# Linux Commands Used in the Electron Microscopy Center (EMC) – Alphabetical Order

|  |  |  |
| --- | --- | --- |
| **Command** | **Target(s) and/or Options** | **Explanation** |
| **alias** | **newCmd def** **newCmd=def** | **alias** is an extremely useful command that allows the user to define a new command (called **newCmd**) that is itself composed of commands that are understood by the shell (called **def**, for "definition"); different flavors of shell use a slightly different syntax for the **alias** command (for example, bash uses **newCmd=def** while tcsh simply puts a space between **newCmd** and **def**); for bash, if the definition contains spaces, it must be enclosed in double or single quotes (**"** or **'**)  the **alias** command is most often invoked when the user realizes that there is a complicated command (or even a series of commands) that is used frequently; for example, if you always want to see long listings of files, the bash command  $ **alias ls=" ls -l "**  effectively defines **ls** to always be **ls -l** (and when you actually want a simple **ls** command, you would then need to type **\ls**, where the backslash says to use the system's **ls** command instead of any user-specified aliases)  NOTE: instead of replacing a command with an aliased version of it (as in the example shown above), it is usually wiser to create a completely new command; for example, instead of aliasing **ls** to always mean **ls -l**, it might be more prudent to create a command such as **ll** that is aliased to mean **ls -l** instead  you can also create much more complicated **alias** commands that use pipes, command substitutions and command separators; for example,  $ **alias createdToday=' TODAY=` date | cut -c5-10 ` ; LIST=` ls -lc | grep "${TODAY}" | cut -f2 -d: | cut -f2 -d" " ` ; for file in $LIST ; do echo $file ; done; unset TODAY LIST '**  defines a new command called **createdToday** which simply lists any files that were made or modified today; a full explanation of this command can be found [here](https://iubemcenter.indiana.edu/createdToday.html) |
|  | *none* | when **alias** is used alone, a list of all defined aliases is sent to the terminal |
| **apropos** | **keyWord** | the **apropos** command searches something called the "whatis database" for the string **keyWord**; the whatis database is a collection of short descriptions of various system commands and tools, all of which have some amount of documentation in the system [manual pages](http://en.wikipedia.org/wiki/Manpage) (usually simply called "man pages"); the output of **apropos** is a list of all the man pages that contain **keyWord** in their whatis database description; the output actually shows the command, the section number of the man pages that contain the entry of interest and a 1-line description of the command; for example  $ **apropos recompress**  produces  znew (1) - recompress .Z file to .gz file  meaning that the **znew** command is found in man page section number 1 and converts from a particular form of compressed file to another form  it is possible to search for multiple key words, but keep in mind that the output is additive and not a combined search; in other words, **apropos module unload** will not show you the system commands that contain both the key words module and unload in the whatis database, but rather it will show all the system commands that contain the key word module plus any additional commands that contain the key word unload |
| **basename** | **name** | the **basename** command removes all leading directory information from **name**; for example,  $ **basename /N/u/userName/Quarry/myData.txt**  produces "MyData.txt"  this is the opposite action to the **dirname** command (see below), which prints everything except the final slash-delimited part of any string |
|  | **name suf** | when a **basename** command ends with **suf** ("suffix"), it removes both the leading directory information and the suffix **suf** (if and only if **suf** matches the end of **name**); for example,  $ **basename /N/u/userName/Mason/myData.txt .txt**  produces "myData" while  $ **basename /N/u/userName/Mason/myData.txt .sh**  produces "myData.txt" |
| **cut** | **-f n input** | the **cut** command cuts out the field(s) specified with the **-f n** option from each line of **input**, where **input** can be a single file, multiple files or standard input ([stdin](http://en.wikipedia.org/wiki/Stdin" \l "Standard_input_.28stdin.29" \t "_blank))  the value of **n** in **-f n** can be a single number, comma separated numbers, a starting and ending range of numbers (*e.g.*, 2-7) or any combination of these (*e.g.*, 1,3-5,8,9,12-15)  NOTE: the order of these numbers has no effect on output  by default, **cut** uses tabs as the field delimiter; if a line contains no tabs, the entire line is treated as "tab-delimited field 1" and either eliminated (if **-f n** does not include field 1) or retained (if it does) |
|  | **-f n -d char** | the **-d char** option changes the field delimiter from tab to **char**, where **char** must represent a single character; when using characters such as double or single quotes as the field delimiter, it will be necessary to use a backslash to "escape" the symbol (*i.e.*, to force the shell to ignore the fact that it would normally find these symbols in pairs and they would have some sort of special meaning); this command uses a blank as the field delimiter  $ **cut -f2-4 -d " " myFile.txt**  while this one uses a double quote  $ **cut -f2-4 -d \" myFile.txt** |
