loop's format is [shell dependent](#).

### C-Shell

If you are using the c-shell or tc-shell the following command will work.

```
foreach f (*.txt)
 cp $f $f.bak
 tr -s "[:space:]" < $f.bak > $f
end
```

Note: You may need to unset the [noclobber option](#) to use the above command.

```
% unset noclobber
```

### Bourne, Korn, Bash and Z-Shell

If you are using the Bourne shell or a derivative (including the Korn, bash or z-shell) the following command will work.

```
for f in *.txt; do
 cp $f $f.bak
 tr -s "[:space:]" < $f.bak > $f
done
```

Note: You may need to turn off the [noclobber option](#) to use the above command.

```
$ set +o noclobber
```

In addition to converting each file, this command will create *filename.bak*, a copy of the original. Add the line *"rm \$f"* to the loop to remove the backup files.

## Power Commands: Uniq

---

The *uniq* command removes duplicate lines from a file. It is often used as part of a filter. Learn to use *uniq* to its fullest potential with this example-laden tutorial.

### Description

```
uniq [options] file1 file2
```

*Uniq* removes duplicate lines from *file1* and writes a single copy of each line to *file2*. If *file2* exists, *uniq* overwrites the file without warning. If *file2* is not specified, *uniq* writes to [standard output](#). If *file1* is not specified, *uniq* reads from [standard input](#). For example,

```
$ cat fruits
apples
apples
oranges
pears
$ uniq fruits
apples
oranges
pears
```

*Uniq* is only useful if the input file is pre-sorted. In the following example

```
$ cat fruits
apples
oranges
apples
$ uniq fruits
apples
oranges
apples
```

*uniq* does not remove the second *apples* line because it is not right after the first *apples* line. For more information on sorting files see [Power Commands: Sort](#).

### Options

| Option    | Description                                                                                                                                                                                                                                                                                                                                                   |
|-----------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>-c</b> | Writes the number of times a line occurs in the input file before each line in the output file.                                                                                                                                                                                                                                                               |
| <b>-d</b> | Writes each duplicate line once but does not output any unique lines.                                                                                                                                                                                                                                                                                         |
| <b>-u</b> | Writes only unique lines. All duplicate lines are discarded.                                                                                                                                                                                                                                                                                                  |
| <b>-n</b> | Ignores the first <i>n</i> fields of a line. Fields are delimited by spaces or tabs. Note: in some Unix flavors including <a href="#">Solaris</a> this option can be specified as "-f <i>n</i> "                                                                                                                                                              |
| <b>+n</b> | Ignores the first <i>n</i> characters of a line or field. When used with the -n option, +n ignores the first <i>n</i> characters of the specified field. When used without the -n option, +n ignores the first <i>n</i> characters of each line. Note: in some Unix flavors including <a href="#">Solaris</a> this option can be specified as "-s <i>n</i> ". |

Note: The -c, -d and -u options cannot be used together.

### Examples

1. Write a single copy of each unique line in *file1* into *file2*.
2. `$ uniq file1 file2`

Be careful! If *file2* exists *uniq* will overwrite it without generating a warning.

---

The next several examples will use the file *errlog*.

```
$ cat errlog
error 11: /tmp directory not found
error 22: out of memory
error 11: /tmp directory not found
error 17: low disk space
error 11: /tmp directory not found
error 22: out of memory
error 04: connection failure
error 11: /tmp directory not found
```

---

2. The first step is to sort the *errlog* file. This can be done using the Unix [sort](#) command and saving the output in the *serrlog* file.
3. **\$ sort errlog -o serrlog**
4. **\$ cat serrlog**
5. error 04: connection failure
6. error 11: /tmp directory not found
7. error 11: /tmp directory not found
8. error 11: /tmp directory not found
9. error 11: /tmp directory not found
10. error 17: low disk space
11. error 22: out of memory
12. error 22: out of memory

Now use the *uniq* command to write a single line for each type of error that occurs and save the output in the file *uerrlog*.

```
$ uniq serrlog uerrlog
$ cat uerrlog
error 04: connection failure
error 11: /tmp directory not found
error 17: low disk space
error 22: out of memory
```

Alternately you can write the unique lines from *serrlog* to [standard output](#) by not specifying an output file.

```
$ uniq serrlog
error 04: connection failure
error 11: /tmp directory not found
error 17: low disk space
error 22: out of memory
```

