Linux Basics HOWTOs

From Matchi Wiki

Contents

Development Packages to Install on your Desktop

Note that the command to install a package varies between Linux distributions.

MadiaDB

This is similar to MySQL, but better. All the servers will eventually be running MariaDB, so start using it now already. It is already installed on Sabayon Linux and has command-line utilities that start with 'mysql_...'.

Resources: https://mariadb.com

First-time initialization

This creates the initial system database and is where you set the default password:

```
$ sudo equo config dev-db/mariadb
!    #  (1/1) dev-db/mariadb-10.1.10 | installed from: sabayon-weekly
!  ::: >>>  (1/1) dev-db/mariadb-10.1.10
!    ## Configuring package: dev-db/mariadb-10.1.10
!    ## SPM: configuration phase
* Please provide a password for the mysql 'root' user now
* or through the /root/.my.cnf file.
* Avoid [\_%] characters in the password
   >
* Retype the password
   >
* Creating the mysql database and setting proper permissions on it ...
* Command: '/usr/share/mysql/scripts/mysql_install_db' '--basedir=/usr'  --loose-skip-grant-tables --loose-skip-host-cache --loose-skip-name-resolve --loose-skip-networking --loo
* Starting mysqld ...
* Command //usr/sbin/mysqld            --loose-skip-grant-tables --loose-skip-host-cache --loose-skip-name-resolve --loose-skip-networking --loose-skip-slave-start --loose-skip-
*             --tmpdir=///tmp/
```

```
2016-01-25 17:18:15 139910085806080 [Note] //usr/sbin/mysqld (mysqld 10.1.10-MariaDB) starting as process 23816 ...   [ ok ]
 * Setting root password ...                                                                                        [ ok ]
 * Loading "zoneinfo", this step may require a few seconds ...                                                      [ ok ]
 * Stopping the server ...
 * Done
 * @@ No configuration files to update.
```

Start up the MariaDB Service

```
sudo systemctl start mariadb
```

Make MariaDBstart up on Boot-up

```
$ sudo systemctl enable mariadb
Created symlink from /etc/systemd/system/mysql.service to /usr/lib64/systemd/system/mariadb.service.
Created symlink from /etc/systemd/system/mysqld.service to /usr/lib64/systemd/system/mariadb.service.
Created symlink from /etc/systemd/system/multi-user.target.wants/mariadb.service to /usr/lib64/systemd/system/mariadb.service.
```

Command Line use of Linux

Some command-line techniques you have to know before setting out on a Linux machine. Practice on a development instance or a Raspberry PI first, or install Linux as your de-facto O/S on your laptop/PC before going at it hammer and tongs on a production machine! Since Matchi servers run the RedHat clone CentOS, install Fedora your laptop for the best approximation of the RedHat server-side environment. Linux Mint and Sabayon Linux are even nicer to use. All of these distros come with graphical desktop environments that will know the pants of any Windows desktop interface.

Help Bailout

To get more information on a command, type `man [command name]`. If the package for this command is not yet installed, then no help on the command would be available. Plenty of information on Google, BTW.

Overview of stuff you should get to know

- Learn BASH. Actually, read the whole bash man page; it's pretty easy to follow and not that long. Type: `man bash`.
- Learn `vi`. There's really no competition for random Linux editing (even if you use Emacs or Nano most of the time). If you are going to spend much time at any Linux command prompt then it's likely that you're involved in administering or operating lots of Linux machines or virtual machine instances in a cluster.
- Get to know `ssh` and the basics of passwordless authentication, via `ssh-agent`, `ssh-add`, etc.
- Be familiar with BASH job management: `Ctrl`-`Z`, `Ctrl`-`C`, `&`, `jobs`, `fg`, `bg`, `kill`, etc.
- Basic file management: `ls` and `ls -l` (learn what every column means), `less`, `head`, `tail` and `tail -f`, `ln` and `ln -s` (learn the differences and advantages of hard versus soft links), `du` (e.g. for a quick summary of disk usage: `du -sk *`), `df`, `mount`.
- Basic network management: `ip` or `ifconfig`, `dig`, `nslookup`.
- Know regular expressions well (no, seriously, you have immense power at your finger tips - know how to use it), and the various flags to `grep`.
- Learn to use `yum` (used in the RedHat/CentOS distro) to find and manage and install software packages.
- Learn some of the tricks for quicker command line navigation and command history. It will save you time, increase your accuracy and impress the bejaysus out of any GUI-guy who can only operate a computer by clicking pretty buttons on a screen.
- Understand how dot-files work
- Understand how file attributes, groups and owners work and commands like `chmod` and `chown` do.
- Learn so `sudo` - not as cool as surfing but more powerful.

Once you know this stuff well, you can be trusted on a production machine. Until then, practice on a development environment!

Command line navigation tricks

History expansion

- `!!` = previous command
- `!$` = last word of previous command
- `!-n` = nth previous command
- `!#$` = last word of current line
- `!<start of command>` will execute the command from history starting with letters after "!"

Brace Expansion:

- `{a..b}` = numbers a to b in order. Useful for indexing arrays, e.g. `for i in {3..7}; do echo $i; done` prints the values 3, 4, 5, 6 and 7 in new lines.
- `{a,b,c}` = words a, b, c. Useful for paths: `touch /tmp/{foo,bar,baz}`, which will create the directories `/tmp/foo`, `/tmp/bar` and `/tmp/baz`.

Parameter Expansion

Suppose that the variable `foo=/usr/local/blah.txt`

- `${variable#word}` = removes word from the beginning of variable. For example, `${foo#*/} = usr/local/blah.txt`
- `${variable##word}` = same thing, but removes longest pattern matching word. For example, `${foo##*/} = blah.txt`
- `${variable%word}` = removes word from end of variable. For example: `${foo%.txt} = /usr/local/blah`
- `${variable%%word}` = same thing but longest matching suffix

Process Substitution

< (command) = treats the output of command as a file. `diff -u < (ssh web{1,2} cat /etc/passwd))` shows you a unified diff between /etc/passwd on web1 and 2

More everyday use scenarios

- In bash, use Ctrl-R to search through command history.
- In bash, use Ctrl-W to kill the last word, and Ctrl-U to kill the line. See man readline for default keybindings in bash. There are a lot. For example Alt-. cycles through prevous arguments, and Alt-* expands a glob.
- To go back to the previous working directory: `cd -`
- If you are halfway through typing a command but change your mind, hit Alt-# to add a # at the beginning and enter it as a comment (or use Ctrl-A, #, Enter). You can then return to it later via command history.
- Use xargs (or parallel). It's very powerful. Note you can control how many items execute per line (-L) as well as parallelism (-P). If you're not sure if it'll do the right thing, use xargs echo first. Also, -I{} is handy.

Examples:

```
find . -name \*.py | xargs grep some_function
cat hosts | xargs -I{} ssh root@{} hostname
```

- pstree -p is a helpful display of the process tree.
- Use pgrep and pkill to find or signal processes by name (-f is helpful).
- Know the various signals you can send processes. For example, to suspend a process, use kill -STOP [pid]. For the full list, see man 7 signal
- Use nohup or disown if you want a background process to keep running forever.
- Check what processes are listening via netstat -lntp. See also lsof.
- In bash scripts, use set -x for debugging output. Use set -e to abort on errors. Consider using set -o pipefail as well, to be strict about errors (though this topic is a bit subtle). For more involved scripts, also use trap.

In bash scripts, subshells (written with parentheses) are convenient ways to group commands. A common example is to temporarily move to a different working directory, e.g.

1. do something in current dir

(cd /some/other/dir; other-command)

1. continue in original dir

In bash, note there are lots of kinds of variable expansion. Checking a variable exists: ${name:?error message}. For example, if a bash script requires a single argument, just write input_file=${1:?usage: $0 input_file}. Arithmetic expansion: i=$(((i + 1) % 5)). Sequences: {1..10}. Trimming of strings: ${var%suffix} and ${var#prefix}. For example if var=foo.pdf, then echo ${var%.pdf}.txt prints "foo.txt".

The output of a command can be treated like a file via <(some command). For example, compare local /etc/hosts with a remote one: diff /etc/hosts <(ssh somehost cat /etc/hosts)

- Know about "here documents" in bash, as in cat <<EOF ....
- In bash, redirect both standard output and standard error via: some-command >logfile 2>&1. Often, to ensure a command does not leave an open file handle to standard input, tying it to the terminal you are in, it is also good practice to add "</dev/null".

Use man ascii for a good ASCII table, with hex and decimal values. On remote ssh sessions, use screen or dtach to save your session, in case it is interrupted. In ssh, knowing how to port tunnel with -L or -D (and occasionally -R) is useful, e.g. to access web sites from a remote server. It can be useful to make a few optimizations to your ssh configuration; for example, this .ssh/config contains settings to avoid dropped connections in certain network environments, not require confirmation connecting to new hosts, forward authentication, and use compression (which is helpful with scp over low-bandwidth connections): TCPKeepAlive=yes ServerAliveInterval=15 ServerAliveCountMax=6 StrictHostKeyChecking=no Compression=yes ForwardAgent=yes To get the permissions on a file in octal form, which is useful for system configuration but not available in "ls" and easy to bungle, use something like stat -c '%A %a %n' /etc/timezone

Data processing To convert HTML to text: lynx -dump -stdin If you must handle XML, xmlstarlet is old but good. For JSON, use jq. For Amazon S3, s3cmd is convenient (albeit immature, with occasional misfeatures). Know about sort and uniq (including uniq's -u and -d options). Know about cut, paste, and join to manipulate text files. Many people use cut but forget about join. It is remarkably helpful sometimes that you can do set intersection, union, and difference of text files via sort/uniq. Suppose a and b are text files that are already uniqued. This is fast, and works on files of arbitrary size, up to many gigabytes. (Sort is not limited by memory, though you may need to use the -T option if /tmp is on a small root partition.) cat a b | sort | uniq > c # c is a union b cat a b | sort | uniq -d > c # c is a intersect b cat a b b | sort | uniq -u > c # c is set difference a - b Know that locale affects a lot of command line tools, including sorting order and performance. Most Linux installations will set LANG or other locale variables to a local setting like US English. This can make sort or other commands run many times slower. (Note that even if you use UTF-8 text, you can safely sort by ASCII order for many purposes.) To disable slow i18n routines and use traditional byte-based sort order, use export LC_ALL=C (in fact, consider putting this in your .bashrc). Know basic awk and sed for simple data munging. For example, summing all numbers in the third column of a text file: awk '{ x += $3 } END { print x }'. This is probably 3X faster and 3X shorter than equivalent Python. To replace all occurrences of a string in place, in one or more files: perl -pi.bak -e 's/old-string/new-string/g' my-files-*.txt To rename many files at once according to a pattern, use rename. (Or if you want something more general, my own tool repren may help.) rename 's/\.bak$//' *.bak Use shuf to shuffle or select random lines from a file. Know sort's options. Know how keys work (-t and -k). In particular, watch out that you need to write -k1,1 to sort by only the first field; -k1 means sort according to the whole line. Stable sort (sort -s) can be useful. For example, to sort first by field 2, then secondarily by field 1, you can use sort -k1,1 | sort -s -k2,2 If you ever need to write a tab literal in a command line in bash (e.g. for the -t argument to sort), press Ctrl-V <tab> or write $'\t' (the latter is better as you can copy/paste it). For binary files, use hd for simple hex dumps and bvi for binary editing. Also for binary files, strings (plus grep, etc.) lets you find bits of text. To convert text encodings, try iconv. Or uconv for more advanced use; it supports some advanced Unicode things. For example, this command lowercases and removes all accents (by expanding and dropping them): uconv -f utf-8 -t utf-8 -x '::Any-Lower; ::Any-NFD; [:Nonspacing Mark:] >; ::Any-NFC; ' < input.txt > output.txt To split files into pieces, see split (to split by size) and csplit (to split by a pattern).