|  | **-c n** | another way to use **cut** is to tell the command which characters to output; for example, this command would show you the first 10 characters in each line of input  $ **cut -c 1-10 myFile.txt**  while this one would show the 5th character, and the characters in positions 20 thru 30  $ **cut -c 5,20-30 myFile.txt** |
| **date** | *none* | the **date** command without an option sends the current date and time to the terminal; the default format is  Day-of-week Month Day Hr:Min:Sec Timezone Year  *e.g.* Thu May 16 19:47:27 EDT 2013  but there are lots of ways to display this information |
|  | **+%m%d%y** **+%H%M%S** | the **+** (plus sign) is the start of an option to the **date** command that changes the output format from the default shown above; the detailed format for the output is set using a series of percent symbols (**%**'s) followed by a single upper or lower case letter; in this example, **%m** means "numeric month", **%d** means "day of the month", **%y** means "2 digit display for the year", **%H** means "hour in the 0 and 23 range", **%M** means "minutes" and **%S** means "seconds"; the following commands  $ **date +%m%d%y** $ **date +%H%M%S**  produce these two lines  051613 194727  you can string together as many of these %(letter) format designators as you would like, and so  $ **date +%m%d%y%H%M%S**  produces this line  051613194727  since this output is hard to read, you can insert your own "separators" between the format designations; valid separators can be symbols such as -, +, /, or even " " (a space); for example  $ **date +%m-%d** $ **date +%m/%d** $ **date +%m" "%d** (or **+"%m %d"**)  $ **date +%H:%M:%S**  produce these lines  05-16 05/16 05 16  19:47:27  there are a huge number of format designators, all of which can be seen by typing **date --help**; just to list a few, **%Y** generates the 4 digit year, **%A** is the day of the week, **%a** is the 3 letter abbreviation for day of the week, **%B** is the name of the month, **%b** is the 3 letter abbreviation for the month, **%Z** indicates the alphabetic timezone abbreviation and **%T** is a short-hand equivalent of **%H:%M:%S**; there are even ways to insert formatting commands (*e.g.*, a tab or a new line) into the output; to tie everything together, one explicit way to create the default format shown above would be to use this option  **+"%a %b %d %H%M%S %Z %Y"** |
|  | **+%s** **+%s.%N** **+%N** | when **+%s** is used to specify date and time, the current date and time is given as the number of seconds since 1970-01-01 00:00:00 [UTC](http://en.wikipedia.org/wiki/Coordinated_Universal_Time) (the [start of the UNIX epoch](http://en.wikipedia.org/wiki/UNIX_epoch), also known as [POSIX](http://en.wikipedia.org/wiki/Posix) time)  **+%s.%N** includes the number of nanoseconds since 1970-01-10 00:00:00 UTC whereas **+%N** simply shows the number of nanoseconds within the current second |
|  | **-d @s** | when **s** is in seconds since 1970-01-01 00:00:00 UTC (see entry above), this command converts that value into the more usual date and time display |
| **diff** | **file1 file2** | displays the differences between **file1** and **file2**; the many, many options to **diff** affect the way output is displayed and details like whether blank lines or any differences in spaces in a text file are considered to be "different" |
|  | **-r dir1 dir2** | the **-r** option causes **diff** to show recursive differences between directories **dir1** and **dir2**; the output shows not only the differences between files with the same name in the two directories, but it also lists files that only occur in one of them |
| **dirname** | **name** | the **dirname** command removes the final slash-delimited component from **name**; for example,  $ **dirname /N/u/userName/Quarry/myData.txt**  produces "/N/u/userName/Quarry"  in the case where **name** contains the absolute path for any file, **dirname** produces the directory location (the pathname) for the file  this is the opposite action to the **basename** command (see above), which produces only the final slash-delimited part of any string |
| **echo** | **string** | the **echo** command is a simple way to display a line of text (called **string** here); **string** can be enclosed within double or single quotes, or left unquoted; if quotes are used, remember that double and single quotes behave differently with the **$**, **`** (back quote) and **\** characters |
|  | **-e string** | the **-e** option is a way to tell **echo** to interpret a set of "backslash escapes" properly; for example, **\n** causes a new line to be inserted, **\t** inserts a tab stop, **\a** inserts an alert (*e.g.*, rings a bell with some types of terminals), *etc*.; a list of the recognized backslash escapes can be obtained by typing **man echo** |
| [**find**](http://en.wikipedia.org/wiki/Find) | **start target** | **find** is an extremely complicated command that in its most fundamental use locates (finds) files on the computer (and remember that everything is a file, so **find** can locate anything that exists on the system); the syntax for **find** is complex, and only a few simple examples will be shown here  **find** requires a directory to start looking in (here called **start**) and something to look for (called the **target**)  in the this example, **find** begins looking "here" (**./**) and locates any DigitalMicrograph files (**\*.dm3**) that live "here" (or the in directories below "here")  $ **find ./ -name "\*.dm3"**  the output is the name (including the relative path) of all the .dm3 files **find** can locate; the double quotes used in this example are important (and single quotes could also be used here); if no quotes are used, **find** only reports the files it locates "here" (*i.e.*, without descending into directories below here)  you could search the entire filesystem for txt files by starting the search at **/** (but keep in mind that places where you do not have [permission](http://en.wikipedia.org/wiki/File_permissions) to read files can not be searched)  $ **find / -name \*.txt**  as can be seen from this example, when **find** does not start "here", the use of quotes is optional  **find** has numerous options, and can (for example) be used to locate files created after date X and before date Y; you can use **find** to locate all files that are a particular [file type](http://en.wikipedia.org/wiki/Unix_file_types) (regular files, directories, symbolic links, *etc.*); you can even tell **find** to execute some sort of command on the files it locates (for example, locate empty directories and then delete them) |