If no input file is specified then *uniq* reads from [standard input](#). Use this feature to [pipe](#) the output from the *sort* command directly to the *uniq* command without saving the sorted output in a file.

```
$ sort errlog | uniq
```

13. Use the *-d* option to display only those errors that occur more than once.
14. **\$ uniq -d serrlog**
15. error 11: /tmp directory not found
16. error 22: out of memory

17. Use the `-u` option to display those errors that only occur once.
18. `$ uniq -u serrlog`
19. error 04: connection failure
20. error 17: low disk space
21. Use the `-c` option to count the number of times each error occurs in the file `errlog`.
22. `$ uniq -c serrlog`
23. 1 error 04: connection failure
24. 4 error 11: /tmp directory not found
25. 1 error 17: low disk space
26. 2 error 22: out of memory

Pipe the results of this `uniq` command to the `sort` command to list the most frequently occurring errors on top.

```
$ uniq -c serrlog | sort -n -r
4 error 11: /tmp directory not found
2 error 22: out of memory
1 error 17: low disk space
1 error 04: connection failure
```

Note that the `-n` option to `sort` orders numerically rather than alphabetically and the `-r` option puts items in reverse order (i.e. highest to lowest). The above example could also be done using the original, unsorted `errlog` file and a series of pipes.

```
$ sort errlog | uniq -c | sort -n -r
```

The next several examples will use the file `purchases`, which gives a customer's name, the date and the item purchased.

```
$ cat purchases
Jimmy James Jan 2 Unit 12
Jane Doe Jan 4 Unit 17
Jimmy James Jan 10 Unit 12
John Huber Jan 15 Unit 17
Sue Butler Jan 22 Unit 05
Jane Doe Jan 30 Unit 12
Liz Tyler Feb 2 Unit 04
Jimmy James Feb 4 Unit 03
```

---

6. Generate a list of how many of each item has been sold. Step one is to sort the file `purchases` starting from fifth field, "Unit 03", "Unit 04", etc.
7. `$ sort -k 5 purchases`
8. Jimmy James Feb 4 Unit 03
9. Liz Tyler Feb 2 Unit 04
10. Sue Butler Jan 22 Unit 05
11. Jane Doe Jan 30 Unit 12
12. Jimmy James Jan 10 Unit 12
13. Jimmy James Jan 2 Unit 12
14. Jane Doe Jan 4 Unit 17
15. John Huber Jan 15 Unit 17

This output can be piped to the `uniq` command with the `-4` option to ignore the first four fields and the `-c` option to output a count of each line.

```
$ sort -k 5 purchases | uniq -4 -c
```

```
1 Jimmy James Feb 4 Unit 03
1 Liz Tyler Feb 2 Unit 04
1 Sue Butler Jan 22 Unit 05
3 Jane Doe Jan 30 Unit 12
2 Jane Doe Jan 4 Unit 17
```

The name and date data (fields one through four) on each line are no longer relevant. *Uniq* ignores the first four fields when determining duplicate lines. If two or more lines are identical starting with field five then *uniq* uses the first four fields from the first line it encounters and discards the first four fields from the subsequent lines.

16. The *cut* command can be used to remove unwanted columns from input before using *uniq*. The following example uses *cut* to discard the name and date fields before using the *sort* and *uniq* commands.

17.

```
18. $ cut -d' ' -f5,6 purchases | sort | uniq -c
```

```
19. 1 Unit 03
```

```
20. 1 Unit 04
```

```
21. 1 Unit 05
```

```
22. 3 Unit 12
```

```
23. 2 Unit 17
```

24. Use the *cut*, *sort* and *uniq* command to generate a list of customers and save the output in a file name *customer*.

```
25. $ cut -d' ' -f1,2 purchases | sort | uniq > customer
```

```
26. $ cat customer
```

```
27. Jane Doe
```

```
28. Jimmy James
```

```
29. John Huber
```

```
30. Liz Tyler
```

```
31. Sue Butler
```

The above example uses [output redirection](#) to save output in the *customer* file because *uniq* does not allow an output file to be specified unless an input file is specified.

32. Generate a list of repeat customers.

```
33. $ cut -d' ' -f 1,2 purchases | sort | uniq -d
```

```
34. Jane Doe
```

```
35. Jimmy James
```