System debugging For web debugging, curl and curl -I are handy, and/or their wget equivalents. To know disk/cpu/network status, use iostat, netstat, top (or the better htop), and (especially) dstat. Good for getting a quick idea of what's happening on a system. To know memory status, run and understand the output of free and vmstat. In particular, be aware the "cached" value is memory held by the Linux kernel as file cache, so effectively counts toward the "free" value. Java system debugging is a different kettle of fish, but a simple trick on Sun's and some other JVMs is that you can run kill -3 <pid> and a full stack trace and heap summary (including generational garbage collection details, which can be highly informative) will be dumped to stderr/logs. Use mtr as a better traceroute, to identify network issues. For looking at why a disk is full, ncdu saves time over the usual commands like "du -sk *". To find which socket or process is using bandwidth, try iftop or nethogs. The ab tool (comes with Apache) is helpful for quick-and-dirty checking of web server performance. For more complex load testing, try siege. For more serious network debugging, wireshark or tshark. Know strace and ltrace. These can be helpful if a program is failing, hanging, or crashing, and you don't know why, or if you want to get a general idea of performance. Note the profiling option (-c), and the ability to attach to a running process (-p). Know about ldd to check shared libraries etc. Know how to connect to a running process with gdb and get its stack traces. Use /proc. It's amazingly helpful sometimes when debugging

live problems. Examples: /proc/cpuinfo, /proc/xxx/cwd, /proc/xxx/exe, /proc/xxx/fd/, /proc/xxx/smaps. When debugging why something went wrong in the past, sar can be very helpful. It shows historic statistics on CPU, memory, network, etc. For deeper systems and performance analyses, look at stap (systemtap) and perf. Confirm what Linux distribution you're using (works on most distros): "lsb_release -a" Use dmesg whenever something's acting really funny (it could be hardware or driver issues).

Better way to change directory: If you are a command-line user, autojump is a must have package. You can change directory by just specifying a part of directory name (without subdirs). You can also use jumpstat to get a statistics of your directory jumps.

$ j log /var/log $ j ard /home/ab/work/arduino

Sys-admin friendly relatively unknown tools: dstat, htop, iotop ethtool, mii-tool dmidecode lsof, netstat -nt freeipmi

Run level editor: Save some boot time. rcconf: Cursed based tool for Debian, Ubuntu and clones ntsysv: Curses based tool for Red Hat and clones chkconfig: Command line tool for Red Hat and clones systemctl: Newer command line tool for Red Hat and clones update-rc.d: Command line tool for Debian, Ubuntu and clones

BASH History Tricks

- Tip 1

Make your `~/.bash_history` under git and back it up on github private repository. The commands you type in your life will never be lost.

Before you commit the `~/.bash_history`, you need unique it and clean the comands a little bi, for example, remove ones containing less than 5 characters.

So insert below line into `~/.bashrc`. Now you run command `cleanfile ~/.bash_history` to clean the history:

```
function cleanfile () {
  if [ -z "$1" ]; then
    echo "Usage: remove duplicated lines without sortdt"
    echo "  cleanfile ~/.bash_history"
  else
      local bkfile="$1.backup"
      # \+ does not work in OSX sed
      # delte short commands, delete git related commands
      sed 's/ *$//g;' $1 | sed '/^.\{1,4\}$/d' | sed '/^g[nlabcdusfp]\{1,5\} .*/d' | sed '/^git [nr] /d' | sed '/^rm /d' | sed '/^cgnb /d' | sed '/^touch /d' > $bkfile
      # @see Page on stack Overflow/questions/11532157/unix-removing-duplicate-lines-without-sorting
      cat $bkfile | awk ' !x[$0]++' > $1
      rm $bkfile
  fi
}
```

- Tip 2

This is actually *NOT TO DO* tip. I turn off the time stamp string because time stamp is a distraction on screen when search history. So DONOT insert below line into your ~/.bashrc: export HISTTIMEFORMAT="%m-%d: "

Thanks for Andrew Daviel reminding. You shouldn't follow my instruction blindly . I always works as a Linux developer. So I usually focus on how to re-use command line as quickly as possible. Anything unrelated to coding is regarded as noise and will be purged.

System Administrators may have different views. Timestamp is regarded as a critical part of system log for them.

Please read my other tip with critical thinking

- Tip 3

C-R search from the latest commands. Sometimes I have vague memory about a command used months ago and I want to reuse that command.

So just type:

```
history|grep "keyword"|grep "keyword2"
```

The ID of that command is displayed, say it's "9899".

Then just type following text to execute that command: !9899

Until now I've not revealed the tip yet!

The trick is to seldom re-use the old command without editing. So insert below line into ~/.bashrc:

```
shopt -s histverify
```

then !9899 will insert the command into shell instead of execute it.

- Tip 4

The `history` command is used so often that you should assign an alias for it in `~/.bashrc`: `alias h=history`

- Tip 5

There could be better way to search, filter the history if you use percol from mooz (Masafumi Oyamada).

You could "h keyword" to find the command and place it into system clipboard.

Steps to install percol: 1. download mooz/percol, that package you will find a directory named percol, put it in ~/bin/, there is also a python program named percol, rename it into percol.py and place it in ~/bin/ too.

2. insert below code into ~/.bashrc:

```
[ $(uname -s | grep -c CYGWIN) -eq 1 ] && OS_NAME="CYGWIN" || OS_NAME=`uname -s`

function pclip() {
    if [ $OS_NAME == CYGWIN ]; then
        putclip $@;
    elif [ $OS_NAME == Darwin ]; then
        pbcopy $@;
    else
        if [ -x /usr/bin/xsel ]; then
            xsel -ib $@;
        else
            if [ -x /usr/bin/xclip ]; then
                xclip -selection c $@;
            else
                echo "Neither xsel or xclip is installed!"
            fi
        fi
    fi
}

function h () {
    # reverse history, pick up one line, remove new line characters and put it into clipboard
    if [ -z "$1" ]; then
        history | sed '1!G;h;$!d' | ~/bin/percol.py | sed -n 's/^ *[0-9][0-9]* *\(.*\)$/\1/p'| tr -d '\n' | pclip
    else
        history | grep "$1" | sed '1!G;h;$!d' | ~/bin/percol.py | sed -n 's/^ *[0-9][0-9]* *\(.*\)$/\1/p'| tr -d '\n' | pclip
    fi
}
```

- Tip 6

Place below setup in ~/.bashrc, which will insert typed command into ~/.bash_history immediately after you execution of the command: 1 PROMPT_COMMAND="history -a" # update histfile after every command

See stackoverflow.comPage on stackoverflow.com, raychi's answer.

- Tip 7

percol.py gives you a second chance to filter the history.

For example, bash cli "xrandr -s 1024x768" is used to switch to SVGA resolution.

But you can also input "xrandr -s 1024x768 # switch resolution vga", the keywords after "#" will not be executed, but it will be recorded into ~/.bash_history, so when you search bash history, "switch" and "vga" could be regarded as a TAG. since perlcol.py support wildcard search, you can use both keywords or either of them.

It doesn't mean you need comment every cli on the spot, you can review and modify ~/.bash_history later.

- Tip 8

Commands containing relative path is not re-usable because they need be executed in certain directory. I exclude these command from ~/.bash_history

Here is my setup in `~/.bashrc` to ignore them: `export HISTIGNORE="cd [a-zA-Z0-9_.*]*:mv [a-zA-Z0-9_.*]*"`

More BASH Tricks

Transferring files without ftp or scp:

```
$ nc -l -p 1234 | uncompress -c | tar xvfp -
```

And on the sending server run:

```
$ tar cfp - /some/dir | compress -c | nc -w 3 [destination] 1234
```

Password-less ssh:

```
ssh-keygen -t dsa -C your.email@ddress
```

Enter a passphrase for your key. This puts the secret key in ~/.ssh/id_dsa and the public key in ~/.ssh/id_dsa.pub. Now see whether you have an ssh-agent running at present:

```
echo $SSH_AGENT_PID
```

Most window managers will run it automatically if it's installed. If not, start one up:

```
eval $(ssh-agent)
```

Now, tell the agent about your key:

```
ssh-add
```

and enter your passphrase. You'll need to do this each time you log in; if you're using X, try adding

```
SSH_ASKPASS=ssh-askpass ssh-add
```

to your .xsession file. (You may need to install ssh-askpass.)

```
ssh-copy-id -i ~/.ssh/id_dsa.pub user@server
It is a more graceful way of doing what copying the public key to #server:~/.ssh/authorized_keys file does.
```

Now for each server you log into, create the directory ~/.ssh and copy the file #~/.ssh/id_dsa.pub into it as ~/.ssh/authorized_keys . If you started the ssh-#agent by hand, kill it with

```
ssh-agent -k
```

when you log out.

Eliminate suid binaries:

```
find / -perm +6000 -type f -exec ls -ld {} \; > setuid.txt &
```

This will create a file called setuid.txt that contains the details of all of the matching files present on your system. To remove the s bits of any tools that you don't use, type:

```
chmod a-s program
```

Backup your bootsector:

```
dd if=/dev/hda of=bootsector.img bs=512 count=1
Restore your bootsector:
<source lang="bash">
dd if=bootsector.img of=/dev/hda
```

Where did that drive mount?:

```
dmesg | grep SCSI
```

This will filter out recognised drive specs from the dmesg output. You'll probably turn up some text like:

```
SCSI device sda: 125952 512-byte hdwr sectors (64 MB)
```

Unmount busy drives:

```
lsof +D /mnt/windows
```

Access your programs remotely:

```
X11Forwarding yes
```

```
ssh -X 192.168.0.2 gimp
```

Grabbing a screenshot without X:

```
chvt 7; sleep 2; import -display :0.0 -window root sshot1.png; chvt 1;
```

Finding the biggest files:

```
ls -lSrh
```

Quick file sharing trick (Share one or more files): If you want to share a file or folder from your current directory in your local network then you can quickly create a web server for this purpose. All you require is python installed on your system. Once Python is installed, this is what you have to do :

```
$ python -m SimpleHTTPServer
```

The above command will start a basic HTTP web server on port 8000 of your system. You can verify it by typing following in your web browser :

```
http://localhost:8000/
```

So, other users on your network can easily download required files from your web server.

Learn how to use Ctrl-R effectively. This is the single thing I've learned over the last few years that's made the biggest difference.

To get the biggest bang, you want to enable permanent shared command line history. Here's how you do it in zsh:

setopt SHARE_HISTORY setopt EXTENDED_HISTORY HISTSIZE=10000000 SAVEHIST=10000000 HISTFILE=~/.history

My .history file goes back several years and has something like 200,000 commands in it. I rarely write longer command lines from scratch anymore. I just Ctrl-R for a similar command from the past.

This also helps you do forensics later on. If you've hacked together an ad-hoc pipeline on the command line, you can go back through your .history file to turn it into a reusable script. And if someone asks for details about a project that you did years ago, you know where to turn.

Other helpful bits: - Use alt-period instead of !$. - Use $() for command substitution instead of `` - Use $((1+1)) to do arithmetic in the command line.

Suppose you start entereing a command in your terminal, but realize that you want to execute another command before this one... use Esc-q to put your current entered line on a stack and clear your command line. Once you entered a new command (which can be a blank line), the line which were placed in the stack will be pop'ed back onto your command line. (Works on Mac, probably Linux as well).

Edit: This isn't working for a lot of people. If you're among them, check the comments. However, there might be another way, which I found (here – click for details and alternatives): Enter the command which you want to postpone executing Press `Ctrl`+`U` Enter one or more commands Press `Ctrl`+`Y` to get the command you removed with `Ctrl`+`U` back.

There are a lot of data manipulation problems that can be tackled with relatively simple Unix command lines. Before you try anything else, see if you can prototype something with simple tools. You might not want to put it into production as a shell script, but given the size of modern computer memories and the speed of SSD disks, a surprising number of jobs lend themselves to being addressed with brute force.

The two hard problems of data manipulation that the traditional Unix command line tools based on regular expressions like awk and sed deal with badly are CSV data and JSON structures. You quickly get into regexes that are messy and also that don't work. For those I recommend

csvkit 0.9.0 (beta) - "csvkit is a suite of utilities for converting to and working with CSV, the king of tabular file formats."

jq - "jq is like sed for JSON data – you can use it to slice and filter and map and transform structured data with the same ease that sed, awk, grep and friends let you play with text."