| [**grep**](http://en.wikipedia.org/wiki/Grep) | **pattern input** | **grep** (and related commands like [egrep & fgrep](http://en.wikipedia.org/wiki/Egrep" \l "Variations" \t "_blank)) are pattern matching tools; the odd name comes from a command in the early Unix editor named [ed](http://en.wikipedia.org/wiki/Ed_%28text_editor%29" \t "_blank) that deals with global patterns, regular expressions and printing  the action of **grep** is to search **input** for instances of **pattern**; **input** can be a single file, multiple files or standard input ([stdin](http://en.wikipedia.org/wiki/Stdin" \l "Standard_input_.28stdin.29" \t "_blank)), **pattern** is any sort of [regular expression](http://en.wikipedia.org/wiki/Regular_expressions) and the output when none of the many options are invoked is the entire line from **input** that contains **pattern**  $ **grep cryoem .modules**  produces the single line  module load cryoem  while  $ **grep / .modules**  produces multiple lines  module load gcc/4.7.2 moab/7.1.1 module load perl/5.16.2 python/2.7.3  **grep** has a huge number of options, and only a few will be mentioned below; also bear in mind that details of the outputs described below may change depending on whether **input** is a single file, stdin or multiple files |
|  | **-i** | the **-i** option causes **grep** to ignore upper/lower case distinctions in **pattern** (and the long word equivalent is **--ignore-case**); for example, if you were looking for all the uses of x-ray in a text, and wanted to be certain not to miss any instances of X-ray (either at the beginning of a sentence or any inconsistency in whether the author used x-ray or X-ray), the command is  $ **grep -i x-ray textFile**  and you could use **X-RAY** (or **x-rAY** or **X-RaY** or...) in place of **x-ray**... |
|  | **-v** | as powerful as it can be to find matches to **pattern** in **input**, there are also times when what you really want is to find all lines that do not match **pattern**; the **-v** option does this, effectively ignoring lines that contain **pattern**; the other way to invoke this action (**--invert-match**) explains why this option uses a "v" |
|  | **-c** **--count** | the **-c** option (from "count") returns only the number of matched lines in **input** (and note that this is not the number of times **pattern** occurs, but rather simply the number of lines that contain one or more instances of **pattern**)  continuing from the example using **--ignore-case**, if you wanted simply to count the lines containing x-ray, X-ray or any possible permutations of lower and upper case versions of x-ray, use these commands  $ **grep -c x-ray textFile** $ **grep -c X-ray textFile** $ **grep -ci x-ray textFile**  the initial thought is that the last command would give a number that is the sum of the first two, but that is incorrect for several reasons; since occurances of x-ray and X-ray can be on the same line, the sum of the first two commands (which count lines containing x-ray and X-ray, respectively) could be less than the count which ignores the case difference; in addition, since the **-i** option looks for not only x-ray and X-ray but also X-Ray, x-Ray, x-RAY, *etc.*, this count could actually be larger than the sum of the lines containing either x-ray or X-ray |
|  | **-n** | the **-n** option causes **grep** to prefix the line number to the output line where **pattern** occurs in **input** (and the other way to invoke this effect is **--line-number**); this can be helpful when you are searching a large file and want to know how far into it the matching **pattern** occurs |
|  | **-l** | the **-l** option is only used when **input** to **grep** is more than a single file (and is not stdin); this option causes **grep** to list individual files that contain **pattern** anywhere within them; the other form of this option is **--files-with-matches** which is clearly more descriptive if not harder to type |
| **head** | **file** | **head** is related to "header" or "heading" and sends the first 10 lines of **file** to the terminal window; different systems may show a different number of lines by default |
|  | **-i file** **-n i file** **--lines=i file** **--lines=-i file** **-n -i file** | show the first **i** lines of **file** in the terminal window; the last two options (**--lines=-i**and**-n -i file**) means "do not show the last **i** lines of **file**, but show every other line" |
| **info** | *none* | when **info** is started without options, it displays some very minimal help about using the command and then lists all the topics for which there is documentation; use the [spacebar](http://en.wikipedia.org/wiki/Spacebar) to scroll through the topics and **q** (for "quit") to exit **info**  typing **h** accesses a help session from anywhere inside **info**; this opens a help session in the lower half of the terminal which shows various ways to navigate through the many pages of documentation; for example, the up/down [arrow keys](http://en.wikipedia.org/wiki/Arrow_keys), the [spacebar](http://en.wikipedia.org/wiki/Spacebar) and other keys such as [Home](http://en.wikipedia.org/wiki/Home_key), [End](http://en.wikipedia.org/wiki/End_key), [Tab](http://en.wikipedia.org/wiki/Tab_key) and [Del](http://en.wikipedia.org/wiki/Del_key) all allow you to go back and forth through the **info** pages; keep in mind that there are many pages of this small help session, and that typing **x** will always exit you from help  an even more extensive help tutorial can be accessed by tying **H** (and **x** will also get you out of this session without leaving **info**) |
|  | **topic** | when started this way,**info** displays documentation specifically about **topic** ; the information accessed using **info** is similar to that found using **man**, but **info** tends to be organized differently, to be somewhat easier to read and to contain examples for many of the topics; in some instances, if **info** contains more documentation than the corresponding [man page](http://en.wikipedia.org/wiki/Man_page), the man page will direct you to the appropriate **info** topic  even when **info** is started with a specific **topic**, you can go to the starting point seen when no options are used simply by typing **d** (which comes from "directory node") |
| **kill** | **[-s sig] pid** | the **kill** command is used to terminate the process associated with [process identifier](http://en.wikipedia.org/wiki/Process_identifier) **pid** (the number used by the operating system to uniquely identify every process); the optional use of **-s sig** (for "signal") allows the user to specify which signal is actually sent to terminate the process; there are over 60 different signals and a list can be obtained by typing **kill -l**  the most common use of **kill** is simply to stop the process in question using the safest signal (number 15, also known as SIGTERM); in a situation where **kill pid** does not stop process **pid**, it may be necessary to use **kill -s 9 pid** (or **kill -9 pid** or **kill -KILL pid**), which sends the SIGKILL signal; this will always stop any process you have permission to stop, but keep in mind that some processes may take a bit of time to stop... |
| **man** | **cmd** | **man** displays the [manual page](http://en.wikipedia.org/wiki/Manpage) (usually referred to simply as the man page) for command **cmd;** the man page documentation describes the command, all it's options and ends with information about the author; it also often lists other, related commands; if there are files associated with a specific command, these will usually be listed near the end of the man page; this documentation can be shockingly brief or extremely extensive; as a general rule, man pages do not give examples of command use (while the **info** command does); it can take considerable practice to read and understand man pages! |
|  | **section cmd** | there are sometimes multiple [man pages](http://en.wikipedia.org/wiki/Manpage) for a single command; the man pages are actually organized into different sections, and the user can explicitly tell the **man** command which section to use by specifying the section number (if it is known); the manual sections are often highly customized on different computer systems, but the following sections are used in most cases   1. User Commands 2. System Calls 3. C Library Functions 4. Devices and Special Files 5. File Formats and Conventions 6. Games 7. Miscellaneous 8. System Administration and Deamons   as an example, there are two entries for the **listen** command on Karst; the first is for a user command described in the section 1 man pages while the second is a command that is described in the [POSIX](http://en.wikipedia.org/wiki/Posix) sub-area of section 3 (refered to as 3p); these individual man pages can be seen by typing **man 1 listen** and **man 3p listen**, respectively |
| **nice** | **-n prio cmd** | if a normal command (**cmd**, which in this context is understood to also include all options and targets) is started as the target of a **nice** command, the user is setting the [scheduling priority](http://en.wikipedia.org/wiki/Scheduling_priority) of the command; this priority dictates the importance of a process (and is used by the system to decide how much of the total computer resources any process receives); the **nice** command allows the user to tell the system that a command is higher (or lower) priority than the default priority; the name of the **nice** command essentially comes from the dictate to play nicely with others...  the obligatory **-n prio** option to **nice** is the part of this command that sets the priority; values for **prio** can range from -20 (the highest priority) to +19 (the lowest)  there are limits on how high a normal user can set any process: the system itself must perform certain tasks in order for it to function, and limits on how high a user can set processes ensure that system processes always have the highest priority (and can perform the tasks necessary for the system to continue to function) |
| [**ps**](http://en.wikipedia.org/wiki/Ps_(Unix)) | *none* | **ps** comes from "process status" and the command reports a snapshot of the processes running on your computer; **ps** can be invoked without any options, but that does not produce particularly useful output.  **ps** is one of the very old Unix commands that both accepts a large number of options and that accepts options in a variety of formats (*e.g.*, some single letter options cannot be invoked with a "-", some must be and there are also "word options" that need a "--"); you will need to look elsewhere for a more detailed description of what can be done using **ps** |