Splitting and Un-splitting Files

This splits a large file into 1024 MB chunks

```
split -b 1024m filename
```

The merges the parts into a single file again:

```
cat xa* > filename
```

Looking at Details of Process

When you know your process PID, look up the details in the `proc` filesystem.

```
cd /proc/<pid>
cd fd
# To look at the open file descripters
cat environ
# To look at the environment variables this process was initiated with
```

Quick & Secure login via SSH Keys: Quick Logins with ssh Client Keys

Awk One Liners: Page on Ork (Very useful for simple things like find replace, etc)

Resolving pid from port using sudo lsof -i :80

Other commands like top, ln, head, tail, grep, netstat, free, df, etc

You can create an entire directory tree with a single 'mkdir -p' command.

I do this in userdata for EC2 spot instances that I spawn each night. I put the sendmail queues on the ephemeral disks, since they have fast I/O. I need to recreate the whole queue structure from scratch on each bootup, so:

mkdir -p /var/spool/clientmqueue/qdir.0/{xf,qf,df} mkdir -p /var/spool/mqueue/qdir.0/{xf,qf,df}

technically that could be made even more terse, now that I look at it:

mkdir -p /var/spool/{clientmqueue,mqueue}/qdir.0/{xf,qf,df}

I take no credit for this tip, it's from a fantastic couple of pages on IBM's site:

UNIX tips: Learn 10 good UNIX usage habits

Edit: Just to sweeten this up a little, here's the super-duper mkdir for setting up sendmail queues:

```
mkdir -p /var/spool/{clientmqueue,mqueue}/qdir.{0,1,2}/{xf,qf,df}
```

and that creates

```
/var/spool/
  clientmqueue/
    qdir.0/
      xf
      qf
      df
    qdir.1/
      xf
      qf
      df
    qdir.2/
      xf
      qf
      df
  mqueue/
    qdir.0/
      xf
      qf
      df
    qdir.1/
      xf
      qf
      df
    qdir.2/
      xf
      qf
      df
```

And here's one I used today to set up chroot bind:

```
mkdir -p /var/lib/named/{etc,dev,var/{cache/bind,run/bind/run}}
```

If you find reading man pages tedious , search your command in explainshell.com for better representation.

To search for a certain string on all directories and sub-directories recursively

```
grep -r -H -E "regexp" *
```

To get the files that contain a certain word:

```
grep -r -o -H -E "regexp" * | awk -vFS=":" '{print $1;}' | sort | uniq
```

Do simple math on the command line

```
echo $((10*33))
```

More complex maths using the `bc` calculator language:

```
r=4
echo "3.141 * 2 * $r^2" | bc
100.512
```

Find and Kill a process

```
ps aux | grep "process_name" | grep -v grep | awk '{print $2;}' | while read p; do kill -9 $p; done
```

Named pipes and redirection (mkfifo)

Process Substitution

< (command) = treats the output of command as a file. diff -u < (ssh web{1,2} cat /etc/passwd)) shows you a unified diff between /etc/passwd on web1 and 2

Basics

- Learn basic BASH. Actually, read the whole BASH man page; it's pretty easy to follow and not that long. Alternate shells can be nice (zsh, csh, ksh, etc), but BASH is powerful and always available.
- Learn VI. There's really no competition for random Linux editing (even if you use Emacs or Eclipse most of the time).
- Know ssh, and the basics of passwordless authentication, via ssh-agent, ssh-add, etc.
- Be familiar with bash job management: &, Ctrl-Z, Ctrl-C, jobs, fg, bg, kill, etc.
- Basic file management: ls and ls -l (in particular, learn what every column in "ls -l" means), less, head, tail and tail -f, ln and ln -s (learn the differences and advantages of hard versus soft links), chown, chmod, du (for a quick summary of disk usage: du -sk *), df, mount.
- Basic network management: ip or ifconfig, dig.
- Know regular expressions well, and the various flags to grep/egrep. The -o, -A, and -B options are worth knowing.
- Learn to use yum to find and install packages

Everyday use

In bash, use Ctrl-R to search through command history. In bash, use Ctrl-W to kill the last word, and Ctrl-U to kill the line. See man readline for default keybindings in bash. There are a lot. For example Alt-. cycles through prevous arguments, and Alt-* expands a glob. To go back to the previous working directory: cd - If you are halfway through typing a command but change your mind, hit Alt-# to add a # at the beginning and enter it as a comment (or use Ctrl-A, #, enter). You can then return to it later via command history. Use xargs (or parallel). It's very powerful. Note you can control how many items execute per line (-L) as well as

parallelism (-P). If you're not sure if it'll do the right thing, use xargs echo first. Also, -I{} is handy. Examples: find . -name *.py | xargs grep some_function cat hosts | xargs -I{} ssh root@{} hostname pstree -p is a helpful display of the process tree. Use pgrep and pkill to find or signal processes by name (-f is helpful). Know the various signals you can send processes. For example, to suspend a process, use kill -STOP [pid]. For the full list, see man 7 signal Use nohup or disown if you want a background process to keep running forever. Check what processes are listening via netstat -lntp. See also lsof. In bash scripts, use set -x for debugging output. Use set -e to abort on errors. Consider using set -o pipefail as well, to be strict about errors (though this topic is a bit subtle). For more involved scripts, also use trap. In bash scripts, subshells (written with parentheses) are convenient ways to group commands. A common example is to temporarily move to a different working directory, e.g.

1. do something in current dir

(cd /some/other/dir; other-command)

1. continue in original dir

In bash, note there are lots of kinds of variable expansion. Checking a variable exists: ${name:?error message}. For example, if a bash script requires a single argument, just write input_file=${1:?usage: $0 input_file}. Arithmetic expansion: i=$(((i + 1) % 5)). Sequences: {1..10}. Trimming of strings: ${var%suffix} and ${var#prefix}. For example if var=foo.pdf, then echo ${var%.pdf}.txt prints "foo.txt". The output of a command can be treated like a file via <(some command). For example, compare local /etc/hosts with a remote one: diff /etc/hosts <(ssh somehost cat /etc/hosts) Know about "here documents" in bash, as in cat <<EOF .... In bash, redirect both standard output and standard error via: some-command >logfile 2>&1. Often, to ensure a command does not leave an open file handle to standard input, tying it to the terminal you are in, it is also good practice to add "</dev/null". Use man ascii for a good ASCII table, with hex and decimal values. On remote ssh sessions, use screen or dtach to save your session, in case it is interrupted. In ssh, knowing how to port tunnel with -L or -D (and occasionally -R) is useful, e.g. to access web sites from a remote server. It can be useful to make a few optimizations to your ssh configuration; for example, this .ssh/config contains settings to avoid dropped connections in certain network environments, not require confirmation connecting to new hosts, forward authentication, and use compression (which is helpful with scp over low-bandwidth connections):

```
TCPKeepAlive=yes
ServerAliveInterval=15
ServerAliveCountMax=6
StrictHostKeyChecking=no
Compression=yes
ForwardAgent=yes
```

- To get the permissions on a file in octal form, which is useful for system configuration but not available in "ls" and easy to bungle, use something like

```
stat -c '%A %a %n' /etc/timezone
```

Data processing

- To convert HTML to text: lynx -dump -stdin
- If you must handle XML, xmlstarlet is old but good.
- For JSON, use jq.
- For Amazon S3, s3cmd is convenient (albeit immature, with occasional misfeatures).
- Know about sort and uniq (including uniq's -u and -d options).
- Know about cut, paste, and join to manipulate text files. Many people use cut but forget about join.
- It is remarkably helpful sometimes that you can do set intersection, union, and difference of text files via sort/uniq. Suppose a and b are text files that are already uniqued. This is fast, and works on files of arbitrary size, up to many gigabytes. (Sort is not limited by memory, though you may need to use the -T option if /tmp is on a small root partition.)

```
cat a b | sort | uniq > c   # c is a union b
cat a b | sort | uniq -d > c   # c is a intersect b
cat a b b | sort | uniq -u > c   # c is set difference a - b
```

- Know that locale affects a lot of command line tools, including sorting order and performance. Most Linux installations will set LANG or other locale variables to a local setting like US English. This can make sort or other commands run many times slower. (Note that even if you use UTF-8 text, you can safely sort by ASCII order for many purposes.) To disable slow i18n routines and use traditional byte-based sort order, use export LC_ALL=C (in fact, consider putting this in your .bashrc).
- Know basic awk and sed for simple data munging. For example, summing all numbers in the third column of a text file: awk '{ x += $3 } END { print x }'. This is probably 3X faster and 3X shorter than equivalent Python.
- To replace all occurrences of a string in place, in one or more files:

perl -pi.bak -e 's/old-string/new-string/g' my-files-*.txt

- To rename many files at once according to a pattern, use rename. (Or if you want something more general, my own tool repren may help.)

rename 's/\.bak$//' *.bak

- Use shuf to shuffle or select random lines from a file.
- Know sort's options. Know how keys work (-t and -k). In particular, watch out that you need to write -k1,1 to sort by only the first field; -k1 means sort according to the whole line.

Stable sort (sort -s) can be useful. For example, to sort first by field 2, then secondarily by field 1, you can use sort -k1,1 | sort -s -k2,2

If you ever need to write a tab literal in a command line in bash (e.g. for the -t argument to sort), press Ctrl-V <tab> or write $'\t' (the latter is better as you can copy/paste it). For binary files, use hd for simple hex dumps and bvi for binary editing. Also for binary files, strings (plus grep, etc.) lets you find bits of text. To convert text encodings, try iconv. Or uconv for more advanced use; it supports some advanced Unicode things. For example, this command lowercases and removes all accents (by expanding and dropping them): uconv -f utf-8 -t utf-8 -x '::Any-Lower; ::Any-NFD; [:Nonspacing Mark:] >; ::Any-NFC; ' < input.txt > output.txt To split files into pieces, see split (to split by size) and csplit (to split by a pattern).

System debugging

- For web debugging, curl and curl -I are handy, and/or their wget equivalents.

To know disk/cpu/network status, use iostat, netstat, top (or the better htop), and (especially) dstat. Good for getting a quick idea of what's happening on a system.

- To know memory status, run and understand the output of free and vmstat. In particular, be aware the "cached" value is memory held by the Linux kernel as file cache, so effectively counts toward the "free" value.

Java system debugging is a different kettle of fish, but a simple trick on Sun's and some other JVMs is that you can run kill -3 <pid> and a full stack trace and heap summary (including generational garbage collection details, which can be highly informative) will be dumped to stderr/logs.

- Use mtr as a better traceroute, to identify network issues.

For looking at why a disk is full, ncdu saves time over the usual commands like "du -sk *". To find which socket or process is using bandwidth, try iftop or nethogs. The ab tool (comes with Apache) is helpful for quick-and-dirty checking of web server performance. For more complex load testing, try siege.

- For more serious network debugging, wireshark or tshark.

Know strace and ltrace. These can be helpful if a program is failing, hanging, or crashing, and you don't know why, or if you want to get a general idea of performance. Note the profiling option (-c), and the ability to attach to a running process (-p).

- Know about ldd to check shared libraries etc.
- Know how to connect to a running process with gdb and get its stack traces.
- Use /proc. It's amazingly helpful sometimes when debugging live problems. Examples: /proc/cpuinfo, /proc/xxx/cwd, /proc/xxx/exe, /proc/xxx/fd/, /proc/xxx/smaps.
- When debugging why something went wrong in the past, sar can be very helpful. It shows historic statistics on CPU, memory, network, etc.
- For deeper systems and performance analyses, look at stap (systemtap) and perf.
- Confirm what Linux distribution you're using (works on most distros): "lsb_release -a"
- Use dmesg whenever something's acting really funny (it could be hardware or driver issues).

Regular Expressions

Basic concepts

A regular expression, often called a **pattern**, is an expression used to specify a set of strings required for a particular purpose. A simple way to specify a finite set of strings is to list its elements or members. However, there are often more concise ways to specify the desired set of strings. For example, the set containing the three strings "Handel", "Händel", and "Haendel" can be specified by the **pattern** `H(ä|ae?)ndel`; we say that this pattern **matches** each of the three strings. In most formalisms, if there exists at least one regular expression that matches a particular set then there exists an infinite number of other regular expression that also match it—the specification is not unique. Most formalisms provide the following operations to construct regular expressions.

Boolean "or"
: A vertical bar separates alternatives. For example, Template:Code can match "gray" or "grey".

Grouping
: Parentheses are used to define the scope and precedence of the operators (among other uses). For example, `gray|grey` and Template:Code are equivalent patterns which both describe the set of "gray" or "grey".