|  | **-u userName** | when **ps** is invoked with **-u userName**, the results show all the active processes that are owned by the user called **userName**; here is an example of some output  PID: 3065 TTY: ? TIME: 00:00:00 CMD: sshd  PID: 3066 TTY: pts/17 TIME: 00:00:01 CMD: bash  PID: 5032 TTY: pts/39 TIME: 00:00:00 CMD: ps  where PID is the [process identifier](http://en.wikipedia.org/wiki/Process_identifier) (a number used by the operating system to uniquely identify every process), TTY is the identification of the "terminal" associated with the process, TIME is the amount of computer time used by the process and CMD is the command associated with the process; from this description, it should be clear that every command you run in linux is a unique process  a common use of **ps** would be to pipe the output into a **grep** command that would then find the PID for a specific command; for example  $ **ps -u userName | grep python**  will show the instance(s) of python being run by user **userName**; this command could be used to find the PID of any python processes that are running, and the PID(s) could be terminated using a **kill** command |
|  | **-f** | another useful option to **ps** is **-f**, which causes the command to output the entire ("full") command that was originally used; some of the output for the command **ps -u userName -f** is  UID: userName PID: 3065 TTY: ? TIME: 00:00:00 CMD: sshd: userName@pts/1  UID: userName PID: 3066 TTY: pts/17 TIME: 00:00:01 CMD: -bash  UID: userName PID: 5032 TTY: pts/39 TIME: 00:00:00 CMD: ps –u username -f  where the new column UID is the name used by the operating system to uniquely identify user userName; in this example, the major change is that instead of simply listing PID 5032 as the command **ps**, the output contains the full command that generated the output |
| **renice** | **newPrio pid** | **renice** is related to the command **nice** (which sets the [scheduling priority](http://en.wikipedia.org/wiki/Scheduling_priority) of a command when that process is started); in the case of **renice** (where the name comes effectively from "redo nice"), the priority of the running process associated with [process identifier](http://en.wikipedia.org/wiki/Process_identifier) **pid** is set to level **newPrio**  values for **newPrio** range from 0 (the highest priority level a normal user can make) to 20 (the lowest); a user must own a process in order to adjust its priority, and a user can only lower the priority of a running process; in other words, a user can adjust a job running with priority 10 to 11 or higher (lower priority) but not to 9 or less (higher priority); this is true even if that user had originally renice'd the priority low and now wants to return it to the starting priority |
| **script** | *none* | the **script** command is a way to save everything printed in a terminal window to a text file; this includes your typing and anything that the computer sends to you; when **script** is invoked without any options, the created file is called "typescript" (and you can think about the command name coming from this typescript filename, or vice versa); saving information to the typescript file is terminated by typing a [control-D](http://en.wikipedia.org/wiki/Control-D), **exit** or **logout** and may depend on which shell you are running |
|  | **name** | when the option **name** is used, saved information is stored in a new file called **name**; if **name** already exists, it will be over-written (and lost); this is also true when information is saved to the typescript file: an existing typescript file will be over-written |
|  | **-a name** | when the option **-a name** is used, saved information is appended into an existing file called **name** (and if file **name** does not exist, it will be created) |
| [**sed**](http://en.wikipedia.org/wiki/Sed) | **script input** | **sed** is an extremely useful "stream editor" that filters and transforms text, and is such a complex command that books have been written about it; a general over-view of **sed** would be to say that it reads the **input** line by line and performs actions that are described by something called a "sed **script"**  **input** can be single or multiple files or standard input ([stdin](http://en.wikipedia.org/wiki/Stdin" \l "Standard_input_.28stdin.29" \t "_blank)); the actions described by the sed **script** can be quite simple or extremely complicated, and **script** can be a separate file or simply a set of actions described on the command line  here are some extremely simple examples of sed; the following command finds every occurance of "**X-ray**" in file **myFile.txt** and replaces it with **x-ray**  $ **sed -e "s/X-ray/x-ray/g" myFile.txt**  the **-e** option to **sed** indicates that the **script** to use is found on the command line (and comes from "execute the following" or something similar); the **script** itself is the phrase within the double quotes, **s/X-ray/x-ray/g**, where the **/**'s break the **script** into four parts: **s** (the first part) means "substitute", **X-ray** (the second part) is the target of the substitution, **x-ray** (the third part) is what gets substituted for the second part and **g** (the fourth part) means "do this whenever possible" (*i.e.*, "globally" - replace every instance of **X-ray** with **x-ray**)  a shorthand way to represent this sort of substitution script is **s/pat1/pat2/g** where **pat1** is the pattern to find and **pat2** is the replacement; changing the **g** in this **script** to a number N causes **sed** to replace only the Nth occurance of **pat1** with **pat2**; the following example changes only the 3rd instance of **X-ray** (and does nothing if there are only two or fewer matches to **X-ray**)  $ **sed -e "s/X-ray/x-ray/3" myFile.txt**  a designation like **Ng** leaves the first (N-1) instances of **pat1** unchanged but changes all the remaining ones to **pat2** (where the g again comes from global)  a common use for a **sed** substitution **script** would be to change the pathname for a series of files from one location to a different location; since the pathnames will likely contain **/**'s, it is easy to see how **sed** might become confused when given a **script** that has more than four parts delimited with **/**'s; fortunately, **sed** can use any other character as the delimiter  $ **sed -e "s]X-ray]x-ray]9g" myFile.txt**  which changes every instance of **X-ray** to **x-ray**, but starting with the 9 occurance  another useful trick is to find occurance of **pat1** and replace them with an augmented **pat1** (*i.e.*, **pat2** is really just **pat1** with some things added to it); in such a case, the following shorthand notation is useful  $ **sed -e "s/X-ray/& (or electron)/g" myFile.txt**  this command replaces every occurance of **X-ray** with the phrase **X-ray (or electron)**  there are ways to use **sed** that manipulate a specific line or lines and ranges of lines, ways that look for a pattern but then replaces a second pattern with a third only on lines that contain the first pattern, ways to add and/or delete lines based on pattern matching and a host of other ways to manipulate text  the key to **sed** is that it uses the power of regular expressions to find patterns and manipulate them |
| **sort** | **file** | the **sort** command sorts lines of a text file; the default behaviour of **sort** will depend on your linux system and your [locale setting](http://en.wikipedia.org/wiki/Locale#General_locale_settings), but in general, the output order will show punctuation followed by numbers followed by letters (and possibly sorted into capital letters first)  because **sort** can work with [stdin](http://en.wikipedia.org/wiki/Stdin" \l "Standard_input_.28stdin.29" \t "_blank), it is an extremely powerful tool when other commands are **piped** into it |
|  | **-n** **--numeric-sort** | the **-n** option causes **sort** to use numeric value (and not to sort numbers based on alphanumeric values!); this may seem like a rather trivial distinction, but when sorting without the **-n** option, 111 comes before 12 instead of after; sometimes that may be what you want, but sometimes it isn't! |
|  | **-r** **--reverse** | the **-r** option causes **sort** to reverse the order that it would normally output; |
|  | **-k field** **--key field**  (often used with **-t sep)** | the **-k field** (**--key field**) option causes **sort** to operate on the kth **field** in each input line; unless the **-t** option is used, fields are separated by a transition from non-blank to blank regions (*i.e.*, any number of spaces acts like a single separator);  the **-t sep** (or **--field-separator sep**) option is only meaningful when used with a **-k field** option; it allows the user to specify what character (**sep**) to use as the separator between fields  one example of this might be if you wanted to sort the type of [comma-separated output](http://en.wikipedia.org/wiki/Comma-separated_values) (abbreviated as csv output) produced by various spreadsheet programs: since each cell in a spreadsheet might contain spaces, it would not be useful to break this output into fields based on (the default) non-blank to blank transition; instead, you could specify (for example) **-k 6 -t ,** and sort based on the content of the 6th cell in the original spreadsheet |
| **tail** | **file** | show the last 10 lines of **file** in the terminal window (and so in a sense, **tail** is an opposite of the **head** command, and the name of this command is either related to this head/tail dicotomy or from something like "show the tailend of **file**"); different systems may show a different number of lines by default |
|  | **-i file** **-n i file** **--lines=i file** **--lines=+i**file **-n +i file** | show the last **i** lines of **file** in the terminal window; the last two options (**--lines=+i file** and **-n +i file**) causes the lines starting at line number **i** to be sent to the terminal window |
| [**tar**](http://en.wikipedia.org/wiki/Tar_%28computing%29) | *none* | **tar** comes from "tape archive" and is both a file format (a "tar file" also referred to as a "tarball") and this program that handles them; when **tar** is started without any options, it displays a specific [error message](http://en.wikipedia.org/wiki/Error_message) with some suggestions and exits; the **--help** and the **--usage** options both show the many options that **tar** handles, but the **--usage** option is really nothing more than a list of options without any sort of explanation  the most common use of the **tar** command is to make [archival](http://en.wikipedia.org/wiki/Archive_file) tar files and/or to extract either the entire contents or specific files from them; tarballs can be created using any of several [lossless file compression](http://en.wikipedia.org/wiki/Lossless_data_compression) algorithms (and then must be uncompressed (inflated) in order to read them); such compressed archives are often used to transfer both programs and data across the [Internet](http://en.wikipedia.org/wiki/Internet), and just about every computer has a way to inflate a compressed archive  Windows machines usually have built in compression and decompression tools; in addition, programs such as WinZip and 7-Zip can be installed; on Macs, **tar** should already be installed and is accessible in the terminal window; in addition, Stuffit Expander should also work on most tarballs  extensive help can be obtained for **tar** using **man tar**, **info tar** and various on-line resources |