Quantification
: A quantifier after a token (such as a character) or group specifies how often that preceding element is allowed to occur. The most common quantifiers are the question mark `?`, the asterisk `*` (derived from the Kleene star), and the plus sign `+` (Kleene plus).

`?`	The question mark indicates *zero or one* occurrences of the preceding element. For example, `colou?r` matches both "color" and "colour".
`*`	The asterisk indicates *zero or more* occurrences of the preceding element. For example, `ab*c` matches "ac", "abc", "abbc", "abbbc", and so on.
`+`	The plus sign indicates *one or more* occurrences of the preceding element. For example, `ab+c` matches "abc", "abbc", "abbbc", and so on, but not "ac".
`{n}`[1]	The preceding item is matched exactly *n* times.
`{min,}`[1]	The preceding item is matched *min* or more times.
`{min,max}`[1]	The preceding item is matched at least *min* times, but not more than *max* times.

These constructions can be combined to form arbitrarily complex expressions, much like one can construct arithmetical expressions from numbers and the operations +, −, ×, and ÷. For example, `H(ae?|ä)ndel` and Template:Code are both valid patterns which match the same strings as the earlier example, `H(ä|ae?)ndel`.

The precise syntax for regular expressions varies among tools and with context; more detail is given in the *Syntax* section.

Formal language theory

Regular expressions describe regular languages in formal language theory. They have the same expressive power as regular grammars.

Formal definition

Regular expressions consist of constants and operator symbols that denote sets of strings and operations over these sets, respectively. The following definition is standard, and found as such in most textbooks on formal language theory.[2][3] Given a finite alphabet Σ, the following constants are defined as regular expressions:

- (*empty set*) ∅ denoting the set ∅.
- (*empty string*) ε denoting the set containing only the "empty" string, which has no characters at all.
- (*literal character*) `a` in Σ denoting the set containing only the character *a*.

Given regular expressions R and S, the following operations over them are defined to produce regular expressions:

- (*concatenation*) *RS* denotes the set of strings that can be obtained by concatenating a string in R and a string in S. For example, {"ab", "c"}{"d", "ef"} = {"abd", "abef", "cd", "cef"}.
- (*alternation*) *R* | *S* denotes the set union of sets described by *R* and *S*. For example, if R describes {"ab", "c"} and S describes {"ab", "d", "ef"}, expression *R* | *S* describes {"ab", "c", "d", "ef"}.
- (*Kleene star*) *R** denotes the smallest superset of set described by *R* that contains ε and is closed under string concatenation. This is the set of all strings that can be made by concatenating any finite number (including zero) of strings from set described by *R*. For example, {"0","1"}* is the set of all finite binary strings (including the empty string), and {"ab", "c"}* = {ε, "ab", "c", "abab", "abc", "cab", "cc", "ababab", "abcab", … }.

To avoid parentheses it is assumed that the Kleene star has the highest priority, then concatenation and then alternation. If there is no ambiguity then parentheses may be omitted. For example, `(ab)c` can be written as `abc`, and `a|(b(c*))` can be written as `a|bc*`. Many textbooks use the symbols `U`, `+`, or `V` for alternation instead of the vertical bar.

Examples:

- `a|b*` denotes {ε, "a", "b", "bb", "bbb", …}
- `(a|b)*` denotes the set of all strings with no symbols other than "a" and "b", including the empty string: {ε, "a", "b", "aa", "ab", "ba", "bb", "aaa", …}
- `ab*(c|ε)` denotes the set of strings starting with "a", then zero or more "b"s and finally optionally a "c": {"a", "ac", "ab", "abc", "abb", "abbc", …}
- `(0|(1(01*0)*1))*` denotes the set of binary numbers that are multiples of 3: { ε, "0", "00", "11", "000", "011", "110", "0000", "0011", "0110", "1001", "1100", "1111", "00000", … }

Expressive power and compactness

The formal definition of regular expressions is purposely parsimonious and avoids defining the redundant quantifiers `?` and `+`, which can be expressed as follows: `a+` = `aa*`, and `a?` = `(a|ε)`. Sometimes the complement operator is added, to give a *generalized regular expression*; here R^C matches all strings over Σ* that do not match *R*. In principle, the complement operator is redundant, as it can always be circumscribed by using the other operators. However, the process for computing such a representation is complex, and the result may require expressions of a size that is double exponentially larger.[4][5]

Regular expressions in this sense can express the regular languages, exactly the class of languages accepted by deterministic finite automata. There is, however, a significant difference in compactness. Some classes of regular languages can only be described by deterministic finite automata whose size grows exponentially in the size of the shortest equivalent regular expressions. The standard example here is the languages L_k consisting of all strings over the alphabet $\{a,b\}$ whose k^{th}-from-last letter equals a. On one hand, a regular expression describing L_4 is given by $(a\mid b)^*a(a\mid b)(a\mid b)(a\mid b)$.

Generalizing this pattern to L_k gives the expression: $(a\mid b)^*a\underbrace{(a\mid b)(a\mid b)\cdots(a\mid b)}_{k-1\text{ times}}. \,$

On the other hand, it is known that every deterministic finite automaton accepting the language L_k must have at least 2^k states. Luckily, there is a simple mapping from regular expressions to the more general nondeterministic finite automata (NFAs) that does not lead to such a blowup in size; for this reason NFAs are often used as alternative representations of regular languages. NFAs are a simple variation of the type-3 grammars of the Chomsky hierarchy.[2]

Finally, it is worth noting that many real-world "regular expression" engines implement features that cannot be described by the regular expressions in the sense of formal language theory; rather, they implement *regexes*. See below for more on this.

Deciding equivalence of regular expressions

As seen in many of the examples above, there is more than one way to construct a regular expression to achieve the same results.

It is possible to write an algorithm that, for two given regular expressions, decides whether the described languages are equal; the algorithm reduces each expression to a minimal deterministic finite state machine, and determines whether they are isomorphic (equivalent).

The redundancy can be eliminated by using Kleene star and set union to find an interesting subset of regular expressions that is still fully expressive, but perhaps their use can be restricted.Template:Clarify This is a surprisingly difficult problem. As simple as the regular expressions are, there is no method to systematically rewrite them to some normal form. The lack of axiom in the past led to the star height problem. In 1991, Dexter Kozen axiomatized regular expressions with Kleene algebra;[6] see Kleene algebra#History for details.

Syntax

A regex *pattern* matches a target *string*. The pattern is composed of a sequence of *atoms*. An atom is a single point within the regex pattern which it tries to match to the target string. The simplest atom is a literal, but grouping parts of the pattern to match an atom will require using `( )` as metacharacters. Metacharacters help form: *atoms*; *quantifiers* telling how many atoms (and whether it is a *greedy* quantifier or not); a logical OR character, which offers a set of alternatives, and a logical NOT character, which negates an atom's existence; and backreferences to refer to previous atoms of a completing pattern of atoms. A match is made, not when all the atoms of the string are matched, but rather when all the pattern atoms in the regex have matched. The idea is to make a small pattern of characters stand for a large number of possible strings, rather than compiling a large list of all the literal possibilities.

Depending on the regex processor there are about fourteen metacharacters, characters that may or may not have their literal character meaning, depending on context, or whether they are "escaped", i.e. preceded by an escape sequence, in this case, the backslash `\`. Modern and POSIX extended regexes use metacharacters more often than their literal meaning, so to avoid "backslash-osis" it makes sense to have a metacharacter escape to a literal mode; but starting out, it makes more sense to have the four bracketing metacharacters `( )` and `{ }` be primarily literal, and "escape" this usual meaning to become metacharacters. Common standards implement both. The usual metacharacters are `{}[]()^$.|*+?` and `\`. The usual characters that become metacharacters when escaped are `dsw.DSW` and `N`.

Delimiters

When entering a regex in a programming language, they may be represented as a usual string literal, hence usually quoted; this is common in C, Java, and Python for instance, where the regex `re` is entered as `"re"`. However, they are often written with slashes as delimiters, as in `/re/` for the regex `re`. This originates in ed, where `/` is the editor command for searching, and an expression `/re/` can be used to specify a range of lines (matching the pattern), which can be combined with other commands on either side, most famously `g/re/p` as in grep ("global regex print"), which is included in most Unix-based operating systems, such as Linux distributions. A similar convention is used in sed, where search and replace is given by `s/re/replacement/` and patterns can be joined with a comma to specify a range of lines as in `/re1/,/re2/`. This notation is particularly well-known due to its use in Perl, where it forms part of the syntax distinct from normal string literals. In some cases, such as sed and Perl, alternative delimiters can be used to avoid collision with contents, and to avoid having to escape occurrences of the delimiter character in the contents. For example, in sed the command `s,/,X,` will replace a `/` with an `X`, using commas as delimiters.

Standards

The IEEE POSIX standard has three sets of compliance: BRE,[7] ERE, and SRE for Basic, Extended, and Simple Regular Expressions. SRE is deprecated,[8] in favor of BRE, as both provide backward compatibility. The subsection below covering the *character classes* applies to both BRE and ERE.

BRE and ERE work together. ERE adds `?`, `+`, and `|`, and it removes the need to escape the metacharacters `( )` and `{ }`, which are *required* in BRE. Furthermore, as long as the POSIX standard syntax for regexes is adhered to, there can be, and often is, additional syntax to serve specific (yet POSIX compliant) applications. Although POSIX.2 leaves some implementation specifics undefined, BRE and ERE provide a "standard" which has since been adopted as the default syntax of many tools, where the choice of BRE or ERE modes is usually a supported option. For example, GNU grep has the following options: "grep -E" for ERE, and "grep -G" for BRE (the default), and "grep -P" for Perl regexes.

Perl regexes have become a de facto standard, having a rich and powerful set of atomic expressions. Perl has no "basic" or "extended" levels, where the `( )` and `{ }` may or may not have literal meanings. They are always metacharacters, as they are in "extended" mode for POSIX. To get their *literal* meaning, you escape them. Other metacharacters are known to be literal or symbolic based on context alone. Perl offers much more functionality: "lazy" regexes, backtracking, named capture groups, and recursive patterns, all of which are powerful additions to POSIX BRE/ERE. (See lazy matching below.)

POSIX basic and extended

In the POSIX standard, Basic Regular Syntax, BRE, requires that the metacharacters `( )` and `{ }` be designated `\(\)` and `\{\}`, whereas Extended Regular Syntax, ERE, does not.

Metacharacter	Description
`.`	Matches any single character (many applications exclude newlines, and exactly which characters are considered newlines is flavor-, character-encoding-, and platform-specific, but it is safe to assume that the line feed character is included). Within POSIX bracket expressions, the dot character matches a literal dot. For example, `a.c` matches "abc", etc., but `[a.c]` matches only "a", ".", or "c".
`[ ]`	A bracket expression. Matches a single character that is contained within the brackets. For example, `[abc]` matches "a", "b", or "c". `[a-z]` specifies a range which matches any lowercase letter from "a" to "z". These forms can be mixed: `[abcx-z]` matches "a", "b", "c", "x", "y", or "z", as does `[a-cx-z]`. The `-` character is treated as a literal character if it is the last or the first (after the `^`, if present) character within the brackets: `[abc-]`, `[-abc]`. Note that backslash escapes are not allowed. The `]` character can be included in a bracket expression if it is the first (after the `^`) character: `[]abc]`.
`[^ ]`	Matches a single character that is not contained within the brackets. For example, `[^abc]` matches any character other than "a", "b", or "c". `[^a-z]` matches any single character that is not a lowercase letter from "a" to "z". Likewise, literal characters and ranges can be mixed.
`^`	Matches the starting position within the string. In line-based tools, it matches the starting position of any line.
`$`	Matches the ending position of the string or the position just before a string-ending newline. In line-based tools, it matches the ending position of any line.
`( )`	Defines a marked subexpression. The string matched within the parentheses can be recalled later (see the next entry, `\n`). A marked subexpression is also called a block or capturing group. **BRE mode requires Template:Nowrap**.
`\n`	Matches what the *n*th marked subexpression matched, where *n* is a digit from 1 to 9. This construct is vaguely defined in the POSIX.2 standard. Some tools allow referencing more than nine capturing groups.
`*`	Matches the preceding element zero or more times. For example, `ab*c` matches "ac", "abc", "abbbc", etc. `[xyz]*` matches "", "x", "y", "z", "zx", "zyx", "xyzzy", and so on. `(ab)*` matches "", "ab", "abab", "abababa", and so on.
Template:Nowrap	Matches the preceding element at least *m* and not more than *n* times. For example, `a{3,5}` matches only "aaa", "aaaa", and "aaaaa". This is not found in a few older instances of regexes. **BRE mode requires `Template:Nowrap}`**.