| **tee** | **name** | the **tee** command accepts standard input ([stdin](http://en.wikipedia.org/wiki/Stdin" \l "Standard_input_.28stdin.29" \t "_blank)) and outputs it to both standard output ([stdout](http://en.wikipedia.org/wiki/Stdout" \l "Standard_output_.28stdout.29" \t "_blank)) and to a file called **name**; if this file already exists, it will be over-written and lost |
|  | **-a name** | the **-a name** option allows **tee** to append its output to a file called **name** instead of over-writing it; as with other commands that append, if file **name** does not already exist, it will be created |
| [**top**](http://en.wikipedia.org/wiki/Top_(Unix)) | *none* | **top** is an example of the class of [task managers](http://en.wikipedia.org/wiki/Task_manager) that are found on various linux machines; **top** produces information about processes running on the system (and so is related to the **ps** command); while **ps** only generates a snapshot of that information, the output from **top** is continuously updated  there are many options to **top**, and users are advised to look at the documentation; for example, default output is sorted by the instantaneous amount of CPU used (and the name of the command comes from this display of the top CPU using processes), but the user can sort based on properties such as memory usage or the total amount of CPU a process has used; the information displayed using **top** can be modified to color code different outputs and the output itself can be totally reformatted  once **top** is running, an extensive set of "interactive" commands can be used to adjust the display and to manipulate the different processes; a common use of this interactive aspect of **top** is to find and then kill a specific process (similar to what could be done with a **ps** command to find the [process identifier](http://en.wikipedia.org/wiki/Process_identifier) and then a **kill** command to terminate it)  another common interactive use of **top** is to adjust the [scheduling priority](http://en.wikipedia.org/wiki/Scheduling_priority) of a process; you can think of this priority as a measure of how important a job is (and thus what percentage of computer resources should be devoted to it), and in certain situations, a user might want to lower the priority so that a process does not soak up all the available resources; this is comparable to what the **renice** command does, but again, the user would need to find the process identifier using a **ps** command and then **renice** to lower the priority  **top** without the **-n** option is a command that runs until the user stops it; the normal termination signal is to type a "q" (for "quit") but [control-C](http://en.wikipedia.org/wiki/Control-C) also works |
|  | **-p pid [,pid2]** | the **-p pid** option tells **top** to monitor only the process associated with process identifier **pid** (and you can list multiple processes using either **-p pid1 -p pid2 -p pid3** *etc.* or **-p pid1, pid2, pid3** *etc.*) |
|  | **-n number** | the **-n number** option tells **top** the number of times to update its output before exiting |
|  | **-u userName** | the **-u userName** option tells **top** to generate output only for processes owned by the user **userName** |
| **tr** | **set1 set2** | **tr** comes from "translate" (or "translitorate") and is a way to translate (or map) one specific character (or a set of characters, both referred to as **set1** here) to another character (or a second set of characters, called **set2** here); when using character sets, the number of characters in each set must be equal; each occurance of a character in **set1** is replaced by the corresponding character in **set2**  a simple example may help clarify how this command works  $ **echo "Hello world" | tr eo 30**  produces  H3ll0 w0rld  where each **e** has been replaced with a **3** and each **o** has been replaced with a **0** (zero)  as noted above, the number of characters in **set1** must equal the number in **set2**, but there is no limit on the actual number of characters in each; it is possible to specify ranges of characters in a set (so 0-9 means all the numbers, and a-z means all the lower case letters); **tr** also understands some "shorthand" sets such as [:lower:] (all the lower case letters), [:upper:] (all the upper case letters), [:alpha:] (all letters), [:punct:] (all punctuation) and [:digit:] (all the numerals)  **tr** only operates on the standard input ([stdin](http://en.wikipedia.org/wiki/Stdin" \l "Standard_input_.28stdin.29" \t "_blank)) and if you want (for example) to use **tr** to turn all the text in a file to upper case, you will need to **pipe** the contents of the file into **tr** or to use redirection operators  $ **cat file | tr [:lower] [:upper]** $ **tr [:lower] [:upper] < file**  in this example using a file, you will probably want to capture the output in a second file (called **file2** here) using redirection (**> file2**) or by **piping** it into (say) a **tee** command ( **| tee file2**) |
|  | **-d set1** | the **-d**option tells **tr** to "delete" the character(s) in **set1**; you can think about this as translating each character in **set1** to nothing; always bear in mind that you are not replacing characters with the blank character (" "), but that the characters are being eliminated  $ **echo "Hello world" | tr -d eo**  produces  Hll wrld  if you really wanted to replace characters with a space, you would need to use  $ **echo "Hello world" | tr eo "  "** $ **echo "Hello world" | tr eo '  '**  which both produce  H ll  w rld  remember that there must be two spaces enclosed in the quotes in the commands, and that there are two spaces between the last l from Hello and the w from world in the output |
|  | **-s set1** | the **-s**option tells **tr** to "squeeze" the character(s) in **set1**; the equivalent of **-s** is **--squeeze-repeats**, which is a better description of what the command does: it replaces every instance of repeated characters listed in **set1** with a single instance  $ **echo Hello world | tr -s l** $ **echo 22 Bookkeeper Rd | tr -s [:alnum]**  produces  Helo world 2 Bokeper Rd  where in the second example, the [:alnum] shorthand indicates all the letters and digits; the (in this case) un-necessary double quotes were also dropped in both examples |
| **uniq** | **input** | **uniq** comes from "unique" and is a way to report the lines in **input** that are not exact repeats (*i.e.*, that are unique to **input**); the many options to **uniq** allow the user to skip the first N fields or characters, to ignore upper and lower case differences, *etc.*  **uniq** only detects repeated lines that are adjacent to each other in **input**; in other words, if two (or more) identical lines are separated by non-identical lines, **uniq** will not treat the identical lines as "the same"; for this reason, input to **uniq** is often **piped** from a **sort** command  $ **sort myFile | uniq** |
|  | **-d input** | the **-d** (or **--repeated**) option to **uniq** causes the output to contain only the duplicated (or repeated) lines; this is the inverse of the normal behavior for **uniq** |
|  | **-c input** | the **-c** option comes from "count" (and another way to invoke this is **--count**); this option causes **uniq** to prefix each unique line with the number of times that line occurs  if you wanted the output to be listed in order of the number of occurences, you could **pipe** this into a numeric **sort** command  $ **sort myFile | uniq -c | sort -n** |
| **vi** | *none* | this command starts a file editor called [vi](http://en.wikipedia.org/wiki/Vi) (and the name comes from a command in another text editor called [ex](http://en.wikipedia.org/wiki/Ex_%28text_editor%29)); **vi** is a non-graphical text editor that is extremely non-intuitive to use; however, every linux machine will have **vi** installed and you may find it useful to learn some simple editing commands; the most useful editing commands to learn are how to exit (type ":q<Enter>") and how to get help (type ":h<Enter>") |
|  | **file(s)** | when the **vi** command includes one or multiple files, the editor starts with the file(s) open (ready to be edited); when multiple files are open, you will need to switch between them; the command to exit from all the files is ":qa!<Enter>" |
| **wc** | **input** | **wc** comes from "word count" and is a command that shows information about the length of **input** (which can be either a file or the [stdin](http://en.wikipedia.org/wiki/Stdin" \l "Standard_input_.28stdin.29" \t "_blank) input stream)  with no options other than **input**, the output shows the number of lines, the number of words and the number of bytes for the input  $ **wc file**  #lines #words #bytes **file**  for example  $ **wc .modules**  17 165 949 **.modules**  **wc** is often used with the **pipe** command to show the various counts for the output from a command; for example, the following command shows the number of DigitalMicrograph files in the current area  $ **ls \*.dm3 | wc**  the resulting output (*e.g.*, 50 50 857) shows that the number of lines is equal to the number of words (and both are equal to the number of files); you can also determine that the file names contain slightly more than 17 characters on average (857/50) |
|  | **-l input** **-w input** **-c input** **-m input** | the options shown here limit the output to the number of lines (**-l**), words (**-w**), bytes (**-c**) or characters (**-m**) |
| **whoami** | *none* | prints the user name; this is extremely useful when writing [shell scripts](http://en.wikipedia.org/wiki/Shell_scripts), but can come in handy at other times also |
| **whereis** | **name** | **whereis** locates a file called **name** on the computer system when it has been stored in some of the more standard linux places; **whereis** is designed to locate system-wide commands, source files, executable files (also called binary files or simply binaries) and manual pages  **whereis** removes common file extensions as it looks for these files; for example, "**whereis ls**", "**whereis ls.1**" and "**whereis ls.1p**" all show the command and two different man pages for the command **ls**  /bin/ls /usr/share/man/man1/ls.1.gz /usr/share/man/man1p/ls.1p.gz |
| **which** | **name** | **which** is another command that searches for a file called **name**, except that the search is done using the current **PATH** variable; as soon as **which** finds **name**, the search is stopped (and so it is impossible for **which** to produce 2 locations for the same value of **name**); you can think of this as meaning that **which** finds the first (and only the first) instance of **name** along your path  more importantly, **which name** shows you the location of the instance of **name** that is executed when you simply type **name** (and in cases where you may have multiple commands or programs called **name**, this shows you which one of them occurs first on your path and is actually used); always keep in mind that having multiple instances of **name** in your path can lead to both confusion and mistakes |