Examples:

- `.at` matches any three-character string ending with "at", including "hat", "cat", and "bat".
- `[hc]at` matches "hat" and "cat".
- `[^b]at` matches all strings matched by `.at` except "bat".
- `[^hc]at` matches all strings matched by `.at` other than "hat" and "cat".
- `^[hc]at` matches "hat" and "cat", but only at the beginning of the string or line.
- `[hc]at$` matches "hat" and "cat", but only at the end of the string or line.
- `\[.\]` matches any single character surrounded by "[" and "]" since the brackets are escaped, for example: "[a]" and "[b]".
- `s.*` matches s followed by zero or more characters, for example: "s" and "saw" and "seed".

POSIX extended

The meaning of metacharacters escaped with a backslash is reversed for some characters in the POSIX Extended Regular Expression (ERE) syntax. With this syntax, a backslash causes the metacharacter to be treated as a literal character. So, for example, `\( \)` is now `( )` and `\{ \}` is now `{ }`. Additionally, support is removed for `\n` backreferences and the following metacharacters are added:

Metacharacter	Description
`?`	Matches the preceding element zero or one time. For example, `ab?c` matches only "ac" or "abc".
`+`	Matches the preceding element one or more times. For example, `ab+c` matches "abc", "abbc", "abbbc", and so on, but not "ac".
`|`	The choice (also known as alternation or set union) operator matches either the expression before or the expression after the operator. For example, `abc|def` matches "abc" or "def".</td>

Examples:

- `[hc]+at` matches "hat", "cat", "hhat", "chat", "hcat", "cchchat", and so on, but not "at".
- `[hc]?at` matches "hat", "cat", and "at".
- `[hc]*at` matches "hat", "cat", "hhat", "chat", "hcat", "cchchat", "at", and so on.
- `cat|dog` matches "cat" or "dog".

POSIX Extended Regular Expressions can often be used with modern Unix utilities by including the command line flag *-E*.

Character classes

The character class is the most basic regex concept after a literal match. It makes one small sequence of characters match a larger set of characters. For example, `[A-Z]` could stand for the upper case alphabet, and `\d` could mean any digit. Character classes apply to both POSIX levels.

When specifying a range of characters, such as `[a-Z]` (i.e. lowercase *a* to upper-case *z*), the computer's locale settings determine the contents by the numeric ordering of the character encoding. They could store digits in that sequence, or the ordering could be *abc...zABC...Z*, or *aAbBcC...zZ*. So the POSIX standard defines a character class, which will be known by the regex processor installed. Those definitions are in the following table:

POSIX	Non-standard	Perl/Tcl	Vim	ASCII	Description
`[:alnum:]`				`[A-Za-z0-9]`	Alphanumeric characters
	`[:word:]`	`\w`	`\w`	`[A-Za-z0-9_]`	Alphanumeric characters plus "_"
		`\W`	`\W`	`[^A-Za-z0-9_]`	Non-word characters
`[:alpha:]`			`\a`	`[A-Za-z]`	Alphabetic characters
`[:blank:]`			`\s`	`[ \t]`	Space and tab
		`\b`	`\< \>`	`(?<=\W)(?=\w)|(?<=\w)(?=\W)`	Word boundaries
`[:cntrl:]`				`[\x00-\x1F\x7F]`	Control characters
`[:digit:]`		`\d`	`\d`	`[0-9]`	Digits
		`\D`	`\D`	`[^0-9]`	Non-digits
`[:graph:]`				`[\x21-\x7E]`	Visible characters
`[:lower:]`			`\l`	`[a-z]`	Lowercase letters
`[:print:]`			`\p`	`[\x20-\x7E]`	Visible characters and the space character
`[:punct:]`				`[][!"#$%&'()*+,./:;<=>?@\^_`{|}~-]`	Punctuation characters
`[:space:]`		`\s`	`\_s`	`[ \t\r\n\v\f]`	Whitespace characters
		`\S`	`\S`	`[^ \t\r\n\v\f]`	Non-whitespace characters
`[:upper:]`			`\u`	`[A-Z]`	Uppercase letters
`[:xdigit:]`			`\x`	`[A-Fa-f0-9]`	Hexadecimal digits

POSIX character classes can only be used within bracket expressions. For example, `[[:upper:]ab]` matches the uppercase letters and lowercase "a" and "b".

An additional non-POSIX class understood by some tools is `[:word:]`, which is usually defined as `[:alnum:]` plus underscore. This reflects the fact that in many programming languages these are the characters that may be used in identifiers. The editor Vim further distinguishes *word* and *word-head* classes (using the notation `\w` and `\h`) since in many programming languages the characters that can begin an identifier are not the same as those that can occur in other positions.

Note that what the POSIX regex standards call *character classes* are commonly referred to as *POSIX character classes* in other regex flavors which support them. With most other regex flavors, the term *character class* is used to describe what POSIX calls *bracket expressions*.

Perl

Because of its expressive power and (relative) ease of reading, many other utilities and programming languages have adopted syntax similar to Perl's—for example, Java, JavaScript, Python, Ruby, Microsoft's .NET Framework, and XML Schema. Some languages and tools such as Boost and PHP support multiple regex flavors. Perl-derivative regex implementations are not identical and usually implement a subset of features found in Perl 5.0, released in 1994. Perl sometimes does incorporate features initially found in other languages, for example, Perl 5.10 implements syntactic extensions originally developed in PCRE and Python.[9]

Lazy matching

The three common quantifiers (`*`, `+` and `?`) are greedy by default because they match as many characters as possible.[10] The regex `".*"` applied to the string

```
"Ganymede," he continued, "is the largest moon in the Solar System."
```

matches the entire sentence instead of only the first quotation. The aforementioned quantifiers may therefore be made *lazy* or *minimal*, matching as few characters as possible, by appending a question mark: `".*?"` matches only the first quotation.[10]

Text Searching Tricks

Searching through PDF Files

Install PDFgrep:

```
$ sudo equo install pdfgrep
```

or

```
$ sudo apt-get install pdfgrep
```

PDFgrep uses many of the same command-line parameters as normal `grep` does, and then some.

Example, to search through a directory of PDF documents for the word FinTech (case-insensitive), do this:

```
$ pdfgrep -i fintech *.pdf
...
```

Remote Access Tricks

It is useful to access a device remotely to run a graphics-based program in that machine for support or for development collaboration. The most fundamental approach is via X-Windows. Other options such as VNC, Teamviewer and many other conferencing applications are also possible.

Basic SSH Remote Access

You can establish a terminal session from your local machine to a remote machine using SSH. To be able to do this, you need to know the user account name and password of the remote machine. It is even safer and more efficient if you have previously installed your public on the remote machine's ~/.ssh/authorized_keys file under the account that you will be accessing it as. Using the public/private key approach allows you batch processes as you will not be prompted for passwords, and is intrinsically more secure.

Configuration

If you do not yet have a public/private keypair, create one:

```
localhost $ ssh-keygen -t rsa
Generating public/private rsa key pair.
Enter file in which to save the key (/home/[me]/.ssh/id_rsa):
...
```

Deploy the public key (/home/[me]/.ssh/id_rsa.pub) to the remote machine. You will need to know the password to that machine, if this is not possible then the remote machine owner needs to copy and paste the content of the file into the remote machine's ~/.ssh/authorized_keys file.

```
localhost $ ssh-copy_id -i /home/[me]/.ssh/id_rsa.pub username@remotehost
...
```

How to open a remote SSH Terminal Session

Establish a secure SSH session to the remote host like this:

```
localhost $ ssh username@remotehost
...
remotehost $
```

X-Windows Remote Access

The idea is to access the remote device and to run a graphics-based application on that machine with the output and user input on your local machine.

Configurations

On the remote device, ensure that X11Forwarding is enabled in the `/etc/ssh/sshd_config`-file:

```
X11Forwarding yes
```

Restart the SSH daemon to effect this change:

```
$ sudo systemctl restart ssh
```

On your local machine, set the following X11-parameters in the `/etc/ssh/ssh_config`-file:

```
ForwardX11Trusted yes
ForwardX11 yes
```

How to launch a Remote Program

In the simplest form, start an SSH session in a command terminal, and then launch the remote program, e.g. firefox

```
localhost $ ssh -X username@remotehost
remotehost $ firefox &
[1234]
```

The `-X`-bit is important!

It is possible to directly invoke the remote program via SSH:

```
localhost $ ssh -X -T -f username@remotehost firefox
[1234]
```

In both cases, firefox will run on your local machine, even though you may well not have it installed. The firefox program will consume the remote host's resources, and just a little of your local resources and network bandwidth.

Internet Tricks

Copy a website

It is possible to copy (a.k.a. mirror) the entire content of some websites on your machine. A well protected website does not allow this through the use of restrictions set in .htaccess files, or through spidering protection mechanism such as Cloudflare. Use these tools as an information security checker against the production website.

HTTrack GUI & Command Line Tool

Use HTTrack at http://www.httrack.com to copy the content of many websites.

WGET Command line tool

```
wget -r https://[some website]
```

Some websites have anti-harvesting protection, so this will not work:

```
wget -r https://matchi.biz
```

This should pull down the minimum of files, but not allow a the creation of a bulk copy of the website's resources, i.e. it is impossible to obtain a complete collection of all organization logos, for example.

How to watch Star Wars on a Text Terminal

If you have **telnet** installed on your machine (Windows/Linux/Apple/WeirdOS/etc..), do this:

```
$ telnet towel.blinkenlights.nl
```

...and observe the action-packed Episode IV in glorious 80x24 resolution:

Get the local Weather

```
$ curl http://wttr.in/Durban
Weather for City: Durban, South Africa

    \   /     Sunny
     .-.      27 – 28 °C
  – (   ) –   ↙ 10 km/h
     `-’      10 km
    /   \     0.0 mm

                                                Wed 07. Sep
┌─────────────────────────────┬─────────────────────────────┬─────────────────────────────┬─────────────────────────────┐
│            Morning          │             Noon            │           Evening           │            Night            │
├─────────────────────────────┼─────────────────────────────┼─────────────────────────────┼─────────────────────────────┤
│    \   /     Sunny          │    \   /     Sunny          │    \  /      Partly Cloudy  │ _`/"".-.     Patchy rain ne… │
│     .-.      20 °C          │     .-.      27 – 28 °C     │  _ /"".-.    23 – 25 °C     │  ,\_(   ).   14 °C           │
│  – (   ) –   ↘ 6 – 7 km/h   │  – (   ) –   ↙ 10 – 12 km/h │    \_(   ).  ↑ 18 – 20 km/h │  /(___(__)  ↑ 17 – 21 km/h  │
│     `-’      10 km          │     `-’      10 km          │    /(___(__) 10 km          │    ‘ ‘ ‘ ‘  10 km           │
│    /   \     0.0 mm | 0%    │    /   \     0.0 mm | 0%    │              0.0 mm | 0%   │   ‘ ‘ ‘ ‘   0.2 mm | 53%   │
└─────────────────────────────┴─────────────────────────────┴─────────────────────────────┴─────────────────────────────┘

                                                Thu 08. Sep
┌─────────────────────────────┬─────────────────────────────┬─────────────────────────────┬─────────────────────────────┐
│            Morning          │             Noon            │           Evening           │            Night            │
├─────────────────────────────┼─────────────────────────────┼─────────────────────────────┼─────────────────────────────┤
│             Cloudy          │    \  /      Partly Cloudy  │    \  /      Partly Cloudy  │    \   /     Clear          │
│     .--.     21 °C          │  _ /"".-.    26 – 28 °C     │  _ /"".-.    25 – 28 °C     │     .-.      18 °C          │
│  .-(    ).   ↓ 3 km/h       │    \_(   ).  ↙ 14 – 16 km/h │    \_(   ).  ↙ 10 – 12 km/h │  – (   ) –   → 7 – 14 km/h  │
│ (___.__)__)  10 km          │    /(___(__) 10 km          │    /(___(__) 10 km          │     `-’      10 km          │
│              0.0 mm | 0%    │              0.0 mm | 0%    │              0.0 mm | 0%   │    /   \     0.0 mm | 0%    │
└─────────────────────────────┴─────────────────────────────┴─────────────────────────────┴─────────────────────────────┘

                                                Fri 09. Sep
┌─────────────────────────────┬─────────────────────────────┬─────────────────────────────┬─────────────────────────────┐
│            Morning          │             Noon            │           Evening           │            Night            │
├─────────────────────────────┼─────────────────────────────┼─────────────────────────────┼─────────────────────────────┤
│    \  /      Partly Cloudy  │             Cloudy          │ _`/"".-.     Patchy rain ne… │ _`/"".-.     Patchy rain ne… │
│  _ /"".-.    21 °C          │     .--.     23 – 25 °C     │  ,\_(   ).   23 – 25 °C     │  ,\_(   ).   20 °C           │
│    \_(   ).  ↑ 26 – 32 km/h │  .-(    ).   ↑ 19 – 22 km/h │   /(___(__)  ↑ 18 – 20 km/h │   /(___(__)  ↑ 11 – 13 km/h  │
│    /(___(__) 10 km          │ (___.__)__)  10 km          │     ‘ ‘ ‘ ‘  10 km          │     ‘ ‘ ‘ ‘  10 km          │
│              0.0 mm | 0%    │              0.0 mm | 7%    │    ‘ ‘ ‘ ‘   0.1 mm | 56%  │    ‘ ‘ ‘ ‘   0.1 mm | 40%  │
└─────────────────────────────┴─────────────────────────────┴─────────────────────────────┴─────────────────────────────┘
```

Other Awesome Console Tricks

Make BIG text

Install `figlet` if you don't yet have it. Then type something like this:

```
$ figlet "Hello  Matchi !"
 _   _          _ _                __  __       _       _     _
| | | | ___| | | ___    |  \/  | __ _| |_ ___| |__ (_) | |
| |_| |/ _ \ | |/ _ \   | |\/| |/ _` | __/ __| '_ \| | | |
|  _  |  __/ | | (_) |  | |  | | (_| | || (__| | | | | |_|
|_| |_|\___|_|_|\___/   |_|  |_|\__,_|\__\___|_| |_|_| (_)
```

You can save the output to a file, e.g.

```
$ figlet "Hello  Matchi !" > hello.txt
$ cat hello.txt
 _   _          _ _                __  __       _       _     _
| | | | ___| | | ___    |  \/  | __ _| |_ ___| |__ (_) | |
| |_| |/ _ \ | |/ _ \   | |\/| |/ _` | __/ __| '_ \| | | |
|  _  |  __/ | | (_) |  | |  | | (_| | || (__| | | | | |_|
|_| |_|\___|_|_|\___/   |_|  |_|\__,_|\__\___|_| |_|_| (_)
```

Neat, huh?

File Handling Tricks

There are no magic buttons on your stupid Windows or dumb-ass Mac desktop that will do any of the cool things below. Either upgrade your Windows PC to Linux, or log in to a Linux server using a terminal program like PuTTY if you want to do any of the stuff below:

Text Files

Convert Windows/DOS to Unix files

This converts the line-ending characters of `CR/LF` to just a `LF`-character on a text file:

```
$ sed $'s/\r$//'  -i [filename]
```

You may need to do if some hapless soul saved a BASH script using a Windows editor.

Convert Unix to Windows/DOS files

This converts the line-ending characters of a `LF`-character to the `CR/LF`-tuple on a text file:

```
$ sed $'s/$/\r/'  -i [filename]
```

There is no need for you to ever want to do this. This is mentioned here for completeness.

Remove trailing spaces from text files (both Unix & Windows/DOS)

Do this to tidy up your source code so that the versioning system does not register a change in a line only because of meaningless trailing spaces in the line.

```
$ sed -e 's/\s*$//' -i [filename]
```

You can tidy an entire source tree from the root directory up like this:

```
$ find . -type f -name "*.php" -exec sed -e 's/\s*$//' -i {} \; -print
```

It is useful to specify the file extension (`.php`, `.sh`, etc...) so that graphics files are not processed and accidentally corrupted.

Graphics Files

You may need to install the following packages on your Linux machine, if they are not already there:

- ImageMagick

Determine the dimensions of an image file

```
$ identify input.png
input.png PNG 421x226 421x226+0+0 8-bit sRGB 103KB 0.000u 0:00.000
```

The file is 421x226 pixels wide and high, and is 8-bit colour-encoded using the sRGB palette.

Resize an image

Resize the file *input.png* to 100 pixels wide with height scaled accordingly, and output the result to a file called *output.png*:

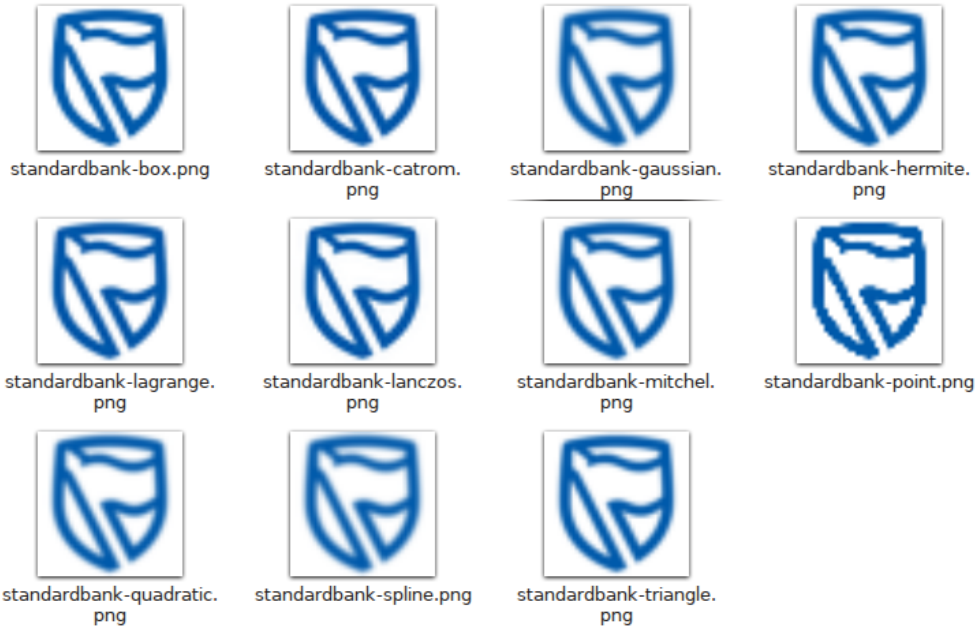
```
$ convert input.png -resize 100x output.png
```

A number of image resizing/scaling algorithms or filters exist of which one can be chosen. If no filter is set the Lanczos filter is used by default. The best filter choices for our work are:

- mitchell
- catrom
- gaussian

```
$ convert input.png -resize 100x -filter [filter]  output.png
```

Here are a series of enlargements of 37x37 pixel-sized scalings of the same high-resolution source image:



Resize into a white square

Centering a square frame with white background around an image is often necessary to when dealing with images or varying form factors and aspect ratios for which a consistently-sized, square display area needs to be allocated.

```
convert input.png -virtual-pixel white -set option:distort:viewport  "%[fx:max(w,h)]x%[fx:max(w,h)]-%[fx:max((h-w)/2,0)]-%[fx:max((w-h)/2,0)]" -filter mitchell -distort SRT 0 -res
```

Resize a directory of images

Resize a whole directory of image files *from* the current directory *into* a new directory `48x48`, sizing to 48x48 pixels wide and high:

```
$ for i in *; do convert $i -resize 48x48 ../48x48/$i; done
```

Convert a JPEG file to PNG

```
$ convert image.jpg image.png
```

Convert a PNG file to JPEG

It is good practise to host large images in JPEG format rather than PNG, as the file size is smaller. This converts the file format without changing the size of the image:

```
$ convert image.png image.jpg
```

Notes:

- You will loose any alpha-channels (the see-through bits) on the PNG image when you convert it to a JPEG file. The transparent areas will default to white
- The standard file name extension for JPEG files is .jpg

Optimize a large JPEG file

This can significantly reduce the download file size without compromising the image quality. It is very important to do this for large images such as page headers and photo galleries.

To strip just JPEG file headers, do this:

```
$ mogrify -strip image.jpg
```

If the JPEG image has already been optimized, doing this again may actually increase the size of the size. So check the file size before and after (and keep a backup of the original if you are not sure):

```
$ ls -al image.jpg
```

Stripping JPEG file headers is not always sufficient to reduce the size of a JPEG file. Frequently, the optional next step to reduce the image quality is required. An image of "65% quality" shows very few compression artefacts compared to the same image in "95% quality", it can further reduce the file size down to 25%, and it is sufficient for news articles and newsletters.

Here is how to compress a single image:

```
$ convert image.jpg -quality 65 image-65.jpg
```

To compress an entire directory of JPEG images in the directory "source" to the target directory "target":

```
$ cd source
$ for i in *.jpg ; do echo $i; convert $i -quality 65 ../target/$i; done
```

Optimize PNG files

Large images are usually in JPEG format and smaller images and larger images containing text should as a rule be held in PNG-format files. It is possible to compress some PNG files further with the *pngcrush* utility:

```
$ pngcrush image.png ../optimized_images/image.png
```

Note that it is not possible to perform this operation in-line on the current file - either specify a different file name or a different directory in which to put the output file.

Watermarking an Image

To apply a pre-made watermark image to an image file *inputimage.png* to get the output *watermarkedimage.png*:

```
composite -dissolve 50% -gravity center ~Watermark-80x80.png inputimage.png watermarkedimage.png
```

Base64-encode a Graphic file

This is useful for embedding images directly in HTML-files instead of serving them from a web server. A common use of this is for email signatures.

The base64 command encodes any file into the base-64 radix.

```
$ base64 mypicture.png
/9j/4AAQSkZJRgABAQEASABIAAD/2wBDAAMCAgMCAgMDAwMEAwMEBQgFBQQEBQoHBwYIDAoMDAsK
CwsNDhIQDQ4RDgsLEBYQERMUFRUVDA8XGBYUGBIUFRT/2wBDAQMEBAUEBQkFBQkUDQsNFBQUFBQU
FBQUFBQUFBQUFBQUFBQUFBQUFBQUFBQUFBQUFBQUFBQUFBQUFBT/wgARCAEMALwDAREA
AhEBAxEB/8QAGwABAAIDAQEAAAAAAAAAAAAAAAAAAQQGAQIDBQj/xAAcAQEAAQUBAQAAAAAAAAAAAAAA
AQIDBAUHBgj/2gAMAwEAAhADEAAAAfqikAAAAAMCHl+fy42DcxDCFUetvsKRnUgAAAAJa0OGH
...
```

A practical example of embedding an image into an HTML file, say, `myfile.html`, is to add the encoded text where you left off after having added the `<img>`-tag on the end of the file, e.g.

```
...
<img src="
```

Save and close the HTML file. Add the Base64-encoded text to the HTML file:

```
$ cat mypicture.png | base64 >> myfile.html
```

Close the `<img>`-tag off:

```
$ echo '"/>'  >> myfile.html
```

Open the HTML file again and continue to code in HTML.

Video Files

Handbrake

Install *Handbrake* as a GUI tool. This can shrink a 1GB .mp4 file into a 150MB .m4v file.

FFMPEG

Use the FFMPEG command line tool to manipulate large video files.

Install FFMPEG on Red Hat

Install it on Red Hat by installing the ATRPMS-repository first, and then enabling the new repository while installing ffmpeg:

```
$ sudo rpm --import http://packages.atrpms.net/RPM-GPG-KEY.atrpms
$ sudo rpm -ivh http://dl.atrpms.net/el6-x86_64/atrpms/stable/atrpms-repo-6-7.el6.x86_64.rpm
$ sudo yum -y --enablerepo=atrpms install ffmpeg
```

Convert a .mp4 file to a .avi file

This conversion also reduces the file size, in some cases by as much as 1GB to 50MB, with a slight degradation quality.

```
$ ffmpeg -i avfile.mp4 avfile.avi
```

PDF Files

You may need to install the following packages on your Linux machine, if they are not already installed:

- Ghostscript (gs)
- PDF Toolkit (pdftk)

Concatenate PDF files into one PDF file

To make a single file from all files in a directory called output.pdf, merged in alpha-numeric order:

```
$ pdftk *.pdf cat output output.pdf
```

Alternatively, you can name the individual files in a preferred order:

```
$ pdftk a.pdf b.pdf c.pdf d.pdf cat output output.pdf
```

Compact a huge PDF file

PDF files can grow completely out of all proportion, especially if they contain graphics. Most email providers have a maximum attachment size of 10MB. Downloadable PDF files should ideally be kept well under 100KB in size. To shrink PDF file input.pdf to output.pdf with minimum pixelation of graphics:

```
$ gs -sDEVICE=pdfwrite -dCompatibilityLevel=1.4 -dPDFSETTINGS=/screen -dNOPAUSE -dBATCH  -dQUIET -sOutputFile=output.pdf input.pdf
```

If there is still pixelation, try this:

```
$ gs -sDEVICE=pdfwrite -dCompatibilityLevel=1.4 -dPDFSETTINGS=/printer -dNOPAUSE -dBATCH  -dQUIET -sOutputFile=output.pdf input.pdf
```

HTML Files

Converting HTML files to PDF files

You can take any HTML file, even if it refers to external assets, can convert it to a PDF file.

```
$ wkhtmltopdf file.html file.pdf
```

It only works with basic HTML files, and files that contain JavaScript and complex CSS will not product good output. Try this:

```
$ wkhtmltopdf https://matchi.biz matchi.pdf
```

Yuck!

How to install this utility

This is how to install it on Ubuntu-based distros:

```
$ sudo apt-get install xvfb xfonts-100dpi xfonts-75dpi xfonts-scalable xfonts-cyrillic
$ sudo apt-get install wkhtmltopdf
```

X Windows - Launch Application as other user

```
$ xhost +local:
$ su - otheruser
$ export DISPLAY=:0.0
$ xeyes
```

Email Backup:

```
offlineimap for imap users.
```

Display blanking: Blank display:

```
$ xset -display :0 dpms force standby
```

Enable / Disable screen sleep:

```
xset s on; xset +dpms
xset s off; xset -dpms
```

Java Script Files

Minify a Java Script File

One of many ways to minify a JS file is to pass it into an online minifier service, such as https://javascript-minifier.com.

Pass the source file using the curl command:

```
$ curl -X POST -s --data-urlencode 'input@raphael.js' https://javascript-minifier.com/raw > raphael.min.js
```

To proces and entire directory of Java Script files, do this:

```
$ for i in *.js; do curl -X POST -s --data-urlencode 'input@${i}' https://javascript-minifier.com/raw > ${i%.js}.min.js; done
```

VI editor HOWTO

This is a quick on how to use `vi` editor for beginners. Some love it, others hate it. If this is not for you, then try `nano` instead.

Getting Started

Introduction

This tutorial will show you how to use vi, a powerful visual editor. This tutorial is designed to make you a proficient vi user without requiring a huge time commitment. In this vi tutorial, you'll learn how to move around, edit text, use insert mode, copy and paste text, and use important vim extensions like visual mode and multi-window editing.

If you either don't know or aren't comfortable using vi, then you owe it to yourself to take this tutorial and get up to speed with one of the most popular and powerful Linux/UNIX visual editing programs.

Introducing vi

There are many versions of vi and the version that comes with most Linux distributions is `gvim`, as it has a number of extensions that make vi a lot nicer. There is also a GUI-based version available called `gvim`, although the point here is to be able to use a tool that works in a text-based terminal

First Steps

Pick a file

Before using vi to edit files, you need to know how to use vi to move around in a file. vi has a lot of movement commands, and we're going to take a look at many of them. For this part of the tutorial, find an unimportant text file and load it into vi by typing:

```
vi myfile.txt
```

If you have vim installed, type `vim myfile.txt` . If you'd prefer to use gvim, type `gvim myfile.txt` . `myfile.txt` should be the name of a text file on your system.

Inside vi

After vi loads, you should see a part of the text file you loaded on your screen. Congratulations -- you're in vi! Unlike many editors, when vi starts up, it is in a special mode called *command mode* . This means that if you press `l` on the keyboard, instead of inserting an `l` into the file at the current cursor position, the cursor will move one character to the right instead. In command mode, the characters on your keyboard are used to send commands to vi rather than insert literal characters into the text. One of the most essential types of commands are movement commands; let's take a look at some.

Moving around

Moving in vi, part 1

When in command mode, you can use the `h` , `j` , `k` and `l` keys to move the cursor left, down, up and right respectively. If you're using a modern version of vi, you can also use the arrow keys for this purpose. The `h` , `j` , `k` and `l` keys are handy because once you're comfortable with them, you'll be able to move around in the file without moving your fingers from the home keyboard row. Try using `h` , `j` , `k` and `l` (and the arrow keys) to move around in the text file.

Try using `h` until you get to the beginning of a line. Notice that vi doesn't allow you to "wrap around" to the previous line by hitting `h` while you're on the first character. Likewise, you can't "wrap around" to the next line by hitting `l` at the end of a line.

Moving in vi, part 2

vi offers special shortcuts for jumping to the beginning or end of the current line. You can press `0` (zero) to jump to the first character of a line, and `$` to jump to the last character of the line. Try 'em and see. Since vi has so many handy movement commands, it makes a great "pager" (like the more or less commands.) Using vi as a pager will also help you to learn all the movement commands very quickly.

You can also use `CTRL` `F` and `CTRL` `B` to move forwards and backwards a page at a time. Modern versions of vi (like vim) will also allow you to use the `PGUP` and `PGDOWN` keys for this purpose.

Word moves, part 1

vi also allows you to move to the left or right by word increments. To move to the *first* character of the next word, press `w` . To move to the *last* character of the next word, press `e` . To move to the first character of the *previous* word, press `b` . Test 'em out.

Word moves, part 2

After playing around with the word movement commands, you may have noticed that vi considers words like `foo-bar-oni` as five separate words! This is because by default, vi delimits words by spaces *or* punctuation. `foo-bar-oni` is therefore considered five words: `foo` , `-` , `bar` , `-` and `oni` .

Sometimes, this is what you want, and sometimes it isn't. Fortunately, vi also understands the concept of a "bigword". vi delimits bigwords by *spaces or newlines only* . This means that while `foo-bar-oni` is considered five vi words, it's considered only one vi bigword.

Word moves, part 3

To jump around to the next and previous bigword, you can use a *capitalized* word move command. Use `W` to jump to the first character of the next bigword, `E` to jump to the last character of the next bigword, and `B` to jump to the first character of the previous bigword. Test 'em out, and compare the matching word and bigword movement commands until you understand their differences.

Bigger moves

We just have a few more commands to cover before it's time to start puting together our cheat sheet. You can use the `(` and `)` characters to move to the beginning of the previous and next sentence. In addition, you can hit `{` or Template:Error} to jump to the beginning of the current paragraph, and the beginning of the next.

Quitting

We've covered the basic movement commands, but there are another couple of commands that you need to know. Typing `:q` will quit vi. If this doesn't work, then you probably accidentally modified the file in some way. To tell vi to quit, throwing away any changes, type `:q!` . You should now be at the command prompt.

In vi, any command that begins with a `:` is said to be an *ex-mode* command. This is because vi has a built-in non-visual editor called *ex* . It can be used similarly to sed to perform line-based editing operations. In addition, it can also be used to quit, as we've just seen. If you ever hit the `Q` key while in command mode, you'll be transported to ex mode. If this ever happens to you, you'll be confronted with a : prompt, and hitting enter will scroll the entire screen upwards. To get back to good 'ol vi mode, simply type vi and hit enter.

Miscellaneous vi

Let's continue our rapid command-covering pace. In command-mode, you can jump to a particular line by typing `G` . To jump to the first line of a file, type `1G` . Note that `G` is capitalized.

If you want to jump to the next occurence of a particular text pattern, type `/<regexp>` and hit `Enter` . Replace <regexp> with the regular expression you're looking for. If you don't know how to use regular expressions, don't fret -- typing `/foo` will move to the next occurence of *foo* . The only thing you'll need to watch out for is when you want to refer to the literal `^` , `.` , `$` or `\` characters. Prefix these characters with a backslash (`\`), and you'll be set. For example, `/foo\.gif` will search for the next occurence of "foo.gif".

To repeat the search forwards, hit `n` . To repeat the search backwards, type `N` . As always, test these commands out in your very own vi editor. You can also type `/` `/` to repeat the last search.

Saving and Editing

Save and save as...

We've covered how you can use the *ex* command `:q` to quit from vi. If you want to save your changes, type `:w` . If you want to save your changes to another file, type `:w filename.txt` to save as *filename.txt* . If you want to save and quit, type `:x` or `:wq` .

In vim (and other advanced vi editors, like elvis) `:w` , you can have multiple buffers open at once. To open a file into a new window, type `:sp filename.txt` . `filename.txt` will appear open for editing in a new split window. To switch between windows, type `<CTR>w<CTR>w` (control-w twice). Any `:q` , `:q!` , `:w` and `:x` commands that you enter will only be applied to the currently-active window.

Simple edits

Now, it's time to start learning some of the simple editing commands. The commands that we'll cover here are considered *simple* because the commands keep you in command mode. The more complex editing commands automatically put you into insert mode -- a mode that allows you to enter literal data from the keyboard. We'll cover those in a bit.

For now, try moving over some characters and hitting `x` repeatedly. You'll see that `x` will delete the current character under the cursor. Now, move to the middle of the paragraph somewhere in your text file, and hit `J` (capitalized). You'll see that the `J` command tells vi to join the next line to the end of the current line. Now, move over a character and hit `r` , and then type in a new character; you'll see that the original character has been replaced. Finally, move to any line in the file and type `dd` . You'll see that `dd` deletes the current line of text.

Repeating and deleting

You can repeat any editing command by hitting the `.` key. If you experiment, you'll see that typing `dd...` will delete 4 lines, and `J.....` will join four lines. As usual, vi provides with another handy shortcut.

To delete text, you can also use the `d` command combined with any movement command. For example, `dw` will delete from the current position to the beginning of the next word; `d)` will delete up until the end of the next sentence, and `d}` will delete the remainder of the paragraph. Experiment with the `d` command and the other editing commands until you're comfortable with them.

Undo!

Now that we're experimenting with deletion, it would be a good time to learn how to undo any changes. By pressing `u` , the original version of vi allowed you to undo the last edit only. However, modern versions of vi like vim will allow you to repeatedly press `u` to continue to undo changes to your file. Try combining some `d` and `u` commands together.

Insert mode

So far, we've covered how to move around in vi, perform file i/o, and perform basic editing operations. However, I still haven't shown you how to actually type in free-form text! This was intentional, because vi's insert mode is a bit complicated at first. However, after you become comfortable with insert mode, its complexity (and flexibility) will become an asset.

In vi *insert mode* , you'll be able to enter text directly to the screen just like you can in many other visual editors. Once you've entered your modifications, you can hit escape to return to *command mode* . You can enter insert mode by pressing `i` or `a` . If you press `i` , your text will be *inserted* before the current character, and if you hit `a` , your text will be *appended* after the current character. Remember, after you enter your text, hit `<ESC>` to return to command mode.

Benefits of insert mode

Go ahead and try using the `a` and `i` commands. Hit either `a` or `i` , type some text, and then hit escape to get back to command mode. After hitting `a` or `i` , try hitting `<ENTER>` , and see what happens. Try using the arrow keys and the `DEL` key to get a feel for how insert mode works. By using the arrow keys and `DEL` key, you can perform significant editing steps without repeatedly entering and leaving insert mode.

Insert options

Here are some other handy ways to enter insert mode. Press `A` (capital) to begin appending to the *end* of the current line, regardless of your current position on the line. Likewise, press `I` (capital) to begin inserting text at the *beginning* of the current line. Press `o` to create a new blank line below the current line into which you can insert text, and press `O` (capital) to create a new line above the current line. To replace the entire current line with a new line, press `cc` . To replace everything from the current position to the end of the line, type `c$` . To replace everything from the current position to the beginning of the line, type `c0` .

In addition to performing a special operation, every one of these commands will put you into insert mode. After typing in your text, hit `<ESC>` to return to command mode.

Changing text

We've used the `c` (change) command a little bit so far when we typed `cc` , `c0` and `c$` . `cc` is a special form of the change command, similar to `dd` . the `c0` and `c$` commands are examples of using the change command in combination with a movement command. In this form, `c` works similarly to `d` , except that it leaves you in insert mode so that you can enter replacement text for the deleted region. Try combining some movement commands with `c` and test them out on your file (hint: `cW` , `ce` , `c(` .)

Compound Commands

vi *really* becomes powerful when you start using compound ("combo") commands, like `d{` and `cw` . In addition to these commands, you can also combine a number with any movement command, such as `3w` , which will tell vi to jump three words to the right. Here are some more movement "combo" command examples: `12b` , `4j` .

vi, in addition to allowing (number)(movement command) combinations, also allows `d` or `c` to be combined with a number or movement command. So, `d3w` will delete the next three words, `d2j` will delete the current and next two lines, etc. Test out some `c` and `d` combo moves to get a feel for how powerful and concise vi editing can be. Once these commands are second-nature, you'll be able to edit files at blazing speed.

Productivity features

So far, we've covered how to move, save and quit, perform simple edits and deletions, and use insert mode. With everything listed on the cheat sheet so far, you should be able to use vi to perform almost any task.

However, vi also has many more powerful commands. In this section, you'll learn how to *cut* , *copy* and *paste* , *search* and *replace* , and use *autoindent* features. These commands will help make vi more fun and productive.

Visual mode

The best way to cut and paste is to use *visual mode* , a special mode that has been added to modern versions of vi, like vim and elvis. You can think of visual mode as a "highlight text" mode. Once the text is highlighted, it can be copied or deleted, and then pasted. If you are using gvim, you can highlight by simply dragging the left mouse button over a particular region:

In addition, you can also enter visual mode by hitting `v` (this may be your only option if you are using vi from the console.) Then, by moving the cursor using movement commands (typically the arrow keys), you'll be able to highlight a region of text. Once highlighted, we are ready to cut or copy the text.

If you're copying the text, hit `y` (which stands for "yank"). If you're cutting the text, hit `d` . You'll be placed back in command mode. Now, move to the position where you'd like to insert the cut or copied text, and hit `P` to insert before the cursor, or `p` to insert after the cursor. Voila, the cut/copy and paste is complete! Test out several copy/cut and paste operations before advancing to the next section.

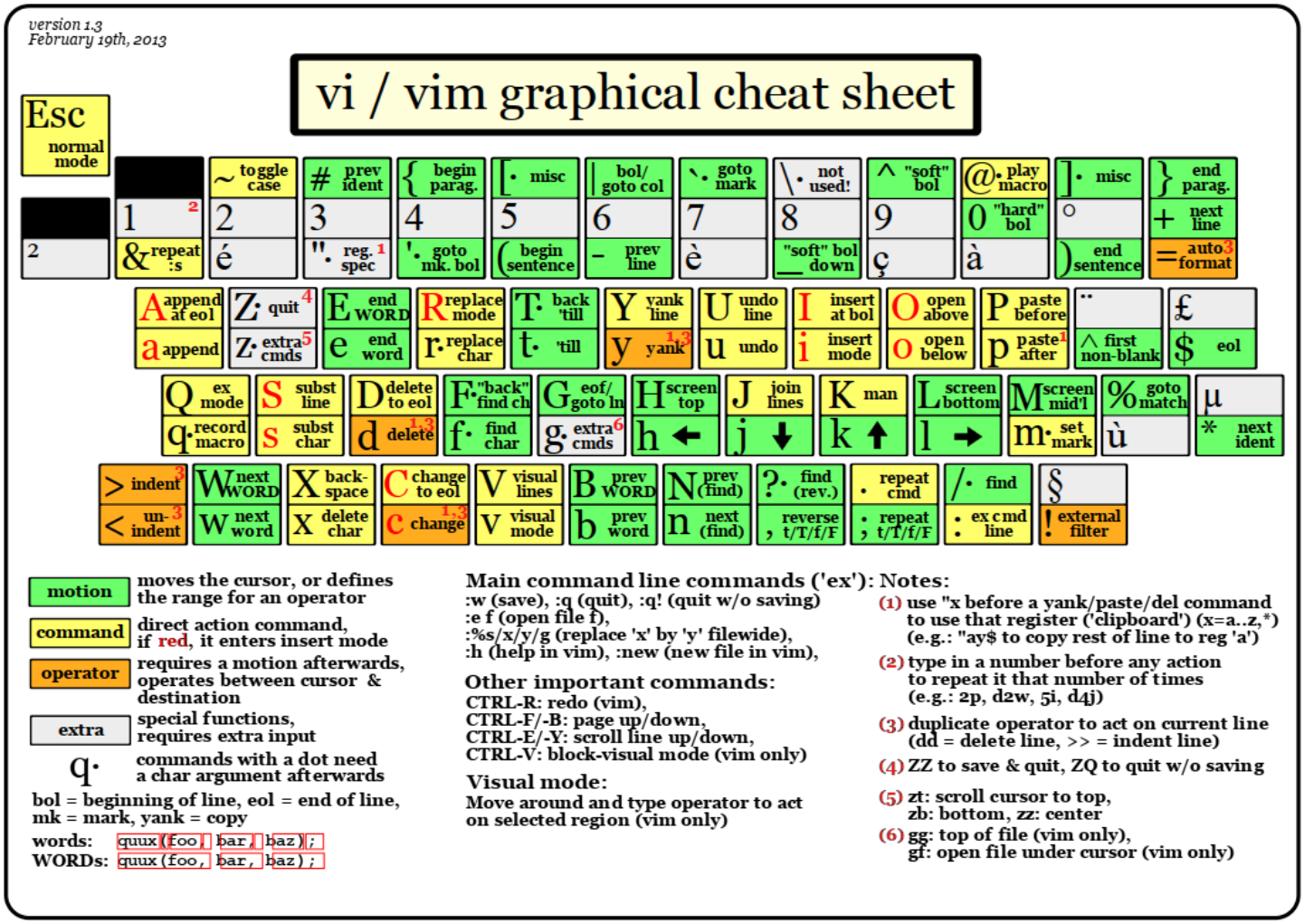
Replacing text

To replace patterns of text, we use *ex* mode. If you'd like to replace the first pattern that appears on the current line, type `:s/<regexp>/<replacement>/` and hit `<ENTER>` , where <regexp> is the pattern you'd like to match and <replacement> is the replacement string. To replace all matches on the current line, type `:s/<regexp>/<replacement>/g` and hit enter. To replace every occurence of this pattern in your file (normally what you want), type `:%s/<regexp>/<replacement>/g` . If you'd like to do a global replace, but have vi prompt you for each change, type `:%s/<regexp>/<replacement>/gc` (stands for "confirm") and hit `<ENTER>` .

Indentation

vi supports autoindentation, for when you are editing source code. Most modern versions of vi (like vim) will auto-enable autoindent mode when you are editing a source file (like a .c file, for example). When autoindent is enabled, you can use `<CTR>d` (control-d) to move one indent level to the left, and `<CTR>t` (control-t) to move one indent level to the right. If autoindent wasn't enabled automatically, you can manually enable it by typing in the *ex* command `:set autoindent` . You can also set the tab size on vi with the setting `:set tabstop`, and because the complexity of some of our code, we agree to always work with a tab stop of 2 characters, so this setting must always be `set tabstop=2`.

VI Cheatsheet



Guide to using the NANO Editor

This guide is meant to be a simple introduction to nano. It will quickly help you to become familiar with its basic operation.

Introduction

Purpose

This guide was written to cover basic operations in nano, and is meant to be very concise. For more information about nano check out: http://www.nano-editor.org.

First Steps

Opening and creating files

Opening and creating files is simple in nano. Simply type:

```
nano filename
```

Nano is a modeless editor (unlike VI) so you can start typing immediately to insert text. If you are editing a configuration file like `/etc/fstab` use the `-w` switch to disable wrapping on long lines as it might render the configuration file unparseable by whatever tools depend on it. For example:

```
nano -w /etc/fstab
```

Warning: It is important that you use the `-w` switch when opening a config file. Failure to do so may prevent your system from booting again or cause other bad things.

Saving and exiting

If you want to save the changes you've made, press `Ctrl` + `O`. To exit nano, type `Ctrl` + `X`. If you ask nano to exit from a modified file, it will ask you if you want to save it. Just press `N` in case you don't, or `Y` in case you do. It will then ask you for a filename. Just type it in and press `Enter`.

If you accidentally confirmed that you want to save the file but you actually don't, you can always cancel by pressing `Ctrl` + `C` when you're prompted for a filename.

Cutting and pasting

To cut a single line, you use `Ctrl` + `K` (hold down `Ctrl` and then press `K`). The line disappears. To paste it, you simply move the cursor to where you want to paste it and punch `Ctrl` + `U`. The line reappears. To move multiple lines, simply cut them with several `Ctrl` + `K` in a row, then paste them with a single `Ctrl` + `U`. The whole paragraph appears wherever you want it.

If you need a little more fine-grained control, then you have to mark the text. Move the cursor to the beginning of the text you want to cut. Hit `Ctrl` + `6` (or `Alt` + `A`). Now move your cursor to the end of the text you want to cut: the marked text gets highlighted. If you need to cancel your text marking, simply hit `Ctrl` + `6` again. Press `Ctrl` + `K` to cut the marked text. Use `Ctrl` + `U` to paste it.

Searching for text

Searching for a string is easy as long as you think *"WhereIs"* instead of *"Search"*. Simply hit `Ctrl` + `W`, type in your search string, and press `Enter`. To search for the same string again, hit `Alt` + `W`.

Note
In nano's help texts the `Ctrl` is represented by a caret (^), so `Ctrl` + `W` is shown as `^W`, and so on. The `Alt` key is represented by an `M` (from "Meta"), so `Alt` + `W` is shown as `M-W`.

Configuraring the Nano editor

You can configure the behaviour of nano by editing the file `/etc/nanorc`. Changes in this file will affect the entire system. To configure nano for yourself only, copy `/etc/nanorc` to `~/.nanorc` and edit your local config file.

Nano Cheatsheet

- `Ctrl` + `X` Exit the editor. If you've edited text without saving, you'll be prompted as to whether you really want to exit.
- `Ctrl` + `O` Write (output) the current contents of the text buffer to a file. A filename prompt will appear; press Ctrl+T to open the file navigator shown above.
- `Ctrl` + `R` Read a text file into the current editing session. At the filename prompt, hit Ctrl+T for the file navigator.
- `Ctrl` + `K` Cut a line into the clipboard. You can press this repeatedly to copy multiple lines, which are then stored as one chunk.
- `Ctrl` + `J` Justify (fill out) a paragraph of text. By default, this reflows text to match the width of the editing window.
- `Ctrl` + `U` Uncut text, or rather, paste it from the clipboard. Note that after a Justify operation, this turns into unjustify.
- `Ctrl` + `T` Check spelling.
- `Ctrl` + `W` Find a word or phrase. At the prompt, use the cursor keys to go through previous search terms, or hit Ctrl+R to move into replace mode. Alternatively you can hit Ctrl+T to go to a specific line.
- `Ctrl` + `C` Show current line number and file information.
- `Ctrl` + `G` Get help; this provides information on navigating through files and common keyboard commands.

- grep(1) man page
- Template:Harvtxt
- Template:Harvtxt
- Template:Harvtxt
- Template:Harvtxt
- Template:HarvtxtTemplate:Page needed
- ISO/IEC 9945-2:1993 *Information technology – Portable Operating System Interface (POSIX) – Part 2: Shell and Utilities*, successively revised as ISO/IEC 9945-2:2002 *Information technology – Portable Operating System Interface (POSIX) – Part 2: System Interfaces*, ISO/IEC 9945-2:2003, and currently ISO/IEC/IEEE 9945:2009 *Information technology – Portable Operating System Interface (POSIX®) Base Specifications, Issue 7*
- The Single Unix Specification (Version 2)
- Template:Cite web
- Template:Cite web